Clojure

The Essential Reference



Renzo Borgatti

MANNING

Clojure

The Essential Reference



Renzo Borgatti

MANNING

MANNING PUBLICATIONS



BRIEF TABLE OF CONTENTS

Part 1: Introduction

1 Know your tools

Part 2: Fundamentals

2 Creating and manipulating functions

3 Basic constructs

4 Creating and inspecting macros

5 Operations on numbers

6 Comparison and equality

7 Reducers and transducers

Part 3: Data structures

8 Collections

9 Sequences

10 Sequential processing

11 Maps

12 Vectors

13 Sets

Part 4: Solving specific problems

14 Concurrency

15 Types, classes, hierarchies and polymorphism

16 Vars and namespaces

17 Evaluation

18 Formatting and printing

19 Strings and regular expressions

20 Mutation and side effects

21 Java interoperation

22 The toolbox

23 Dynamic variables in the standard library

Appendixes

A Alphabetic Index

B Glossary and recurring terminology

C Bibliography

1KNOW YOUR TOOLS

This chapter covers

- The standard library as a fundamental piece in your toolbox
- Transforming "dry specifications" into a fun and educational experience
- Clojure standard library high level structure
- Some examples of the library in action

Software development is often compared to a craft, despite the fact that it’s predominantly an intellectual activity. While software development is abstract in nature there are many craft-oriented aspects to it:

- The keyboard requires time and dedication to operate correctly. There are endless discussions on the best keyboard layout for programmers, for example to speed up typing [1].

- The development environment is a key aspect of programmers productivity and another source of debate (almost reaching a religious connotation). Mastering a development environment often translates into learning useful key combinations and ways to customize the most common operations.
- Libraries, tools and idioms surrounding the language. Almost everything above the pure syntax rules.
- Proficiency in several programming languages is definitely a plus in the job marketplace and the way to achieve it is by practicing them on a regular basis including getting familiar with APIs and libraries the language offers.
- Many other aspects require specific skills depending on the area of application: teaching, presenting or leadership.

The focus on mastering programming skills is so important that it became one of the key objectives of the Software Craftsmanship Movement [2]. Software Craftsmanship advocates learning through practice and promotes an apprenticeship process similar to other professions.

The standard library is definitely one of the most important tools to master a language. One aspect that characterizes the

standard library is the fact that it is already packaged with a language when you first experiment with it. Interestingly, it doesn't get the amount of attention you would expect for such an easy to reach tool. This book will show you how much wisdom and potential is hidden inside the Clojure standard library.

1.1 WHY SHOULD I CARE ABOUT THE STANDARD LIBRARY?

The expressiveness of a language is often described as the speed at which ideas can be translated into working software. Part of the expressiveness comes from the language itself in terms of syntax, but another fundamental part comes from the standard library which is usually provided out of the box. A good standard library liberates the programmer from the most mundane tasks like connecting to data sources, parsing XML, dealing with numbers and a lot more. When the standard library does a good job, developers are free to concentrate on core business aspects of an application, boosting productivity and return of investment.

Consider also that a deep knowledge of the standard library is often what distinguish an average developer from the expert. The expert can solve problems more elegantly and faster than

the beginner because, apart from having solved the same problem before, they can compose a complex solution by pulling small pieces together from the standard library

Finally, the standard library contains solutions to common programming problems that have been battle-tested over generations of previous applications. It is certainly the case for Clojure. The robustness and reliability that comes with that kind of stress is difficult to achieve otherwise. There will be possibly just a handful of cases where something in the standard library won't fit your needs and will need to be re-implemented.

[1] Dvorak users often claim huge benefits compared to QWERTY users. Here's one comparison, including other kind of layouts: lifehacker.com/should-i-use-an-alternative-keyboard-layout-like-dvorak-1447772004
[2] manifesto.softwarecraftsmanship.org

1.2 AREN'T SPECIFICATIONS JUST PLAIN BORING?

The short answer is: they don't need to be. Also consider that a language specification doesn't always mandate the content of the additional language utilities and tools. The standard library is then free to be described and documented in a less formal

way. Clojure doesn't have a formal language specification, let alone a standard library specification. Clojure's informal (and pragmatic) approach is also its main weakness in terms of documentation: the language and the standard library are described in a very essential style which is often considered beginner-unfriendly [3]. A lot of effort has been put lately into improving Clojure documentation although, at the time of this writing, the standard library is still lacking a comprehensive and centralized reference.

This book puts a great deal of effort illustrating functions in a readable and pleasant way, using a lot of real-life examples and visual structure to attract attention to the essential parts. Despite not being designed as a book to read cover-to-cover, each function is a pleasant and interesting reading on its own that also offers insight into functional (and general) programming. The following is a simplified version of the function fnil very similar to how it appears in the book. It has been annotated to show what is the purpose of each section:

Figure 1.1. The template for a function as it is illustrated in the book, with ovals explaining what each section is about.

2.2.1. fnil

`function` `since 1.2`

Type (function, macro, var, or special-form) and in which Clojure version it was introduced

Tags: a list of the most important aspects of programming the function is used for

-> Function generation, parameter handling, default argument values

```
(fnil
  ([f default1])
  ([f default1 default2])
  ([f default1 default2 default3]))
```

Signature: shows the different group of arguments the function can be invoked with (colloquially referred as "aritities")

Abstract: a short narration of what the function does

`fnil` generates a new function starting from an input function f. The main use case for `fnil` is to decorate another function so that it can default to optional values in case of a `nil` input. [...]

Contract

Contract: formal specification of the arguments present in the signature with specific emphasis around types

- "f" is a function of any number of arguments returning any type.

Examples

`fnil` operates well with any Java interop invocation [...]

```
(def request-params
 {:name "Jack"
  :selection nil})

(defn as-nums [selection]
  (let [nums (clojure.string/split selection #",")]
```

Examples: one or more examples showing how the function is typically used. When possible the example replicates a specific real-life scenario or the most common usage.

The `:selection` key is normally a comma separated collection of numbers but it comes as a `nil` if the user doesn't fill the related input field. `as-nums` will throw an exception in case of a `nil` selection,

Call-out: optional section is dedicated to any interesting fact about the function beyond its formal specification. It can be around some Clojure feature, historical notes, curiosities and so on.

Beyond the 3rd argument: an extended `fnil`

`fnil` can handle up to 3 default positional values for a given input function. It's quite easy to extend `fnil` to handle any number of default options:

```
(defn fnil+ [f & opts]
  (fn [& args]
    (apply f (map #(if (nil? %1) %2 %1) args (concat opts (repeat nil))))))
```

The new function `fnil+` above [...]

See Also: List of related functions with a similar classification or effect. More examples can be present in this section to illustrate similarities and best practices.

See Also

- some-> can be used to to obtain a similar effect to a nil default.
- some->> is similar to `some->` but the nil handling happens for the last argument.

Performance Considerations and Implementation Details

→ `O(1)` ***function generation***
→ `O(1)` ***generated function***

Performance considerations: section dedicated to the performance profile and implementation details.The section opens with an approximated Big-O notation

`fnil` usage should not have relevant performance implications as it is constrained by the number of arguments it can handle.

[3] See the latest "State of Clojure" survey 2020: clojure.org/news/2020/02/20/state-of-clojure-2020. Documentation is still ranking high in the list of major problems with the language

1.3 THE DIFFERENT VERSIONS OF CLOJURE

There are many actively maintained implementations of Clojure targeting different "runtimes". A "runtime" is a platform-dependent executable that allows Clojure application to run on different operating systems and hardware architectures. The de-facto reference implementation runs on the Java Virtual Machine but there are notably two others actively maintained Clojure versions: ClojureCLR (github.com/clojure/clojure-clr) and ClojureScript (clojurescript.org).

Rich Hickey, the inventor of Clojure, maintained for quite a long time both the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) versions of Clojure [4]. Rich later abandoned the CLR version because of the huge amount of work required to maintain both of them. Luckily for the Clojure community, David Miller decided in 2009 to work on a brand

new port of Clojure JVM into the CLR into what is now called "ClojureCLR" [5].

Similarly to the ClojureCLR, a version of Clojure able to translate into JavaScript (and then running in a browser) was discussed very early in Clojure project history resulting in a proof of concept [6]. Rich Hickey finally announced ClojureScript at the New York City user group in 2011 [7].

ClojureCLR and ClojureScript are very closely related to the Clojure JVM version but they were never meant to be drop-in replacements. It means that while many of the functions available in Clojure (JVM) are indeed available on the other platforms, there might be substantial differences in the way they work. So a Clojure application written for the JVM version could (and very likely will) require changes to run on the ClojureCLR or ClojureScript.

Although this book is specifically dedicated to the JVM version of Clojure, many of the functions and macros described here work pretty much the same on other Clojure runtimes. But if you need specific information about how a function or macro works on other runtimes, please refer to the documentation provided on their respective websites. From now on, please

consider any topic in this book related to the JVM version of the Clojure standard library.

[4] The interested reader can see the extent of the effort by checking out the Clojure project from Github and using the following git command: `git rev-list --reverse --format="- %B %cd" -n 1 HEAD—src/cli/runtime`. The C# files were finally removed from the project sometimes in 2007 with commit `b6db84aea2db2ddebcef58918971258464cbf46f`

[5] David Miller speaks about the history of ClojureCLR on this episode of the "Defn" podcast: soundcloud.com/defn-771544745/48-david-miller-and-clojure-on-the-clr

[6] The ClojureScript effort can be traced back to IRC discussions in May 2008 web.archive.org/web/http://clojure-log.n01se.net/date/2008-05-29.html#15:26

[7] The original ClojureScript release announcement was captured on video and available at www.youtube.com/watch?v=tVooR-dF_Ag

1.4 THE CLOJURE STANDARD LIBRARY

The Clojure standard library is quite comprehensive and can be divided roughly into 3 parts:

1. **The content of the single namespace `clojure.core`** (what is commonly referred as "core"). Core contains the functions that have evolved to be the main public API for the language, including basic math operators, functions to create and manipulate other functions, conditionals. Core currently contains around 700 definitions between functions and macros. Functions in core are always available without any explicit reference from any namespace.
2. **Namespaces other than "core"** (still shipped as part of Clojure). These are usually prefixed with `clojure` followed by a descriptive name, like `clojure.test`, `clojure.zippers` or `clojure.string`. Functions in these namespaces are sometimes available just prefixing their namespace (like `clojure.string/upper-case`) but in other cases they need to be imported in the current namespace using require [8].
3. **The content of the Java SDK** which is easily available as part of Clojure Java interoperability features. This book shows many examples of use of the Java standard library from Clojure, but doesn't go in the details of describing the Java examples.

In this book we will refer to the Clojure standard library as the first two parts described above, basically everything that you get by just downloading the Clojure package and without

downloading other libraries. In general, items in the standard library are marked as public, although some functions are marked as "alpha" in the Clojure documentation string and subject to change. The book will warn the reader about functions that can be used but are not guaranteed to stay in the library.

The standard library content can be roughly categorized by looking at the major features Clojure introduces and by the most common programming tasks. There are, for example, big groups of functions dedicated to Software Transactional Memory [9], concurrency and persistent collections. Of course Clojure also adds all the necessary support for common tasks like IO, sequence processing, math operations, XML, strings and many others. Apparently missing from the Clojure standard library are solutions already provided by the Java SDK, for example cryptography, low-level networking, HTTP, 2D graphics and so on. For all practical purposes those features are not missing, but just usable as they are from Java without the need to re-write them in Clojure. Java interoperability is one of the big strength of Clojure, opening the possibility to easily use the Java SDK (Standard Development Kit) from a Clojure program.

This book will cover both `clojure.core` (vast majority of functions in the standard library) as well as the additional

namespaces described in the following diagram and broadly grouped by area of application.

Figure 1.2. All other non-core namespaces.

Core Support

clojure.template
clojure.string
clojure.pprint
clojure.set
clojure.stacktrace

REPL

clojure.main
clojure.repl
clojure.core.server

General Support

clojure.walk
clojure.zip
clojure.sh
clojure.xml
clojure.test
clojure.core.reducers

Java

clojure.java.javadoc
clojure.java.browse
clojure.inspector
clojure.reflect
clojure.java.io

Data Serialization

clojure.edn
clojure.instant
clojure.data

- **Core support** namespaces integrate core with additional functionalities on top of those already present. `clojure.string` is possibly the best example. Core already contains str but any other useful string functionalities have been moved out into the `clojure.string` namespace. `clojure.template` contains a few helpers for macro creation. `clojure.set` is about the "set" data structure. `clojure.pprint` contains formatters for almost all Clojure data types so they can print in a nice, human-readable form. Finally `clojure.stacktrace` contains functions to manipulate exceptions and their formatting.
- **REPL** namespaces contain functionalities dedicated to the REPL, the read-evaluation-print-loop Clojure offers. `clojure.main` includes handling of the main entry point into the Clojure executable and part of the REPL functionalities that have been split into `clojure.repl` in later time. The latest addition, `clojure.core.server` implements the server socket functionality.

- **General support** is about additional APIs beyond what core has to offer. The namespaces present here enrich Clojure with new functionalities. `clojure.walk` and `clojure.zip` for example are two ways to walk and manipulate tree-like data structure. `clojure.xml` offers XML parsing capabilities. `clojure.test` is the unit test framework included with Clojure. `clojure.sh` contains functions to "shell-out" commands to the operative system. `clojure.core.reducers` offers a model of parallel computation.

- **Java** are namespaces dedicated to Java interop beyond what core already has to offer. `clojure.java.browser` and `clojure.java.javadoc` offer the possibility to open a native browser to display generic web pages or javadoc documentation respectively. `clojure.reflect` wraps the Java reflection APIs offering an idiomatic Clojure layer on top of it. `clojure.java.io` offers a sane approach to `java.io`, removing all the idiosyncrasies that made Java IO so confusing, like knowing the correct combination of constructors to transform a Stream into a Reader and vice-versa. Finally the `clojure.inspector` offers a simple UI to navigate data structures.
- **Data Serialization** is about ways in which Clojure data can be encoded as string as an exchange format. `clojure.edn` is the main entry point into EDN [10] format serialization. `clojure.data` contains only one user-dedicated function "clojure.data/diff" to compute differences between data structures. `clojure.instant` defines encoding of time related types.

Despite the classification above giving a nice overview of what's available beyond core functions, the book is structured so that `clojure.core` functions and non-core functions are re-grouped when necessary to reflect their area of application. A couple of notable examples are:

- clojure.reflect/reflect appears in the "Java Interop" chapter along with proxy, gen-class or doto which are instead core functions.
- clojure.walk/stringify-keys appears along with other core hash-map functions.

The book makes the assumption that readers are relatively interested in knowing where exactly a function lives (if not just to require it at the top of the namespace to use it) but they are more interested in knowing that the function exists when they have a particular problem to solve.

Although the vast majority of items in the standard library are either functions or macros, the book also describes some dynamic variables. Dynamic variables are a special kind of reference type that can be re-bound on a thread-local basis (see the great description of dynamic variables from "Joy of Clojure" for a detailed explanation [11]). The reason for dynamic variables to be also described in this book is because they are

often the way other functions in the standard library are configured.

[8] this is due to the fact that while bootstrapping, Clojure already imports several namespaces that are automatically available for the end user. Very popular tools like nRepl (github.com/nrepl/nrepl) or CIDER (github.com/clojure-emacs/cider) also load libraries while bootstrapping, which are then available at the prompt. It is good practice to always require what is useful in a namespace explicitly
[9] For a good introduction to STM see Wikipedia: en.wikipedia.org/wiki/Software_transactional_memory
[10] The EDN format is described here: github.com/edn-format/edn
[11] The "Joy of Clojure" is available on the Manning website: www.manning.com/books/the-joy-of-clojure-second-edition

1.5 MAKING YOUR DEVELOPMENT LIFE EASIER

The standard library is not just there to solve the usual recurring programming problems but to offer elegant solutions to new development challenges. "Elegant" in this context translates to composable solutions that are easy to read and maintain. Let’s look at the following example.

Suppose that you're given the task to create a report to display information on screen in a human readable form. Information is coming from an external system and a library is already taken care of that communication. All you know is that the input arrives structured as the following XML (here saved as a local `balance` var definition):

```
(def balance
  "<balance>
    <accountId>3764882</accountId>
    <lastAccess>20120121</lastAccess>
    <currentBalance>80.12389</currentBalance>
  </balance>")
```

The balance needs to be displayed in an user-friendly way:

1. Removing any unwanted symbols other than letters.
2. Separating words like "lastAccess" or "currentBalance" (using the uppercase letter as delimiter).
3. Formatting the balance as a currency with 2 decimal digits.

You might be tempted to solve the problem like this:

```
(require '[clojure.java.io :as io])
(require '[clojure.xml :as xml])
```

```
(defn- to-double [k m]
  (update-in m [k] #(Double/valueOf %)))

(defn parse [xml] ; ❶
  (let [xml-in (java.io.ByteArrayInputStream. (.
        results (to-double
                  :currentBalance
                  (apply merge
                         (map #(hash-map (:tag %
                              (:content (xml/par
    (.close xml-in)
    results))

(defn clean-key [k] ; ❷
  (let [kstr (str k)]
    (if (= \: (first kstr))
      (apply str (rest kstr))
      kstr)))

(defn- up-first [[head & others]]
  (apply str (conj others (.toUpperCase (str hea

(defn separate-words [k] ; ❸
  (let [letters (map str k)]
    (up-first (reduce #(str %1 (if (= %2 (.toLow

(defn format-decimals [v] ; ❹
  (if (float? v)
```

```
      (let [[_ nat dec] (re-find #"(\d+)\.(\d+)" (
        (cond
          (= (count dec) 1) (str v "0")
          (> (count dec) 2) (apply str nat "." (ta
          :default (str v)))
      v))

  (defn print-balance [xml] ; ❺
    (let [balance (parse xml)]
      (letfn [(transform [acc item]
                (assoc acc
                       (separate-words (clean-key
                       (format-decimals (item bala
        (reduce transform {} (keys balance)))))

  (print-balance balance)
  ;; {"Account Id" 3764882, "Last Access" "2012012
```

❶ `parse` takes the XML input string and parses it into a map containing just the necessary keys. `parse` also converts `:currentBalance` into a double.

❷ `clean-key` solves the problem of removing the ":" at the beginning of each attribute name. It checks the beginning of the attribute before removing potentially unwanted characters.

❸ `separate-words` takes care of searching upper-case letters and pre-pending a space. reduce is used here to store the accumulation of changes so far while we read the original string as the input. `up-first` was extracted as an handy support to upper-case the first letter.

❹ `format-decimals` handles floating point numbers format. It searches digits with re-find and then either append (padding zeros) or truncate the decimal digits.

❺Finally `print-balance` puts all the transformations together. Again reduce is used to create a new map with the transformations while we read the original one. The reducing function was big enough to suggest an anonymous function in a letfn form. The core of the function is assoc the new formatted attribute with the formatted value in the new map to display.

While being relatively easy to read (the 3 formatting rules are somehow separated into functions) the example shows minimal

use of what the standard library has to offer. It contains map, reduce, apply and a few others including XML parsing, which are of course important functions (and usually what beginners learn first). But there are definitely other functions in the standard library that would make the same code more concise and readable.

Let's have a second look at the requirements to see if we can do a better job. The source of complexity in the code above can be tracked down to the following:

- String processing: strings need to be analyzed and decomposed. The `clojure.string` namespace comes to mind.
- Map related computations: both keys and values need specific processing. reduce is used here because we want to gradually mutate both the key and the value at the same time. But zipmap sounds a viable alternative worth exploring.
- Formatting rules of the final output: things like string padding of numerals or rounding of decimals. There is an interesting "clojure.pprint/cl-format" function that might come handy.

- Other details like nested forms and IO side effects. In the first case threading macros can be used to improve readability. Finally, macros like with-open removes the need for developers to remember to initialize the correct Java IO type and close it at the end.

By reasoning on the aspect of the problem we need to solve, we listed a few functions or macros that might be helpful. The next step is to verify our assumptions and rewrite the example:

```
(require '[clojure.java.io :as io])
(require '[clojure.xml :as xml])
(require '[clojure.string :refer [split capitali
(require '[clojure.pprint :refer [cl-format]])

(defn- to-double [k m]
  (update-in m [k] #(Double/valueOf %)))

(defn parse [xml] ; ❶
  (with-open [xml-in (io/input-stream (.getBytes
    (->> (xml/parse xml-in)
         :content
         (map #(hash-map (:tag %) (first (:conte
         (into {})
         (to-double :currentBalance))))
```

```
(defn separate-words [s]
  (->> (split s #"(?=[A-Z])") ; ❷
       (map capitalize)       ; ❸
       (join " ")))

(defn format-decimals [v]
  (if (float? v)
    (cl-format nil "~$" v) ; ❹
    v))

(defn print-balance [xml]
  (let [balance (parse xml)
        ks (map (comp separate-words name) (keys
        vs (map format-decimals (vals balance))]
    (zipmap ks vs))) ; ❺

(print-balance balance)
;; {"Account Id" 3764882, "Last Access" "2012012
```

❶ `parse` now avoids the let block, including removing the need to close the input stream. This is achieved by with-open. The -> threading macro has been used to give a more linear flow to the previously nested XML processing.

❷ `separate-words` now uses a few functions from `clojure.string`. `split` takes a regular expression that we can use to divide the string by upper case letters. Compare this version with the previous one using reduce: this is easier to read and understand.

❸ We now `capitalize` each word and finally `join` everything together in new string.

❹ `format-decimals` delegates almost completely to "clojure.pprint/cl-format" which does all the job of formatting decimals.

❺ zipmap brings in another dramatic change in the way we process the map. We can isolate changes to the keys (composing words separation and removing the unwanted ":") and changes to the values into two separated map operations. zipmap conveniently combines them back into a new map without the need of reduce or assoc.

The second example shows an important fact about "knowing your tools" (in this case the Clojure standard library): the use of a different set of functions not only cuts the number of lines from 45 to 30, but also opens up the design to completely

different decisions. Apart for the case where we delegated entire sub-tasks to other functions (like `cl-format` to format decimals or name to clean a key), the main algorithmic logic took a different approach that does not use reduce or assoc. A solution that is shorter and more expressive is clearly easier to evolve and maintain.

1.6 THE PROBLEM OF FRAGMENTED INFORMATION

Since 2010, Chas Emerick has been asking the Clojure community a few questions in the form of an yearly survey to collect feedback about Clojure adoption in the industry. Cognitect, the company that is actively sponsoring the development of Clojure, is continuing the tradition with the last available results for 2019 published on their website [12]. Since the beginning of the survey, one of the major concern that people reported has been about the quantity and quality of the Clojure documentation.

The Clojure community (mainly under the guidance of Alex Miller and others from the core team) has made tremendous progress to enhance the Clojure guides and tutorials, culminating with the open source release of the Clojure documentation website, enabling anyone with an easy way to

contribute [13]. The documentation that comes with Clojure itself instead, is terse and to the point. This is good to quickly remember how something is supposed to work, but not necessarily exhaustive. If you type `(doc interleave)` at the REPL for example, you are welcomed with:

```
user=> (doc interleave)
-------------------------
clojure.core/interleave
([] [c1] [c1 c2] [c1 c2 & colls])
  Returns a lazy seq of the first item in each c
nil
```

"Returns a lazy seq of the first item in each coll, then the second etc." is precise and essential. It assumes you understand what a "lazy seq" is and leaves out details like what happens with unevenly sized collections. You could further explore interleave by typing examples at the REPL or, missing ideas about what to type, search for snippets on the Internet. Some of the background concepts are documented on the Clojure website under the "reference" section (clojure.org/reference). The reference documentation has been there since the beginning and is following the same essential style of `doc` at the REPL. If you are a seasoned programmer with some functional

experience you'll be definitely comfortable with that, but that's not always the case for Clojure starters. The recently introduced "Clojure Documentation" website at clojure-doc.org is the beginning of that community contributed effort more directed at "getting started".

Although clojure-doc.org is now here, multiple efforts started over the years to fill the gaps left by the original documentation. The following is a summary of the other resources available at the time of this writing:

- clojuredocs.org is a community powered documentation engine. It basically offers examples and notes on top of the standard library documentation including cross-links. The quality of the documentation for a function varies from nothing to many examples and comments.
- groups.google.com/forum/#!forum/clojure is the main Clojure mailing list. Absolutely great threads are recorded in there, including topics discussing the overall Clojure vision and design by Rich Hickey himself and the rest of the core team.

- web.archive.org/web/http://clojure-log.n01se.net the IRC Clojure channel logs. Same as the mailing list, with some important discussions shaping the design of the future Clojure releases.
- Books. The number of Clojure books written so far is impressive. People really like to write books on Clojure and this book is no exception!
- stackoverflow.com/search?q=clojure Clojure related questions is an amazing source of great information. Almost any conceivable problem, philosophical or practical, has been answered there.
- Blogs: too many good blogs to enumerate all here. planet.clojure.in/ does a great job continuously aggregating blog posts, articles and other Clojure sources that appear constantly on the Web.

As you can see documentation exists in many forms and is overall very valuable, but it is fragmented: jumping between all the different sources is time consuming, including the fact that searching the right place it's not always obvious. One of the main goals of this book is to do that work on your behalf: bringing together all the valuable sources of information in a single accessible place.

[12] The first survey for 2010 is available here: cemerick.com/2010/06/07/results-from-the-state-of-clojure-summer-2010-survey/. The last is available on the Cognitect blog: clojure.org/news/2019/02/04/state-of-clojure-2019

[13] Here's the request for help related to the open source release of the Clojure.org website: clojure.org/news/2016/01/14/clojure-org-live

1.7 THE WELL KEPT SECRET OF THE CLOJURE NINJA

Learning about the functions in the standard library is usually a process that starts at the very beginning. It happens when you first approach some tutorial or book, for example when the author shows a beautiful one-liner that solves an apparently big problem.

Usually developers don't pay explicit attention to the functions in the standard library, assuming knowledge will somewhat increase while studying the features of the language. This approach can work up to a certain point but it is unlikely to scale. If you are serious about learning the language consider to allocate explicit time to understand the different nuances of similar functions or the content of some obscure namespace. The proof that this is time well spent can be found reading

other's people experience: the web contains many articles describing the process of learning Clojure or documenting discoveries (possibly the best example is Jay Field's blog [14]).

The following is a trick that works wonders to become a true Clojure Master. Along with learning tools like tutorials, books or exercises like the Clojure Koans [15], consider adding the following:

- Select a function from this book's table of content every day. It could be lunch or commuting time for example. Another option is to have this book on your desk and randomly open up a page every once in a while.
- Study the details of the function sitting in front of you. Look at the official docs first, try out examples at the REPL, search the web or www.github.com for Clojure projects using it.
- Try to find where the function breaks or other special corner cases. Pass `nil` or unexpected types as arguments and see what happens.
- Repeat the next day or regularly.

Don't forget to open up the sources for the function, especially if belonging to the "core" Clojure namespace. By looking at the

Clojure sources, you have the unique opportunity to learn from the work of Rich Hickey and the core team. You'll be surprised to see how much design and thinking goes behind a function in the standard library. You could even find the history of a function intriguing, especially if it goes back to the origins of Lisp: "apply" for example, links directly to the MIT AI labs where Lisp was born in 1958! [16] Only by expanding your knowledge about the content of the standard library you'll be able to fully appreciate the power of Clojure.

[14] blog.jayfields.com
[15] github.com/functional-koans/clojure-koans
[16] eval and apply are at the core of the meta-circular interpreter of Lisp fame. The whole Lisp history is another fascinating reading on its own. See any paper from Herbert Stoyan on that matter

1.8 THE PERFECT COMPANION BOOK

We think this book is a perfect fit for the intermediate or seasoned Clojure programmer who wants to understand how a function (and ultimately Clojure) works.

If you are just starting with Clojure, this book covers some theory and background to get you comfortable understanding the main goal of a function, but it shouldn't be your only text. If

you already own or plan to buy a more introductory text, we think this book is a perfect complementary choice: feel free to dig deeper in this book to find more exhaustive information about a function or a macro.

If you are an intermediate or experienced Clojure programmer, we think you'll find the resources in this book valuable. We put a lot of effort in collecting real-world examples and analyzing the Clojure internals. The references, pointers and bibliography are also adding to the overall experience by guiding curious readers to additional learning material.

It's the hope of the authors that this book is going to be a de-facto reference to the standard library and it's certainly our desire to provide the reader with one of the most comprehensive source of information about Clojure.

1.9 SUMMARY

- The standard library is the collection of functions and macros that comes out of the box by installing Clojure.
- The Clojure Standard Library is rich and robust, allowing developers to concentrate on core business aspects of an application.

- Information about the Standard Library tends to be fragmented, but this book collects everything in a single accessible place.
- Deep knowledge of the content of the Standard Library improves code expressiveness exponentially.
- While the standard library is considered by many a passive resource to access in case of a specific need, this book suggests the more interesting approach to learn it with a more systematic approach.
- A lot of effort has been put in this book to make what follows in Part II an interesting and enriching experience, not just a dry list of specifications.

PART 2. FUNDAMENTALS

This part of the book describes some of the most fundamental functions of the Clojure standard library. This classification (as all the others presented in this book) is just a guiding tool and certainly there are no "sharp edges" between the chapters in the book. The part is divided into chapters for area of application and each chapter is further divided into sub-groups of related functions. The structure of the chapters should gradually orientate the use toward the best approach to solve a problem and give an idea of other related functions in the standard library that might help. Other classifications are given at the end of the book to further help searching the best solution.

2CREATING AND MANIPULATING FUNCTIONS

Without too much surprise, a functional language is specifically good at providing developers with tools and syntax support for creating and composing functions. This chapter groups together the functions in the Clojure standard library that are dedicated to manipulate or generate other functions. The chapter splits them into 4 broad categories:

1. Function Definition. A function is the fundamental unit of composition in Clojure. This section contains the main macros dedicated to declaring new functions.
2. Higher order functions. This section describes functions and macros whose main goal is to produce new functions guided by an user defined computation or other existing definitions.
3. Threading macros. This important group of macros gives Clojure a visually appealing syntax to describe processing pipelines.
4. Function execution. Finally, another group of functions dedicated to manage the execution of other functions.

Other functions and macros exist that can be categorized using the same criteria, but in this initial "fundamental" chapter, we concentrate on the most important ones while others are described in other parts of the book.

2.1 FUNCTION DEFINITION

Clojure is specifically designed around the concept of a "function". A function is, not surprisingly, the main mechanism of composition in a functional language: once a group of computational steps is captured as a function, it can be invoked or composed with others to promote higher levels of abstraction. The most important entry in the standard library to define a function is defn. Additionally, Clojure offers other ways to help modularize applications: definline, for example, improves performance in case of Java interop, while fn is embeddable in other functions. There is an overlap with macros described later, but considering they introduce a small language on their own, they have been dedicated a specific chapter.

2.1.1 defn and defn-

`macro` `since 1.0`

Listing 2.1. → *Function definition*

```
(defn [name & fdecl])
```

`defn` (and its private version `defn-`) is one of the fundamental constructs and main entry point for function creation in Clojure. It supports a rich set of features like destructuring, multiple arities, type hinting, `:pre` and `:post` conditions and more (via fn, which is closely related). The calling contract is like a small language in itself and `defn` is dedicated to parse this little grammar. The most used form of `defn` is probably the simple single-arity case:

```
(defn hello [person] ; ❶
  (str "hello " person))
```

❶A simple function definition. The function `hello` takes a string and return a string.

`defn` works in conjunction with def (for interning its name in the current namespace) and fn (for pre-post conditions and destructuring). Since `defn` is a macro, we can call macroexpand on it to understand how it works:

```
(macroexpand  ; ❶
  '(defn hello [person]
     (str "hello " person)))

;; (def hello
;;   (clojure.core/fn ; ❷
;;     ([person] (str "hello " person))))

(hello "people") ; ❸

;; "hello people"
```

❶We can call macroexpand on the previous function definition to see how Clojure assembles the creation of an anonymous function with a var definition in the current namespace.

❷The lambda just created via fn is assigned to a new Var object "hello".

❸The "hello" symbol is available for execution in the current namespace using surrounding parenthesis.

Contract

The contract for `defn` is quite elaborated and in the spirit of maintaining this book practical, we are not going to use a rigorous grammar to describe this or other functions in the book. We are going to use an informal grammar instead, which

is hopefully more expressive than the corresponding function documentation, but does not aim to cover the full spectrum of all possible cases (nor being 100% consistent). The reader should use the grammar to get a general sense of what the function does, but then refer to the examples further below to cover the remaining cases. Here's a glossary of the syntax used in this book (the same glossary has also been added to the appendix):

- "<term>": terms in angle brackets are terminal and they don't result in any further expansion.
- "[]": terms in square brackets are actual vectors in the function definition.
- "()": terms in round brackets are actual lists in the function definition.
- "OR": expresses the option of choosing between one the possible choices.
- ":=>": shows a term that can be further expanded into other terms.
- "..": means "many other repetitions of" the expressions surrounding it.
- "?": after a term means that the term is optional.

With this small grammar in mind, here's how we could express how `defn` could be used:

```
(defn ^<metamap>? <name> fdecl)

fdecl :=> <docstring>? <metamap>? arities <meta

arities :=> ^<metamap>? [arity] body
            OR
            (^<metamap>? [arity1] body)
            (^<metamap>? [arity2] body)
            ..
            (^<metamap>? [arityN] body)

arity :=> <ret-typehint>? [<arg1-typehint>? <a
                           ..
                           <argN-typehint>? <arg

body :=> <metamap> <forms>
```

- "<metamap>" is an optional map of key-value pairs (for example `^{:a true}`). "<metamap>" also accepts a concise syntax using just the keys ^:tagname1 ^:tagname2 where the value is implicitly `true`. Please note that the alternate syntax is only available in some of the location where "<metamap>" is allowed. Tags are stored along with the var created by the function definition. Also depending on position, `<metamap>` might require the `^` prefix to appear just before the opening curly brace.
- "<name>" is mandatory and must be a valid symbol [17].
- "<docstring>" is an optional string that describes the function. The documentation string is also stored in the var object resulting from the function definition. You can see the doc string using the doc function.
- `([arity1]) ([arity2]) .. ([arityN])` are arguments vectors of different lengths. In case of a single `[arity]` the wrapping parenthesis are optional.
- "<ret-typehint>" is an optional type hint that applies to the return value for the arity. "<ret-typehint>" can also appear inside "<metamap>" for that arity as `{:tag <typehint>}` with equivalent behavior.

- "<arg-typehint>" is an optional type hint for an argument in the argument vector. This as well can be expressed as "<metamap>".
- "body" contains the expressions evaluated when calling the function. It is implicitly wrapped in a do block. It is assumed to be `nil` when there is no body. An additional "<metamap>" (without caret prefix "^") is allowed before other expressions in the body. When both the argument vector and the body contain "<metamap>", the one in the body takes precedence in case of clashing keys. Note that pre- and post-condition (see below) are allowed only in "<metamap>" after the argument vector.

`defn` returns a `clojure.lang.Var` referencing the function object that was just created. The function name becomes available in the current namespace without any additional prefixing. It's worth noting that there are many places where metadata maps can appear.

Examples

`defn` usage is of course widespread. The following examples illustrate some of the most important aspects.

Metadata Placement

We can see from the contract that `defn` accepts metadata definitions, with slightly different syntax and meaning, in multiple positions. Some metadata are available as part of the resulting var definition, others inform the compiler during expression parsing, others are used at runtime during evaluation of the function. We can see all of them at once in the following (admittedly contrived) example:

```
(defn ^{:t1 1} foo  ; ❶
  "docstring"
  {:t2 2} (^{:t3 3} [a b] {:t4 4} (+ a b)) ; ❷
  {:t5 5}) ; ❸

(meta #'foo) ; ❹

;; {:ns #object[clojure.lang.Namespace 0x33f17f8
;;  :name foo,
;;  :t2 2,
;;  :file
;;  "/private/var/form-init5372276344059831979.c
;;  :column 1,
;;  :line 1,
;;  :arglists ([a b]),
;;  :doc "docstring",
;;  :t1 1,
```

```
  ;;  :t5 5}

  (meta (first (:arglists (meta #'foo)))) ; ❺
  ;; {:t3 3, :t4 4}
```

❶The first place for metadata is right after the `defn` declaration. In this position, it requires the caret `^` prefix.

❷Metadata maps appear 3 times in this line: before the first arity declaration, inside the first arity declaration and before the body of the function.

❸The final metadata map is available after all definitions.

❹We can see that `:t1, :t2, :t5` appear in the var metadata. Other context dependent information (like the namespace object or column/line information might differ when printed from another REPL).

❺The 2 metadata maps inside the arity definition have been merged into a single metadata map attached to the ":arglists" value.

It's worth noting that metadata at the end of the function are available only if when arities (the argument vector followed by the body) are wrapped by parenthesis. It's important to remember this aspect when the function has a single arity, as this is commonly written without the pair of surrounding

parenthesis. For instance, the example above works because `([a b] (+ a b))` is wrapped in parenthesis.

Along with reserved keys like `:doc`, `:tag`, `:pre` or `:post`, we can use metadata to annotate functions with any sort of contextual information. Annotations can be later analyzed and acted upon by libraries or tools. Metadata are used extensively in `core.test` namespace for example, to mark a function as "test" and run it accordingly. Key-value pairs in metadata can be of any type, such as strings, numbers or other structured data. However, `boolean` values are so frequent that they are commonly referred as "tags". Tags are provided with and additional prefix syntax that uses the caret symbol "^:".

The following example showcases a profiling feature that scans a namespace in search of functions tagged `^:bench`. When such a tag exists, the corresponding function is redefined to check execution time:

```
(ns profilable)

(defn ^:bench profile-me [ms] ; ❶
  (println "Crunching bits for" ms "ms")
  (Thread/sleep ms))

(defn dont-profile-me [ms]
```

```
  (println "not expecting profiling"))

(ns user)

(defn- wrap [f]
  (fn [& args]
    (time (apply f args))))

(defn- make-profilable [v]
  (alter-var-root v (constantly (wrap @v))))

(defn- tagged-by [tag nsname]
  (->> (ns-publics nsname)
       vals
       (filter #(get (meta %) tag))))

(defn prepare-bench [nsname] ; ❷
  (->> (tagged-by :bench nsname)
       (map make-profilable)
       dorun))

(profilable/profile-me 500) ; ❸
;; Crunching bits for 500 ms

(prepare-bench 'profilable)

(profilable/profile-me 500) ; ❹
;; Crunching bits for 500 ms
```

```
  ;; "Elapsed time: 502.422309 msecs"

  (profilable/dont-profile-me 0) ; ❺
  ;; not expecting profiling
```

❶The function `profile-me` in the "profilable" namespace has a `:bench` annotation that enters the metadata map
❷ `prepare-bench` does the search for all the function tagged with `:bench` in the given namespace and wraps them into a new function that is doing profiling.
❸Before `prepare-bench` is invoked `profile-me` prints the expected message.
❹But after invoking `prepare-bench`, `profile-me` also prints the elapsed time along with the message.
❺Other functions that were not tagged are unaffected.

Documenting

It is good practice to attach a short documentation string to a function to describe its purpose. Clojure provides a specific position for the documentation string, so the compiler can store this information appropriately. You can then use the clojure.repl/doc function to print useful information about the function including the documentation string:

```
(defn hello
  "A function to say hello"    ; ❶
  [person]
  (str "Hello " person))

(clojure.repl/doc hello)       ; ❷
;; -------------------------
;; user/hello
;; ([person])
;;   A function to say hello

;; ([person])
;;     A function to say hello ; ❸
;; nil

(:doc (meta #'hello))          ; ❹

;; "A function to say hello"
```

❶The documentation string appears just after the name of the function.
❷We use the function doc passing the var "hello" as a parameter.
❸The documentation string prints on screen along with the function signature.
❹Alternatively, we can extract the key `:doc` from the metadata.

Pre/post conditions

The next example shows how to use pre- and post-conditions. Conditions are functions with access to arguments and return value (post-conditions only). Clojure inspects the metadata map appearing after the argument vector in search for a `:pre` or `:post` keys. When `:pre` or `:post` keys exist, their value must be a collection of predicates. Predicates are invoked before or after function execution, respectively.

The following `save!` function, saves an item to some storage. Before pushing it to storage it checks a few facts about the input using pre-conditions. After saving to storage, it verifies that the item has the correct ":id" with a post-condition:

```
(require '[clojure.test :refer [are]])

(defn save! [item]
  {:pre [(are [x] x
           (map? item)
           (integer? (:mult item))
           (#{:double :triple} (:width item)))]
   :post [(clojure.test/is (= 10 (:id %)))]}
  (assoc item :id (* (:mult item) 2)))

(save! {:mult "4" :width :single})

;; FAIL in () (form-init828.clj:2)
;; expected: (integer? (:mult item))
;;   actual: (not (integer? "4"))
;;
;; FAIL in () (form-init828.clj:2)
;; expected: (#{:double :triple} (:width item))
;;   actual: nil
;;
;; AssertionError Assert failed:
;; (clojure.test/are [x] x (map? item) (integer?
;;  (#{:double :triple} (:width item)))  user/sa

(save! {:mult 4 :width :double})

;; FAIL in () (form-init828856234333710567 8.clj:
;; expected: (= 10 (:id %))
```

```
;;   actual: (not (= 10 8))
;;
;; AssertionError Assert failed:
;; (clojure.test/is (= 10 (:id %)))

(save! {:mult 5 :width :double})
;; {:mult 5, :width :double, :id 10}
```

❶"clojure.test/are" groups together multiple assertions. The assertions in this example are all expected to return logical `false` (including `nil`) if the assertion fails.

❷This predicate checks that item is of type map. Note that the argument "item" is available in pre- and post-conditions.

❸Similarly, this predicates is checking that the value for the key `:mult` is of type integer.

❹Set inclusion is used to verify the value of the `:width` key belongs to a small enumeration of allowed values.

❺Post-conditions work similarly with the addition of the placeholder `%` percent sign to access the returned value from the function. In this case we check the returned map contains an id equals to 10.

❻Failing assertions are nicely printed because of the clojure.test functions. `clojure.test` is part of the standard library.

❼In the next attemp, we fix pre-conditions, but we have a problem with post-conditions.

❽We can finally see a successful call to `save!`.

Note

The example demonstrate an useful trick consisting of wrapping pre- and post-conditions with `clojure.test/is` or `clojure.test/are` macros. The conditions still fail with `java.lang.AssertionError`, but the `clojure.test` wrappers show a much nicer message.

Type hinting

Type hints are the bridge between the dynamically-typed world of Clojure (where almost everything is treated as a generic `java.lang.Object`) and the statically-typed world of Java. Type hints in Clojure are optional in most cases, but they are required when speed is important (other common tips include disabling checked-math, using primitive unboxed types, using transients and many others techniques dependent on the specific case).

Type hints are usually required when Clojure functions call into Java methods. The Clojure compiler uses type information to avoid reflection in generated Java bytecode. Reflection is a very

useful (but slow) Java API to discover and invoke Java methods needed by the Clojure runtime.

To illustrate the point, the following example is about signing a request using a secret key. The Java standard library contains everything we need for this task, so no external libraries are required. The idea of signing a request is the following:

1. There is some unique string representation of the event we want to sign. We are going to use an URL in this example.
2. Two parties want to exchange the information but they want to be sure the information hasn't been tampered with in between. So they generate and store a secret-key that is used to sign the messages that no one else can access.
3. When "A" wants to send a message to "B", it signs the message by attaching a signature to it. "B" receives the message, applies the same process and verifies that the resulting signature is the same that was received.

Here's some code that implements the protocol:

```
(ns crypto
  (:import java.io.ByteArrayOutputStream
           javax.crypto.spec.SecretKeySpec
           javax.crypto.Mac
           java.util.Base64
```

```
            java.net.URLEncoder
            java.nio.charset.StandardCharsets))

(set! *warn-on-reflection* true) ; ❶

(defn get-bytes [s]
  (.getBytes s (StandardCharsets/UTF_8)))

(defn create-spec [secret]
  (SecretKeySpec. (get-bytes secret) "HmacSHA256

(defn init-mac [spec]
  (doto (Mac/getInstance "HmacSHA256")
    (.init spec)))

(defn compute-hmac [mac canonical]
  (.doFinal mac (get-bytes canonical)))

(defn encode [hmac]
  (URLEncoder/encode
    (.encodeToString (Base64/getEncoder) hmac)))

(defn sign [canonical secret] ; ❷
  (-> secret
      create-spec
      init-mac
      (compute-hmac canonical)
      encode))
```

```
(defn sign-request [url] ; ❸
  (let [signature (sign url "secret-password")]
    (format "%s?signature=%s" url signature)))

(sign-request "http://example.com/tx/1")
;; "http://example.com/tx/1?signature=EtUPpQpumB
```

❶Clojure provides the `*warn-on-reflection*` dynamic var to show where the compiler was unable to infer the types.

❷the `sign` function shows the steps required for the signature. We don't need to go deep into the details of the algorithms, but creating a sha256 hmac is pretty common procedure [18]

❸`sign-request` takes an url representing the transaction. The function returns the same URL appending the signature as one of the request parameters, ready to be sent across the wire.

When we look at the output during compilation, Clojure prints something like the following:

```
Reflection warning, crypto.clj:12:3 - call to me
Reflection warning, crypto.clj:21:3 - call to me
```

Source lines/column references might be different, but the message says that there are at least two places where the compiler is unable to infer the types and is using reflection. If in our example we assume peaks of 100k transactions per second, we might want to review how `sign-request` is performing. Advanced tools like Criterium are the best choice for benchmarking, but in this specific case we can clearly see what happens just by using `time`:

```
(time (dotimes [i 100000] ; ❶
  (sign-request (str "http://example.com/tx/" i)

;; "Elapsed time: 1054.507977 msecs"
```

❶ `time` is a small macro that wraps an expression with time measurement.

Note

The elapsed time displayed here (and in other parts of the book) is dependent on the hardware the benchmark is executed, so it could display a different number on other machines. What matters is the relative difference between instances of the benchmark, which should instead be the same independent from hardware.

Let's now add type hints to the function definitions highlighted by the compiler warnings:

```
(defn get-bytes [^String s]                    ; 
  (.getBytes s (StandardCharsets/UTF_8)))

(defn compute-hmac [^Mac mac canonical]        ; 
  (.doFinal mac (get-bytes canonical)))

(time (dotimes [i 100000]
  (sign-request (str "http://example.com/tx/" i)

;; "Elapsed time: 449.417098 msecs"            ; 
```

❶ The parameter "s" was tagged as `^String` so the following `.getBytes` is fully qualified.
❷ The parameter "mac" was tagged to be of type `^Mac`. The other warnings from the compiler also disappear, as `.doFinal` becomes fully qualified as well by inference.
❸ After adding the two type hints, we are able to cut the processing time down 50%.

As shown by the new measured time, we can achieve better performance when Reflection calls have been removed.

No more than 20!

Function definition in Clojure is limited to a maximum of 20 non-variadic parameters:

```
(defn a [a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a

;; CompilerException java.lang.RuntimeException: Can't specify mo
```

The above results in a compile time exception. The limit might seem arbitrary or restrictive but the rationale behind the choice is simple: Clojure puts a great deal of attention to speed and there are compiler optimizations that greatly benefit from having a specific Java method for each parameters number. There are several places in the Clojure codebase where this is

visible [19] and of course it is not easy to read, maintain or evolve.

Apart from the compiler implementation details any function with more than three or four parameters should look suspicious. Too many parameters should raise the question if there is a missing abstraction that groups them together.

See also

- fn is used under the hood by `defn` to generate the body of the function and implement destructuring. Differently from `defn`, `fn` does not create a var object or alter the current namespace as a side effect. Thus `fn` is a better choice for local use of functions without the need for an external name. `fn` is often used with sequential operations such as reduce to create an anonymous function of two arguments.
- definline creates a `defn` definition but also include an inlined version of the function body to improve Java interoperability. Consider using definline for performance sensitive functions if the function body does not do much more than wrapping a Java method call.

- letfn is syntactic sugar for an anonymous function definition associated to a let binding. Prefer `letfn` to create one or more named local functions.

Performance considerations and implementation details

`defn` is a macro with an impact primarily on Clojure compilation time. Common usage of `defn` should not generate concerns during program runtime. The definition of `defn` happens quite early during bootstrap of the standard library, when most of the common Clojure facilities are not yet defined. This aspect, along with the complexity related to inlining and type hints makes `defn` sources not easy to follow.

2.1.2 fn

`macro` `since 1.0`

Listing 2.2. → *Anonymous function definition, lambda*

```
(fn [params] exprs)
```

`fn` creates a new function and supports important features like destructuring, type hints, pre- and post-conditions (illustrated in defn) and multiple signatures based on the number of parameters (or "arities" as they are commonly called

in Clojure documentation). `fn` functions are first class objects in Clojure: you can pass them as arguments or bind them locally.

Function objects (also known as lambdas) are so common in functional programming that Clojure offers a special reader syntax for them (the reader macro `#()`). The following example shows the same function created with `fn` and the shorter reader syntax:

```
((fn [x] (* (Math/random) x))   ; ❶
  (System/currentTimeMillis))   ; ❷

;; 1.3144654837186 98E12

(#(* (Math/random) %)           ; ❸
  (System/currentTimeMillis))

;; 1.2215726280027874E12
```

❶ `fn` is used to declare an anonymous function of one argument.

❷ The created function is readily available to invoke and requires one argument. In this case we pass the current time in milliseconds which is multiplied to a random number between 0 and 1. As a result, it returns a random instant in the past (but after 1/1/1970, the beginning of Epoch time [20]).

❸ The same function is expressed using the `#()` function literal syntax. The percent sign inside the function (`%`) is the placeholder for the required parameter (replacing "x" from the previous example). Numbered parameters are also possible if the function literal takes more than one: `%1`, `%2` and so on. There is also a variadic catch-all version `%&` which means "use here all the arguments of the function literal".

Contract

Along with a few other functions and macros in the standard library, `fn` has quite an articulated signature that resembles a little grammar on its own. The following informal contract shows the most important features of `fn` (check the examples below to see unambiguously how they work).

```
(fn <name>? arities)
```

```
arities :=> ^<metamap>? [arity] body
            OR
            (^<metamap>? [arity1] body)
            (^<metamap>? [arity2] body)
            ..
            (^<metamap>? [arityN] body)

arity :=> [<arg1-typehint>? <arg1>
          ..
           <argN-typehint>? <argN>]

body :=> <metamap>? <forms>
```

- "<name>?" is an optional symbol that bounds the generated function to the local scope of the function itself. The name allows the function to be recursive (see examples below).

- "arities" is a list of one or more arity declarations (for example, the function `(fn ([] "a") ([x] "x"))` contains two "arities" of zero and one argument each). Each arity allows for an optional metadata map prefixed with "^", followed by a mandatory vector of arguments and an optional body. In case of single arity, the wrapping parenthesis can be omitted. The content of each vector can be plain symbols or more complex destructuring expressions.
- "^<metamap>" is an optional map of keywords-values pairs. It might contain type hints or custom metadata. The map needs to use the prefix "^" when it appears before the argument vector. When "<metamap>" appears after the argument vector, it doesn't require the prefix.
- "arity" is the content of the argument vector. Except the name, each argument can be individually type-hinted.

- <body>, when present, contains the actual function expressions. It is implicitly wrapped in a do block. It is assumed to be `nil` when there is no body. When the body contains a top-level map and is followed by an expression, then the map is used as metadata. When "<metamap>" appears both before and after the argument vector, the one after takes precedence in case of clashing keys.
- returns: the function object that was just created.

Examples

`fn` is the minimum common denominators for all declaring functions and macros. For instance, type hints given for arguments in a function declared with `defn` are processed by `fn` under the hood. Although implemented in `fn`, type hints or pre- and post-conditions are usually present in `defn` declarations. The reader is invited to check defn examples as well for what is not present in this section.

Named recursion

The first example demonstrates a possible use of the optional name that makes the function bound inside the innermost

scope. It could be used for example in the recursive definition of a basic Fibonacci [21]:

```
((fn fibo [n] ; ❶
  (if (< n 2)
    n
    (+ (fibo (- n 1))
       (fibo (- n 2)))))
10)

;; 55
```

❶An implementation of a function to find the n-th Fibonacci number.

By adding an additional arity with 3 parameters we could now offer a tail-call optimized Fibo without changing the previous definition:

```
((fn fibo ; ❶
  ([n] (fibo 1 0 n))
  ([a b cnt]
    (if (zero? cnt)
      b
      (recur (+ a b) a (dec cnt)))))
10)
```

```
;; 55
```

❶An example of `fn` named lambda declaration that contains two "arities".

fn and destructuring

The function literal syntax `#()` is quite idiomatic in Clojure, but there are cases in which the features it provides are not sufficient: destructuring, for example, is not available with function literal syntax. The following example shows a map being transformed into another by applying a mix of key and value changes. Instead of using the concise but limited `#()` reader literal, we make the lambda explicit with `fn` and introduce destructuring:

```
(def sample-person
  {:person_id         1234567
   :person_name       "John Doe"
   :image             {:url "http://focus.on/me.
                       :preview "http://corporat
   :person_short_name "John"})

(def cleanup
  {:person_id     [:id str]
```

```
    :person_name   [:name (memfn toLowerCase)]
    :image         [:avatar :url]})

(defn transform [orig mapping]
  (apply merge
    (map (fn [[k [k' f]]] {k' (f (k orig))})
      mapping)))

(transform sample-person cleanup)
;; {:id "1234567", :name "john doe", :avatar "ht
```

❶ `cleanup` is a mapping between input key names and a vector pair. The pair contains the new name of the key in the output map and a function to apply to transform the value. For example the first key says that `:person_id` should be renamed into `:id` and the str function should be applied to the value.

❷ The `transform` function takes an input map `orig` and the `mapping` rules as arguments (`sample-person` and `cleanup` are the instances used in the example). The map function is used here to apply all the transformation rules. By using `fn` we can destructure the content of `cleanup` that would not be possible if we used the special reader form `#()`.

Without destructuring the `fn` lambda would be polluted with first or second calls to access the vector elements, as showed by the following re-write of the `transform` function:

```
(defn transform [orig mapping] ; ❶
  (apply merge
    ;; prefer destructuring instead of this
    (map (fn [rules]
           (let [k (first rules)
                 k' (first (second rules))
                 f (second (second rules))]
             {k' (f (k orig))}))
      mapping)))
```

❶Re-write of the `transform` function to illustrate how many repetitions of `first` and `second` are necessary when not using detructuring.

`fn` is Clojure's functional lambda

Functional languages are often characterized by two main features:

- They support higher order functions
- They push towards referential transparency [22]

Higher order functions are functions that can accept other functions as parameters or return functions to their callers. A language needs to support functions as first class objects in the

language so they can be sent around as "data" to other functions. The way function objects are created is different from language to language but historically they have been named **lambdas** (from the Lambda Calculus, the first widely adopted formalized notations for mathematical functions introduced by Alonzo Church around 1930 [23]). Some languages even use `lambda` as a keyword to stress that connection. Clojure doesn't have a `lambda` keyword but `fn` is definitely Clojure's lambda implementation.

Referential transparency guarantees that the return value of a function is only dependent on its parameters and nothing else. Functional languages that enforce referential transparency at some level, often get a number of other features as a consequence: laziness, immutable values, infinite sequences and so on. Clojure is definitely part of the group of mainstream functional languages supporting all of the above.

See also

- fn* is a slight variation of `fn` that also performs "locals clearing" after the first invocation. Please refer to the documentation of `fn*` for more information.

- defn is obviously related to `fn`. The main difference is that `defn` is designed to "intern" the function object to the enclosing namespace through a var object. You should probably think of refactoring an `fn` definition out into a `defn` every time there is some chance of reuse by other functions.
- identity is an example function returning an anonymous function of one argument.

Performance considerations and implementation details

Similarly to `defn`, `fn` processing mainly happens at compile time, so it's not usually a concern in terms of runtime performance. Differently from `defn`, `fn` does not side-effect into creating a var definition that is then added to the mappings of the current namespace.

2.1.3 fn*

(Thanks Nicola Mometto for contributing this section)

`special form` `since 1.0`

Listing 2.3. → *Anonymous function definition, Optimization, Locals Clearing*

```
  (fn* [& fdecl]) ; ❶
```

❶Please refer to defn for an extended version of the supported feature while declaring a new function.

`fn*` is the special form underlying the fn macro. It supports less features, lacking for example support for pre- and post-conditions (or destructuring). The main goal of `fn*` is memory optimization. `fn*` has unique support for creating closure objects with only-once run guarantees.

A normal lambda created by `fn` could be referenced in multiple places (which is usually the case in large applications) and re-used as needed. The Clojure compiler cannot keep track of all the references to the lambda, so after an execution the lambda (and its internal state in the related Java class that is generated) needs to stay around for potential new executions. But there is a certain class of lambdas that are known in advance to run just once: it is the case of delay or future macros for example, which run in an external thread.

These threads are often kept alive in a thread pool and with them, the function objects they ran. The function object, in turns, could be holding a reference to arbitrarily large data, even if the function already returned its results. `fn*` ensures that the references the function hold are set to `nil` atfer the result returns. This is also an important feature to have when writing macros that delegate to wrapping functions, quite a common idiomatic pattern, in order to avoid retaining memory for longer that is actually needed.

Contract

Refer to the contract of fn, keeping in mind the only two differences:

- It has no support for the various metadata maps that `fn` accepts.
- It will assign special compile-time meanings to forms where the `fn*` symbol has `^:once` metadata, (while `fn` doesn't support this feature).

Examples

We'll only showcase the unique "once-only" feature of `fn*`, for all other examples and usages, refer to fn and refrain from using `fn*` directly.

It is both a common pattern and a good practice to implement macros by delegating to their function version, by wrapping the unevaluated bodies in an anonymous function [24]. This has several advantages:

- It makes it easier to understand the implementation of the macro
- By providing a function version it improves its composability and power, since it makes the functionality also available for runtime use rather than just as a compile-time feature.

This exact pattern appears in several places in `clojure.core` itself: `future` is a macro that delegates to the `future-call` function using the same technique just described:

```
(defmacro future [& body] ; ❶
  `(future-call (^{:once true} fn* [] ~@body)))
```

❶The definition of the future macro as it appears in the `clojure.core` namespace.

❷Note the use of the `:once true` metadata key just before invoking `fn*`.

future is going to execute the body at some later point in time in a separate thread. The other important aspect about future design is that the body is meant to be executed only once (that is, the thread is supposed to run once and never re-scheduled). So as the author of a macro like future we know already that the resources used by the function, once executed, can be claimed back by the JVM. We have basically the power to tell the Clojure compiler that once the body has executed, every reference to the lambda in the compiled code can be set to `null`, allowing the JVM to claim resources back as soon as possible. This is an important memory optimization done by the Clojure compiler called "locals clearing" [25].

By just replacing `fn` with `^:once fn*` (thus promising the compiler that the body will never be executed more than once), the compiler is now able to perform the locals-clearing optimization and avoids potential memory leaks [26].

The reader is invited to review future-call where the book explores an example showing the effect of locals clearing.

See also

- fn is the macro that should **always** be used over `fn*` unless you need the `^:once` feature.

- future wraps an expression in a `fn*` function of no arguments with once-only semantic.

Performance considerations and implementation details

Similarly to defn or fn, `fn*` has very little impact during runtime as the actual generation of the function happens during compile-time. For this reason the user should not be concerned with `fn*` when searching for performance improvements.

`fn*` is a special form, which means that its implementation it's a "given" while the compiler is executing. For Clojure in particular, this means that `fn*` implementation only exists as Java code.

[17] See the main Clojure Reader documentation at clojure.org/reader for the definition of a valid Clojure symbol.
[18] More info about how to create a signature with sha256 can be found here: security.stackexchange.com/questions/20129/how-and-when-do-i-use-hmac
[19] For places in the Clojure source where the group of 20 Java methods is visible see for example: `clojure.lang.IFn`
[20] The Unix Epoch time is a system to measure relative time:

en.wikipedia.org/wiki/Unix_time

[21] The popular Fibonacci series is often used to show implementation of recursive calls. For more information see Wikipedia: en.wikipedia.org/wiki/Fibonacci_number

[22] Please see: en.wikipedia.org/wiki/Referential_transparency

[23] From the abundant literature available on the subject, I suggest this gentle introduction to the Lambda Calculus: www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf

[24] This is called creating a thunk: en.wikipedia.org/wiki/Thunk

[25] Rich Hickey describes this feature quite extensively in the following mailing list post: groups.google.com/forum/#!topic/clojure/FLrtjyYJdRU

[26] See also Christophe Grand, who describes this type of memory leak in his blog: clj-me.cgrand.net/2013/09/11/macros-closures-and-unexpected-object-retention/

2.2 HIGHER ORDER FUNCTIONS

2.2.1 fnil

`function` `since 1.2`

Listing 2.4. -> *Function generation, parameter handling, default argument values*

```
(fnil
  ([f default1])
  ([f default1 default2])
  ([f default1 default2 default3]))
```

`fnil` generates a new function starting from another input function "f". The main use case for `fnil` is to decorate "f" so that it can default to optional values in case the input is `nil`. `fnil` operates positionally: "default1" will be used for a `nil` passed as first argument, "default2" for a `nil` passed as the second argument and "default3" for a `nil` as the third argument. `fnil` doesn't support more than 3 defaults, so `(fnil + 1 2 3 4)` causes an exception to be thrown.

Contract

- "f" can be a function of any number of arguments returning any type.
- "default1,default2,default3" are the default values that should be used if the generated function receives a `nil` as its first, second or third argument respectively.

Examples

`fnil` main use case is to wrap an existing function that doesn't handle `nil` arguments the way we want (for example, it could even throw exception). `fnil` replaces the `nil` input with a given default and the default is given to the original function in turn.

One example of exceptional behavior in the presence of `nil` is inc, the simple function that increments a number. We could use `fnil` to define an alternative behavior if, for any reason, `inc` is given `nil` as input. In the following example, we want to update the numerical values in a map with update [27]:

```
(update {:a 1 :b 2} :c inc) ; ❶
;; NullPointerException

(update {:a 1 :b 2} :c (fnil inc 0)) ; ❷
;; {:a 1 :b 2 :c 1}
```

❶ We try to update the ":c" key in a map, but without knowing what is the content in advance, we don't know if the map contains the key or not. inc fails badly if the input is `nil` which is what happens in this case.

❷ We can use `fnil` to wrap the `nil` argument case for inc. If `inc` is given a `nil`, `fnil` replaces the `nil` with 0, which is then given to `inc`.

A typical unpredictable value (especially for a web application) is a string coming from an input form. `fnil` can be handy in this case. In this example, an input form is transformed into the `request-params` map:

```
(require '[clojure.string :refer [split]])

(def request-params ; ❶
  {:name "Jack"
   :selection nil})

(defn as-nums [selection] ; ❷
  (let [nums (split selection #",")]
    (map #(Integer/valueOf %) nums)))

(as-nums (:selection request-params)) ; ❸
;; NullPointerException
```

❶ `request-params` simulates the content of a web form already transformed into a Clojure data structure. Some parameters are structured, like ":selection" which is a comma separated string.

❷ `as-nums` is designed to take the ":selection" parameter, split it into a list of strings and convert those strings into numbers.

❸Unfortunately the user on the web page didn't fill out ":selection" as expected (or something else went wrong) producing a `nil` selection.

The `:selection` key is normally a comma separated list of numbers but it could result in a `nil` if the user doesn't fill the related input field. `as-nums` throws an exception in case of a `nil` selection, because it's calling split on a null string. We can wrap `as-nums` with `fnil` to solve this problem:

```
(def as-nums+ (fnil as-nums "0,1,2")) ; ❶

(as-nums+ (:selection request-params)) ; ❷
;; (0 1 2)
```

❶ `fnil` is now wrapping `as-nums`. The default ":selection" is `0,1,2`.

❷ `as-nums+` is now used instead of the old function, correctly handling a `nil` ":selection" key.

The new function `as-nums+` handles the case by replacing `nil` (the result of retrieving the `:selection` key from the parameters) with "0,1,2" as string (for this particular example we are assuming that "0,1,2" is equivalent to "no selection"). Once defined, the new `as-nums+` can be safely replace any old use of the normal, exception throwing, `as-nums`. `fnil` can operate similarly for the 2nd and 3rd argument, for example:

```
(require '[clojure.string :as string])

(def greetings
  (fnil string/replace "Nothing to replace" "Mor

(greetings "Good Morning!" "Morning" "Evening")
;; "Good Evening!"
(greetings nil "Morning" "Evening")
;; "Nothing to replace"
(greetings "Good Morning!" nil "Evening")
;; "Good Evening!"
(greetings "Good Morning!" "Morning" nil)
;; "Good Evening!"
```

❶The example shows `fnil` handling `nil` arguments for replace and 3 potential exception-throwing `nil` invocations.

Beyond the 3rd argument: an extended `fnil`

`fnil` can handle up to 3 default positional defaults for a given input function. It's relatively easy to extend `fnil` to handle any number of default options:

```
(defn fnil+ [f & defaults]
  (fn [& args] ; ❶
    (apply f
      (map (fn [value default] ; ❷
             (if (nil? value) default value))
        args
        (concat defaults (repeat nil))))))

(+ 1 2 nil 4 5 nil) ; ❸
;; NullPointerException

(def zero-defaulting-sum ; ❹
  (apply fnil+ + (repeat 0)))

(zero-defaulting-sum 1 2 nil 4 5 nil)
;; 12
```

❶ `fnil+` returns a function of any number of arguments.

❷map can take 2 or more sequential arguments. We use this aspect to combine actual arguments and possible defaults.

❸If we try to pass `nil` as part of sum of 5 numbers it throws an error.

❹Assuming "0" is a good default when a number is `nil`, we can use apply with repeat to enhance `+` to accept `nil` arguments in any position.

The new function `fnil+` accepts default values for `nil` in any position. map can take any number of sequential collections to map over, which is handy in many situations. The first sequence "args" is the list of proper arguments to the function. The second sequence passed to map is a concatenation of the given

"defaults" passed to `fnil` and any number of additional `nil` arguments to map over "args".

We also take advantage of map laziness to cover a potentially infinite number of default arguments, like illustrated by `zero-defaulting-sum`. The infinite sequence of zeroes created by `(repeat 0)` covers `nil` values for all the (potentially infinite) arguments to `+`.

The other important aspect to consider with this use of map is that it will automatically stop mapping when reaching the end of the shortest sequence. This is a great example of a function that in just 3 lines shows much of the power available in Clojure. [28]

See also

- some-> can be used to achieve a similar effect to `fnil`. Consider for example `(some-> nil clojure.string/upper-case)`: the form correctly returns `nil` without throwing an exception. `some->` might be a better choice if you need to prevent a function of a single argument to throw exception. However `some->` default value is fixed and can't be changed (it always returns `nil`).

Performance considerations and implementation details

→ `O(1)` **function generation**

→ `O(1)` **generated function**

`fnil` usage doesn't have relevant performance implications. The output function is produced in constant time. Invoking the generated function is also constant time, considering it is constrained by the number of arguments it can handle (3 only).

2.2.2 comp

Listing 2.5. → *Function generation, function composition, transducers*

```
    (comp
      ([])
      ([f])
      ([f g])
      ([f g & fs]))
```

`comp` accepts zero or more functions and returns another function. The new function is the composition of input. Given for example the functions f1, f2, f3, `comp` creates a new function so that: `((comp f1 f2 f3) x)` is equivalent to `(f1`

`(f2 (f3 x))`. This equivalence is the reason why `comp` apparently reads backwards, for example:

```
((comp inc +) 2 2) ; ❶
;; 5
```

❶A simple example of function composition between + and inc.

In the example above, + appears last in the arguments but is the first one to apply.

Note

when invoked with no arguments `comp` returns the identity function. This is helpful in situations where the list of function to compose is dynamically generated at runtime and potentially empty. Instead of dealing with the error case, `comp` will gladly accept an empty list of arguments.

Contract

input

With the exception of the rightmost function (that can take any number of arguments), all other argument functions must accept a single argument. In the case of `(comp f g h)` for example, "h" is the only function that can accept multiple parameters while "f & g" receive a single argument.

notable exceptions

`IllegalArgumentException` when any of the input functions (except the last) does not support a single argument call.

Output

`comp` returns a function of the same number of arguments of the rightmost input parameter, representing the functional composition of all the input functions.

Examples

A concatenation of functions is the main use case for `comp`. Consider the following example where we produce how many stamps we need to buy to send letters to different destinations:

```
(require '[clojure.string :refer [split-lines]])

(def mailing ; ❶
   [{:name "Mark", :label "12 High St\nAnchorage
```

```
      {:name "John", :label "1 Low ln\nWales\n9978
      {:name "Jack", :label "4 The Plaza\nAntioch\
      {:name "Mike", :label "30 Garden pl\nDallas\
      {:name "Anna", :label "1 Blind Alley\nDallas

  (defn postcodes [mailing] ; ❷
    (map #(last (split-lines (:label %))) mailing)

  (postcodes mailing)
  ;; ("99501" "99783" "43793" "75395" "75395")

  (frequencies (postcodes mailing)) ; ❸
  ;; {"99501" 1, "99783" 1, "43793" 1, "75395" 2}
```

❶The input is in the form of a vector or maps, a common format to transfer data with similar structure but different values.

❷The function `postcodes` returns a list of (potentially repeating) postcodes after parsing the content of the `:label` value. Note that the body of the function contains 4 nested calls to other functions (map, last, split-lines and the key ":label" used as function).

❸We can use frequencies to count the number of occurrences of each postcode.

The function `postcodes` contains a mapping function that applies 3 transformations to each mailing item. We could use `comp` to build the same expression using composition [29]:

```
(defn postcodes [mailing] ; ❶
  (map
    (comp
      last
      split-lines
      :label)
  mailing))

(frequencies (postcodes mailing)) ; ❷
;; {"99501" 1, "99783" 1, "43793" 1, "75395" 2}
```

❶We rewrote `postcodes` using `comp`.
❷After the changes we make sure that the results are the same as before.

By using `comp` we added emphasis on the sequence of transformations. This is the effect of removing parenthesis which in turns allow for a natural vertical alignment. Note that the use of `comp` in this case is possible because all functions take 1 parameter.

`comp` is also the main construct to compose **transducers**. Here's the same `postcodes` seen before written using transducers:

```
(defn postcodes [mailing] ; ❶
  (sequence (comp ; ❷
              (map :label)
              (map split-lines)
              (map last))
    mailing))

(frequencies (postcodes mailing)) ; ❸
;; {"99501" 1, "99783" 1, "43793" 1, "75395" 2}
```

❶We rewrote `postcodes` to use transducers.
❷map is now used to generate transducers and we need to use sequence to apply them.
❸Results are still the same as the previous example.

Note the reverse order of the transducing functions compared to the previous version of `postcodes` using map instead of sequence. This is an effect of how transducers are implemented, but the results are the same.

In the following example, we add a step to the transformations to remove Alaska from the list of postcodes and we prevent duplicates in the final output. Note that thanks to composition, we can add transformations using a more appealing vertical alignment:

```
(require '[clojure.string :refer [starts-with? s

(defn alaska? [postcode]
  (starts-with? postcode "99"))

(defn unique-postcodes [mailing]
  (sequence (comp
             (map :label)
             (map split-lines)
             (map last)
             (remove alaska?) ; ❶
             (distinct))
    mailing))

(unique-postcodes mailing)
;; ("43793" "75395")
```

❶The new `unique-postcodes` function removes Alaska from the list and removes duplicates.

See also

- juxt is another function generator. It doesn't compose functions like comp does, but executes them independently and collects the results. Use `juxt` when the input functions operates independently in the input.
- sequence accepts composition of transducers as demonstrated by the examples.
- transduce is the other transducing function that appears frequently with `comp`.

Performance considerations and implementation details

→ `O(1)` **function generation**

Calling `comp` generates a new function in constant time and has little or no interest in performance analysis.

2.2.3 complement

`function` `since 1.0`

Listing 2.6. → *Boolean logic, Function generation*

```
  (complement [f])
```

`complement` is a simple function wrapper. It takes an input function `f` and produces another function as output. The newly created function takes any number of arguments and invokes the wrapped function `f` negating its output with not:

```
((complement true?) (integer? 1)) ; ❶
;; false
```

❶A simple example of using `complement` to invert the meaning of checking if a value is an integer.

Contract

input

- "f" is a function of any number of arguments returning any type.

output

- returns: a function of any number of arguments returning boolean true or false.

Examples

`complement` takes advantage of the fact that everything in Clojure has an extended boolean meaning and always returns either `true` or `false`:

```
((complement {:a 1 :b 2}) :c) ; ❶
;; true

((complement {:a 1 :b nil}) :b) ; ❷
;; true
```

❶The example shows how to invert the meaning of validating the presence of a key in a map. If `:c` is not present in the map it returns `true`.

❷However, we should pay attention using `complement` in the presence of `nil` values. In this second case `:b` is present in the map but its value is `nil`.

`complement` should be used with care in the presence of `nil`, as demonstrated by the example above. However, if the presence of `nil` is under control, we can check if an item does not belong to a set in a very concise way:

```
(filter (complement #{:a :b :c}) [:d 2 :a 4 5 :c
;; (:d 2 4 5)
```

```
(filter (complement #{nil :a 2}) [:a 2 nil nil])
;; (nil nil)
```

❶A concise way to filter all items that don't match a set of values.

❷The approach assumes the complemented set does not contain `nil` as one of the values to remove. In that case, it won't be able to remove `nil` from the input.

`complement` offers the possibility to extract a function from a negated function. We are unable to do the same with not which is a boolean operator. Here's for example a way to express typical opposites like "left" and "right" in terms of each other:

```
(defn turning-left? [wheel]
  (= :left (:turn wheel)))

(def turning-right?
  (complement turning-left?)) ; ❶

(defn turn-left [wheel]
  (if (turning-left? wheel)
    (println "already turning left")
    (println "turning left")))

(defn turn-right [wheel]
```

```
    (if (turning-right? wheel) ; ❷
      (println "already turning right")
      (println "turning right")))
```

❶We define `turning-right?` in terms of `turning-left?` using `complement`. Note that we cannot use not to negate a function, as not only accepts boolean input.

❷We could also write: `(if-not (turning-left? wheel))` at the price of a negation in the statement.

The result of calling `(turning-right? wheel)` and `(not (turning-left? wheel))` is the same, but the presence of not produces a slightly less readable form. This is especially true when modelling opposite concepts with an established name (such as left and right, up and down, North and South and so on). Clojure itself contains such an example. Here's the definition of remove:

```
  (defn remove [pred coll] ; ❶
     (filter (complement pred) coll))
```

❶Some implementation details have been removed from the implementation of remove as it appears in the standard library.

See also

- not does not produce a function but just inverts the boolean meaning of its argument.

Performance considerations and implementation details

→ `O(1)` **function generation**

`complement` is not particularly interesting in performance analysis: it generates the requested function in constant time. The generated function, when invoked, applies its arguments (up to 20) using apply.

2.2.4 constantly

`function` `since 1.0`

Listing 2.7. → *Functional generation, extended boolean logic*

```
  (constantly [x])
```

`constantly` generates a function that always returns the same result independently from the number and type of arguments it is called with. The output function always returns the initial argument as the only answer.

Contract

input

- "x" a mandatory argument of any type used as the returned result from the generated function.

output

- returns: a new function of 0 or more parameters of any type.

Examples

`constantly` can be used in all situations where an updating function is required, but the new value doesn't depend on the old. update for instance, takes a map, a key and a function. The function receives the old for the key and is expected to use that value to compute the new.

The following example implements a `quantize-volume` function to calculate the average volume in a collection of musical notes. The sound expressiveness is expressed by both the `:volume` and the `:expr` keys:

```
  (def notes [{:name "f" :volume 60 :duration 118
              {:name "f" :volume 63 :duration 120
```

```
             {:name "a" :volume 64 :duration 123

(defn- expressiveness [average exp]
  (case exp
    ">" (+ average 5)
    "<" (- average 5)
    average))

(defn- process-note [note fns]
  (letfn [(update-note [note [k f]] (update note
    (reduce update-note note fns)))

(defn quantize-volume [notes]
  (let [avg (quot (reduce + (map :volume notes))
        fns {:volume (constantly avg)
             :expr (partial expressiveness avg)}
    (map #(process-note % fns) notes)))

(quantize-volume notes)

;; ({:name "f", :volume 62, :duration 118, :expr
;;  {:name "f", :volume 62, :duration 120, :expr
;;  {:name "a", :volume 62, :duration 123, :expr
```

❶ `process-note` takes a note and a map of functions. Clojure maps support sequential access and can be used as input for reduce.

❷ `update-note` is locally bound with letfn. It defines the reducing function used by reduce in the following line. Apart from destructuring the second argument, it applies update on a note with the given key and function.

❸ `process-note` applies update multiple times (one for each updating function passed as parameter in the "fns" `map`). Since Clojure hash-map are persistent data structures, we need to make sure that the updated note that each function produces is the input of the next updating function. reduce implements exactly the updating semantic we are looking for, making sure each intermediate step is passed as input to the following. Our "initial value" for reduce becomes "the note" and we are starting the update-chain from there.

❹ `quantize-volume` main goal is to prepare the input functions for update and apply them to all the notes.

❺Each note has a `:volume` key and we want all volumes to be the average. `constantly` is a good choice here: we need the same "average" value for all notes and a function wrapper that returns that value.

❻ `:expr` key needs the old value to determine the new value, so we pass in a function of the old `expressiveness` to the

new.

❼ When we finally process the notes, we can see that the maps are updated as expected, with `:volume` updated to the average and `:expr` updated in relation to the volume being above or below the average.

Another use of `constantly` is for "stubbing" function calls in testing. A good setup for a test isolates the function under test from less predictable behavior (such as network requests) providing stubbed responses. The stubbed response is also useful to control a particular aspect of the function under test, so its behavior can be verified. with-redefs is often used in conjunction with `constantly` for this purpose:

```
(ns book.unit
  (:require [clojure.test :refer [deftest testin

(defn- third-party-service [url p1 p2] ; ❶
  "Simulation of expensive call"
  (Thread/sleep 1000)
  {:a "a" :b "b"})

(defn fn-depending-on-service [s] ; ❷
  (let [result (third-party-service "url" "p1" "
    (if (= "b" (:b result))
      (str s "1")
      (str s "2"))))
```

```
(deftest test-logic
  (with-redefs [third-party-service (constantly
    (testing "should concatenate 2"
      (is (= "s2" (fn-depending-on-service "s"))
```

❶ `third-party-service` is a function simulating some expensive call to a service we don't control. It requires 3 parameters.

❷ `fn-depending-on-service` is a function that has the 3rd party service call as a dependency.

❸The unit test creates a local redefinition binding with with-redefs, swapping the 3rd party call with a stubbed version. Since we are not interested in the parameters that are passed to the function, we can use `constantly` to stub the return value.

Warning

With direct linking [30] switched on, with-redefs stops working. `with-redefs` relies on var indirection to temporarily swap the function implementation. When Clojure is compiled with direct linking, the vars content is inlined directly and cannot be changed.

Polyvariadic functions in statically typed languages

High-order functions like `constantly` are simple to implement in dynamic languages and quite idiomatic. It is possible, but trickier, to do the same in statically typed languages. The level of complexity varies from language to language. The Hindley-Milner language family for example (ML and descendants like Haskell or OCaml), supports "polyvariadic" functions, but it requires some effort to get types right (see this paper for instance). Considering their type complexity, "polyvariadic" functions are not idiomatic in statically typed languages like Haskell, but equally powerful alternatives like pattern matching definitely are.

See also

- identity also returns the argument that is passed in as a parameter. But `identity` doesn't return a function, just the value itself. `identity` is often used with similar goals to `constantly`, with the only restriction that `identity` only accepts one argument instead of many.

- with-redefs is used often in conjunction with `constantly` to generate stubbed responses while testing.

Performance considerations and implementation details

→ `O(1)` **function generation**

→ `O(1)` **generated function**

`constantly` simply returns the same results each invocation. Both function generation and use of the generated function are constant time operations

2.2.5 identity

`function` `since 1.0`

Listing 2.8. → *utility function, higher order function*

```
  (identity [x])
```

`identity` is a little function in the standard library. It just returns its single argument as output:

```
  (identity 1)
```

```
  ;; 1
```

Despite the apparent simplicity, there are many cases in which `identity` can be useful (see the example section). `identity` derives its name from the equivalent mathematical concept (also called identity transformation or relation [31].

Listing 2.9. Contract

```
    (identity x)
```

input

- "x" is the only mandatory argument and can be of any type.

output

- returns: the argument that was passed as input.

Examples

The first example illustrates an idiomatic way to transform a map into a flat sequence of keys and values. It is a single liner and a single function call. All other options would include a second function call:

```
(mapcat identity {:a 1 :b 2 :c 3}) ; ❶
;; (:a 1 :b 2 :c 3)
```

❶ An idiomatic use of mapcat with `identity` to transform a map into a sequence.

mapcat iterates and concatenates at the same time. Since iterating over a hash-map produces a sequence of vectors containing key-value pairs, we just need the `identity` transformation before concatenating all vectors together.

`identity` can also be used as "noop" (contraction for no-operation) to provide a function when one is required, but it shouldn't produce any effect. One useful case is when we need to filter all logical false elements from a sequence (anything that is either `nil` or `false`):

```
(defn custom-filter [x] ; ❶
  (if (or (nil? x) (false? x))
    false
    true))

(filter custom-filter [0 1 2 false 3 4 nil 5])
;; (0 1 2 3 4 5)
```

```
  (filter identity [0 1 2 false 3 4 nil 5])
  ;; (0 1 2 3 4 5)
```

❶ `custom-filter` implements what we want to achieve in a very verbose way: we are not considering the fact that Clojure accepts any value as logical true/false, so this is considered not idiomatic.

❷Shows that `custom-filter` works as expected, filtering out all unwanted `nil` and `false` from the sequence.

❸The same result can be achieved without a custom function using `identity`. The reason this works is that values like `false` or `nil` are part of Clojure logical `false` definition. filter works using `nil` or `false` as markers for what items should or should not be in the final results. `identity` passes values as they are, so filter can use them directly.

The following example shows how to use `identity` with some to retrieve the next logical true element from a collection. A list of cashiers in a super-market is marked available by adding a number in a vector at its corresponding index. As soon as the customer picks a lane, the cashier becomes busy and we need to update the value at the index so no other customer can pick the same lane. To avoid concurrent read/write of the cashier line we use a ref, one of the concurrency primitives in Clojure. By

using a `ref` we can check the availability and book the lane in a single transaction:

```
(def cashiers (ref [1 2 3 4 5])) ; ❶

(defn next-available [] ; ❷
  (some identity @cashiers))

(defn make-available! [n] ; ❸
  (alter cashiers assoc (dec n) n) n)

(defn make-unavailable! [n]
  (alter cashiers assoc (dec n) false) n)

(defn book-lane [] ; ❹
  (dosync
    (if-let [lane (next-available)]
      (make-unavailable! lane)
      (throw (Exception. "All cashiers busy!")))))

(book-lane) ; ❺
;; 1

(book-lane)
;; 2

(dosync (make-available! 2))
;; 2
```

```
@cashiers
;; [false 2 3 4 5]
```

❶ `cashiers` contains a vector initialized with numbers (representing free cashiers lanes). The vector is wrapped by a `ref`.

❷ `next-available` uses identity and some on the `vector` of cashiers. It returns the first `true` result, or `nil` after reaching the end of the `vector`. Note that `next-available` is a read-only operation on the `ref` that doesn't need an explicit transaction context.

❸ `make-available!` and `make-unavailable!` take a number as argument and add or remove the element at that index. This effectively marks the cashier available or not, because marking "not available" adds a `false` in the vector at that index causing `next-available` to continue the search.

❹ `book-lane` coordinates searching for the next available cashier and booking a lane. `dosync` needs to wrap both read/write operations to be effective, as other customers might be trying to use the same lane simultaneously. In case there are no more lanes available, `book-lane` throws an exception.

❺ We can see a quick simulation of the system by booking and releasing a few lanes.

`identity` can be used with partition-by when we are interested in grouping consecutive elements in a sequence. The

following example shows how to search for prominence in words, assuming repeating letters indicates emphasis:

```
(def they-say  ; ❶
  [{:user "mark" :sentence "hmmm this cake looks
   {:user "john" :sentence "Sunday was warm outs
   {:user "steve" :sentence "The movie was sooo
   {:user "ella" :sentence "Candies are bad for

(defn- enthusiast? [s]
  (> (->> (:sentence s)
          (partition-by identity) ; ❷
          (map count)
          (apply max)) 2))

(defn enthusiatic-people [sentences]
  (->> sentences
       (filter enthusiast?)
       (map :user)))

(enthusiatic-people they-say) ; ❸
;; ("mark" "steve")
```

❶ `they-say` contains a sample of sentences and their authors.
❷ We use partition-by with `identity` as the discriminating function, so letters appears together only when they are the same in a sequence. After creating groups, we can count them and see if there is a group bigger than 2 letters.
❸ We can see who is using a "colorful" language in the sample.

See also

- nil? is a better option to use as a predicate with filter to deal with `nil` elements in the sequence. As we have seen in the examples, `identity` use with filter also removes `false` elements, while nil? does not:

```
(remove nil? [0 1 2 false 3 4 nil 5])
;; (0 1 2 false 3 4 5)
```

- constantly returns a function that accepts any number of arguments but always the same given result. Use constantly instead of `identity` if you need a variable number of arguments and to return the same result.

Performance considerations and implementation details

=> `O(1)` **function generation**

`identity`, one of the simplest functions in the standard library, is not particularly interesting in performance analysis.

2.2.6 juxt

`function` `since 1.1`

Listing 2.10. → *Function generator, function aggregation, higher order function*

```
(juxt
  ([f])
  ([f g])
  ([f g h])
  ([f g h & fs]))
```

`juxt` takes as an argument a list of functions and returns a new "juxtaposing" function that applies each original function to the same set of arguments. All results are then collected in a vector. `juxt` could be described as a "function multiplexer" since it calls multiple functions to return multiple result. Here's how you can use `juxt` to see the different effects of calling first second and last on a list:

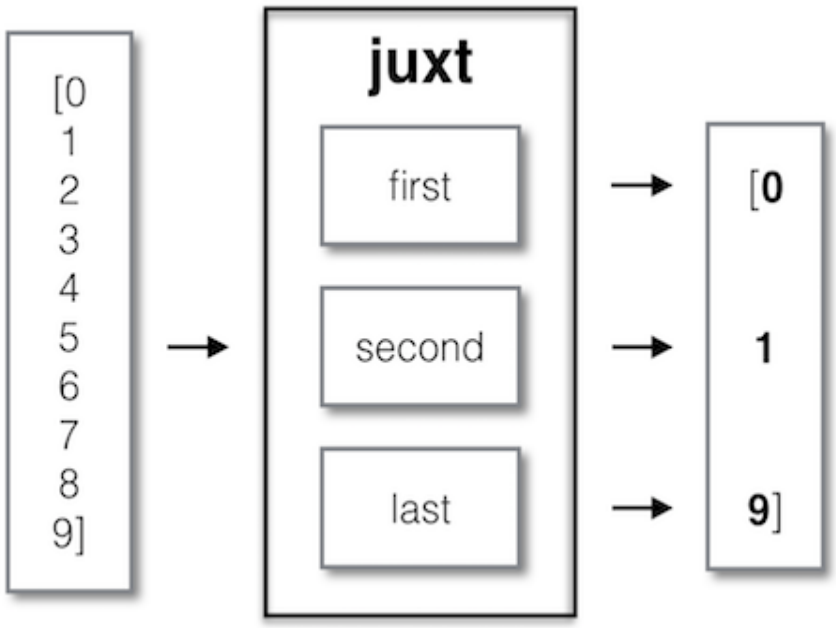
```
((juxt first second last) (range 10)) ; ❶
```

```
;; [0 1 9]
```

❶A simple `juxt` example.

We can describe the example above "visually" with the following picture:

Figure 2.1. Effect of calling juxt with 3 simple functions on a collection.



Contract

input

- `juxt` requires at least one argument up to an unlimited number of arguments.
- "f", "g" and "h" are functions. They need to accept the same number of arguments the output function will be called with. If for example the generated function is called with 2 parameters, then "f", "g" and "h" will be called with those 2 parameters.
- "fs" is any additional function after "f", "g" and "h".

notable exceptions

- `clojure.lang.ArityException` when `juxt` is invoked without arguments or when the generated function is called with the wrong number of arguments.

output

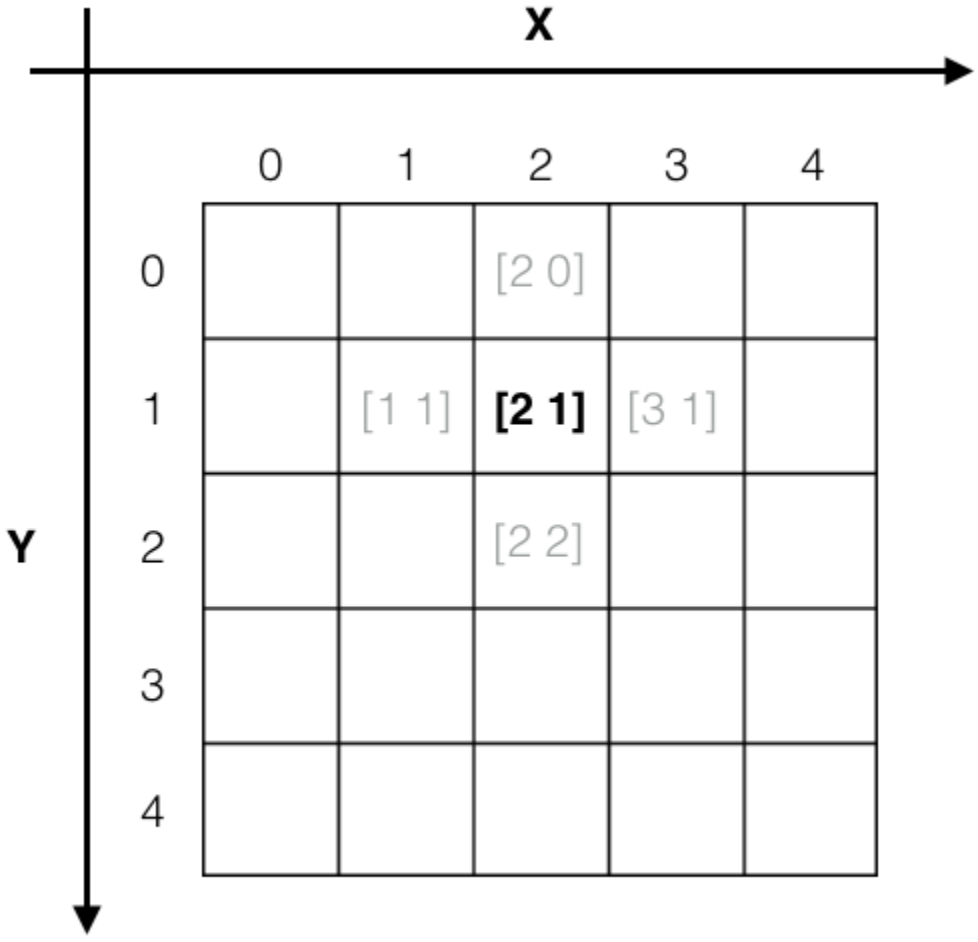
- `juxt` returns a function of any number of arguments returning a vector. The resulting vector has size equal to the number of the initial functions.

Examples

`juxt` is useful to group multiple actions together. One simple case is searching for neighbors in a grid of cells identified by

two-dimensional coordinates. The neighbors are the 4 cells sitting above, below, left and right of another cell. Given a pair of coordinates `[x, y]`, we need to apply 4 transformations to find the adjacent cells. The following diagram shows the cell `[2 1]` and its neighbors:

Figure 2.2. The neighbors of cell [2 1].



We need to be careful though, because the grid has finite dimensions and we don't want to return non-existing neighbors:

```
(def dim #{0 1 2 3 4}) ; ❶

(defn up    [[x y]] [x (dec y)]) ; ❷
(defn down  [[x y]] [x (inc y)])
(defn left  [[x y]] [(dec x) y])
(defn right [[x y]] [(inc x) y])

(defn valid? [[x y]] ; ❸
  (and (dim x) (dim y)))

(defn neighbors [cell]
  (filter valid?
    ((juxt up down left right) cell))) ; ❹

(neighbors [2 1]) ; ❺
;; ([2 0] [2 2] [1 1] [3 1])

(neighbors [0 0]) ; ❻
;; ([0 1] [1 0])
```

❶ `dim` defines the possible values for the coordinates of a grid using a 0-based indexing.
❷ `up, down, left, right` are functions taking a coordinate pair `[x y]` and computing the coordinates of the cell above, below, left or right respectively.
❸ `valid?` is a function that returns `true` if the given [x,y] cell is contained inside the given grid dimensions.
❹ `juxt` groups together the functions we need to calculate the neighbors in a single call.
❺ We can see that these are valid coordinates looking at the diagram above.
❻ This is an example of cell at the edge of the grid returning the only two available neighbors.

Another idiomatic use of `juxt` serves the purpose of maintaining some unaltered version of a value along with its transformations. If for example we have a vector of words and we want to show their length we could use `juxt` and identity:

```
(def words ["book" "this" "an" "awesome" "is"])

(map (juxt count identity) words) ; ❶
;; ([4 "book"] [4 "this"] [2 "an"] [7 "awesome"]
```

❶An example of using `juxt` to decorate each word in a sentence with its length.

By using `juxt` we are able to map over the sequence of words, keep a copy of the word unchanged and decorate the word with its length. We could achieve a similar result using an anonymous function but we would have to deal explicitly with parameters and wrap the results in a vector:

```
(map #(vector (count %) %) words) ; ❶
;; ([4 "book"] [4 "this"] [2 "an"] [7 "awesome"]
```

❶An alternative version for juxtaposing functions using an anonymous function instead of `juxt`, resulting in a more complicated form to read.

Another fairly common use of `juxt` is as a helper to extract values from a map. The following example shows how we can create a message by joining together the relevant keys:

```
(require '[clojure.string :refer [join]])

(def post
  {:formatted-tag "Fireworks 2016"
   :destinations ["north" "south"]
```

```
    :count 200
    :css-align "ending"
    :normal-title "people expected tonight"
    :headline "Admiral Derek on the ship to Nebra

(->> post
  ((juxt :count :normal-title)) ; ❶
  (join " "))                   ; ❷

;; "200 people expected tonight"
```

❶ `post` is an example map of data that we thread with `->>` through a set of transformations. The first transformation creates a function with `juxt` that is applied to the `post` map. The output shows the values corresponding the `:count` and `:normal-title` keys.

❷`join` produces a single string by interposing a space between the input sequence.

In the presence of a list of maps, we could use `juxt` with `sort-by` (or `group-by`) to sort a sequence of maps by more than one attribute in a nested fashion:

```
(sort-by (juxt count str) ["wd5" "aba" "yp" "csu
```

❶This call to sort-by is first sorting by count and then by alphabetical order between those strings with the same size.

Nested grouping is common when handling tabular data, such as a database result-sets. The following `person-table` definition shows how some raw data might appear once loaded in memory. We can query the table using a combination of sort-by, group-by and `juxt`:

```
(def person-table ; ❶
  [{:id 1234567 :name "Annette Kann" :age 31 :ni
   {:id 1000101 :name "Emma May" :age 33 :nick "
   {:id 1020010 :name "Johanna Reeves" :age 31 :
   {:id 4209100 :name "Stephen Grossmann" :age 3

(def sort-criteria (juxt :age :nick)) ; ❷
(def group-criteria (juxt :age :sex))

(defn sort-by-age [t] ; ❸
  (->> t
    (sort-by sort-criteria)
    (map sort-criteria)))

(sort-by-age person-table)
;; ([31 "Ann"] [31 "Jackie"] [33 "Emma"] [33 "St

(defn group-by-age-sex [t] ; ❹
```

```
    (->> t
      (group-by group-criteria)
      (map (fn [[k v]] {k (map sort-criteria v)})))

  (group-by-age-sex person-table)
  ;; ({[31 :f] ([31 "Ann"] [31 "Jackie"])}
  ;;  {[33 :f] ([33 "Emma"])}
  ;;  {[33 :m] ([33 "Steve"])})
```

❶ A person table is represented here as a sequence of maps. This is typically the result of querying a table from a SQL database. Each record contains attributes for a person and we are interested in presenting them in some useful way.

❷ `sort-criteria` and `group-criteria` are definitions for two functions returned by `juxt`. We extract them into their own var definitions, so we can reuse these criteria elsewhere.

❸ `sort-by-age` uses the sort criteria created with `juxt` to sort the table before mapping it (again using `juxt`) to only show the relevant attributes. Note that when used in conjunction with `sort-by` or `group-by`, `juxt` has the meaning of "first sorting-by the first argument" and when in this sorted condition already also "sort-by the second argument". `juxt` is effectively nesting sort and grouping operations.

❹ `group-by-age-sex` is similar to `sort-by-age` just applying slightly different criteria. As before, we are using criteria created with `juxt` both for grouping and for filtering only the interesting keys in the map operation.

See also

- comp has some similarities with `juxt` in that they both compose multiple functions into one, but they are different in the way the functions are composed to obtain the final result, for example: `((comp f g h) x)` is equivalent to `(f (g (h x)))` while `((juxt f g h) x)` is equivalent to `[(f x) (g x) (h x)]`. Use comp instead of `juxt` when the goal is for each function output to be the input for the next function. Use `juxt` when functions should operate in parallel on the same input.
- select-keys should be preferred to filter keys and values from a map. We've seen in our examples that `juxt` can be use effectively as a "select-values" only instead.
- zipmap can be used to create pairs from a sequence similarly to `(map (juxt :somekey identity) maps)`, but with a slightly different syntax: `(zipmap (map :somekey maps) maps)`. While zipmap results in an unordered map, map with `juxt` creates a vector of pairs that maintains order.

Performance considerations and implementation details

→ `O(1)` **function generation**

→ `O(n)` **generated function** (n is the number of arguments)

`juxt` is unlikely to be involved with performance bottlenecks because the number of input functions that are typically composed together is rather small. In the event of macros expanding into calls to `juxt` with many input functions (> 1000 as a starting point), you should consider the linear profile which is necessary to invoke the generated function. This consideration excludes the performance profile of the input functions that should be considered separately.

The implementation of `juxt` for arities less than 3 arguments simply invokes each function in a vector to return the result. The variable-arity case is handled with apply or reduce.

2.2.7 memfn

`macro` `since 1.0`

Listing 2.11. → *Java Interop, Higher Order Functions*

```
  (memfn [name & args])
```

`memfn` allows Java instance methods to be passed as arguments to Clojure functions. The following example shows

the Java method `toUpperCase` of the `java.lang.String` class passed to the map function with and without `memfn`:

```
(map (memfn toUpperCase) ["keep" "calm" "and" "d
;; ("KEEP" "CALM" "AND" "DRINK" "TEA")

(map toUpperCase ["keep" "calm" "and" "drink" "t
;; RuntimeException: Unable to resolve symbol: t
```

❶The correct approach uses `memfn` to make sure `toUpperCase` is called on each string in the vector.
❷The wrong approach shows that Clojure is trying to resolve `toUpperCase` as a symbol.

`memfn` produces effects similar to an anonymous function: it wraps some computation that can be sent later to other functions. Differently from `fn` or the corresponding `#()` reader literal, `memfn` just serves the single purpose of wrapping Java interop calls.

Listing 2.12. Contract

```
(memfn [<tags> name & <args>])
```

input

- "tags" is an optional list of meta tags (in the form of ^:tagname1 ^:tagname2 separated by spaces). "tags" are propagated to the target object receiving the method call. The main use of tags is for type hinting.
- "name" must be a symbol representing a callable method on a Java class.
- "args" is an optional enumeration of symbols. The optional "args" are delegated to the Java method.

notable exceptions

- `RuntimeException` in the unlikely case you need to pass more than 20 parameters to `memfn`.

output

- `memfn` returns a new function of at least one argument. The first argument is the Java object instance the function will receive. Any additional arguments after the first is passed as is to the method invocation.

Examples

The value provided by `memfn` is mainly related to Java interoperability scenarios, especially when higher order

functions are executed on Java objects. The example below shows how to process a sequence of `java.time.Instant` to find their duration from an initial instant `t0`. Alternatively, the time at the invocation is taken as the starting point:

```
(import '[java.time Instant Duration])

(def instants
  (repeatedly (fn []                          ; ❶
    (Thread/sleep (rand-int 100))
    (Instant/now))))

(defn durations [instants & [t0]]
  (let [start (or t0 (Instant/now))]
    (->> instants
      (map #(Duration/between % start))       ; ❷
      (map (memfn toMillis)))))               ; ❸

(let [two (doall (take 2 instants))]
  (durations two))                            ; ❹
;; (67 0)

(let [t1 (Instant/now)                        ; ❺
      times (doall (take 2 instants))]
  (Thread/sleep 200)
```

```
    (first (durations times t1)))
  ;; 21144
```

❶repeatedly takes a function of no arguments and returns an infinite sequence of invocations of that function. `instants` uses repeatedly to sleep some rand-int amount of time and then add the current time in the sequence. The result is lazy infinite sequence of instants.

❷ `Duration/between` is a call to a static Java method. Compared to an instance method, a static method does not require the use of `memfn`.

❸ `memfn` wraps the instance method symbol `toMillis`. This is then used as higher order function for map.

❹We take and realize a couple of instants for our experiment. Without doall, `(Instant/now)` is invoked at some later time confusing the results.

❺The second invocation shows that we can pass our own starting point to measure durations.

`memfn` expands into an anonymous function definition that can be seen if we macroexpand the form:

```
  (macroexpand '(memfn toMillis))
  ;; (fn* ([target24222] (. target24222 (toMillis)
```

❶macroexpand shows that `memfn` is effectively creating a function. The body of the function uses the Java interop syntax (single-dot) to invoke an instance method on the given argument.

One aspect to keep in mind when using `memfn` is related to multiple arguments. If the instance method requires one or more arguments `memfn` can be instructed to pass them through by adding arguments like "ch" in this example:

```
(map (memfn indexOf ch) ["abba" "trailer" "dakar
;; (0 2 1)
```

The map of two sequences requires a function of two arguments, one for the item from the first sequence and one for the second. The first argument is implicit (because `memfn` generates a function of at least one argument to pass to the instance method) while the second must be explicit like in the example above (indicated by the symbol "ch"). You need to be careful though. Additional parameters passed to `memfn` should not to be confused with partial application: for example the following attempt to find the index of a letter in a string doesn't compile:

```
(map (memfn indexOf "a") ["abba" "trailer" "daka
;; CompilerException java.lang.Exception: Unsupp
```

The reason for the problem becomes clear if we macroexpand the form:

```
(macroexpand '(memfn indexOf "a"))
;; (fn* ([target12358 p__12359]
;;   (clojure.core/let ["a" p__12359] (. target1
```

❶The macro expansion shows that the string "a" is used as a local binding in a let form. This is the reason why all arguments after the first need to be valid symbols (at compile-time) and valid for the method signature (at run-time).

See also

- fn and related function literal `#()`, can be used in all places where `memfn` is used. `memfn` is however a better choice to invoke Java instance methods as higher order functions.

- doto helps with side-effecting Java methods invocations, allowing multiple invocations to be chained together. If the instance method you need to call is side-effecting (like a setter method), prefer doto instead.

Performance considerations and implementation details

→ `O(1)` Constant time macro expansion and run-time invocation

`memfn` implementation is relatively simple. The macro expands into a let form followed by a basic function definition using fn. At run-time the function invokes the Java interop call:

```
(defmacro memfn [name & args]
  (let [t (with-meta (gensym "target") (meta nam
    `(fn [~t ~@args]
       (. ~t (~name ~@args))))) ; ❷
```

❶Note that any metadata associated with the first parameter is propagated to the target object instance.

❷The optional args are "unquote-spliced" after the method name.

One important performance aspect affecting `memfn` is related to reflective calls during Java interoperability. The following example shows what happens when `memfn` is used without type hinting:

```
(set! *warn-on-reflection* true)

(time (dotimes [n 100000] ; ❶
  (map (memfn toLowerCase) ["A" "B"])))
;; Reflection warning, form-init41.clj:1:6 call
;; "Elapsed time: 11.294886 msecs"
;; nil
```

❶We use dotimes and `time` (a rudimentary but sufficient benchmarking approach) to show a rough estimate of the time consumed to map `toLowerCase` 100000 times.

`memfn` accepts and propagates metadata when present on the first argument. We can use this aspect to type hint the Java call to remove reflection:

```
(time (dotimes [n 100000]
  (map (memfn ^String toLowerCase) ["A" "B"])))
```

```
  ;; "Elapsed time: 5.701509 msecs"
  ;; nil
```

❶ The type hint appears to affect `toLowerCase` but `memfn` is propagating it to the right place after macro expansion.

With the type hint in place the time consumed cuts down into approximately a half. It's probably safe to suggest that the presence of `memfn` (especially when used as higher order function on collections) should trigger a check for the presence of expensive reflective calls.

2.2.8 partial

`function` `since 1.0`

Listing 2.13. → *Higher order functions, currying, deferred invocation*

```
  (partial
    ([f])
    ([f arg1])
    ([f arg1 arg2])
    ([f arg1 arg2 arg3])
    ([f arg1 arg2 arg3 & more]))
```

`partial` is used when a function requires one or more arguments but not all of them are available at the time of the invocation (for example because the function is passed to another function):

```
(def incrementer (partial + 1)) ; ❶
(incrementer 1 1) ; ❷
;; 3
```

❶ `partial` "injects" the parameter "1" into the function `+` creating another `incrementer` function as output. The `incremeneter` function does not evaluate, waiting for other parameters to be available at some later point.
❷ `incrementer` is evaluated with 2 additional parameters, bringing the total of passed parameters to 3.

`partial` produces an effect similar to suspending the execution of a function by injecting the available parameters at that point and then waiting for all others to be available.

Contract

- "f" is a function of at least one argument. `partial` will accept a function of zero arguments, but at that point `partial` would unnecessarily wraps the target function without adding value.
- "arg1, arg2, arg3" are optional parameters of any type
- "more" just represents the rest of the potential arguments as a variable arity
- returns: a newly generated function of a variable number of arguments that will append any additional argument to the target function before invoking it.

How the generated function will behave depends on the initial target function: invoking the generated function with more than the total allowed arguments will result in an error:

```
(def finder (partial clojure.string/index-of "to

(finder "tons")
;; 0

(finder "tons" (count "tons"))
;; 9

(finder "tons" 5 "unsupported")
;; ArityException Wrong number of args (4) passe
```

❶ `finder` holds a reference to a partial created on top of clojure.string/index-of which takes up to 3 arguments. We pass in the target text to be searched.

❷ the first occurrence of the word "tons" is found at index zero as expected.

❸ to search other occurrences, we need to search past the first. So we use the third argument to tell where we want to start searching from this second time.

❹ `partial` is not preventing us to go beyond the max number of arguments for the target function, despite the fact that the generated `finder` accepts any number of arguments.

Maybe more interesting is the fact that `partial` is positional and can capture arguments strictly starting from the left. Suppose from the previous example that we only want to "fix" the word to search to always be "tons", but leave the target text as a free variable. `partial` won't allow us to do that.

Examples

`partial` achieves a similar effect to fn (or the literal syntax `#()`) but with reduced flexibility. While fn supports the missing arguments in any position (and with possible gaps), `partial` only allows missing arguments at the end of function signature:

```
(let [f (partial str "thank you ")] (f "all!"))
;; "thank you all!"

(let [f #(str %1 "thank you " %2)]  (f "A big "
;; "A big thank you all!"
```

❶"all!" with `partial` can only close the sentence. We can't add something before "thank you" for example.

❷Using the function literal `#()` (which is syntactic sugar for the anonymous function fn) we have the flexibility to accept additional arguments to place before the others.

`partial` still has an advantage compared to fn in removing some superfluous parenthesis. `partial` works particularly well when part of the arguments of a function are "configuration" for some other argument, for example like naming reusable computational bits that can be later composed:

```
(defn as [x]
  (partial = x))                          ; ❶

(defn same? [item coll]
  (apply (as item) (seq coll)))           ; ❷
```

```
(def all-a? (partial same? \a))            ; ❸
(def all-red? (partial same? :red))        ; ❹

(all-a? "aaaaa")                           ; ❺
;; true
(all-red? [:red :red :red])                ; ❻
;; true
```

❶ `partial` is used to suspend equality "=" after the first argument. We want something to be equal to "x" but we still don't know what that is going to be.

❷ `same?` contains the call to `(as item)` passing as many arguments are in the collection passed as argument. This will always work for equality that can take any number of arguments.

❸we use `partial` again to suspend the second argument to `same?` to create another specialization of "=" to just check for the single character `\a` in a sequence.

❹similarly we can do for other kind of items (for example keywords) and re-use everything we have written so far.

❺as expected it works for strings (which are sequences of characters)

❻thanks to another specialization of partial we can use `all-red?` on a collection of keywords

Although it can always be re-written as a custom function, `partial` should be preferred to give emphasis to the fact that an already existing function isn't receiving all the necessary arguments at write-time. This aspect of `partial` works especially good with sequence processing functions (map, filter etc.) like demonstrated in this short example about validating a map:

```
(defn- validate [whitelist req]                ; (
  (and (every? not-empty (vals req))
       (every? whitelist (keys req))))

(def valid-req
  {:id "1322"
   :cache "rb001"
   :product "cigars"})

(def invalid-req
  {:id "1323"
   :cache "rb004"
   :spoof ""})

(map (partial validate #{:id :cache :product}) ;
  [valid-req invalid-req])
;; (true false)
```

❶ `validate` applies some simple rule to keys and values of a map, combining them into a final true/false answer with every. It also requires a list of "whitelisted" keys for one of the validation check.

❷ the map operation happens on a collection of maps to be validated. We know at this point which keys are allowed in each map while the actual map is coming from the internal iteration. `partial` is giving us all the aspect of the higher order function we need: it accepts the argument we already know have and creates the single argument function we need for map. The alternative of using an anonymous function here is still possible but more verbose.

Currying

`partial` application in Clojure is often compared to "currying". Depending on the level of formalism, the two concepts are often conflated but there is usually a difference related to the level of support to currying by the hosting programming language. Currying is a mathematical inspired concept related to the fact that a function of multiple arguments can be expressed as a concatenation of functions of a single argument. The following examples illustrates the idea:

```
(defn f1 [a b c d]
  (+ a b c d))

(defn f2 [a b c]
  (fn [d]
    (+ a b c d)))
```

```
(defn f3 [a b]
  (fn [c]
    (fn [d]
      (+ a b c d))))

(defn f4 [a]
  (fn [b]
    (fn [c]
      (fn [d]
        (+ a b c d)))))

(f1 1 2 3 4)
((f2 1 2 3) 4)
(((f3 1 2) 3) 4)
((((f4 1) 2) 3) 4)
```

`f1, f2, f3` and `f4` all produce the same result, but they differ in the number of arguments they need to be invoked with and the level of nesting of the returned function, forcing us to "unroll" the arguments one by one. If we were to be constrained by the language to only have functions of a single argument, then we would have to simulate functions of multiple arguments like `f4` does. Luckily for us, such a constraint doesn't exist in Clojure (or any other mainstream language) but as we have seen in the examples for `partial` in this chapter, sometimes it's handy to suspend part of a function until more arguments are available. So in Clojure we won't define `f2, f3` or `f4` explicitly and prefer `partial` instead:

```
((partial f1 1 2 3) 4)
((partial f1 1 2) 3 4)
((partial f1 1) 2 3 4)
```

The approach Clojure is taking here is to just use higher order functions to remove all the nesting that would be necessary otherwise. Other languages go beyond partial application supporting currying at the compiler level. In Haskell, for example, all functions called with less than the declared number of arguments are automatically turned in their curried form:

```
f = (+ 1) ❶
-- is equivalent to:
f x = x + 1
```

❶ `f` is a function of 1 argument incrementing its input by 1. Instead of creating our own implementation we used the already existing function `+` and "curried" the first argument to be always 1.

In the Haskell example above, the compiler accepts that plus is called with less than the allowed number of arguments to produce a function `f` that accepts the remaining one. No need to invoke an explicit `partial` function like in the Clojure

version as well as no need to have explicit multiple artities in Haskell.

See also

- fn (or the equivalent reader macro `#()`) can be used instead of `partial` when the argument we want to suspend is not coming last in the argument list. fn is effectively a general `partial` that let us wrap the target function the way we need. For all other cases `partial` could be clearer in conveying the idea that a function is waiting for more arguments. In general the two forms are a matter of taste.

Performance considerations and implementation details

=> `O(1)` **function generation**

As other function in this section of the book, `partial` is not an usual target for performance analysis or optimization as the generation of the function itself is constant time.

The number of arguments called on the generated function has a small but visible performance impact. `partial` provides some special arities to the generated function to deal with 1, 2 or 3 arguments calls. Beyond 3 arguments `partial`

implements a catch-all variable arguments call that results in a small speed decrease:

```
(require '[criterium.core :refer [bench]])

(let [myvec (partial vector :start)] (bench (myv
;; Execution time mean : 9.466062 ns
(let [myvec (partial vector :start)] (bench (myv
;; Execution time mean : 214.156293 ns
(let [myvec (fn [a b c d] (vector :start a b c d
;; Execution time mean : 8.156293 ns
```

❶ `myvec` is a small function that always insert ":start" at the beginning of a vector. We call `myvec` with 3 arguments first. vector also has several optmized arities and we try to take advantage of this fact.

❷The second benchmark adds an additional argument while calling `myvec` resulting in a visible performance impact despite the fact that vector has a specific `arity` for a 5 arguments call (the threshold before switching to the catch-all variable `arity` is 6 arguments for vector).

❸As a comparison, let's have a look at a similar solution using an explicit lambda function.

The reader should remember that in real-life scenarios micro-benchmarks like the ones we are performing are influenced by many other factors and the described speed impact is tiny in absolute terms. But if you happen to use `partial` in a tight loop with more than 4 arguments, you should probably look into using fn instead.

2.2.9 every-pred and some-fn

`function` `since 1.3`

Listing 2.14. → *Function generator, boolean logic*

```
(every-pred
  ([p])
  ([p1 p2])
  ([p1 p2 p3])
  ([p1 p2 p3 & ps]))

(some-fn
  ([p])
  ([p1 p2])
  ([p1 p2 p3])
  ([p1 p2 p3 & ps]))
```

`every-pred` and `some-fn` take one or more predicates and produces a new function. The returned function takes zero or more arguments and invokes all predicates against all arguments combining the results with the equivalent of an and or or operation, respectively.

Both `every-pred` and `some-fn` applies short-circuiting behavior to the combination of predicates, stopping evaluating at the first predicate that returns:

- Either `nil` or `false` for `every-pred`.
- A truthy value in case of `some-fn`.

Contract

input

- "p", "p1", "p2" and "p3" must be functions supporting a one argument call and returning any type. Even when a predicate returns something different from a boolean type, the returned value is evaluated as a `true` or `false` following Clojure conventions.
- "ps" is the list of all remaining predicates after the third.

notable exceptions

- `ArityException` when `every-pred` or `some-fn` is invoked without arguments, or when any predicate requires more than 1 argument.

output

- `every-pred`: returns a function of any number of arguments of any type that returns either `true` or `false`.

- `some-fn`: returns a function of any number of arguments of any type that returns any type. The return value is usually interpreted using extended boolean logic.

Examples

`every-pred` and `some-fn` are often found when a sufficiently large combination of predicates needs to be applied to one or more values. For example it might be natural to think that this is possible and works correctly:

```
(remove (and number? pos? odd?) (range 10)) ; ❶
;; (0 2 4 6 8)
```

❶A tricky expression that seems to work correctly when it's not.

What we would like to achieve is to combine a set of predicates using and so each element from the input collection is subject is accepted or not by remove. But and is a macro and its evaluation happens while the form is compiling, resulting in the following being evaluated:

```
(remove odd? (range 10)) ; ❶
```

❶How the form from the example before appears after it is compiled by the Clojure compiler.

The and expression returns the last value that is not `false` or `nil`, which is the function `odd?` in this case. What we really want is to combine the predicates in such a way that they are evaluated on each item, so we could do something like this:

```
(remove #(and (number? %) (pos? %) (odd? %)) (ra
;; (0 2 4 6 8)
```

❶The correct way to combine multiple predicates require to add a wrapping anonymous function and repeating the argument for each predicate.

The correct way to combine multiple predicates doesn't read as well as before after adding a wrapping anonymous function and repeated the argument for each predicate. The situation becomes worse the more predicates need to be combined. For example, here's how we could go to find palindromes (words that read the same both directions) in a collection of items:

```
(defn symmetric? [xs] ; ❶
  (= (seq xs) (reverse xs)))
```

```
(defn palindromes [coll] ; ❷
  (filter
    (fn [word]
      (and
        (some? word)
        (string? word)
        (not-empty word)
        (symmetric? word)))
    coll))

(palindromes ["a" nil :abba 1 "" "racecar" "abba
;; ("a" "racecar" "abba")
```

❶ `symmetric?` verifies if the sequential collection "xs" is equal to its reverse. This is a good (although not efficient) definition of palindromic sequence.

❷ `palindromes` applies a series of steps to establish if a word in a collection is a palindrome: first it should not be `nil`, second it should be a string, third it should not be empty and finally, it should be equal to its reverse.

The function used to filter palindromes, contains a concatenation of predicates with the argument "word" repeated 4 times. By using `every-pred` we can remove the anonymous function, the need to pass the "word" argument and the need to use and:

```
(defn symmetric? [xs]
  (= (seq xs) (reverse xs)))

(defn palindromes [coll] ; ❶
  (filter
    (every-pred some? string? not-empty symmetri
    coll))

(palindromes ["a" nil :abba 1 "" "racecar" "abba
;; ("a" "racecar" "abba")
```

❶ `palindromes` is significantly easier to read after introducing `every-pred`.

The example shows `every-pred` used in-place, without giving a name to the generated function. We could also assign `every-pred` a name and reuse the same combination of predicates in different places. The following example extracts the check for palindromes into a new `palindrome?` function:

```
(defn symmetric? [xs]
  (= (seq xs) (reverse xs)))

(def palindrome? ; ❶
  (every-pred some? string? not-empty symmetric?
```

```
(defn palindromes [coll] ; ❷
  (filter palindrome?  coll))

(palindromes ["a" nil :abba 1 "" "racecar" "abba
;; ("a" "racecar" "abba")
```

❶The combination of predicates to check for palindromic words is now available for reuse in other parts of the code.

❷The `palindromes` function is now just invoking filter on the collection.

Let's now have a look at `some-fn`. In the following example, we are going to perform multiple checks on a value to determine if an email message is spam:

```
(defn any-unwanted-word? [words] ; ❶
  (some #{"free" "sexy" "click"} words))

(defn any-link? [words] ; ❷
  (some #(re-find #"http[s]?://.*\." %) words))

(defn any-blacklisted-sender? [words] ; ❸
  (some #{"spamz@email.com" "phish@now.co.uk"} w

(def spam? ; ❹
  (some-fn any-unwanted-word? any-link? any-blac
```

```
(defn words [email] ; ❺
  (clojure.string/split email #"\s+"))

(spam? (words "from: alex@tiv.com just wanted to
;; nil

(spam? (words "from: nobody@all.tw, click here f
;; "click"
```

❶ `any-unwanted-word?` contains a Set of unwanted words and we want to know if any of them is present in a message. We use some applied to the sequence of words using the set itself as a function. If the word is in the set, the word itself is returned. `some` returns the first occurrence of the word in the set or `nil` otherwise.

❷ The second function applies a regular expression to verify if any of the word is a link to external content on the web. In this simplified example we consider every email with a link as suspicious.

❸ `any-blacklisted-sender?` checks the content of the email against a collection of email addresses that we consider spam. `any-blacklisted-sender?` works exactly as `any-unwanted-word?`.

❹ We combine all the functions using `some-fn` in a new function returning the first positive check (logical `true`). We also opted to def the generated function as `spam?` to promote further reuse.

❺ `words` splits a string using "clojure.string/split".

❻ We can see an example of "clean" email with no matching words. `some-fn` eventually returns `nil` after calling the chain of predicate functions.

❼ The last call discovers a blacklisted word.

`some-fn` can be used to retrieve the result from a matching predicate (like in the example above) or in conditional statements, where we have an option to completely ignore the precise value returned. With reference to the previous example, we can use when-let to combine conditional logic and value matching:

```
  (when-let [match (spam? (words "from: nobody@all
    (throw (Exception. (str "Spam found: " match))
  ;; Exception Spam found: click
```

❶ We can use `some-fn` in conditional logic as well as to retrieve a specific value. In this case, when-let both assigns the matching value to a local binding "match" as well as entering the condition and throwing an exception.

`some-fn` and `every-pred`: what's up with names?

Someone at Netscape once said that "there are only two hard things in Computer Science: cache invalidation and naming things" [32]. As many other languages, Lisp-derived ones had to put a great deal of attention in getting names right, also considering that the absence of types reduces the surrounding context and provides less hints about what a function does. In general, conveying the full semantic of a function in just one or two words is a complex problem.

You generally note meaningful names in the standard library, but there are cases which generate recurring debates in public forums. `some-fn` and `every-pred` are

one of those examples. You may ask why `some-fn` is not called `some-pred`? Or similarly: why `every-pred` is not called `every-fn`?

Let's consider the definition of "predicate": strictly speaking a predicate is a function that returns a boolean type. `some-fn`, as we can also see in the examples, is returning the first matching value. The function generated by `every-pred` is instead returning a strict `true` or `false` value, hence the "pred" suffix in the name.

There is also an additional consideration that justifies the lack of interest in a potential `every-fn` function. Let's try to model our own `every-fn` to understand why. The goal of `every-fn` would be to behave like `every-pred` but returning matches instead of true/false:

```
(defn every-fn [& ps] ; ❶
  (fn [& xs]
    (partition (count ps)
      (for [x xs p ps] (p x)))))

(def contains-two? #(re-find #"two" %)) ; ❷
(def is-7-long? #(= 7 (count %)))

((every-fn contains-two? is-7-long?) "guestimate" "artwork" "thre
;; ((nil false) ("two" true) (nil false))

((every-pred contains-two? is-7-long?) "guestimate" "artwork" "th
;; false
```

❶ `every-fn` has the same interface as `every-pred`: takes a list of functions (supposedly predicates) and return another function that calls all predicates in "ps" on each of the items in the input. Our implementation generates permutations for all predicates "ps" against each input and group them together with partition.

❷ The example provides two simple predicates (an extended definition that includes logical true and logical false.

❸ We can now call `every-fn` using a sample input. The result shows returned values from the predicates grouped for each input.

❹ By comparison, `every-pred` would look at each value and stops at the first occurrence of a `nil` or `false` returning `false`.

`every-fn` is designed to return the explicit result of applying each predicate to each input. Without further processing, we can't see the same useful information that `every-pred` is returning, such as if there is at least one input that doesn't satisfy all predicates at once. We can still use the results in some other way, but at that point we can solve the same problem without the need of creating custom functions:

```
(map (juxt contains-two? is-7-long?) (vector "gu
;; ([nil false] ["two" true] [nil false])
```

❶ The effect of `every-fn` are produced without using a custom function. juxt applies multiple functions to the same input and it looks a good choice in this case.

See also

- every? is used by `every-pred` implementation. If you have a single predicate to apply to multiple arguments, prefer "every?" instead of `every-pred`.
- some-fn is very similar to `every-pred` except that instead of verifying if all predicates are `true`, it verifies if at least one is `true` (equivalent to the boolean "or" operator).
- some applies a similar logic to `some-fn`. But instead of composing together multiple predicates it uses a single predicate against each element of the sequence. Use `some` instead of `some-fn` if you are only interested in a single predicate applied to multiple values.

Performance considerations and implementation details

→ `O(1)` **function generation**

→ `O(n)` **generated function** (n number of arguments, worst case)

The performance consideration in this section are equally valid for `every-pred` and `some-fn` unless otherwise stated.

`every-pred` and `some-fn` generate functions in constant time with a small performance penalty for 4 or more

arguments. The reader should consider these aspects in case `every-pred` or `some-fn` appears as part of processing large collections or in a fast loops. The following benchmark illustrates the point:

```
(require '[criterium.core :refer [quick-bench]])

(quick-bench (every-pred 1)) ; ❶
(quick-bench (every-pred 1 2))
(quick-bench (every-pred 1 2 3))
(quick-bench (every-pred 1 2 3 4))
(quick-bench (every-pred 1 2 3 4 5))

;; Execution time mean : 4.072343 ns
;; Execution time mean : 4.276523 ns
;; Execution time mean : 4.333870 ns
;; Execution time mean : 18.539700 ns
;; Execution time mean : 22.359127 ns
```

❶We compare creating functions with `every-pred` starting from 1 argument up to 5.

Please note that the difference between 3 and 4 arguments is small and there could be other kind of dominating computations to consider. A straightforward solution to eliminate the generation time from the computation is to name

the generated function as a let binding or a var in the namespace. If we look at the performance of the generated function we can see a similar behavior:

```
(require '[criterium.core :refer [quick-bench]])

(defn p [x] true) ; ❶

(let [e1 (every-pred p) ; ❷
      e2 (every-pred p p)
      e3 (every-pred p p p)
      e4 (every-pred p p p p)
      e5 (every-pred p p p p p)]
  (quick-bench (e1 1))
  (quick-bench (e2 1))
  (quick-bench (e3 1))
  (quick-bench (e4 1))
  (quick-bench (e5 1)))

;; Execution time mean : 4.403670 ns
;; Execution time mean : 4.577825 ns
;; Execution time mean : 3.701983 ns
;; Execution time mean : 112.987134 ns
;; Execution time mean : 133.402111 ns
```

❶A dummy predicate `p` always returning `true`. This is the worst case scenario for `every-pred` that doesn't have a way to short-circuit the computation.
❷We generate a few functions with `every-pred` passing a different number of predicates.

The difference between 3 and 4 arguments depends on the fact that also the generated function is optimized for up to 3 arguments. In general, the more arguments evaluated `true` by the predicates, the higher the time required for the computation, as the generated function needs to find a `false` value to return immediately. Hence the linear behavior in the worst case scenario.

[27] This is the original use case for `fnil` as documented by this thread in the Clojure mailing list: <groups.google.com/d/msg/clojure/mcxKa_5mWm4/CkSrutnPUfIJ, https://groups.google.com/d/msg/clojure/mcxKa_5mWm4/CkSrutnPUfIJ>
[28] There is already an improved version of fnil proposed in this patch ready to be added to Clojure core.
[29] This style of composition is also called point-free style
[30] Please see the official Clojure documentation about direct linking here: clojure.org/reference/compilation#directlinking

[31] A Wikipedia article introducing the identity function concept en.wikipedia.org/wiki/Identity_function

[32] This quote is attributed to Phil Karlton, once Netscape architect. See: skeptics.stackexchange.com/questions/19836/has-phil-karlton-ever-said-there-are-only-two-hard-things-in-computer-science

2.3 THREADING MACROS

2.3.1 ->

`macro` `since 1.0`

Listing 2.15. → *functional composition, threading macro, function application*

```
  (-> [x & forms])
```

`->` (also known as thread first macro or thrush operator) can be used to compose or group together a list of operations. The arguments of the `->` macro consist of an expression (mandatory) and a list of forms (optional).

The idea is to position the expression as the first argument of the following forms. For example here's a step-by-step explanation of what happens when evaluating `(-> {:a 2} :a`

`inc)`. Although this is not strictly how the macro is implemented, it is a good thinking model:

1. The keyword `:a` is the first optional form after the initial expression `{:a 2}`. Internally the form is added to a list (unless it is already). Since `:a` it's not already a list, it is transformed into `(:a)`.
2. The expression `{:a 2}` is placed as the second item in the previously created list resulting in `(:a {:a 2})`. The resulting form is evaluated and passed downstream. In this case `(:a { :a 2})` equals 2 and 2 is passed down to the next form.
3. inc is the next item in the list of forms. As before it's not a list, so it needs to be transformed into `(inc)`.
4. The previous result, 2, is then placed as the second item in the previous list: `(inc 2)`.
5. We finally reached the end. The final form is evaluated and returned.

During macro-expansion, the Clojure compiler transforms `(-> {:a 2} :a inc)` in `(inc (:a {:a 2}))`. During evaluation the form evaluates to the number 3. macroexpand confirms our theory:

```
(macroexpand '(-> {:a 2} :a inc))
;; (inc (:a {:a 2}))
```

`->` tends to improve the readability of some class of sequential operations that would otherwise read backward (or inner-most to outer-most). Transformation pipelines (where the result of a first operation needs to be passed down to the next operation) are usually a good candidate to be "threaded" using `->`.

Contract

input

- "x" can be any valid Clojure expression. It can be useful to remember "x" as the "x" in "eXpression", which is what `→` threads through the following "forms".
- "forms" is an optional list of arguments. If any of the optional forms is not a list already, it will be made so by invoking list on it. The first element of each form must be a callable function (such that `(ifn? (first form))` evaluates to `true`).

notable exceptions

- `ArityException` if called with no arguments.

- `ClassCastException` if any form is not callable, for example `(→ 1 2 [])`.

output

- `->` produces the evaluation of the last form, using the result of the previously evaluated forms, following the thread-first rules exposed above. If no forms are provided, it returns the evaluation of the first argument "x".

Examples

`->` is particularly useful for processing pipelines where an initial input is transformed at each step. This is true for the common case of map processing. The following example shows a way to parse an HTTP request into a hash-map:

```
(def req {:host "http://mysite.com" ; ❶
          :path "/a/123"
          :x "15.1"
          :y "84.2"
          :trace [:received]
          :x-forward-to "AFG45HD32BCC"})

(defn prepare [req] ; ❷
  (update
```

```
      (dissoc
        (assoc req
          :url (str (:host req) (:path req))
          :coord [(Double/valueOf (:x req)) (Doubl
        :x-forward-to :x :y)
      :trace conj :prepared))

  (pprint (prepare req)) ; ❸
  ;; {:host "http://mysite.com",
  ;;  :path "/a/123",
  ;;  :trace [:received :prepared]
  ;;  :url "http://mysite.com/a/123",
  ;;  :coord [15.1 84.2]}
```

❶ `req` is an example request. Some web framework is taking care of transforming the request into a map for us.

❷ `prepare` takes the request and assoc a few additional keys. It then removes keys that are no longer needed and finally updates the `:trace`.

❸We can use `clojure.pprint/pprint` to better format the output. `pprint` is available at the REPL directly, but requires an explicit require otherwise.

To prepare the request above, we need a few transformations: join the host and path together to form the `:url`,create a vector out of the coordinates, remove the coordinates and the

forward header and finally, update the trace to include the preparation step was done. In a real life application, request processing could be arbitrarily long and complicated. We could take advantage of `->` to increase the readability of the transformation:

```
(defn prepare [req] ; ❶
  (-> req
      (assoc :url (str (:host req) (:path req))
             :coord [(Double/valueOf (:x req)) (
      (dissoc :x-forward-to :x :y)
      (update :trace conj :prepared)))
```

❶The `prepare` function has been refactored to take advantage of `->`.

Introducing `->` in the `prepare` function creates a visual top-to-bottom flow that is easier to read: the `req` input is passed "down to" the first assoc operation, then dissoc and finally update.

Another interesting use of `->` is in conjunction with the anonymous lambda form `#()`. When applied to a single argument, `->` behaves similarly the identity function so `(-> 1)` is equivalent to `(identity 1)`. To understand how this

could be useful, let's take a look at the following failing example:

```
(def items [:a :a :b :c :d :d :e])

(map #({:count 1 :item %}) items) ; ❶
;; ArityException Wrong number of args (0) passe
```

❶ We'd like create a map out of each element in `items`, but this is not the right way to do it.

What we would like to achieve in the example above, is to create a map containing a key `:count` which is always 1 and a key `:item` which is the original element from the `items` vector. The problem with the above is that the anonymous function `#({:count 1 :item %})` is trying to invoke the map with no arguments. We need to macroexpand the form to see what's happening:

```
(macroexpand '#({:count 1 :item %})) ; ❶
;; (fn* [p1] ({:count 1, :item p1}))
```

❶ macroexpand is an useful debugging tool to visualize macro transformations.

macroexpand shows the attempt of calling an array-map as a function without arguments (which fails). There are now several options to fix the problem. The following forms are all producing the expected results:

```
(map #(hash-map :count 1 :item %) items) ; ❶
(map #(identity {:count 1 :item %}) items) ; ❷
(map #(do {:count 1 :item %}) items) ; ❸
(map #(-> {:count 1 :item %}) items) ; ❹
```

❶A set of 4 forms all producing the same result on the same vector of `items`. The first uses hash-map, which is an idiomatic choice.

❷We could use identity and keep using the map literal syntax with curly braces `{}` but the need for identity is hard to understand.

❸A shorter alternative to identity is do. However, do presence is often associated with side-effects which are nowhere in this form. Overall, this option is confusing like the second one.

❹The final form uses `->` and is short and to the point.

The last option making use of `->` conveys information about the operation effectively: it's short and easy to read, without introducing the semantic cluttering of identity or do. Both the

option using hash-map and `->` are idiomatic, but they produce slightly different results:

```
(map type (map #(hash-map :count 1 :item %) [1]))
;; (clojure.lang.PersistentHashMap)
(map type (map #(-> {:count 1 :item %}) [1])) ;
;; (clojure.lang.PersistentArrayMap)
```

❶Using hash-map to create a map produces the `clojure.lang.PersistentHashMap` type.
❷Using the map literal syntax `{}` with a minimal set of keys produces a `clojue.lang.PersistentArrayMap` type.

Please refer to array-map and hash-map to understand this type difference. Most of the time, Clojure handles the transition from one map type to another transparently without the user needing to know.

Thread macro and the T-combinator

Combinatory logic is a notation (like the lambda calculus) that eliminates the need for free variables in mathematical logic [33]. It attracted attention in programming, since combinators can be useful in functional composition to increase expressive power. The T-combinator in particular, allows for a function application to happen "in reverse". The Clojure thread operator produces similar effects to the T-combinator but as a macro, it is limited in its application:

```
(/ (Math/abs (- (* (inc 1) 5) 1)) 3) ; ❶
(-> 1     ; ❷
  inc
  (* 5)
  (- 1)
  (Math/abs)
  (/ 3))
```

❶A simple mathematical expression written as nested function application
❷The same expression written vertically using the `->` macro.

The two expressions in the example above return the same result but the second exposes the flow much clearly. However, `->` as T-combinator is limited by the fact that it does not support nested function with arguments, for example:

```
(-> 1 (fn [x] (inc x)))
;; IllegalArgumentException Parameter declaratio
```

The above results in a compile error. The macroexpansion clearly shows what's wrong:

```
(macroexpand-1 '(-> 1 (fn [x] (inc x))))
;; (fn 1 [x] (inc x))
```

This is why sometimes the thread operator in Clojure is compared to a limited T-combinator [34].

See also

`->` is just one of the several flavors of thread macros offered by Clojure. Initially it was only `->`, followed by `->>` in 1.1 and a bigger expansion with the 1.5 release of Clojure which added `as->`, `some->`, `some->>`, `cond->` and `cond->>`. The other related threading macros are:

- ->> is called "thread last" macro and is very similar to `->` but it puts the element at the end of the next form instead of as the second element. It is particularly useful for sequence processing, where the input sequence usually appears last in the list of arguments.
- "as->" enables the selection of a placeholder making explicit where the element is placed in the next form. Use `as->` when it's necessary to fine control the placement of the element in the next form. This thread macro has the drawback to be more verbose, because the placeholder is repeated in each form.
- some-> takes care of any initial or intermediate `nil` value, stopping right away instead of passing it to the next form. `some->` is useful when a form evaluates to `nil` causing exception.

- cond-> enables a custom condition to decide if the processing should continue or not. This is the only thread macro allowing to skip a step completely.
- get-in fetches the value for an arbitrarily nested associative data structure such as a Clojure map. For example: `(-> {:a 1 :b {:c "c"}} :b :c)` is equivalent to `(get-in {:a 1 :b {:c "c"}} [:b :c])`. Consider using `get-in` instead of `->` if you need to access values in a deeply nested map.

Performance considerations and implementation details

→ `O(n)` **n number of forms (expansion time)**

`->` macro expansion is not relevant for run-time performance analysis, as the cost of the expression applies just to compilation. At compile time, reversing the order of application of the function arguments is linear in the number of forms.

2.3.2 ->>

`macro` `since 1.1`

Listing 2.16. → *functional composition, threading macro, functional application*

```
(->> [x & forms])
```

`->>` (also known as thread last macro) can be used to compose or group together a list of operations by positioning the first expression as the last argument of the following form (similarly to "->" which places it first instead). `->>` tends to improve the readability of some class of sequential operations that would otherwise read backward (or inner-most to outer-most). Transformation pipelines (where the result of a first operation needs to be passed down to the next operation) are usually a good candidate to be "threaded" using `->>`.

The arguments of the `->>` macro consist of an expression (mandatory) and a list of forms (optional). The idea is that the first expression is "piped through" the other forms that get a chance to process the expression at each step before the final output is returned.

Contract

input

- "x" is a mandatory expression. The expression is evaluated and placed last in the following form (if any).
- "forms" is an optional list of forms. If any form is not a list already, it is wrapped by a list. The first element of each form must be a callable object (such that `(ifn? (first form))` is `true`). Each evaluated form is placed last in the following and then evaluated until there are no more forms.

notable exceptions

- `ArityException` if called with no arguments.
- `ClassCastException` if any form is not callable. For example in `(->> "a" "b" [])` the string "a" is treated as a function.

return

- returns: the result of evaluating the last form (if any), by placing the previously evaluated form as the last argument of the next. If no forms are provided, it returns the evaluation of the expression "x".

Examples

`->>` is well suited for sequential processing pipelines where an initial input is transformed by each step into the final output. The following example shows how we could rewrite the nesting of several filter operations using `->>`. We want to filter all even positive numbers divisible by 3 made by the same repeated digit:

```
(filter pos? ; ❶
  (filter #(apply = (str %))
    (filter #(zero? (mod % 3))
      (filter even? (range 1000)))))
;; (6 66 222 444 666 888)

(->> (range 1000) ; ❷
     (filter even?)
     (filter #(zero? (mod % 3)))
     (filter #(apply = (str %)))
     (filter pos?))
;; (6 66 222 444 666 888)
```

❶This set of nested filters is reasonably easy to follow, but we still need the mental effort of searching for the inner-most form and move outward to understand it.

❷ `->>` inverts the previous flow starting with the input sequence first, then the set of operation in the order they are actually applied.

The following example illustrates how flexible `->>` is, for example when different sequence operations are involved, like the case of parsing the query string of a web request. Here's the first option that nests each step processing step without using the threading last macro:

```
(require '[clojure.string :refer [split]])
(def sample-query "guidx=123&flip=true")

(defn params [query]                ; ❶
  (apply merge                      ; ❷
    (map #(apply hash-map %)        ; ❸
      (map #(split % #"=")          ; ❹
        (split query #"&")))))      ; ❺

(params sample-query)
;; {"guidx" "123", "flip" "true"}
```

❶ `params` is a function that applies a series of transformation to a string input.

❷This merge operation is the last executed, but it appears first.

❸In this step we take each pair of splits and build a map with them.

❹This step splits each string where the "=" sign appears.

❺The initial string input is split by the "&" sign.

The function that process the parameters is not easy to follow, as it reads backward. Here's the new version of `params` that takes advantage of `->>`:

```
(defn params [query] ; ❶
  (->> (split query #"&")
       (map #(split % #"="))
       (map #(apply hash-map %))
       (apply merge)))
```

❶The rewrite of the `params` function to take advantage of the thread last macro.

The new `params` function contains exactly the same number of steps as before, just arranged in a different order. Worth noticing how the same operations naturally flow in order this time, so we can start reading about the first operation from the top and follow the vertical flow to the bottom.

See also

`->>` is one of the most used and generic of the thread-last macros. There are other thread-last variations that interact more specifically with the processing pipeline:

- some->> is a nil-aware thread last macro that stops processing at the first `nil` occurrence in the evaluation chain.
- cond->> allows the presence of a condition each step to decide if to continue or not. This version of the thread last macro allows to skip one or more steps completely.

Performance considerations and implementation details

→ `O(n)` **n numbers of forms (macro expansion)**

There are no special performance considerations to do before using the `->>` macro. Reversing the order of application of the function arguments is linear in the number of functions but this happens at macro expansion time and in general the list is never more than a few items long. In general all the threading macro are not considered particularly important during performance analysis.

2.3.3 cond-> and cond->>

`macro` `since 1.5`

Listing 2.17. → *functional composition, threading macro, conditional functional application*

```
(cond-> [expr & clauses])
(cond->> [expr & clauses])
```

`cond->` and `cond->>` are a specialized version of the basic threading macros `->` and `->>` respectively. `cond->` takes an expression and "threads first" the expression into the following form (as the first argument of the formed function) if and only if a clause is `true`. Similarly `cond->> "threads last" the expression when the condition is `true`. Each form is preceded by a clause that is used to decide if the previous evaluation should go through the form or not.

One important fact about conditional threading macros is that they are **not short-circuiting**. If the clause is `false`, the related form is simply skipped and computation resumes from the next. Also worth noticing that the clauses don’t have access to the evaluation of other forms but just the surrounding local bindings, like any other non-macro evaluated part of the code. Specifically, the clause can’t reference the result of the previous form (what is "threaded-through"). This behavior can be used to repeatedly check an initial expression (or some other given

option) independently from the transformations that are happening before.

Here's a step-by-step explanation to clarify `(cond->)` logic:

```
(let [x \c]              ; ❶
  (cond-> x              ; ❷
    (char? x) int        ; ❸
    (char? x) inc        ; ❹
    (string? x) reverse ; ❺
    (= \c x) (/ 2)))    ; ❻

;; 50
```

❶A local binding "x" is established as the character `\c`.

❷"x" is threaded through `cond->`.

❸The clause `(char? x)` is evaluated. Since "x" is a character type, then the related form evaluates. Since the int function is not a sequence, it is transformed in a list with `(list int)` at macro expansion. Then "x" is used as the second argument of `(int x)` which is evaluated to the number 99 (the ascii equivalent of `\c`).

❹The clause `(char? x)` is evaluated again and it's still "true" as it is bound to "x" which didn't change. But this time the result of the previous form is used as the argument to inc resulting in the number 100.

❺The clause `(string? x)` is evaluated. x is not a string (it is a character) so the reverse doesn't take place.

❻`(= \c x)` is `true`, so the following form is evaluated. The result of the previous evaluated form (100) is used as the second argument to the form as `(/ 100 2)` resulting in the number 50.

Similarly, here's a step by step example of how `cond->>` operates on its argument:

```
(let [x [\a 1 2 3 nil 5]]                ; ❶
  (cond->> x                             ; ❷
    (char? (first x)) rest               ; ❸
```

```
      true (remove nil?)                    ; ❹
      (> (count x) 5) (reduce +)))          ; ❺

 ;; 11
```

❶A local binding "x" is established for the vector `[\a 1 2 3 nil 5]`.

❷"x" is threaded through the `cond->>` macro.

❸The clause `(char? (first x))` is evaluated. Since `\a` is a character type, then the form is evaluated. Since the rest function is not a sequence, it is transformed in a list with `(list rest)` internally. x is used as the second argument of `(rest x)` which is evaluated to the list `(1 2 3 nil 5)`.

❹When `true` is used as a clause, the form is always evaluated. The previous evaluation is added as the last argument of `(remove nil?)` resulting in `(remove nil? (1 2 3 nil 5))` which evaluates to the new list `(1 2 3 5)`.

❺The last clause `(> (count x) 5)` counts the elements in "x" (this is the original expression, not the previously evaluated list). Since there are more than 5 items, the previously evaluated list is used as the last argument of the current form: `(reduce + (1 2 3 5))`. The final result is 11.

Contract

input

- "expr" is a mandatory expression. "expr" is evaluated and placed second in the following form if the clause condition evaluates to `true`.
- "clauses" is an optional list of clause-form pairs. In each pair, a "clause" is an expression that evaluates logical true/false. A "form" must be present for each clause. If the form is not a list already, a new wrapping list is created. The first element of each form must be an callable object (so that `(ifn? (first form))` is `true`).

output

- The result of evaluating the last form (if provided), using the result of the previously evaluated form. If no forms are provided (or all conditions evaluate to `false`), it returns the evaluation of "expr".

Examples

One idiomatic use of `cond->` is in conditional forms where the "true" branch should transform the input while the "false" branch leaves it untouched. For example the following forms are equivalent:

```
(let [x "123"] (if (string? x) (Integer. x) x))
(let [x "123"] (cond-> x (string? x) Integer.))
```

❶The variable "x" can be a string or a number. If it's a string, we want to convert it to a number, otherwise we don't do anything. The conditional form needs to repeat "x" one more time at the end, just to leave it as it is.

❷In the `cond->` version, we avoid repeating "x" a third time, as it is threaded through `Integer` constructor only when the condition is `true`.

`cond->` can be used to process heterogeneous data so they appear eventually under the same "shape". This situation can happen for example when an application receives XML or JSON for the same entity but there are small differences in the structure or values (tree-like data structures can be directly represented and processed as hash-maps in Clojure). The following `shape-up` function checks if the incoming hash-map conforms to a set of rules and changes it accordingly:

```
(defn same-initial? [m]
  (apply = (map (comp first name) (keys m))))

(defn shape-up [m]
```

```
    (cond-> m
      :always (assoc-in [:k3 :j1] "default")
      (same-initial? m) (assoc :same true)
      (map? (:k2 m)) (assoc :k2 (apply str (vals (

  (map shape-up
    [{:k1 "k1" :k2 {:h1 "h1" :h2 "h2"} :k3 {:j2 "j
     {:k1 "k1" :k2 "k2"}
     {:k1 "k1" :k2 {:h1 "h1" :h3 "h3"} :k3 {:j1 "j

  ; ({:k1 "k1", :k2 "h1h2", :k3 {:j2 "j2", :j1 "de
  ;  {:k1 "k1", :k2 "k2", :k3 {:j1 "default"}, :sa
  ;  {:k1 "k1", :k2 "h1h3", :k3 {:j1 "default"}, :
```

❶The first form enforces the presence of a `:k3` key pointing at the map `{:j1 "default"}` (and if the key already exists it will be replaced). After this first step we are sure that `:k3` `:j1` key combination exists possibly with a "default" value. To enforce that the condition should always be applied, ":always" was used as the clause. ":always" is logical `true`, so the related expression is always evaluated.

❷The second clause checks if all the keys are starting with the same letter. If that's the case, we add a key `:same` true.

❸In the last step if value for key `:k2` is another hash-map then we take all the values of that inner hash-map and concatenate them together as a string. We finally replace the same `:k2` key with the new string.

An interesting `cond->>` usage is similar to the non-conditional `->>` thread-last macro but with optional steps. Sequences are a good fit for pipeline-like processing because they take their input data at the end. `cond->>` ads the conditionals for enhanced flexibility. Here's for example a function for a hypothetical audio signal processing application:

```
(def signals
  [111 214 311 413
   107 221 316 421
   112 222 317 471
```

```
    115 223 308 482])

(defn process [signals opts]
  (let [{:keys [boost? bypass? interpolate? nois
    (cond->> signals
        (< (count signals) 10) (map inc)
        interpolate? (mapcat range)
        bypass? (filter bypass?)
        noise? (random-sample noise?)
        cutoff? (take-while #(< % cutoff?)))))

(process signals {:bypass? even? :interpolate? t
;; (0 4 12 14 16 ...
```

❶Using destructuring we can extract the relevant keys from the input map.

❷Signal processing starts by checking how many sampling events we received. If less than some amount, each signal gets incremented. In our example this operation is not evaluated.

❸To simulate the introduction of new data (interpolation) range is invoked on each signal in the sequence, generating a list of nested sequences of different sizes. mapcat takes care of joining everything back together. In our example, the option was set in the map and the interpolation takes place on the original signal list, because the previous step wasn't executed.

❹This step filters the signals based on the `bypass?` key. If `bypass?` is `nil`, then there is no filtering. When `bypass?` contains something other than `nil`, it assumes `bypass?` is the predicate for the filter. The filter operation takes place in our example using `even?` as predicate.

❺This step optionally adds noise to the signal by random sampling the list using random-sample. Since the `noise?` key is set, also this step takes place using a 50% (0.5) probability.

❻Finally the cutoff step removes all signals above a certain threshold. The step gets executed with a 200 threshold.

Visualize the vertical flow

Clojure functions and macros can be very rich in functionality. The more they are expressive and rich, the more documentation is required to understand them. This is

the case for thread macros like `cond->` and `cond->>`. Luckily for us, their natural vertical flow helps better understanding them.

In the case of `cond->>`, with the help of some formatting, it's easier to see two columns that flow vertically during execution: one for conditions (on the left), the other for forms (on the right). This is the same example used in the introduction:

```
(let [x [\a 1 2 3 nil 5]]
  (cond->> x
    (char? (first x))     rest
    true                  (remove nil?)
    (> (count x) 5)       (reduce +)))
```

The **condition column** has access to local and global bindings (as any other part of the code) but it doesn't know anything about the right-hand column. Similarly, the **processing column** on the right can't have any impact on the conditions on the left (assuming no side-effects).

Please keep in mind that the above indentation style is used here to emphasize the vertical flow of `cond->>` and is not normally used. Consider emphasis through indentation (or columns) a special case of documentation to be used sparingly. When in doubt on how to use the correct indentation style for a function or a macro, the user contributed clojure style guide is the definitive reference on the subject [35].

See also

- "->" is the "thread-first" macro. Differently from `cond->` it doesn't apply any clause for the execution of the next form.
- ->> is the "thread last" macro. Differently from `cond->>` it doesn't check a condition for the execution of the next form.
- "some->" can be roughly compared to a `cond->` where all conditions are only checking for `nil`. However, "some->" short-circuits and returns right away in case of `nil`.
- some->> roughly compares to `cond->>` where all conditions check for `nil`. However, some->> short-circuits and returns right away if some of the forms evaluates to `nil`.

Performance considerations and implementation details

→ `O(n)` **n number of forms (compile time)**

`cond->` and `cond->>` are not usually relevant for run-time performance analysis, as the cost of processing the forms applies at compilation. At compile time, reversing the order of

application of the function arguments is linear in the number of forms.

2.3.4 some-> and some->>

`macro` `since 1.5`

Listing 2.18. → *functional composition, threading macro, conditional functional application*

```
(some->  [expr & forms])
(some->> [expr & forms])
```

`some->` and `some->>` are variation of the thread first -> and thread last "->>" macros which return immediately if any of the forms evaluates to `nil`. This is especially useful for those functions throwing `NullPointerException` in the presence of `nil` (a common situation with Java interop but not only):

```
(-> {:a 1 :b 2} :c inc) ; ❶
;; NullPointerException

(some-> {:a 1 :b 2} :c inc) ; ❷
;; nil
```

❶An attempt to increment the value for the key `:c` in a map. The key does not exist, returning `nil`. inc throws exception in case of `nil`.
❷The same example using `some->` returns `nil`.

Contract

input

- "expr" is mandatory argument and can be any valid Clojure expression.
- "forms" are additional optional arguments. If any of the optional forms is not a list, a new list is created to wrap the form. The first element of each form must be an callable object (so that `(ifn? (first form))` is `true`).

output

- `some->` returns the result of evaluating the last form by placing the result of the previous form as the second argument of the next, while `some->>` places the evaluated form as the last argument instead. It returns `nil` if any of the forms evaluates to `nil`.

Examples

An idiomatic use of `some->` is when using Java interop, for example to convert strings into numbers. This is often the case when reading from environment variables, something frequent during system startup:

```
(defn system-port []
  (or (some-> (System/getenv "PORT") Integer.) ; ❶
      4444))

(system-port) ; ❷
;; 4444
```

❶The presence of `some->` here prevents a `NumberFormatException` when the "PORT" variable doesn't exist.

❷Invoking `(system-port)` works regardless of the presence of the "PORT" environment variable. When the "PORT" is present, it overrides the default which is used in when the "PORT" is not present.

re-seq is a good candidate for conditional processing with `some->>`: re-seq receives the target string as the last argument and it doesn't tolerate `nil` arguments. Here's a function to extract the content between `<title></title>` tags from some HTML document:

```
(defn titles [doc]                          ; ❶
  (some->> doc                              ; ❷
    (re-seq #"<title>(.+?)</title>")  ; ❸
    (map peek)))                            ; ❹

(titles nil)
;; nil
(titles "<html><head>Document without a title</h
;; nil
(titles "<html><head>
            <title>Once upon a time</title>
            <title>Kingston upon Thames</title>
        </head></html>")
;; ("Once upon a time" "Kingston upon Thames")
```

❶ `match-title` is a simple function that searches for a pair of `<title></title>` tags in a HTML document and then verifies if the title contains a given regular expression.

❷ `some->>` prevents the need for guards against a possible `nil` value.

❸If the entire document is `nil`, we don't want `re-seq` to generate a `NullPointerException`.

❹re-seq returns matching results in vector pairs. peek is the optimal way to access the last item in a vector.



Warning

Using regular expressions to match large HTML documents is possible but not efficient. For intensive HTML processing it is a better idea to use one of the many HTML parsing libraries available (for example Enlive [36]).

A tale of macro families

You could be tempted to think that Clojure already covers all the nuances of pipeline processing with the available threading macros. But threading macros are so useful that they there are several libraries extending them. Two noticeable examples are: LonoCloud Synthread [37] and Pallet Ops thread-expr [38].

Here are a few examples of threading macros coming from those libraries:

Listing 2.19. as→ macro with destructuring

```
(->/as {:a 2 :b 2} {:keys [b]}
  (assoc :large-b (> b 10)))
;; {:a 2 :b 2 :large-b false}
```

Synthread contains an enhanced as-> macro that supports normal Clojure destructuring. The key `:b` is extracted from the map and assigned as local binding for the following form.

Listing 2.20. for-> repetition

```
(-> 1
  (for-> [x [1 2 3]]
    (+ x)))
;; 7
```

The thread-expr `for->` from Pallet Ops allows for repetition of forms in an already existing -> thread first macro. The example shown here expands into: `(-> 1 (+ 1) (+ 2) (+ 3))`

Listing 2.21. binding->

```
(def ^:dynamic *a* 0)
(-> 1
  (binding-> [*a* 1]
    (+ *a*)))
;; 2
```

`binding->` (again from Pallet Ops) allows for binding directly inside the threaded forms. As you can see here the value of `*a*` changes to 1 on a per-thread basis.

Listing 2.22. updating macros

```
(-> {:a 1 :b {:c 2}}
  (->/update :a inc -)
  (->/in [:b :c]))
;; 2
```

These are two examples from the Synthread library. `->/update` and `->/in` are two thread macros dedicated to maps, similar to update and get-in but supporting threading multiple updates in a single call as shown by `(->/update :a inc -)` which is incrementing and changing sign to the value pointed by the key `:a`

See also

- fnil is a function generator that works by wrapping another function. fnil is preferable when the check around `nil` values happens in relation to arguments.

Performance considerations and implementation details

→ `O(n)` **n numbers of forms (compile time)**

There are no special performance considerations to do before using the `some->` macro. It essentially delegates down to ->

which elaborates the forms at compile time (usually without too much performance impact considering the small number of forms normally used in a thread macro).

2.3.5 as->

`macro` `since 1.5`

Listing 2.23. → *functional composition, threading macro, parametrized functional application*

```
  (as-> [expr name & forms])
```

`as->` specializes the two basic threading macros, `->` and `->>`, by adding a new parameter which is used as a placeholder to position the evaluation of the previous form into the next. With `->` and `->>`, the evaluation of the expression at the top is placed at the second place or at the end of the next form, respectively. All the forms in the chain need to obey the same positioning. `as->` enables a precise placement of the evaluation for the next form:

```
(as-> {:a 1 :b 2 :c 3} x   ; ❶
  (assoc x :d 4)           ; ❷
  (vals x)                 ; ❸
  (filter even? x)         ; ❹
```

```
    (apply + x))
  ;; 6
```

❶ `as->` chain starts with 2 elements, the expression to be thread-in and the local binding "x".

❷"x" is used as placeholder in the next form to drive its positioning, in this case right after <assoc,assoc>>.

❸Note that even when there is no ambiguity, "x" needs to be explicit in the form.

❹This is an example of placement as last argument, equivalent to `->>` positioning.

Macroexpanding the form shows how this is easily accomplished:

```
(macroexpand-1 ; ❶
  '(as-> {:a 1 :b 2 :c 3} x
    (assoc x :d 4)
    (vals x)
    (filter even? x)
    (apply + x)))

(let [x {:a 1, :b 2, :c 3}
      x (assoc x :d 4)
      x (vals x)
      x (filter even? x)
```

```
        x (apply + x)]
    x)
```

❶The use of macroexpand-1 prevents expansions beyond `as→` itself.

Since `as->` is based on let, it also supports destructuring (although this was only enabled starting with Clojure 1.8).

Contract

input

- "expr" is any valid Clojure expression. The result of the evaluation of the expression is bound to the placeholder.
- "name" can be either a symbol or a destructuring form. If "name" is a symbol, it can be used as a placeholder in the following forms. If a destructuring form is used instead, the subsequent evaluations in the chain have to be compatible with the destructuring form.
- "forms" an optional list of forms, potentially making use of the placeholder defined before.

notable exceptions

- It throws a generic `Exception` if the binding placeholder is not a symbol or destructuring expression.

output

- `as->` returns the result of evaluating the last form, using the placeholder to refer to the previously evaluated form. If no forms are provided, it returns the evaluation of the expression.

Examples

`as->` is useful in those cases where the threaded value is positioned differently in each form. Here's an example where sequence processing (usually a thread-last operation) is mixed with map processing (a thread-first operation). The example simulates fetching data from some URL endpoint that contains id, name, count triplets:

```
(defn fetch-data [url] ; ❶
  [{:id "aa1" :name "reg-a" :count 2}
   {:id "aa2" :name "reg-b" :count 6}
   {:id "aa7" :name "reg-d" :count 1}
   {:id "aa7" :name nil :count 1}])
```

```
(defn url-from [path] ; ❷
  (str "http://localhost" "/" path))

(defn process [path] ; ❸
  (as-> path <$>
    (url-from <$>)
    (fetch-data <$>)
    (remove #(nil? (:name %)) <$>)
    (reduce + (map :count <$>))))

(process "home/index.html")
; 9
```

❶ `fetch-data` simulates a response after fetching data from a remote service. The `url` parameter is not used in this example.

❷ `url-from` creates a valid URL from a simple path.

❸ We can see `as->` in action. The first 3 forms require the threaded value to appear last, while the last form takes the placeholder in nested position.

In the example above, `process` uses the `as->` threading macro. The chain of operations required to sum the `:count` key for the relevant items requires a mix of function calls and sequence operations, so the evaluation of the previous form is required at a different positions. The choice of the placeholder

symbol `<$>` is arbitrary, but this one is more visible through the forms.

The following example illustrates the use of destructuring with `as->`. One important aspect to understand is that the same destructuring applies during each evaluation despite appearing only once at the top. This allows each form to see fresh update of local bindings based on previous evaluations:

```
(let [point {:x "15.1" :y "84.2"}]
  (as-> point {:keys [x y] :as <$>} ; ❶
    (update <$> :x #(Double/valueOf %))
    (update <$> :y #(Double/valueOf %))
    (assoc <$> :sum (+ x y)) ; ❷
    (assoc <$> :keys (keys <$>)))) ; ❸

;; {:x 15.1, :y 84.2, :sum 99.3, :keys (:x :y :s
```

❶A map contains the coordinates x,y of a point as strings. We destructure the map while declaring the placeholder for `as->`.

❷The value of x and y at this step of the computation is the result of applying destructuring to the previous form, after both x and y have been converted from strings into doubles.

❸Note that the placeholder `<$>` can be used at any location in the expression not just at the beginning.

See also

- The basic threading macros -> and ->> can be regarded as specialized forms of `as->` where the position of the result in the next form is fixed (either the first parameter or the last in the next form).

Performance considerations and implementation details

→ `O(n)` **n number of forms (compile time)**

`as->` is not usually relevant for run-time performance analysis, as the cost of processing the forms applies at compilation. At compile time, reversing the order of application of the function arguments is linear in the number of forms.

[33] The Wikipedia article on combinatory logic is good introduction to the subject: en.wikipedia.org/wiki/Combinatory_logic

[34] Around the reason why the Clojure thread operator cannot be considered a true T-combinator, see the very good explanation by Michael Fogus on his blog blog.fogus.me/2010/09/28/thrush-in-clojure-redux/

[35] The Clojure Style Guide github.com/bbatsov/clojure-style-guide#literal-col-syntax

[36] The Enlive HTML parsing library can be found here: github.com/cgrand/enlive

[37] github.com/LonoCloud/synthread

[38] github.com/pallet/thread-expr

2.4 FUNCTION EXECUTION

2.4.1 apply

`function` `since 1.0`

Listing 2.24. → *Function application, Runtime invocation of functions*

```
(apply
  ([f args])
  ([f x args])
```

```
    ([f x y args])
    ([f x y z args])
    ([f a b c d & args]))
```

`apply`, in the most used form, takes a function and a collection of arguments and returns the result of invoking the function against the arguments in the list. `apply` is useful in those cases where the parameters of a function are generated dynamically and are not known at the time of writing the expression. `apply` can be visualized thinking about "unrolling" or "spreading" arguments from a list to call a function.

Contract

input

- "f" is the function to invoke and is mandatory argument.
- "args" must be a sequential collection and along with "f" it is required argument.
- "x", "y", "z", "a", "b", "c" and "d" are optional arguments that can precede "args".

notable exceptions

- `IllegalArgumentException` when the last parameter is not a sequential collection.
- `ClassCastException` if the first argument is not a callable object.

output

- `apply` returns the result of invoking "f" against the specified parameters.

Examples

A common case for `apply` is string concatenation when the collection of strings to concatenate is known as the result of some runtime computation. Here is for example a function to generate random binary strings of length "n":

```
(defn rand-b [n]
  (->> #(rand-int 2)        ; ❶
       (repeatedly n)       ; ❷
       (apply str)))        ; ❸

(rand-b 10)
; "1000000011"
```

❶The first step creates a function of no arguments to generate random either 0 or 1 with equal probability. This function is required by repeatedly below.
❷The random generator is passed to repeatedly which creates a lazy sequence of randmon "n" bits.
❸We use `apply` with str for the final string concatenation.

The example shows how to generate a random list of bits before converting it into a single string. Another common use of `apply` is to create maps using a list as input:

```
(defn event-stream [] ; ❶
  (interleave (repeatedly (fn [] (System/nanoTim

(apply hash-map (take 4 (event-stream))) ; ❷
; {52284399855900 1, 52284399847705 0}
```

❶ `event-stream` simulates a stream of events coming from some external source in the form of a timestamp followed by a value in a simple sequence.
❷hash-map requires key-value pairs as arguments. We can use `apply` to transform the collection of events into a list of arguments.

The following example illustrates `apply` used in conjunction with map. map accepts any number of collections as input, so `apply` can be used to "spread" arguments to map, for example to process a table of two-dimensional vectors:

```
(def header [:sold :sigma :end]) ; ❶
(def table [[120 3 399] [100 2 242] [130 6 3002]])

(defn totals [table]
  (->> table
    (apply map +)           ; ❷
    (interleave header)))   ; ❸

(println "totals" (totals table))
;; totals (:sold 350 :sigma 11 :end 3643)
```

❶ `header` and `table` represents a typical destructuring of a two-dimensional table into Clojure data structures. `table` contains the actual rows by group of 3 items, while `header` is the title for each column.

❷ `(apply map + table)` for this example is equivalent to `(map + [120 3 399] [100 2 242] [130 6 3002])`. `+` can take any number of arguments (in this case 3) creating a total for each column.

❸Finally, we add the title to each total.

Apply: a piece of Lisp history

Clojure is a dialect of Lisp and Lisp is an old language (the original paper "Recursive Functions of Symbolic Expressions and Their Computation by Machine" is dated 1960, but research in the direction of a new language for symbolic expression started around 1957). Initially Lisp adopted a syntax based on M-expressions, or meta-language expressions, that was not based on lists (as it is now) and looked something similar to:

```
λ[[y;z];cons[car[y];cdr[z]]]
```

The above is a function of two list arguments y,z that produces a new list as output merging `(first y)` and `(last z)` equivalent to `(fn [y z] (cons (first y) (last z)))` in Clojure.

Sometimes between 1958 and 1959, McCarthy wanted to prove that Lisp was better at expressing computability than the formalism of the Turing Machine. Part of that challenge was also to define an "universal Lisp function", a function able to parse and execute another copy of itself written with the same syntax (exactly like the universal Turing Machine is able to accept a definition of itself).

McCarthy had to find a way to express Lisp functions in a form that could be digested by Lisp itself and decided to encode them in lists, using the convention that the first element of the list

was the name of the function and the rest of the list the parameters. McCarthy called this new notation an S-expression (where S stands for Symbolic). The above "cons" M-expression would look like the following as an S-expression (which is perfectly valid modern Lisp):

```
(LAMBDA (Y Z) (CONS (CAR Y) (CDR Z)))
```

The universal function that was able to parse S-expressions and apply them to arguments was called indeed `apply`. McCarthy envisioned `apply` purely for research with no practical scope, until Steve Russel (one of his graduates) decided to implement `apply` in machine language, effectively creating the first Lisp REPL.

See also

- into can be used to create maps (along with other collection types), similarly to what we saw in the examples. One difference is that the input sequence needs to be already in the form of a collection of vector pairs.

- zipmap is the perfect choice to create a hash-map when you have two collections, one containing the keys the other containing the values. Combining the keys and values together and pass them to `apply` would be more verbose.
- reduce can be used to concatenate strings similarly to `apply` with the restriction that `reduce` only takes functions of 2 arguments. For example: `(apply str ["h" "e" "l" "l" "o"])` produces the same result as `(reduce str ["h" "e" "l" "l" "o"])`.
- eval evaluates expressions as lists.

Note

`reduce` performs worse than `apply` for string concatenation. `str` takes advantage of `java.lang.StringBuilder`, a mutable Java object to build strings incrementally, but only when arguments are passed at the same time. `reduce` instead calls `str` repeatedly with 2 arguments only, creating many intermediate string builders. As a rule of thumb, use `apply` when the function is specifically optimized for long sequences of input.

Performance considerations and implementation details

→ `O(1)` **constant time** (up to the 5 arguments)

→ `O(n)` **linear time** (beyond the 5 argument)

`apply` is constant time operation up to the 5th argument, excluding the final collection. It becomes linear after explicitly passing more than 5 arguments:

```
(require '[criterium.core :refer [quick-bench]])
(defn noop [& args])

(quick-bench (apply noop 1 2 []))     ; ❶
(quick-bench (apply noop 1 2 3 4 []))
(quick-bench (apply noop 1 2 3 4 5 6 []))
(quick-bench (apply noop 1 2 3 4 5 6 7 8 []))
(quick-bench (apply noop 1 2 3 4 5 6 7 8 9 10 []

Execution time mean : 61.081153 ns
Execution time mean : 63.026292 ns
Execution time mean : 126.053233 ns
Execution time mean : 202.979379 ns
Execution time mean : 306.982878 ns
```

❶The benchmark measures `apply` while increasing the number of explicit arguments.

Beyond the 5th explicit argument `apply` creates a nested cons list using recursion. The case with more than 5 arguments is uncommon, so `apply` should not be considered a problematic performance hot-spot in normal circumstances.

2.4.2 memoize

`function` `since 1.0`

Listing 2.25. → *Caching, Function Wrapper, Function Generation, Performance*

```
  (memoize [f])
```

`memoize` generates a function that stores the results of an existing one using the argument values as key. When the wrapped function is invoked with the same list of arguments, the result is returned immediately from the cache without any additional computation. The effects of `memoize` are readily visible if we print some message from the wrapped function. We expect the message to appear once for each key:

```
(defn- f* [a b] ; ❶
  (println (format "Cache miss for [%s %s]" a b)
  (+ a b))

(def f (memoize f*)) ; ❷

(f 1 2)
;; Cache miss for [1 2]
;; 3

(f 1 2)
;; 3

(f 1 3)
;; Cache miss for [1 3]
;; 4
```

❶ `f*` is the function we intend to put under caching.

❷ `memoize` takes the target function as argument and nothing else. It produces a new function that we can define in the current namespace.

The first invocation generates the message while the following for the same combination of keys are not, confirming that the wrapped function `f*` is not invoked again.

There is no universal convention for naming, but given the connection between the target function and the one generated by `memoize` the two names should be somewhat related. In our examples, the public interface of the function remains the same, while the memoized version is private and is added a star "*" at the end.

Contract

input

- "f" needs to be a function and is mandatory argument.

notable exceptions

- `ClassCastException` if "f" is not callable.
- `ArityException` when called without arguments.

output

- A new function of a variable number of arguments that stores the results of the evaluation in an internal map.

Examples

`memoize` works well for non-trivial computations that accept and return values with a small memory footprint. The following example illustrates the point. The Levenshtein distance [39] is a simple metric to measure the difference between two strings. The distance can be used, for example, to suggest corrections for common spelling mistakes. The distance is straightforward to implement but becomes computationally intensive for longer strings (above 10 characters or more). We could use `memoize` to save us from computing the distance of the same pair of strings over and over again. The input (the strings arguments) and the output (a small integer) are relatively small in size, so we can cache a large amount of them without exhausting memory (assuming the list of words with which the function is invoked is some finite number that we can estimate).

To feed our example we are going to use a dictionary of words in plain text format (on Unix systems such file is available at "/usr/share/dict/words"). If we were asked to implement an auto-correction service, it could work as follow:

1. The user input a misspelled word.
2. The system checks the distance of the word against the words in the dictionary.
3. Results are returned in order of smaller distance.

We are also going to pre-compute several small dictionaries starting with the initials of the word, a technique to further speed-up the distance calculation:

```
(defn levenshtein* [[c1 & rest1 :as str1]
                    [c2 & rest2 :as str2]]
  (let [len1 (count str1)
        len2 (count str2)]
    (cond (zero? len1) len2
          (zero? len2) len1
          :else
          (min (inc (levenshtein* rest1 str2))
               (inc (levenshtein* str1 rest2))
               (+ (if (= c1 c2) 0 1) (levenshtei

(def levenshtein (memoize levenshtein*))

(defn to-words [txt init]
  (->> txt
       slurp
       clojure.string/split-lines
       (filter #(.startsWith % init))
       (remove #(> (count %) 8))
       doall))

(defn best [misp dict]
  (->> dict
       (map #(-> [% (levenshtein misp %)])))
```

```
        (sort-by last)
        (take 3)))

(defn dict [init]
  (to-words "/usr/share/dict/words" init))

(def dict-ac (dict "ac"))

(time (best "achive" dict-ac))
;; "Elapsed time: 4671.226198 msecs"
;; (["achieve" 1] ["achime" 1] ["active" 1])

(time (best "achive" dict-ac))
;; "Elapsed time: 0.854094 msecs"
;; (["achieve" 1] ["achime" 1] ["active" 1])
```

❶The Levenshtein algorithm presented here is a variation of the many similar ones available online. The important aspect to remember is that it growths roughly as `O(n*m)` where `m` and `n` are the length of the strings, or in other words `O(n^2)` in the worst scenario.

❷This def actually builds the wrapping function through `memoize`, conveniently called `levenshtein` without the final `*` that is reserved for the non-memoized version.

❸`to-words` is an helper function to prepare the dictionary filtered by the initial string. `to-words` is part of the "static" or "learning" phase of the algorithm, since we can prepare words by initial off-line and store them for later use.

❹The `best` function is responsible for the application of the `levenshtein` memoized function to the words in the dictionary. It then sorts the results with sort-by and returns the lowest distances.

❺The def invocation is defining a filtered dictionary starting by "ac" so it doesn't need to be computed multiple times. This also prevents the `time` function to report on the time needed to read and process the file.

❻The first invocation to search the best matches for the misspelled word returns in almost 5 seconds.

❼The second invocation returns much faster.

The memoized version of the distance function stores each new pairs of strings as key and the returned distance as the value in an internal map. Each time the function is invoked with the same arguments the return value is fetched from the map.

The example also shows a way to "train" the memoized distance before actual use. A real application could pre-compute a set of dictionaries by initials similar to the indexing happening inside a database. This technique contributes to the speed-up seen in our implementation, but for serious applications there are algorithms outperforming Levenshtein [40].

What's in a name: `memoize`?

There is a reason why storing arguments and return values is called "memoization" instead of just "caching". Memoization is more specific because it implies two features normally present in functional languages: pure and higher order functions.

Pure functions

The wrapped function needs to be referentially transparent. If there are factors other than the input arguments influencing the results, then cached results could be different. The cache would then need to be aware of this side effecting "context" and use it as part of the key (if possible). Memoization becomes straightforward in functional languages supporting referential transparency.

Higher order functions

"Higher order" refers to the property of a function to be treated as a value. As such, the function can be stored, passed to other functions or returned. Not all languages offer higher order functions, although it is now more common to offer this feature. By describing this kind of caching as "memoization" it is implied that a function can be transparently decorated with caching capabilities. "Transparently" in this context means that the original wrapped function remains untouched.

See also

- lazy-seq creates a "thunk" (wrapper function around a value) that evaluates its content on first access and return a cached version on following calls. When the thunks are joined together in a sequence it forms a lazy sequence. Lazy sequences are comparable to a cache where the order and value of the keys is predetermined. An "evaluate once" semantic on collections can be achieved with lazy-seq. Since all Clojure sequences are lazy, you might be already using a "cached data structure" without knowing it.

- atom creates a Clojure Atom, one of the possible Clojure reference types. `memoize` uses an `atom` to store results. Use a custom atom when `memoize` implementation is too restrictive for a specific kind of caching. You can for example look into something different than a Clojure hash-map to store items in the map, like a mutable Java map with soft-references [41]. Keep in mind that there are already libraries like core.cache (github.com/clojure/core.cache) to provide common caching strategies if this is what you're looking after.

Performance considerations and implementation details

=> `O(1)` **steps (function generation)**

=> `O(n log n)` **steps (generated function), n number of unique keys**

=> `O(n)` **space (generated function), n number of unique keys**

The main aspect to consider about `memoize`, is that it stores cached items indefinitely. Constant accumulation of new cached values will eventually exhaust memory. `memoize` users should pay attention to these facts when designing their solution, more

specifically around the prospected distribution of keys in the cache. `memoize` should not be used in case of long-running services when the amount of argument permutations is potentially infinite or not easy to predict.

We can gather some statistics about the key distribution with some changes to the original `memoize` function. The following `memoize2` contains additional atoms to collect data cache hits, misses and total number of calls at run-time.

```
(defn memoize2 [f]
  (let [mem (atom {}) ; ❶
        hits (atom 0)
        miss (atom 0)
        calls (atom 0)]
    (fn [& args]
      (if (identical? :done (first args)) ; ❷
        (let [count-chars (reduce + (map count (
          {:calls @calls
           :hits @hits
           :misses @miss
           :count-chars count-chars
           :bytes (* (int (/ (+ (* count-chars 2
        (do (swap! calls inc) ; ❹
            (if-let [e (find @mem args)]
              (do (swap! hits inc) (val e))
              (let [ret (apply f args)
```

```
                    _ (swap! miss inc)]
                (swap! mem assoc args ret)
                ret)))))))
```

❶Along with the actual cache, additional counters are added to the initial let block.

❷ `:done` is a sentinel value that can be used to extract statistics during run-time.

❸This is an estimate of the amount of memory necessary to store the keys given the number of chars [42].

❹Additional swap! operations are performed to update counters.

By making access to the additional stats at run-time, we can estimate the key-space size or the memory footprint. If we run the same Levenshtein example replacing `memoize` with `memoize2` we can extract the following results:

```
(def levenshtein (memoize2 levenshtein*))

(best "achive" dict-ac)
;; (["achieve" 1] ["achime" 1] ["active" 1])

(levenshtein :done)
;; {:calls 400, :hits 0, :misses 400 :count-char
```

```
(best "achive" dict-ac)
;; (["achieve" 1] ["achime" 1] ["active" 1])

(levenshtein :done)
;; {:calls 800, :hits 400, :misses 400 :count-ch
```

As you can see, the first time the `best` function is invoked it generates 400 misses while the second time it results in all hits. We can also an estimate of the memory taken by the strings stored in memory which is around 10Kb.

Second aspect to consider when using `memoize` is the additional hash-map assoc operation and atom swap! that is added for each new key combination presented as input. The hash-map adds `O(n log n)` steps to add a new key while the atom could under perform under heavy thread contention. Depending on the application requirement, `memoize` could be built on top of a transient data structure to avoid the performance penalty of filling the cache. Another option to consider, when possible, is "warming the cache": while the

application is still not serving live traffic, cache could be populated artificially with the most common keys.

2.4.3 trampoline

`function` `since 1.0`

Listing 2.26. → *Recursion, Function Invocation, Performance*

```
  (trampoline
    ([f])
    ([f & args]))
```

`trampoline` is a function invocation helper usually found in conjunction with mutual recursion (see the call-out further below for more details). It invokes the given function checking the results: if the invocation returns another function, `trampoline` invokes it again until the result is not a function anymore. `trampoline` is itself a recursive function (based on loop-recur) which uses the type of the input as the exit condition.

Contract

input

- "f" is a function of any number of arguments that can return an invocable object (such that `(fn? object)` yields `true`). "f" will need to return an object so that `(fn? object)` is `false` at least once to prevent `trampoline` from going into an infinite recursion.
- "args" are the optional arguments to pass to "f".

output

- The result of invoking "f" over optional "args" until the return type is not a function. `trampoline` exit condition checks the returned type with fn?. Vectors, sets, keywords and symbols are also invocable objects, but they are not considered invocable by `trampoline`.



Warning

If the input function "f" already returns a function as the final result, that function will need to be wrapped in a collection (or other object so that `(fn? object)` is `false`) to make sure `trampoline` has a proper exit condition.

Examples

`trampoline` can be used to transform stack consuming mutually recursive functions into a tail-recursive iteration. Mutual recursion doesn't occur that often in everyday programming, but it has a couple of interesting applications. State machines, for instance, are a well known example of problem that mutual recursion solves in an elegant way. The following example shows how a traffic light (based on US traffic laws) can be implemented as a state machine and how `trampoline` can be used to prevent stack-overflow in case of very long sequences of state transitions:

```
(defn- invoke
  [f-key & args]
  (apply (resolve (symbol (name f-key))) args))

(defn green [[light & lights]]
  #(case light
     :red false
     nil true
     (invoke light lights)))

(defn red [[light & lights]]
  #(case light
     :amber false
     nil true
     (invoke light lights)))
```

```
(defn amber [[light & lights]]
  #(case light
     :green false
     nil true
     (invoke light lights)))

(defn flashing-red [[light & lights]]
  #(if (nil? light) true (invoke light lights)))

(defn flashing-amber [[light & lights]]
  #(if (nil? light) true (invoke light lights)))

(defn traffic-light [lights]
  (trampoline flashing-amber lights))

(traffic-light [:red :amber :red])
;; false

(traffic-light [:red :green :amber :red])
;; true

(time (traffic-light (take 10000000 (cycle [:amb
;; "Elapsed time: 5919.991775 msecs"
;;true
```

❶ `invoke` takes a function as a keyword (such as `:+`) and related args `(1 2)` and invokes `(+ 1 2)` provided `:+` can be found in the current namespace. The example uses `invoke` to call one of the possible traffic light states passing the rest of required transitions as arguments.

❷ The `green` state function deals with the traffic light when the green light is already on. The function will determine what should happens given the next required state transition. Other functions for other colors work the same way. The case switch is instructed to return `false` if the transition is not possible, a condition that forces `trampoline` to break the chain. `nil` needs to be handled separately, since this is the transition list terminator marker. The termination marker signals that all transitions were successful. The catch-all branch at the end of the case statement deals with any additional valid transition. `invoke` calls the next transition once the color keyword (any of `:green`, `:amber` or `:red`) has been translated into the corresponding function.

❸ `flashing-red` and `flashing-amber` have one case less to deal with, because all states are allowed from a flashing light condition. The case statement has been replaced with an if compared to previous states.

❹ `traffic-light` is the entry point. It starts the chain of calls through `trampoline`. Once the traffic light is turned on for

the first time, the first state is `flashing-amber`.
The last call to `traffic-light` in the example shows what happens when we call a life-time long list of traffic light states (with a 2 minutes total time per loop, 10 million cycles correspond to roughly 39 years of continuous traffic light activity). Every item in the list could potentially create a new stack frame, but thanks to `trampoline` the mutual recursion executes on the heap.

See also

- iterate has similar effects to recursion, but it creates a sequence of intermediate results instead of returning the final result. iterate is not an alternative to `trampoline` as they solve different problems.
- loop-recur is at the core of `trampoline` implementation, eliminating the problem of consuming the entire stack space.

Performance considerations and implementation details

=> `O(n)` **with n depending on the input function**

=> `O(1)` **space**

As discussed throughout the chapter, `trampoline` is an important tool for mutually recursive functions. It is probably a good idea to always use it considering the negligible cost for the additional wrapping function. The number of steps required by `trampoline` to finish the computation is determined exclusively by the input function. `trampoline` makes good use of loop-recur to prevent consuming the stack and doesn't use any other memory space.

[39] The Wikipedia article contains a good introduction to the Levenshtein Distance algorithm: en.wikipedia.org/wiki/Levenshtein_distance

[40] See the list of metrics available on Wikipedia: en.wikipedia.org/wiki/String_metric

[41] There are several examples of use of SoftReference for caching in Java. This is a good starting point: www2.sys-con.com/itsg/virtualcd/java/archives/0507/shields/index.html

[42] A good enough formula to estimate the amount of memory necessary to store strings in Java is: www.javamex.com/tutorials/memory/string_memory_usage.shtml

2.5 SUMMARY

In this chapter we learned how to define and use functions. Many functions and macros in the standard library generate

other functions with specific properties, for example fnil or juxt and these new functions can be used to compose new behavior. Threading macros in particular are very interesting, as they alter the normal evaluation flow to produce a different syntax.

3BASIC CONSTRUCTS

This chapter groups together some of the most important constructs in Clojure (and similarly other programming languages): conditional branching, iteration and local scope definition. There are other aspects that could be added to this category like namespaces, variables or functions, but because of their complexity they are treated separately.

You might be surprised to see things like conditionals, switch statements or loops as being part of the standard library. But Clojure (as many other Lisps before) builds up from a small core of primitives called **special forms** and many constructs that would be considered "reserved words" in other languages are instead defined in the standard library. This is why the Clojure standard library could be considered also a language specification.

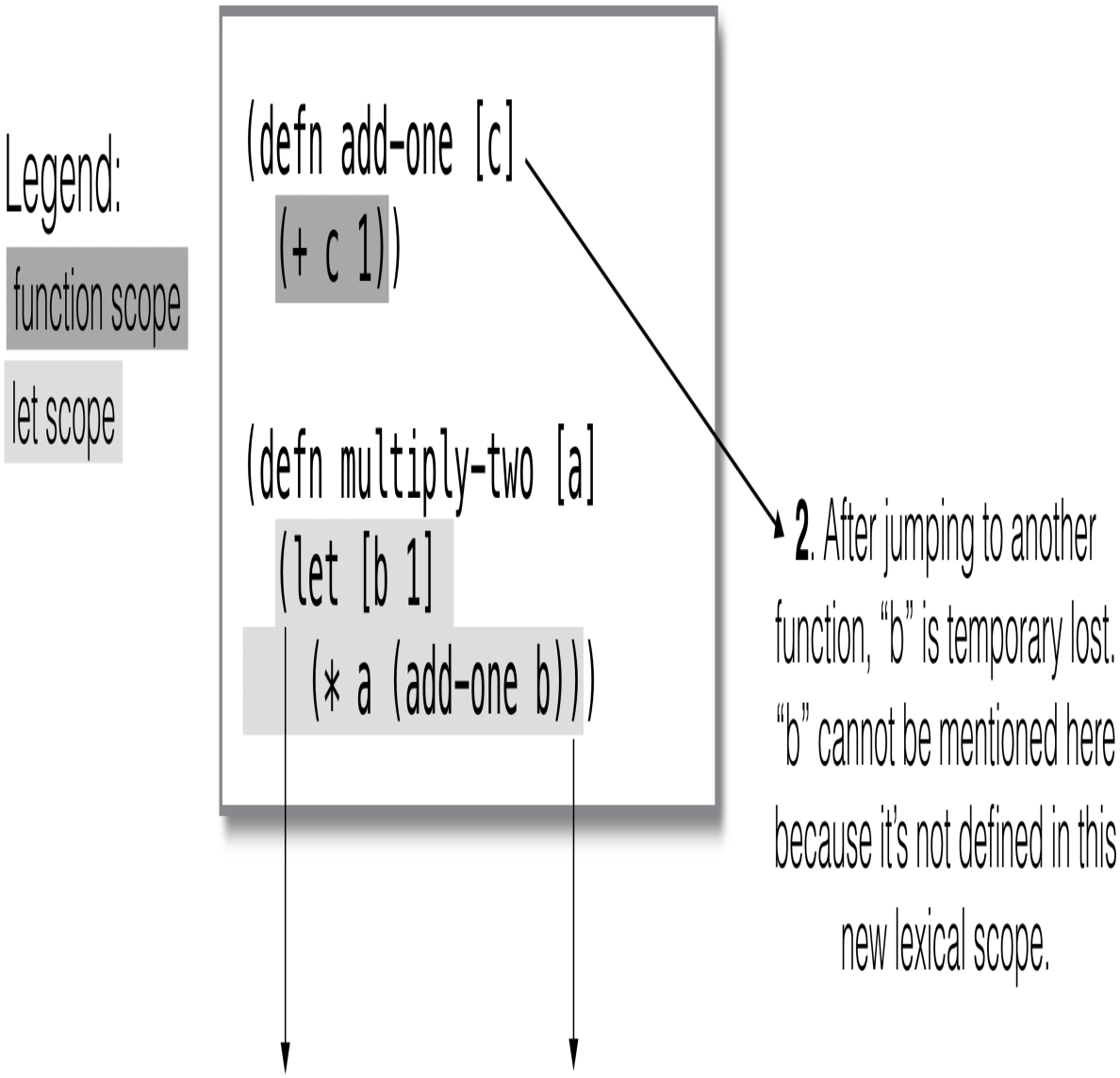
The book treats special forms, macros and functions equally, provided they are publicly available for use after Clojure

bootstrap. Some of them however, are meant for Clojure internal use and just mentioned briefly.

3.1 LEXICAL BINDING

Lexical binding is a mechanism to create an accessibility boundary for values. The mechanism works by assigning values to symbols in the scope defined by surrounding parenthesis (hence why it’s called "lexical"). The following picture for example, shows what boundaries are created by the let macro:

Figure 3.1. Lexical scope for locally bound symbol "b".


Legend:
function scope
let scope
(defn add-one [c]
(+ c 1))
(defn multiply-two [a]
(let [b 1]
(* a (add-one b))))
2. After jumping to another function, "b" is temporary lost. "b" cannot be mentioned here because it's not defined in this new lexical scope.
1. "b = 1" is the initial value of "b" as marked by the parenthesis surrounding the let block.

The symbol "b" defined by the let macro is only visible when you consider the surrounding parenthesis. When `add-one` is invoked, we can't use "b" anymore because it cannot be resolved in the newly created scope. There is indeed a close relationship between the scope created by a function declaration and the scope created by a let-like form. Indeed, `let` can be considered syntactic sugar for a lambda function invocation, as illustrated by the following example:

```
((fn [a b] (* (+ a b) b)) 1 2) ; ❶

(let [a 1 b 2] (* (+ a b) b))  ; ❷
```

❶The anonymous function created with fn is invoked right away on a couple of arguments. The function declares two arguments, "a" and "b", which are locally bound to the values "1" and "2" during evaluation. Once inside the body of the function, the arguments can be used many times without any further evaluation. The scope of "a" and "b" is bound lexically by the parenthesis defining the anonymous function.

❷This `let` declaration achieves the same effect of the previous anonymous function but reads much better: the symbols and values are now close together followed by the main code block.

There is a clear equivalence between `let` and anonymous functions which sets local bindings apart from the usual procedural variable assignment: it's all just immutable parameter passing. Despite this, even purely functional lexical binding is colloquially referred to as "assignment" because of the striking similarities. Like imperative assigned variables, let-bound symbols are available throughout the lexical scope without any further evaluation of the expression they refer to. Although it's common to refer to symbols as "assigned variables", the similarity with the imperative world stops right there:

- There is no concept of location where the value has been stored.
- Once bound, there is no way to mutate a symbol so it produces a different value.
- The same symbol can be re-bound by shadowing the previous (that doesn't mutate at all) using another binding form.

The macros and special forms in this group offers different possibilities to create lexical bindings. The most general let is followed by a few variants that can conditionally define symbols or functions. if-let and letfn for example are useful to remove some typing overhead when creating local symbols. All

let-like forms (except `letfn` which has a slightly different syntax) accept a vector of pairs which are then used to create the bindings and a body to execute against those bindings. Lexical binding forms additionally offer facilities like destructuring, a concise syntax to allow portions of Clojure collections to be directly assigned to symbols (see destructure for details on how destructuring works and its syntax).

3.1.1 let and let*

`macro` (let) `special-form` (let*) `since 1.0`

Listing 3.1. → *Local Binding, Lexical Scope, Assignment*

```
(let [bindings & body])
```

`let` is a very frequently used Clojure macro. One of the main uses of `let` is to create a local name which stands for the evaluation of an expression, so the expression doesn't need re-evaluation every time it's used. For example:

```
(let [x (rand-int 10)] ; ❶
  (if (>= x 5)
    (str x " is above the average")
    (str x " is below the average")))
```

❶There is a 50% probability for "x" to be be below or above 5.
The evaluation of rand-int happens only once.

Once the local binding "x" has been established, the symbol can be used without re-evaluation of rand-int (which would then become problematic, since it would return different values for each invocation).

destructuring is another common case for using `let`, when applying the equivalent in the function parameters is not possible or impractical. `let*` is the special form used by `let` internally to parse and validate bindings. From the user perspective there is no specific reason to use `let*` directly, so this chapter focus mainly on `let`.

Listing 3.2. Contract

```
    (let [bindings & <body>])

    bindings :=>
    [<bind1> <expr1>, <bind2> <expr2> .. <bind-N> <
```

input

- "bindings" is a (possibly empty) vector containing an even number of elements.
- "bind1", "bind2", .. , "bind-N" are valid binding expressions as per destructuring semantic. They must appear on an even index in the bindings vector (position 0, 2, 4 and so on).
- "expr1", "expr2", .. , "expr-N" are valid Clojure expressions and must appear on an odd index in the bindings vector (position 1, 3, 5 and so on).
- "<body>" is an optional group of expressions (they don't need explict wrapping in a list or other data structure). The "body" is automatically wrapped in a do block.

notable exceptions

- `UnsupportedOperationException` when type hinting a local binding with a primitive type. For example the following expression is not valid: `(let [^long i 0])`. `let` automatically recognizes types for primitive locals (like longs, doubles, etc.) and does not accept type hints in this case.

output

`let` returns the evaluation of the last expression in "body" (if multiple are present) allowing expressions to refer the bound names set by the binding pairs. It returns `nil` when "body" is empty.

Examples

The following code implements the interaction loop commonly found in games with multiple players. If we assume a human playing against the computer, there is usually a phase of "input" followed by an action taken by the computer, including printing the current move on the screen or deciding who is the winner. Let's take for example the console version of rock-paper-scissor [43]:

```
(defn rule [moves]
  (let [[p1 p2] moves]  ; ❶
      (cond
        (= p1 p2) "tie game"
        (every? #{"rock" "paper"} moves) "paper w
        (every? #{"scissor" "rock"} moves) "rock
        (every? #{"paper" "scissor"} moves) "sci
        :else "computer can't win that!")))

(defn game-loop []   ; ❷
  (println "Rock, paper or scissors?")
```

```
    (let [human (read-line)                  ; ❸
          ai (rand-nth ["rock" "paper" "scissor"])
          res (rule [human ai])]
      (if (= "exit" human)
        "Game over"
        (do
          (println (format "Computer played %s: %s
          (recur)))))    ; ❹

  (game-loop)
  ;; Rock, paper or scissors?
  ;; Bang
  ;; Computer played scissor: computer can't win t
  ;; Rock, paper or scissors?
  ;; paper
  ;; Computer played rock: paper wins over rock
  ;; Rock, paper or scissors?
  ;; exit
  ;; "Game over"
```

❶ `rule` contains the rock paper scissor rules, which are easy to implement. We need to check if the two choices are included in one of the possible sets (independently from the order) and return the corresponding messages. This is for instance an idiomatic use of a Set as a function-predicate and every? to verify each of the choices. `let` is used here for destructuring only: `p1` and `p2` can now be referenced without any assistance from first or last to extract them from the `moves` parameter.

❷the `game-loop` is a recursive function that repeats multiple plays until the human player types "exit" at the console. read-line is used to read from standard input.

❸ `let` is declaring three local binding that will be used (potentially multiple times) over the contained block. You can see that `ai` is also used directly in the following binding to retrieve the rule results.

❹We finally recur over the function (no loop statement).

The rock-paper-scissor example shows two facts about `let` (this extends to the other flavors letfn and if-let): the locally bound symbol (in this case `ai`) is immediately available for other binding definitions. This implicitly defines an ordering for the evaluation of the right-side expressions, so they can mutually refer the defined symbols.

The second interesting aspect of the example is that `let` has been used in the `rule` function to destructure the single sequential (vector) argument into its first and last component. Destructuring is removing the need to use `(= (first moves) (last moves))` for the condition in the if statement, saving quite a few keystrokes. Since `let` is so connected with the concept of function parameters, destructuring is available for defn exactly in the same way. Using it in defn or in the inner let is essentially a matter of opportunity and taste.

Different `let` for different Lisps

People coming from Common Lisp might be confused by `let*` ("let star"), a Clojure special form that doesn't support destructuring and not meant for public use. The reason is that in Common Lisp there are two flavors of `let`:

- Common Lisp `let` creates bindings independently (and potentially in parallel, although this is a compiler implementation detail) so each individual pair cannot see local symbols defined by another pair. All local symbols will be then available in the main let block at the same time.
- Common Lisp `let*` is instead the same as Clojure `let`, allowing the expression under evaluation to establish a binding to see previously declared symbols right away.

The reason why Common Lisp offers the two forms and takes the less imposing `let` as the default choice is often subject to debate [44]. The author of Clojure decided to incorporate `let*`

only flavor into Clojure once and for all (simply renamed as `let`), preventing any further debate.

See also

- letfn creates a local binding from a symbol directly into a function definition. It replaces the slightly more verbose `(let [f (fn [x])])` to declare a local function.
- if-let and when-let are specialized `let` version wrapping a condition on top of the `let` definition. Use them when the `let` body starts with if or when. In this case the `let` binding can be completely skipped if the expression in the pair is evaluated to `nil`.
- for could be considered a sequential `let` and indeed, it also supports destructuring. Consider using for when the symbol should be bound to the next element of a sequence each time the body is evaluated.

Performance considerations and implementation details

=> `O(n)` **Linear in the number of binding pairs**

`let` performance implications are negligible or unimportant for all practical purposes. One reason is that as a macro, `let`

has impact at compile time only. A second reason is that its normal use doesn't imply a huge number of bindings.

Once agreed that `let` performance profile doesn't have normal practical implications, the following machine automated `let` generator is shown here for the curious reader. We can use a macro to generate a huge `let` definition:

```
(defn- generate-symbol [n]
  (symbol (str "a" n)))

(defn- generate [n]
  (->> (range n)
       (map (juxt generate-symbol identity))
       flatten
       vec))

(defmacro large-let [n]
  (let [bindings (generate n)]
    `(let ~bindings
       (reduce + [~@(map generate-symbol (range

(macroexpand '(large-let 2))                    ; ❶
;; (let* [a0 0 a1 1] (reduce + [a0 a1]))
```

```
  (large-let 5000)                                    ; ❷
  ;; CompilerException java.lang.RuntimeException:
```

❶ macroexpand shows what the macro is doing, which is simply declaring a few symbols `a0`, `a1`, `..` sequentially and reducing their values in the body.

❷ `large-let` is then used to forge an unusually large let

As you can see, `large-let` generates a large let definition that in turns generates enough bytecode to go beyond the limit allowed by the JVM for the length of a single method. Let's use a disassemble utility like `no.disassemble` [45] to see what's going on under the hood:

```
  (require '[no.disassemble :refer [disassemble]])
  (println (disassemble (fn [] (large-let 2))))

  public final class LetPerf extends clojure.lang.A

    // Omitted some static class attributes declar

    // Method descriptor #11 ()Ljava/lang/Object;
    // Stack: 6, Locals: 5
    public java.lang.Object invoke() {              ❶
       // 0  lconst_0
       // 1  lstore_1 [a0]
```

```
       // 2  lconst_1
       // 3  lstore_3 [a1]
       // Omitted bytecode related to loading redu
      // 28  lload_1 [a0]
      // 29  invokestatic clojure.lang.Numbers.num
      // 32  lload_3 [a1]
      // 33  invokestatic clojure.lang.Numbers.num
      // 36  invokeinterface clojure.lang.IFn.invo
      // 41  invokeinterface clojure.lang.IFn.invo
      // 46  areturn
    }

    // Omitted static block initializer
  }
```

❶ `no.disassemble` output has been cleaned up a little to show the most important features. Basically the `invoke()` method generated to allow the function created by fn to be invoked, is allocating a long `const` on the stack for each pair in the bindings, explaining why a large number of them can go beyond the allowed method length.

The generated bytecode also explain the linear aspect of the performance profile, since the `let*` Java code needs to iterate

through each passed binding to create the necessary bytecode invocation.

3.1.2 if-let, when-let, if-some and when-some

`macro`

`since 1.0 (if-let, when-let)`

`since 1.6 (if-some, when-some)`

Listing 3.3. → *Local Binding, Lexical Scope, Conditional Assignment*

```
(defmacro if-let
  ([bindings then])
  ([bindings then else))

(defmacro when-let
  [bindings & body])

(defmacro if-some
  ([bindings then])
  ([bindings then else))

(defmacro when-some
  [bindings & body])
```

`if-let`, `when-let`, `if-some` and `when-some` are specialized versions of let to create lexically bound names. They support a single symbol-expression pair in the binding vector. The form in the body is conditionally evaluated (with the symbol included in the local scope) based on the expression being logical true/false (`if-let` and `when-let`) or `nil` (`if-some` and `when-some`).

`if-let` and `if-some` allow the selection between two possible forms to be executed based on the condition, while `when-let` and `when-some` either execute the forms (using an implicit do) or return `nil` (equivalent to if and when semantic respectively). Here is some simple example to demonstrate their use:

```
(if-let [n "then"] n "else")
;; "then"

(if-let [n false] n "else")
;; "else"

(when-let [n "then"] n)
;; "then"

(when-let [n false] n)
;; nil
```

`if-some` and `when-some` are based on the expression being evaluated as "not nil". They are better understood with a mental translation into "if-not-nil?" and "when-not-nil?":

```
(if-some [n "then"] n "else")
;; "then"

(if-some [n nil] n "else")
;; "else"

(when-some [n "then"] n)
;; "then"

(when-some [n nil] n)
;; nil
```

The only case where you need to be careful is where the concepts of being "logical true" and "not nil" overlap and differ, such as testing `false`:

```
(if-let [n false] n "else") ; ❶
;; "else"

(if-some [n false] n "else") ; ❷
;; false
```

❶ `if-let` is testing for logical true/false. The expression is `false` hence the alternative body "else" is returned.

❷ `if-some` tests for `not nil`. Since `false` is different from `nil` the expression `(not (nil? false))` is `true` and the first body returning the content of the bound variable is returned for evaluation.

Listing 3.4. Contract

```
(if-let [bind expr] <then-form> <else-form>)
(if-some [bind expr] <then-form> <else-form>)

(when-let [bind expr] <then-form>)
(when-some [bind expr] <then-form>)
```

input

- "bind" must be a valid binding expression as per destructure.
- "expr" is any evaluable form. The result of its evaluation is bound (and potentially destructured) to "bind".

- "then-form" is any evaluable form. In the case of `if-let/if-some` "then-form" must be present as a single top level expression. In the case of `when-let/when-some` "then-form" is wrapped by an implicit do block, allowing multiple forms at the same level (or no forms at all).
- "else-form" is optional.

output

- `if-let` returns the evaluation of "then-form" if "expr" is logical `true`, the evaluation of "else-form" if "expr" is logical `false`. It returns `nil` if "expr" is logical `false` and no "else-form" expression is present.
- `if-some` returns the evaluation of "then-form" if "expr" is not `nil`, the evaluation of "else-form" otherwise. It returns `nil` if "expr" is logical `false` and no "else-form" expression is present.
- `when-let` returns the evaluation of "then-form" if "expr" is `true`, `nil` otherwise.
- `when-some` returns the evaluation of "then-form" if "expr" is not `nil`. `nil` otherwise.

Examples

The most common usage of conditional let expressions is in the context of a let form immediately followed by an if or when condition testing for the content of the locally bound symbol. The following function for example, is counting the lines of code (LOC) for files in the classpath (the virtual file system that Java implements aggregating all known code sources):

```
(require '[clojure.java.io :as io])
(require '[clojure.string :as s])

(defn loc [resource]
  (let [f (io/resource resource)]             ;
    (when f
      (count (s/split-lines (slurp f))))))    ;

(defn total-loc [& files]                     ;
  (reduce + (keep loc files)))

(total-loc "non-existent" "clojure/core.clj" "cl
;; 7570
```

❶clojure.java.io/resource is a function creating a `java.net.URL` object from a file in the classpath. It returns `nil` if the file is not found.

❷We don't want to execute slurp on a potentially `nil` resource, since it will force us to handle an exception. So we guard against using when.

❸`total-loc` accepts a variable number of files. The total count is calculated with reduce after calling `loc` on each file and removing any potential `nil` for non-existent files using keep.

The `loc` function can be improved by combining creation and check on the resource with `when-let`:

```
(defn loc [resource]
  (when-let [f (io/resource resource)]     ; ❶
    (count (s/split-lines (slurp f)))))

(total-loc "non-existent" "clojure/core.clj" "cl
;; 7570
```

❶The when simply disappeared, removing one set of parenthesis in the process.

Note that using `when-some` in this case could potentially allow "f" to be `false`, forcing `slurp` to generate exception. We know that the contract of `io/resource` does not allow `false`, but in general, we might not know the exact details of the function we are calling.

`if-let` expands the possibilities of `when-let` offering an additional body to execute in case the local binding is `nil` or `false`. We could for example handle the case in which the file isn't found, returning a count of 0 instead of `nil`:

```
(defn loc [resource]
  (if-let [f (io/resource resource)]    ; ❶
    (count (s/split-lines (slurp f)))
    0))                                 ; ❷

(defn total-loc [& files]
  (reduce + (map loc files)))           ; ❸

(total-loc "non-existent" "clojure/core.clj" "cl
;; 7570
```

❶ `if-let` is now replacing `when-let`. Since the "else" body is optional, this would work like before without any other changes. In this case though, we want a specific value other than `nil` to be returned.

❷ The "else" body is simply "0". This effectively prevents the function from returning `nil`.

❸ A positive effect of the introduction of `if-let` and the 0 default propagates down to the reduce: we don't need to think about the potential presence of `nil` anymore.

Despite the missing "let" in the name, `if-some` and `when-some` works the same as `if-let` and `when-let` with a modification to accommodate scenarios where `nil` has a different meaning than `false`. One example of this behavior happens while processing core.async channels [46].

core.async models computation as streams of items "flowing" through channels from producer to consumer. Channels are designed to be "open-ended" and it's an agreement between consumer and producer to mark the end of the computation. By calling `close!` on a channel, the producer sends a conventional `nil` element to signal the consumer that there are no more items. This is the reason why `nil` cannot be sent down a channel for other reasons other than to close the communication.

The following example shows the typical master-worker model using core.async. The worker needs to loop on available items until the `nil` signal is reached, processing them one by one. This is a good use case for `if-some`:

```
(require '[clojure.core.async :refer
           [go go-loop chan >! <! <!! close!]])

(defn- master [items in]                ; ❶
  (go
    (doseq [item items]
      (>! in item))
    (close! in)))

(defn- worker [out]                     ; ❷
  (let [in (chan)]
    (go-loop []
       (if-some [item (<! in)]          ; ❸
         (do
           (>! out (str "*" item "*"))  ; ❹
           (recur))
         (close! out)))                 ; ❺
    in))

(defn process [items]                   ; ❻
  (let [out (chan)]
    (master items (worker out))         ; ❼
```

```
      (loop [res []]
        (if-some [item (<!! out)]          ; ❽
          (recur (conj res item))
          res))))
```

❶The `master` function takes the items to process as input along with the channel the items should be sent to. The `master` signals the end of the items by closing the channel.
❷The `worker` receives the channel where results should be sent and creates the input channel that will be used by the `master` to send items through.
❸ `if-some` assigns the next element to the symbol `item` in the following lexical scope. If the `item` is different from `nil` (thus including potential boolean `true` or `false`) the item gets processed and the loop recurs.
❹Processing is simulated by decorating each item with "*".
❺If the channel returns a `nil` the output will be closed.
❻ `process` coordinates `worker` and `master`. It also iterates the results from the output channel once the computation finishes, transforming the result back into a sequence.
❼This line effectively starts the computation. The worker call is evaluated first. Thanks to the go-block the wait on the input channel won't block (but just park) returning the input channel that is needed by the master.
❽Another example of `if-some` used for the same reasons as before.

Conditional let extensions

A little bit like Threading macros, lexical scope related functions are often subject to extensions and improvements. Clojure already provides a very good coverage of

common use cases with let, `if-let`, `when-let` and letfn so there is no need to depend on external libraries. For those special cases requiring more flexibility, here are a few ideas coming from other languages and libraries. The following are a couple of examples coming from outside Clojure: `letrec` and `aif`.

Scheme `letrec`

`letrec` in Scheme expand on the concept of visibility, making symbols available even to expressions coming before that symbol definition. `letrec` can be used to make mutually referencing let bindings, for example (here translated in how it would look like in Clojure):

```
(letrec [is-even? #(or (zero? %) (is-odd? (dec %)))
         is-odd? #(and (not (zero? %)) (is-even? (dec %)))]
  (is-odd? 11))
```

The specific problem of mutually recursive functions, can be solved with letfn in Clojure. A potential `letrec` macro in Clojure requires some tricks. The main complexity is to "suspend" the symbol that is not yet defined at the point of the first invocation and deliver the right expression when it's first used. One attempt at this was made by Michal Marczyk some time ago and is available as a gist [47].

Arc anaphoric macros

The Arc programming language [48] contains a copy of the basic Lisp macros which are named the same but with a starting "a" prefix (aif, acond, awhen and so on). The "a" stands for

"anaphoric": they are inspired from natural language pronouns (anaphora). Like in natural language we use "it" to refers to the same subject just mentioned in a sentence, anaphoric macros "capture" the symbol "it" for internal use in the macro:

```
(defmacro aif [expr then & [else]]
  `(let [~'it ~expr]
     (if ~'it
       ~then
       ~else)))

(aif true (println "it is" it) (println "no 'it'
(aif false (println it) (println "no 'it' here")
```

`aif` is similar to a simplified `if-let` macro that doesn't require the binding vector. The fact that `it` is injected brings two consequences:

- `aif` cannot be (easily) nested, since the `it` bindings would wrap and hide each other ambiguously.
- As any captured binding, the user might legitimately use `it` in the outer scope and thinking that it would resolve correctly inside `aif` as well:

```
(let [it 3]
  (aif true (println "it is" it)))
```

`it` has been captured from the macro and its value cannot be 3 during println.

See also

- let is the generic version of `if-let`, assigning the local binding unconditionally.
- if and when are the basic conditionals upon which `if-let` and `when-let` are based. If there is no need for locally bound variables, you can can directly use those.

3.1.3 letfn and letfn*

`macro` (letfn) `special-form` (letfn*) `since 1.0`

Listing 3.5. → *Lexical Binding, Function Definition, Local Scope*

```
(letfn [fnspecs & body])
```

`letfn` is similar to the combination of let and fn. Apart from being able to declare locally scoped functions only, `letfn`

differs from let for the fact that function names are immediately available to all functions at the same time, enabling mutually recursive calls. `letfn` use should also be considered whenever a non-trivial portion of the code inside a function is sufficiently self-contained to deserve its own name but not general enough to be extracted away in the namespace. A trivial example of `letfn` would be to extract a `square` function from a map operation:

```
(letfn [(square [x] (* x x))]
  (map square (range 10)))
;; (0 1 4 9 16 25 36 49 64 81)
```

`letfn*` is instead the special form responsible for most of the feature in the more documented and widely used `letfn` and there is no particular value in using it directly.

Listing 3.6. Contract

```
(letfn [fnspec+ & <body>])

fnspec ==> (<fname> [<params>*] <exprs>?)
```

input

- "fnspec" is an argument list containing a mandatory function name, a mandatory vector of parameters (possibly empty) and an optional body for the function.
- "fname" is any valid Clojure symbol
- "params" is an optionally empty vector of arguments to the function. Destructuring is available as usual for `letfn` functions.
- "exprs" is the optional body of the function that will be executed when the function is invoked. When no body is provided it is considered an implicit `nil`.
- "body" are the optional forms that will be evaluated in the context of the local bindings generated by `letfn`.

output

`letfn` returns the result of the evaluation of "body" in the context of the local bindings created by "fnspec" or `nil` if no "body" provided.

Examples

One aspect that for `letfn` is slightly different than other lexical binding functions is that bindings are visible forward and backward in the symbol assignment chain. This allows for

mutually recursive functions (see trampoline for a detailed explanation about mutual recursion). This allows us to define a `letrec` like behavior in Clojure (`letrec` is a common form in other kind of Lisp, see the call-out section in let):

```
(letfn [(is-even? [n] (or (zero? n) #(is-odd? (d
        (is-odd? [n] (and (not (zero? n)) #(is-e
  (trampoline is-odd? 120))
;; false
```

❶ `is-odd?` is not yet defined, but `is-even?` is able to invoke it nonetheless.

❷trampoline is always a good idea to avoid stack overflow when mutual recursion is involved. trampoline is adding the necessary recur call to the invocation chain to avoid consuming the stack.

Other use cases involving `letfn` are related to self-contained bits of computation that are private to a function and would otherwise break readability when left in the middle. Have a look for example at the following `locs-xform` transducer. `top-locs` uses the transducer to return the top 10 longest functions in a matching namespace:

```
(require '[clojure.string :as s]
         '[clojure.repl :refer [source-fn]])

(defn locs-xform [match]
  (comp
    (filter (fn [ns]                        ; ❶
              (re-find
                (re-pattern match)
                (str (ns-name ns)))))
    (map ns-interns)
    (mapcat vals)
    (map meta)
    (map (fn [{:keys [ns name]}]            ; ❷
           (symbol (str ns) (str name))))
    (map (juxt identity
               (fn [sym]
                 (count
                   (s/split-lines           ; ❸
                     (or (source-fn sym) ""))))))))

(defn top-locs
  ([match] (top-locs match 10))
  ([match n]
   (->>
     (all-ns)
     (sequence (locs-xform match))   ; ❹
     (sort-by last >)
     (take n))))
```

```
(top-locs "clojure.core" 1)
;; ['clojure.core/generate-class 382]
```

❶The transducer chain starts by filtering out of a sequence of namespaces all the ones that are not matching the given name. To do so it uses re-find.

❷At some point in the transducer chain we need to transform a Var object into a fully qualified symbol (such as from `#'clojure.core/+` to `'clojure.core/+`)

❸Counting the lines of code is done by asking `clojure.repl/source-fn` to retrieve the original text of the function, splitting into lines and counting. This is a very simple approach that doesn't take into account empty lines or comments.

❹The transducer is transformed into a sequence that is then sorted by count and the last n elements are returned. Please note: the result you see here could be different when executing on a different machine depending on the content of your local classpath.

The top-down flow of the transducer returned by `locs-xform` is interrupted by horizontal indentation every time a non trivial function needs to be used. Consider also that the anonymous

function is not necessarily communicating what is producing. By moving the anonymous function in a `letfn` form, we have a better understanding about what's going on.

```
(defn locs-xform [match]
  (letfn [(matching? [ns]   ; ❶
            (re-find
              (re-pattern match)
              (str (ns-name ns))))
          (var->sym [{:keys [ns name]}]
            (symbol (str ns) (str name)))
          (count-lines [fsym]
            (count
              (clojure.string/split-lines
                (or (clojure.repl/source-fn fsym
    (comp                     ; ❷
      (filter matching?)
      (map ns-interns)
      (mapcat vals)
      (map meta)
      (map var->sym)
      (map (juxt identity count-lines)))))
```

❶The 3 functions have now a name inside `letfn`.

❷We can read again the transducer top to bottom without too much cluttering.

As you can see in this second version, the transducer chain inside comp almost reads like plain english:

1. filter the matching namespaces
2. Extract all the interned symbols with ns-interns
3. Just take the vals of the resulting maps
4. Extract the meta data from the related vars
5. Translate the var name into a symbol name
6. Assemble the pairs of names and their LOCs

Lisp `labels` and `flet`

`letfn` is directly inspired by Common Lisp `labels`:

```
(labels ((even? (n) (if (= n 0) t (odd? (- n 1))))
         (odd?  (n) (if (= n 0) nil (even? (- n 1)))))
  (even? 11))
```

Common Lisp also includes a slightly different macro `flet` that doesn't have a Clojure equivalent but it would be the same as Clojure let followed by fn declaration: `(let [a (fn []))`. The reason why one would use `flet` instead of `labels` is not immediate and involves shadowing of functions with the same name. Using Clojure let + fn to simulate `flet` syntax:

```
(let [a (fn [n] (* 2 n))]
  (let [a (fn [n] (+ 3 (a n)))]
```

```
    (a 2)))
;; 7
```

You can note how the inner `let` is declaring a function `a` that is both defined in the outer `let` and re-defined in the inner `let`. The second function `a` is making a call to `(a n)` that is not resulting in stack overlflow because it's not recursive. The same attempt using `letfn` would instead consume the stack because the call to `a` from the inner `letfn` would be recursive:

```
(letfn [(a [n] (* 2 n))]
  (letfn [(a [n] (+ 3 (a n)))]
  (a 2)))
;; StackOverflowError
```

See also

- let is more generic than `letfn`. With `let` you can assign local bindings to any expression not just function definitions. At the same time, `let` is unable to look ahead for other symbol definitions, preventing mutually referencing expressions (like we've seen in the first example). Prefer `letfn` when the only reason for the local binding is a function declaration, or there are mutually referencing expressions.
- trampoline should be used to invoke locally defined functions that are mutually referencing, one of the options offered by `letfn`.

Performance considerations and implementation details

=> `O(n)` **Linear (number of bindings)**

`letfn` performance implications are negligible or unimportant like all other lexical binding macros.

[43] Rock, Paper, Scissors is a very easy and popular game: en.wikipedia.org/wiki/Rock-paper-scissors
[44] This StackOverflow question summarizes the debate about the two different `let` forms in Common Lisp: stackoverflow.com/questions/554949/let-versus-let-in-common-lisp

[45] "no.disassemble" is available on Github: github.com/gtrak/no.disassemble

[46] core.async is a popular library in Clojure to model concurrent or asynchronous processes. The homepage of the project is: github.com/clojure/core.async

[47] `letrec` implementation in Clojure can be found here: gist.github.com/michalmarczyk/3c6b34b8db36e64b85c0

[48] The Arc programming language: arclanguage.github.io/ref/

3.2 BOOLEAN AND BITWISE OPERATORS

Boolean operators (in Clojure like in other languages) combine booleans in some meaningful way. Strictly speaking, a boolean expression is an expression returning either `true` or `false`. Clojure contains just the basic short-circuiting and operators (compared to Java that includes `&` and `|` variants which are not short-circuiting). The negation not operator simply inverts the boolean meaning of its single argument. A truth table [49] is provided for each of the functions in this chapter.

Depending on the definition, other operators might be included in the boolean operators group like =, >, < etc. But since "comparison" has a much deeper meaning in Clojure, an entire chapter is dedicated to comparison and identity in detail.

An important distinction to make when talking about boolean operators is the meaning of **logical true** and **logical false**. The boolean type has the only two possible values `true` and `false` and Clojure extends them to all other types. So, for instance, the integer value 1 is also considered `true` and allowed in a place where a boolean type would be expected. The following table shows some notable examples of boolean translation of other Clojure values:

Table 3.1. Common Clojure values and their boolean translation

Description	Example	Boolean Value
Number zero	`0`	`true`
Empty string	`""`	`true`
Empty list	`()`	`true`
List of nil	`(nil)`	`true`
Nil	`nil`	`false`

Clojure departs from Common Lisp in what is considered `false`: in Lisp, for example, the empty list `()` is `false` while in Clojure it `true`. In Clojure, the only value (other than `false` itself) that is evaluated to `false` is `nil`.

Clojure also contains a rich set of bitwise operators (these are just functions, but since they are often found implemented directly in hardware, we tend to call them "operators" like others belonging to the CPU instruction set). Bitwise operators are more efficient for some class of operations frequently found in computer science. We should also remember that math arithmetic is always reduced to bit manipulation inside the registers of the CPU (even when normal programming happens at a much higher level of abstraction). We are going to see how to use them in the following sections.

3.2.1 not

`function` `since 1.0`

Listing 3.7. → *Boolean logic, Complement, Boolean Negation*

```
(not [x])
```

`not` is an extremely simple function implemented (literally) as `(if x false true)`. It just inverts the boolean meaning of its single argument and can be used simply as:

```
(not true)
;; false
```

Like complement, `not` takes any kind of input (not necessarily boolean) mapping it to either `true` or `false`. Despite its simplicity, `not` has an important role improving readability and expressiveness of code and used pervasively in the standard library itself. Many functions and macros like some?, complement, if-not are implemented directly on top of `not`.

The following is not-table-of-truth:

Table 3.2. `not` table of truth

x	(not x)
true	false
false	true

Contract

input

- "x" is a single mandatory argument of any type.

output

Returns a boolean `true` or `false`.

Examples

It's common for strings to be tested to see if they are empty (zero character length) but sometimes this definition needs to be extended to space-only strings. The `clojure.string` namespace already contains blank? to test such a condition, but it's missing a complement version. In the following `pluralize` function for instance, we use `not` to prevent appending "s" to a blank string:

```
(defn pluralize [s] ; ❶
  (if (not (clojure.string/blank? s))
    (str s "s")
    s))

(pluralize "flower")
;; flowers
```

```
(pluralize "")
;; ""

(pluralize "     ")
;; "     "
```

❶ `pluralize` is a simple function that returns the plural of a word by appending "s".

When the negation of a boolean test has a strong conventional name, it might be good to extract the form and make the name explicit, like the following `weekday?` function:

```
(defn weekend? [day]
  (contains? #{"saturday" "sunday"} day))

(defn weekday? [day]
  (not (weekend? day)))  ; ❶

(weekday? "monday")
;; true

(weekend? "sunday")
;; true
```

```
  (weekend? "monday")
  ;; false
```

❶A week day is unambiguously everything outside a weekend. Instead of using `(not (weekend? day))` throughout the code, is better to just name a week day directly avoid the mental effort involved in the parsing of a negative form.

`not` in other languages

The negation operator is one of those ubiquitous constructs in programming languages. C-inspired languages (C++ or Java for instance) use the very short `!` operator to negate. The venerable Fortran uses `.NOT.` to negate a logical variable. Algol inspired languages (including Basic, Pascal, Ada, Eiffel) use the (upper-cased) `NOT` operator, as well as Haskell and Erlang (but conventionally lower-cased). APL uses the unicode U+2212 "−" to negate and J the compound dash-dot "-." .

See also

Related `not` functions and macros in the standard library are often dealing with specific cases of "negation". In general, prefer the more idiomatic use of a specific alternative (when available) instead of building the same logic on top of `not`.

- complement uses of `not` to negate the output of the function passed as argument. Use complement for the specific case of negating the output of a function, instead of the longer `(not (f))`.
- boolean can be considered the opposite of `not`, since it transforms its input into a boolean without negating it. `not` achieves the same result returning the logical opposite of its input.
- bit-not is negation for binary numbers. It negates a numeric operand by considering its binary representation and converting each 1 to 0 and vice-versa.

Performance considerations and implementation details

=> `O(1)` **constant time**

`not` operates on a single argument with minimal overhead involved. There is no interesting performance analysis relevant for this section.

3.2.2 and, or

`macro` `since 1.0`

Listing 3.8. → *Boolean logic*

```
(and
  ([])
  ([x])
  ([x & next]))

(or
  ([])
  ([x])
  ([x & next]))
```

`and` and `or` are widely used macros. They implement logic conjunction and disjunction respectively. One of the best way to illustrate the behavior of logic operators is through a truth table, where all the combinations of `true` and `false` are described [50]:

Table 3.3. `and` and `or` table of truth

p	q	(and p q)	(or p q)
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

From the table you can see that `or` is more tolerant of the presence of `false` while `and` only returns `true` when all operands are `true`. Although the table only shows `p` and `q`, Clojure allows both "and" and "or" to receive more than two arguments (see the contract section). Here's for example a typical use of `and` for conditional branching:

```
(let [probe {:temp 150 :rpm "max"}]
  (when (and (> (:temp probe) 120) ; ❶
             (= (:rpm probe) "max"))
    (println "Too hot, going protection mode.")))
```

```
  ;; Too hot, going protection mode.
```

❶ `and` and `or` are frequently seen in conditions for if and when statements.

You can also use `and` and `or` outside conditions, for example for `nil` checking. We are going to see this and other idiomatic uses in the example section below.

Contract

Both "and" and "or" accept 0 or more expressions and evaluates them left to right. `and` returns:

- `true` in the absence of arguments.
- The argument in case of a single argument (behaving like identity)
- `false` if any expression evaluates to `false`.
- `nil` if any expression evaluates to `nil`.
- The evaluation of the last expression in any other case.

`or` returns:

- `nil` in the absence of arguments.

- The argument in case of a single argument (behaving like identity)
- The evaluation of the first expression that is not `nil` or `false`
- The evaluation of the last expression in any other case.

Examples

As you can see from the contract, `and` and `or` both contains a rule that stops further evaluation of other operands when reaching specific conditions (this logic is also called "short-circuiting"). So both `and` and `or` can "walk" a chain of expressions:

- stopping at the first `false` or `nil` occurrence in the `and` case.
- stopping at the first logical `true` item in the `or` case.

Instead of concentrating on "normal" use of `and` and `or` in conditional branching, we are going to see what else they can be used for. `and` behavior can be exploited to "protect" further expressions from processing `nil`, for example when using Java interoperability. The following example shows a way to

extract the "path" portion of an absolute file name (the portion that doesn’t contain the file name itself):

```
(defn path [s]
  (let [s (and s (.trim s))]  ; ❶
    (and
      (seq s)                 ; ❷
      (subs s 0 (.lastIndexOf s "/")))))

(path "/tmp/exp/lol.txt")
;; "/tmp/exp"

(path "   ")
;; nil

(path "")
;; nil

(path nil)
;; nil
```

❶The first `and` guard enables "s" to be safely trimmed, potentially resulting in a `nil` or an empty string. This second "s" local binding will hide the one coming from the function parameter.
❷The second `and` guard prevents `subs` to execute on an empty string. `(seq coll)` is an idiomatic way to verify if a collection is empty in Clojure.

`or` can be used to provide a default value in case of `nil` expressions, for example parsing optional command line options:

```
(defn start-server [opts]
  (let [port (or (:port opts) 8080)]
    (str "starting server on localhost:" port)))

(start-server {:port 9001})
;; "starting server on localhost:9001"

(start-server {})
;; "starting server on localhost:8080"
```

Both examples illustrated in this section are very idiomatic and used very often in Clojure projects.

What about operators that are not short-circuiting?

Java `&` operates as a logical `and` operator when both operands are of boolean type (bitwise otherwise) and differs from `&&` because it always evaluates both the left-hand and the right-hand sides, even when the left-hand side is false. Same goes for the difference between `||` and `|`.

The presence of `&` and `|` in Java has mainly to do with the possibility of executing side effecting expressions independently from the result of the previous evaluations. Clojure is definitely pushing toward a more pure approach trying to isolate side effects, so it comes to no surprise that there is no such operator in Clojure.

See also

- `and` and `or` are macro-expanded in terms of nested if statements. See the implementation details further down in this chapter.
- every? can be used to check if a collection of expressions are all evaluated `true` with `(every? identity [e1 e2 e3])` instead of the not applicable `(apply and [e1 e2 e3])`
- some-> or some->> is another option to exit a processing chain in the presence of a `nil`.

Performance considerations and implementation details

=> `O(n)` **Worst case, linear in n (the number of expressions)**

`and` and `or` macros are linear in the number of arguments at compile time. Their run time form is a chain of nested if expressions which is not usually a concern during performance

analysis. Invoking macroexpand-all on `and` for example, reveals how it works:

```
(clojure.walk/macroexpand-all '(and false true t

(let* [and__4467__auto__ false]     ; ❶
  (if and__4467__auto__
    (let* [and__4467__auto__ true]
      (if and__4467__auto__         ; ❷
          true
          and__4467__auto__))
    and__4467__auto__))
```

❶ `and` expands at compile time to invoke itself on the rest of the expressions until the last one is reached.

❷ At runtime the nested if statements are executed, possibly stopping ahead of touching the bottom of the chain at the first logical false value.

As you can see from the first let* expression, the short-circuiting logic applies at run-time. So if some machine generate code was to produce `and` forms with a large enough number of expressions, they might potentially incur in a `StackOverflow` exception even in the presence of a `false` as the first condition:

```
(clojure.walk/macroexpand-all ; ❶
  `(and
     false
     ~@(take 1000 (repeat true))))

CompilerException java.lang.StackOverflowError
```

❶ We purposely create troubles for `and` by generating a compile time expression with 1000 arguments.

The scenario described above is unlikely and shouldn't be of any concern in normal applications.

3.2.3 bit-and and bit-or

`function` `since 1.0`

Note

This section also touches briefly on other related functions such as: `bit-xor`, `bit-not`, `bit-flip`, `bit-set`, `bit-shift-right`, `bit-shift-left`, `bit-and-not`, `bit-clear`, `bit-test` and `unsigned-bit-shift-right`.

Listing 3.9. → *bitwise, bit-processing*

```
(bit-not [x])

(bit-and [x y & more])
(bit-or [x y & more])
(bit-xor [x y & more])
(bit-and-not [x y & more])

(bit-clear [x n])
(bit-set [x n])
(bit-flip [x n])
(bit-test [x n])
(bit-shift-left [x n])
(bit-shift-right [x n])
(unsigned-bit-shift-right [x n])
```

Clojure provides a rich set of bitwise operators. There is no "bit-set" type in Clojure, but we can use bytes, shorts, integers or longs as bit containers:

```
(Long/toBinaryString 201) ; ❶
;; "11001001"

(Long/toBinaryString 198)
;; "11000110"
```

```
(bit-and 201 198) ; ❷
;; 192

(Long/toBinaryString ; ❸
  (bit-and 2r11001001 2r11000110))
;; "11000000"
```

❶Using `Long/toBinaryString` we can see the binary representation of a number.

❷ `bit-and` performs a boolean and operation using bits from the first and second numbers at the same position.

❸We can perform the same operation in binary representation. The bit string "11000000" is the number 192 in binary representation. We can use Clojure syntax literal for binary numbers by prefixing "Xr" to the number where "X" is the base.

bitwise operators operate on bit patterns providing a fast mean to perform certain classes of artihmetic functions. The speed gain is also a consequence of bits mapping naturally to CPU internal registers: modern hardware usually offers native bitwise operators that Clojure leverages via the JVM. One negative aspect of using bitwise operators is that they are low level and tight to a particular bit size and representation.

Contract

input

Bitwise operators can be divided into groups based on their input. Unless otherwise specified, arguments have to be of type `byte`, `short`, `int` or `long` and cannot be `nil`:

- `bit-not` takes a single argument.
- `bit-and`, `bit-or`, `bit-xor` and `bit-and-not` require at least 2 arguments up to any number.
- `bit-clear`, `bit-set`, `bit-flip`, `bit-test`, `bit-shift-left`, `bit-shift-right` and `unsigned-bit-shift-right` all take 2 arguments. The first is the numerical bit-set representation and the second is the index of a bit in the set (starting from the least significant).

notable exceptions

- `IllegalArgumentException` if the type of the argument is different
- `NullPointerException` if any argument is `nil`.

output

All bitwise operators except `bit-test` returns a `java.lang.Long` that, interpreted as binary, is the result of the related bitwise operation. `bit-test` returns a boolean `true` if the bit at index "n" is "1", `false` otherwise.

Examples

bitwise operations are normally introduced to speed up recurring arithmetic operations using the lowest number of CPU cycles [51]. `bit-and`, `bit-or`, `bit-xor`, `bit-shift-left`, `bit-shift-right` and `unsigned-bit-shift-right` are the fundamental operations on which the other are built on. We'll have a look at them first and introduce shorter forms when available.

`bit-and` takes 2 or more arguments and performs the and operation on each pair (triplet, quadruplet and so on) of corresponding bits:

```
(require '[clojure.pprint :refer [cl-format]])

(defn bin [n] ; ❶
  (cl-format nil "~2,8,'0r" n))

(bin (bit-and 2r11001001
              2r11000110
```

```
                2r01011110)) ; ❷
  ;; "01000000"
```

❶ `bin` uses cl-format to properly format binary numbers to a fixed 8 bits size. It is used here and the rest of the section for readability.

❷In this example, `bit-and` accepts more than 2 arguments. The vertical alignment helps visualizing the bit triplets involved in the operation.

We call "bit mask" a bit-set built on purpose to "mask" certain bits. Given a target bit "x", the result of performing an and operation with "1" (`true`) answers the question if "x" is `true` or `false` :

```
(def fourth-bit-set-mask 2r00001000) ; ❶

(bin (bit-and 2r11001001 fourth-bit-set-mask)) ;
;; "00001000"
```

❶ This binary number has a "1" in 4th place. When used with `bit-and` it represents a mask to answer the question "is the 4th bit set in the other argument?". We named the binary number in a definition to clarify its meaning in the following bitwise operation.

❷ With `bit-and` we can perform "masking" to check if one or more bits are set to "1". The answer in this example is that the 4th bit is indeed set to "1".

`bit-test` collapses the creation of the mask and checking for a bit into a single operation (`bit-and` is useful to perform the same operation on multiple bits at once):

```
(bit-test 2r11001001 3) ; ❶
;; true
```

❶ `bit-test` returns `true` if the bit at index 3 (0-based) is set to "1". `bit-test` internally creates the necessary mask before delegating the question to Java's bitwise `and` operation.

By flipping the bits in the masking bit-set, we achieve the effect of setting the corresponding bits to zero:

```
(def turn-4th-bit-to-zero-mask 2r11110111)

(bin (bit-and 2r11001001 turn-4th-bit-to-zero-ma
;; "11000001"
```

❶Note that the bit paired up with a "0" in the mask gets set to "0" in the result. Anything else paired with "1" remains unchanged. We can infer that `true` (or "1") is the "identity" value for `and`.

`bit-clear` achieves the same effect of setting a bit to "0" without the need to providing a masking bit-set:

```
(bin (bit-clear 2r11001001 3)) ; ❶
;; "11000001"
```

❶Using `bit-clear` to set the bit at index "3" (zero-based) to "0" (or `false`).

`bit-or` works similarly to `bit-and` by applying the boolean operation or on bit pairs, but `bit-or` masking is inverted compared to `bit-and`.

More interesting is the case of `bit-xor`. "xor" (which stands for "exclusive or") is a variation on or where, if both bits are

`true`, it results in `false` instead of `true`. The following example illustrates the effect comparing `bit-or` and `bit-xor`:

```
(map bin ((juxt bit-or bit-xor) 2r1 2r1)) ; ❶
;; ("00000001" "00000000")
```

❶We present "1" and "1" as operand to `bit-or` and `bit-xor` respectively (using juxt). This is the only case the two bitwise operator differs.

`bit-xor` is particularly useful for comparison of similar bit-set. For example we can tell that two bit-sets are the same if the result only contain "0". The result contains "1" for every bit that is different:

```
(bin (bit-xor 2r11001001
              2r11001000)) ; ❶
;;              "00000001"
```

❶The bit-set contains "0" if the corresponding bit pair was the same, "1" if they were different. In this example we can see the two bit-set differs in one place only.

`bit-xor` is also useful with masking. A mask containing "1" achieve the effects of "flipping" the bit at that position:

```
(bin (bit-xor 2r11001001
              2r00010001)) ; ❶
;;             "11011000"
```

❶ `bit-xor` with a mask where the bit in the least significant position (index 0 from the right) and the 4th bit have been inverted.

Another big class of bitwise operations is shifting. Shifting consists of pushing all bits to the right or left, discarding the least or the most significant, respectively. In Java all numerical types are signed so the most significant bit represents the sign. However, during a right shift, the sign bit is preserved and "1" introduced as padding. By preserving the sign bit, positive numbers remain positives and negative numbers remain negatives (this is also called "arithmetic shifting").

Let's start by illustrating a simple right shift on a negative number. As you can see Clojure inherits Java's semantic for bit operations, including the two's complement format to represents negatives [52]:

```
(Integer/toBinaryString -147) ; ❶
;; "11111111111111111111101101101"

(Integer/toBinaryString (bit-shift-right -147 1)
;; "11111111111111111111110110110"

(Integer/toBinaryString (bit-shift-right -147 2)
;; "11111111111111111111111011011"
```

❶ We can print binary numbers using `Integer/toBinaryString`. This is similar to use cl-format like we did at the beginning of the section, but `cl-format` preserves zeroes on the left (if any). Note that the number is expressed using the two's complement format by flipping all the bits and adding 1.

❷ `bit-shift-right` shifts -147 1 bit to the right. The most significant bit (first from left) is the sign bit which is left unchanged. The least significant bit on the right has been dropped.

❸ This time `bit-shift-right` pushes 2 bits to the right. Two "1"s are added on the left hand side and "01" was dropped from the right.

Every position shifted to the right is equivalent to dividing the number by 2. More in general, the number is divided by 2^n,

with "n" the number of shifts:

```
(bit-shift-right -144 1) ; ❶
;; -72

(bit-shift-right -144 2) ; ❷
;; -36
```

❶ `bit-shift-right` of one position is equivalent to an integer division by 2.

❷Every additional shift to the right further divides the number.

It should come without surprise that `bit-shift-left` has symmetrical effects to `bit-shift-right`. One interesting property is that every left shift corresponds to multiplying the number by $2^{\overline{n}}$ with "n" corresponding to the number of left shifts:

```
(dotimes [i 5] ; ❶
  (println [(int (* -92337811 (Math/pow 2 i)))
            (Integer/toBinaryString (bit-shift-l

;; [-92337811    11111010011111110000100101101
;; [-184675622   11111001111111100001001011011
;; [-369351244   11110011111111000010010110110
```

```
;; [-738702488   11010011111110000100101101101000
;; [-1477404976  10100111111100001001011011010000
```

❶The effect of calling `bit-shift-left` up to 4 positions for -92337811. The expression prints both the decimal and the corresponding binary number.

❷The first line printed correspond to a shift of zero positions, which is equivalent to the bit-set itself. As shift progresses, we can "0" pushed from the right, while the sign bit is preserved.

For those cases where we can ignore the sign bit (because it doesn't actually represent a sign) we can use `unsigned-bit-shift-right`:

```
(require '[clojure.pprint :refer [cl-format]])

(defn right-pad [n] ; ❶
  (cl-format nil "~64,'0d" n))

(dotimes [i 5] ; ❷
  (->> i
       (unsigned-bit-shift-right -22)
       Long/toBinaryString
       right-pad
       println))
```

```
;; 11111111111111111111111111111111111111111111111111
;; 01111111111111111111111111111111111111111111111111
;; 00111111111111111111111111111111111111111111111111
;; 00011111111111111111111111111111111111111111111111
;; 00001111111111111111111111111111111111111111111111
```

❶ `right-pad` takes care of larger 64 bits sets padding with "0" from the right.

❷ We can see the effect of shifting the number -22 right of 4 positions (the first line is the bit-set no shifting).

❸ Zeroes start to appear from the left, pushing ones to the right. By using a negative number, we make sure we can see this effect clearly, contrasting zeroes and ones on the left side.

Unsigned shift right (also known as "logical shifting") always pads with zeros from the left, independently from the presence of a sign bit. Since Clojure always returns 64 bits numbers of type `long`, we can now see the full resolution of bitwise operators. Logical shifting on negative numbers always return a positive number, as a "0" will appear as the most significant bit after padding.

Note

there is no `unsigned-bit-shift-left` because the effect would be exactly the same as `bit-shift-left`.

See also

- and-or are the common boolean operators. Unless you're interested in processing multiple operations at once, you should probably use and-or instead of bitwise operators.

Performance considerations and implementation details

=> `O(1)` **Constant time**

For all practical purposes, bitwise operation are constant time. Potential dependencies on the number of bits are mostly not relevant and hardware dependent. Depending on the specific hardware implementation, bitwise operations execute in a single or few clock cycles.

There are several known algorithms that have been translated to use bitwise operators. One of them is the algorithm to retrieve all subsets of a set, also known as "powerset". We've

seen a `powerset` function implementation while talking about hash-set and the following is a different formulation based on `bit-test`. The idea is to use an integer index up to the number of possible subsets and use the permutation of bits to select the element for the subset. For example, the following bit-set shows all possible permutations of 3 bits:

```
(dotimes [i 8] ; ❶
  (println (Integer/toBinaryString i)))

;; 0
;; 1
;; 10
;; 11
;; 100
;; 101
;; 110
;; 111
```

❶This expression shows that increasing binary numbers form all the possible combinations of bits in different position in the bit-set.

Taking advantage of this fact, we can formulate a new `bit-powerset` function that uses one for loop to iterate the bit-sets

and an inner loop to fetch the corresponding indexes from the input collection:

```
(defn bit-powerset [coll]
  (let [cnt (count coll)
        bits (Math/pow 2 cnt)] ; ❶
    (for [i (range bits)]
      (for [j (range cnt)
            :when (bit-test i j)] ; ❷
        (nth coll j)))))

(bit-powerset [1 2 3]) ; ❸
;; (() (1) (2) (1 2) (3) (1 3) (2 3) (1 2 3))
```

❶We need $2^{\overline{n}}$ bit-sets, corresponding to the number of possible combinations of the items in "coll".

❷The `:when` constraint in the `for` controls which elements from the input collection should end up in the subset.

❸`bit-powerset` returns all the combinations of the input, including the empty collection and the input itself.

The implementaiton presented in hash-set and `bit-powerset` use completely different approaches. `bit-powerset` has the advantage that combinations are generated just by incrementing integers and there is no equality involved to check if a bit is set or not:

```
(require '[criterium.core :refer [quick-bench]])
(require '[clojure.set :refer [union]])

(defn powerset [items] ; ❶
  (reduce
    (fn [s x]
      (union s (map #(conj % x) s)))
    (hash-set #{})
    items))

(let [s (vec (range 10))] (quick-bench (powerset
;; Execution time mean : 765.768984 µs

(let [s (vec (range 10))] (quick-bench (doall (b
;; Execution time mean : 48.088184 µs
```

❶This is the quite elegant solution presented in hash-set to produce a `powerset`.

❷The benchmark confirms that the solution based on `bit-set` is more than 10 times faster.

[49] en.wikipedia.org/wiki/Truth_table
[50] See the Wikipedia page related to logical connectives for more information at en.wikipedia.org/wiki/Logical_connective
[51] Please refer to the Wikipedia page at en.wikipedia.org/wiki/Bitwise_operation for an in depth overview
[52] A good overview of bitwise operations including some language implementation details is available at <<en.wikipedia.org/wiki/Bitwise_operation#Arithmetic_shift,this link on Wikipedia>>

3.3 CONDITIONAL BRANCHING

Branching is one of the most used features in programming languages and one of the most useful. The existence of branching instructions assumes the program executes following some natural flow. For imperative languages, the flow is usually top to bottom in a vertical layout (with occasional jumps to procedures written elsewhere) while for functional languages like Clojure it tends more to a left to right concatenation of functions.

One common traits of conditional forms in Clojure is that they don't follow common evaluation rules for parameters. Normally arguments are evaluated left to right and ahead of the function they are passed into. Conditional forms may alter the

normal order or not evaluate parameters at all. This is definitely a consequence of their nature of being selective about what to evaluate.

Branching has usually a negative impact on code readability, since the execution of some portion of the code is now subject to conditions that could be far away from what is currently visible. To keep branching under control, Clojure offers a few constructs and helper functions that are the subject of the following sections. More in general, functional programming helps reducing software complexity by removing or minimizing side effects between mutually exclusive branches (a common problem for imperative-style state-altering languages). One evident consequence of purity is that conditional forms always return a value.

3.3.1 if, if-not, when and when-not

`special-form (if)` `macro (if-not, when, when-not)` `since 1.0`

Listing 3.10. → *Conditional Branching, Boolean Logic*

```
(if
  ([test then])
  ([test then else]))
```

```
(if-not
  ([test then])
  ([test then else]))

(when
  [test & then])

(when-not
  [test & then])
```

`if`, `if-not`, `when` and `when-not` are at the core of conditional branching in Clojure. They are used (as in many other languages) to enable or prevent evaluation of some part of the code. The condition for evaluation is any valid Clojure expression that is used as logical `true` or `false`. `if` and `if-not` can be used to pick one of two branches, while `when` and `when-not` supports decision on a single branch. The `-not` suffix in either forms simply inverts the meaning of the condition, resulting in enhanced expressiveness when the "negative" should be given more prominence.

`if` can be used as simply as:

```
(if true :a :b)
;; :a
```

Contract

input

- "test" is a mandatory Clojure expression. After evaluation the expression produces a logical boolean value that is used to evaluate another argument.
- "then" is the first evaluable argument. Unlike normal functions this argument might not actually evaluate. It is mandatory for `if` and `if-not` and optional for `when` and `when-not`. `when` and `when-not` automatically consider "then" wrapped in a do block.
- "else" is meaningful for `if` and `if-not` (`when` and `when-not` will just treat it as additional "then" forms part of the implicit do block). When present, it evaluates when the "test" is `false` (for `if`) or when the test is `true` (for `if-not`). When not present it behaves like if a `nil` was passed: `(if false :a)` is equivalent to `(if false :a nil)`

output

Returns: the result of the evaluation of the expressions depending on the condition.

`if` evaluates:

- "then" if "test" is logical true.
- "else" if "test" is logical false.
- `nil` if "test" is logical false and "else" is not given

`if-not` evaluates:

- "then" if "test" is logical false.
- "else" if "test" is logical true.
- `nil` if "test" is logical true and "else" is not given

`when` evaluates:

- "then" if "test" is logical true.
- `nil` otherwise

`when-not` evaluates:

- "then" if "test" is logical false.
- `nil` otherwise

Examples

The toss of a coin is the typical two-way problem where one of two possibilities is returned. The following shows a simple `if` where both branches have equal probability of evaluation:

```
(defn toss []
  (if (> 0.5 (rand))                    ; ❶
    "head"
    "tail"))

(take 5 (repeatedly toss))              ; ❷
;; ("head" "head" "head" "head" "tail")
```

❶rand returns a float between 0 and 1. Asking if what was returned is above or below the mid-point of 0.5 is equivalent to a 50% chance.

❷repeatedly is a nice function to call another function continuously. We can then simulate multiple toss of a coins easily and take as many as we wish.

Another common use of conditions is in recursive algorithms to determine when it's time to stop recursion. Although this is not a general rule, for this example we want to design recursive functions so that the first branch of the condition exits the loop. The effect is that the exit branch will be immediately visible at the top of the function. The last branch should continue the recursion if conditions are not met.

A simple recursive problem is walking a tree to return how many sub-levels there are (also known as the depth of the tree).

The recursion is done by checking the type of the current element and if it's a type we can "navigate" further (such as a vector) then we recur again. If we use the vector? function as is, we'll have to recur on the first branch of the `if`. We can simply use `if-not` to keep having the exit condition at the top without using a not:

```
(def tree
  [:a 1 :b :c
   [:d
    [1 2 3 :a
     [1 2
      [1 2
       [3
        [4
         [0]]]]]    ; ❶
     [:z
      [1 2
       [1]]]
     8]]
   nil])

(defn walk [depth tree]
  (if-not (vector? tree)   ; ❷
    depth
    (map (partial walk (inc depth)) tree)))
```

```
(defn depth [tree]
  (apply max (flatten (walk 0 tree))))  ; ❸

(depth tree)
;; 8
```

❶We simulate a tree by arbitrarily nesting vectors. The most indented item is 8 levels deep.

❷We take advantage of `if-not` to enforce the fact that the first branch, when selected, means a few important facts: we reached a leaf, we return a result and we don't go into further recursion.

❸The result of mapping over a sequence using the function itself as the mapping function produces a similarly nested sequence where elements have been replaced with a count (in this case). Therefore we need to flatten and take the max.

Although not an universal rule, `when` and `when-not` presence may indicate side effects when the returned `nil` is just discarded. For example `when` is pretty common in the tear-down phase of component systems to close connections:

```
(defn start []   ; ❶
  (try
    (java.net.ServerSocket. 9393 0
```

```
        (java.net.InetAddress/getByName "localhost
      (catch Exception e
        (println "error starting the socket"))))

(defn stop [s]
  (when s    ; ❷
    (try
      (.close s)
    (catch Exception e
      (println "error closing socket")))))

(def socket (start))
(.isClosed socket)
;; false
(stop socket) ; ❸
;; nil
(.isClosed socket)
;; true
```

❶Starting a socket with Java interop is quite simple. `start` returns the newly create socket in open state.
❷ `when` is used here as a guard against a potentially `nil` socket that wasn't correctly setup during initialization. We really care about closing the socket if the socket is there, nothing otherwise.
❸The client of the side effecting `when` is not interested in knowing the results of the operation.

See also

- not is the explicit way to invert the meaning of `if` or `when`. It's unlikely you'll have to use it instead of `if-not` or `when-not`.
- cond is essentially nested if-else statements in a readable form. Use them when multiple nested conditions are necessary.
- if-let and when-let can be used when conditional branching follows a let binding and the condition happens on the symbol that was just bound.

Performance considerations and implementation details

=> `O(1)` **Compile Time**

Conditional branching is not particularly expensive. Branching forms are macros and special forms with negligible impact even at compile time.

`if` is interesting for being a special forms. It just means that there is no Clojure-side defined function or macro that needs to be invoked for `if` to be executed. `if` implementation sits in the compiler's `IfExpr` inner class. The `Parser` part closely resemble other macros in the standard library (but in this case is written in Java). The generation of the byte code is relatively simple (compared to other special forms), with the main complexity related to emit un-boxed evaluation of expressions when the type hints around the `if` expressions are available.

3.3.2 cond

`macro` `since 1.0`

Listing 3.11. → *Conditional Branching, Multiple Options Choice*

```
  (cond [& clauses])
```

`cond` iterates an argument list of condition-expression pairs, evaluating (and returning immediately) the expression that corresponds to the first logical `true` condition:

```
(let [a false
      b true]
  (cond
    a :a
    b :b          ; ❶
    :else :c))    ; ❷
;; :b
```

❶ `b` is declared as `true` in the let binding. `cond` will then return the corresponding expression, in this case the keyword `:b`.

❷ Note the last `:else :c` condition-expression pair, which will be used as a default in case no other condition matches. `:else` is a completely arbitrary "truthy" value (any other keyword or string could be used except `nil` and `false`).

`cond` behaves equivalently to a nested if construct. The previous example could be expressed as:

```
(let [a false
      b true]
  (if a :a
    (if b :b
      (if :else :c))))
;; :b
```

`cond` reads easier than the corresponding nested if since conditions and expressions are vertically aligned, quickly showing which branch belongs to which test expression. The catch-all `:else :c` last pair for example is much easier to see than the corresponding nested if where it ends up as the most nested form. Worth noticing that `:else` is conventionally used as the last condition, but any logical `true` value could be used (which in Clojure is anything other than `nil` and `false`).

Listing 3.12. Contract

```
    (cond [clauses])

    clause :=> <condition> <expression>
```

input

- "clauses" can be zero or more and will be evaluated in order.
- "clause" is a pair formed by a "condition" and an "expression".
- "condition" is any valid Clojure form.
- "expression" can be any valid Clojure form.

notable exceptions

- `IllegalArgumentException`: when the number of arguments passed is odd, implying there is at least one incomplete pair, for example `(cond (= 1 1))` would throw exception because there is no form to evaluate as the result of the `true` expression.

output

`cond` returns:

- `nil` when invoked without clauses.
- The result of the evaluation of the first expression whose condition evaluates to logical `true`.
- `nil` when all conditions are logical `false`.

Examples

`cond` is essential tool for mutually exclusive conditional branching beyond the simple case of two branches that if already covers. We could for example use `cond` to set the correct HTTP response code for a web request:

```
(defn response-code [data]
  (cond                    ; ❶
    (:error data) 500
    (not= :failure (:status data)) 200
```

```
      :else 400))

  (def good-data {:id 8498
                  :status :success
                  :payload "<tx>489ajfk</tx>"})

  (def bad-data {:id 8490
                 :error "database error"
                 :status nil
                 :payload nil})

  (response-code good-data)
  ;; 200

  (response-code bad-data)
  ;; 500
```

❶ `response-code` contains a `cond` form with 3 options. The `data` parameter is inspected for errors or failures. A default `400` options is returned if no other matches.

As a general rule of thumb, `cond` should be used for any condition requiring 3 or more branches, while it would be overkill compared to if for the frequent case of 2 branches.

`if-else` in other languages

While in other languages there is a special syntax or keyword to denote that the `else` branch of an if statement contains another if statement, Clojure leverages its Lisp heritage and solves the problem with macros. For example, Ruby has the keyword `elsif`:

```
if a > b
    print "X"
elsif a == b
    print "Y"
else
    print "Z"
end
```

Python has a similar `elif` keyword:

```
var = 100
if var == 200:
   print "1 - Got a true expression value"
elif var == 150:
   print "2 - Got a true expression value"
elif var == 100:
   print "3 - Got a true expression value"
else:
   print "4 - Got a false expression value"
```

The Ruby's `elsif` and Python's `elif` are reserved words that the compiler understands natively. By defining a macro like `cond` Clojure solves the problem of having additional

conditional branches without adding any additional complexity to the compiler.

See also

- if is still a possible solution for short "if-else" combinations but `cond` usually reads better. Prefer `cond` over 2 or more nested if statements.
- defmulti along with defmethod defines multimethods in Clojure. Consider using multimethods if the quality and number of conditions in a `cond` tends to expand frequently to handle previously unknown cases. defmulti offers a flexible polymorphic dispatch including the possibility to extend the multimethod from different namespaces (while all `cond` expressions need to be defined inside a single form).
- cond-> combines multiple conditions evaluation with the option to thread a value through the expressions. Use when, based on conditions, you also want to gradually build results.
- condp avoids some typing if the condition just repeats over different values, for example `(cond (= x 1) "a" (= x 2) "b")`.

Performance considerations and implementation details

=> `O(n)` **Linear in the number of clauses**

`cond` macro has a fairly simple implementation that evaluates the first condition before going into a stack-consuming recursion. clojure.walk/macroexpand-all executed on the example in this section shows the expected nested if:

```
(clojure.walk/macroexpand-all
  '(cond
    (:error data) 500
    (not= :failure (:status data)) 200
    :else 400))

;; (if (:error data)
;;   500
;;   (if (not= :failure (:status data))
;;     200
;;     (if :else
;;       400
;;       nil)))
```

Normal use of `cond` with just a few clauses should not be particularly relevant during performance analysis. In some cases where `cond` is part of a tight loop, there could be an option to replace it with case. However, this is not always

possible. Please visit the performance section in case to know more.

3.3.3 condp

`macro` `since 1.0`

Listing 3.13. → *Conditional Branching, Predicate Matching*

```
(condp [pred expr & clauses])
```

`condp` is another useful tool for conditional branching along with cond and case. `condp` uses the same predicate function (the "p" in the name) against different values to decide which branch to execute. Using equality as a predicate is a very common case, for example:

```
(defn op [sel] ; ❶
  (condp = sel
    "plus"  +
    "minus" -
    "mult"  *
    "div"   /))

((op "mult") 3 3)
;; 9
```

❶A simple calculator that parses operations from text.

The predicate ("=" in the example) is applied to "plus", "minus" and finally "mult" that is the first evaluation returning logical `true`, hence `*` is selected as the return value.

Listing 3.14. Contract

```
    (condp <pred> <expr> [clauses] [<default>])

    clause :=> pair || triplet

    pair    :=> <selector> <choice>
    triplet :=> <selector> :>> <f>
```

input

- "pred" is a mandatory function of 2 arguments ("selector" and "expr"). The return value is interpreted as logical boolean.
- "expr" is mandatory and can be any valid Clojure expression.
- "clauses" can be zero or more and are evaluated in order.
- "clause" can contain 2 (a "pair") or 3 items (a "triplet")

- "pair" is a "selector" followed by a "choice". Both are valid Clojure expression of any type.
- "triplet" is a "selector" followed by the symbol `:>>` and a function "f". The selector is any valid Clojure expression while "f" must take a single argument of any type and can return any type.
- "default" is any valid Clojure expression.

notable exceptions

- `condp` throws `IllegalArgumentException` when a matching clause cannot be found (in contrast with cond that would return `nil` instead) and no default provided.

output

- "default" when there is no matching clause.
- the evaluation of "choice" of the first pair-clause where `(pred selector expr)` is logical `true`.
- the evaluation of `(f (pred selector expr))` for the first triplet-clause where `(pred selector expr)` is logical `true`.

Examples

The `mime-type` function is in charge of setting the right mime-type (the media type, also known as mime-type, is used by browsers to interpret the response returned by a web server, that is ultimately a just stream of bytes) by looking at the extension of the URL given as argument. We could use `condp` to decide what mime-type to assign:

```
(defn extension [url]                          ; ❶
  (last (clojure.string/split url #"\.")))

(defn mime-type [url]
  (let [ext (extension url)]
    (condp = ext                               ; ❷
      "jpg" "image/jpeg"
      "png" "image/png"
      "bmp" "image/bmp"
      "application/octet-stream")))

(mime-type "http://example.com/image.jpg")  ; ❸
;; "image/jpeg"

(mime-type "http://example.com/binary.bin")
;; "application/octet-stream"
```

❶ `extension` is a helper function to extract the last part of the url after ".".

❷ `mime-type` passes the extension through `condp` to decide which mime-type it corresponds to. Note that a default "octect-stream" identifies a generic binary type that we couldn't recognize.

❸ The returned string is the mime-type that can be used in the response.

We could use `condp` to implement "FizzBuzz" [53]:

```
(defn fizz-buzz [n]
  (condp #(zero? (mod %2 %1)) n   ; ❶
    15 "fizzbuzz"                 ; ❷
    3 "fizz"
    5 "buzz"
    n))

(map fizz-buzz (range 1 20)) ; ❸
;; (1 2 "fizz" 4 "buzz" "fizz" 7 8 "fizz"
;; "buzz" 11 "fizz" 13 14 "fizzbuzz" 16 17 "fizz
```

❶To solve FizzBuzz, we use a predicate of two arguments. The predicate returns `true` if the numbers are multiple of each other.
❷The string "fizzbuzz" needs to appear first to avoid returning results divisible by 3 or 5 (which are both 15 divisors).
❸This FizzBuzz implementation works on natural positive numbers to retrieve results. We could use nth to isolate a single item from the results.

The last example shows how we can use `:>>`, a special keyword in `condp`, to attach actions to choices. It works the same as the basic `condp` but when the `:>>` keyword is present in the clause, the last element of the triplet is considered a function and invoked with the result of the predicate. In the following (simplified) Poker game implementation `condp` is at the core of the game decision step [54].

The first set of functions are helpers used later on to identify relevant combination of cards:

```
(def card-rank first) ; ❶
(def card-suit second)

(defn freq-by-rank [hand] ; ❷
  (->> hand
       (map card-rank)
```

```
      frequencies))

(defn sort-by-rank [hand]
  (->> hand
       (map card-rank)
       sort))

(defn max-rank [hand]
  (->> hand
       freq-by-rank
       (sort-by card-suit)
       card-suit
       card-rank))

(defn- n-of-a-kind [hand n]
  (when (->> hand
             freq-by-rank
             vals
             (some #{n}))
    hand))
```

❶ `card-rank` and `card-suit` are aliases for first and second respectively. Using aliases in this case helps readability by giving a precise meaning to an otherwise very general standard library function (thanks to Ted Schrader for suggesting this and other changes in this section).

❷ The first functions of the example are helpers arranging cards by suit (one of the 4 types) or by rank (in our example, the Jack, Queen, King and Ace have been numbered 11, 12, 13 and 14 respectively).

The next set of functions builds on top of the previous to identify winning combinations for the game of Poker. There are more, but in this example we implemented just a few to keep the example shorter:

```
(defn three-of-a-kind [hand] ; ❶
  (n-of-a-kind hand 3))

(defn four-of-a-kind [hand]
  (n-of-a-kind hand 4))

(defn straight-flush [hand] ; ❷
  (let [sorted (sort-by-rank hand)
        lower (card-rank sorted)
        expected (range lower (+ 5 lower))]
    (when (and (= sorted expected)
```

```
                  (apply = (map card-suit hand)))
        hand)))

  (defn n-of-a-kind-highest [hands] ; ❸
    (->> hands
        (sort-by max-rank)
        last))

  (defn straight-flush-highest [hands]
    (->> hands
         (filter straight-flush)
         (sort-by (comp card-rank sort-by-rank))
         card-suit))
```

❶Using `n-of-a-kind` we can create functions to identify if the hand contains 3 or 4 of the same kind of cards.

❷A straight flush requires additional logic to sort cards.

❸Functions to recognize winning combinations use the thread-last operator `->>` to combine helper functions in a meaningful way.

Finally, `condp` sits at the core of the game to determine who is the winner given a set of players sitting at the table. This is accomplished by filtering players based on different kind of winning combinations and then selecting the highest in case of tie:

```
(defn game [players]
  (sort
    (condp (comp seq filter) players  ; ❶
      straight-flush  :>> straight-flush-highest
      four-of-a-kind  :>> n-of-a-kind-highest
      three-of-a-kind :>> n-of-a-kind-highest
      (n-of-a-kind-highest players))))
```

❶ `condp` combines a filter operation with seq through comp so that if the filter returns an empty list, then it results in a `nil`.

The simplified game of Poker described here checks only 3 out of the 7 potential winning conditions for a real game, not considering a full-house or a straight. `condp` aggregates the decision logic around the following design:

- Higher ranking combinations should be checked first because as soon as we have a match (for example four of a kind) we are not interested in other lower ranking combinations.
- The predicate gives us the possibility to filter players by cards combinations and passing them through to the related clause.

- In case of players with equally ranking combinations, we pass the matching players to the clause function (through `:>>`) that sorts the combination based on a more specific ranking.
- We finally sort the resulting set of cards, so they always appear ordered in the same way.

The following games verify if the Poker game has been implemented correctly. Each card is encoded as a pair of rank-suit where clubs (♣) is ":c", diamonds (♦) is ":d", hearts (♥) is ":h" and spades (♠) is ":s":

```
(game [#{[2 :h] [2 :s] [2 :c] [2 :d] [8 :h]} ; ❶
       #{[8 :h] [1 :h] [1 :s] [1 :c] [1 :d]}
       #{[2 :h] [2 :s] [2 :d] [12 :s] [12 :h]}
       #{[5 :d] [4 :s] [7 :d] [14 :s] [14 :h]}
       #{[8 :s] [4 :c] [3 :d] [10 :s] [10 :h]}])

;; ([2 :h] [2 :s] [2 :c] [2 :d] [8 :h])     ; ❷

(game [#{[1 :h] [1 :s] [1 :c] [1 :d] [8 :h]}
       #{[4 :d] [5 :d] [6 :d] [7 :d] [8 :d]}
       #{[3 :h] [5 :h] [4 :h] [7 :h] [6 :h]}])

;; ([4 :d] [5 :d] [6 :d] [7 :d] [8 :d])      ; ❸
```

❶Games are implemented as collections of sets. Each set represents a player. We encode cards as pairs with a rank and a suit.

❷This game simulation has 4 players. The hand with a four of a kind wins.

❸This game contains two straight flushes. The one with the highest rank wins.

`condp` and `fcase`

Sometimes it's useful or interesting to know how a function in the standard library ended up being the way it is. `condp` has some documented history, since it was proposed and extended with the help of the Clojure community.

`condp` inclusion in core Clojure was discussed a long ago in the mailing list [55]. Initially it was authored by Stuart Sierra in "contrib", the old external repository for all users contributions. In "contrib" it was called `fcase` and it was the same as `condp` except missing the `:>>` facility.

The "needle" `:>>` symbol (originally `:>`) was discussed a couple of months later in another thread [56]. `:>>` is inspired by the same functionality in Scheme `cond` [57].

See also

- cond supports similar functionalities to `condp`. Use cond when you need different predicates each clause. Use `condp` if you have the same predicate or you are interested in the `:>>` form to trigger a function after a match.

- cond-> has similar intent, of selecting one or more branches (although is not short-circuiting, so it might execute multiple `true` branches). Use `cond->` when you don't need to execute the same predicate and you are interested in multiple branches execution.

Performance considerations and implementation details

=> `O(n)` **Linear in the number of clauses**

`condp` macro implementation is based on cond so the same performance considerations apply and as cond it is not usually a source of hotspots in a running application. Please refer to cond for any additional information.

3.3.4 case

`macro` `since 1.2`

Listing 3.15. → *Conditional Branching, Fast Switch*

```
  (case [expression & clauses])
```

`case` is a conditional statement which accepts a list of testing conditions to determine which branch to evaluate. On the surface, `case` is not much different from cond or condp and it can be considered part of the same family of macros:

```
  (let [n 1] ; ❶
    (case n
      0 "O"
      1 "l"
      4 "A"))

  ;; "l"
```

❶A simple example of `case` expression.

Under the surface, `case` diverge from cond for its treatment of tests expressions which are not evaluated at macro-expansion time. This means that an expression like `(inc 0)` is not

replaced with "1" as test expression. In the context of `case`, `(inc 0)` is equivalent to the set containing the symbol `inc` and the number 0:

```
(let [n 1]
  (case n
    (inc 0) "inc" ; ❶
    (dec 1) "dec" ; ❷
    :none))

;; "dec"
```

❶This branch of the `case` statements verifies if the number "1" (the current local binding of the symbol "n") is present in the set formed by "inc" and "0". The answer is `false` and the control moves forward.

❷The following branch contains the number "1" and "dec" is selected as the answer.

Compared to other conditional forms, `case` is specifically designed with performance in mind. `case` implementation compiles into the optimized "tableswitch" JVM bytecode instruction [58] that provides constant time lookup (instead of linear as in cond). `case` constant lookup time comes at the cost

of some restrictions around the test expressions which we are going to investigate in this chapter.

Listing 3.16. Contract

```
(case <expr> [clauses] [<default>])

clause :=> <test> <then>
```

input

- "expr" is mandatory and can be any valid Clojure expression.
- "clauses" are grouped into one or more pairs. If there are no clauses, there should be at least one "default" exit.
- "test" is a compile-time literal and is not evaluated at macro-expansion time. Examples of valid literals are: `:a` (keywords), `'a` (symbols), `1`, `1.0`, `1M`, `1N` (numbers), `{} #{} () []` (collection literals), "a" (strings), `\a` (chars), `1/2` (ratios), #"regex" (regexes).
- "then" is any valid Clojure form. The form is evaluated when the corresponding "test" constant is matching.
- "default" is any valid Clojure form or expression.



Warning

Note that list literals `()` in test expressions are compared for inclusion and not equivalence. See the examples below for additional information.

---

exceptions

`java.lang.IllegalArgumetnException` when:

- there is no matching "test" for the given expression and no "default" is given.
- there is a duplicate "test" constant.

output

- `case` returns the "default" if one or more clauses are present but none is matching.
- `case` returns the evaluation of "then" for the first pair-clause where `(identical? test expr)` is `true`.

Examples

Let's first clarify some aspects of the contract. `case` tests are compile time literals with implications like the following trying

to use symbols like `'alpha`, `'beta` and `'pi` for branching:

```
(case 'pi ; ❶
  'alpha  \α
  'beta   \β
  'pi     \π)

;; IllegalArgumentException: Duplicate case test

(macroexpand ''alpha) ; ❷
;; (quote alpha)

(case 'pi  ; ❸
  (quote alpha) \α
  (quote beta)  \β
  (quote pi)    \π)

;; IllegalArgumentException: Duplicate case test

(case 'pi ; ❹
  alpha  \α
  beta   \β
  pi     \π)
;; \π
```

❶ Symbols like `'alpha` that would be evaluated as the symbol itself at the REPL, are not evaluated here. This `case` expression fails claiming that there is a "quote" symbol somewhere that we don't see immediately.

❷ `case` sees the quoted version of `'alpha` at macro expansion time, which is equivalent to "double-quoting" the symbol at the REPL like shown here.

❸ If we replace the single quote char `'` using the full call `(quote)` instead, we can see what is wrong. The symbol "quote" is appearing inside all tests expressions on the left hand side, resulting in ambiguous multiple matching branches. Also note that the list `(quote alpha)`, is testing for the presence of "'pi" inside the set formed by "quote" and "alpha" and it's not a proper `clojure.lang.PersistentList` instance.

❹ The correct way to match against symbols is to completely remove the single quote from test constants.

You should take particular care using `case` with test expressions other than numbers, strings and keywords. The special cases to remember are:

- Expressions containing reader macros are compared ahead of their expansion. We saw the example of a single quoted symbol, but other common cases are var literal `#'` or deref literal `@`.
- List literals are compared for inclusion rather than equivalence (see example below).
- Other collection literals, such as vectors, sets and maps are compared using normal equality.

`case` compares list literals by checking if they contain the test expression. We can take advantage of list literals to enumerate matching operators in the following infix calculator:

```
(defn error [& args]
  (println "Unrecognized operator for" args))

(defn operator [op] ; ❶
  (case op
    ("+" "plus" "sum") +
    ("-" "minus" "subtract") -
    ("*" "x" "times") *
    ("/" "÷" "divide") /
    error))

(defn execute [arg1 op arg2] ; ❷
  ((operator op)
```

```
          (Integer/valueOf arg1)
          (Integer/valueOf arg2)))

(defn calculator [s] ; ❸
  (let [[arg1 op arg2] (clojure.string/split s #
    (execute arg1 op arg2)))

(calculator "10 ÷ 5") ; ❹
;; 2
```

❶ `operator` translates an operator as string into the corresponding Clojure function. We can use `case` to select between operations or an error function to handle unrecognized operators. Note how we can add multiple synonyms for the four basic operations using a list literal.

❷ `execute` takes the operator and operands and evaluate the corresponding operation once it has been translated by the `case` statement.

❸ `calculator` takes the raw unevaluated string and converts it into "tokens" ready for evaluation.

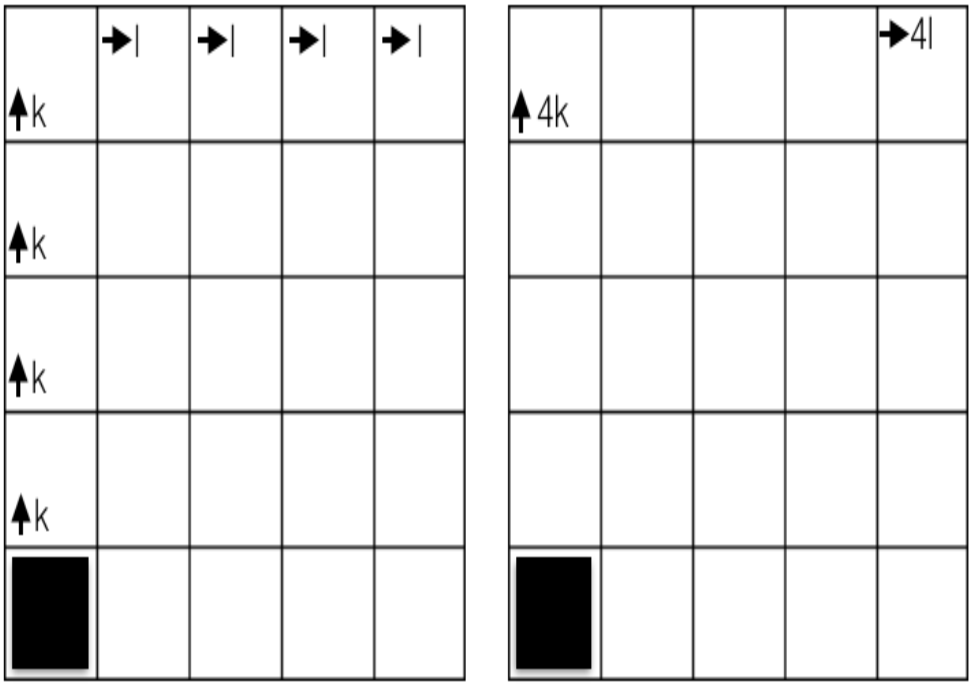
❹Invoking the calculator produces the expected results.

Considering lists have a special meaning for `case`, we are apparently in trouble if we want to compare lists as actual collections. Clojure equality does not distinguish between lists

and vectors as container types, but only compares their contents allowing us to match against lists. We are going to see how in the following example designed to score effectiveness of Vim users at the keyboard [59]

Vim is a popular editor that leverage short mnemonics key sequences to execute arbitrarily complex tasks. We could score an user based on the best key combination to achieve some editing task (usually the shortest amount of keystrokes wins). For simplicity we are going to consider the very simple task of moving the cursor from the lower-left corner of a 5x5 grid terminal to the upper right corner, like shown in the picture below:

Figure 3.2. Visually representing Vim keystrokes movement to move from one corner to the other.



The letter "k" moves up the cursor while the letter "l" moves it to the right. One poor solution would be to hit "k" four times followed by hitting "l" four more times (diagram on the left): in this case we are going to acknowledge the accomplishment but giving a low score of "5". A better solution would be to press "4" followed by the moving letter, halving the number of keystroke compared to the previous solution (picture on the right). The

code to score such a result could be implemented as the following `case` statement:

```
(defn score [ks]
  (case ks ; ❶
    [\k \k \k \k \l \l \l \l] 5
    [\4 \k \4 \l] 10
    0))

(defn check [s]
  (score (seq s))) ; ❷

(check "kl")
;; 0

(check "kkkkllll")
;; 5

(check "4k4l")
;; 10
```

❶We group the movement constants in a vector, each vector representing one test expresison in the `case` statement. Note that `case` does not consider the presence of the letter "k" or "l" in multiple vectors as duplication (it would be an exception if we used list literals).
❷Since the input is a string, we just need to call seq on it to transforms it into a sequence of characters.

There are a couple of things to note about the previous example:

- There is implicit ordering of keystrokes determined by the vector ordering. Clojure Set could be used as a test expression where ordering doesn't matter.
- `score` is invoked by passing a sequence as argument. `case` compares their content using Clojure equivalence.

"case" and table branching

`case` is implemented similarly to a well known compiler optimization used in switch statements (also called "case" or "select" statements).

The idea is the following: transform the tests constants into keys suitable for hashing and use hash-table lookup to check if there is a match. The problem then translates to transforming constants into integers. There is also another important aspect to consider: if the keys are contiguous (that is, no gaps between consecutive integers) then it's possible to enter the switch based on a simple condition to check if the

expression is or not in the allowed range. Clojure has the advantage that the JVM already provides some abstraction to build the lookup table with the `tableswitch` opcode which requires the following:

- The test values to be `int` or int-equivalents (char, bytes, shorts).
- The test values to be contiguous (potentially adding the default case label as many times as needed in between to fill the gaps).
- The total size of the switch table shouldn't be more than 8192 bytes.

The practical implications for Clojure is that there must be a way to transform compile time constants or grouping thereof into integers and shift/mask the integers to obtain the smallest possible gap in between keys. Another potential problem happens on hash-collisions and in general when transforming composites into integers. So despite the simple idea, Clojure has to do quite a lot of non-trivial processing to get it right [60]. A few fairly complicated functions (`prep-hashes` , `merge-hash-collisions` , `fits-table?` and others) are dedicated in "core.clj" to transform `case` constants into a gap-less list of non-clashing integers.

See also

- cond has a similar semantic compared to `case` . The most notable difference is the possibility to evaluate test expressions at compile time.

- condp allows to input the predicate that should be used for matching and adds the additional `:>>` semantic.
- distinct?: Clojure uses `distinct?` internally to find a suitable combination of hashes when implementing case. The example in `distinct?` shows how this work.

cond and condp are in general more flexible. As a rule of thumb, prefer `case` in the presence of literals or when performance is specifically important.

Performance considerations and implementation details

=> `O(n)` **macro expansion time**

=> `O(1)` **runtime**

The main selling point of `case` is the constant time access lookup independently from the number of test-then pairs present in the statement. We can quickly verify the claim using Criterium:

```
(require '[criterium.core :refer :all])

(defn c1 [n]
```

```
 (cond
  (= n 0) "0" (= n 1) "1"
  (= n 2) "2" (= n 3) "3"
  (= n 4) "4" (= n 5) "5"
  (= n 6) "6" (= n 7) "7"
  (= n 8) "8" (= n 9) "9"
  :default :none))

(bench (c1 9))
;; Execution time mean : 10.825367 ns

(defn c2 [n]
 (case n
  0 "0" 1 "1"
  2 "2" 3 "3"
  4 "4" 5 "5"
  6 "6" 7 "7"
  8 "8" 9 "9"
  :default))

(bench (c2 9))
;; Execution time mean : 6.716657 ns
```

As you can see the mean execution time goes from 10.825367 ns for the version using cond to the 6.716657 ns for the version using `case` which is about 40% faster. The speedup is also given by the fact that cond is using the "=" equality operator

while `case`, being based on constant literals, is implicitly using reference equality. A more "fair" benchmark could use identical?, but that would restrict the normal operational spectrum of cond with potentially surprising results:

```
(defn c1 [n]
  (case n 127 "127" 128 "128" :none))

(c1 127)
;; "127"
(c1 128)    ; ❶
;; "128"

(defn c2 [n]
  (cond (identical? n 127) "127" (identical? n 1

(c2 127)
;; "127"
(c2 128)    ; ❷
;; :none
```

❶ `case` correctly reports "128" as the correct answer

❷cond with identical? doesn't enter the expected branch because of the internal JVM caching of boxed Integers only being available up to 127 [61].

Please note that there is nothing wrong with the implementation of cond but it has more to do with the implication of using identical? as the equality operator. `case` simply avoids the additional cognitive time required to understand the implications of using identical?.

If we macroexpand a simple example, we can see how `case` delegates down to `case*` (a special form) passing down the arguments that are needed to create the necessary bytecode:

```
(macroexpand
  '(case a 0 "0" 1 "1" :default))

;; (let*
;;   [G__759 a]
;;     (case* G__759
;;       0 0 :default
;;       {0 [0 "0"], 1 [1 "1"]}
;;       :compact :int))
```

Going further down to the produced JVM bytecode, the `case*` special form produces the following (showing just the main `tableswitch` and related details):

```
(require '[no.disassemble :refer [disassemble]])
(println
```

```
   (disassemble ; ❷
     #(let [a 8] (case a 0 "0" 1 "1" :default))))

;; [...] ; ❸
 0  ldc2_w <Long 8> [12]
 3  lstore_1 [a]
 4  lload_1 [a]
 5  lstore_3 [G__22423]
 6  lload_3 [G__22423]
 7  l2i
 8  tableswitch default: 54
      case 0: 32
      case 1: 43
;; [...]
```

❶ `disassemble` is a library used in this example to de-compile the object produced by evaluating a Clojure form.

❷ We call `disassemble` on a `case` expression wrapped in a let block.

❸ The disassembled object is long on contains many other parts that are not shown here. We are only interested in showing the specific portion regarding the translation of the `case` statement in the expression. As you can see, the `case` was translated into a `tableswitch` bytecode instruction.

As a consequence of the `case` restriction about compile-time constants, the produced `tableswitch` instruction already contains all the necessary information to execute without the need of further evaluations.

[53] FizzBuzz, also popular for developers interviews, is a game to teach division to children: en.wikipedia.org/wiki/Fizz_buzz

[54] Here is a nice summary of the standard rules of Poker: en.wikipedia.org/wiki/List_of_poker_hands

[55] The thread talking about fcase and condp inclusion into the standard library: groups.google.com/forum/#!topic/clojure/3ukQvvYpYDU

[56] The thread where addition of :>> was discussed for condp: groups.google.com/d/msg/clojure/DnULBF2HAfc/1nfJS7n3BQYJ. It was proposed by Meikel Brandmeyer.

[57] `cond` documentation in Scheme is available here: docs.racket-lang.org/guide/conditionals.html

[58] To know more about "tableswitch" JVM instruction please read the following article about control flow in the Java virtual machine: www.artima.com/underthehood/flowP.html

[59] Vim is a popular text editor that thanks to editing contexts has very short key combinations. See en.wikipedia.org/wiki/Vim_(text_editor) to know more.

[60] A good selection of `case` corner cases is visible on this ticket: clojure.atlassian.net/browse/CLJ-426

[61] See www.owasp.org/index.php/Java_gotchas#Immutable_Objects_.2F_Wrapper_Class_Caching to know how Java internal caching of boxed values works

3.4 ITERATION AND LOOPS

This section explores some of the operations in Clojure dedicated to iteration and looping. Iteration is a language feature that allows repetition of the same portion of code in a controlled way. One aspect that differentiate functional from other languages is the absence (or very reduced scope) of state, a fact that reflects during iteration: Clojure promotes side-effects free loops in which input values are passed from an iteration to the other without the need of mutable variables. Let's compare the typical for-statement in Java with a loop construct in Clojure. The following is printing the square of numbers between 0 and 9:

```
Stack s = new Stack();
for (int i = 0; i < 10; i++) {
  s.push(* i i);
  // i = 9;                    ❶
}
System.out.println(s);
// [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

❶This commented out line would cut the loop short. Possible, but potentially dangerous. What if we assign `i = 8` instead?

In the Java version, the mutable variable "i" is created at the beginning of the loop and mutated at each iteration. "i" controls

the loop and we can interfere by changing it from within the loop, something difficult and explicit to achieve with Clojure. In Clojure we would pass successive values "parameters" (more properly local bindings) to the next iteration:

```
(loop [i 0 s []]
  (if (< i 10)
    (recur (inc i) (conj s (* i i)))
    s))
;; [0 1 4 9 16 25 36 49 64 81]
```

In Clojure there is no way to mutate "i" inside the body of the loop simply because "i" is not mutable. Secondly, the Java "for" statement only allows interaction with the outside world by mutation (in this case the outer-scope `java.util.Stack` object) while Clojure returns the last expression before exiting the loop. To be fair, both languages also allow the non-idiomatic alternative approach:

```
public static Stack square(int i, Stack s) {
    if (i < 10) {
        s.push(i * i);
        square(++i, s); ❶
    }
    return s;
}
```

```
System.out.println(square(0, new Stack()));
// [0 1 4 9 16 25 36 49 64 81]
```

❶Recursive `square` invocation happens as the last instruction in a mutually exclusive branch (either "i" is less than 10 or is not). This recursive computation could be transformed into iterative [62].

Java doesn't push strongly on recursion because of the compiler lacks automatic tail-call optimization capabilities (see the tail-recursion section in loop for a detailed explanation). Any sufficiently large recursive iteration in Java would eventually consume the entire stack, even if the recursion happens as the last instruction (like in our example). Similarly Clojure would allow the following mutating loop:

```
(let [i (atom 0) s (atom [])]       ; ❶
  (while (< @i 10)
    (swap! s #(conj % (* @i @i)))
    (swap! i inc))
  @s)
;; [0 1 4 9 16 25 36 49 64 81]
```

❶Clojure would only allow controlled mutation through an atom (or other concurrency aware primitives like references.

Like the non-idiomatic Java recursion, the above usage of while with mutating atoms significantly increases complexity of the code and is strongly discouraged (and very non-idiomatic Clojure).

Recursion is so common that it comes with a specific vocabulary:

- loop and fn function declaration family are considered the recursion "targets", the instruction where execution jumps after a recur.
- The "exit condition" is a conditional form (usually if or a cond) that decides when to exit the loop. A condition is always present (if we exclude the degenerated single-iteration and infinite-iterations cases).
- When the recursive call happens as the last instruction of the current scope, then the recursion is called "tail recursive".

3.4.1 loop, recur and loop*

`macro and special-form` `since 1.0`

Listing 3.17. → *Iteration, Recursion, Tail-Recursion*

```
(loop [bindings & body])
(recur & args)
(loop* [bindings & body])
```

`loop-recur` is the most basic recursive construct in Clojure. `loop` is one of the possible targets to resume execution, while `recur` performs the controlled "jump" to transfer control to the inner-most `loop` or fn form (including defn, defn-, fn* and anonymous function literal `#()`). In general, Clojure allows 3 ways to recur:

1. A call to the function from within the function itself. No `loop` or `recur` is used in this case. The recursive call can appear anywhere, not just as last instruction, like in this example returning the n-th element in the Fibonacci series [63]:

```
(defn fib [n]
  (if (<= 0 n 1)
    n
    (+ (fib (- n 1)) (fib (- n 2))))   ; ❶

(map fib (range 10))
;; (0 1 1 2 3 5 8 13 21 34)
```

❶Note that despite the line is the last in a mutually exclusive if branch, the first `fib` invocation is not in tail position, because there is a second `(- n 2)` evaluation that follows.

2. The `recur` special form invoked without a `loop` target will use the innermost containing function definition. Here's is the same example re-written to avoid recurring twice:

```
(defn fib [a b cnt]
  (if (zero? cnt)
    b
    (recur (+ a b) a (dec cnt)))) ; ❶

(map (partial fib 1 0) (range 10))
;; (0 1 1 2 3 5 8 13 21 34)
```

❶ `recur` target here is the top function definition `fib`, because there is no other inner-most definition and there is no `loop` instruction either.

3. The `recur` special form invoked with a `loop` target will restart computation from the innermost `loop` form. The following is probably the most effective of the 3 `fib` versions presented so far. The presence of `loop` takes care of some initialization that doesn't need to be outside the function and at the same time we are not recurring twice:

```
(defn fib [n]
  (loop [a 1 b 0 cnt n]                ; ❶
    (if (zero? cnt)
      b
      (recur (+ a b) a (dec cnt)))))

(map fib (range 10))
;; (0 1 1 2 3 5 8 13 21 34)
```

❶Note that 0 and 1 initialization parameters had to be sent in when invoking the function. They are now handled by `loop` without being exported in the function parameters.

These are the main differences between calling the function recursively and using the `recur` variants:

- The compiler ensures that if `recur` is present, it needs to be in tail position. The compiler throws exception if `recur` is in not in tail position.
- The compiler enables a special optimization when using `recur` that doesn't consume the stack (see the tail-recursion section below for a detailed explanation).

- `loop` offers additional control over local bindings without interfering with the function arguments, for example to initialize values or to add additional parameters to the recursion.
- `loop` is also the main choice for iteration when speed is important. `loop` takes care of propagating type information to avoid unnecessary boxing/unboxing steps, which is usually an important factor for fast code execution [64]. We are going to illustrate additional aspects about `loop-recur` speed further down the section.

Listing 3.18. Contract

```
    (target [binding-parameters]
      (<body>
        (recur <params>?)))

    target :=> <loop> | <defn> | <defn-> | <fn> | <
    binding-parameters :=> params | bindings
    params :=> <sym1>, <sym2> .. <sym>
    bindings :=> <bind1> <expr1>, <bind2> <expr2>
```

input

- "target" can be any of `loop`, defn, defn-, fn* or anonymous function literal `#()`. A target for `recur` must always be present, although the short `(recur)` is valid Clojure resulting in an infinite loop.
- "binding-parameters" is a vector containing bindings of symbols and their initial value.
- "params" are used with `recur`. `recur` invocation must have the same number of parameters declared by the "target" (including no parameters).
- "bindings" is used in case of recursion with `loop` as target. The bindings are a (potentially empty) vector containing an even number of elements. `recur` invocation must have the same number of bound locals, which is equivalent to the number of bindings. The "bindings" in `loop` are essentially equivalent to "bindings" in let.
- "body" contains everything regarding the computation, including `recur` as the last instruction. It needs to contain at least one condition instruction to select when to `recur` and when instead to return results.

notable exceptions

- `java.lang.UnsupportedOperationException`: Can only recur from tail position. The message of the exception explains that `recur` was used but another form will need evaluation after the recursion returns. In this case `loop-recur` cannot be used. If the algorithm cannot be re-formulated with tail-recursion, then the only available option is to use explicit recursion.

output

- The evaluation of the last non-recurring instruction in the body.

Examples

We briefly described the possibility for `recur` to use any of the macros dedicated to the creation of functions. The following is a rewrite of the Fibonacci example using fn and the function literal `()`. Apart from requiring to be invoked in a different way, they are equivalent to `recur` with defn as seen at the beginning of the chapter but they are definitely less readable. While the first example with `fn` as target is tolerable for very small function, the second example using the function literal `()` is rarely used:

```
(map
  (partial
    (fn [a b cnt] ; ❶
      (if (zero? cnt)
        b
        (recur (+ a b) a (dec cnt)))) 1 0)
  (range 10))

;; (0 1 1 2 3 5 8 13 21 34)

(map
  (partial
    #(if (zero? %3) ; ❷
      %2
      (recur (+ %1 %2) %1 (dec %3))) 1 0)
    (range 10))
;; (0 1 1 2 3 5 8 13 21 34)
```

❶fn is the target for `recur` in this example.

❷The function literal `#()` is expanded into an anoymous function declaration with three parameter, equivalent to the other forms of function declaration seen so far. The resulting form is not very readable now that the parameters `a`, `b` and `cnt` have been replaced with `%1`, `%2` and `%3` respectively.

`loop` - `recur` can also be used in cases where the iteration is not necessarily collection traversal (in that case, sequence operations like map would be an obvious choice). This book contains interesting examples of `loop` used in contexts other than collection traversal. The reader is invited to take a look at the following:

- let shows an infinite (and side-effecting) loop to collect user input for an interactive game.
- if-let shows a master-worker computational pattern where workers wait for work in an infinite `loop` - `recur`.
- clojure.zip/zipper shows how to traverse a tree with zippers, another typical use of `recur`.

The following example explores another of the good reason to use explicit recursion: speed. Let's see how `loop` - `recur` can replace an example of collection traversal when speed of execution is paramount. The Newton method to compute the square root of a number describes an algorithm where an initial guess converges to an approximate solution [65]. Assuming we don't know about the existence of `Math/sqrt`, let's implement a solution using sequences. The following approach consists of pulling from an infinite stream of

gradually improving approximations and then stop when the solution is precise enough:

```
(set! *warn-on-reflection* true)

(defn lazy-root [^double x] ; ❶
  (->> 1.                   ; ❷
       (iterate #(/ (+ (/ x %) %) 2)) ; ❸
       (filter #(< (Math/abs (- (* % %) x)) 1e-8
       first))

(defn sq-root [x] ; ❺
  (cond
    (or (zero? x) (= 1 x)) x
    (neg? x) (Double/NaN)
    :else (lazy-root x)))

(sq-root 2)
;; 1.4142135623746899
```

❶By type hinting the argument to double we make sure that Clojure makes the right call to `(Math/abs)` without requiring reflection, as *warn-on-reflection* was correctly pointing out. That alone would slow down `lazy-root` one order of magnitude.

❷Although there are more sophisticated ways to select the initial guess, 1 is reasonable enough here [66].

❸The anonymous function passed to iterate takes the current guess `%` and produce a better one by averaging `%` with `(/ x %)`. We use iterate to calculate one step and feed the result of the newly improved guess into the next iteration, effectively producing an infinite lazy sequence of guesses from which we pull as much improved guessed as reasonable.

❹We can now filter the best guess out of the increasingly better ones and take the first item. The predicate function is using the square of the guess `(* % %)` to verify how far off we are from the perfect solution. We use a very small number like `1e-8`. By making this number even smaller we can get more precise at the price of more guesses to compute.

❺The wrapper function `sq-root` just make sure special cases are accounted for. Something we don't want to allow for instance is the square root of a negative number.

`lazy-root` is sufficiently readable, idiomatic and reasonably fast. Notice how `lazy-root`, by adopting a stream-like model

for processing guesses, implicitly collects all the results: we could just remove the call to last to see all of them. This additional feature could be useful or not depending on the context, but it's implicit in the way the stream of guesses is processed on demand. Let's now concentrate on performance and check how well we are doing against Java's `Math/sqrt` (which is likely a faster rival):

```
(require '[criterium.core :refer [bench]])

(bench (lazy-root 2.))
;; Execution time mean : 590.703818 ns
;; [extended output omitted]

(bench (Math/sqrt 2.))
;; Execution time mean : 6.250582 ns
;; [extended output omitted]
```

The benchmark shows that `lazy-root` is about 100 times slower than the JDK `Math/sqrt` function. Apart from teaching us to use the JDK math functions when possible, the benchmark also shows that producing and consuming lazy sequences comes with an associated cost that can be considered a problem or not depending on the use case. For instance:

- We need to pass functions to iterate and take-while, which adds some invocation indirection. Secondly, this forces Clojure to compile for generic types, because there is no way for the compiler to know at compile time that `x` is a double without type hinting. Clojure greatly benefits from high order functions, but when searching for pure speed, `loop` recursion removes both problems.
- Each sequence processing step is generating intermediate data structures. take-while maintains a distinct collection of items compared to iterate. The `lazy-root` function contains a small processing pipeline, but the problem becomes bigger for longer chains of operations.
- Traversing a sequence (like in the case of last) is more expensive than just retain the last.

It turns out that the Newton method to calculate square roots can be easily reformulated for recursion. Indeed the recursive version is often presented in functional programming books:

```
(defn recursive-root [x]
  (loop [guess 1.]
    (if (> (Math/abs (- (* guess guess) x)) 1e-8
      (recur (/ (+ (/ x guess) guess) 2.))
```

```
        guess)))

  (bench (recursive-root 2.))
  ;; Execution time mean : 14.237573 ns
  ;; [other output omitted]
```

The `recursive-root` version reads certainly good and perform much better: it is now comparable to the Java version despite still being twice as slow. Note how:

- Anonymous functions are not necessary. The double type of the `guess` local binding is now enforced between iterations and doesn't require type hinting.
- There is no collection of guesses or results, or any intermediate ones. Only the last guess is ever passed between invocations.
- The recursive model is sufficiently easy to reason about in this case, but in general it requires some practice to create a mental model for recursion compared to other types of computations.

The example in this chapter is not suggesting to abandon powerful tools like lazy sequences, but to search for an equivalent explicit `loop` - `recur` when speed is an important factor.

Tail recursion optimization and tail-recursive languages

Tail-recursion is an interesting property of recursive code. Before talking about the related optimization, we need to refresh just enough information about sub-routines and the call stack. The concept of "procedure" (also called routine, function, or method depending on the language) is at the core of many programming paradigms. It would be difficult to think programming today without the possibility to name a specific portion of the code. Naming is fundamental tool to enable re-use and is given for granted in mainstream programming languages.

With sub-routines come the complexity related to passing parameters and returning values. The language run-time might need to remember any local scope created before the "jump" so when the procedure returns a value, that value can be used in the context existing before the call. The data structure that holds the entire context is called "execution stack" (or simply call stack) and an item in the stack is called a "frame". At some point during the program execution, each stack frame corresponds to a procedure that has not yet returned a result plus any additional contextual information about the calling site.

Conceptually, recursive calls are not different from heterogeneous calls: a new stack frame is created at each invocation, independently from the fact that the caller is calling itself or something else. But while normal call chains are driven by how the code is laid out manually, recursive calls are driven by data: they usually map around some data structure executing operation on each element until exhaustion, or like in the case of square root, until reaching some wanted precision. The room available for creating frames is limited by the amount of memory available and recursion can easily consume all the space available (the dreaded `StackOverflow` exception).

Tail-recursion is important because when a recursive call is the last instruction of a repeating set, there is no need to remember the state of the function at that point in time: in this situation there is no need to create a frame as no other instruction would benefit from remembering the execution state at that point in time. Advanced compilers (Scheme being a notable example) are able to automatically recognize the presence of a recursive call in tail position and prevent the stack-based propagation. The compiler can then treat the sequence of repeating instructions as if there was a "jump" or "goto" instruction as the last call in the procedure, without any stack creation and just the current value as parameter.

Clojure doesn't offer automatic tail-recursive optimization, but can optimize tail recursion with the `loop` - `recur` construct. It would be relatively simple to have an automatic way to detect tail-call optimizable code, but Clojure prefers to rely on Java semantic for method calls and Java doesn't implement tail-call optimization [67].

See also

- trampoline handles the case of mutual recursion, something that loop-recur is not designed for. Interestingly it implements mutual recursion in a straightforward way based on loop-recur.
- while performs side-effect based iterative code. It is there specifically to handle those (mostly Java-interop) cases where side-effects are necessary to manage the exit condition. It should be used sparingly.

- for is the Clojure list comprehension form. for is very useful for generating potentially complicated sequences to drive further processing. If we consider recursion as an algorithmic recipe composed by argument passing and argument processing, for represents the sequence of parameters as they are passed over time, while other sequence functions perform the actual computation. Both models have advantages in different situations, with loop-recur being generally lower level and better performing.

Performance considerations and implementation details

=> `O(n)` **Linear in the number of iterations**

The performance impact of explicit recursion with `loop` - `recur` has been the focus of this chapter. In terms of the `loop` - `recur` construct itself, it has a constant time cost at compile time (the actual effort of producing the relevant bytecode) and linear in the number of iterations it needs to perform.

As we have seen in the examples, `loop` is smart enough to recognize and maintain primitive types declared within the

bindings of the loop. Let's disassemble a small snippet to see what happens:

```
(require '[no.disassemble :refer [disassemble]])

(println (disassemble (fn [n] (loop [i 0] (< i n

// Compiled from form-init72854.clj (version 1.5
// some details removed for clarity
public final class user$eval444$fn__445 extends

public java.lang.Object invoke(java.lang.Object
 0  lconst_0
 1  lstore_2 [i]
 2  lload_2 [i]
 3  aload_1 [n]
 4  invokestatic clojure.lang.Numbers.lt(long, ja
 7  pop
 8  lload_2 [i]
 9  invokestatic clojure.lang.Numbers.inc(long)
15  areturn
}
```

❶The disassemble library can be found here: github.com/gtrack/no.disassemble

❷The function presented here is just to illustrate the types. It's not actually meaningful and it doesn't need to be.

❸Clojure knows what type `i` is thanks to the loop bindings. It doesn't know what type `n` is, so it needs to invoke the generic `Numbers.lt` which will need to cast it to a number in turn.

❹No problems incrementing `i` as a `long` thanks to the `loop` construct.

But `loop` type recognition wouldn't be exploited to its full potential without adding types to bindings that are not automatically recognized by Clojure. We just need to add the necessary hint:

```
(println (disassemble (fn [^long n] (loop [i 0]

// Compiled from form-init789662854.clj (version
// some details removed for clarity
public final class user$eval448$fn__449 extends

public final java.lang.Object invokePrim(long n)
 0  lconst_0
 1  lstore_3 [i]
 2  lload_3 [i]
 3  lload_1 [n]
```

```
   4  invokestatic clojure.lang.Numbers.lt(long, l
   7  pop
   8  lload_3 [i]
   9  invokestatic clojure.lang.Numbers.inc(long)
  15  areturn
  }
```

❶Clojure is now producing the perfect call with primitive types which doesn't incur in any casting or boxing penalties.

If you remember our `lazy-root` function from the examples, we didn't add type hinting of `x` in the function arguments. The reason is that despite the type hint would produce a better performing byte code, the kind of operations performed in the loop outweighs the optimization. The only way to know this is by always being consistent by measuring with tools like Criterium before taking any decision.

3.4.2 range

`function` `since 1.0`

Listing 3.19. → *List comprehension, Integer generation*

```
(defn range
  ([])
```

```
    ([end])
    ([start end])
    ([start end step]))
```

`range` is a general purpose number generator with many practical applications. One of the most used arities is the one with a single argument producing a sequence of integers:

```
(range 10)
;; (0 1 2 3 4 5 6 7 8 9)
```

The main use case for `range` is to provide a sequence of numbers that can be used by other sequence processing operations to create more complex behavior.

Contract

input

- "end" is the number delimiting when the generated sequence should stop (it stops at "end" minus 1). `(number? end)` must return true. When "end" is not given (no arguments) it defaults to positive infinity creating an infinite range.

- "start" is the number at which the generated sequence should start. `(number? start)` must return true. "start" defaults to 0 when not given (only "end" is present).
- "step" is the increment between each element in the sequence. `(number? step)` must return true. "step" defaults to 1 when only "start" and "end" are given.

notable exceptions

- `clojure.lang.ArityException` when more than 3 arguments are present.

output

`range` returns:

- With no arguments: an infinite sequence of monotonically growing numbers starting at 0.
- With a single and positive "end" argument: a monotonically growing sequence starting at 0 and ending at `(- end 1)`. Empty list when "end" is 0 or negative.

- With a "start" and "end" arguments and "start" smaller than "end": a monotonically growing sequence starting at "start" and ending at `(- end 1)`. Empty list when `(>= start end)`.
- With a "start", "end" and "step" arguments: a monotonically increasing/decreasing sequence of numbers by "step". Empty list when: `(>= start end)` with a positive "step" or `(<= start end)` with a negative step.

`range` returns a list of "numbers" following rules for arbitrary precisions operators, including auto-promoting longs into bigint on crossing the long type boundary:

```
(range (dec Long/MAX_VALUE) (+' Long/MAX_VALUE 3
;; (9223372036854775806
;;  9223372036854775807
;;  9223372036854775808N ; ❶
;;  9223372036854775809N)
```

❶The resulting sequence has items promoted from long to bigint automatically.

Examples

Let's start with some interesting ways of using `range` before going into more complex problems. A list of even numbers can be easily obtained using the "step" parameter:

```
(range 0 20 2) ; ❶
;; (0 2 4 6 8 10 12 14 16 18)
```

❶A step of 2 starting from 0 produces even numbers.

Negative ranges can be produced using negative steps and using negative numbers as extremes:

```
(range -1 -20 -2) ; ❶
;; (-1 -3 -5 -7 -9 -11 -13 -15 -17 -19)
```

❶Negative odd numbers sequence using a negative "end" and a negative "step" to decrease from the bigger "start".

Worth remembering that `range` works with any kind of number. To work with other numerical types it might be necessary to remember the rules of type conversion for addition. In general, having the "step" or the "end" of a specific type triggers an output sequence with that type:

```
(range 0.5 5 0.5) ; ❶
;; (0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5)
(range 1 0 -1/10) ; ❷
;; (1 9/10 4/5 7/10 3/5 1/2 2/5 3/10 1/5 1/10)
```

❶The decimal step produces 0.5 increments in the range.
❷Another example of sequence using the ratio type.

By combining `range` and map we can obtain some interesting behavior, for example a sequence of other sequences gradually increasing in size:

```
(map range (range 10)) ; ❶

;; (()
;;  (0)
;;  (0 1)
;;  (0 1 2)
;;  (0 1 2 3)
;;  (0 1 2 3 4)
;;  (0 1 2 3 4 5)
;;  (0 1 2 3 4 5 6)
;;  (0 1 2 3 4 5 6 7)
;;  (0 1 2 3 4 5 6 7 8))
```

❶Each application of map produces a `range` with a different "end" position.

The same concept can be expanded to create successive sequences in which the "extremes" get removed:

```
(->> (reverse (range 10))   ; ❶
     (map range (range 10)) ; ❷
     (remove empty?))       ; ❸

;; ((0 1 2 3 4 5 6 7 8)     ; ❹
;;    (1 2 3 4 5 6 7)
;;      (2 3 4 5 6)
;;        (3 4 5)
;;          (4))
```

❶The starting point is a reversed positive range.

❷The first step is to get the same range and its reverse combined into a map operation. The map function is again `range` that will receive two input parameters, each one coming from another range. The first number coming from the forward range will be the "start" while the second number coming from the reverse range will be the "end".

❸We need to prune off some of the last elements which become empty lists when the "end" of the range is bigger than the "start".

❹The output is indented for clarity, but will normally display as a single line.

Let's have a look now at how we can use `range` in practice. Many algorithms are based on some non-trivial iteration over a collection. Comparing the edges of a list is at the base of searching palindromes for example [68]. A palindrome is a sequence of letters which reads the same backward: "Was it a car or a cat I saw" is a typical example. One way to find if a string is a palindrome is to check if the middle letters are the same and then proceed outward to verify the others until we reach the end of the sequence:

```
    |<-------------------------------(  n  )------
    w   a   s   i   t   a   c   a   r   o   r   a
```

```
    |<------------(quot n 2)----------->|
                          |<-- (- idx) -->|<-- (+ id
```

The following implements the algorithm described above:

```
(require '[clojure.string :as s])

(defn palindrome? [xs cnt] ; ❶
  (let [idx (range (quot cnt 2) -1 -1)] ; ❷
    (every? #(= (nth xs %) (nth xs (- cnt % 1)))

(defn string-palindrome? [s] ; ❹
  (let [chars (some->> s
                s/lower-case
                (remove (comp s/blank? str)))]
    (palindrome? chars (count chars))))

(string-palindrome? "Was it a car or a cat I saw
;; true
```

❶ `palindrome?` is a function taking a sequence `xs` and a count of the elements in the sequence.

❷ `idx` contains indexes to access the sequence in reverse starting from half the count down to 0. We use quot to avoid conversion into a ratio type that would occur through the division operation /.

❸ We access the sequence by index with nth. Note that in case of a lazy sequence `xs`, the first `nth` call realizes roughly half of the sequence (if the collection supports chunking evaluation might stop beyond half point up to the end of the current chunk). After comparing all the symmetrical pairs with = we verify if there is any `false` with every?.

❹ `string-palindrome?` performs some initial preparation, like lower-casing letters and removing spaces. some->> guards against potential `nil` inputs.

The palindrome example presented here is one of the many ways to check if a sequence is a palindrome. Depending on problem requirements (like memory allocation, length of the sequence or probability for palindromes) other solutions based on vectors are likely to perform better (see how rseq can be used with vectors to check for palindromes for example). The performance section contains a few more considerations around `range` efficiency and laziness trade-offs.

See also

- for can be considered `range` big brother. It allows for more flexibility in selecting how the sequence should be generated. Use `range` if you need a simple numeric sequence, use for if you need to filter out elements of the sequence in a more complicated way or you need to cross multiple generating methods or different item types.
- iterate accepts a function that is called with the result of the previous computation to generate the next item. `(take 10 (iterate inc 0))` for instance is equivalent to `(range 10)` but with the added flexibility to change inc to another function.

Performance considerations and implementation details

=> `O(n)` **linear (worst case, fully consumed)**

=> `O(n)` **memory (worst case, retaining head)**

`range` creates a lazy sequence (more specifically, a Java object implementing the `clojure.lang.ISeq` interface). Laziness is a promise for computation: a consumer is needed to request elements to start the computation. This is why `(def a (range 1e20))` returns immediately when evaluated at the REPL: none

of the `1e20` iterations has performed yet. For the same reason it's possible to have an infinite range `(range)` provided it's never fully consumed.

`range` computes the sequence generating a linear behavior in the number of items requested. Memory is also consumed linearly assuming we hold the head of the sequence (worst case, constant space otherwise):

```
(let [r (range 1e7)] (first r) (last r))  ; ❶
;; 9999999

(let [r (range 1e7)] (last r) (first r))  ; ❷
;; OutOfMemoryError GC overhead limit exceeded
```

❶The example show access to the last element of a large sequence created with `range`. Since last is also the final result of evaluating the entire form, the rest of the sequence can be safely garbage collected as the sequence is processed.

❷The last operation appears before another operation to access the large sequence. As a result the sequence produces by `range` needs to remain in memory in full, creating a possible `OutOfMemoryError` (also depending on the allowed heap size).

`range` (like iterate, repeat and cycle) is implemented as a Java class and provides a specialized algorithm for reduce and related functions including transducers. To activate the fast path, you need to pay attention not to wrap `range` in a sequence generating function:

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (range 1000000)]
  (quick-bench (reduce + (map inc xs)))) ; ❶
;; Execution time mean : 42.022867 ms

(let [xs (range 1000000)]
  (quick-bench (transduce (map inc) + xs))) ; ❷
;; Execution time mean : 16.044716 ms
```

❶reduce cannot activate the `range` fast path because the `range` is wrapped in a map function. The default sequential path for `reduce` is selected instead.

❷The transformation is now part of a transducer and the `range` type is left visible for transduce that can activate the fast path. `transduce` uses `reduce` internally.

Similar considerations are valid for apply, which does not follow the fast `reduce` path. The following function `kth`

calculates the k-th coefficient of $(x - 1)^{\overline{n}}$ (part of the calculation necessary to test if a number is prime following the AKS primality test [69]). The function uses `range` to create a potentially long sequences and it has been implemented with `apply` and with `range` for comparison:

```
(defn kth [n k] ; ❶
  (/ (apply *' (range n (- n k) -1))
     (apply *' (range k 0 -1))
     (if (and (even? k) (< k n)) -1 1)))
(quick-bench (kth 820 6))
;; Execution time mean : 924.071439 ns

(defn kth [n k] ; ❷
  (/ (reduce *' (range n (- n k) -1))
     (reduce *' (range k 0 -1))
     (if (and (even? k) (< k n)) -1 1)))
(quick-bench (kth 820 6))
;; Execution time mean : 401.906780 ns
```

❶The function uses apply twice to create the necessary coefficient.

❷ `apply` has been replaced with `reduce`. The benchmark confirms the speed improvement.

3.4.3 for

`macro` `since 1.0`

Listing 3.20. → *Iteration, Sequence Processing, Comprehension, Lazy Sequence Building*

```
  (for [seq-exprs body-expr])
```

`for` is a powerful Clojure macro. It is usually described as "list comprehension" in analogy to other languages featuring a similar construct (see the call-out inside while for additional background). "comprehension" means both "understanding" and "inclusion", rightly suggesting the use of `for` to aggregate, filter, or otherwise manipulate multiple collections into the final output. `for` helps eliminating nested map - filter compositions (which can be more complicated to follow) and also generating non-trivial sequences. `for` should not be confused with the imperative iteration construct found in other languages, since in Clojure it’s not based on any mutation mechanism.

The following example illustrates some of the features of `for` that will be further explained in the chapter:

```
(for [i (range 10)                          ; ❶
      [k v] {:a "!" :b "?" :c "$"}          ; ❷
      :let [s (str i k v)]                  ; ❸
      :while (not= :b k)                    ; ❹
      :when (odd? i)]                       ; ❺
  s)
;; ("1:a!" "3:a!" "5:a!" "7:a!" "9:a!")
```

❶"i" is declared as a local binding and will be visible further down the bindings and in the body of the `for` macro.

❷"k" and "v" are also locals demonstrating that destructuring is available over the map. While the first value of "i" is assigned, "k" and "v" will assume all the values in the map as pairs ":a !", ":b ?" and so on until all the permutations of "i" and "k v" have been formed.

❸The ":let" expression creates an additional local binding which is not based on iterating over a sequence like "i" or "k v".

❹The ":while" expression accepts a predicate that evaluates for each permutation. As soon as the predicated is `false`, the presence of `:while` stops the current iteration (in our case the "k v" iteration of local bindings against the map). In this case, the iteration will stop as soon as "k" is equal to the keyword ":b" preventing that permutation and any other following in the map to enter the final results.

❺The ":when" filter operates similarly to the ":while" filter by preventing some permutation to enter the final sequence of results. Differently from ":while" it's not going to affect other elements in the iteration after the one that makes the predicate `false`.

Listing 3.21. Contract

```
    (for [bindings] <body>)

    binding :=>
    bind-expr OR let-expr OR while-expr OR when-ex

    bind-expr   :=> [<name> <range>]
    let-expr    :=> [:let [<local> <name>]]
    when-expr   :=> [:when <predicate>]
    while-expr  :=> [:while <predicate>]
```

- At least one binding pair and a non-empty "body" are required.
- "name" in a bind expression is any valid Clojure symbol that can represent a local variable. Bound variables are visible in other binding pairs and the body, plus any following :let, :when, :while expressions.
- "range" is any Clojure expression that can be converted into a sequence, so that `(instance? clojure.lang.Seqable range)` is true.

- "let-expr", "when-expr" and "while-expr" are optional but can appear multiple times. They are not allowed at the beginning of the bindings, at least one bind-expr must appear first.

`for` returns: a `clojure.lang.LazySeq` instance, a lazy sequence of the elements collected by applying:

1. The generation rule given by the permutations of all the binding expressions containing a sequence.
2. The filtering rule given by optional `:when` and `:while` binding expressions.
3. The assembling rule dictated by the "body".

Examples

The following table collects a few notable examples of `for` focusing on some non-trivial aspects. Each rows contain an example and a description.

Table 3.4. Some interesting aspects of the "for" macro.

Description	Example
`:when` or `:while` with a dependency on multiple local bindings. Equivalent to a constraint based on a function `f(x1,x2,..xn)` of the local bindings. This is to point out the fact that constraints are flexible and can depend on multiple local bindings at once.	`(for [x (range 100)` `  y (range 10)` `  :when (= x (* y y))]` `  [y x])` `;;([0 0] [1 1] [2 4] [3 9] [4`

Use of infinite lazy sequences as input. `for` consumes the input sequences lazily, so we can use infinite sequences as input, potentially producing another infinite sequence as output.

```
(take 6
  (for [i (range)
        j (range)
        :while (< j 3)]
    (str i j)))
;; ("00" "01" "02" "10" "11" "
```

Conflicting constraints on infinite lazy sequences. When working with infinite lazy sequences you need to be careful not to create conflicting

```
;; WARNING: never ending.
(take 3
  (for [i (range)
        j ["a" "b"]
        :when (= i 1)]
    (str i j)))
```

constraints leading to never ending iterations. Here we are asking 3 elements where "i" is equal 1 but because the other binding is on a sequence of 2 letters only, "i" will switch to 2 before we are able to collect the requested third element. `for` is not aware of the fact that the infinite `(range)` will never contain the value 1 again!

Use of multiple `:while` expressions. It is possible to constraint input sequences with independent `:while` bindings. Each `:while` is affecting the immediately preceding binding expression by stopping the current iteration when the predicate becomes `false`.

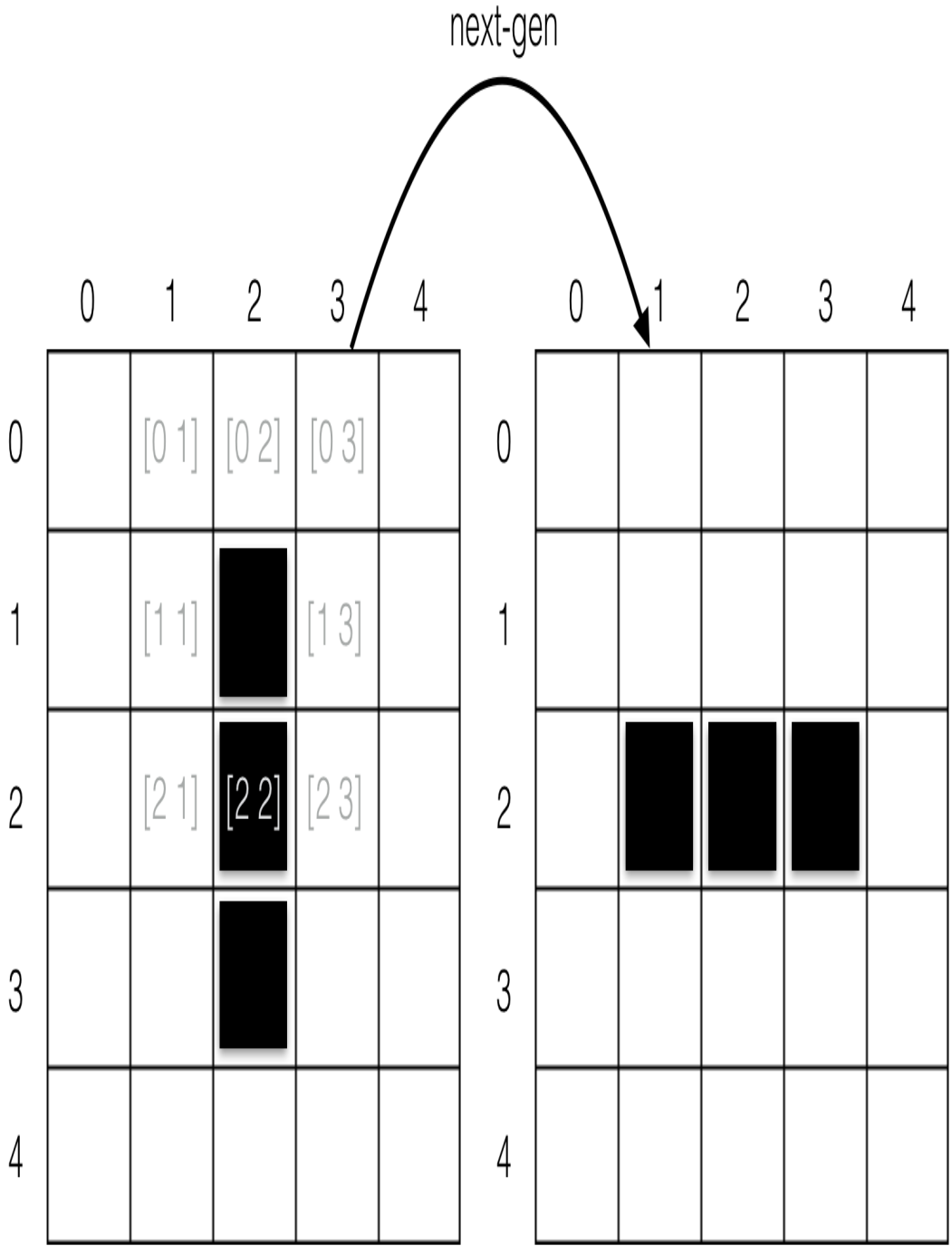
```
(for [x (range) :while (< x 4)
      y (range) :while (<= y x
  (+ x y))
;; (0 1 2 2 3 4 3 4 5 6)
```

The following is instead a complete example making good use of the `for` macro. Conway's Game of Life is a classic example

of cellular automaton [70]. Cellular automata are mathematical models exhibiting a high degree of complexity generated in turn by very simple rules [71]. They are usually visualized as two-dimensional grids where elements in the grid are cells and rules are applied to define interaction between them, as can be seen in the following picture.

Figure 3.3. A 5x5 Game of Life grid, with a "blinker" presented at initial state and after one iteration. The numbers [x y] in square brackets are the coordinates of the neighbors of cells [1 2].




next-gen
0 1 2 3 4
0 1 2 3 4
0
1
2
3
4
[0 1]
[0 2]
[0 3]
[1 1]
[1 3]
[2 1]
[2 2]
[2 3]

Passing of time can be implemented as a discrete "tick" during which rules are applied transforming cells from dead (uncolored white square) to alive (black square) or vice-versa. There is some analogy between the 4 rules governing the Game of Life and some society of living organisms (hence the name):

1. Health: any live cell with two or three live neighbors lives on to the next generation.
2. Reproduction: any dead cell with exactly three live neighbors becomes a live cell.
3. Underpopulation: any live cell with fewer than two live neighbors dies.
4. Overpopulation: any live cell with more than three live neighbors dies.

The following example shows how the Game of Life could be implemented in Clojure. Since `for` can easily create non-trivial sequences, we can use to "navigate" the grid and defines the meaning of being a neighbor:

```
(defn count-neighbors [h w x y cells]
  (->> (for [dx [-1 0 1]
             dy [-1 0 1]
             :let [x' (+ x dx)
                   y' (+ y dy)]
             :when (and (not (= dx dy 0))
```

```
                          (<= 0 x' (dec w))
                          (<= 0 y' (dec h)))] [x' y
        (filter cells)
        count))

(defn under? [n alive?] (and (< n 2) alive?))
(defn healthy? [n alive?] (or (and alive? (= n 2
(defn over? [n alive?] (and (> n 3) alive?))
(defn reproduce? [n alive?] (and (= n 3) (not al

(defn apply-rules [h w x y cells]
  (let [n (count-neighbors h w x y cells)
        alive (contains? cells [x y])
        should-live (or (healthy? n alive) (repr
        should-die (or (under? n alive) (over? n
    (and should-live (not should-die))))

(defn next-gen [h w cells]
  (into #{}
        (for [x (range 0 w)
              y (range 0 h)
              :when (apply-rules h w x y cells)]

;; testing a blinker:
(next-gen 5 5 #{[2 1] [2 2] [2 3]})
;; #{[1 2] [2 2] [3 2]}
```

```
(next-gen 5 5 (next-gen 5 5 #{[2 1] [2 2] [2 3]}
;; #{[2 1] [2 2] [2 3]}
```

❶ The first application of `for` is used to count the neighbors of a [x y] cell. In a two-dimensional system where cells are identified by x and y coordinates (like our case), the problem of finding neighbors is about moving the coordinates up-down, left-right and diagonals (by incrementing and decrementing in turn). The two increments `dx` and `dy` are the ranges of permutations we need.

❷ The `:let` expression inside the `for` macro help us defining temporary locals available inside the loop without them being necessarily part of the value comprehension (like is happening instead for `dx` and `dy`). In the `:let` we define the cell found by incrementing or decrementing the given cell `[x y]`.

❸ `:when` defines a filter for the comprehension. In our case we don't want the `[x y]` cell itself and we don't want cells outside the grid either.

❹ The application of all the rules combined happens inside the `apply-rules` function, which essentially operates on boolean logic. This result will be used later in the last `for` macro to keep or remove cells we don't want in the final computation.

❺ This last `for` generates all the possible cell coordinates for a grid of size w,h. Assuming the presence of a cell pair of coordinates indicates the cell is alive, our job is to remove all

those cells that are not going to live the next generation. At the same time, we want other cells to become alive if they weren't based on the rules of the game. The filter is achieved with another `:when` expression that just delegates to `apply-rules`.

See also

- while. If `for` is the functional way of iterating without mutable variables, while is offered for those cases where side effects are needed to control the loop. Use while for Java interoperability, especially when the Java code is in some external library that you can't control requiring explicit use of side-effects to control the loop.
- range is all you need if the comprehension should produce a monotonically increasing sequence of integers.
- map, filter and take-while are somewhat comparable to the `:let`, `:when` and `:while` expressions inside `for`. Prefer `for` when the input is coming from multiple collections and should result in a permutation of elements coming from both, avoiding nested map expressions. If we were to generate the string encoding for common poker cards, the following:

```
(for [i (range 1 14)
      a ["D" "C" "H" "S"]
      :let [card (str i "-" a)]]
  card)
```

Should be preferred to the equivalent but less easy to read nested maps version:

```
(mapcat
  (fn [i]
    (map
      (fn [a] (str i "-" a))
  ["D" "C" "H" "S"]))
(range 1 14))
```

Performance considerations and implementation details

=> `O(n)` **linear compile-time, with n number of bindings**

=> $O(n^c)$ **polynomial run-time with c bindings of n elements each**

`for` computational complexity is mainly a run-time aspect. In most cases there is nothing to be worried about in terms of performance impact. The result of processing the output of a `for` macro has the same performance profile of using any

other lazy-sequence (see lazy-seq for an in-depth analysis of the performance implication of using lazy-sequences) and depends exponentially on the number of items produced. So, for example, the following `for` macro has 5 binding expressions each one containing a range of 10 elements:

```
(for [a1 (range 10)
      a2 (range 10)
      a3 (range 10)
      a4 (range 10)
      a5 (range 10)]
  (+ a1 a2 a3 a4 a5))
```

The resulting lazy-sequence, if fully unrolled (for example with count), would result in "n" (number of elements in each range) to the power of "c" (number of bindings) iteration steps. `for` laziness and abundance of features might not be the optimal solution for tight loops where performance is important. In that case it might be better option to use a custom loop (or even transients).

The implementation details are mainly related to the mechanics of the creation of lazy sequences, as can be seen in the following macro-expansion of a simple form with macroexpand (code has been formatted and cleaned-up):

```
(macroexpand
  '(for [i (range 3)] i))

(let* [main-fn
       (fn recur-fn [xs]
         (lazy-seq
           (loop [xs xs]
             (when-let [xs (seq xs)]
               (if (chunked-seq? xs)
                 (let [fchunk (chunk-first xs)
                       chunk-size (int (count fc
                       chunk-buff (chunk-buffer
                   (if (loop [i (int 0)]
                         (if (< i chunk-size)
                           (let [i (.nth fchunk
                             (do (chunk-append c
                                 (recur (uncheck
                           true))
                     (chunk-cons
                       (chunk chunk-buff)
                       (recur-fn (chunk-rest xs)
                     (chunk-cons (chunk chunk-bu
                 (let [i (first xs)]
                   (cons i (recur-fn (rest xs)))
  (main-fn (range 3)))
```

❶The input sequence is iterated based on the fact that it is itself a lazy-sequence or not.
❷Chunks of the input sequence are appended using <lazy-seq> to the output lazy-sequence.

Despite not being the easiest code to follow, the main goal of `for` is to create a "chunked" lazy sequence (the default Clojure implementation of lazy sequences). The snippet is complicated by the fact that the input sequence needs to be treated differently if it is already lazy, so the internal chunks can be iterated accordingly: from this point of view, `for` can be thought as a sophisticated machine for lazy-sequence building.

3.4.4 while

`macro` `since 1.0`

Listing 3.22. → *Iteration*

```
  (while [test & body])
```

The `while` iteration macro is possibly the closest to the loop construct found in other imperative languages. `while` takes a test expression and a body and repeatedly executes the body until the expression evaluates as `false`. It follows that some

side effect needs to mutate the test expression from `true` to `false` other than the result returned by the body. The following snippet for example uses rand in the test expression to exit the `while` loop:

```
(while (> 0.5 (rand))
  (println "loop"))

;; loop
;; loop
;; nil
```

rand is impure function since the final returned value is dependent on something outside the application control (usually some operative system primitive). Usage of `while` should be restricted to a few special cases such as Java interoperability, since more idiomatic iteration forms exist in Clojure that don't require side effects (for example for to build an initial range followed by map or filter functions). Despite this, there are still a few legitimate cases to use `while` that will be illustrated in the examples.

Contract

input

- "test" is any Clojure expression yielding logical `true` or `false` as a result.
- "body" can be 0 or more Clojure forms

output

- `while` returns: the result of evaluating the "body" or `nil` in case no "body" was given or the body wasn't evaluated.

Examples

"while true" expressions in Java are quite common to create daemon threads to run a parallel task along with the main application. We could use `while` to start a never ending loop, for example to print a health-check message on the console output to monitor the good health of the application:

```
(defn forever []
  (while true
    (Thread/sleep 5000)
    (println "App running. Waiting for input..."

(defn status-thread []
  (let [t (Thread. forever)]
    (.start t)
    t))
```

```
(def t (status-thread))
;; App running. Waiting for input...
;; App running. Waiting for input...
;; App running. Waiting for input...

(.stop t)
;; nil
```

❶We can create an infinite `while` loop by using an expression that can only be `true`.

❷We sleep the current thread 5 seconds to prevent a flood of output messages.

❸Threads are created by simply using the constructor and passing the function they need to execute. The thread is then started right away.

❹The always `true` expression used in the `while` macro can only be affected from outside the body of the loop. The consequence in this case is that we need to stop the entire thread to stop the loop.

`while` is usually found with Java IO. Java IO often requires to test the status of a stream to understand when the end has been reached. The main operation of reading bytes from the stream has also the side effect of advancing a "cursor" holding the

current reading position, which is what we want to read inside the test expression. The following Clojure code computes the SHA-256 hashing [72] of a file.

Note

for Java version 9 and above, the `javax.xml.bind` package is available separately from the main JDK and needs to be added to the dependencies in the classpath. Please see the following StackOverflow question tinyurl.com/yd89g4qn for further information.

```
(import
  'java.io.File
  'javax.xml.bind.DatatypeConverter
  'java.security.MessageDigest
  'java.security.DigestInputStream)

(require '[clojure.java.io :as io])

(defn sha [file]
  (let [sha (MessageDigest/getInstance "SHA-256"
    (with-open [dis (DigestInputStream. (io/inpu
      (while (> (.read dis) -1)))
    (DatatypeConverter/printHexBinary (.digest s
```

```
(sha (File. "/etc/hosts"))
;; "04F186E74288A10E09DFBF8A88D64A1F33C0E698AAA6
```

❶We need to obtain a `MessageDigest` instance for the type of hashing we need. The `sha` instance created here hold the current state of the SHA-256 computation and can be updated at each read from the input stream reading the file.

❷The `DigestInputStream` instance is created on top of the `sha` instance. Notice that with-open is used to automatically close the stream after we finish reading from it the line below.

❸`while` is used here to keep reading from the `DigestInputStream` until it returns "-1", a pattern that is commonly used in Java. This `while` form is side-effecting in two ways: the expression becomes `false` as the state of the cursor in the file goes beyond the end of the file and finally because it has no body: the `sha` instance is updated by just reading from the input stream.

❹The computed `sha` is finally converted into readable form.

Iteration and List Comprehension

Imperative languages usually implement iteration by mutation. This is for example the typical Java `do while` loop:

```
int count = 1;
do {
    System.out.println("Count is: " + count);
    count++;
} while (count < 4);
```

The `count` mutable variable needs to be mutated by the body of the loop in order for the loop to exit at some point (here mutated using the `++` operator). Functional languages don't support (or strongly discourage) iteration using this style, preferring instead recursion or list comprehension.

Recursion is obtained with a function invoking itself (or multiple mutually recursive functions, see trampoline) passing the mutating variable as the argument of the next invocation. The following example is the re-working of the `do while` Java code into Clojure:

```
(loop [count 1]
  (when (< count 4)
    (println "Count is:" count)
  (recur (inc count))))

;; Count is: 1
;; Count is: 2
```

```
;; Count is: 3
;; nil
```

As you can see, the mutating element becomes the argument of the recur function and it's incremented every iteration. Compared to the Java code, the test expression previously inside the `while` has been translated into a when invocation in the Clojure code: a condition is always requested inside the loop-recur to exit the loop and is typical in recursive code.

A list comprehension instead, is the concatenation of many processing steps starting from an initial list of values. Comprehension can be also used to mimic iteration, but it goes beyond that formulating a new programming style. Instead of mutating or recursively changing the value to check the test expression, the sequence of all values is assembled first and the computation builds up from those. If we look at the previous example we can collect the different values of the `count` variable during each iteration like this:

```
(loop [count 1
       res []]
  (if (< count 4)
    (recur (inc count) (conj res count))
    res))
```

```
;; [1 2 3]
```

Once the values upon which the iteration should be performed are decided, we can build up the computation using sequence manipulation functions. In this case we don't need the loop-recur just to build the natural positive numbers from 1 to 4, we could use map or for:

```
(dorun (map #(println "Count is:" %) (range 1 4)
(dorun (for [i (range 1 4)] (println "Count is:"

;; Count is: 1
;; Count is: 2
;; Count is: 3
;; nil
```

Both forms produce the same output of the initial example by feeding println with an initial list of values. We could add more processing steps on top of the initial value generation, simulating the equivalent of multiple isolated loops in an imperative language. Thanks to Clojure map, for, filter, reduce (and many other functions), programming by list

comprehension results in code that is more concise and expressive than their imperative counterpart.

See also

- for is an idiomatic alternative to iteration by mutation in a functional language like Clojure. It offers a powerful syntax to generate driving values to process with sequence manipulating functions like map or filter. Prefer for instead of `while` unless mutation is an essential part of the iteration.
- loop is the low common denominator for many iteration-like forms in Clojure and is also used inside `while` implementation. loop gives greater control on the iteration, including the definition of local bindings. Use loop and recur when other parameters (which likely not side-effects) are controlling the loop and should appear as locally bound variables.

Performance considerations and implementation details

=> `O(1)` **macro expansion**

=> `O(n)` **linear in the number of loops**

`while` performance considerations are quite similar to dotimes, with usually small to non-existent correlation to the application hot-spots (this is of course not considering the content of the expression or the body that `while` merely receives as arguments).

As far as implementation details `while` is quite simple macro. See for example the macroexpansion of the rand example presented at the beginning of the chapter:

```
(macroexpand '(while (> 0.5 (rand))
  (println "loop")))

;; (loop* [] (when (> 0.5 (rand))
;;   (println "loop") (recur)))
```

The expansion reveals a basic use of the loop-recur recursion pattern with a when to verify the expression.

3.4.5 dotimes

`macro` `since 1.0`

Listing 3.23. → *Iteration, Side Effects*

```
(dotimes [bindings & body])
```

`dotimes` is used to repeat some portion of code multiple times. The form to be repeated appears as the last argument of the macro while the first argument is a binding vector that contains a local binding and the number of desired repetitions, for example:

```
(dotimes [i 3] (println i))
;; 0
;; 1
;; 2
;; nil
```

`dotimes` is part of a family of Clojure iterators designed to handle side-effecting operations (or to realize lazy-sequences) that also includes doseq, doall, dorun and run!.

Listing 3.24. Contract

```
(dotimes <bindings> body)

bindings :=> [local n]
```

- "bindings" must be a vector of two elements only.
- "body" is any number of expressions that will be evaluated zero or more times.

- "local" is the name of a local binding variable that can be used inside the "body". "local" will increment by one each iteration (except when "n" is less or equal to zero).
- "n" is normally a positive number since negative and zero prevents the evaluation of the "body".
- returns: nil.

Examples

The main use case for `dotimes` is handling side effects, as demonstrated by the `nil` result which is always returned. `dotimes` can be used at the REPL for measuring performance of Clojure code, for example:

```
(time (dotimes [_ 1000000] ; ❶
  (apply max (range 100))))

;; "Elapsed time: 1305.668357 msecs"
;; nil
```

❶A typical use of `dotimes` to repeat the execution of some code and calculate the total elapsed time.

To measure the performance of max above, the form evaluates some large number of times and the total elapsed is measured

with `time`. By using `dotimes` it's possible to quickly verify assumptions about performance before using more rigorous methods (such as Hugo Duncan's Criterium).

Outside the REPL use, `dotimes` is often connected to the execution of side effects. The locally bound variable provided with `dotimes` is a perfect fit for array access operations. The following example shows a faster version of the fizz-buzz game presented in the condp chapter:

```
(require '[criterium.core :refer [quick-bench]])

(defn fizz-buzz-for [n]                          ; ❶
  (condp #(zero? (mod %2 %1)) n
    15 "fizzbuzz"
    3 "fizz"
    5 "buzz"
    n))

(defn fizz-buzz-slow [n]                         ; ❷
  (doall (map fizz-buzz-for (range n))))

(defn fizz-buzz [n]
  (let [res (transient [])]                      ; ❸
    (dotimes [i n]
      (assoc! res i (fizz-buzz-for i)))          ; ❹
    (persistent! res)))                          ; ❺
```

```
(quick-bench (fizz-buzz-slow 1000))
;; Execution time mean : 34.320885 µs

(quick-bench (fizz-buzz 1000))
;; Execution time mean : 28.308923 µs
```

❶ `fizz-buzz-for` is the function that contains the conditional deciding if the number needs to be replaced with the corresponding string based on the divisors.

❷ `fizz-buzz-slow` is exactly the same as before, with just a doall added to realize the lazy sequence in full. Despite the claim that this version is slower, `fizz-buzz-slow` is still a very idiomatic and natural way to solve the problem and it should be considered the best solution unless raw performance is an important factor.

❸The new `fizz-buzz` function first creates an empty transient vector and uses `dotimes` to perform side effects on the indexes.

❹assoc! is used here to permanently alter the transient vector at the current index "n" of the the `dotimes` iteration.

❺The transient is finally returned as a normal persistent collection for results.

As you can see from the benchmark, there is a noticeable speed up by using a transient [73]. `dotimes` is a perfect choice to perform the side effect of adding elements to the vector, including the necessary incrementing index.

Retaining the head of a sequence

The Clojure API documentation (and Clojure literature in general) often warns about "holding the head" or "retaining the head" of a lazy sequence. The advantage of lazy sequences (the default for several data structures in Clojure) is that they don't need to be loaded into memory completely, except for the part the application is currently processing. The behavior of lazy sequences is of course important for "big data" applications where the input can be much larger than the available memory.

Clojure developers need to pay specific attention not to retain the head (or any other initial part) when processing potentially big lazy sequences. Sometimes head retention troubles can be tricky to spot [74] but a more evident example is present when using iterators like doall, which is part of the same family `dotimes` belongs to.

When doall iterates through a lazy sequence, it forces items to realize. Since doall returns the realized collection, all items in the collection are forced into memory at some point preventing garbage collection. Although doall behavior can be desirable at times, doseq dorun and `dotimes` all returns `nil` by design to avoid any memory overflow in case the iteration produces a collection.

See also

- doseq is very similar to `dotimes` but it supports extended bindings including multiple locals and destructuring. Prefer doseq when the single incrementing integer provided by `dotimes` is not sufficient.
- doall takes a sequence as input and iterates the sequence realizing its items. Use doall when the only goal of the iteration is realizing a lazy sequence.
- dorun are similar to doall but they return `nil` without holding the head of the sequence. Prefer dorun when the input is a sequence containing side-effecting items once realized.

Performance considerations and implementation details

=> `O(1)` **macro expansion**

=> `O(n)` **runtime execution**

`dotimes` macro expands into a loop-recur form which executes the "body" argument the given number of times. The macro expands in constant time and the loop adds almost nothing to the overhead of the executing body which is usually the dominant part of the algorithm. For this reason `dotimes`

usage is rarely involved in performance hot-spots. Here's for example an expanded form (lightly cleaned-up for clarity):

```
(macroexpand '(dotimes [i 3] (println i)))

;; (let* [n (long 3)]
;;   (loop [i 0]
;;     (when (< i n)
;;       (println i)
;;       (recur (unchecked-inc i)))))
```

Worth noticing that in order for the loop-recur loop to be the fastest possible, the numeric binding (the number of times to execute the iteration) is cast to a long and incremented with unchecked-inc.

[62] Perhaps one of the best explanation of recursive computation and tail-call optimization is in SICP, Structure and Interpretation of Computer Programming: mitpress.mit.edu/sites/default/files/sicp/full-text/book/book-Z-H-11.html#%_sec_1.2.1

[63] Fibonacci sequence is characterized by the fact that every number in the series is the sum of the preceding ones: en.wikipedia.org/wiki/Fibonacci_number

[64] Autoboxing is the automatic conversion of primitive types

into the corresponding wrapper class (int to Integer for example in Java). Boxing has usually a minimal cost, but a big impact in Clojure when primitive types could be used and are instead converted into their wrapping object by a function call. Without type hinting, Clojure would need to compile a function into a generic bytecode able to deal with any type of argument (e.g. using a generic `java.lang.Object`).

[65] The Newton method can be generalized for other problems not just the square root calculation. More details available on en.wikipedia.org/wiki/Newton%27s_method#Square_root_of_a_number

[66] Here's a more detailed explanation about how to pick the initial guess: math.stackexchange.com/questions/787019/what-initial-guess-is-used-for-finding-n-th-root-using-newton-raphson-method

[67] Clojure support for automatic tail-recursion has been often discussed on the mailing list. One thread that explains the rationale behind Clojure opting for loop-recur instead can be found here: groups.google.com/forum/#!msg/clojure/4bSdsbperNE/tXdcmbiv4g0J

[68] Palindromes are described very well on Wikipedia: en.wikipedia.org/wiki/Palindrome

[69] Please see en.wikipedia.org/wiki/AKS_primality_test

[70] en.wikipedia.org/wiki/Conway%27s_Game_of_Life

[71] en.wikipedia.org/wiki/Cellular_automaton

[72] SHA-256 is a very well known cryptographic and hashing function. See en.wikipedia.org/wiki/SHA-2 for the details.

[73] In the spirit of searching the best possible performance, there are other important factors to consider for the version of Fizz Buzz presented here that are not discussed in this chapter because not relevant to the discussion

[74] See for example this excellent StackOverflow answer regarding a common problem found with retaining the head processing sub-sequences: stackoverflow.com/questions/15994316/clojure-head-retention

3.5 A GLIMPSE OF COLLECTION PROCESSING

The Clojure standard library contains a large number of functions and macros dedicated to collection processing. This is not a coincidence for functional languages, where algorithms tend to be designed on top of a small group of core data structures (while in other paradigms, such as Object Oriented Programming, custom objects and their relationships also play an important part).

Collection processing is so important that functions like map or first are amongst the most used overall. We can easily see this if we query Github, the largest public repository of Clojure projects, with the name of the functions and macros from the index of this book (around 700):

Table 3.5. The top 20 most used functions/macros when

searching Clojure repositories.

Name	Frequency
ns	394490
defn	293918
require	279210
let	237654
def	172983
refer	163654
map	159781
fn	154482
str	145899
nil?	125109

use	119952
test	115419
first	98908
get	93911
true?	91826
when	91463
name	90469
string?	86492
if	85942
keys	85435

Even if this section offers a small overview of what can be done with collections (more specifically of the "sequential" type) the following subset is powerful enough to get you started:

- first are handy helpers to fetch the head, the second element or the tail of a collection.
- map is the primary way to apply transformations to the elements.
- filter yields specific elements from a collection depending on a predicate function.
- reduce can be used to converge the collection to a final result, obtained by combining a group of items in some meaningful way.

Other collection/sequences functions will be discussed further on in their dedicated chapters [75].

3.5.1 first, second and last

`function` `since 1.0`

Listing 3.25. → *Sequence processing, Iteration*

```
(first [xs])
(second [xs])
(last [xs])
```

`first`, `second` and `last` are functions taking a sequence-able collection (any Clojure collection that can be iterated using

the sequence interface) and extracting the element at the position described by their names. They can be used easily like:

```
(def numbers '(1 2 3 4))

(first numbers)
;; 1
(second numbers)
;; 2
(last numbers)
;; 4
```

`first`, `second` and `last` are part of a larger group of functions to access specific parts of a sequential collection.

Ultimately is the specific collection type to decide how to implement the sequential access operation. For example, unordered collections like sets and maps also implement `clojure.lang.Seqable`:

- hash-maps: when iterated sequentially, a map becomes a list of key-value pairs. But when fetching elements, they are not necessarily following insertion order:

```
(def a-map (hash-map :a 1 :b 2 :c 3 :d 4

(first a-map)
;; [:e 5]

(second a-map)
;; [:g 7]

(last a-map)
;; [:a 1]
```

- sets: similarly to hash-maps, they have no notion of ordering (see sorted-set for that purpose), so the same uncertainty factor applies:

```
(def a-set #{1 2 3 4 5 6 7 8 9})

(first a-set)
;; 7

(second a-set)
;; 1

(last a-set)
;; 8
```

Contract

input

`first`, `second` and `last` all accept one parameter "xs":

- "xs" should be sequential or implement the sequence interface, so that `(instance? clojure.lang.Seqable xs)` is `true`. "xs" can be nil or an empty sequence.

notable exceptions

- None. `first`, `second` and `last` all use `nil` to signal exceptional conditions.

output

- the element at the first, second or last position in the sequence, if available. If the wanted element is not existent at the desired position it returns `nil`. If the input sequence "xs" is `nil`, returns `nil`.

Examples

first

One common use case is to pass `first` as a parameter to higher-order functions. The following example shows how `first` can be used with map to extract just the first element from a small sequence. Extracting parts of a string (in this case a phone number) is a common case:

```
(def phone-numbers ["221 610-5007"
                    "221 433-4185"
                    "661 471-3948"
                    "661 653-4480"
                    "661 773-8656"
                    "555 515-0158"])
```

```
(defn unique-area-codes [numbers]
  (->> numbers
       (map #(clojure.string/split % #" "))
       (map first)                              ; ❶
       distinct))                               ; ❷

(unique-area-codes phone-numbers)
;; ("221" "661" "555")
```

❶At this point, the string containing the entire phone number has been split into two parts based on the position of the space character. We just want the area code, so we ask for the first.

❷distinct can be used to get rid of repetitions inside a sequence. We use it here to remove duplicated area codes.

Another example of `first` usage involves recursion, where it could be needed to operate on the first element before continuing with the rest. Here's for example a function `all-positives?` checking if all the given numbers are positives:

```
(defn all-positives? [coll]
  (cond
    (empty? coll) true                          ; ❶
    (pos? (first coll)) (recur (rest coll))     ; ❷
    :else false))                               ; ❸
```

```
(all-positives? (list 1 2 3))
;; true

(all-positives? (list -1 0 1))
;; false
```

❶If the collection is empty? we assume the result is true.
❷If the first element is not positive, we are done and cond returns `false`. In case the first element is positive, we need to check the others by recurring the rest of the sequence.
❸In any other case, we return `false`.

second

Extracting the second element from a sequence is frequent enough to grant a dedicated function. One reason for that is that many intermediate steps in data processing involve small lists and `second` can save a few keystrokes compared to the equivalent `(first (rest xs))`. The following example shows a sequence of temperature samples from different locations, reporting on the maximum and minimum temperatures recorded for the day. The max temperature is appearing right after the first element. We can use sort-by by the second element to extract the highest temperature like this:

```
(def temp '((60661 95.2 72.9) (38104 84.5 50.0)

(defn max-recorded [temp]
  (->> temp
       (sort-by second >)             ; ❶
       first))                        ; ❷

(max-recorded temp)
;; (60661 95.2 72.9)
```

❶sort-by takes a function and optionally a comparator to decide how to order the sequence. Here we use `second` to define which element in the triplet should we sort with. The second parameter is the comparator `>` "greater than" to sort in reverse order.

❷After sorting the sequence, we can just drop everything expect the highest temperature recorded that is now at the top.

last

Similarly to `first` and `second`, `last` can be used to fetch the last element in a sequence. The following example shows `last` in action with re-seq and regular expressions. Given a long string of commands, we want to know which user was last

set before sending the message, assuming users are set with the syntax `user:username` in the message:

```
(def message "user:root echo[b]
              user:ubuntu mount /dev/so
              user:root chmod 755 /usr/bin/pwd")

(last (re-seq #"user\:\S+" message))
;; "user:root"
```

❶re-seq returns a list of matching patterns, in this case anything in the form "user-colon-name".

`car`, `cadr` and `cdr`

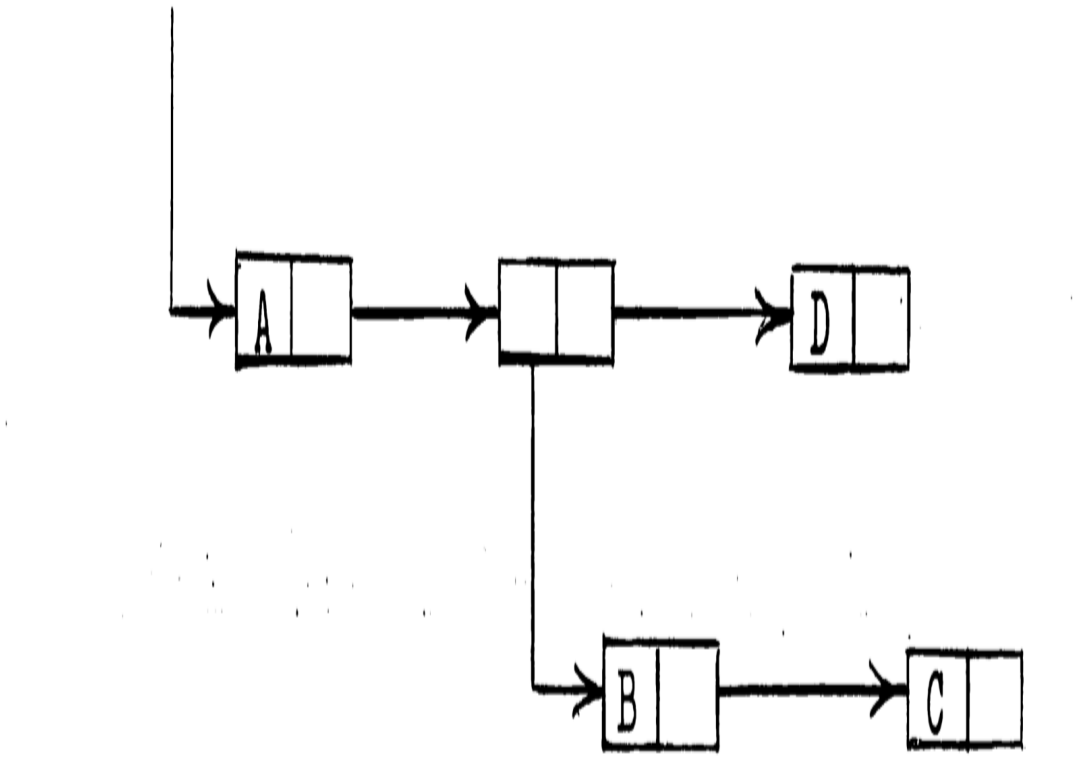
`first` and `second` bear some relationship to similar Lisp functions connected to the concept of "cons cell": `car` and `cadr` respectively. `cdr` is instead the equivalent of rest which along with `car` is the fundamental ingredient to access all parts of a list: `cadr` for example is just the contraction of `(car (cdr x))` which makes access to the second element in the list.

Although similar on the surface, the original Lisp implementation of these functions was tied to the hardware constraints of the time. On the IBM 704 on which Lisp was invented, "car" and "cdr" were register (part) names: "content address register" and "content decrement register" respectively.

The first Lisp implementations used memory "words" to allocate lists, splitting each word in two parts: the address part containing the result of `(car x)` and the decrement containing the result of `(cdr x)`. Here's a diagram showing the memory

structure of the simple list `(A, (B,C), D)` coming straight from the first Lisp Programmer's Manual:

Figure 3.4. A structure of a simple list based on cons cells.



Many of the Lisp implementations that followed continued the tradition of naming functions to access the first and last element of a list as `car` and `cdr`, even when the hardware didn't have such registers anymore. Nowadays, Common Lisp, Scheme, Arc (and many others) still use `car` and `cdr` and all combinations thereof, while Clojure decided to name them differently to detach itself from this old part of the Lisp heritage:

Table 3.6. Lisp VS Clojure naming of list access functions.

Lisp	Clojure
car	`first`
cdr	`rest` or next
caar	ffirst
cadr	`second` or fnext
cdar	nfirst
cddr	nnext

Clojure names might be slightly longer but they better convey the semantic of the function.

See also

`first`, `second` and `last` are just a few of the many ways you can access the parts of a sequence. These functions will be extensively explained in their own section, but the following is an useful summary of what is available:

- next and rest return what's remaining after throwing away the first element of the sequence. They differ in the way they treat empty collections.

- drop accepts the number of elements to remove from the head of the sequence, not just the first.
- drop-last and butlast drop the last element and keeps the rest.
- take and drop-last remove elements from the end of a sequence and keep what's left. The difference is in the interpretation of "n": take will return a collection of "n" elements, drop-last will make sure that the last "n" elements are removed.
- ffirst, nfirst, nnext and fnext are shortcuts for common operations involving sequence containing other sequences. The first letter "f" or "n" indicates the first operation, either `first` or next and the rest of the name the second operation. So for example ffirst is equivalent to `(first (first xs))`, fnext to `(first (next xs))` and so on.
- nth is a general way to access an element by index in a collection.
- rand-nth extracts a random element from a sequence.
- nthrest and nthnext are returning everything after the nth element.

There are also other functions similar to the one above that are optimized for a specific collection type:

- peek grab the first elements for lists and queues. Last element for vectors.
- pop returns everything but the last element for queues, vectors and lists.
- pop! returns the last element of a transient vector.
- get is mainly for hash-maps, but works also on vectors and strings to fetch the element at the specific index. It works on Set to check for the inclusion of an element.
- subvec is dedicated to splitting vector apart at some index n.

Performance considerations and implementation details

=> `O(1)` **first and second**

=> `O(n)` **last**

`first` and `second` are constant time operations, while `last` generally needs to "unroll" the entire sequence to access the last element, thus performing linearly in "n" number of input elements. The key aspect is that `first`, `second` and `last` are designed to work on sequences or collection that can be iterated sequentially. Although other collection types are accepted as input, they need to be converted into sequences, potentially

producing sub-optimal performance. `last`, for instance, should be avoided on vectors for which there are better performing functions (such as peek).

The following table shows the most used collection types, suggesting a faster alternative to `first` or `last` when one exist.

Note

please note that `O(1)` is used as an approximation of `O(log32N)` here and in several other places in the book. `O(log32N)` is very close to `O(1)` for most practical purposes. When the difference is important, it's appropriately made clear.

Table 3.7. Alternative ways to access the head or the tail for

ordered collection types.

Type	Example
`PersistentList`	`'(1 2 3)`
`PersistentVector`	`[1 2 3]`
`PersistentQueue`	`(import 'clojure.lang.PersistentQueu (PersistentQueue/EMPTY)`
`PersistentTreeSet`	`(sorted-set 1 2 3)`
`PersistentTreeMap`	`(sorted-map :a 1 :b 2)`

`PersistentArrayMap`	`(array-map :a 1 :b 2)`
`LongRange`	`(range 10)`
`LazySeq`	`(for [i [1 2]] i)`
`String`	"abcd"

3.5.2 map and map-indexed

`function` `since 1.0`

Listing 3.26. → *Sequence processing, Higher-order functions, Data Pipelines_*

```
(map
  ([f])
  ([f c1])
  ([f c1 c2])
```

```
    ([f c1 c2 c3])
    ([f c1 c2 c3 & colls]))

  (map-indexed
    ([f])
    ([f coll]))
```

`map` is a fundamental tool in almost every functional language. The basic form takes a function and collection and return the sequence of results of the function applied to each element in the collection. The following, for instance, inverts the sign of each number in the list:

```
(map - (range 10))
;; (0 -1 -2 -3 -4 -5 -6 -7 -8 -9)
```

`map-indexed` performs similarly to `map` adding an additional parameter for the function `f` which is the index of the current item in the input collection. We can for example use `map-indexed` to build a map with integer keys given a collection of items:

```
(into {} (map-indexed vector [:a :b :c]))
;; {0 :a, 1 :b, 2 :c}
```

Contract

The contract of `map` is different based on how many collections are passed to the input after the mapping function. "f" should preferably be free of side effects, because `map` and `map-indexed` operates on lazy sequences there is no guarantee about a specific "once-only" calling semantic for "f".

```
  Let's divide the contract based on those cases.
```

`(map f)` : no input collections

- When `map` is invoked with just "f" it returns a transducer and no actual invocation of "f" is performed until the transducer is invoked.

`(map f coll)` : single collection as input

- "f" is invoked with 1 argument and can return any type. "f" needs to support at least artiy-1 but it can also have others, e.g.: `(map - (range 10))`.
- "coll" is a collection that can be iterated sequentially, so that `(instance? clojure.lang.Seqable coll)` returns true.

- returns: a lazy sequence containing the result of applying f to all the elements in the input collection.

`(map f c1 c2 & colls)`: with any number "n" of collections

- "f" needs to support n-arguments invocation, where n is the number of collections parameters. "f" can return any type.
- "c1", "c2", .. , & "colls" are "seqable" collections, so that `(instance? clojure.lang.Seqable c)` is true.
- returns: a lazy sequence containing the result of applying f to all the first elements, then the second elements and so on, stopping at the shortest collection. "f" is called multiple times, starting with all the first items from each collection in order, then the seconds and so on. The fact that iteration stops when reaching the end of the shortest collection can be illustrated with the following example:

```
(map str (range 10) ["a" "b" "c"] "hijklm")
;; ("0ah" "1bi" "2cj")
```

As you can see the "middle" vector containing the 3 letters `["a" "b" "c"]` is determining when the map operation is

going to end. str receives 3 arguments each invocation: "0 a h", "1 b i" and "2 c j".

The contract for `map-indexed` is more restrictive:

- When `map` is invoked with just "f" it returns a transducer that can be later composed or applied.
- "f" must be a function of at least 2 arguments returning any type.
- returns: a lazy sequence containing the result of applying `f(idx,item)` to each item in the collection.

Examples

`map` is often present in data transformations (along with filter) to prepare the data for further processing. In the following example a list of credit products contains essential data like the annual interest rate and the minimum credit allowed. Given a loan amount and a desired number of years, we would like to output how much we will have to repay back and the cost of the credit. The final result gives us a way to compare the cheapest credit for the amount of money we wish to borrow:

```
(def products
  [{:id 1 :min-loan 6000 :rate 2.6}
   {:id 2 :min-loan 3500 :rate 3.3}
```

```
   {:id 3 :min-loan 500 :rate 7.0}
   {:id 4 :min-loan 5000 :rate 4.8}
   {:id 5 :min-loan 1000 :rate 4.3}])

(defn compound-interest [product loan-amount per
  (let [rate (inc (/ (:rate product) 100. 12))]
    (* loan-amount (Math/pow rate (* 12 period))

(defn add-costs [loan-amount period]
  (fn [product]
    (let [total-cost (compound-interest product
          credit-cost (- total-cost loan-amount)
      (-> product
          (assoc :total-cost total-cost)
          (assoc :credit-cost credit-cost)))))

(defn min-amount [loan-amount]
  (fn [product]
    (> loan-amount (:min-loan product))))

(defn round-decimals [product]
  (letfn [(round-2 [x] (/ (Math/ceil (* 100 x))
    (-> product
      (update-in [:total-cost] round-2)
      (update-in [:credit-cost] round-2))))

(defn cost-of-credit [loan-amount period]
  (->> products
```

```
        (filter (min-amount loan-amount))
        (map (add-costs loan-amount period))
        (map round-decimals)
        (sort-by :credit-cost)))

(cost-of-credit 2000 5)

;; ({:id 5 :min-loan 1000 :rate 4.3 :total-cost 1
;;  {:id 3 :min-loan 500  :rate 7.0 :total-cost 1
```

❶ The list of products is short and in memory for this example. It would probably come from a separate source and contain much more detailed data.

❷ The compound interest formula is a direct translation of the Wikipedia version [76].

❸ `add-cost` is the function that injects two new keys into the input product. The total payments and cost of credits are double with many digits.

❹ `min-amount` returns a function predicate that is dependent on the requested loan amount. It will be used by filter in the main calculation below.

❺ `round-decimals` is the second function we use with `map`. In this case given a product we want the two costs to be rounded to the second decimal. update-in is relatively straightforward to use for this goal.

❻ Finally we chain everything together using ->>. filter operations appear first so downstream parts of the computation receives less work to do.

From the example we can see that for our request to borrow 2000 and repay them in 5 years, product id "5" is the best option, although other products like id "1" have a very competitive rate but they don’t allow 2000 borrowing.

Now an example related to `map-indexed`, which comes handy when we want to associate an ordinal number (usually a natural number) to the elements in a collection, so that it's possible to relate them to their position. `map-indexed` saves us from explicitly passing a range. Showing the winning tickets for the lottery could be such an example:

```
(def tickets ["QA123A3" "ZR2345Z"
              "GT4535A" "PP12839"
              "AZ9403E" "FG52490"])

(defn draw [n tickets]
  (take n (random-sample 0.5 tickets)))

(defn display [winners]
  (map-indexed
    (fn [idx ticket]
      (format "winner %s: %s" (inc idx) ticket))
    winners))

(display (draw 3 tickets))
;; ("winner 1: QA123A3" "winner 2: GT4535A" "win
```

❶ `draw` takes the tickets and performs a random-sample of `n` winners. 0.5 is probability of that element in the collection to be part of the final sequence. Note that depending on how many tickets are actually sampled, there is a chance for the function to return less than 3 tickets.

❷ `display` uses `map-indexed` to interleave the order of the extraction (and thus a higher prize) to the extracted tickets, printing them in a nice format.

map and the lambda notation

`map` is another early Lisp function (along with eval or apply for example) that was originally called `maplist`. Herbert Stoyan, in his Early LISP history paper, shows that `maplist` requirements were possibly responsible for the introduction of the lambda notation into Lisp. The following is for instance a very early design of a differential function `diff` that makes use of `maplist` to return the derivative of another function:

```
diff = (ctr(J) = 1 -> 0,
  car(J) = "x" -> 1,
  car(J) = "plus"  -> consel("plus", maplist(cdr(J), K, diff(K)))
  car(J) = "times" -> consel("plus", maplist(cdr(J), K, consel(
      "times" -> consel(
      "plus",  maplist(cdr(J), K, consel(
      "times", maplist(cdr(J), L, (L = K -> copy (L)))))))))))
```

❶ Note the use of M-Expressions, the way Lisp was originally designed to be written in.

❷ `maplist` first use appears in this line. Also note that newlines were added for clarity, but initial Lisp was supposed to be written as a single continuous line.

We don't need to enter into the details of how `diff` was supposed to work, but we can have a look into how `maplist` is used in this fragment:

```
maplist(cdr(J), K, diff(K))
```

In this early design (early 1958), `maplist` takes 3 arguments: a list of items (for example `(cdr(J))`), a target list to collect the results (`K` `L`) and the actual invocation of a function (`diff`). McCarthy, after finding impractical to implement `maplist` as designed, introduces the lambda notation. The following is a re-write of the `diff` function some time later:

```
diff(L,V) = (car(L)=const->copy(C0),
  car(L)= var -> (car (cdr(L)) = V -> copy(C1, 1
  car(L)= plus -> consel(plus, maplist(CDR(L), λ
  car(L)= times-> consel(plus, maplist(cdr(L),
    λ(J, consel(times, maplist(cdr(L),
      λ(K, (J != K -> copy(car(K)), l->diff(car(
```

Calls to `maplist` are now making use of 2 arguments, like the following fragment:

```
maplist(CDR(L), λ(J diff(car(J), V)))
```

The first argument is now the list to map over (like for example `CDR(L)`) and a lambda `λ(J, f)` function of J followed by the body of the function, removing the need to pass as argument a list `K` to hold the results. `maplist` eventually made it to the famous 1960 original Lisp paper with the following definition:

```
maplist[x; f] = [null[x] -> NIL; T -> cons[f[x];
```

The `maplist` definition can be explained as:

- `maplist[x; f]` is a function of a list `x` and a function `f`.
- When `null[x]` then just returns NIL.
- In any other case, return `cons` of `f` applied to `x` with recurring on `maplist` with the rest of the collection `x` and the function `f` again.

In Clojure this is very similar to the current `map` implementation (although in Clojure this is complicated by building the resulting sequence as a lazy-sequence).

See also

- mapcat is useful when the result of applying `f` to an item is again a sequence, with the overall results of producing a sequence of sequences. mapcat applies a final concat operation to the resulting list, flattening the result.
- amap operates with the same semantic of `map` on Java arrays.
- mapv is a specialized version of `map` producing a vector instead of a lazy-sequence as output. It uses a transient internally so it's faster than the equivalent `(into [] (map f coll))`.
- pmap executes the `map` operation on a separate thread thus creating a parallel `map`. Replacing `map` with pmap makes sense when the overall cost of handling the function `f` to separate threads is less than the execution of `f` itself. Long or otherwise processor-consuming operations usually benefit from using pmap.
- clojure.core.reducers/map is the version of `map` used in the context of reducers. It has the same semantic of `map` and should be used similarly in the context of a chain of reducers.

Performance considerations and implementation details

=> `O(n)` **Linear in the number "n" of items in the input collection**

`map` requires linearly more computation time the larger is the input collection. In terms of allocated memory space, `map` applies `f` to the items in the input collection "lazily", which means that the actual evaluation of the `f` function happens only when something upstream requires more elements. So unless the user explicitly holds on to the head of the output sequence, `map` is not going to load all the processed items at once. Look for example at the following:

```
(let [res (map inc (range 1e7))] (first res) (la

(let [res (map inc (range 1e7))] (last res) (fir
```

❶last is forcing `map` to perform the computation on all elements to return the last one in the sequence. Since nothing else needs the local binding `res` after closing the scope, every item before the last can be safely garbage collected.

❷Here `(last res)` is requested first, forcing `map` to go trough all the 10M elements and increment them. Differently from before, we still need `res` after that because there is another instruction in the scope of the local binding. This second version will likely exhaust the memory (depending on hardware and the JDK settings) because no elements of the output collection can be garbage collected until first is evaluated.

Although in the contract section we talked generically about "collections" as input, we can now be more precise by looking at the implementation. `map` calls seq on the input collection which means the input collection must implement `clojure.lang.Seqable` interface. Since virtually all "iterable" things in Clojure implement `Seqable` we can talk about an "input collection" but `map` is technically a sequence-in sequence-out operation. The only reason you might be interested in this detail is if you wanted to create your own

sequence in that integrates nicely with the rest of Clojure ecosystem.

3.5.3 filter and remove

`function` `since 1.0`

Listing 3.27. → *Sequence Processing, Items Selection*

```
(filter
  ([pred])
  ([pred coll]))

(remove
  ([pred])
  ([pred coll]))
```

`filter` and `remove` are very common operations on sequences. They perform the same action of removing/keeping an item in a sequence based on a predicate (a function returning logical `true` or `false`):

- `filter` allows the item through when the predicate is `true`.
- `remove` prevents the item to appear in the resulting sequence when the predicate is `true`.

`filter` is essentially the complemented `remove` operation (and the other way around):

```
(filter odd? [1 2 3 4 5])
;; (1 3 5)

(remove even? [1 2 3 4 5])
;; (1 3 5)
```

Contract

- "pred" is mandatory argument. It must be a function of 1 argument returning any type (which will be interpreted as logical `true` or `false`). "pred" should preferably be free of side effects, because `filter` and `remove` operates on lazy sequences there is no guarantee about a specific "once-only" calling semantic for "pred".
- "coll" can be any sequential collection (such that `(instance? clojure.lang.Seqable coll)` is `true`).

- returns: a (potentially empty) lazy sequence which has the same size or less than the input sequence. `filter` keeps items when `(true? (pred item))` while `remove` removes them for the same predicate result.

Examples

`filter` and `remove` are typically found in processing pipelines. Some data enters the pipeline on one end and is subject to a mix of transformation to produce the result. It's usually a good idea to remove unwanted elements before doing any other expensive computation. For this reason operations like `filter` or `remove` most likely appear at the top of the chain. There are `filter` examples throughout the book worth reviewing:

- Filtering out interesting sentences in the sentiment analysis example.
- Preparing an index by initial letter for a dictionary in the Levenshtein distance example.
- `filter` is also common in transducer chains, like the following example to find the longest function in a namespace.

In this section we are going to show a common usage of `remove` in conjunction with some-fn to remove some type of values accumulating during the computation. In the following example, a network of sensors connected to weather stations produces regular readings that are encoded as a list of maps. Each map contains some identification data, a timestamp and a payload containing the data for all the available sensors. One potential problem is that any of the sensors could fail resulting in that particular key missing or an `:error` value being reported. We want to be able to process such events and take care of possible errors:

```
(def events
  [{:device "AX31F" :owner "heathrow"
    :date "2016-11-19T14:14:35.360Z"
    :payload {:temperature 62
              :wind-speed 22
              :solar-radiation 470.2
              :humidity 38
              :rain-accumulation 2}}
   {:device "AX31F" :owner "heathrow"
    :date "2016-11-19T14:15:38.360Z"
    :payload {:wind-speed 17                ; ❶
              :solar-radiation 200.2
              :humidity 46
              :rain-accumulation 12}}
```

```
   {:device "AX31F" :owner "heathrow"
     :date "2016-11-19T14:16:35.362Z"
     :payload {:temperature :error        ; ❷
               :wind-speed 18
               :humidity 38
               :rain-accumulation 2}}
   {:device "AX31F" :owner "heathrow"
    :date "2016-11-19T14:16:35.364Z"
    :payload {:temperature 60
              :wind-speed 18
              :humidity 38               ; ❸
              :rain-accumulation 2}}])

(def event-stream                        ; ❹
  (cycle events))

(defn average [k n]
  (let [sum (->> event-stream
                 (map (comp k :payload))  ; ❺
                 (remove (some-fn nil? keyword?)
                 (take n)
                 (reduce + 0))]
    (/ sum n)))

(average :temperature 60)                ; ❼
;; 61
```

```
  (average :solar-radiation 60)
  ;; 335.200000004
```

❶No temperature reading for this event.

❷A temperature reading of ":error" means that the sensor was not able to measure the temperature.

❸No solar-radiation reading for this other event.

❹To simulate abundance of such events, we iterate on them to create an infinite sequence.

❺Notice how we can use comp to access a deeply nested map. Keys must appear in opposite order of access.

❻Similarly, we can use some-fn with `remove` to prevent reduce to receive values other than numbers. If we don't remove `nil` or `:error` elements, reduce will fail. Also notice how `remove` appears conveniently before take so we can be sure we have the requested number of elements before calculating the average.

❼Considering one event per second, this temperature `average` for 60 events would be the average for the last minute.

Note

We are going to see the same example of processing events and calculate their average with transducers while talking about completing.

Extending `filter` to support multiple collections

One of the nice features in map is the possibility to pass multiple collections as input arguments. By passing a predicate function `pred` that can accept multiple parameters (one for each collection) we can extend `filter` in a similar way. We are going to call this new function `filter+`:

```
(defn walk-all
  "Returns a lazy-seq of all first elements in coll,
  then all second elements and so on."
  [colls]
  (lazy-seq
    (let [ss (map seq colls)]                                    ; ❶
      (when (every? identity ss)                                 ; ❷
        (cons (map first ss) (walk-all (map rest ss)))))))       ; ❸

(defn filter+
  ([pred coll]                                                   ; ❹
   (filter pred coll))
  ([pred c1 c2 & colls]                                          ; ❺
   (filter+ #(apply pred %) (walk-all (conj colls c2 c1)))))
```

❶We use `walk-all` helper function to create a lazy sequence of all the first elements in a list of input collections, then the second elements and so on, stopping when we reach the end of the first collection. Before doing that, we make sure all collections are not empty using seq.

❷We also need to make sure we didn't reach the end of any of the collection. We can make sure there are no `nil` by checking that identity is returning true for every? element in the sequence.

❸We build the lazy sequence by using cons for all the first elements so far into the recursion of `walk-all` again for all the remaining elements.

❹The basic arity of `filter+` is just calling into `filter`.

❺The extended arity for `filter+` is taking the results of the `walk-all` function and applying the predicate to all the first elements, then the second ones and so on.

Compared to map it's not immediately obvious how to use our new extended `filter+`. One idea is to consider the predicate as a function of multiple arguments returning a result that will be interpreted as logical `true` or `false`. We could for example filter only those numbers (as strings) containing at least one instance of the index they appear in the input collection:

```
(filter+ re-seq                                    ;
  (map re-pattern (map str (range)))               ;
  ["234983" "5671" "84987"])
;; ((#"1" "5671"))                                 ;
```

❶re-seq is a function of two arguments, exactly what we need for the two-collections input in this example.
❷The first collection builds up from an infinite range into a list of regular expressions: #"1", #"2" and so on. It uses a string as input for re-pattern
❸"5671" appears at index "1" in the input vector and contains the number "1", so it appears in the final results.

See also

- keep is a cross between map and `remove`: like map it applies a function to a sequence and like `remove` with nil? it removes `nil` from the output. It could be used with similar effect to `(remove nil?)` when using identity as the function: `(keep identity coll)`.
- filterv is the equivalent operation optimized for vectors. You should use mapv whenever a vector input type can be assumed, since the operation in that case is much faster.

Performance considerations and implementation details

=> `O(n)` **Linear in the number n of input elements**

Unsurprisingly, `filter` and `remove` needs to walk all elements of the input collection to return the results, producing a linear computational cost. Like map, `filter` operates lazily so the same considerations apply: the full computational cost is only happening if the entire output is requested. The full memory allocation cost can be avoided by being careful and not "holding on the head" of the sequence (once again, check the map performance section for an example.

From the implementation perspective, `filter` is similar to other sequence-in-sequence-out functions. The implementation is complicated by dealing with laziness and chunks, so that different sequences implementation are treated correctly.

3.5.4 reduce and reductions

`function` `since 1.0`

Listing 3.28. → *Recursion, Folding, Accumulation*

```
(reduce
  ([f coll])
  ([f val coll]))

(reductions
```

```
    ([f coll])
    ([f val coll]))
```

`reduce` takes a function of two arguments. After calling the function against the first item (or "val" if provided), it proceeds to call the same function with the previous result against the next item in the sequence. At each step in walking the input sequence, the function has an opportunity to do something with the "result-so-far" and the next element.

Similarly to the other functions presented in this chapter, `reduce` is a well known functional tool. When describing operations on sequences (or Clojure collections in general) `reduce` is often mentioned as part of the trio with map and filter, as it frequently appears as the last step of a processing pipeline. The following example shows an initial list of numbers transformed into squares and their total sum used to calculate the average:

```
(defn sum-of-squares [n]
  (->> (range n)      ; ❶
       (map #(* % %)) ; ❷
       (reduce +)))   ; ❸

(defn average-of-squares [n]
  (/ (sum-of-squares n) (double n)))
```

```
(average-of-squares 10)
;; 28.5
```

❶ `range` produces the initial dataset.
❷ `map` transforms each number into the square of that number.
❸ `reduce` iterates the results so far adding up the numbers.

`reductions` helps visualizing the `reduce` process. It has the same interface as `reduce` but it also outputs all the intermediate results:

```
(reductions + (map #(* % %) (range 5)))
;; (0 1 5 14 30)
```

The output of `reductions` above is the result of the following steps:

- `(+ 0 0)` is the first step. The "sum-so-far" is still 0 and the first element is also 0.
- `(+ 0 1)` the sum-so-far is then added to the second element in the sequence: 1.
- `(+ 1 4)` the sum-so-far becomes 1 and that is added up to the square of the second number in the range.

- `(+ 5 9)` the step before the last continues with the same idea.
- `(+ 14 16)` the square of the last element of the sequence 16 gets added to the sum-so-far. There are no more inputs, the last evaluation is "30".

As a consequence of the relationship between `reduce` and `reductions` it is possible to say that given a collection "coll" and a function "f": `(= (reduce f coll) (last (reductions f coll)))`.

`reduce` implements the typical recursive iterative process (where the incremental results appear in the argument list) and includes a standard vocabulary:

- "accumulator" is the name given to the "result-so-far". It is sometimes abbreviated as "acc" in source code.
- "reducing function" is the function "f" of two arguments. Note that "reducing" does not necessarily mean a scalar value or "single object" as output. You can indeed use `reduce` with hash-maps to enrich them with new keys (see the examples section after the contract).

- "fold" is the class of operations `reduce` belongs to, more specifically a "fold-left". This is because the elements of the input collection are gradually consumed from the left, like if we were "folding up" the sequence.

Contract

input

- "f" should provide both a 0 and 2 arguments arity (invokable with 0 or 2 arguments) and is a required argument. The 0-argument version is only called if there is no "val" initial value and the collection is either empty or `nil`:

```
(reduce + nil) ; ❶
;; 0
(reduce / [])  ; ❷
;; ArityException
```

❶The collection is `nil`, so `(+)` is invoked without arguments returning the identity for addition.
❷An exception is thrown on an empty collection because division does not have a 0-arity call.

- "coll" is also required and can be `nil` or empty. If "coll" is not `nil`, "coll" needs to implement the `Seqable` interface such that `(instance? clojure.lang.Seqable coll)` returns `true` (the only not supported type are transients).
- "val", when present, is used instead of the first item in the collection to start the computation. It follows that `(reduce + 1 [1 2 3])` and `(reduce + [1 1 2 3])` are equivalent. When "coll" is either `nil` or empty, then "val" is always returned.

notable exceptions

- `IllegalArgumentException` when "coll" is not a sequential collection (transients and scalars).
- `ArityException` when the function "f" does not support an arity with zero arguments and the collection is either `nil` or empty.

output

- In the most common case of "coll" containing at least one item, `reduce` returns the result of applying "f" to "val" (or missing that the first item) and the next item in "coll". Then "f" applied to the previous result and the next item and so on, up to the last item.
- When "coll" contains a single item: if there is no initial "val" then that item is returned. If "val" is provided then "f" is invoked once with "val" and the item.
- When "coll" is empty or `nil`: if there is no initial "val" then "f" is invoked without arguments. If "val" is provided, then "val" is returned.

`reductions` returns a sequence of all the intermediate results of calling the function "f" with the same `reduce` rules.

Examples

`reduce` has a broad range of applications (usually in conjunction with map and filter or other sequence-based functions) to prepare the input. `reduce` is also mentioned in association with big-data applications [77]. The computation model called "map-reduce" solves problem by constraining computation to a mix of map and `reduce` operations. We can see map-reduce main idea in action while counting words in a string of text:

```
(defn count-occurrences [coll]
  (->> coll
       (map #(vector % 1))
       (reduce (fn [m [k cnt]]
         (assoc m k (+ cnt (get m k 0)))) {})))

(defn word-count [s]
  (count-occurrences (.split #"\s+" s)))

(word-count "To all things, all men, all of the

;;{"To" 1
;; "all" 3
;; "and" 1
;; "children" 1
;; "men," 1
;; "of" 1
;; "the" 1
;; "things," 1
;; "women" 1}
```

❶ The first operation is to associate the number "1" to each item in the list.
❷ `reduce` comes next, to "reduce" multiple "1" appearing for the same key. We destructure here each vector-item in the input into a key "k" and value "cnt" bindings.
❸ `reduce` starting point is an empty map. We assoc the element at key "k" knowing that it might not be found. By using get to fetch the current counter we can pass a default initializer of 0 for the sum.

Conveniently, the `count-occurrences` function in the example can handle any item type, not just "words" (provided items contains some definition of equality that can be used to store them in the hash-map). Even more conveniently, Clojure contains such a function in the standard library already, it's called frequencies:

```
(defn word-count [s]
  (frequencies (.split #"\s+" s)))

(word-count "To all things, all men, all of the

;;{"To" 1
;; "all" 3
;; "and" 1
;; "children" 1
```

```
  ;; "men," 1
  ;; "of" 1
  ;; "the" 1
  ;; "things," 1
  ;; "women" 1}
```

❶The custom made `count-occurrences` has been replaced with the standard library equivalent frequencies.

frequencies implementation in the standard library is very similar to our `count-occurrences` (with the addition of using transients to boost performance). Although counting words can be easily solved with frequencies, the general mechanism illustrated in the example can be used in other more specific scenarios.

Let's now have a look at an example involving `reductions`. `reductions` is designed to keep track of each evaluation of the reducing function over time. This aspect could be useful in showing time series, like a "moving average". This technique is well known in finance, where it's used to eliminate local outliers and highlight trends [78]. The following example calculates the average for each incoming value of a stock price (but there are other possibilities like batching up all values for a

day that we are not considering there to remove some complexity):

```
(defn next-average [[cnt sum avg] x]
  (let [new-cnt (inc cnt)
        new-sum (+ sum x)
        new-avg (/ new-sum (double new-cnt))]
    [new-cnt new-sum new-avg]))

(defn stock-prices [values]
  (reductions next-average [0 0 0] values))

(stock-prices [5.4 3.4 7 8.2 11])
;; ([0 0 0]
;;  [1 5.4 5.4]
;;  [2 8.8 4.4]
;;  [3 15.8 5.266666666666667]
;;  [4 24.0 6.0]
;;  [5 35.0 7.0])
```

❶ `next-average` is our reducing function. It destructures the results so far into a counter, the sum and the last average calculated. It then proceeds with generating a new average that is stored in a new triplet ready to be returned for the next iteration.

❷ `reductions` is invoked with the reducing function, an initializer triplet of all zeroes and a collection of values.

❸The result of invoking `stock-prices` shows all generated triplets. If we are interested in just the average, we could `(map last)` the results and ignore the rest.

Do you need a right-fold or a left-fold?

Early Lisp didn't have `reduce`. The concept of "fold" (the more generic classification for operators like `reduce`) is first mentioned in a 1952 book by Stephen C. Kleene "Introduction to Mathematics" and introduced to mainstream programming by APL in 1962 foonote:[APL is an example of applicative language that inspired features in many other languages. See en.wikipedia.org/wiki/APL_(programming_language) for more information.]. A fold can be seen as a mechanism to process a recursive list (like those made of cons cells in Lisp) so that the list can be reduced to some starting value. We could sum up a list of numbers like this:

```
(def numbers (cons 1 (cons 2 (cons 3 (cons 4 (list)))))) ; ❶

(defn foldl [f init xs]
  (if (empty? xs)
    init
    (foldl f (f init (first xs)) (rest xs))))           ; ❷
```

```
(foldl + 0 numbers)
;; 10
```

❶ `number` has been defined with the typical "cons-cell" design to show the left to right movement in folding the list operated by `foldl`.

❷ The recursion "unfolds" the list at each iteration applying "f" to the first element and the results so far (stored in "init").

The above is conceptually how `reduce` is implemented in Clojure for lists [79]. The example shows what happens at each step of the iteration over the input list. The first recursion, "init" is `(+ 0 1)`, then `(+ 1 2)`, then `(+ 3 3)` and finally `(+ 6 4)`. Visually, the computation starts by applying "f" from the left, reason why Clojure `reduce` is also called a left-fold. Also note how `foldl` is tail-recursive, since the new `foldl` invocation is the very last operation in the loop.

There is also another way to write the same operation, suspending the application of "f" until we reach the end of the list:

```
(defn foldr [f init xs]
  (if-let [x (first xs)]
    (f x (foldr f init (rest xs))) ; ❶
    init))
```

```
  (foldr + 0 numbers)
  ;; 10
```

❶The last operation is now "f" invoked over the arguments, where the collection is represented by the recursive call to `foldr`.

The implementation above is also called a right-fold, because the first invocation of "f" happens using the tail of the collection (the number 4) and moves backward until it reaches the head to perform the last operation. To obtain this effect, the recursive `foldr` invocation happens inside "f" in the last line, forcing the computation to suspend until the frame returns. Note how `foldr` is now not tail-recursive and potentially subject to stack overflow (aggressively lazy languages like Haskell have instead the option of making good use of `foldr` without exhausting the stack).

A practical distinction between `foldr` and `foldl` is about non-associative operations, in which the order the list is consumed matters. Operations like division `/` for example, behave differently with `foldl` or `foldr`: the unfolding of `foldl` with `/` would result in `(/ (/ (/ (/ 1. 1.) 2.) 3.) 4.)` while `foldr` would produce the equivalent of `(/ 1 (/ 2 (/ 3 (/ 4 1.))))` generating a different output:

```
(foldl / 1. numbers)
;; 0.04166666666666664

(foldr / 1. numbers)
;; 0.375
```

`foldr` is not part of the Clojure standard library, in part for the problem with tail-recursion in part because it can be easily implemented using reverse (although with an higher performance cost):

```
(defn foldr [f init xs]
  (reduce (fn [x y] (f y x)) init (reverse xs)))

(foldr / 1. numbers)
;; 0.375
```

❶ `foldr` implemented using `reduce` and reverse. Note that the reducing function "f" needs to swap its arguments.

See also

- reduce-kv is the analogous of `reduce` for associative data structures. Instead of a function of 2 arguments, `reduce-kv` takes 3 arguments: a map accumulator and a key-value pair. Prefer `reduce-kv` when reducing against a hash-map.
- loop is the low common denominator of almost all sequential processing functions. There is always a way to transform a `reduce` into a `loop-recur` where you can customize all aspects of the reduction, including propagating types if necessary.
- frequencies was mentioned in the examples as a perfect application for `reduce`, where a final data structure is created incrementally by walking a sequential input.
- reduced are a group of functions that you can use to fine tune the behavior of `reduce` or `reductions`. When an element in the sequence is reduced?, `reduce` stops the computation and return the result immediately. This behavior requires a reducing function that knows how to deal with the special element:

```
(reductions
  (fn [acc itm]
    (if (> itm 5)
```

```
        (reduced (+ itm acc)) ; ❶
        (+ itm acc)))
    (range 10))
  ;; (0 1 3 6 10 15 21)
```

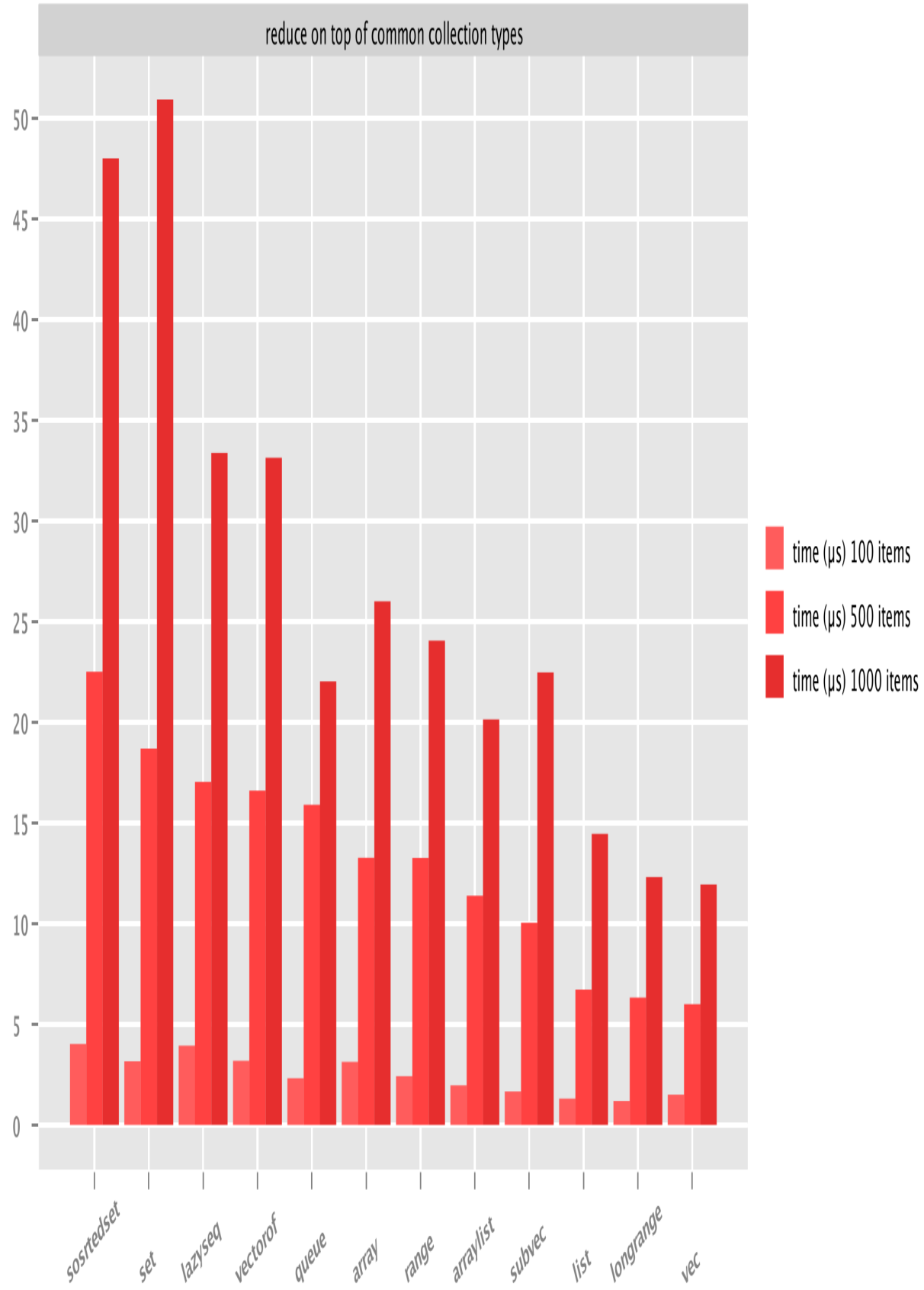
❶ `reduce` and `reductions` stops looping if an element is wrapped by reduced.

Performance considerations and implementation details

=> `O(n)` **Linear in the number "n" of items in the input collection**

`reduce` is implemented differently based on the type of the input and protocols like `CollReduce` and `InternalReduce` (in the `clojure.core.protocols` namespace) are available for new collection implementations to provide a custom `reduce`. The following chart shows the standard `reduce` invoked on several collection types and sizes:

Figure 3.5. `reduce` invoked on different collection types and sizes. Lower number means faster execution.


reduce on top of common collection types
50
45
40
35
30
25
20
15
10
5
0
time (µs) 100 items
time (µs) 500 items
time (µs) 1000 items
sosrtedset
set
lazyseq
vectorof
queue
array
range
arraylist
subvec
list
longrange
vec

The diagram shows the linear behavior of `reduce` while increasing the collection size from 100 to 500 and then 1000 items. It also shows that `reduce` on sets (ordered or unordered) is roughly 5 times slower than vectors, the fastest of the benchmark. In absolute terms, `reduce` (especially on vectors or lists) is hard to beat, even with a loop-recur.

`reduce` walks the entire sequence by design, so it's not lazy (although there are ways to short-circuit using reduced). The memory footprint depends largely on the reducing function. Assuming "f" is not accumulating the entire input in memory, even large sequences can be reduced in linear time without worrying about going out of memory:

```
(let [xs (range 1e8)] (reduce + xs))      ; ❶
;; 4999999950000000

(take 10 (reduce merge '() (range 1e8)))  ; ❷
;; java.lang.OutOfMemoryError: GC overhead limit
```

❶ `+` uses the items to complete the sum, but after that they can be safely garbage collected, resulting in just a portion of the large collection being in memory at any given time.

❷ In this second example, the reducing function is merge. The result is a collection with the same size of the input, forcing all elements in memory. The likely outcome (depending on the JVM settings) is an out of memory error.

Other cases exhausting memory are less easy to find:

```
(let [xs (range 1e8)] (last xs) (reduce + xs)) ;

;; OutOfMemory
```

❶ The call to last happens before `reduce`. Since they appear in the same form, the content of `xs` cannot be garbage collected before also `reduce` has an opportunity to scan the sequence.

Both last and `reduce` function calls wouldn't normally produce an out of memory when taken in isolation. The problem is that they appear inside the same expression, so the garbage collector that would normally kick-in while `last` is scanning through the sequence cannot run, as `reduce` holds the head of the sequence preventing garbage collection.

One last word about `reductions` which is also part of the section. Despite exhibiting the same behavior, `reduce` and `reductions` are quite different in performance. `reductions` is not a drop-in replacement for `reduce` because it always walks the input collection sequentially, regardless of a potential custom `reduce` implementation:

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (range 1000)]
  (quick-bench (last (reductions unchecked-add-i

;; Execution time mean : 56.880736 µs

(let [xs (range 1000)]
  (quick-bench (reduce unchecked-add-int xs)))

;; Execution time mean : 4.692101 µs ; ❶
```

❶ It's worth noticing that `reductions` does not have the same performance of `reduce` and the two should be not be used interchangeably.

[75] The book will try to clarify the difference between collections and sequences when necessary, but a good starting point is this article on sequences by Alex Miller: insideclojure.org/2015/01/02/sequences/

[76] See en.wikipedia.org/wiki/Compound_interest for an example of compound interest calculation

[77] See the Google paper that popularized the topic a while ago: research.google.com/archive/mapreduce.html

[78] Please read the Wikipedia article on moving averages available at en.wikipedia.org/wiki/Moving_average to know more

[79] reduce is instead implemented as a for loop in Java for most of the Clojure collections

3.6 SUMMARY

This chapter contains a mix of powerful Clojure functions and macros. They've been selected because they cover some of the most important aspects of programming, such as conditional branching, iteration and the basics of collection processing. More than other languages, Clojure makes extensive use of sequences and collections but the list of functions dedicated to them is so long that they have been dedicated their own chapters.

But before we can talk about data structures (part III of the book), we need to see more Clojure fundamentals, such as macros (chapter 4), arithmetic operations (chapter 5), equality (chapter 6) and transducers (chapter 7).

4CREATING AND INSPECTING MACROS

Thanks to Nicola Mometto, Clojure core committer, for contributing this chapter (except definline).

Arguably one of the most powerful aspects of any LISP is their ability to define custom and arbitrarily complex macros, and Clojure is no exception. Although many languages have the concept of macros, LISP ones are an entirely different beast, effectively providing the users of the language with an expressive power that in other languages only compiler authors can have.

Given their power one would expect macros to be a complex and advanced feature to use, this is in fact not the case: because of the homoiconic nature of Clojure [80], defining a macro is as simple as defining functions and manipulating data: macros are indeed just regular functions that the compiler invokes at compile time, passing as inputs their arguments as if wrapped in an implicit `quote` invocation and returning a valid Clojure expression that will be evaluated at run time.

Macros can be used for a variety of reasons, from simply reducing the amount of repeated code, to allowing code to be expressed in a more concise way, to writing complex DSLs or embedding small compilers [81].

This chapter is dedicated to the facilities in the standard library (and the language) to create, inspect and help using macros. Here's a brief summary:

- defmacro is the main entry point in the language to create macros. The body of the macro is assigned to newly created var in the current namespace and ready to be referenced. Although several Clojure facilities can be used also outside macros, many are found almost exclusively when creating them (like syntax-quote for instance). We are going to see a few of them while illustrating `defmacro`.
- macroexpand are debugging tools to show how the macro will process some input without actually executing. The "expanded" macro is simply printed out for inspection.
- `quote` is a function that prevents evaluation of what is passed in as argument. It is simple but fundamental for macro programming.

- gensym is a helper function to generate unique symbol names. It is part of macro hygiene [82].
- definline takes a body and defines both a function and an "inilined" version of that function. The inlined version is very similar to a macro and shares the same syntax.
- destructure is used by many macros in the standard library to implement destructuring, a key feature of Clojure.
- apply-template and do-template are dedicated to replacement of symbols in expressions during macro expansion.

4.1 DEFMACRO

`macro` `since 1.0`

Listing 4.1. → *Macro definition, Metaprogramming*

```
  (defmacro [name & fdecl])
```

`defmacro` is to macros what defn is to functions, but while a function evaluates after compilation, a macro evaluates when its body is compiling. This gives the macro an opportunity to alter the output of the compiler, including intercepting

arguments before they are evaluated (which is the case in normal Clojure functions). Being a macro itself, `defmacro` action can be revealed with macroexpand:

```
(macroexpand '(defmacro simple [a] (str a))) ; ❶

;; (do
;;   (clojure.core/defn simple ([&form &env a] (
;;   (. (var simple) (setMacro)) ; ❸
;;   (var simple)) ; ❹
```

❶The macro being defined is simply returning a string conversion of its only argument.

❷ `defmacro` produces a do block starting with a call to defn to define what is at the beginning just a function. As you can see two arguments are automatically added to the generated function, &form and &env the meaning of which will be explained further down in the chapter.

❸Once the function is defined, it is transformed into a macro making direct access to the `clojure.lang.Var` object that the previous line just interned in the current namespace.

❹The last form returns the var object just created and set to be a macro.

Because `defmacro` is built on top of defn, it supports all of its features, including multiple arities, destructuring, `:pre` and `:post` conditions and more. Please check defn contract and examples for any of these features.

All the macros provided by the Clojure standard library are defined using `defmacro` itself (except defn that needs to come first). This is for example how when is defined:

```
(defmacro when
  "Evaluates test. If logical true, evaluates
  body in an implicit do." ; ❶
  {:added "1.0"} ; ❷
  [test & body]
  (list 'if test (cons 'do body))) ; ❸
```

❶The string documenting the macro needs to be between the name of the macro and the arguments declaration.

❷Here an additional metadata map is present.

❸The macro body returns a list. Since the macro is executed at compile time, the list is evaluated (basically removing "list" and replacing "test" and "body" with proper expansions) and "in-lined" at the calling site.

We can verify the expected behavior with macroexpand-1 which for clarity is not further expanding past the first level:

```
(macroexpand-1 '(when (= 1 2) (println "foo")))

;; (if (= 1 2)                ; ❶
;;   (do (println "foo")))

(when (= 1 2)                 ; ❷
  (println "foo"))

;; nil
```

❶The arguments passed to the macro are not evaluated like for normal function calls, but are instead passed to the macro as their quoted value and a valid Clojure expression is returned.

❷Because `when` is implemented as a macro, the body expression is only evaluated if the test expression returns `true`. This wouldn't be possible using a normal function. b .Contract

input

`defmacro` uses the same syntax as defn. The reader is invited to review defn for the complete set of options.

- "name" must be a valid Clojure symbol. "name" is used as the name for the macro and is required.
- "fdecl" is commonly given as a vector of arguments and a body. The argument vector is always added 2 implicit arguments, &form and &env.

output

`defmacro` returns a `clojure.lang.Var` object referencing the macro just created. The macro "name" becomes available in the current namespace as a side effect.



Note

`defmacro` hard limit on the number of fixed arguments is not 20 but 18 because of the 2 implicit arguments.

Examples

One of the most common usage is the `with-` style of macros, a class of macros that execute their bodies within a defined context, automatically performing some logging or cleanup logic.

Here's an example of such a macro usage applied to the problem of contacting some third party service that involves a network call. When networking is involved, an application needs to always prepare for the worst, such as intermittent connections, unreachable hosts and so on. For this reason one common pattern is to keep track of networking error and re-try to contact the third party service some number of times before giving up completely and raising a proper error:

```
(defn backoff! [attempt timeout] ; ❶
  (-> attempt
      (inc)
      (rand-int)
      (* timeout)
      (Thread/sleep)))

(defn frequently-failing! [] ; ❷
  (when-not (-> (range 30)
                (rand-nth)
                (zero?))
    (throw (Exception. "Fake IO Exception"))))

(defmacro with-backoff! ; ❸
  [{:keys [timeout max-attempts warning-after] :
  `(letfn [(warn# [level# n#] ; ❹
             (binding [*out* *err*]
               (println
```

```
                  (format "%s: expression %s fail
                          (name level#) '(do ~@bo
      (loop [attempt# 1] ; ❻
        (when (not= :success (try
                               ~@body
                               :success
                               (catch Exception e
          (when (= ~warning-after attempt#)
            (warn# :WARN attempt#))
          (if (not= ~max-attempts attempt#)
            (do
              (backoff! attempt# ~timeout)
              (recur (inc attempt#)))
            (warn# :ERR attempt#))))))

(with-backoff! ; ❼
  {:timeout 10
   :max-attempts 50
   :warning-after 15}
  (frequently-failing!))

;; WARN : expression (do (frequently-failing!))
;; nil
```

❶The function `backoff!` implements a simple backoff algorithm: taking as input an attempt number and a timeout it then picks `n`, a random number between 0 and `attempt` and sleeps for `n*timeout` ms.

❷The function `frequently-failing!` simulates a function that is subject to frequent failures, only succeeding `1/30` of the times

❸The macro `with-backoff!` takes a map defining the desired backoff behavior and a body to execute in that backoff context.

❹We start right off by using syntax-quote on the returned expression of the macro, making sure that we return a data structure representing a program rather than executing that program. We immediately make use of the additional features that syntax-quote has over normal `quote`, by using the `gensym` feature for both the local function we're defining and for its arguments. In particular the function `warn` that we're defining will deal with printing a warning or an error message to *err* reporting the number of retries and the expression that is being retried

❺Here we make use the unquote-splicing feature of syntax-quote, to splice the list of expressions into a `do` body. Note that if, for example, we defined `with-backoff!` as a function taking an anonymous function, this level of

reporting wouldn't have been possible, as functions don't have a way of accessing the actual representation of the arguments they're handed.

❻ The macro then emits a loop in which the body is evaluated. If its evaluation causes an exception, the exception is caught and we proceed with the potential backoff and retry, otherwise the loop simply returns.

❼ Here we demonstrate how `with-backoff!` is used, using the previously defined `frequently-failing!` function as its body, with a backoff timeout of 10 ms, a maximum number of attempts of 50 and telling the macro to print a warning after 15 failed attempts at executing its body.

Syntax Quote

Although usage of syntax-quote (identified by a back-tick) is not necessarily tied to writing macros, it is the most frequent of its use cases. syntax-quote is the big cousin of `quote`, offering four important features over simple quoting that make it the perfect tool for writing macros: auto-qualification of symbols, `gensym` (`#`), unquote (`~`) and unquote-splicing (`~@`).

1. auto-qualification of symbols is the feature of syntax-quote making it so that every symbol literal used inside a syntax-quote expression will be automatically qualified with the

namespace that symbol resolves to, or the current namespace if it's not possible to resolve it. With an example:

```
(require '[clojure.string :as s :refer [lower-ca

`s/upper-case
;; clojure.string/upper-case

`lower-case
;; clojure.string/lower-case

`foo
;; user/foo
```

2. auto-gensym is one of the two ways to produce unqualified symbols inside a syntax-quote expression: if a symbol ends with a `#`, syntax-quote will automatically append an unique identifier to the end of that symbol, as if by `gensym`:

```
`(let [x# 1] x#)
;; (clojure.core/let [x__37__auto__ 1] x__37__au
```

The only other way to produce an unqualified symbol inside a syntax-quote expression is to use the `~'` (tilde-single-quote) "pattern": by unquoting a quoted symbol, syntax-quote will not

try to qualify or `gensym` that symbol and will instead embed the unquoted symbol:

```
`[foo foo# ~'foo]
;; [user/foo foo__45__auto__ foo]
```

> Warning
> Usage of the tilde-single-quote "pattern" is highly discouraged: the reason for the auto-qualification and `gensym` features is to avoid the age-old problem of LISP macros of accidental symbol capture, without having to implement purely hygienic macro system [83] and this "pattern" sidesteps those safety measures. There are legitimate cases where this is indeed the desired behavior (some instances appear in the `clojure.core` code-base itself), but they are extremely rare and usually only needed in very complex contexts.

---

3. unquote is the escape mechanism that turns syntax-quote into full blown templating engine for Clojure expressions. By prefixing an expression used from within a syntax-quote

context with the unquote symbol `~`, that expression is normally evaluated as opposed to being quoted and the result of its evaluation is embedded into the syntax-quote expression:

```
`[1 2 (+ 1 2)  ~(+ 1 2)]
;; [1 2 (clojure.core/+ 1 2) 3] ; ❶
```

❶Everything inside the square brackets should be quoted and unevaluated but unquote (tilde) temporarily turns on the normal evaluation engine for the form it precedes (including all its inner forms).

4. unquote-splicing is the last powerful feature of syntax-quote: not only does it act like unquote (tilde), but making the assumption that the returned value is a collection, it splices each element of that collection into the outer expression, which must be a collection itself:

```
`[1 2 ~[3 4] ~@[3 (+ 1 2)]]
;; [1 2 [3 4] 3 3] ; ❶
```

❶unquote-splicing (tilde-at) turns on evaluation and treats the following form as a collection.

unquote-splicing will cause an exception to be thrown if:

- Is not used from inside syntax-quote in the first place.
- The value to be spliced is not a collection (or `nil`).
- It's not being used from within a collection.

For instance:

```
[~@[1 2]]
;; java.lang.IllegalStateException: Attempting t

`[~@:foo]
;; IllegalArgumentException Don't know how to cr

`~@[1]
;; IllegalStateException splice not in list  clo
```

The &form and &env implicit arguments

It was previously mentioned that macros have 2 implicit arguments: `&form` and `&env`, while rarely used, they can turn out to be useful writing complex macros. Let's have a look at what they are about:

&form

This one is almost self-descriptive: `&form` will resolve to the original form (as data) that the current macro has been invoked

as. An example is worth a thousand words:

```
(defmacro just-print-me [& args] ; ❶
  (println &form))

(just-print-me foo :bar 123)     ; ❷
;; (just-print-me foo :bar 123)
;; nil
```

❶We define the `just-print-me` macro, which does exactly what its name suggests: it prints the form that is being invoked and returns `nil`.

❷A quick invocation of this macro shows that it's behaving as expected, printing exactly the form that is being invoked.

Some might be tempted to observe that the previous macro could be rewritten without any need for `&form`, like this:

```
(defmacro just-print-me [& args]
  (println (apply list 'just-print-me args)))
```

What's the point of `&form` then? The answer is twofold:

- `&form` is resilient to eventual re-naming. If we were to rename the previous macro, we'd have to rename its body as well to reflect that change.

- `&form` gives us access to the original form metadata. This is perhaps the most important reason for its existence, allowing macro writers to propagate the metadata attached by users of the macro to the produced form. This is especially useful to preserve type-hints attached to macro invocations.

&env

This implicit argument will hold a map of the local bindings available at the point of macro-expansion. The keys will be the local symbols, the values will hold an undocumented internal compiler representation of those local bindings.

There's just a handful of macros that will ever need to use `&env` and an even smaller set of macros that will need to access the value part of this map [84].

Here's an example of a macro that makes usage of `&env` by automatically converting all the available lexical bindings into their `toString` representation:

```
(defmacro with-locals-to-string [& body]
  (let [locals (vec (keys &env))]         ; ❶
    `(let [~locals (mapv str ~locals)]    ; ❷
       ~@body)))
```

```
(let [a 1
      b [:foo :bar]]
  (with-locals-to-string [a b]))          ; ❸

;; ["1" "[:foo :bar]"]
```

❶The `with-locals-to-string` macro retrieves the local symbols available at the point of macro-expansion using `(keys &env)` and puts them into a vector so that it will be possible to use that vector in a destructuring `let`.

❷It then emits a destructuring `let` statement wrapping the body, where every local will be rebound to the result of invoking `str` on itself.

❸Here's an example of how `with-locals-to-string` is used. Unfortunately it's not possible to inspect a macro that uses `&env` using `macroexpand-1` and preserving the lexical context, but this is what that expression will macro expand to:

```
(let [a 1
      b [:foo :bar]]
  (let [[a b] (mapv str [a b])]
    [a b]))
```

Although macros are nowadays a fundamental and iconic feature of all LISPs, macros did not exist in McCarthy's original LISP paper, in fact they only started appearing in LISP implementations more than half a decade after the first LISP implementation [85].

For a few years before macros where proposed, a number of LISP implementations had something similar: `fexpr`, functions that like macros accepted unevaluated arguments but did not evaluate the returned value. The only evaluation that happens in a `fexpr` is that of its body; in Clojure-speak a pseudo `fexpr` could be implemented as a macro that returns a value wrapped in an extra quote:

```
(defmacro deffexpr [name args & body]
  `(defmacro ~name ~args (list 'quote (do ~@body))))

(deffexpr if-pos [test then else]
  (if (pos? (eval test))
    (eval then)
    (eval else)))

(if-pos 1 (println "pos") (println "neg"))
;; pos
;; nil
```

Macros where later proposed and quickly replaced `fexpr` being both easier to reason about for humans and allowing the compiler's to do a better job at optimizing expressions [86]. At this day there are still a small number of minor LISPs that make use of `fexpr` instead of macros, such as newLISP [87] and PicoLisp [88].

The evolution of LISP macros didn't stop with their proposal though, different LISPs had different implementations and thus

offered different behaviors: for example the MIT PDP-6 LISP expanded macros on the fly at function call rather than at function definition. This had the advantage of allowing macros redefinition without requiring redefinition of the functions using those macros, but required the interpreter to expand the same macro call every time, reducing execution speed.

A big jump forward in the LISP macros evolutionary time-line happened in the mid '70 with the introduction of the syntax-quote templating system in ZetaLisp. This allowed macros to be written in a significantly more concise style and allowed also normal people to write macros (at that time writing complex macros was considered an activity that only real gurus could perform).

During the '80 the problem of macro hygiene arose, and caused Scheme to diversify significantly from the other major LISP of that time, Common Lisp. While Common Lisp tried to side-step that problem by instructing programmers to make use of gensym (clojure uses the same style of macros as Common Lisp, but rather than relying on users to make use of gensym, forces them to by auto-qualifying symbols in syntax-quote expressions), Scheme decided that `defmacro` style macros were too hard to write, allowing both arbitrary computation to

happen in macro bodies and forcing users to deal with problems like macro hygiene and manual parsing.

To solve those problems, Scheme ditched `defmacro`-style macro definitions in favor of `define-syntax`, `syntax-rules` and later `syntax-case`. Those primitives allow users to create macros as syntax transformers, by simply defining the input language in a BNF [89] style and declaring a transformation, here's an example of how the `when` macro would be defined in Scheme (note the lack of explicit quoting/unquoting):

```
(define-syntax when
  (syntax-rules ()
    ((when pred body ..)
     (if pred (begin body ..)))))
```

There are several libraries that implement similar functionality in Clojure and Clojure itself will probably include something similar in future releases [90].

See also

- eval is a function that offers the opposite functionality of macros, by taking a quoted expression and evaluating it.

- macroexpand are invaluable functions when debugging or trying to understand a macro, allowing to inspect the result of a macro call sidestepping the evaluator.
- `quote` is a special form used to prevent the compiler from evaluating an expression. Conceptually a macro can be simulated appropriately combining `eval` and `quote`.
- definline blurries the difference between `defn` and `defmacro`, defining a function that can also act as a macro when not used in a higher-order context.

Performance considerations and implementation details

=> `O(1)` **Constant time macro generation**

`defmacro` evaluates at compile time and is not particularly interesting during performance analysis of an application at runtime. Although iteration over the arguments is performed, `defmacro` is constant time for all practical purposes considering the hard limit of 18.

The compiler implements a different behavior when analyzing a function that has the macro-flag set. When such flag is `true` the compiler proceeds to analyze the body (and any further

macro call from within the body) until no further expansion is possible, that is when the innermost var is not flagged as macro anymore. The presence of the recursive analysis step is what distinguish a macro from a normal function which would instead proceed directly to call the generated Java class. macroexpand allows the user to invoke the recursive analysis process stopping just before the evaluation step.

[80] Homoiconicity is the property of a language in which its syntax is represented in terms of data structures of the language itself, see en.wikipedia.org/wiki/Homoiconicity

[81] The core.async library is perhaps one of the best examples, implementing a source-to-source rewriting compiler as a single macro github.com/clojure/core.async

[82] Hygiene in macros has to do with preventing symbols defined outside the macro to collide with what happens inside the macro. For an initial overview on the topic please see en.wikipedia.org/wiki/Hygienic_macro

[83] More on the problem of accidental symbol capture and hygienic macros here: en.wikipedia.org/wiki/Hygienic_macro

[84] An actual real world example of a macro that uses `&env` is core.async's `go` macro, possibly the most complex Clojure macro ever written to date.

[85] For a more in depth analysis of the history of macros in LISPs, refer to chapter 3.3 of "The Evolution of LISP" by Guy

Steele and Richard Gabriel: www.csee.umbc.edu/courses/331/resources/papers/Evolution-of-Lisp.pdf

[86] This is discussed at length in Kent Pitman's 1980's paper "Special Forms in LISP": www.nhplace.com/kent/Papers/Special-Forms.html

[87] www.newlisp.org/

[88] picolisp.com/

[89] Backus–Naur Form, a language for describing the syntax of languages, see: en.wikipedia.org/wiki/Backus–Naur_Form

[90] See the Clojure wiki page on macro grammars to have an idea of the kind of work that is currently in progress: archive.clojure.org/design-wiki/display/design/Home.html)

4.2 MACROEXPAND, MACROEXPAND-1 AND MACROEXPAND-ALL

`function` `since 1.0`

Listing 4.2. → *Macro inspection, REPL util, Debugging*

```
(macroexpand form)

(macroexpand-1 form)
```

```
(clojure.walk/macroexpand-all form)
```

Writing macros can sometimes be a haunting experience and debugging a misbehaving one can be a true nightmare. When one finds themself in either situation, the set of macro expanding functions `macroexpand`, `macroexpand-1` and `clojure.walk/macroexpand-all` are the tools to reach for. The set of macro expanding functions are almost exclusively used at the REPL, either to understand how a macro works or as an aid when writing or debugging one. They all perform the macro expansion steps on the input form, without evaluating the result but instead returning it:

```
(macroexpand-1 '(when false (println "this will
;; (if false (do (println "this will never be pr
```

❶The result of macro expanding a simple when form. Note the required use of syntax quoting (`'`) so the Clojure runtime does not evaluate the form straight away.

The difference between `macroexpand-1`, `macroexpand` and `clojure.walk/macroexpand-all` is in how deep they will macro expand the form before returning:

- `macroexpand-1` just performs macro expansion on the outermost form and returns immediately (thus the `-1` suffix).
- `macroexpand` repeatedly macro expands all the outermost expressions until they no longer returns a macro, but without entering potential inner forms. `macroexpand` is the most frequently used macro expansion.
- `clojure.walk/macroexpand-all` behaves like `macroexpand` but macro expands other forms inside the first if necessary.

Contract

input

- "form" must be a valid Clojure expression.

output

- Returns a macro expanded version of "form", with depth of macro expansion depending on the `macroexpand*` variant used, as described above and illustrated below in the examples.

Examples

Here's an example using all the three `macroexpand*` variants on the same form, showcasing the difference in how they work:

```
(macroexpand-1 '(when-first [a [1 2 3]] (println
;; (clojure.core/when-let [xs__5218__auto__ (clo
;;   (clojure.core/let [a (clojure.core/first xs
;;     (println a)))

(macroexpand '(when-first [a [1 2 3]] (println a
;; (let* [temp__4670__auto__ (clojure.core/seq [
;;   (clojure.core/when temp__4670__auto__
;;     (clojure.core/let [xs__5218__auto__ temp__
;;       (clojure.core/let [a (clojure.core/first
;;         (println a)))))

(clojure.walk/macroexpand-all '(when-first [a [1
;; (let* [temp__4670__auto__ (clojure.core/seq [
;;   (if temp__4670__auto__
;;     (do (let* [xs__5218__auto__ temp__4670__au
;;           (let* [a (clojure.core/first xs__52
;;             (println a))))))
```

❶ `macroexpand-1` runs the macro expander exactly once on the input form, as we can see `when-first` macro expands into a combination of `when-let`, `seq` and `let`.
❷ `macroexpand` loops `macroexpand-1` on the form until the first element doesn't resolve to a macro anymore, in this case it will run 3 times: `when-first` macro expands to a `when-let` expression, `when-let` macro expands to a `let` expression, `let` macro expands to a `let*` expression.
❸ `clojure.walk/macroexpand-all` walks the expression running `macro expand` on each subform, using a breadth-first traversal. All macro calls in the returned form have been fully macro expanded.

While it is true that the macro expansion utilities are almost exclusively used in the REPL for interactive exploration and debugging, they can be useful in code as well to implement really complex macros or tooling utilities. In the following example we use `macroexpand-all` and clojure.walk/walk to find an approximation of all the functions called by another function:

```
(require '[clojure.walk :as w])

(defn find-invoked-functions [expression]
  (let [!fns (atom #{})
```

```
          walkfn! (fn walkfn! [expr]
                    (if (and (seq? expr) (symbol?
                      (let [head (first expr)]
                        (when-not (= 'quote head)
                          (some->> head resolve (s
                          (w/walk walkfn! identity
                      (when (coll? expr)
                        (w/walk walkfn! identity e
      (walkfn! (w/macroexpand-all expression))
      @!fns))

  (find-invoked-functions
    '(when-first [a (vector 1 2 3)]
       (inc a)))

  ;;#{#'clojure.core/vector #'clojure.core/seq #'c
```

❶ `find-invoke-functions` is a function that takes a quoted expression and returns a set of vars that approximates the actual set of functions that that expression references.

❷ `!fns` is an atom that we will use to collect referenced vars while walking the expression.

❸ `walkfn!` is a recursive function that is invoked on each sub-form that could contain function calls and collects invoked functions. It starts by checking whether the sub-expression is a sequence whose first element is a symbol, the syntax for function call in Clojure.

❹ If the sub-expression is a function call we try to resolve the symbol in function position to a var using resolve, if that returns a var we conjoin it to `!fns`, then we recurse `walkfn!` on the sub-expressions using `clojure.walk/walk`. We skip the resursive walk if the symbol in function call is `quote` since nothing inside a `quote` body is evaluated and thus there can be no function referenced.

❺ If the sub-expression is a collection then we recurse `walkfn!` on its content, otherwise we do nothing.

❻ Here we invoke `walkfn!` on the given expression, invoking `clojure.walk/macroexpand-all` on it first to make sure we find all the functions referenced by the expression body.

❼ Finally we invoke `find-invoke-functions` on a simple expression, the result shows a set of `clojure.core` vars. As we can see, the resulting set contains `seq` and `first`, none of which appear explicitly in our expression but are used by the expansion of `when-first`; hadn't we used `clojure.walk/macroexpand-all`, we wouldn't have been able to know they were referenced.

The function just showcased isn't perfect (it won't find functions used as values, for instance), but it's a good example of how we can implement a simple call resolution algorithm without the use of complex analysis tools.

Shotcomings

The `macroexpand*` functions have a couple of known shortcomings that can be potentially surprising and should be kept in mind:

- They're not aware of the surrounding lexical environment, meaning it's not always possible to macro expand macros that make use of `&env`

- `clojure.walk/macroexpand-all` macro expands without taking into account the synctactic rules of Clojure, meaning it will potentially macro expand subforms that should not be macro expanded because they either appear in the body of a special form, or the referenced macro has been shadowed by a local binding

See also

- eval is a function that takes a Clojure expression and evaluates it as code; macro expansion happens as part of the evaluation of a form in the Clojure compiler pipeline.
- read-string is a function that takes a Clojure expression as a string and returns its representation as a Clojure data structure; reading precedes macro expansion in the Clojure compiler pipeline.

- `quote` is a special-form used to prevent the compiler from evaluating an expression, Clojure forms can be passed to `macroexpand*` either through the use of `quote` or through the use of `read-string`.

4.3 QUOTE

`special form` `since 1.0`

Listing 4.3. → *Metaprogramming, Macros, Evaluation*

```
(quote [expr])
```

`quote` is a special form that simply returns its input expression without evaluating it:

```
(quote (+ 1 2)) ; ❶
;; (+ 1 2)

'(+ 1 2) ; ❷
;; (+ 1 2)
```

❶We can quote an expression by just calling `quote` on it.
❷Alternatively, Clojure also provides an equivalent reader macro `'` (single quote).

As with all the utilities that affect how Clojure forms are evaluated, `quote` is mostly useful in metaprogramming contexts. Because of how primitive `quote` is to the language, Clojure provides a shortrand syntax to quote expressions via the reader macro `$$'ff`. In other words `(quote foo)` can be conveniently re-written using the more concise and equivalent syntax `'foo`.

Contract

input

- "expr" is the required and only argument.

output

- returns: the argument that was passed as input, unevaluated.

Examples

Because of the evaluation rules of Clojure, if a symbol points to a var, then the value of that `var` is dereferenced in place. Using `quote` is the only way to embed lists and symbol literals in code. Literal symbols are used for all kind of purposes in

Clojure, most frequently as input to functions that provide runtime introspection functionalities such as resolve:

```
(resolve '+)
;; #'clojure.core/+
```

Without the `quote` special form, one would be forced to write that call as:

```
(resolve (read-string "+"))
;; #'clojure.core/+
```

This is not only more cumbersome to write, but also less performant: rather than embedding a constant at compile time, this will force Clojure to parse the string and create a new symbol every time that expression is evaluated.

Besides the more common usage of embedding symbol literals in code, `quote` is sometimes used in macros inside complex syntax-quote expressions as an escape hatch for its automatic namespace qualification feature, via the "unquote-quote" pattern `~'`. To showcase this pattern we define a macro called `defrecord*` which augments defrecord by making it implement the `clojure.lang.IFn` interface, so that records created with `defrecord*` are callable just like maps:

```
(defmacro defrecord* [name fields & impl]
  `(defrecord ~name ~fields
     ~@impl
     clojure.lang.IFn
     (~'invoke [this# key#]
      (get this# key#))
     (~'invoke [this# key# not-found#]
      (get this# key# not-found#))
     (~'applyTo [this# args#]
      (case (count args#)
        (1 2) (this# (first args#) (second args#
        (throw (AbstractMethodError.))))))

(defrecord* Foo [a])

((Foo. 1) :a)
;; 1

((Foo. 1) :b 2)
;; 2
```

❶ We define the `defrecord*` macro, taking as input the record name, fields and default implementations and we insert those args into a `defrecord` expression

❷ After the provided record impl, we implement `clojure.lang.IFn` and the two arities of its `invoke` method that just delegate to `get`. Here we make use of the unquote-quote pattern so that the method name will be `invoke` rather than `user/invoke`

❸ Similarly we implement the `applyTo` method so that we can also use `apply` on our record.

❹ We can verify that our macro is doing what it's supposed to by instantiating an example record and invoking it as a function.

See also

- eval is a function that takes as input a quoted expression and returns its evaluated value
- syntax-quote is a reader macro that can be considered `quote` on steroids and the go-to tool for writing macros

Performance considerations and implementation details

=> O(1) **constant time**

`quote` is basically a no-op, as all it does is returning a constant value, it doesn't have any performance implications.

4.4 GENSYM

`function` `since 1.0`

Listing 4.4. → *Macro utility, Random symbol generation*

```
(gensym
  ([])
  ([prefix-string]))
```

`gensym` is a simple function whose only purpose is to return an unique symbol each time it's invoked. It's mainly used in the context of writing macros to avoid the problem of accidental symbol capture when the automatic symbol generation feature of syntax-quote is not enough, but can be used in any reason when there's need for a random symbol, such as generating unique labels.

```
(gensym) ; ❶
;; G__14
```

```
  (gensym "my-prefix") ; ❷
  ;; my-prefix17
```

❶ `gensym` without argument returns a symbol prefixed by "G__".

❷If we provide a prefix, the generated symbol starts with the given prefix instead.

Contract

input

- "prefix-string" is the only optional argument. If no prefix is provided, "G__" will be used as prefix.

output

- `gensym` returns a symbol whose name is a prefix followed by a random number guaranteed to be unique in the current Java instance.

Examples

Here's an example showcasing `gensym` while manipulating symbolic expressions for a small logic language. First order logic [91] allows for quantification of logic variables with quantifiers such as "any" (there is at least one item for which

the expression is `true`) and "all" (the expression should be `true` for all items). Expressions written in first order logic are amenable for transformations that maintain logic equivalence between formulae. One of them allows to pull a quantified expression "up":

```
(OR (EXIST x (Q x)) (P y)) ; ❶
```

❶The quantifier "EXIST" applies to just one expression within "OR".

The logic formula above reads: either there is at least one "x" such that "Q" of "x" is `true` or "P" of "y" is `true`. "Q" and "P" represents logic predicates. Logic predicates are similar to functions: they take a logic variable (such as "x" or "y") and they evaluates as `true` or `false` in a logic expression. We can claim that this expression is logically equivalent to another using the "<=>" symbol (which means "if and only if"):

```
(OR (EXIST x (Q x)) (P y)) <=> (EXIST x (OR (Q x
```

❶Two logic expressions are logically equivalent when they evaluate the same given the same "x", "y" input.

Our goal is to write a Clojure function that "pulls up" a nested quantifier in a logic formula, so the quantifier appears external to the expression, like illustrated by the logic equivalence above. One problem related to this transformation is the potential accidental capturing of logic variables. Observe the following:

```
(OR (EXIST x (Q x)) (P x)) <!=> (EXIST x (OR (P x
```

❶The accidental capture of "x" does not guarantee equivalence between these expressions. In one case "x" is quantified but the quantification should not be extended to other predicates arbitrarily.

In the last example, the predicate "(P x)" suddenly becomes part of the quantification of the variable "x" when previously it wasn't, breaking the logic equivalence between the expressions. We need to make sure that when transforming the expression, we change the quantified variable to avoid accidental capture. We can achieve this by using `gensym` as follows:

```
(defn- quantifier? [[quant & args]] ; ❶
  (#{'EXIST 'ALL} quant))
```

```
(defn- emit-quantifier [op expr1 expr2] ; ❷
  (let [new-local (gensym "local")     ; ❸
        [quant local [pred _]] expr1]
    `(~quant ~new-local (~op ~expr2 (~pred ~new-

(defn pull-quantifier [[op expr1 expr2 :as form]
  (cond (quantifier? expr1) (emit-quantifier op
        (quantifier? expr2) (emit-quantifier op
        :else form))

(pull-quantifier '(OR (EXIST x (Q x)) (P x)))
;; (EXIST local2747 (OR (P x) (Q local2747)))

(pull-quantifier '(OR (P x) (EXIST x (Q x))))
;; (EXIST local2750 (OR (P x) (Q local2750)))
```

❶The function `quantifier?` returns `true` if the argument is a sequence starting with either "EXIST" or "ALL".
❷ `emit-quantifier` assembles a new quantified expression given a quantifier and the original expressions.
❸To assemble the new expression, `emit-quantifier` makes sure the quantified variable is brand new, so it cannot clash with an already existing variable in either expressions.
❹At the same time we need to make sure the predicate that was originally part of the quantified expression also receives the newly create variable name. Note how the final expression is assembled easily using syntax-quote.
❺Callers perform transformations using the `pull-quantifier` function. This function understands which expression contains the quantifier and calls `emit-quantifier` accordingly.

The call to `pull-quantifier` with an example expression verifies the transformation performs substitutions as expected. The generated symbols have the "local" prefix and a number. Please note that based on the age of your REPL, this number could be arbitrarily big (but always monotonically increasing).

See also

- defmacro is a macro where `gensym` is mostly used in, to generate unique symbols at macro expansion time that the macro will use in its expansion.
- symbol is a function used to create a symbol at runtime from a given name.

Performance considerations and implementation details

=> `O(1)` **constant time**

Internally `gensym` uses a `java.util.concurrent.atomic.AtomicInteger` instance to generate unique numbers across the running application instance. As such, whenever it's invoked, it causes a memory barrier to be used by the JVM. However, on modern JVM implementations, atomic operations are implemented in hardware and they should not be of any concern in performance analysis.

[91] First order logic is a formal system for logic reasoning. Compared to other kind of formal systems (such as propositional logic) first order logic also allows quantification of logic expression over collection of items. Please see en.wikipedia.org/wiki/First-order_logic for more information

4.5 DEFINLINE

`experimental` `macro` `since 1.0`

Listing 4.5. → *Function declaration, Function inlining, Macro Definition, Compiler optimization*

```
  (definline [name & decl])
```

`definline` is a macro that takes a function body and expands it into a standard `defn` declaration that also includes an "inlined" version of the same body definition [92]. `definline` is equivalent to declaring a function that also includes an `:inline` metadata keyword for the same body. In this sense `definline` behaves similarly to defmacro, providing a compile-time version of the function along with a standard run-time version.

Typical and effective use of `definline` is bridging Clojure and Java offering a Clojure entry-point over some interoperability call. `definline` allows clients to correctly propagate type information (usually to boost performance) and removes the need to add a secondary function implementation through the `:inline` metadata keyword.

Warning

As of Clojure 1.10 it is the only remaining instance of "experimental" declaration in the clojure.core namespace. Experimental should be read as "use at your own risk". `definline` had at least one serious issue related to AOT compilation [93] on Clojure versions prior to 1.6.0 and may be replaced with a different solution in future Clojure releases. Lisp-like [compiler-macros] are being considered for example.[94] Despite these problems, `definline` and the `:inline` keyword are widely used in small and big projects.[95]

A macro-expansion will clarify how `definline` works. Here's a `timespi` function to multiply a number to Pi and its macro expanded form:

```
(definline timespi [x]  ; ❶
  `(* ~x 3.14))

(timespi 3)             ; ❷
;; 9.42

(macroexpand-1 '(definline timespi [x] `(* ~x 3.
```

```
;; after removing core namespaces

;; (do
;;   (defn timespi [x]                ; ❹
;;     (* x 3.14))
;;   (alter-meta! (var timespi)       ; ❺
;;                assoc :inline
;;                (fn timespi [x]
;;                  (seq
;;                    (concat (list (quote *))
;;                            (list x)
;;                            (list 3.14)))))
;;   (var timespi))
```

❶The `timespi` function is defined using the `definline` macro

❷To use the `timespi` function we just invoke it as usual

❸Prints the macro-expansion of the same `timespi` function

❹ `definline` expands into the expected definition of the `timespi` function exactly like we would define that normally

❺But `definline` also alters the metadata definition of the var to add the `:inline` key. The `(fn timespi [x]…)` function definition that can be seen here is just a very elaborated equivalent of `(* x 3.14)`.

When the inlined version of the function is just the same as the body (at least for a subset of the arities), using the `:inline` metadata keyword creates a duplication that `definline` can take care of. Like the metadata keyword, `definline` allows the compiler to treat a function differently based on the way it is invoked. Direct invocations of the inlined function will be expanded similarly to macros, while high-order uses in which the function is passed as an argument will be treated like any other function definition.

The main use case of function inlining has to do with performance optimizations during Java interoperability (commonly referred as "interop"). With an inlined version of a function the compiler has a chance to use the presence of type hints to make calls to the right Java method (when many overrides are present). Without the inlined version, Clojure would have to wrap the primitive Java type argument into a `java.lang.Object`.

Contract

input

- "name" is the name of the function that `definline` will generate as part of the macro expansion. The name should be a valid symbol as per Clojure Reader rules [96].
- "&decl" despite the presence of "&", "decl" is not really optional. Because `definline` hash to expand into a `defn` declaration, "decl" must contain at least a vector (representing the list of parameters for the function). So: `(definline f [])` is perfectly valid but `(definline f)` is not permitted.

output

- `definline` returns a `clojure.lang.Var` object pointing at the function that was just declared. The function is created in the current namespace, so there is normally no need to hold on the returned var to invoke the function.



Note

Unlike normal macros and functions, `definline` doesn't support multiple arities.

Examples

The following example is going to explore a hypothetical integer math Java library that we want to use from Clojure. To keep things simple, the library accepts different numeric types but only output integers. The math library contains a `plus` method that is overloaded for boxed numbers (e.g. `java.lang.Integer`) and also for primitive types (int). It also contains a catch-all `plus` method that accepts generic `java.lang.Object` as a last resort for other types that can be cast to `java.lang.Number`:

```
public class IntegerMath { ; ❶

  public static int plus(Object op1, Object op2)
    System.out.println("int plus(Object Object)"
    return ((Number) op1).intValue() + ((Number)
  }

  public static int plus(Integer op1, Integer op
    System.out.println("int plus(Integer Integer
    return op1 + op2;
  }

  public static int plus(int op1, int op2) {
    System.out.println("int plus(int int)");
    return op1 + op2;
  }
```

```
    //[...] other types
  }
```

❶The `IntegerMath` Java class, simulates a fast math library that we wish to use from a Clojure program.

Our goal, as the developers of the Clojure layer on top of the `IntegerMath` class, is to be able to invoke the right `plus` method based on the inferred or explicit type. This also includes the possibility for clients to call the native-unboxed `int` option if needed. Finally, we would like to hide all of the complexity of the Java interoperability to the developers of the Clojure application. To achieve this isolation we design the following intermediate layer:

```
(ns math-lib ; ❶
  (:import IntegerMath))

(defn plus [x y] ; ❷
  (IntegerMath/plus x y))
```

❶A Clojure namespace that hides the complexity related to invoking methods on the Java class.
❷Clojure clients wishing to use the `IntegerMath` class only see a `plus` function of 2 arguments.

The `math-lib` namespace is designed to be the public interface to clients wishing to use the `IntegerMath` class. The following example illustrates the use of the namespace to sum a number to a list of other numbers:

```
(ns math-lib-client
  (:require [math-lib :as m]))

(defn vsum [x xs]
  (map #(m/plus x %) xs))     ; ❶

(vsum 3 [1 2 3])

;; int plus(Object Object)    ; ❷
;; int plus(Object Object)
;; int plus(Object Object)
;; (4 5 6)
```

❶the client code requires the library and executes a sum of some numbers without any knowledge that Java-interop is required for this operation.

❷printouts are showing that we end up calling the generic plus of objects instead of the more specialized integer version

Clojure doesn't have a clue about what kind of sum `vsum` is executing once everything is compiled: it could be summing up

boxed or unboxed numbers, floats or integers. The reason why this information is missing is because `plus` was compiled to a Java class with an `invoke` method that accepts and return Objects. An attempt at coercing types would not work either since the `math-lib` library is already compiled, as demonstrated by the following:

```
(ns math-lib-client
  (:require [math-lib :as m]))

(defn vsum [x xs]
  (map #(m/plus (int x) (int %)) xs)) ; ❶

(vsum 3 [1 2 3])

;; int plus(Object Object)            ; ❷
;; int plus(Object Object)
;; int plus(Object Object)
;; (4 5 6)
```

❶the only change was to cast x and the item from the vector to be integer, but still the compiler won't take advantage of this.

❷despite the type coercion to int, we are still calling into the generic plus version of the Java method

`definline` allows type discovery at compile-time, offering the client a way to communicate type information to the compiler. With `definline` we can reformulate the `plus` function as:

```
(ns math-lib
  (:import IntegerMath))

(definline plus [x y] ; ❶
  `(IntegerMath/plus ~x ~y))
```

❶Rewriting of the `plus` function in `math-lib` using `definline`. Note the similarity to macro writing.

Now `plus` is expanded in the place of the invocation, where information about the types are still available for use:

```
(ns math-lib-client
  (:require [math-lib :as m]))

(defn vsum [x xs]
  (map #(m/plus (int x) (int %)) xs))

(vsum 3 [1 2 3]) ; ❶
;; int plus(int int)
;; int plus(int int)
```

```
;; int plus(int int)
;; (4 5 6)
```

❶The new printout confirms `plus` is now routed to the more specific Java method for unboxed integers.

Definline and code reloading

`definline`, like any other macro-based expansion, can lead to surprises while code-reloading. Code reloading often happens at the REPL for development or in production for specific situations. To illustrate this fact, let's use a simple function to square a number and then change it:

```
(definline sq [x]                          ; ❶
  `(let [x# ~x] (* x# x#)))

(defn direct-use [x]                       ; ❷
  (sq x))

(defn higher-order-use [xs]                ; ❸
  (map sq xs))

(direct-use 2.0)
;; 4.0

(first (higher-order-use [2.0]))
;; 4.0
```

❶ `sq` simply multiplies its argument by itself. The let form and the apparent redefinition of the symbol "x" is there to prevent double evaluation (common practice for generic macro programming, since "x" could be an entire form including side effects). The "#" pound sign suffix in a macro is syntactic sugar for gensym.

❷ `direct-use` is a function invoking `sq` directly

❸ `higher-order-use` is a function that passes `sq` to another function, in this case map

As expected direct use and higher order use returns the same results. Clojure compiles the direct use of the function using the inlined form, effectively replacing direct invocation of `sq` with its macro expanded form. `direct-use` function above is effectively replaced by:

```
(defn direct-use [x]
  (let [x__1 x]
    (* x__1 x__1)))
```

This is the reason why `definline` needs to use macro syntax, because it will be treated similarly to a macro-expansion at compile time to replace all direct uses of the function. Now let's assume the situation where we are playing at the REPL to solve a problem. We decide that the square function must return integers and we cast the result using int. One very common thing at the REPL is to go back to the definition of the function in local history, change what we want to change and re-evaluate

the function, which is exactly what are going to do below, without redefining `direct-use`:

```
(definline sq [x]                    ; ❶
  `(let [x# ~x] (int (* x# x#))))

(direct-use 2.0)
;; 4.0

(first (higher-order-use [2.0]))
;; 4
```

❶Note that the new definition of `sq` is just the same as before with an added cast to int.

As you can see, `direct-use` does not truncate the return value to be an integer, while the higher-order version is returning "4" as expected. The same would happen changing a macro and forgetting to re-evaluate the functions using it, a common "reloading" problem. In a simple example like this one it’s easy to see why this is happening, but in much bigger namespaces, whose dependency graphs are evaluated at the REPL, this behavior can trip you up.

See also

- memfn is a good choice when wrapping calls to instance methods of Java objects for use in higher order functions. `definline` has a similar effect with a better control of type passing at the cost of an additional function to write. For example, the following invocations of the `toString` method on a Java object are equivalent. Prefer the `memfn` solution in this case:

```
(map (memfn toString) [(Object.) (Object.
;; ("java.lang.Object@65b38578" "java.lar

(definline to-string [o]
  `(.toString ~o))

(map to-string [(Object.) (Object.)])
;; ("java.lang.Object@4ea61560" "java.lar
```

- defmacro if the logic of the function is mostly related to the compile time aspect (as a macro) and the higher-order function is never used, consider using defmacro instead to make explicit that the only intended use of the function is as a macro.

[92] Function inlining is an internal process by which a compiler replaces a function invocation with the body of that function at compilation time. More information is available on Wikipedia: en.wikipedia.org/wiki/Inline_expansion

[93] see clojure.atlassian.net/browse/CLJ-1227

[94] See archive.clojure.org/design-wiki/display/design/Inlined%2Bcode.html on the Clojure wiki

[95] A partial list of projects making use of inlining is available on the Clojure mailing list: groups.google.com/d/msg/clojure-dev/UeLNJzp7UiI/WA6WALO6EPYJ

[96] See the main Clojure Reader documentation at clojure.org/reference/reader

4.6 DESTRUCTURE

`function` `since 1.0`

Listing 4.6. → *Code transformation, Destructuring, Macro helper*

```
  (destructure [bindings])
```

`destructure` is an undocumented function in the Clojure standard library that is used by macros like fn and let to augment their special-form primitives fn* and let* with support for destructuring. It can be considered a source-to-source compiler taking as input a destructuring expression and

emitting as output normal Clojure let pairs used to produce those bindings. Here's an example:

```
(destructure '[[x y] [1 2]]) ; ❶

; [vec__14 [1 2]
;  x (nth vec__14 0 nil)
;  y (nth vec__14 1 nil)]
```

❶ `defstructure` returns the form that when evaluated produce the destructuring of a collection type (in this case vector).

We can demonstrate that the output of `destructure` works as expected by using it in a let binding:

```
(eval
  `(let ~(destructure '[[x y] [1 2]]) ; ❶
     (+ ~'x ~'y)))
;; 3
```

❶We can compose a let expression using syntax-quote and decide what kind of destructuring to use programmatically.

Contract

Destructuring expressions can get very complex, the syntax supports a lot of different options and can be arbitrarily nested; here's our attempt at a pseudo-formal specification of it:

```
(destructure [bindings])

bindings :->
  [bind1 expr1 .. bindN exprN]

bind :->
  sym
    OR
  vec-bind
    OR
  map-bind

vec-bind :->
  [bind1 .. <& bindN> <:as sym>]

map-bind :->
  {<:keys [qbind1 .. qbindN]>
   <:strs [sym1 .. symN]>
   <:syms [qbind1 .. qbindN]>
   <:or {sym1 expr1 .. symN exprN}>
   <:as sym>
   <bind1 expr1 .. bindN exprN>}
```

- "sym" is any non-namespace-qualified Clojure symbol.
- "qbind" is any symbol or keyword.
- "expr" is any Clojure expression
- "vec-bind" represents sequential destructuring expressions, each binding "expr" will be bound to the nth element of the matching expression, it supports "tail destructuring" (where all items after the first are grouped together) via the `&` symbol and collection aliasing via the `:as` keyword.
- "map-bind" represents associative destructuring expression, it supports a multitude of different options: `:strs`, `:keys` and `:syms` can be used to destructure, respectively, string keys, keyword keys or symbol keys in a map, `:or` can be used to provide default values in case the map to destructure doesn't contain a specific key, `:as` can be used to alias the original collection and normal symbols can be used to destructure specific keys in a map.

While the above specification describes the synctactic contract of destructuring expressions, their semantic meaning will be explained on a per-case basis in the examples [97].

Examples

Sequential destructuring

Sequential destructuring works over any collection type that implements the concept of sequential ordering, this includes Clojure sequences and vectors, strings, Java arrays and lists. It is used to efficiently and concisely alias the nth or nthnext elements of a collection, without having to explicitly access each element at its index. For example:

```
(let [my-vec [1 2 3 4]
      [a b] my-vec                ; ❶
      [_ _ & r] my-vec            ; ❷
      [_ _ c d e :as v] my-vec]  ; ❸
  [a b c d e r v])
;;[1 2 3 4 nil (3 4) [1 2 3 4]]
```

❶This is the simplest usage of sequential destructuring: the destructuring expression `[a b]` is applied to the vector `[1 2 3 4]`, causing `a` and `b` to be bound to `1` and `2`, the rest of the vector is ignored.

❷This destructuring expression uses the "tail destructuring" feature of sequential destructuring via the `&` symbol: after ignoring the first two elements of the vector, `r` is bound to the remainder of the collection as per nthnext (meaning that if the sequence is over, `r` will be bound to `nil` rather than to an empty sequence). Note that `_` is not a special symbol used in destructuring, it's just a regular local binding name that is idiomatically used for values that we're not interested in.

❸Finally this destructuring expression uses the "collection aliasing" feature via the `:as` keyword: `v` will be bound to the original collection being destructured, preserving its original type and metadata (if applicable). This destructuring expression also showcases how it's possible to destructure more elements than there are in the destructured collection: in this case `e` will be bound to nil.

Sequential destructuring finds lots of usage in idiomatic Clojure programs, a common pattern is to use it when traversing sequences using via recursion, here's an example definition of `dedupe-string`, a function that removes consecutive duplicate characters in a string:

```
(defn dedupe-string [s]
  (loop [[el & more] s                          ; ❶
         [cur ret :as state] [nil ""]]          ; ❷

    (cond
      (not el)                                  ; ❸
      (str ret cur)
      (= el cur)                                ; ❹
      (recur more state)
      :else                                     ; ❺
      (recur more [el (str ret cur)])))))

(dedupe-string "")
;; ""
(dedupe-string "foobar")
;; "fobar"
(dedupe-string "fubar")
;; "fubar"
```

❶The function is implemented as a loop over the string, during each step of the loop we want to consider the first character of the remaining string so we use destructuring to split apart the first character (bound to `el`) from the rest of the string (bound to `more`).

❷The loop also needs to keep some internal state representing the character we're currently deduping and the deduped string it's built so far. We use destructuring to bind the current char to `cur` (initialized to `nil`), the deduped string to `ret` (initialized to the empty string) and aliasing the whole state vector to `state`.

❸We're in the body of the loop now, if `el` is `nil` it means the string has been fully consumed, so we exit the loop by concatenating the current deduped string with the last char being deduped.

❹If there is a char to consider and it's the same as the char being deduped, we simply `recur` on the remainder of the string and we keep the state unaltered, discarding the current char.

❺If there the current char to consider is not the same as the char being deduped, we `recur` on the remainder of the string and we update `cur` to be the current character and `ret` to be the concatenation of `ret` and `cur`.

Associative destructuring

Associative destructuring works over any collection that implements the concept of key-value pairs, this includes Clojure maps, sets, vectors, records and strings. It is used to efficiently and concisely extract and alias values from associative collections:

```
(let [my-map {:x 1 :y 2 :z nil}
      {x :x y :y :as m} my-map            ; ❶
      {:keys [x y]} my-map                ; ❷
      {:keys [z t] :or {z 3 t 4}} my-map] ; ❸
  [x y z t m])
;; [1 2 nil 4 {:x 1, :y 2, :z nil}]
```

❶ This is the simplest example of associative destructuring: the destructuring expression `{x :x y :y :as m}` is applied to the map `{:x 1 :y 2 :z nil}`, causing `x` and `y` to be bound to `1` and `2` respectively. The other values in the map are ignored. The `:as` keyword causes `m` to be bound to the original collection, just like in sequential destructuring.

❷ Because we usually want to bind the value of the key of a map to a symbol with the same name, to avoid the repetition we can use `:keys` (when the keys are keyword, we could use `:strs` or `:syms` when the keys are strings or symbols, respectively). The destructuring expression `{:keys [x y]}` is equivalent to `{x :x y :y}`

❸ `:or` can be used to provide default values when destructuring keys that are not present in the destructured collection, as per contains?. In this case `t` will be bound to `4` while `z` will be bound to `nil`.

Because of the increasing preference to use namespaced keywords in map, since Clojure `1.6.0` `:keys` and `:syms` have been improved to support destructuring of namespaced keywords and symbols:

```
(let [{:keys [::x foo/bar]} {::x 1 :foo/bar 2}]
  [x bar])
```

```
  ;; [1 2]
```

❶The double colon "::" notation denotes a keyword qualified with the current namespace. So if this expression was evaluated at the repl in the `user` namespace, `::x` would be equivalent to `:user/x`.

Because Clojure encourages the use of maps for named or optional args to functions (over the more typical lISP keyword args), map destructuring is very commonly found in the arguments of function definitions.

Nested and composed destructuring

Both sequential and associative destructuring expressions can be composed and arbitrarily nested. Deeply nested destructuring expressions can quickly become hard to read, so idiomatic Clojure usually doesn't nest more than 2 destructuring expressions. Here's for example a destructuring `extract-info` function that takes keys like `:address` or `:contacts` in a map and additionally destructure them:

```
  (defn extract-info [{:keys [name surname] ; ❶
                       {:keys [street city]} :addr
                       [primary-contact secondary-
```

```
    (println name surname "lives at" street "in" c
    (println "His primary contact is:" primary-con
    (when secondary-contact
      (println "His secondary contact is:" seconda

  (extract-info {:name "Foo" :surname "Bar"
                 :address {:street "Road Fu 123" :
                 :contacts ["123-456-789", "987-65

  ;; Foo Bar lives at: Road Fu 123 Baz
  ;; His primary contact is: 123-456-789
  ;; His secondary contact is: 987-654-321
```

❶First we extract "name" and "surname" from the input map using `:keys` destructuring.

❷Without closing the first destructuring, we further destructure ":address" into "street" and "city".

❸Finally, ":contacts" are subject to further sequential destructuring.

Destructured vectors of arguments are also useful to describe the shape of the input data structure in the function's signature, since they will be included in the output of doc.

See also

- let is arguably the macro where destructuring is used more frequently, as destructuring reduces the mental overhead of having to extract values out of nested collections.
- fn also supports destructuring in its argument vectors by relying internally on `destructure`. Keyword arguments support can be achieved by combining varargs and associative destructuring, since destructuring a sequence using associative destructuring just converts the sequence to a map as per `(apply hash-map the-sequence)`.
- loop, doseq, for and all the other macros that support argvecs or binding vectors support destructuring, as they usually build on top of either let or fn.

Performance considerations and implementation details

`destructure` is optimized to perform similarly to the same data lookup written explicitly:

- Sequential destructuring has the same performance characteristics as repeatedly using nth on the input collection (and using nthnext for tail destructuring).

- Associative destructuring has the same performance characteristics as repeatedly using get on the input collection.

[97] For a more in-depth guide on destructuring, refer to: clojure.org/guides/destructuring

4.7 CLOJURE.TEMPLATE/APPLY-TEMPLATE

`function` `since 1.0`

Listing 4.7. → *Code transformation, Symbolic substitution*

```
(apply-template [argv expr values])
```

`clojure.template/apply-template` is a function that performs symbolic substitution in Clojure expressions:

```
(require '[clojure.template :refer [apply-templa

(apply-template '[x y] '(+ x y x) [1 2]) ; ❶
; (+ 1 2 1)
```

❶A simple example of `apply-template` after requiring the function from the `clojure.template` namespace.

`apply-template` has some specific use cases in macro programming and symbolic manipulation in general.

Contract

input

- "argv" is a vector of symbols.
- "expr" is a valid Clojure expression that potentially contains one or more instances of some of the symbols in "argv".
- "values" is a collection of Clojure values that will be used to replace in "expr" the symbols at the matching position in "argv"

output

`apply-template` returns an expression that is the same as "expr" but with the symbols in "argv" that have a matching value in "values" replaced by the matching value

Note

if `(count argv)` is not the same as `(count values)`, only the symbols that can be matched with a value will be replaced. Any extra symbols in "argv" or extra expressions in "values" will be ignored.

Examples

While its docstring explicitly states that its main usage should be in macros, it is actually not good idea to use `apply-template` in that scenario unless its mechanism of action is completely understood, as it can lead to some unexpected results. `apply-template` expands lexically without knowledge of a specific form semantic. For example:

```
(require '[clojure.template :refer [apply-templa

(apply-template '[x] '(let [x x] x) [1]) ; ❶
; (let [1 1] 1)
```

❶A small example of `apply-template` expansion shows that using the same symbol "x" in both arguments generates incorrect Clojure.

Even though this small example is sufficiently trivial to understand what is going on and what the fix should be ("expr" should not use binding symbols that appear in "argv"), issues like this one could happen if the "expr" is provided by users of macros that use this function.

For this and other reasons, the original author of `apply-template` has in more occasions stated that its inclusion in the Clojure standard library might have been a bad idea [98].

For other cases, `apply-template` could be useful tool to apply simple substitutions. For example, here's how to replace the variable "x" with "y" in an arbitrarily nested expression:

```
(apply-template '[x] '(P(x) ∧ (∃ x Q(x))) '[y])
;; (P (y) ∧ (∃ y Q (y)))
```

❶We use `apply-template` to replace all occurrences of the symbol "x" with "y" in a logic expression.

See also

- clojure.template/do-template is a macro that uses `clojure.template/apply-template` to expand the same template "expr" multiple times, using a diffent set of values as substitute for the symbol in "argv".
- postwalk-replace is a function that deep walks Clojure expressions replacing matching exrepssions along the way. It's a more general version of `clojure.template/apply-template`

Performance considerations and implementation details

=> `O(n)` **linear time**

Like all the code walking functions, `clojure.template/apply-template` is linear on the number of values in "expr". The number of symbols in "argv" or the number of values in "values" don't affect its performance.

[98] grokbase.com/t/gg/clojure/124q5bb8y1/stackoverflowerror-caused-by-apply-template#20120423oadz7ag6ufqed27u2jsxsk5e64

4.8 CLOJURE.TEMPLATE/DO-TEMPLATE

`macro` `since 1.0`

Listing 4.8. → *Code transformation, Symbolic substitution*

```
(do-template [argv expr & values])
```

`clojure.template/do-template` is a macro that repeatedly executes a template expression each time replacing the template symbols with provided substitution values:

```
(require '[clojure.template :refer [do-template]

(do-template [x] (println x) 1 2 3)
;; 1
;; 2
;; 3
;; nil
```

`do-template` is equivalent to evaluating apply-template multiple times with different substitutions. As you can see from the example, `do-template` always returns `nil` hinting at the fact that side effects (such as printing to standard output) are required.

Contract

input

- "argv" is a vector of symbols.
- "expr" is a valid Clojure expression that potentially contains one or more instances of some of the symbols in "argv"
- "values" is a collection of Clojure values that will be partitioned by the count of "argv" and each partition will be used to replace in "expr" the symbols at the matching position in the current partition.

output

- `do-template` repeatedly executes the template expression substituting the symbols in "argv" with the matching value in the current partition of "values". It returns `nil`.

Examples

The following is a simple example that prints the same expression with different substitutions:

```
(require '[clojure.template :refer [do-template]

(do-template [x] (println '(P(x) ∧ (∃ x Q(x)))) 
```

```
  ;; (P (y) ∧ (∃ y Q (y)))
  ;; (P (z) ∧ (∃ z Q (z)))
```

❶We need a side effecting function such as println to see the effects of `do-template`. Also note that some white spaces in the result are not present in the original form.

`do-template` has the same shortcomings and issues as clojure.template/apply-template and because of that, usage of `do-template` for macro substitutions is discouraged unless its limitations and issues are properly understood. If similar functionality is needed, a better solution would be to reach for something like the contrib library `core.unify` [99], a proper unification library that provides templating function that have explicit expansion points.

See also

- clojure.template/apply-template is a the function that is used by `clojure.template/do-template` to perform the template expansion
- clojure.test/are is a `clojure.test` macro that uses `clojure.template/do-template` to repeatedly test an expression using different values

Performance considerations and implementation details

=> `O(n)` **at macro expansion time ("n" number of values)**

=> `O(n)` **at runtime**

`do-template` has a linear dependency in the number of substitution values both at compile time as well as run time.

[99] github.com/clojure/core.unify

4.9 SUMMARY

In this chapter we've seen how to create macros, an elegant tool to interact with the Clojure compilation process. Macros are powerful, but they can be hard to understand at first because of their special syntax. defmacro contains a summary of all the essential parts of a macro while macroexpand is our tool of choice for macro debugging. `quote` (which yields the unevaluated form), apply-template, do-template and gensym are also essential tools for macro development. definline has limited use related to performance improvement for Java interoperability. We've also seen destructure which is the destructuring engine powering forms like fn or let.

5OPERATIONS ON NUMBERS

5.1 OVERVIEW

Arithmetic operations are a fundamental feature of a language. This chapter collects the main arithmetic operations offered by Clojure. The list might seem quite small and the reason is that Clojure is neither re-implementing nor wrapping the vast selection of math functions that Java offers. If you are for example searching a function to truncate decimals or square root a number they can be easily leveraged through Java interoperability.

Clojure still offers explicit versions of the most common math operations in the standard library. This is mainly to provide optimal performance without requiring explicit type hints. The operations Clojure provides are part of this chapter and are summarized by the following table:

+	Are the 4 basic arithmetic operations. Unlike Java they throw exception on overflow.
inc	Are commonly used shortcuts for incrementing and decrementing numbers by one.
quot	Clojure offers one function to retrieve the quotient of a number and two types of reminder operations.
max	Calculates the max and min between a set of numbers.
max-key	Calculates the max or min after applying a transformation function.
rand	Generation of random numbers.
with-precision	Utilities to set the rounding strategies for decimals operations.

+'	Core set of arithmetic operations with auto-promoting capabilities (note the single quote appended to the name).
unchecked-add and others `unchecked-*` functions	Java style arithmetic on longs. Subject to truncation on overflow. This is the way Clojure can call the corresponding Java basic math operators.
unchecked-add-int and others `unchecked-*-int` functions	Java style arithmetic on ints. Also subject to potential truncation on overflow.
pos?	Predicates to determine if a number is positive, zero or negative

For many applications default precision for integer types is good enough. But there are classes of applications that require representation of numbers bigger than 2^{63} (the largest long number representable with 64 signed bits). If that's is the case, "single-quote" operators will make your life much easier compared to Java. Java big-integer arithmetic is based on classes and objects without overloaded math operators, which

means that there is no easy way to sum up two `BigInteger` other than creating their respective instances and calling methods on them. Clojure would automatically use the right precision just by using `+'` .

> Note
> You might have noticed that Clojure arbitrary precision operators are missing a /' (divide-quote) equivalent. You should consider that / is already special because it potentially produces fractional numbers that already preserve all possible precision (for example `(/ 10 3)` returns the symbolic representation `10/3` without actually realizing any decimals). Secondly, `/` cannot create long overflow if both arguments are longs (excluding the "zero" special case).

- Upon reaching `Long/MAX_VALUE` increasing a number by one results in `LONG/MIN_VALUE`
- Upon reaching `Long/MIN_VALUE` decreasing a number by one results in `Long/MAX_VALUE`

The roll-over behavior is implemented by the set of functions ending with the "-int" suffix: unchecked-add-int. Clojure roll-

over functions mimic Java in case of overflow restarting from the lowest number at the opposite end of the scale. The next table shows what happens to the 64 available bits during a long type overflow (we need to remember the two's complement integer representation [100]):

64 bits (split 2 rows)	decimal value	relative value
0111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1101	9223372036854775805	(-Long/MAX_VALUE 2)
0111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1111-1110	9223372036854775806	(-Long/MAX_VALUE 1)
0111-1111-1111-1111-	9223372036854775807	Long/MAX_VALUE, (unchecked-

1111-1111-1111-1111-1111-1111-1111-1111		subtract Long/MIN_VALUE 1)
1000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000	-9223372036854775808	Long/MIN_VALUE, (unchecked-add Long/MAX_VALUE 1)
1000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0001	-9223372036854775807	(+ Long/MIN_VALUE 1)

1000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0000-0010	-9223372036854775806	(+ Long/MIN_VALUE 2)

As you can see from the table, approaching `Long/MAX_VALUE` fills up 63 bits with "1"s and the change of sign happens by resetting everything to zero except the first bit.

The `int` native type in Java has 32 bits and it's stored using 2's complement format [101]. The "unchecked-*-int" operators overflow into the opposite sign when reaching the `(Integer/MAX_VALUE)` and `(Integer/MIN_VALUE)` limits. The following table shows the bits layout upon reaching those limits and the effect of the related operation:

32 bits	decimal value	r
`01111111111111111111111111111101`	2147483645	(- I 2
`01111111111111111111111111111110`	2147483646	(- I 1
`01111111111111111111111111111111`	2147483647	I (s I 1
`10000000000000000000000000000000`	-2147483648	I (I 1
`10000000000000000000000000000001`	-2147483647	(- I

```
                                                                1

  10000000000000000000000000000010      -2147483646      (
                                                                I
                                                                2
```

[100] en.wikipedia.org/wiki/Two%27s_complement is the Wikipedia article describing how the two's complement binary representation works

[101] See en.wikipedia.org/wiki/Two%27s_complement for the details

5.2 ARBITRARY PRECISION

The basic math operations +, -, *, inc and dec are all examples of simple precision operators. When their short, int or long type operands go beyond the boundaries of `Long/MIN_VALUE` and `Long/MAX_VALUE`, these operators throw exception. Clojure also offers a different option: the arbitrary precision operators +' (note the single quote appended at the end of the name) automatically promote their return values to a `BigInt` type

which holds arbitrary size numbers (subject to memory availability).

5.3 ROLL-OVER PRECISION

Roll-over precision defines a set of functions in the Clojure standard library that do not result in an exception (or a type promotion) when the allocated storage space for that type is reached. The roll-over behavior for the long type refers to the fact that:

5.4 NON-CASTING, ROLL-OVER PRECISION

Another group of math operators is named after the pattern "unchecked-*-int" (replacing * with the name of the operation): `unchecked-add-int`, `unchecked-subtract-int`, `unchecked-multiply-int`, `unchecked-divide-int`, `unchecked-inc-int`, `unchecked-dec-int`, `unchecked-negate-int`, `unchecked-remainder-int`. These 8 functions are very similar by operating on the int type only and we are

going to describe them as a single group all under the `unchecked-add-int` function.

5.5 +, -, * AND /

`function` `since 1.0`

Listing 5.1. → *Math, Simple Arithmetic, Numeric Processing*

```
(+
  ([])
  ([x])
  ([x y])
  ([x y & more]))

(-
  ([x])
  ([x y])
  ([x y & more]))

(*
  ([])
  ([x])
  ([x y])
  ([x y & more]))

(/
  ([x])
```

```
    ([x y])
    ([x y & more]))
```

The basic arithmetic operations have a lot of features in common. The following description applies to `+`, `-`, `*`, `/` unless specified otherwise. One of the main aspects of main math operations in Clojure is that they take advantage of multiple "arities" to work in different contexts with great flexibility and performance.

Note

Before Clojure 1.2, basic math operators worked equivalently to the current auto-promoting version (the functions ending with a single quote). After Clojure 1.2, their behavior was converted to the current (by throwing instead of auto-promoting) to avoid the related performance penalty.

Contract

- `-` and `/` do not support the no-arguments arity.
- When invoked with no arguments `(+)` and `*` returns their identity value, 0 and 1 respectively.

- When invoked with one argument `(- x)` inverts the sign of the "x" argument.
- When invoked with one argument `(/ x)` returns the reciprocal of "x", commonly indicated as `(1/x)`.
- When invoked with a single argument both `+` and `*` just return the argument.
- All arguments must be of type `java.lang.Number` or subclasses of it (`(number? x)` must return `true` for all arguments). It will throw `ClassCastException` when an argument is not of type Number.

Return types change based on the input arguments. The following table summarizes the possibilities (excluding for the moment a few exceptional corner cases listed right after). Each box in the table shows the return type for each of the math operations considering the operand type at the x-y axis. If multiple return types are present for an operation (such as `(/)` `ratio long`) it means the return type also depends on aspects other than the types of the operands:

	short/int/long	float/double
short/int/long	(+)long (-)long (*)long (/)ratio long	(+)double (-)double (*)double (/)double
float/double	(+)double (-)double (*)double (/)double	(+)double (-)double (*)double (/)double
`BigInt`	(+)bigint (-)bigint (*)bigint	(+)double (-)double (*)double (/)double

```
(/)ratio
   bigint
```

`BigDecimal`

```
(+)bigdec
(-)bigdec
(*)bigdec
(/)bigdec[!]
```

```
(+)double
(-)double
(*)double
(/)double
```

Ratio

```
(+)ratio
(-)ratio
(*)ratio
(/)ratio
   bigint
```

```
(+)double
(-)double
(*)double
(/)double
```



Warning

Operands type marked with **[!]** can result in a `ArithmeticException` "Non-terminating decimal expansion". See with-precision.

Other special cases

Assuming we stay with long for the operands (using bigdec or double in any of the arguments causes the operation to return something different than another long) then there are a few exceptional cases that you need to be aware of:

- When any of `+` `-` `*` operating on long types results in number that is either `> Long/MAX_VALUE` or `< Long/MIN_VAUE` the operation throws `ArithmeticException integer overflow`.
- `/` by 0 results in a `ArithmeticException Divide by zero` exception.

Examples

0-arity and 1-arity

Let's have a look at the 0-arity and 1-arity first. `+` and `*` support invocations without arguments and this can be useful

to avoid superfluous checking for empty sequences. Here's for example the zero-arity in action with apply even on empty collections:

```
(def empty-coll [])          ; ❶

(apply + empty-coll)         ; ❷
0

(apply * empty-coll)         ; ❸
1
```

❶A simple `empty-coll` var simulates the result of some computation of which we don't know in advance the cardinality and resulting in an empty collection.

❷Since `+` is equipped with a zero-arity variant it works fine on empty sequences, without requiring an explicit check.

❸`*` works the same way, just returning 1 instead of 0.

In more general terms, `+` and `*` implement the identity element for addition and multiplication respectively [102].

The single operand version of `/` can be used to represent reciprocal of a number series. The values of the Riemann zeta function at 2 for example, is the sum of the reciprocal of the squares of natural numbers [103]. Other Riemann zeta functions

are important in statistics and physics. We can approximate the value of zeta at 2 by taking some number of elements from the series (it is possible to demonstrate the series converges to `PI^2/6`, the Basel problem solved by Euler in 1734):

```
(defn x-power-of-y [x y] (reduce * (repeat y x)))
(def square #(x-power-of-y % 2)) ; ❷
(def cube #(x-power-of-y % 3))

(defn reciprocal-of [f] ; ❸
  (->> (range)
    (map f)
    rest
    (map /)))

(defn riemann-zeta [f n] ; ❹
  (->> f
    reciprocal-of
    (take n)
    (reduce +)
    float))

(riemann-zeta square 1000) ; ❺
;; 1.6439346

(/ (* Math/PI Math/PI) 6) ; ❻
;; 1.6449340668482264
```

```
(riemann-zeta cube 100) ; ❼
;; 1.2020074
```

❶ `x-power-of-y` is a generic helper to elevate x to the power of y.

❷ `square` is a specific instance of `x-power-of-y` to square a number. Similarly for `cube`.

❸ `reciprocal-of` contains `(map /)` mapping of `/` divide operator that creates a reciprocal. We pass in `f` which is the way to decide what we want the reciprocal of (for example squares or cubes).

❹ `riemann-zeta` takes `f` as the kind of Riemann we want to calculate. Riemann zeta at 2 means for the reciprocal of the squares, at 3 for the cubes and so on.

❺ We invoke `riemann-zeta` for squares and we indicate we want to stop at the 1000th element of the series. At 10000 it's already above one minute on a normal laptop, so be careful.

❻ The actual number the series is supposed to converge to is quite close at 1000 of precision.

❼ Riemann zeta at 3 also converges and the number is called Apéry's constant.

2-arity

The next example shows what is probably the most used number of arguments with basic math operators: two operands. The annual interest rate formula, for example, is a way to determine how much an initial capital will increase over time. We are going to see how easy the mathematical formula can be translated into Clojure by using partial application:

$ca\overline{1}$, $ca\overline{2}$, $ca\overline{3}$, $ca\overline{4}$, ...

where:

- **c** = initial investment
- **r** = interest rate
- **a** = 1 + r

Each item in the series represents the total amount each year. So if we assume an initial investment of c = $1000 and we want to know how much will be in the bank after 3 years with a 20% interest rate, we'll have to look at the 3rd element in the list: `1000 * (1 + 0.2)^3 = 1728`. We can generalize the formula using Clojure, creating an infinite sequence from which we take as many yearly forecasts as we want:

```
(defn powers-of [n]                          ; ❶
  (iterate (partial * n) 1))
```

```
(defn interest-at [rate initial year]     ; ❷
  (->> (powers-of (inc rate))
    (map (partial * initial))
    (take year)
    last))

(interest-at 0.2 1000 4)
;; 1728.0
```

❶ `powers-of` creates an infinite sequence of powers of the number `n`. We use partial along with `*` to let iterate pass down the result of the previous multiplication.

❷ `interest-at` groups together the rest of the formula. Again the use of partial prepares for one element from the previous series of powers to be multiplied for the initial investment.

Precision

The common (and default) math operators can throw exception (in this respect Clojure departs from Java). Clojure numbers literals are handled as long by default, corresponding to the `java.lang.Long` Java class. So for example, the very fast growing series $n = (n - 1)^{\overline{n-1}}$ will throw `ArithmeticException` pretty soon:

```
(take 7 (iterate #(* % %) 2))
;; ArithmeticException integer overflow

(take 7 (iterate #(*' % %) 2))
;; (2 4 16 256 65536 4294967296 18446744073709551616)
```

❶The 7th element of the series is: $4294967296^{4294967296}$ which exceeds `Long/MAX_VALUE`. Since Clojure basic math operators won't auto-promote, it will throw Exception on crossing that limit.

❷In order to accept results bigger than those we can use *'. We can see now that the last element in the sequence contains the "N" suffix indicating that it is a BigInt type.

Note

Please note that despite the fact that Clojure treats numbers as Long by default, the error message is always referring to an "Integer" overflow. It should be read more generally as "Natural" numbers overflow, being those Integers or Longs.

In Java the `+` operator will happily execute an overflowing operation and return a negative number! This is why it's

common idiom to check for over/underflow in Java or use the BigInteger class [104]. Clojure took the more conservative approach that an operation should never result in some implicit truncation or sign change. Developers can still access that behavior if they need by using "unchecked" version of the same operators.

Things you'll never forget: your first (+ 1 1)

This isn't absolutely a rule, but addition is often the first function executed by people starting with Clojure. One typical first Clojure lesson is to open up a REPL and play some Clojure. One of the first things new learners have to learn is the use of parenthesis to execute functions (unless of course they are coming from some other Lisp language). By executing a simple (+ 1 1) newbies get a good idea about invoking functions with parenthesis and prefix operators. That's why `+` is likely the first function ever executed by someone learning Clojure.

See also

- A single quote `'` suffix appended to the operation symbol defines the auto-promoting version of it. Instead of throwing exception upon reaching the limit of Java longs, it will promote the long to a `BigInt` instance (that can take care of arbitrary precision). Use for example *' when it is important for your application to maintain precision. Beware that precision comes at a cost.

- unchecked-add are versions of math operators (including the basic ones described in this chapter) removing the over/under flow checks. This is the standard Java behavior. Use the unchecked version if you are willing to trade performance for the possibility to have a sign change when overflowing. If your application will never see big numbers and you need a performance boost you can use unchecked versions with confidence.
- unchecked-add-int are even faster. All other operators will promote int operands to longs and return longs. Use the unchecked-int version when working primarily with integers to avoid unnecessary casts to long. Unless you're doing fast integer math, it is unlikely you'll ever need unchecked integer operations.

Performance considerations and implementation details

=> `O(n)` **linear in the number of arguments**

As seen in the related functions, basic math operators are quite sophisticated in checking for overflow/underflow conditions and treating all numbers as longs. Your primary performance concern would then be to remove those checks and casts. But

even considering these performance aspects, replacing `+` with unchecked-add or unchecked-add-int for example, should only be done when dealing with specific numerical aspects of an application in search for raw speed. Another aspect to consider is the amount of integers to sum. `+` will use reduce for arities above 2 and reduce is linear in the number of arguments. In summary: + shouldn't be primary concern in search for speed while developing general purpose applications unless there are parts of the application specifically dealing with large amount of integers.

[102] This simple Wikipedia article also illustrates identity elements for other operations: en.wikipedia.org/wiki/Identity_element

[103] To know more about the Riemann zeta function see the introductory article at Wikipedia en.wikipedia.org/wiki/Riemann_zeta_function

[104] with the introduction of Java 8 there is now a new set of arithmetic operations to throw exception in case of overflow/underflow exactly like Clojure. See for example: docs.oracle.com/javase/8/docs/api/java/lang/Math.html#addExact-long-long-

5.6 INC AND DEC

`function` `since 1.2`

Listing 5.2. → *Counting, Incrementing and Decrementing*

```
(inc [x])

(dec [x])
```

`inc` and `dec` functions are basically shortcuts for `#(+ % 1)` and `#(- % 1)` respectively. Incrementing and decrementing is a common operation in everyday programming, so `inc` and `dec` are responsible for saving a good amount of keystrokes in a typical Clojure development life. Using `inc` or `dec` is as easy as:

```
(inc 1)
;; 2

(dec 1)
;; 0
```

Contract

- "x" is the single mandatory argument of numerical type (i.e. `(number? x)` must be true)

- returns: the value obtained by incrementing or decrementing x by 1. It throws `ArithmeticException` on `(Long/MAX_VALUE)` or `(Long/MIN_VALUE)` overflow.

Examples

Maps, atoms or any other data structure offering an "update" function are good candidates to store a counter and can be used in conjunction with `inc` or `dec`. The following example shows an `instrument` function that takes another function as argument and "injects" it with a counter to store the number of calls it receives. The number can be read later using a special keyword:

```
(defn instrument [f]
  (let [calls (atom 0)]                    ; ❶
    (fn [& args]
      (if (= "s3cr3tC0d3" (first args))    ; ❷
        @calls
        (do (swap! calls inc)              ; ❸
            (apply f args))))))

(def say-hello                             ; ❹
  (instrument #(println "hello" %)))
```

```
(say-hello "john")
;; hello john
;; nil

(say-hello "laura")
;; hello laura
;; nil

(say-hello "s3cr3tC0d3")                    ; ❺
;; 2
```

❶ an atom instance initialized to zero is created every time `instrument` is invoked. Compare and swap semantic (CAS) will prevent missing (or double) counts effectively even in case this function is called in highly concurrent environments.

❷ we intercept the arguments and when the first one is a special "secret" code the function returns the count so far instead of delegating to the wrapped function.

❸ in all other cases, we increment the counter. With atoms it's as easy as passing the incrementing function for updating to swap!. What we need is a function of one argument that increments it: the perfect spot for `inc`.

❹ `say-hello` shows how `instrument` can be used to wrap another function.

❺ after using `say-hello` a few times we can see what happens when we use the secret code that displays the number of times the inner println was called.

Incrementing and Decrementing in other languages

Probably every language ever invented (beyond the purely research ones) has an addition operator "+" or equivalent function. But not all of them have a special increment/decrement operator. The reason why it's special is because it implicitly considers "1" to be the second operand to add/subtract saving some keystrokes. Here are three representative examples from Haskell, Ruby and Java:

- Haskell

Haskell has a "pred" and a "succ" functions that work similarly to Clojure:

```
> succ 1
2

> pred 0
-1
```

Haskell can also define curried functions in a very compact form, so despite the fact that we are using the common `+` and `-` operators we can express incrementing and decrementing easily like:

```
> (+1) 1
2

> ((-) 1) 1
0
```

Although for subtraction it doesn't work as good because of the ambiguity generated by `-1` as a negative number literal.

- Ruby

Ruby main inspiration in this case is object orientation. Numbers are objects and can receive "messages". We can send the message `succ` or `pred` to a number like this:

```
irb(main):001:0> 1.succ
=> 2
irb(main):002:0> 1.succ.pred
=> 1
```

- Java

Differently from Ruby, Java number literals cannot receive methods calls directly. Although numbers can be wrapped in a new Integer() object first, there is no method to get the next of a number. The only way is through mutation. Java derives increment and decrement operators from C. There is a big difference between Java's `++` increment operator and Haskell's `succ` for example: Java `++` will also mutate a variable while making it bigger:

```
class Test
{
  public static void main (String[] args)
  {
    int i = 0;
    System.out.println("incrementing " + ++i);
```

```
    System.out.println("and i is? " + i);
  }
}

>> incrementing 1
>> and i is? 1
```

See also

- inc' and dec': similarly to +' and -' the single quote `'` identifies the auto-promoting version of `inc` and `dec` respectively. If the number is `Long/MAX_VALUE` or `Long/MIN_VALUE` it will throw exception in the attempt to `inc` or `dec`. The single quote version will avoid the problem promoting the long to a BigInt.
- unchecked-inc and unchecked-dec: this version of the operators is not auto-promoting nor throwing exception. Upon reaching the upper/lower limit the result will simply invert the sign and start from the other side:

```
(unchecked-inc Long/MAX_VALUE)
;; -9223372036854775808
```

- unchecked-inc-int and unchecked-dec-int: like other unchecked int function versions, it won't attempt an implicit cast to Long for all numbers. It means that both arguments and result type will be treated as java.lang.Integer. The upper limit before incrementing or decrementing will change sign to the operand is thus Integer/MAX_VALUE, Integer/MIN_VALUE.

Performance considerations and implementation details

=> `O(1)` **constant time**

It's unlikely for incrementing or decrementing operators to be the source of a performance bottleneck and unless your core business is specifically integer math then you should not worry. For those cases where it is important, Clojure offers alternatives like the ones described in the "See Also" section above. It's important to understand that although there are faster ways to `inc` or `dec`, any change in search for raw performance should be accurately profiled. Having said that, the following example shows an small improvement from plain `inc` to unchecked-int:

```
(require '[criterium.core :refer [bench]])

(defn slower [n]
  (loop [n (int n) i 0]
```

```
      (if (< i n)
        (recur n (inc i)) ; ❶
        i)))

  (defn faster [n]
    (loop [n (int n) i 0] ; ❷
      (if (< i n)
        (recur n (unchecked-inc i)) ; ❸
        i)))

  (bench (slower 10000))
  ;; Execution time mean : 13.381100 µs

  (bench (faster 10000))
  ;; Execution time mean : 2.595440 µs
```

❶The function `slower` uses normal `inc` to increment a number.

❷Note the trick we used in both functions when initializing the loop. We cast "n" to be an int type to prevent "<" comparison to require a conversion from object to number. If we don't optimize this aspect, the object conversion dominates `inc` execution.

❸ `faster` differes from `slower` only because it uses unchecked-inc instead of simple `inc` .

Note

The example shows a visible improvement by removing the check for integer overflow. But the example also shows that there could be other operations dominating the performance profile other than the one we are trying to optimize. In our case, the comparison `(< i n)` (where `n` is not cast to int) dominates over unchecked-inc for the worse. Always use a profiler to verify the assumptions about a target hotspot before engaging in changes that don't produce a noticeable effect.

5.7 QUOT, REM AND MOD

`function` `since 1.0`

Listing 5.3. → *Math Operations, Integer Division, Modulus*

```
(quot [num div])
(rem [num div])
(mod [num div])
```

In the Euclidean division (the process of division of two integers [105]) the "quotient" is the result of the division while the "remainder" is anything that is left when the two numbers are not multiple of each other. In Clojure `quot` returns the result of dividing the numbers while `rem` returns the rest of the integer division, if any. The quotient can also be defined as the number of times the divisor divides "num", excluding any fractional part which is equivalent in turn to take the result of the division and truncate the decimals. Finally, the "modulus" operation `mod` is very similar to `rem` but with slightly different rules regarding how to return results in the presence of negative numbers [106]. So for example:

```
(quot 38 4)
;; 9

(rem 38 4)
;; 2
```

The above means that the number "4" (the divisor) sums up 9 times before being larger than the dividend "38". The remainder of the operation would be 2. This description is not very rigorous because things get hairy when negatives numbers are involved. This is where `mod` and `rem` differs:

```
(rem -38 4)
;; -2

(mod -38 4)
;; 2
```

We won't go into the glory details of `rem` and `mod` for negative numbers since the most common uses for real-world applications are around positive quantities.

Contract

input

- "num" and "div" must be numbers (which means that `(and (number? num) (number? div))` must be true). So although `quot` deals conceptually with integer quantities it can return doubles representing them:

```
(quot 38. 4)
;; 9.0
```

notable exceptions

- `AritmeticException` when "div" is 0.

output

Returns the quotient or the remainder of dividing "num" by "div". If any of "num" or "div" is a double, then the result will also be a double.

Examples

`quot` and `rem` are often seen in problems where some items should be partitioned in containers. If we don’t have the possibility to physically cut the item into a half or other fraction, `quot` is handy to discover how many items we can distribute evenly in the containers. Let’s say we need to load a truck with some goods and the truck only accepts 22 containers. Given 900 items to transport, we want to know how many items we should put in each container:

```
(defn optimal-size [n m]
  (let [size (quot n m)
        left? (zero? (rem n m))]
    (if left?
      size
      (inc size))))

(optimal-size 900 22)
;; 41
```

```
(partition-all (optimal-size 900 22) (range 900)
;; ((0 1 2 3 ... 38 39 40) (41 42 43 ...
;; 79 80 81) ...  (82 83 84 ...
```

❶ `optimal-size` will return the best size for a container given n items and m containers. It makes good use of `quot` to find out how many items can stay in all containers

❷ `rem` is used then to to see how many items would be left out. If there are items left out, the optimal size is increased by one.

❸When we call `optimal-size` with the example parameters we can see that the optimal amount of items each container is 41

❹We can also see how the items should be distributed in each container using partition-all. The output is quite big, so it has been omitted from the example.

Although the example above is interchangeable between `rem` and `mod`, there is one case that commonly correlates with `mod` more than `rem` (although even in this case, being all the quantities positive, it still doesn't make a difference): the definition of basic operators on top of a finite set. Let's take the alphabet for example: we want to implement an increment operator so that it returns the next letter. After creating the alphabet as an array we can make good use of indexes.

```
(def alpha
  ["a" "b" "c" "d"
   "e" "f" "g" "h"
   "i" "j" "k" "l"
   "m" "n" "o" "p"
   "q" "r" "s" "t"
   "u" "v" "w" "x"
   "y" "z"])

(defn ++ [c]      ; ❶
  (-> (.indexOf alpha c)
      inc
      (mod 26)
      alpha))

(++ "a")
;; "b"

(++ "z")
;; "a"
```

❶several interesting things are going on in this single liner: first we extract the position of the letter in the array using Java interop `indexOf` that is supported by the vector type. After incrementing it we need to make sure we are not at the boundaries and if we are we want to overflow gracefully and restart from the beginning. The key enabler is `(mod x 26)` that will recompute any number x to be in the domain 0-25 including any potential roll-over of the rest from the beginning. Once we have the new index we just access the array again. Notice that `alpha` is used as a function, passing the index to retrieve the element at that index.

So, `rem` or `mod`? Some of the confusion stems from the fact that in "C" languages there is a `%` operator that is commonly referred as "mod" but is instead implementing the remainder operation. So some classic example of use of remainder are referred as using `mod` in those languages.

Why not calling Java quotient and remainder operators directly?

One of the Clojure main selling point is Java interoperability. Although it works very well, there isn't a way to call Java operators like `+`, `-`, `/`, `%` directly. Not even in Java, where the reflection mechanism stops at anything that is invocable from an object instance or class. So Clojure offers a wrapper for each one of them, sometimes in short form (like single character symbols) sometimes in a slightly longer form (like `quot` or `rem`) presented in this chapter. One positive fact about the longer form offered by Clojure is that it's easier to remember what the operator is. Consider for

example the Clojure unsigned-bit-shift-right compared to the Java `>>>`: it might be more verbose but at least is much easier to remember what it does!

See also

- / is the common division operator. It will return fractions instead of removing decimals as `quot` does. Use `quot` only when you need to deal exclusively with integer quantities.
- unchecked-remainder-int is available to improve performance when quantities are between Integer/MIN_VALUE and Integer/MAX_VALUE. This version of the remainder operation won't auto-promote or cast to long. See the performance considerations section for additional details.

Performance considerations and implementation details

=> `O(1)` function generation

For all the practical uses `quot`, `rem` and `mod` are constant time operations. Modern hardware knows how to optimize what is practically division so it shouldn't be the source of surprises during performance optimization. Some difference exists on tight loops when BigInt promotion is necessary.

For some class of applications (hashing comes to mind, where graceful overflowing is considered a feature), Clojure also provides unchecked-reminder-int, a `rem` version that does not check for integer overflow. The following benchmark compares the two:

```
(require '[criterium.core :refer [quick-bench]])

(let [num 100 div 10] (quick-bench (rem num div)
;; Execution time mean : 15.733921 ns

(let [num (int 100) div (int 10)] (quick-bench (
;; Execution time mean : 8.169385 ns
```

❶The presence of let communicates the types of of "num" and "div" to the compiler, so when the expression is generated as Java bytecode "num" and "div" appears as primitive long types. This is important to avoid conversions between the uppercase Long class type and the primitive type (also known as "boxing/unboxing").

❷We can definitely observe a speed up.

We can observe a 50% improvement which is substantial but small in absolute terms (timings are nanoseconds). Only use

unchecked-remainder-int when you are able to prove there are no other factors dominating the execution of the expression.

[105] Quotient and remainder are well explained in the following Wikipedia article: en.wikipedia.org/wiki/Euclidean_division

[106] The Wikipedia article about the Reminder operation also clarifies how each language distinguish between modulo and reminder: en.wikipedia.org/wiki/Remainder

5.8 MAX AND MIN

`function` `since 1.0`

Listing 5.4. → *Math, Statistics, Numbers*

```
(max
  ([x])
  ([x y])
  ([x y & more]))

(min
  ([x])
  ([x y])
  ([x y & more]))
```

`max` and `min` are two useful math-related functions in the standard library. Given a list of numerical arguments they return the biggest or the smallest number in the list, respectively. So:

```
(max 5 7 3 7) ;; 7
(min -18 4 -12) ;; -18
```

`max` and `min` functions imply some notion of ordering which is almost always guaranteed with numbers. Notable cases of tricky numbers to order are "Infinity" and "NaN". While for infinity we have negative and positive by convention (so it's always possible to determine which is bigger), `max` or `min` operations involving NaN (Not A Number [107]) will always return NaN.

Contract

input

- "x", "y" and "more" are numbers (such that `(number? x)` is `true`).

known exceptions

- `ClassCastException` if any argument is not a number.

output

`max` and `min` returns the biggest or smallest of their respective arguments. If negative infinity is one of the argument, `min` will always return negative infinity. If positive infinity is an argument, `max` will always return positive infinity. When one of the argument is NaN they both return NaN:

```
(max Long/MAX_VALUE 5 (/ 1.0 0)) ; ❶
;; Infinity

(min 1 Long/MIN_VALUE (/ -1.0 0))
;; -Infinity

(max (/ 1.0 0) (/ 0.0 0))
;; NaN
```

❶The most important corner cases of `max` and `min` when dealing with special numbers.

Examples

`max` and `min` are obviously related to statistics. One first application could be to find the extremes of a given collection, which can be solved with an easy one liner:

```
(apply (juxt min max) (range 20)) ; ❶
;; [0 19]
```

❶Since `max` and `min` don't take a collection directly, we use apply to spread the collection. To combine `max` and `min` in a single call we used juxt.

To expand the example in the direction of searching for local maxima, we could implement a "best-lap" function that can be used to show what is the current best lap in a competition:

```
(defn tracker []
  (let [times (atom [])]
    (fn [t]                    ; ❶
      (swap! times conj t)     ; ❷
      (apply min @times))))    ; ❸

(def timer (tracker))
(timer 37.21)
;; 37.21
(timer 38.34)
;; 37.21
```

```
(timer 36.44)
;; 36.44
(timer 37.21)
;; 36.44
```

❶The `tracker` function initializes a new state atom each time it is invoked. It then "closes over" the initialized state returning a new function accepting a single "time" parameter. The higher order function is returned to the caller ready to be used.

❷Every time we receive a new time measure we add it to the collection of time events stored in the atom

❸The best of the timings collected up to now is returned. We use `min` and apply to find out the smallest number in the collection.

What exactly is the meaning of NaN?

NaN in computing is a convention to store undefined or unrepresentable values as bits. It was introduced by the IEEE 754 floating-point standard in 1985. NaN, which is literally "Not A Number", can be the result of several mathematical "singularities":

```
(def ∞ (/ 1. 0))

(/ 0. 0)
(/ ∞ ∞)
(* 0 ∞)
(- ∞ ∞)
```

```
(Math/pow 1 ∞)
(Math/sqrt -1)

;; all producing:
;; NaN
```

Clojure mainly follow Java rules for NaN and is conveniently available as static attribute of `java.lang.Double`. Despite the name claiming it's not, NaN is a number and can be used in arithmetic operations:

```
(number? Double/NaN) ; ❶
;; true

(inc Double/NaN) ; ❷
;; NaN

(instance? Double Double/NaN) ; ❸
;; true
```

❶"Not a Number" is a number!

❷As such, we can use NaN in arithmetic operations or number comparison.

❸More specifically, NaN is a double.

The surprising behavior of NaN does not stop here. One of the interesting properties of NaN is that is the only number that is not equal to itself:

```
(== Double/NaN Double/NaN) ; ❶
;; false
```

❶NaN is not equal to itself.

See also

- max-key works very similar to `min` and `max` except that they can accept any type of argument with the constraint that there must be a way to turn them into numerical. A function that is passed as the first argument will be used to determine how the arguments should be turned into numbers.

Performance considerations and implementation details

=> `O(n)` **linear time**

=> `O(1)` **space allocation**

The algorithm to find the `min` or `max` needs necessarily to compare the current candidate against all the other numbers in the collection, even when the collection is ordered (like a vector or a sorted-set). The linearity can be verified without any particular performance tooling (so don't pay too much attention to the actual numbers, just the order of magnitude):

```
(let [s (apply sorted-set (range 100000))] (time
;; "Elapsed time: 25.701355 msecs"

(let [s (apply sorted-set (range 10000000))] (ti
;; "Elapsed time: 1292.216983 msecs"
```

So if speed is a constraint and the algorithm can be designed to accumulate the collection of results over time, the ideal solution would be to push elements in a sorted-set as they are ready and take first or last instead of calling `min` or `max`.

[107] NaN (or Not a Number) is a special kind of number that is not representable in modern computers architectures. See en.wikipedia.org/wiki/NaN

5.9 MAX-KEY AND MIN-KEY

`function` `since 1.0`

Listing 5.5. → *Math, Statistics, Numbers*

```
(max-key
  ([k x])
  ([k x y])
  ([k x y & more]))
```

```
  (min-key
    ([k x])
    ([k x y])
    ([k x y & more]))
```

`max-key` and `min-key` build conceptually on top of max and min functions. While max would just operate on numbers, `max-key` and `min-key` can also accepts other types and an additional function (the key) to help transform or extract those arguments into numbers and finally determine the biggest or the smallest, respectively. Instead of just returning the min or the max number, `max-key` and `min-key` return a value of the same type as the argument:

```
  (min-key last [:a 1000] [:b 500])
  ;; [:b 500]

  (max-key :age {:name "anna" :age 31} {:name "jac
  ;; {:name "anna", :age 31}
```

Contract

input

- "k" is a function of a single argument returning a numerical type. "k" is invoked over the arguments: `(k x)`, `(k y)` and so on.
- "x", "y" (and any additional arguments) can be of any type.

notable exception

- `ClassCastException` when "k" return a result which is not a number.

output

`max-key` and `min-key` return the argument that after calling "k" returns the biggest or the smallest result respectively.

Warning

Differently from max, `min-key` and `max-key` do not handle NaN (while it handles positive and negative infinity just fine). So you need to pay attention when using `min-key` and `max-key` and number processing that can potentially generate NaN in any of the arguments.

The following example shows an erroneous result produced by the presence of NaN. This function to calculate the speed of sound contains a slightly wrong formula:

```
(def air-temp [[:cellar 4]
               [:loft 25]
               [:kitchen 16]
               [:shed -4]
               [:porch 0]])

(defn speed-of-sound [temp]
  (* 331.3 (Math/sqrt temp))) ;; ouch

(apply max-key #(speed-of-sound (last %)) air-te
;; [:porch 0]
```

❶ Speed of sound in air roughly increases with the square root of the temperature. The problem in this formula is that we should be talking about the absolute temperature instead: `(inc (/ temp 273.15))`. But we forgot to do the conversion.

❷ When searching for the part of the house where sound moves fastest we get a wrong result without exceptions being raised. The problem is in the square root of a negative number generating NaN forcing max-key to return always the last argument of the collection, despite others being bigger.

The correct formulation that prevents the problem above is for example the following:

```
(defn speed-of-sound [temp]
  (* 331.3 (Math/sqrt (inc (/ temp 273.15)))))

(apply max-key #(speed-of-sound (last %)) air-te
;; [:loft 25]
```

❶ We change the formula so it transforms the temperature in absolute Kelvin (which is never negative, assuming the input is correct).

❷ We can see that sound moves quickly in the loft, where the temperature is higher.

Examples

When talking about `min` we used a `tracker` function that, when invoked sequentially with measures of times, was always responding with the current minimum. We want now to extend the example so it can also record additional information such as the athlete name, so we can use it to show who the winner of a competition, not just the best time. `min-key` solve the problem quickly:

```
(defn update-stats [event stats]
  (let [events (conj (:events stats) event)
        newmin (apply min-key :time events)
        newmax (apply max-key :time events)]
    (assoc stats :events events :min newmin :max

(defn new-competition []
  (let [stats (atom {:min {} :max {} :events []}
    (fn
      ([] (str "The winner is: " (:min @stats)))
      ([t] (swap! stats (partial update-stats t)

(def race1 (new-competition))

(:min (race1 {:athlete "Souza J." :time 38.34}))
;; {:athlete "Souza J.", :time 38.34}
```

```
(:min (race1 {:athlete "Kinley F." :time 37.21})
;; {:athlete "Kinley F.", :time 37.21}

(:max (race1 {:athlete "Won T." :time 36.44}))
;; {:athlete "Souza J.", :time 38.34}

(race1)
;; "The winner is: {:athlete \"Won T.\", :time 3
```

❶ `update-stats` is a function that takes a new event and the current stats and calculates the new stats.

❷Using both `min-key` and `max-key` we can calculate the best and the worst time keeping all the other related information.

❸ `new-competition` wraps the setup of the initial state. The initial state is stored inside an atom and consists of a few nested data structures identified by relevant keys. The local state is part of the bindings of the function that is returned to the caller.

❹The no-arguments arity is called at the end of the competition to show the winner.

❺The second arity takes the new event as input and proceeds to update the state with swap!, passing the `update-state` function the old state. The new event is also passed down to `update-state` through partial.

❻A new competition is created by invoking the main `new-competition` function without arguments.

❼The examples are showing that after sending new events to the competition, the returned value can be queried for statistics, like best time or worst time.

❽The `race1` function invoked without arguments prints the final result.

Another way to look at problems involving minimum and maximum is when trying to minimize or maximize a function,

for example the nearest neighbor search. Donald Knuth informally called this class of searches the "post-office problem" [108].

Let's take the challenge literally and try to solve the following: given a list of geographical coordinates of postal offices, we want to know which post-office to assign to a new house in the area. We can solve the problem searching the value that minimizes the euclidean distance (approximated for a spherical surface). This is more rigorously expressed by the Haversine formula [109]:

```
(defn sq [x] (* x x))
(defn rad [x] (Math/toRadians x))
(defn cos [x] (Math/cos (rad x)))
(defn sq-diff [x y] (sq (Math/sin (/ (rad (- x y

(defn haversine [[lon1 lat1] [lon2 lat2]]
  (let [earth-radius-km 6372.8
        dlat (sq-diff lat2 lat1)
        dlon (sq-diff lon2 lon1)
        a (+ dlat (* dlon (cos lat1) (cos lat1))
    (* earth-radius-km 2 (Math/asin (Math/sqrt a

(defn closest [geos geo]
  (->> geos
       (map (juxt (partial haversine geo) identi
```

```
        (apply min-key first)))

(def post-offices
  [[51.74836 -0.32237]
   [51.75958 -0.22920]
   [51.72064 -0.33353]
   [51.77781 -0.37057]
   [51.77133 -0.29398]
   [51.81622 -0.35177]
   [51.83104 -0.19737]
   [51.79669 -0.18569]
   [51.80334 -0.20863]
   [51.74472 -0.19791]])

(def residence [51.75049331 -0.34248299])

(closest post-offices residence)
;; [2.2496423923820656 [51.74836 -0.32237]]
```

❶ `sq`, `rad`, `cos` and `sq-diff` are all helper functions necessary for conversions and other geometry related transformations

❷ `haversine` calculates the spherical distance which would be approximated on Earth since the radius is not always the same. It is definitely a good enough approximation for this exercise.

❸ `closest` is the function that determines which of the positions of the postal offices minimizes the distance with the target residence coordinates. We start by taking the vector of the positions of postal offices, decorate it with the distance to the target residence with a map operation and the using `min-key` to see which one is the closest.

❹ Invoking the `closest` function returns a pair containing the shorted distance in Kilometers.

See also

- max and min have a similar goal but they only accept numerical arguments, without the option to pass a "key" function to decide how to process arguments of other types. Prefer max when you are only interested in retrieving the actual min or max from a collection of numbers, without any other related data.

- reduce is the generalization of the process adopted by `min-key` and `max-key`. reduce can iterate over a sequence while storing some information for the next step to execute. This is exactly what happens when electing a local minimum or maximum for the current comparison and then repeating the process with the next. Consider reduce if you need different kind of aggregation statistics, such as the median.

Performance considerations and implementation details

=> `O(n)` **linear time**

=> `O(1)` **space allocation**

Performance considerations are similar to the max or min functions, plus the overhead generated by the "key" function. In summary: if the number of arguments is big but it is possible to accumulate them gradually in an ordered collection (for example in a sorted-set) then try to avoid `min-key` or `max-key` by just taking the first or last from the ordered results.

[108] A nice introduction to nearest neighbor problems is available on wikipedia: en.wikipedia.org/wiki/Nearest_neighbor_search

[109] For examples and explanation of the Haversine formula see the following Wikipedia article: en.wikipedia.org/wiki/Haversine_formula

5.10 RAND AND RAND-INT

`function` `since 1.0`

Listing 5.6. → *Random Numbers Generation*

```
(rand
  ([])
  ([n]))

(rand-int
  [n])
```

Generation of random numbers is a common feature in many languages. In Clojure they generate random numbers of type double or int respectively using the Java pseudo-random generation capabilities. For many practical applications, this kind of random numbers are good enough. They are not considered good for stronger randomness requirements like

Monte Carlo Generation [110]. By default `rand` produces a double number in the range `0 ⇐ x < 1` while `rand-int` generates a random integer from 0 to the specified bound:

```
(rand)
;; 0.6591252808399537

(rand-int -10)
;; -5
```

Contract

input

- `rand` with no arguments: returns a pseudo random double number between 0.0 inclusive and 1.0 exclusive. `rand-int` requires at least one argument.
- "n": is the bound for both functions and it must be of type `double` for `rand` and of type `int` for `rand-int` (negatives included).

output

The output is an integer for `rand-int` and a double for `rand`.

Examples

Randomness finds applications in many fields. A simple use case is in A/B testing, when we want some percentage of the users to pick the new feature. Let's say we want to increase the number of people answering a survey and we think that showing how many questions still remain is a good idea. At the same time we don't want to risk to make a potential bad decision on the entire population of customers taking the survey, so we decide to roll-out only to the 50% of the total requests. We can achieve this by using `rand`:

```
(def questions
  [["What is your current Clojure flavor?" ["Clo
   ["What language were you using before?" ["Jav
   ["What is your company size?" ["1-10" "11-100

(defn format-options [options]
  (apply str (map-indexed #(format "[%s] %s " (i

(defn render [[question options] prefix]
  (str "Q" prefix question " " (format-options o

(defn A-B [prob A B]
  (if (< prob (rand)) A B))

(defn progress-handler [total]
  (A-B 0.5
    (fn [progress] (format "(%s/%s): " total pro
```

```
    (constantly ": ")))

(defn ask [questions]
  (let [progress (progress-handler (count questi
    (loop [[q & others] questions a []]
      (if q
        (do
          (println (render q (progress (inc (cou
          (recur others (conj a (read))))
    a))))

(ask questions)
;; Q(3/1): What is your current Clojure flavor?
;; 2
;; Q(3/2): What language were you using before?
;; 1
;; Q(3/3): What is your company size? [1] 1-10 [
;; 3
;; [2 1 3]
```

❶ `A-B` is a function for branching very similar to if but it also accept a number between 0 and 1 to determine with which probability one branch of the condition will be used compared to the other. It can be implemented very simply by just comparing the input probability with the result of invoking `rand` which effectively converts into a probability to pick option A or option B in the if statement.

❷ `progress-handler`, of all the features offered by our survey manager, is the feature impacted by the A/B testing. So it just uses `A-B` to put in effect a 90% probability that the new feature "A" will be presented to the user compared to the old feature "B". We use 50% (here translated to 0.5) so the user will be presented with a "progress indicator" instead of just a colon ":" symbol 50 times over 100.

❸ when it's time to `ask`, we start the survey by looping over the questions. Since we want that all questions are either showing the progress or not showing it, the A-B conditional branching needs to happen ahead of the loop.

❹ since we are missing the current index of the question being asked, we need to wait until that data is available inside the loop. This is why the `progress-handler` returns a function of the progress so far that needs to be passed as an argument at the time of the actual invocation. During this println we

have the information available, so we invoke the higher order function that was created in the `progress-handler`

Randomness is not an easy task!

In the context of programming languages we usually talk about "pseudo-random" number generation instead of "exact". The reason is that being effectively random is not an easy task and often one that requires a trade-off between speed and easy of use. There are many algorithms related to random numbers generation and research in the field is constantly in progress. The one that Java (hence Clojure) uses is a linear congruential generator (LCG), which essentially uses some initial seed (often connected with the hardware epoch time) and then a specific formula of multiplication and modulo to generate a finite set of different numbers until the sequence repeats (which is called **period** of the generator).

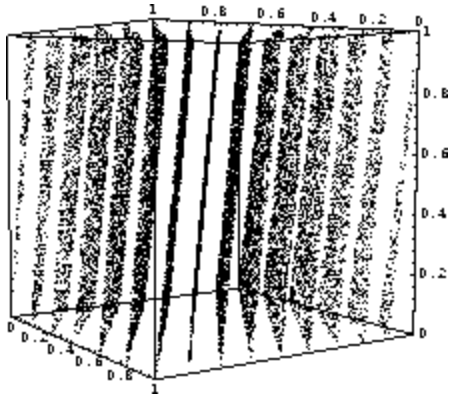
As said at the beginning, this class of random generators is good enough for most applications, but they fall short of many scientific simulations where strong randomness is important. The period of Java's LCG (2^{48}) is not the only problem, because LCGs also suffer from some level of:

1. "Falling the planes"
2. The randomness does not distribute equally across the bits.

The first problem has to do with the distribution of the random numbers and can be visualized if we plot random points in a 3D space, thus the name (this paper www.pnas.org/content/61/1/25.full.pdf from 1968 is probably the first to call the problem this way). One LCG algorithm that suffers more than others about the problem is the RANDU [111] and although it has been replaced with much better ones, it

helps to understand the problem. As you can see in the picture below, the plotted points are not distributed evenly as one would expect, but forms a predictable pattern in the form of 2D planes.

Figure 5.1. RANDU generated 3D points: the pattern is visible in the form of 2D planes.



The second problem has to do with the way the numbers are generated, which essentially is a repeating bit-wise mutation every new number. When we look at the number generated as bits, we can see a pattern emerging. This time we can verify our assumption directly on `java.util.Random` and visualize the problem with the following example:

```
(import 'java.awt.image.BufferedImage
        'java.util.Random
        'javax.imageio.ImageIO
        'java.io.File)
```

```
(def ^:const width 256)
(def ^:const height 256)
(def ^:const black 0xffffff)
(def ^:const white 0x000000)

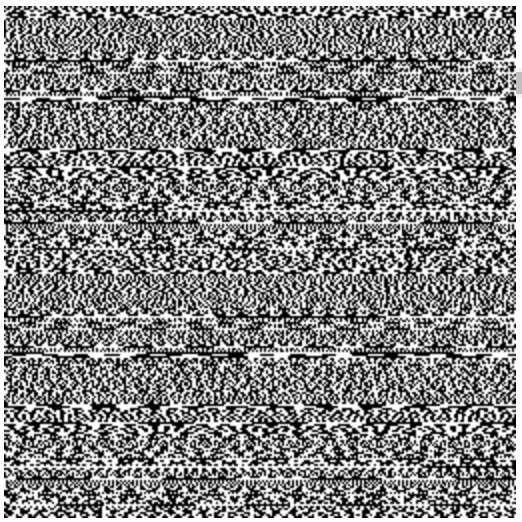
(defn coords [x y]
  (for [m (range x)
        n (range y)] [m n]))

(defn save! [img]
 (ImageIO/write img "PNG" (File. "/tmp/rand-nois

(defn rand-pixel [r]
  (if (== 0 (bit-and (.nextInt r) 1))
    white
    black))

(defn a []
  (let [r (Random.)
        img (BufferedImage. width height Buffere
    (doseq [[x y] (coords width height)]
      (.setRGB img x y (rand-pixel r)))
    (save! img)))
```

Figure 5.2. java.util.Random::nextInt() bits visualized as back and white dots



The code in the example creates a binary image object of 256x256 pixels and colors each pixel randomly black or white. As you can see in the generated image, the dots are aligning horizontally in some places instead of distributing uniformly demonstrating the other limitation of LCG.

See also

- rand-nth takes a sequence and returns a random element from that sequence. Use it when you're interested in a random integer to use to access a collection, instead of the random integer itself.
- shuffle takes a collection as input and returns a random shuffle of the elements back into a new collection. Use shuffle when your only interested is to swap the elements in a collection randomly.

- random-sample takes a collection as input and a probability factor "p". It will return a new sequence that has probability "p" of containing the original elements. The smaller the probability, the less elements are returned in the sequence.

Performance considerations and implementation details

=> `O(1)` **Constant Time, Memory and Computation**

There are no variables involved in calculating the complexity of `rand` or `rand-int` and only 32 bits for the currently generated random are kept in memory at any given time. Clojure implementation is simply delegating to Java `Math.random()` which in turn uses an LCG (Linear Congruential Generator) algorithm. LCG might not be the best random generators for all possible problems, but it's a good trade-off between simplicity, speed and accuracy.

[110] As usual, Wikipedia has a good introductory article about the type of random algorithm used in Java called a Linear Congruential Generator en.wikipedia.org/wiki/Linear_congruential_generator
[111] The infamous RANDU generator of the first era of programming is described in en.wikipedia.org/wiki/RANDU

5.11 WITH-PRECISION

`macro` `since 1.0`

Listing 5.7. → *Arbitrary Precision Math, BigDecimal Precision, Precision Settings*

```
(with-precision [precision & exprs])
```

`with-precision` creates a context in which `BigDecimal` operations are executed with the given precision and rounding mode. `with-precision` is useful when the wrapped expressions are or can produce `BigDecimal` numbers, such as those containing "big M" literals (for example `2M` is the literal to express 2 as a `BigDecimal` number). Without specifying a precision, operations on `BigDecimal` can result in an `ArithmeticException` when the number of decimals is "non-terminating". Here's an example that shows the first 3 decimals of 22/7. Without precision specification the same operation would generate exception:

```
(/ 22. 7) ; ❶
;; 3.142857142857143

(/ 22M 7) ; ❷
```

```
;; ArithmeticException Non-terminating decimal ex

(with-precision 4 (/ 22M 7)) ; ❸
;; 3.143M
```

❶The first decimal expansion is sufficiently precise but not perfect. It is allowed assuming there is no need of additional precision.

❷If we try to get maximum precision, we need to face the problem of a never terminating list of decimals, requiring an infinite amount of memory. This is clearly not allowed and produces an exception.

❸Using `with-precision` we can specify how much memory we want to allocate to store decimal numbers. We ask to stop and the fourth significant decimal (not included).

Listing 5.8. Contract

```
(with-precision <precision> [<rounding>] [<exp

precision :=>
0 <= X <= Integer/MAX_VALUE

rounding :=>
:rounding [CEILING|FLOOR|HALF_UP|HALF_DOWN|HAL
```

```
    exprs :=>
    form1,form2,..,formN
```

❶ `with-precision` takes a mandatory precision argument, an optional rounding type and zero or more forms to evaluate.
❷ the mandatory precision is a positive integer from zero inclusive to `Integer/MAX_VALUE`.
❸ the optional rounding-type is the keyword `:rounding` followed by any of the values in square brackets. If the rounding-type is not specified it is assumed `:rounding HALF_UP` by default.
❹ `exprs` are optional Clojure forms. If no form is passed as input `with-precision` returns `nil`.

Examples

`with-precision` has application in any numerical calculation handling `java.math.BigDecimal` object instances. This can happen directly because of the use of `BigDecimal` Clojure literals or because a function receives them as parameters. `BigDecimal` type is said to be "contagious" because once an expression introduces it somewhere in the code, the rest would be affected. One strategy to design the internals of an application based on `BigDecimal` is to assume that outside the function the parameters are of type double and their precision

has already taken care of. Inside the function we promote to `BigDecimal`, execute calculations and finally return double type again following some contract regarding the required precision.

For example the following shows how to calculate the value of an account after purchasing some number of shares at the current market price:

```
(defn share-qty [account price]
  (let [accountM (bigdec account)
        priceM (bigdec price)]
    (if (zero? priceM)
      0
      (long (with-precision 5 :rounding DOWN (/ a

(defn increment [current price qty]
  (let [currentM (bigdec current)
        priceM (bigdec price)]
    (double (with-precision 5 (+ currentM (* pri

(share-qty 800 1.03)
;; 776

(increment 210 0.38 20)
;; 217.6
```

❶ Input parameters are promoted to `BigDecimal` using bigdec. They are now ready for any subsequent calculation without the fear of causing a loss of precision.

❷ `with-precision` is used to prevent unwanted `ArithmeticException` if the ratio between the current account total and the market price of the share option has an infinite amount of decimals. In this case we also want to always returns integer quantities for the amount of share in the account, so we accept to always round them `DOWN`.

❸ The `qty` parameter was not converted assuming the quantity of shares is always an integer number.

Why do we need BigDecimal?

One of the most discussed use of BigDecimal is for money calculations. The problem stems from the fact that the binary floating point format used to store `double` cannot represents some powers of 10 correctly, including common fractions of a currency, like cents:

```
(- 1.03 0.42)
;; 0.6100000000000001 ; ❶
```

❶ Why all those zeroes?

Although the error is very small, it can accumulate into more worrying amounts when scaled up to millions of operations. For this reason all mainstream languages have some way to deal

with this potential loss of precision. Java has `BigDecimal` for floating point numbers and `BigInteger` for unlimited size integer numbers. Clojure builds on top of those making the code immensely less verbose when promoting, constructing or operating on them.

See also

- bigdec is used to transform other types into `BigDecimal` while `with-precision` deals with the parameters that should be used for the following operations.
- *math-context* is the dynamic variable that `with-precision` is setting with the selected precision and rounding mode. If you need to be more precise around the `MathContext` surrounding `BigDecimal` you can wrap your code directly around it, bypassing `with-precision`:

```
(binding [*math-context* (java.math.MathContext.
  (/ 10M 3))

;; 3.333333334M
```

Performance considerations and implementation details

=> `O(1)` **Constant Time**

`with-precision` prepares all the expressions passed in the block, to obey a specific `MathContext` in which `BigDecimal` operations are executed. The overhead is negligible and independent from the number or type of parameters. For this reason `with-precision` should not be a concern in performance analysis.

5.12 +', -', *', INC' AND DEC'

`function` `since 1.0`

Listing 5.9. → *Integer Math, High Precision Arithmetic, Numeric Processing*

```
(+'
  ([])
  ([x])
  ([x y])
  ([x y & more]))

(*'
  ([])
  ([x])
  ([x y])
  ([x y & more]))
```

```
(-'
  ([x])
  ([x y])
  ([x y & more]))

(inc' [x])
(dec' [x])
```

As expected `+'` `-'` `*'` `inc'` and `dec'` are very similar to their "unquoted" counterparts. They implement the basic operations following similar rules to `+` `-` `*` `inc'` and `dec'`, changing behavior only for results beyond `Long/MIN_VALUE` and `Long/MAX_VALUE`. This chapter will focus mainly on those differences.

Contract

For the general contract please refer to + and inc. The only difference is that when the result "x" of the operation goes beyond `x < Long/MIN_VALUE` OR `x > Long/MAX_VALUE` (lower and upper long precision limits respectively), the result is automatically converted to `clojure.lang.BigInt`.

Note that self-promotion works on the assumption that arguments are integer types (potentially

`clojure.lang.BigInt` instances). If we want to use arbitrary precision operators with floating point numbers, we need to cast arguments to `java.math.BigDecimal` first:

```
(+' Double/MAX_VALUE Double/MAX_VALUE) ; ❶
;; Infinity

(+' (bigdec Double/MAX_VALUE) (bigdec Double/MAX_
;; 3.59538626972463 14E+308M

(+ (bigdec Double/MAX_VALUE) (bigdec Double/MAX_V
;; 3.59538626972463 14E+308M
```

❶Arbitrary precision addition with `+'` works assuming arguments are integer types. If we try to use very large floating point values, it results in `Infinity`.

❷If we want the operator to work on floating point numbers, we need to explicitly cast them using `bigdec`.

❸This is valid for plain addition as well.

Examples

The Diffie-Hellman key-exchange algorithm is first published algorithm that allows two peers to exchange a private key to start an encrypted conversation [112].

Before Diffie-Hellman two parties had to use some physical medium (such as paper) to exchange the key, like during World War 2 to setup the Enigma machine. Once the key is shared, the two peers can start an encrypted conversation using a symmetric key cipher. One important part of the algorithm is that the modulus "p" and the initial secrets "a" and "b" are sufficiently large to prevent brute force attacks. For this reason they are chosen way beyond the long limit of 2^{64} bits space. Related math operations will need to operate on bigint, so arbitrary-precision operators fit perfectly.

What follows is a simplified example that is not using more complicated parameters conventions included later by modern protocols like SKIP (Simple Key-Management for Internet Protocol), but it is still functional and usable:

```
(defn genbig [n]
  (->> #(rand-int 10)
       (repeatedly n)
       (apply str)
       (BigInteger.)
       bigint))

(defn I [n] (.toBigInteger n))

(defn prime?  [n accuracy]
```

```
  (.isProbablePrime (.toBigInteger (bigint n)) a

(defn next-prime [n]
  (loop [candidate n]
    (if (prime? candidate 10)
      candidate
      (recur (if (neg? candidate)
               (dec' candidate)
               (inc' candidate))))))

(defn- mod-pow [b e m]
 (bigint (.modPow (I b) (I e) (I (next-prime m)))

(defn public-share [base secret modulo]
  (mod-pow base secret modulo))

(defn shared-secret [public secret modulo]
  (mod-pow public secret modulo))

(def modulo-pub (genbig 30))
(def base-pub (genbig 30))

(def a-pub (public-share base-pub 123456789N modu
(def b-pub (public-share base-pub 987654321N modu

(def a-common-secret (shared-secret b-pub 123456
a-common-secret
;; 39597682996955611981793282698 3N
```

```
(def b-common-secret (shared-secret a-pub 987654
b-common-secret
;; 39597682996955611981793282698 3N

(= a-common-secret b-common-secret)
;; true
```

❶ `genbig` is an helper function to generate the extremely large numbers needed for strong protection during the exchange. The final product is a Clojure bigint type.

❷ The `I` function has an unconventional name. It is however a catchy mnemonic for the goal of transforming a Clojure int type (small "i") into a Java `BigInteger` (big "I").

❸ `next-prime` is necessary to find the closest prime to a given number. We need this function to make sure the modulo is a prime number. Since we randomly generate the number, we use `next-prime` to grab the closest prime instead. This is where we make good use of `inc'` and `dec'`, because if by any chance the number n we pass in doesn't need the bigint range, we don't need to self promote.

❹ `mod-pow` wraps the same function provided by Java `BigInteger` class. There is no "pow" operation in Clojure that deals with big integers, so we use the one provided by Java.

❺ `public-share` applies the `mod-pow` operation on the base, secret and modulo as from diffie-hellman specifications.

❻ `shared-secret` is exactly the same operation that has been defined as a different function so the name of the parameters can clearly highlight the two different contexts in which the two functions should be used.

❼ this step is where our example usage starts. We first decide a modulo and a base. All values produced by functions ending in `-pub` can be shared in public. So two parties A and B can agree on the base and modulo by email for example, without any encryption necessary.

❽ A can now generate a public "mask" covering the private key "123456789N" that can be sent in clear. Although this public part is sent in clear, there are too many possibilities to try to find which key generated the public mask. B does the same with "987654321N" as the key. Notice that this private keys will never have to leave the local computer.

❾ `a-common-secret` is generated in a similar way to the `public-share`, using what B shared as public and A-private key. This number is what A can use to encrypt all the following messages.

❿ B executes the same operation, obtaining `b-common-secret` that will be used both to decrypt and encrypt messages to A.

⓫ As you can see `a-common-secret` and `b-common-secret` are effectively the same number.

The example above highlights a potential issue when working with bigint: despite being way less verbose than the Java equivalent syntax, some casting back and forth from Java BigInteger is necessary as soon as there is the need of functionalities implemented on the Java side only.

`clojure.lang.BigInt` and `java.math.BigInteger`

You might have noticed that there seem to be a duplication between Clojure and Java. Why a bigdec maps to a `java.lang.BigDecimal` a Clojure bigint maps over a custom Clojure Java class:

```
(type 2M)
;; java.math.BigDecimal

(type 2N)
;; clojure.lang.BigInt
```

The class `clojure.lang.BigInt` appears to be replacing the correspondent `java.math.BigDecimal` when we type a bigint literal at the REPL. The reason for Clojure to implement this class is because `Long.hashCode()` and `BigInteger.hashCode()` are not returning the same value. For Clojure it is important for the two hashcodes to be the same, because if not there could be 1N and 1 inside the same map or Set. `BigInt` is also optimized to make use of numeric primitives if the number they represent is in the precision limit for longs.

See also

- bigint creates a `clojure.lang.BigInt` starting from all numeric types, including `java.math.BigInteger` and even strings: `(bigint "1")` gives 1N.
- If you are sure that the application will never have the need to upgrade to `BigInt` you can just use normal operators, +.
- unchecked-add are the related unchecked-* versions, which maps directly to the Java basic operations. The unchecked version is not going to auto-promote and on crossing the long boundaries it won't throw exception but simply start from the opposite sign side. Unchecked math version should not be used if precision is important.

Performance considerations and implementation details

=> `O(n)` **Linear in the number of arguments**

`+'` `-'` `*'` are linear in the number of arguments. `inc'` and `dec'` are equivalent to adding one to a number, thus they are constant time. The additional overhead necessary to implement self-promotion is negligible compared to the return of value in terms of making big integers operations easy to write. In case

you need specific optimizations, this is the kind of performance loss to expect:

```
(require '[criterium.core :refer [quick-bench]])

(quick-bench (loop [i 0] (if (< i 10000) (recur
;; Execution time mean : 14.304079 µs

(quick-bench (loop [i 0] (if (< i 10000) (recur
;; Execution time mean : 53.189780 µs
```

As you can see, unless the application is specifically dealing with a lot of big integer math, these operations shouldn't be the primary source of concern in performance analysis. However, if you can constrain a performance sensitive part of the application to only work with longs (even better with ints) then it might be worth moving to unchecked operators (of which you can see an example in the performance section about inc).

[112] This is the very accessible Wikipedia entry about the Diffie-Hellman algorithm for key exchange: en.wikipedia.org/wiki/Diffie–Hellman_key_exchange

5.13 UNCHECKED-ADD AND OTHER UNCHECKED OPERATORS

Note

This section also includes `unchecked-subtract`, `unchecked-multiply`, `unchecked-inc`, `unchecked-dec` and `unchecked-negate`.

`function` `since 1.0`

Listing 5.10. → *Math, Overflowing Operators, Integer Math*

```
(unchecked-add [x y])

(unchecked-subtract [x y])

(unchecked-multiply [x y])

(unchecked-inc [x])

(unchecked-dec [x])
```

```
(unchecked-negate [x])
```

The "unchecked-*" family of operators is a group of functions in the Clojure standard library that replicates the behavior of the basic math operators except for not handling more than two arguments and removing the integer overflow check. They can be readily used like this:

```
(unchecked-add 1 2)
;; 3

(unchecked-subtract 2 38)
;; -36

(unchecked-multiply 10 3)
;; 30

(unchecked-inc 100)
;; 101

(unchecked-dec 12)
;; 11

(unchecked-negate 1)
;; -1
```

Contract

input

Unlike other math operators `unchecked-add`, `unchecked-subtract`, `unchecked-multiply`, `unchecked-inc`, `unchecked-dec` and `unchecked-negate` have restricted argument numbers:

- `unchecked-add`, `unchecked-subtract` and `unchecked-multiply` only accepts 2 numeric arguments.
- `unchecked-inc`, `unchecked-dec` and `unchecked-negate` accepts a single numeric argument.

output

They all return the result of the corresponding operation. The final type depends on the types of the arguments, following the rule of the most "contagious" type. See the call-out in this chapter to know more about the precedence rules of type promotion.

Examples

The long type number in Clojure uses to the equivalent signed long type in Java which has 64 bits. One bit is used for the sign, so the available range for a long type goes from -2^{63} to 2^{63} inclusive:

```
[(long (- (Math/pow 2 63))) (long (Math/pow 2 63
;; [-9223372036854775808 9223372036854775807]
```

❶The range of numbers representable using a long type. Note that we need a cast to long because `Math/pow` returns a double in scientific notation.

If we increment the largest possible long number using `unchecked-inc` the result restarts from the other end with a change of sign. Similarly if we decrement the smallest number:

```
(unchecked-inc (long (Math/pow 2 63))) ; ❶
;; -9223372036854775808

(unchecked-dec (long (- (Math/pow 2 63)))) ; ❷
;; 9223372036854775807
```

❶The effect of using `unchecked-inc` on the largest long number representable on 63 bits.
❷Similarly, `unchecked-dec` on the smallest (negative) long number achieves the effect of restarting from the largest one.

As often mentioned in literature, integer overflowing operations are useful in hashing. To understand why this is the case, let's implement a hashing function for strings similar to the one used by Java itself `(.hashCode "string")` but returning longs instead of ints. Instead of producing 2^{32} (around 4.3 billion) keys, this version of the hashing function provides 2^{64} (around 18 billion billions) potentially different keys. Such a high number of possibilities could be justified if we had to hash multiple millions strings and reduce the risk of collisions.

```
(defn scramble [^long x ^long y]
 (unchecked-add (unchecked-multiply 31 x) y))

(defn hash-str [s]
  (let [large-prime 1125899906842597]
    (reduce scramble large-prime (map int s))))

(hash-str "couple words")
;; 16640822305292632 78
```

```
(hash-str "even longer sentences")
;; -76747456202083966l4
```

❶ `scramble` is a function of two longs, "x" and "y", that applies some simple arithmetic. We need to "type-hint" the parameters to long type because the function is passed to reduce later below. The reason why we need to do this is that the compiler is not able to see types at runtime and treats them as boxed objects instead. Unfortunately unchecked operators will use normal math if any of the operands are boxed Java objects, something that is under discussion for a possible fix in later Clojure releases [113].

❷ This is where we need to use something that deals with potentially big numbers, beyond long values. Since we are not interested in the precision but only about the bits relationship, we can accept the operation to overflow. The overflow still produces a valid long which is what we want. The number "31" is a prime number. Prime numbers have the property of being less prone to introduce bit bias in the operation.

❸ `hash-str` is a function taking the arbitrarily long string we want to hash.

❹ We need a large number to start the sum with. Main reason for this is to avoid upper bits to be always set to zero for shorter strings. The large number is also a prime, again to avoid to introduce unwanted bit bias.

❺ We need to transform each character in the string into a number. For this we can use the ASCII table. The next step is to apply `scramble` and sum up the result.

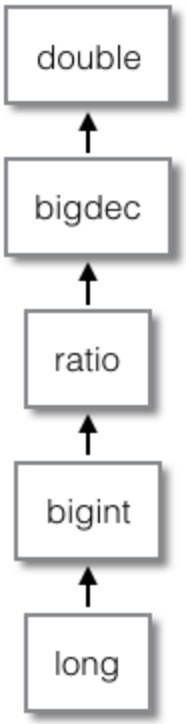
❻ when called, `hash-str` returns a number that converted into binary is the expected 64 bits long.

Contagious Types

Programming languages often support a selection of numerical types with different behaviors. Numerical types can be classified based on the supported precision which describes what are the lower/upper limits each type can handle. Functions in the standard library are affected by the types present in the language and they should describe what happens when operations are applied on different numerical types.

Intuitively, it makes sense that if an operator with greater precision is used along with one that has less, the returned type should have at least the precision of the most precise operand. Clojure roughly applies the precedence list shown in the figure to determine how the result of an operation should be promoted:

Figure 5.3. Contagion rules: the result of an operator will roughly return the most precise of the operands types.



Clojure "roughly" applies the contagion rules the picture describes, because as we have seen in the return types relationships table at the beginning of the current chapter, there are several exceptions to the rule. Also consider that double, the most contagious type, also has a very peculiar definition of precision:

```
(double (- 10 1/3)) ; ❶
;; 9.666666666666668
```

❶A generic operation showing automatic rounding of the last digit of a periodic decimal number.

The above snippet just shows a very well known fact in modern computer floating point arithmetic, since CPU registries only

have a limited precision and some rounding needs to happen for things like periodic floating point numbers. So if precision matters, you should design code that handle precision explicitly instead of leaving the language to decide using contagion rules.

See also

- The unchecked-*-int family of functions is strictly related to the unchecked-* described in this chapter. They differs mainly for two aspects: the arguments will be cast to int (so values above or below `(Integer/MAX_VALUE)` and `(Integer/MIN_VALUE)` respectively will generate error) and the overflow is operated on integers instead of longs. Apart from that, the argument and operation semantic is the same.
- hash is the function that Clojure offers to create hash (int) numbers for all the types in the standard library. Sometimes it delegates to Java sometimes it implements better options (like Murmur3 for collections [114]). It is unlikely you'll have to implement your own hashing function and before doing so check if hash is good enough already.

- *unchecked-math* is a dynamic variable that enable unchecked math even when using the standard arithmetic operators. There are a few conditions that need to be met: it only works on primitive types so type-hint information is essential.

Performance considerations and implementation details

=> `O(1)` **Constant Time**

Given the restriction in the number of arguments, `unchecked-add`, `unchecked-subtract`, `unchecked-multiply`, `unchecked-inc`, `unchecked-dec` and `unchecked-negate` are all constant time. Given missing checks for overflow or self-promotion features, unchecked-* functions also perform better than any other math operators on longs. Please check inc for further information and benchmarks.

[113] For the problem of unchecked operators falling back on checked math in the presence of Boxed numbers, see the following ticket: clojure.atlassian.net/browse/CLJ-1832
[114] The original Murmur3 algorithm was written in C++ ans is visible here: github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

5.14 UNCHECKED-ADD-INT AND OTHER UNCKECKED-INT OPERATORS

Note

This section also includes: `unchecked-add-int`, `unchecked-subtract-int`, `unchecked-multiply-int`, `unchecked-divide-int`, `unchecked-inc-int`, `unchecked-dec-int`, `unchecked-negate-int` and `unchecked-remainder-int`.

`function` `since 1.0`

Listing 5.11. → *Integer math, Overflowing*, *More Use Cases*

```
(unchecked-add-int [x y])
(unchecked-subtract-int [x y])
(unchecked-multiply-int [x y])
(unchecked-divide-int [x y])
(unchecked-inc-int [x])
(unchecked-dec-int [x])
(unchecked-negate-int [x])
(unchecked-remainder-int [x y])
```

`unchecked-add-int`, `unchecked-subtract-int`, `unchecked-multiply-int`, `unchecked-divide-int`, `unchecked-inc-int`, `unchecked-dec-int`, `unchecked-negate-int`, `unchecked-remainder-int` is a family of 8 functions with similar features to the basic math operators that we have seen so far in this chapter.

Differently from the 6 "unchecked" operators (those without the "int" suffix at the end) they cast their arguments to the primitive int type. Another noticeable difference is that they also return an int while other "unchecked" operators would instead promote the return type to accommodate the argument with greater precision, for example:

```
(unchecked-add 2.0 1) ; ❶
;; 3.0

(unchecked-add-int 2.0 1) ; ❷
;; 3

(type (unchecked-add-int 2.0 1))
;; java.lang.Integer
```

❶ `unchecked-add` operates a promotion of the second argument to double type to return a double.
❷ `unchecked-add-int` returns a primitive int type instead.

The `unchecked-*-int` family is especially useful to implement hashing algorithms and integer math in general, when the default Clojure behavior of using the long type is not useful or desirable.

Contract

input

- `unchecked-add-int`, `unchecked-subtract-int`, `unchecked-multiply-int`, `unchecked-divide-int` and `unchecked-remainder-int` all take 2 arguments. The arguments must be numeric and between the integer precision range: `(Integer/MAX_VALUE)` and `(Integer/MIN_VALUE)`.
- `unchecked-inc-int`, `unchecked-dec-int`, `unchecked-negate-int` take a single numeric int argument.

output

They all return the primitive int type result of the operation, overflowing when reaching the limits of int precision (32 signed bits).

Examples

Integers-only algorithms are pretty common in computer science. Hashing is one domain usually restricted to the int (and boxed related `java.lang.Integer`) type and we already illustrated how to extend hashing to long in the example presented for the "unchecked-*" operators.

In the next example, we are going to use a multi-dimensional Java array to process simple images (where pixels can be either 0 or 1). This representation is integer-based for coordinates (a pixel in the image can only have integer coordinates because Java arrays max size is `Integer/MAX_VALUE`) but also for data. To keep things simpler, the color depth is going to be only 2: black (1) or white (0). Since we know that the domain will be integers only, we can take advantage of the "unchecked-*-int" functions described in this chapter.

We want to be able to draw points and lines on a digital canvas. Oblique lines are especially tricky to draw with discrete pixels, forcing us to "interpolate" points so that they are roughly

aligned. The Bresenham's line algorithm [115] can help us figuring out which points are forming the line under this conditions. Since there are many parts to explain, let's start with a few helper functions:

```
(defn- steep? [x1 x2 y1 y2]
  (> (Math/abs (unchecked-subtract-int y1 y2))
     (Math/abs (unchecked-subtract-int x1 x2))))

(defn- adjust-slope [x1 x2 y1 y2]
  (if (steep? x1 x2 y1 y2)
    [y1 x1 y2 x2]
    [x1 x2 y1 y2]))

(defn- adjust-direction [x1 x2 y1 y2]
  (if (> (int x1) (int x2))
      [x2 y2 x1 y1]
      [x1 y1 x2 y2]))

(defn- adjust [x1 x2 y1 y2]
  (->> [x1 x2 y1 y2]
       (apply adjust-slope)
       (apply adjust-direction)))

(defn- swap [steep?]
```

```
    (fn [[x y]]
      (if steep? [y x] [x y])))
```

❶ the line is represented by the four coordinates x1,y1 and x2,y2 which are the two line extremes. `steep?` helps us finding the direction of the slope, upward or downward.

❷ `adjust-slope` swaps coordinates for extremes if the line points upward. We need this along with other transformations to "normalize" the line information before searching for the line points.

❸ `adjust-direction` swaps the extremes of the line considering "x" coordinates.

❹ `adjust` combines transformations together, adjusting the slope incline and the line direction.

❺ `swap` creates a function that based on the slope incline is swapping the x,y point coordinates.

The following `to-points` function takes the two extremes of a line (x1, x2, y1, y2) and applies the Bresenham's line interpolation algorithms to return the collection of the points forming the line. As you can see, we cast or use unchecked int operators throughout:

```
(defn to-points [x1 y1 x2 y2]
  (let [[^int x1 ^int y1 ^int x2 ^int y2] (adjus
        dx (unchecked-subtract-int x2 x1)
        dy (Math/abs (unchecked-subtract-int y1 
    (map (swap (steep? x1 x2 y1 y2))
         (loop [x x1
                y y1
                error (unchecked-divide-int dx 2
                points []]
           (if (> x x2)
             points
             (if (< error dy)
               (recur (unchecked-inc-int x)
                      (if (< y1 y2)
                        (unchecked-inc-int y)
                        (unchecked-dec-int y))
                      (unchecked-add-int error
                        (unchecked-subtract-int 
                      (conj points [x y]))
               (recur (unchecked-inc-int x)
                      y
                      (unchecked-subtract-int er
                      (conj points [x y])))))))))
```

❶let with type hinting is a common idiom to force the Clojure compiler to understand types at compile time. If type hinting was not in place, Clojure would use generic less-than `<` operator for objects instead of basic integers. This behavior can be seen by setting a warning on boxed math with `(set! **unchecked-math** :warn-on-boxed)`

❷The main loop of the computation recurs on the next point coordinates (x,y) which are found by applying increment/decrement on previously found points.

❸We recur in two different ways based on the approximation error resulting from placing the current point in a line. The new x,y coordinates are found using `unchecked-inc-int` or `unchecked-dec-int`.

Finally, after extracting the points forming the line, we need to draw them. The next function `paint!` takes a canvas (a bidimensional array of zeros) and modifies it by drawing the given points:

```
(defn paint! [^"[[I" img points]
  (let [pset (into #{} points)]
  (dotimes [i (alength img)]
    (dotimes [j (alength (aget img 0))]
      (if (pset [i j])
        (aset-int img i j 1))))))
```

```
  img)

(defn zeros [n]
  (take n (repeat 0)))

(defn new-image [n]
  (into-array (map int-array (take n (repeat (ze

(mapv vec (paint! (new-image 12) (to-points 2 3

;; after adding line breaks for clarity:
;; [
;; [0 0 0 0 0 0 0 0 0 0 0 0]
;; [0 0 0 0 0 0 0 0 0 0 0 0]
;; [0 0 0 1 0 0 0 0 0 0 0 0]
;; [0 0 0 0 1 0 0 0 0 0 0 0]
;; [0 0 0 0 0 1 0 0 0 0 0 0]
;; [0 0 0 0 0 0 1 0 0 0 0 0]
;; [0 0 0 0 0 0 0 1 0 0 0 0]
;; [0 0 0 0 0 0 0 0 1 0 0 0]
;; [0 0 0 0 0 0 0 0 0 1 0 0]
;; [0 0 0 0 0 0 0 0 0 0 1 0]
;; [0 0 0 0 0 0 0 0 0 0 0 0]
;; ]
```

❶By type-hinting the image as an array of array of ints (you can actually type-hint a local with a string containing the Java rendition of a bidimensional array of integers "[[I") Clojure can use the right polymorphic calls for ints and avoid any reflective call.

❷aset is used to mutate the array at the given position with "1" if this is a point of the line.

❸For our example we need a few more helpers just to set up an empty canvas. But once everything is ready, we can invoke `to-points` to produce the line approximation between (2,3) and (10,10) (for example) and pass the points to the `paint!` function for the final draw. We need to convert the array into a vector to actually print it on the screen.

As you can see the "line of 1" is approximately a line that sometimes align a couple of "1" on top of each other because of the line incline. When seen on a much bigger canvas this would actually appear as a straight line.

The main reasons to used unchecked-int operators in this case are consistency and performance. Since a Java array is integer-indexed and even the content of the arrays are integers, it makes sense to avoid any transformation (even if implicit) to further stress the fact that all calculations are in the integer range. Secondly, unchecked-int operators are removing

unnecessary boxing and reflective calls resulting in better performance overall.

A bug affecting primitive ints in loops

The Bresenham's algorithm implementation presented in this chapter (the content of the `to-points` function) despite being formally correct, suffers from a bug affecting all Clojure versions up to 1.8 that prevents primitive integers (and also floats) to be used in a loop without being automatically widened by the compiler to longs (or doubles) [116].

Depending on the content of the loop, since `unchecked-inc-int` is casting arguments to int and producing ints, it can actually make things worse compared to a similar use of unchecked-inc because of the constant (unavoidable) casting from long to int.

Since other operations might be dominating over the performance loss due to casting, it's always good to measure before drawing any conclusion. Indeed, this is the case of our Bresenham's algorithm:

```
;; New version of to-points that is using long throughout.

(defn to-points-long [x1 y1 x2 y2]
  (let [[^long x1 ^long y1 ^long x2 ^long y2] (adjust x1 x2 y1 y2
        dx (unchecked-subtract x2 x1)
        dy (Math/abs (unchecked-subtract y1 y2))]
    (map (swap (steep? x1 x2 y1 y2))
         (loop [x x1
                y y1
                error (long (mod dx 2))
                points []]
           (if (> x x2)
             points
             (if (< error dy)
```

```
                  (recur (unchecked-inc x)
                         (if (< y1 y2)
                           (unchecked-inc y)
                           (unchecked-dec y))
                         (unchecked-add error (unchecked-subtract dx
                         (conj points [x y]))
                  (recur (unchecked-inc x)
                         y
                         (unchecked-subtract error dy)
                         (conj points [x y])))))))))

(bench (to-points 3 0 214 197))                    ; ❶
;; Execution time mean : 10.191922 µs

(bench (to-points-long 3 0 214 197))               ; ❷
;; Execution time mean : 9.765748 µs
```

❶Bench measurement for the int only version

❷Bench measurement for the version using longs

As you can see, the difference between the two versions is minimal, showing that other parts of the computation are dominating the cast int-long. Since the results might be different in the next version of Clojure containing the fix for the unwanted widening to long, we suggest to use unchecked-int math operators anyway. See the performance considerations section for further information.

See also

In order of less restrictive behavior:

- If you are not in an integer-only domain, you can use simple unchecked operators: unchecked-add
- If you don't want the overflowing behavior but you need precision instead, the self-promoting operators are what you need: +'.
- hash is the function that Clojure offers to create integer hashes of all the Clojure common objects.

Performance considerations and implementation details

=> `O(1)` **Constant Time**

The `unchecked-*-int` family is the ultimate tool in search of raw speed for math related problems. When a problem can be restricted to the domain of 32 bits integers, the related operators and no overflow checks we can achieve exactly the same bytecode the JVM would produce for a fast loop. Unfortunately the bug affecting loops described in the call-out will prevent the full speed gain in certain conditions. We can show the problem using a simple loop and measuring the difference:

```
(require '[criterium.core :refer :all])

(bench (loop [i (int 0)]
  (if (= i 1000000) i (recur (unchecked-inc-int
```

```
Evaluation count : 79680 in 60 samples of 1328 c
             Execution time mean : 770.432197 µs
    Execution time std-deviation : 41.971531 µs
   Execution time lower quantile : 737.707313 µs
   Execution time upper quantile : 894.806741 µs
                   Overhead used : 1.717123 ns

Evaluation count : 206280 in 60 samples of 3438
             Execution time mean : 291.866997 µs
    Execution time std-deviation : 7.085264 µs
   Execution time lower quantile : 282.293988 µs
   Execution time upper quantile : 307.504766 µs
                   Overhead used : 1.747339 ns
```

❶The first test is using the current Clojure master branch without any additional change. The execution time mean is around 770 microseconds.

❷The second test is conducted against a locally patched Clojure, showing more than a 50% performance increase (at around 291 microseconds)

The suggestion is to use unchecked int operators anyway, since the problem is known and will be likely fixed in the next Clojure releases.

[115] See this article with nice illustration around the mechanism used by the Bresenham's family of algorithms www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html

[116] See the Jira ticket releated to this issue at clojure.atlassian.net/browse/CLJ-1905

5.15 POS?, ZERO? AND NEG?

`function` `since 1.0`

Listing 5.12. → *Number Predicates*

```
(pos? [x])
(zero? [x])
(neg? [x])
```

These predicates are very simple functions to verify if a number is positive, zero or negative, respectively. The reader might wonder why Clojure bothers creating such simple function instead of comparing the number "x" with zero and the reason is that they are of very common use. Besides being immediately readable, these predicates spare a few keystrokes compared to creating a custom function as seen in the following example:

```
(filter #(> % 0) [4 -3 2 -6 10 9 0 -22]) ; ❶
;; (4 2 10 9)

(filter pos? [4 -3 2 -6 10 9 0 -22]) ; ❷
;; (4 2 10 9)
```

❶Removing negative numbers (also including zero) using a comparator.

❷The same can be achieved with significantly less keystrokes by just using `pos?`.

The use of `pos?`, `zero?` or `neg?` on something that is not a number (including `nil`) generates an exception.



Warning

as usual in the presence of `NaN`, be careful processing double numbers as `(pos? Double/NaN)`, `(neg? Double/NaN)` and `(zero? Double/NaN)` are all `false`.

5.16 SUMMARY

In this chapter we've seen how much care Clojure puts into high preforming and precise math functions while maintaining

a concise syntax. Compared to Java, working with large numbers (including in Clojure is much easier and pleasant. When required, Clojure offers ways to go almost as fast as Java, a feature that has been appreciated since the early days. This chapter does not include equality and comparison operators, which require specific treatment. We are going to see them in the next chapter.

6COMPARISON AND EQUALITY

Equality in Clojure takes into account several aspects when comparing "objects" ("object" here is intended in the most general meaning, without relationship to Object Oriented Programming):

- **Their values**. The "value" of an object is Clojure's equality main ingredient. One consequence is that collections don't necessarily need to have the same type to be equal. Values satisfy the common intuition that `(1 2)` and `[1 2]` are equal, despite the fact that `clojure.lang.PersistentList` and `clojure.lang.PersistentVector` are not the same type. The equality operator `=` is mainly based on this principle.
- **Their types**. Types are definitely a relevant aspect for comparison. Types are used for example to add semantic meaning to equality in case of ordered collections.

- **Their identity**. Finally, Clojure also offers a way to check if two objects are the same instance (which most of the time resolves into checking if they have the same address in memory) with identical?.

Clojure equality is designed to be intuitive most of the time (without requiring a specific knowledge of types) with a few corner cases that are going to be illustrated in this chapter. Compared to Object Oriented Programming, functional programming puts less emphasis on container objects and their need for a custom implementation of equality (and hashing as a consequence). Clojure, on the other hand, offers analogous custom data structures called records (created with defrecord) which come with a built-in equality based on attributes.

In this chapter we are going to see all functions that deal with comparison in Clojure, starting from the building blocks and finishing with diff:

- =>> are used mainly as operators in conditions. They know how to deal with Clojure data structures and compare most of them intuitively.
- ==>> is a specialized equality operator for numbers. There are specific comparison rules for different types of numbers.

- < are the standard comparison operators, requiring arguments to implement a notion of ordering when processing results.
- compare provides complete information about the relative ordering of arguments. It is also used in standard sorting when no other comparator is provided.
- identical? allows Clojure to access Java equality by reference.
- hash is the main hashing function in Clojure. Clojure hashing needs a specific optimization to deal with collection as keys in hash-maps compared to basic Java hashing.
- clojure-data-diff builds on top of equality and provides a way to retrieve differences in deeply nested data structures.

Correctly addressing equality in a programming language is a very difficult task and Clojure had to tackle the additional problem of integrating with Java semantic. The overall result is the best possible tradeoff. If from one side there are still rough edges you need to be aware of, on the other Clojure achieved

the perfect balance between the JVM, purely functional data structures and usable equality.

6.1 = (EQUAL) AND NOT= (NOT EQUAL)

`function` `since 1.0`

Listing 6.1. → *Equality, Boolean Logic, Comparison*

```
(=
  ([x])
  ([x y])
  ([x y & more]))

(not=
  ([x])
  ([x y])
  ([x y & more]))
```

`=` ("equals to") is one of the very frequently used Clojure functions. `not=` ("not equals to") is just the opposite of `=` and can help shortening the more verbose `(not (= a b))`. They both take one or more arguments to compare, returning `true` (or `false` for `not=`) when they are the "same". Despite the simple explanation, the meaning of "same" depends on the kind of things being compared which is not necessarily the definition

found in other programming languages. Basic usage is pretty simple:

```
(= "a" "a" "a") ;; true
(not= 1 2) ;; true
```

Equivalence implemented with `=` takes into account values and their representations, which means that in case of collections for example, the comparison happens against the content and not the type of the container. The contract section in this chapter is more specific about rules and exceptions.

Contract

input

"x", "y" and "more" can be any kind of Clojure expression, literal or `nil`. At least one argument is required.

output

`=` returns `true` or `false` depending on types and content. If we call `T1` the type of "x" and `T2` the type of "y", we can broadly split types using the concept of compatibility group:

- **List-like group** (ordered comparison): lists, vectors, subvectors, persistent queues, ranges, lazy sequences.
- **Map-like group** (unordered comparison): hash maps, sorted maps, array maps.
- **Set-like group** (unordered comparison): sets and sorted sets (sorted-map and sorted-set are sorted during sequential access, but ordering is ignored when comparing).
- **Integer group**: byte, short, int, long, big-integer, ratios.
- **Floating point group**: float, double

Any other type not mentioned in the groups above can only be compared with an object of the same type: big-decimal, strings, symbols, keywords, deftypes, defrecords, plain Java types (following normal Java semantic). Given the mentioned compatibility groups, equality between "Object1" and "Object2" is roughly described by the following:

- If Object1 Object2 are compatible and ordered containers, `=` returns `true` if the content is the same, in the same order.
- If Object1 Object2 are compatible and unordered containers, `=` returns `true` if the content is the same, in any order.

- If Object1 Object2 are compatible but not containers, `=` returns `true` if the objects have the same value.
- Return `false` in any other case.

The following section contains many examples going through each comparison in deeper details.

Examples

The contract for `=` is difficult to express in a formal way to include all possible permutations of object types. The best way to understand equivalence is by examples:

- `(= 1N (byte 1))` is `true` because the operands are part of a compatible numeric integer class (byte and big-integer). The main reason for them for being compatible is that their value is not subject to loss of precision during a potential type conversion.
- `(= 1M 1.)` is `false`, because big-decimals and floating point numbers are not in the same compatibility group. There is a dedicated == operator for number equivalence that returns `true` in this case.

- `(= '(1 2 3) [1 2 3])` is `true` because both collections belong to the compatible list-like ordered group and their content is the same in the same order.
- `(= [0 1 2] [2 1 0])` is `false` because vectors belong to the compatible list-like ordered group and even if content and types are the same the order is not.
- `(= #{1 2 3} #{3 2 1})` is `true` because sets belong to the same compatible set-like unordered group.
- `(= (sorted-set 2 1) (sorted-set 1 2))` is `true` despite the "sorted" designation. Sets are always compared without considering ordering, but they could be ordered for sequential access.
- `(= {:a "a" :b "b"} {:b "b" :a "a"})` is `true` similarly to sets comparison, because maps are unordered and compared without considering ordering.
- `(= (sorted-map :a "a" :b "b") (sorted-map :b "b" :a "a"))` is `true`, because sorting of sorted-map is applied only to the sequential access. Sorting is not considered when comparing.

- `(defrecord SomeRecord [x y]), (= (SomeRecord. 1 2) (SomeRecord. 1 2))` is `true`, since the two fields x and y in the two records are the same.
- `(deftype SomeType [x y]), (= (SomeType. 1 2) (SomeType. 1 2))` is `false`, because there is no handling of equality other than what is inherited from Java Object (which by default uses identical?).
- `(= [1 2 3] #{1 2 3})` is `false` because of incompatible group types.
- `(= "hi" [\h \i])` is `false` because of incompatible group types.
- `(= (Object.) (Object.))` is `false`, since Clojure delegates equality for objects to Java semantic (the `.equalsTo(Object o)` method call).

Equality accepts any number of arguments, not just two. We could use a variable number of arguments in the following simulation. In the classic slot-machine gambling game, three (or more) reels come to a stop after spinning for some time. The user wins if the reels are aligned showing the same symbol. Here's how a simulated slot-machine could be implemented in Clojure:

```
(defn generate [& [{:keys [cheat reels]
                    :or {cheat 0 reels 3}}]] ; ❶
  (->> (repeatedly rand)                     ; ❷
       (map #(int (* % 100)))                ; ❸
       (filter pos?)                         ; ❹
       (map #(mod (- 100 cheat) %))          ; ❺
       (take reels)))                        ; ❻

(defn play [& [opts]]
  (let [res (generate opts)]
    {:win (apply = res)                      ; ❼
     :result res}))

(play)
;; {:win false, :result (12 9 19)}

(play {:cheat 100})
;; {:win true, :result (0 0 0)}

(play {:reels 10})
;; {:win false, :result (38 29 31 2 16 7 14 4 46
```

❶The `generate` function is responsible for generating numbers (instead of symbols or patterns like in real slot-machines). It takes two optional parameters for the number of reels to use and a cheating factor going from 0 (no cheating) to 100 (100% cheating) so we can manipulate results. This feature should be obviously well hidden in a real system!

❷We start the generation from an infinite stream of random numbers between 0 and 1.

❸The next step transforms the floating point numbers into integers between 0 and 100.

❹We need to filter out zeros because we don't what the next step using `mod` to incur in a division by zero error.

❺mod is used to force the generation to return the same number multiple time. The higher the "cheating" factor the closest the probability for the numbers to be the same.

❻Finally, we just extract how many generated numbers we need.

❼To check if the generated numbers are a winning position, we use apply with `=` to see if they are all the same.

What is so difficult about equality?

While it is often intuitive to call two things "the same" in spoken language, it is more difficult to find a formal definition in programming. Humans often infer the meaning of equality based on context or other aspect of the conversation, adjusting precision

as a consequence. In computing equality involves a (potentially expensive) bit by bit comparison. What happens in most modern languages though, is that there is an agreed set of types that can skip complete byte comparison.

For Clojure, equality is further complicated by the differences with Java semantic. Java maintains a distinction between "primitives" types and "object" types. When declaring variables in a class as primitive type (like int, short, long and so on), we inform the Java compiler about two facts: that the value of the variable has a well known specific size (for example 4 bytes for `int`) and that when the compiler sees an integer literal (like 1, 2, 3 and so on) there is no actual need to store the same bit pattern over and over again (indeed some of the Java integers and strings are cached in constant pools).

Java `==` works just fine for primitive types. But Java doesn't support primitive types in Collections and automatically performs a "boxing" operation to transform a primitive type into the related "boxed" type (usually the same name with uppercase initial). Boxed numbers of different types are completely different even if their content is not:

```
// primitive types comparison behave intuitively:
System.out.println(1 == 1L)                                    ; /
System.out.println(1 == (short)1)                              ; /
System.out.println(1 == 1.)                                    ; /

// reference types can't compare even when they represent the sam
System.out.println(new Integer(1).equals(new Long(1)))         ; /
System.out.println(new Integer(1).equals(new Short((short)1))); /
System.out.println(new Integer(1).equals(new Double(1)))       ; /
```

Clojure definition of `==` removes any trace of the primitive VS reference duality: `(== 1 1N (Integer. 1) 1. 1M)` returns `true` as one would expect. Despite the huge improvement, `==`

still suffers from the notorious problem affecting floating points represented as binary:

```
(== 0.1 (float 0.1))
;; false
(== 1. (float 1.))
;; true
```

In all fairness, there is not a lot that Clojure can do here since the representation of 0.1 is not exact in the floating point standard that Java uses [117].

The situation is even worse when we consider hashing, the operation that would be used to decide if a key is already present in a hash-map or a hash-set. The level of approximation of a floating point value is even affected by different JVM implementations! It would take a great deal of effort for Clojure to compensate for such differences and guarantee that a program with floating point keys in a hash-map works the same across different architectures. That is why `=` (single equal) approach is to create artificial categories of comparable types and guarantee some intuitive behavior happens without allowing very subtle bugs to appear.

Equality with `=` enables proper hashing behavior in the following key aspects:

1. `=` for collections allow different types of containers with the same content to be represented as a single key in a hash map.
2. `=` for numbers ensure that there is always a distinction between numbers with different precision features, so if they compare `false` and they are not floating point numbers we certainly know that they are different (with floating points you should always consider potential precision problems).

The lesson we should take away from the implementation of `=` and `==` is that you're pretty much free to use anything you want as a key in a hash map, but there could be very subtle bugs happening if you use floating point numbers as keys. Secondly, if you know that you're comparing numbers and only numbers, you should use `==` to remove the artificial categories that `=` introduces. Equality design in Clojure is possibly the perfect compromise between Java constraints, intuitive behavior and the much more liberal use of collections that Clojure enables.

See also

- == is what should be used to compare numbers in the most general case. `==` follows Java semantic for primitive numbers comparison but prevents Java `equals()` requirement to box primitive numbers.
- <= and >= are also dedicated to numbers and additionally checking for relative ordering between arguments.
- identical? verifies if two objects are the same by checking if they live at the same address in memory (following Java semantic). It follows that `(identical? 1. 1.)` is `false` because two `java.lang.Double` objects are created and they live at two different addresses in memory. Since integers between -128 and 127 are cached by the compiller, `(identical? 127 127)` is `true` but `(identical? 128 128)` is `false`.
- distinct can be an interesting option if you are searching all the unique elements in a collection. This is equivalent to remove elements from a collection if they are the same.

Performance considerations and implementation details

=> `O(n) worst-case` **Linear in the number of arguments**

=> `O(1)` **Constant space**

`=` is usually well optimized for most common situations and unless the profiling tool is showing otherwise, `=` is not expected to be a performance bottleneck. Without too much surprise, `=` is linear in the number of arguments:

```
(require '[criterium.core :refer :all])

(def k (take 1000 (repeat 0)))
(def m (take 1000000 (repeat 0)))
(def half (seq (into [] (concat
  (take 500000 (repeat 0))
  (take 1 (repeat 1))
  (take 500000 (repeat 0))))))

(bench (apply = k)) ; ❶
;; Execution time mean : 63.865057 µs

(bench (apply = m)) ; ❷
;; Execution time mean : 62885.110 µs

(bench (apply = half)) ; ❸
;; Execution time mean : 18051.236 µs
```

❶ `=` is applied to a relatively short sequence of 1000 zeros.
❷ `=` is then applied to a 1000 times bigger sequence of zeros, slowing down around 1000 times as expected for a linear relation.
❸ `=` is finally applied to a similarly big sequence where we place a "1" in the middle. We expect `=` to short-circuit at the first `false` occurrence, as confirmed by the benchmark reporting a time between the two previous measures.

When searching for raw speed in tight loops, it might be worth considering the Java `.equals()` method which removes some of overhead of Clojure `=` at the price of being exposed to Java equality semantic. Here's for example what happens when comparing longs:

```
(require '[criterium.core :refer [bench]])
(set! *warn-on-reflection* true) ; ❶

(defn plain= [m n] (= m n))
(bench (let [m 1 n 2] (plain= m n)))
;; Execution time mean : 6.963935 ns

(defn raw= [^Long m ^Long n] (.equals m n))
(bench (let [m 1 n 2] (raw= m n)))
;; Execution time mean : 5.215350 ns  ; ❷
```

❶Setting the warning on reflection dynamic variable to `true` shows any use of `.equals` that forces the compiler to use reflection. We just want to make sure correct type hinting is in place to avoid unfair benchmarking.

❷Please note that timings is related to the specific hardware where the tests are running, so if you try the examples yourself these numbers can be different (although you should see a speed improvement using `.equals`).

As you can see there is around 25% speed improvement by using `.equals`, although there are trade-offs to consider

1. We were forced to deal with explicit type hinting to avoid incurring in reflection penalties.
2. We are now forced to use boxed numbers (upper case `java.lang.Long` instead of plain simple long).
3. Java equality on boxed numbers results in surprises like `(.equals (Integer. 1) (Short. (short 1)))` being `false` that we need to be aware of.

[117] See the following StackOverflow answer to know more stackoverflow.com/questions/588004/is-floating-point-math-broken

6.2 == (DOUBLE EQUAL)

`function` `since 1.0`

Listing 6.2. → *Number Comparison, Java Interoperability*

```
(==
  ([x])
  ([x y])
  ([x y & more]))
```

`==` is a specific equivalence operator for numbers. While `=` (single equal sign) is stricter and only compare numbers if they are from the same numerical category (see `=` for a definition of them), `==` can also compare across categories:

```
(== 1M 1.) ; ❶
;; true

(= 1M 1.) ; ❷
;; false
```

❶A bigdec "1M" and a double "1." are the same for `==`.
❷= "single-equal" considers numbers belonging to certain categories. `bigdec` and `double` are not part of the same category, hence returning `false`.

Contract

input

- `==` requires at least one argument. Calling `==` with a single argument (numeric or not) always returns `true` by default.
- "x", "y" and any other arguments have to be numbers such that `(number? x)` is `true`.

notable exceptions

- `ClassCastException` if there is more than one argument and any argument is not a numeric type.

output

`==` returns:

- `true` if all numbers are the same or have a different type, provided there is a transformation from the least precise type to the more precise type. The available transformations are governed by the Java language specifications for binary promotion [118] or ultimately delegating to the `java.lang.Object::equalsTo()` implementation of the first operand.
- `false` in all other cases.

Examples

One of the main reason for Clojure to include both `=` and `==` operators is that they are specialized for specific tasks without being mutually exclusive. `==` is best for numbers because it respects the general notion that number equivalence is independent from types or binary representation.

In the next example, we want to implement an exchange service to enable transactions between buyers and sellers. Trades happen on different markets and each market provides a slightly different API to list current buy/sell requests. We can match requests through their stock symbols and create a transaction for each compatible pair. Requests are compatible if

their buying price is matched by an equivalent selling price. Here's how requests look like when they enter the system:

```
(def tokyo ; ❶
  [{:market :TYO :symbol "AAPL" :type :buy :bid
   {:market :TYO :symbol "CSCO" :type :buy :bid
   {:market :TYO :symbol "EBAY" :type :sell :bid

(def london ; ❷
  [{:market :LDN :symbol "AAPL" :type :sell :bid
   {:market :LDN :symbol "AAPL" :type :sell :bid
   {:market :LDN :symbol "INTC" :type :sell :bid
   {:market :LDN :symbol "EBAY" :type :buy :bid

(def nyc ; ❸
  [{:market :NYC :symbol "YHOO" :type :sell :bid
   {:market :NYC :symbol "AAPL" :type :buy :bid
   {:market :NYC :symbol "INTC" :type :buy :bid
   {:market :NYC :symbol "PYPL" :type :sell :bid
```

❶The Tokyo market API returns big-decimals bidding values.
❷London rounds amounts to the nearest integer.
❸New York City market uses doubles.

Prices are ultimately a mix of integers, doubles and big-decimals, depending on the market they come from. The service

needs to compare numbers using mixed type arithmetic, something that `==` is most suitable for:

```
(defn group-orders [& markets] ; ❶
  (group-by :symbol (apply concat markets)))

(defn- compatible? [{t1 :type b1 :bid}
                    {t2 :type b2 :bid}] ; ❷
  (and (not= t1 t2) (== b1 b2)))

(defn- matching [orders] ; ❸
  (for [order1 orders
        order2 orders
        :when (compatible? order1 order2)]
    #{order1 order2}))

(defn exchange [listing] ; ❹
  (->> listing
       (map last)        ; ❺
       (mapcat matching) ; ❻
       distinct))        ; ❼

(exchange (group-orders tokyo london nyc)) ; ❽
;; (#{{:bid 22 :market :LDN :symbol "AAPL" :type
;;    {:bid 22.0 :market :NYC :symbol "AAPL" :ty
```

❶ `group-orders` is a helper function to aggregate listings from different markets grouping them by stock symbol. The resulting map contains the symbol as a key and a list of requests as value.

❷ `compatible?` tells us if two requests match. The rules are: they need to be a buy/sell pair and have the same amount. not= is used here to make sure the requests are not buy/buy or sell/sell pairs. `==` is used to verify that the bid price is the same for both requests independently from the type. Notice that both requests are coming in as maps that are destructured in the function definition.

❸ `matching` creates all possible permutations given a list of requests for the same stock symbol. for is perfect to create permutations, including filtering through the compatibility rule applied directly using the `:when` directive to remove unwanted pairs.

❹ `exchange` is the main entry function for the computation. It accepts requests from different markets grouped by symbol and tries to match them.

❺We need to access the last of each input pair. The first is the symbol each listing is grouped by. The last element is instead the list of requests we want to match.

❻We need to mapcat each of the results coming from `matching` because they are all returned contained in their

own sequence (which could also be empty). By concatenating we make sure they are all flattened into the same sequence without "gaps".

❼ The distinct is necessary because `matching` returns matching pairs in both directions: `#{order1 order2}` and `#{order2 order1}` are both returned in the results if they are compatible. distinct gets rid of this duplication (thanks to `=` operators that knows how to deal with unordered set).

❽ We can finally see matching an example request for 22$ worth of Apple stocks that can proceed to trade.

Comparison of floating point types

Floating point arithmetic in programming languages is one of the most debated subjects and each language offers trade-offs and solutions to the problem of representing them precisely.

It all stems from the restrictions in precision imposed by the binary representation of a floating point number (which by default is a trade-off between range and precision [119]. The number doesn't need to be huge or complicated to reveal the problem:

```
0.1 ; ❶
;; 0.1

(BigDecimal. 0.1) ; ❷
;; 0.1000000000000000055511151231257827021181583404541015625M
```

❶Typing "0.1" at the REPL produces a double type.

❷The same number used to initialize a `BigDecimal` instance produces a much larger number of decimals, revealing interesting facts about the concept of "precision".

The number 0.1 for example is printed as expected with its literal form, because rounding is operating to show that correctly. As soon as we turn on full precision creating a `java.lang.BigDecimal` we can see what exactly is stored inside the 64 bits. Even without using big decimals, we can see propagation of the rounding problems with a simple operation:

```
(== 0.3 (+ 0.1 0.1 0.1)) ; ❶
;; false
```

❶Typical case of unexpected result deriving from a lack of precision in floating point representation.

This fundamental imprecision is the reason why currencies shouldn't be represented as "float" or "double". A more exact representation can be achieved by a wrapper like big-decimal for example, which ultimately delegates precision to a Java array (of size `Integer/MAX_VALUE`) inside a `BigDecimal` instance.

See also

- = is a more general equality operator (since it operates on other types than numbers), but at the same more tailored around Clojure data structures (especially using collection as keys in maps and sets). You should use `=` single equal operator in mixed contexts (numbers and other types) or in any other not numerical case.
- identical? verifies if two objects are the same by checking if they live at the same address in memory (following Java semantic). You should use `identical?` mainly in Java interoperability scenarios requiring reference equality.

Performance considerations and implementation details

=> `O(n) worst-case` **Linear in the number of arguments**

=> `O(1)` **Constant space**

`==` has a similar performance profile to =, being linear in the number of arguments and able to short-circuit at the first `false` occurrence without scanning the entire list of arguments.

There is a small (and possibly negligible) difference in speed between the 2 equalities when comparing more than 2

arguments, for example:

```
(require '[criterium.core :refer [quick-bench]])

(quick-bench (= 1 1 1 1 1)) ; ❶
Execution time mean : 86.508844 ns

(quick-bench (== 1 1 1 1 1)) ; ❷
Execution time mean : 63.125153 ns
```

❶ Both `=` and `==` have a catch-all variable arity argument to deal with more than 2 arguments.
❷ `==` is just a bit faster than `=`.

As suggested for `=`, there is some speed-up improvement possible by using Java interop with `.equals` directly on the number (with some trades-off to consider, please check `=` to know more).

[118] The Java Language Specification contains an extensive description of the possible conversions and promotions: www.cs.cornell.edu/andru/javaspec/5.doc.html

[119] The floating point entry in Wikipedia is a nice summary of everything related to this tricky subject: en.wikipedia.org/wiki/Floating_point

6.3 < , > , <= , >=

`function` `since 1.0`

Listing 6.3. → *Comparison Operators, Ordering*

```
(<  ([x]) ([x y]) ([x y & more]))
(>  ([x]) ([x y]) ([x y & more]))
(<= ([x]) ([x y]) ([x y & more]))
(>= ([x]) ([x y]) ([x y & more]))
```

`<` ("less than" or "lt"), `>` ("greater than" or "gt"), `<=` ("less than or equal" or "lte") and `>=` ("greater than or equal" or "gte") are common operators in many languages. They work by assuming the existence of an order relationship between the given arguments and returning `true` or `false` if this relationship holds for all the arguments. Usage of ordering predicates is ubiquitous. In Clojure, they accept only one or more numeric arguments:

```
(< 0 (byte 1) 2 2.1 3N 4M 21/2) ; ❶
;; true

(< 0 (byte 1) 2 2.1 3N 2M 21/2) ; ❷
;; false
```

❶A showcase of all possible numeric arguments supported by the comparison operators.

❷Note that since `(== 2 2M)` then `<` is not satisfied for all arguments. `<=` would instead returns `true` in this case.

Contract

input

- When a single parameter "x" is present, "x" can be of any type (it is not restricted to just numbers). `(< "a")` is for example a valid expression returning `true` while `(< "a" "b")` is not allowed and throws exception.

- When 2 or more parameters are present: in this case arguments must be numeric, that is `(number? x)` must be `true` for all arguments. Integers, decimals, ratios, big integers and big decimals are all accepted as well as other Java numeric types (like `AtomicLong`).

notable exceptions

- `clojure.lang.ArityException` when calling with no arguments.
- `java.lang.ClassCastException` if there is more than one argument and any of them is not a number (that is, `(number? x)` returns `false`).

output

- `true` when there is only one argument (of any type).
- `true` when there are 2 or more arguments and the ordering relationship holds for all of them (see below).
- `false` in any other case.

The order relationships that `<`, `>`, `<=` and `>=` are designed to verify are defined as follows:

- **Strictly monotonically increasing** `<` : when the arguments taken from the left to the right are always in a relationship that goes from the smaller to the bigger (but never equal).
- **Monotonically increasing** `<=` : same as before, but also allow the arguments to be equal (using the same semantic as == for numeric equality).
- **Strictly monotonically decreasing** `>` : when the arguments taken from the left to the right are always in a relationship that goes from the bigger to the smaller (but never equal).
- **Monotonically decreasing** `>=` : same as before but allows the arguments to be equal (using the same semantic as == for numeric equality).

Examples

The fact that `<`, `>`, `<=` and `>=` accept more than two arguments comes handy to check if a quantity belongs to a range. We've seen this already in other chapters throughout the book:

- In the game of life example when talking about for, we saw how to constrain the permutations in search for neighbors cells by using `(<= 0 x' (dec w))`. In this expression the quantity to check `x'` sits in between 0 and `(dec w)`. The expression only allows `x'` that are more or equal to zero and at the same time less or equal to `(dec w)`. This is the same as typing: `(and (>= x' 0) (<= x' (dec w)))`.
- A comparison predicate is often present in loop-recur or recursion in general. This is a consequence of the presence of a condition to exit the loop. You can see an example of this in the Fibonacci sequence implementation included in fn.

In general `<`, `>`, `<=` and `>=` can be used to verify if a collection of numbers is ordered:

```
(apply < [2.1 4 5.2 8 124 9012 1e4]) ; ❶
;; true
```

❶One way to verify if the input collection is ordered.

Other noticeable use of `<`, `>`, `<=` and `>=` is in conjunction with sorting operations like sort or sort-by. Assuming the input

contains numbers, we can for example sort in reverse by using an ordering predicate:

```
(sort > (range 10)) ; ❶
;; (9 8 7 6 5 4 3 2 1 0)
```

❶Reverse sorting a sequence.

In the following example we want to consume the elements from a sequence until they reach some threshold, operation that can be readily accomplished using drop-while and a comparison predicate:

```
(drop-while
  (partial > 90) ; ❶
  (shuffle (range 100))) ; ❷
;; (96 23 46 18 61 84 60 83 56 32 38 54 87...) ;
```

❶With partial we can fix the threshold above which elements are returned.
❷We simulate random numbers with shuffle.
❸The first element in this sequence is always a number bigger than 90.

See also

- compare returns -1, 0, 1 to signal what relationship relates the arguments. Use when you need different actions based on the ordering of the operands. compare also works for other types like strings, provided they are "comparable".
- reverse can be used to invert the order of a collection.

Performance considerations and implementation details

=> `O(n) worst-case` **Linear in the number of arguments**

=> `O(1)` **Constant space**

In short: performance is not generally a problem with comparison predicates. They are "type enabled" and once type hints are in place they correctly propagate to bytecode generation. Observe the following examples and the relevant call within the produced bytecode:

```
(require '[no.disassemble :refer [disassemble]])

(defn boxed< [x y] (< x y)) ; ❶
(println (disassemble boxed<))
;; ... omitted output
;; invokestatic clojure.lang.Numbers.lt(java.lan
```

```
(defn unboxed< [^long x ^long y] (< x y)) ; ❷
(println (disassemble unboxed<))
;; ... omitted output
;; invokestatic clojure.lang.Numbers.lt(long, lo
```

❶The first example does not contain type hints. The compiler does not know what happens at run-time and is forced into the conservative approach of considering the arguments of type `Object`. The relevant bytecode is shown below.

❷The second example contains type hints, so the compiler can infer long types. The resulting bytecode shows that the generated method call can be now more specific about the type (resulting in a speed-up).

Similarly to single or double equal operators, `<`, `>`, `<=` and `>=` consume processing time linearly with the number "n" of arguments. Any pair of inputs determining a `false` predicate immediately terminates the evaluation avoid to walk the entire sequence.

An easy improvement has been proposed on the Clojure Jira board (visible at clojure.atlassian.net/browse/CLJ-1912) to speed up predicates for more than 2 arguments. `<` implementation is fairly easy to follow, so let's have a look at the proposed improvement by first looking at the current version:

```
(defn <
  {:inline (fn [x y] `(. clojure.lang.Numbers (l
   :inline-arities #{2}
   :added "1.0"}
  ([x] true)
  ([x y] (. clojure.lang.Numbers (lt x y))) ; ❶
  ([x y & more]
   (if (< x y)
     (if (next more)                         ; ❷
       (recur y (first more) (next more))    ; ❸
       (< y (first more)))
     false)))
```

❶Comparison predicates delegate the actual comparison to `clojure.lang.Numbers` that dispatches to the relative Java operator or a type specific comparison (for example `java.math.BigDecimal::compareTo()` for `BigDecimal`). The operation is inlined for the second arity, which enables type propagation into JVM bytecode at compile time (see definline).

❷When operands are more than two, recursion takes place to unroll the input sequence. Each iteration we check the predicate against the first two arguments in the sequence. The code short-circuit here into `false` if necessary.

❸If the iteration needs to move forward, we recur with the next pair to check and the rest of the sequence. Note how `(next more)` is evaluated twice.

The improvement removes the double evaluation of `(next more)` that happens when the iteration needs to move forward toward the end of the sequence when there are more than 2 arguments. It can be fixed easily by introducing a let:

```
(defn new<
  {:inline (fn [x y] `(. clojure.lang.Numbers (l
   :inline-arities #{2}
   :added "1.0"}
  ([x] true)
```

```
    ([x y] (. clojure.lang.Numbers (lt x y)))
    ([x y & more]
     (if (< x y)
       (let [nmore (next more)] ; ❶
         (if nmore
           (recur y (first more) nmore)
           (< y (first more))))
       false)))
```

❶The let statement is now preventing the double evaluation of `(next more)`

We can see the benefit using Criterium to benchmark the new function:

```
(require '[criterium.core :refer [bench]])

(bench (< 0 1 2 3 4 5 6 7 8 9 10 11 12))
;; Execution time mean : 258.386277 ns

(bench (new< 0 1 2 3 4 5 6 7 8 9 10 11 12))
;; Execution time mean : 228.939113 ns
```

Despite not being huge improvements on an absolute scale, easy improvements like the above can make an appreciable

difference in very tight loops. The same change would be applied to all other predicates, also including `==` and `=`.

6.4 COMPARE

`function` `since 1.0`

Listing 6.4. → *Comparator, Sorting*

```
  (compare [x y])
```

`compare` is one of the option Clojure offers to compare values. The main difference with comparison predicates like `=, ==, <, >, <=, >=` is that `compare` returns a `java.lang.Integer` (`1`, `0` or `-1`, or more generically a negative, zero or positive number) to indicate that "x" is less, the same or greater than "y" respectively. As a consequence, an exhaustive `compare` for a conditional expression would require 3 branches:

```
(let [c (compare 1 2)]
  (cond
     (neg? c) "less than" ; ❶
     (zero? c) "equal"
```

```
      (pos? c) "greater than"))
  ;; "less than"
```

❶An example showing the 3 branches necessary to cover all possible results of `compare`.

A function that given two arguments returns an integer to indicate relative order is also called a "comparator". `compare` is the default comparator in Clojure (if no other is given) in functions like sort or sort-by. `compare` also works for types that are not necessarily numbers or even custom types, provided they implement the `java.lang.Comparable` interface [120]. Many Java classes already implement `Comparable`, so `compare` can be used for example on `java.lang.String`, `java.util.Calendar` and many others:

```
(import 'java.util.GregorianCalendar)

(def t1 (GregorianCalendar/getInstance)) ; ❶
(def t2 (GregorianCalendar/getInstance)) ; ❷

(compare t1 t2)
;; -1
```

❶The instant t1 is recorded with the first call.
❷Instant t2 is necessarily after instant t1.

Similarly, Clojure implements the `Comparable` interface for several Clojure internal types:

- Vectors created with vector or the `[]` literal syntax (`clojure.lang.Vector`).
- Keywords (`clojure.lang.Keyword`).
- Ratios (`clojure.lang.Ratio`) like the literal `2/3`.
- Refs (`clojure.lang.Ref`) created with ref.
- Symbols (`clojure.lang.Symbol`).

Each Clojure type provides a specific interpretation of `compare` and some of them are not obvious. For example ref are compared based on their creation order (which is mostly to be treated as implementation detail):

```
(compare (ref :a) (ref :a)) ; ❶
;; -1
```

❶ref are compared independently of their content based on creation order. This is an implementation detail used in transactions to establish precedence of `ref` updates.

We are going to see a few examples of how compare works for internal Clojure types and other Java types in the extended examples below.

Contract

INPUT

"x" and "y" are mandatory. "x" can be compared with "y" when:

- `nil` appears as one of the arguments or both.
- `(instance? java.lang.Number x)` and `(instance? java.lang.Number y)` are both `true`.
- `(instance? java.util.Comparable x)` is `true` (there is one generalization to this case when `(identical? x y)` is `true` and at that point "x" can be of any type).
- The implementation of `compareTo()` provided by "x" (if any) allows comparison with an instance of type "y". `compareTo()` is the method required by the `java.util.Comparable` interface.

EXCEPTIONS

- `clojure.lang.ArityException`: `compare` requires exactly 2 arguments.
- `java.lang.ClassCastException`: when "x" doesn't implement the `java.lang.Comparable` interface (but not when `(identical? x y)` is `true`).
- Any other exceptional condition specific to the implementation of `compareTo()` provided by "x".

OUTPUT

- A negative, zero or positive `java.lang.Integer` depending on how "x" and "y" compare. A negative number if "x" is less-than "y", 0 if "x" equals "y", positive otherwise.

```
(compare nil (/ -1. 0)) ; ❶
```

❶An extreme example of exceptional result provided by `compare`. The number `(/ -1. 0)` is negative infinity and `compare` reports that `nil` is smaller. This should be regarded more properly as an example of undetermined result when either operand is `nil`.

Examples

Let's start with some simple example that better explains some of the peculiarities of `compare` described in the contract section. `nil` is accepted as possible argument and is always considered the "smallest" value, even smaller than negative infinity:

```
(def -∞ (/ -0 0.))
(map compare [nil nil "a"] [-∞ nil nil]) ; ❶
;; (-1 0 1)
```

❶ An extreme example of exceptional result provided by `compare`. This should be regarded more properly as an example of undetermined result when either operand is `nil`.

`NaN` is always the same as any other number or itself:

```
(def nan Double/NaN)
(map compare [nan nan 1] [1 nan nan]) ; ❶
;; (0 0 0)
```

❶ `NaN` is another special case of undetermined result when present in comparison.

When the two arguments are the same (as in identical?, which means they are the same Java object) `compare` returns 0. In

this example we are apparently comparing ranges, but compare returns zero because they are the same object instance:

```
(instance? java.lang.Comparable (range 10)) ; ❶
;; false

(compare (range 10) (range 10)) ; ❷
;; ClassCastException

(let [x (range 10) y x] (compare x y)) ; ❸
;; 0
```

❶A `clojure.lang.LongRange` object is not `Comparable`.
❷If we try to `compare` ranges, we get the expected exception.
❸However, we can apparently compare identical ranges when instead we are just in front of the obvious result of equal objects when they are the same instance.

The final decision about how to compare the arguments is delegated to the specific implementation of `compareTo()` defined in the type of the first argument (when available). For example vectors are compared first by size and then by juxtaposing each pair of items:

```
(compare [1 1 1 1] [2 2 2]) ; ❶
;; 1

(compare [1 2 4] [1 2 3]) ; ❷
;; 1
```

❶The first vector contains 4 elements, while the second only 3. This is equivalent to compare `(compare 4 3)` ignoring the content altogether.

❷Provided the size is the same, the first pair is compared. If they are equal, the second pair is compared and so on, until the first pair that is not equal or the end of the vector. The last pair `[4 3]` is the one producing the result.

Clojure strings are `java.lang.String` instances, thus providing the same comparable behavior defined for Java strings. `compare` s1, s2 returns:

- The ASCII distance between the first different characters, if any.
- The difference in length between s1 and s2 when s1 is a substring of s2.
- 0 when the strings are the same.

```
(compare "a" "z") ; ❶
;; -25

(compare "abcz" "abc") ; ❷
;; 1
```

❶"a" and "z" are in ascending order in the ASCII table, with "a" appearing first determining the negative result. There are 25 letters in between.

❷The two strings are of different sizes with "abc" substring of "abcz". Their length is compared.

Clojure keywords and symbols behave like strings with the addition that if they are namespace qualified, then the namespace string comparison takes precedence:

```
(map compare [:a :my/a :a :my/a :abc123/a] ; ❶
             [:z :my/z :my/a :a :abc/a ])
;; (-25 -25 -1 1 3)

(map compare ['a 'my/a 'a 'my/a 'abc123/a] ; ❷
             ['z 'my/z 'my/a 'a 'abc/a ])
;; (-25 -25 -1 1 3)
```

❶When both keywords are namespace qualified, the comparison of the namespace takes precedence following the same rules for strings and just ignoring the name of keyword. When both keywords are not namespaced qualified, the string comparison happens on the keyword name only. If the first keyword is not qualified but the second is, the result is always -1. If the second keyword is namespace qualified but the first is not, then the result is always positive 1.

❷Exactly the same applies to symbols, which are internally stored as keywords.

In the next example we are going to provide Clojure with a way to sort custom types using our definition of `compare`. We want to know what is the closest gas station given an origin location where our car is currently located. Both the gas station and the concept of location have been modeled through a defrecord declaration and enhanced with our own version of `compareTo` to make them comparable. We assume the locations are on a two-dimensional plane for simplicity [121]:

```
(defn- sq [x] (* x x))

(defn- distance [x1 y1 x2 y2]
  (Math/sqrt (+ (sq (- x1 x2)) (sq (- y1 y2)))))
```

```
(defrecord Point [x y distance-origin]
  Comparable
  (compareTo [this other]
    (compare (distance-origin (:x this) (:y this
             (distance-origin (:x other) (:y oth

(defn relative-point [x1 y1 x2 y2]          ; ❹
  (->Point x1 y1 (partial distance x2 y2)))

(defrecord GasStation [brand location]      ; ❺
  Comparable
  (compareTo [this other]
    (compare (:location this) (:location other))
```

❶ `distance` calculates the euclidean distance between two points using the classic formula.

❷ We define a `Point` as a record containing the expected coordinates (x,y) plus an additional function. `distance-origin` is used to calculate the distance of this location from "the origin", an arbitrary selected point from which all other distances are calculated.

❸ Our implementation of `compareTo` uses `compare` on the result returned by `distance-origin` which is a number. Any other implementation obeying the contract of returning an integer (negative, zero or positive) would do.

❹ `relative-point` contains the logic to build a new point. Our point are all requiring a function to calculate a distance given the coordinates. `relative-point` creates a new location which also embeds the information about the "origin" a special location to describe where the user doing the search is located.

❺ The second defrecord definition is used to describe a `GasStation` which contains a brand (or name of the company selling gas) and where it is located.

❻ Also gas stations can compare. A gas station is "less than" another gas station when it is closer to the origin and vice-versa. `compareTo` has been implemented by calling `compare` passing the locations of the two gas station. A

location is represented as a `Point` object, so the `compareTo` implementation of the `Point` object will be invoked to compare gas stations.

By providing a `compareTo` logic to both `GasStation` and `Point` objects, we are now able to sort gas stations based on the position where we are located:

```
(def gas-stations
  (let [x 3 y 5] ; ❶
    [(->GasStation "Shell" (relative-point 3.4 5
     (->GasStation "Gulf" (relative-point 1 1 x
     (->GasStation "Exxon" (relative-point -5 8
     (->GasStation "Speedway" (relative-point 10
     (->GasStation "Mobil" (relative-point 2 2.7
     (->GasStation "Texaco" (relative-point -4.4
     (->GasStation "76" (relative-point 3 -3 x y
     (->GasStation "Chevron" (relative-point -2
     (->GasStation "Amoco" (relative-point 8 -1

(map :brand (sort gas-stations)) ; ❷
;; ("Shell" "Mobil" "Gulf" "Chevron"
;; "Amoco" "76" "Exxon" "Speedway" "Texaco")
```

❶Our coordinates are used to create the gas stations objects. The `relative-point` constructor takes care of creating a `Point` that is related to an origin.

❷sort works without the need for a specific comparator. `compare` will be used by default and dispatched at run-time to the provided `compareTo` implementations.

Beware of the NaN

Should come as no surprise that when `NaN` is involved things become special pretty quick. We already seen the effect of `NaN` when talking about max and `compare` is no exception:

```
(compare Double/NaN 1)
;; 0
(compare 1 Double/NaN)
;; 0
(compare Double/NaN Double/NaN)
;; 0
```

`compare` always returns 0 when `Double/NaN` is present, with the consequence that `NaN` is the same against any other number. Quite confusing, especially when we remember that `(== Double/NaN Double/NaN)` is instead `false`. To add to the confusion consider the following:

```
(sort [3 2 Double/NaN 0])
;; (0 2 3 NaN)
```

```
(sort [2 3 Double/NaN 0])
;; (2 3 NaN 0)
```

sort by default uses `compare` as a comparator and since `compare` of `NaN` is always 0, different results are produced based on the relative ordering of the elements appearing before `NaN` in the vector. This behavior could be especially problematic if we wanted to use collections of numbers as keys in a hash-map (or set), resulting in unexpected unique keys after sorting. The lesson to take away in this case is to refrain from using numbers as keys, first because of potential rounding problems and additionally if they are `NaN`.

See also

- comparator, given a function of two arguments, returns a wrapper that translates the function results into -1, 0, or 1. Use when you want need a comparator-like function and you already have one predicate-like function whose results can be translated into integers.
- identical? verifies if two objects are the same by checking if they live at the same address in memory (following Java semantic). You should use `identical?` mainly in Java interoperability scenarios requiring reference equality.

- = or == should be used if you are only interested in knowing if two arguments are the same, without specific interest in their relative order relationship.

Performance considerations and implementation details

=> `O(1)` **Best case for some standard library types**

=> `O(n)` **Worst case for other standard library types**

`compare` is essentially a polymorphic function for dispatching comparison logic at runtime. There is no single performance profile, but many of them depending on the types the comparison is applied to. The following table is a quick summary of what to expect comparing the most common Clojure and Java types:

Table 6.1. Various performance profiles for `compare` based on

the type of the arguments.

Types	Best Case	Worst Case
All types	• `O(1)` one argument is `nil` • `O(1)` arguments are identical?	Same as Best Case.
Vector	• `O(1)` different length • `O(1)` same length with different first element	• `O(n)` same length and same content
Numbers	• `O(1)`	• `O(1)`

Strings	• `O(1)` different first character	• `O(n)` same length and same content
Keyords And Symbols	• `O(1)` one is not namespace qualified • `O(1)` different first char in namespace • `O(1)` different first char in name	• `O(n)` no namepsace, same name length and same name content • `O(m*n)` same namespace, same name

In general, you'll have to pay attention at the potential linear scanning of very big vectors or strings.

[120] Please see docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html about Comparable in Java

[121] The formula to calculate the distance between to geographical locations would probably be different, like the Haversine formula we saw in the post office problem

6.5 IDENTICAL?

`function` `since 1.0`

Listing 6.5. → *Comparison, Reference equality*

```
(identical? [x y])
```

`identical?` is possibly the strongest of the comparison functions in Clojure. `identical?` main purpose is to determine if the two given arguments are exactly the "same" . "Same" in this context means to verify if the arguments are stored at the same memory location. When `identical?` returns `true` it means that there is a single object allocated in memory that is referenced by two (possibly distinct) bindings, for example:

```
(let [x #{1 2 3}    ; ❶
      y x]          ; ❷
```

```
    (identical? x y))
  ;; true
```

❶"x" is a local binding associated with a hash-set.
❷"y" is another local binding associated with "x". "x" and "y" are thus two references pointing at the same entity.

`identical?` has a few specific use cases related to object identity and should not be used as a general purpose equality mechanism. `identical?` results are sometimes surprising because several basic types (both Java and Clojure) implement "interning", an automatic caching mechanism to improve performance. Please check the section "Surprises using `identical?`" for a few interesting cases.

Contract

input

"x" and "y" are mandatory arguments of any type (including `nil`).

notable exceptions

- `clojure.lang.ArityException`: when not exactly 2 arguments are given.

output

- `true` when the arguments are references to the same object instance.
- `false` in any other case.

Examples

`identical?` is used when we need to keep track of the identity of a particular value that could otherwise be confused with other instances that are logically the same. This condition happens for example when checking "sentinel" values. A sentinel is a special value that is normally used in data-driven algorithms [122].

Sentinel values can be mixed freely with normal business data, since there is a mutual agreement between the producer side and the consumer side about the meaning of the special value. Sentinels are especially useful for daemon-like processes: when the lacking of new data is insufficient condition to signal the end of the input, a sentinel can be used to force a result.

The following example shows a consumer-producer pair agreeing on a sentinel object to stop computation. The daemon thread prints the hash of the object sent by the client. The server executes in an infinite loop, waiting for any number of

events sent by the client. Since any object can be hash-coded, we need a way for them to mutually agree on a signal to exit the loop:

```
(import 'java.util.concurrent.LinkedBlockingQueu

(def channel (LinkedBlockingQueue. 1)) ; ❶
(def SENTINEL (Object.))               ; ❷

(defn encode []                  ; ❸
  (let [e (.take channel)]
    (if (identical? SENTINEL e)
      (println "done")
      (do (println (hash e))
          (recur)))))

(defn start []                   ; ❹
  (let [out *out*]
    (.start (Thread.
        #(binding [*out* out]
          (encode))))))

(do
  (start)
  (.offer channel :a)
  (.offer channel (Object.))     ; ❺
  (.offer channel SENTINEL)
  (.offer channel :a))           ; ❻
```

```
;; -2123407586
;; 1430420663
;; done
```

❶We use a blocking queue to orchestrate the communication between the producer thread (the REPL thread in this case) and the consumer. The server thread can then run in am infinite loop waiting for input. The `take` operation on the queue is blocking, so the server waits for at least one element to be present each loop.

❷The sentinel is a generic `java.lang.Object` instance defined in the current namespace.

❸ `encode` contains a loop to examine the next event offered by the blocking queue. The loop stops as soon as the sentinel is identified. Since any object could be sent by the producer, we don't want to use = single equal to eliminate the risk of confusing another object instance with our sentinel.

❹ `start` is dedicated to preparing the two thread for correct communication, for example making sure that they both use the same standard output.

❺We can send any kind of event to the channel (even another object instance) because we are sure nothing can return `true` when compared with `identical?` if not the sentinel object itself.

❻But when we actually send the `SENTINEL` object, the loop exits as the "done" message shows. Additional offers to the channel won't print the hash code anymore.

Surprises using `identical?`

Interning is a form of caching used by compilers to reduce the number of allocated objects. Java uses them for string literals (the strings that appear in double quotes throughout the code) and Clojure inherits the same mechanism for `java.lang.Long` instances from -127 to 128. Clojure additionally introduces interning of keywords. The implication of interning is that `identical?` might return `true` or `false` comparing some types of literals, exposing their implementation details:

```
(map identical? ["A" 1 \a :a] ["A" 1 \a :a]) ; ❶
;; (true true true true)
```

❶A demonstration of automatic interning for a few data literals.

On the other hand, other data literals are not subject to interning, or are proper global constants:

```
(identical? '() '()) ; ❶
;; true

(identical? 2/1 2/1) ; ❷
;; true
```

❶The empty list literal is the only instance of the empty list inside a running JVM.

❷The `clojure.lang.Ratio` instance `2/1` is saved internally as long type: `(class 2/1)` is `java.long.Long`. Ratios in general are not interned.

Interning doesn't work for all possible long and char type literals, but just the most used between -127 and 128 (for longs) and 0-127 ASCII conversion (for chars). On the other hand interning works for all string literals. Complete string interning is done under the assumption that an average program would just contain a limited number of strings in source code. Larger string instances are of course created during the application lifetime that are not interned.

The following example shows a collection of literals that are not identical, either because they are out of range of their interning capabilities or because they don't support interning at all:

```
(map identical? [128 \λ 1N 1M 1/2 1. #"1" [1] '(
                [128 \λ 1N 1M 1/2 1. #"1" [1] '(
;; (false false false false false false false fa
```

❶A collection of data values that are not interned. Note for instance that symbols are not interned.

Finally, notice that as soon as a native type is used to create a new instance of the corresponding reference type, the interning is no longer possible because Java doesn't have a chance to look into the cache to return the interned instance:

```
(identical? (Long. 100) 100) ; ❶
;; false
```

❶ By wrapping the number 100 with the corresponding number initializer we explicitely ask the JVM to create a new instance of the number ignoring any interned option.

To give Java the option to look into the interned cached of numbers, we need to use `Long/valueOf` instead of the constructor. This is exactly the mechanism that Java uses to transform a native parameter into a reference, for example passing arguments to methods:

```
(identical? (Long/valueOf "100") (Long/valueOf "
;; true
```

❶ `Long/valueOf` is interning-aware.

The last interesting behavior of `identical?` discussed here is related to "boxing". Boxing is the informal name given to the action of wrapping a native data type with the corresponding full fledged class. Let's compare the following application of `identical?` where the arguments are bound as vars or let respectively:

```
(def a 1000)
(def b a)
(identical? a b) ; ❶
;; true

(let [x 1000 y x]
  (identical? x y)) ; ❷
;; false
```

❶On creation of the var "a" the primitive `1000` has been passed to the constructor of `clojure.lang.Var` as `(Long/valueOf 1000)` because a var is created from a generic `java.lang.Object`. Note that `1000` does not belong to the pool of interned integer constants. Clojure automatically de-reference any usage of "a" from now on, including the definition of the second var "b", passing the same long instance contained in "a" to "b". Note that `1000` was already transformed into a reference type before and it doesn't need this again. As expected, comparing the var "a" to "b" with `identical?` reports that we are talking about the same long instance.

❷The same number `1000` is now used in a `let` block without var indirection. "y" is assigned "x" again but as a native primitive value. We would expect the same equivalence as before but this is not happening, showing that two independent instances of `1000` have been built.

In the second case, "x" is a native type without a `var` wrapping. In the generated code, the primitive long `1000` is passed as argument to `clojure.lang.Util/identical?(Object x, Object y)` forcing Java to box the primitive into a reference `java.lang.Long` twice, one for "x" and one for "y". This is equivalent to the following rewrite:

```
(let [x 1000 y x]
  (identical? (Long/valueOf x) (Long/valueOf y)))
;; false
```

❶This is a rewrite of the previous snippet showing what kind of behavior is determining the unexpected result from `identical?`.

As you can see from the example, `identical?` is subject to a lot of exceptional behavior that you should be aware of. Clojure equality `=`, when possible, should be the preferred option for equivalence testing.

See also

- Use compare when you are interested in comparing quantities (not references) and respecting the relative order of the operands.
- = is the most generic and flexible of the comparison operators. It's not the best choice for mixed numbers comparison, but it works on collections and other Clojure data types.
- == is the operator dedicated to numerical equivalence. Use it when you are interested in comparing numerical quantities instead of references.

Performance considerations and implementation details

=> `O(1)` function generation

`identical?` is a constant time operation. It is also quite fast considering the lack of dispatching logic and narrow focus on reference equality only.

The implmenentation of `identical?` is delegated to `clojure.lang.Util` class on the Java side that in turn just calls Java `==` double equal operator. Differently from other Clojure wrapping methods, `clojure.lang.Util/identical(Object k1, Object k2)` signature takes only objects. This prevents dispatching of primitives types from Clojure, forcing the auto-boxing that we've have seen in the previous examples.

[122] Please read the Wikipedia article about sentinels for more background: en.wikipedia.org/wiki/Sentinel_value

6.6 HASH

Note

This section also mentions other related functions such as: `mix-collection-hash`, `hash-ordered-coll` and `hash-unordered-coll`.

`function` `since 1.0 (hash)`

Listing 6.6. → *Hashing function*

```
(hash [x])
(hash-ordered-coll [coll])
(hash-unordered-coll [coll])
(mix-collection-hash [hash-basis count])
```

`hash` is the default hashing function implementation in Clojure. `hash-ordered-coll`, `hash-unordered-coll` and `mix-collection-hash` are functions to help with the integration of objects and collections with the Clojure hashing implementation. `hash` takes an object of any kind and turns it into a 32 bits (signed) integer:

```
(hash "hello") ; ❶
;; 1715862179

(type (hash "hello")) ; ❷
;; java.lang.Integer
```

❶ `hash` implements the default Clojure hashing algorithm.

❷ `hash` returns a `java.lang.Integer`.

Note

Java comes with its own hashing algorithm accessible through the `hashCode()` method on each object instance. Clojure supports Java hashing requirements by implementing `hashCode()` on its own collections. However, Java hashing falls short of certain idiomatic Clojure scenarios, such as using collections as keys in associative data structures. This is one of the main reasons Clojure provides its own hashing function.

Contract

input

- `hash` : "x" is the only required argument. It can be any type including `nil` .
- `hash-ordered-coll` and `hash-unordered-coll` : "coll" is the only mandatory argument. It has to implement the `java.lang.Iterable` interface.
- `mix-collection-hash` : "hash-basis" and "count" are both required arguments of type long.

notable exceptions

- `NullPointerException` : when passing `nil` to `hash-ordered-coll` or `hash-unordered-coll` .
- `ClassCastException` : if the argument to `hash-ordered-coll` or `hash-unordered-coll` does not implement the `java.lang.Iterable` interface.

output

- `hash` returns a `java.lang.Integer` number between -2^{31} and $2^{31}-1$. When "x" is a number, string or implements `clojure.lang.IHashEq` , the output is consistent with Clojure hashing implementation. For all other types, `hash` delegates to `.hashCode()` from `java.lang.Object` .

- `hash-ordered-coll` and `hash-unordered-coll` returns a `java.lang.Long` number between -2^{31} and $2^{31}-1$.
- `mix-collection-hash`: "hash-basis" and "count" are both required arguments of type `long`.

Examples

The last changes to the `hash` function (including the introduction of `hash-ordered-coll`, `hash-unordered-coll` and `mix-collection-hash`) are relatively new. Before Clojure 1.6, it was possible to produce inefficient programs by using composite keys in maps. The inefficiency was the result of frequent collisions in such situations [123]. To understand why Clojure needs its own hashing, have a look at the following example:

```
(def long-keys [-3 -2 -1 0 1 2])
(def composite-keys [#{[8 5] [3 6]} #{[3 5] [8 6

(map (memfn hashCode) long-keys) ; ❶
;; (2 1 0 0 1 2)

(map (memfn hashCode) composite-keys) ; ❷
;; (2274 2274)
```

❶ We can see the effect of Java `hashCode()` on numbers of type `long`. Java simply combines the upper and lower bits of a 64 bits long integer to shrink them to the required 32 bits size. But in doing so it creates some evident collisions between negative and positive numbers.

❷ Another problem with `hashCode()` manifests on small collections with repeating patterns of items (a common case with Clojure). The sets presented here collide if we use `hashCode()`.

Java `hashCode()` creates relatively easy collisions, for example on longs, vectors or sets in algorithms or data structures that require hashing. `hash` improves on Java taking these factors into account:

```
(map hash long-keys) ; ❶
;; (-1797448787 -1438076027 1651860712 0 1392991

(map hash composite-keys) ; ❷
;; (2055406432 -916052234)
```

❶ `hash` improves hashing on longs compared to Java by removing the effect of simple compression from 64 to 32 bits.

❷ Similarly, `hash` has been extended to Clojure collections producing evenly distributed hash numbers.

Clojure collections uses the same hashing function provided by `hash` internally, so we are free to use numbers or small collections as keys without the risk of generating frequent collisions. When it comes to Java interoperability scenarios though, we need to be careful:

```
(import 'java.util.ArrayList)

(def k1 (ArrayList. [1 2 3])) ; ❶
(def k2 [1 2 3])

(def arraymap {k1 :v1 k2 :v2}) ; ❷
;; IllegalArgumentException: Duplicate key [1, 2

(= k1 k2)
;; true

(def hashmap (hash-map k1 :v1 k2 :v2)) ; ❸
(= (hash k1) (hash k2))
;; false
;; #'user/hashmap
```

❶In this interoperability scenarios with Java, we have 2 collections with the same content but different types. One is a vector the other a `java.util.ArrayList`.

❷We are unable to create an array-map because it does not use hashing to check for the presence of keys. It uses clojure equality that correctly claims the two collections are the same.

❸We are instead able to create a hash-map because the two collections use different hashing algorithms and they appear as different keys.

The different behavior between array-map and hash-map is consistent with their design goals and `ArrayList` uses a different hashing algorithm than Clojure collections. A similar behavior should be expected for other custom types (not necessarily collections) not implementing the `clojure.lang.IHashEq` interface.

If we wanted to use Clojure hashing on collections consistently with their content (but independently from their type), we could use `hash-ordered-coll` or `hash-unordered-coll`. With the "unordered" version, the output hash doesn't change when the order of the content changes. This could be useful to mix different collection types as keys in a Clojure hash-map while guaranteeing hash consistency:

```
(import 'java.util.ArrayList)
(import 'java.util.HashSet)

(defn hash-update [m k f] ; ❶
  (update m (hash-unordered-coll k) f))

(def k1 (ArrayList. [1 2 3]))
(def k2 (HashSet. #{1 2 3}))
(def m (hash-map))

(-> m ; ❷
  (hash-update [1 2 3] (fnil inc 0))
  (hash-update k1 (fnil inc 0))
  (hash-update k2 inc))
;; {439094965 3}
```

❶ `hash-update` is a small wrapper function around the normal update for maps. Instead of using the given key directly, `hash-update` is first calling `hash-unordered-coll`.

❷ The `hash-map` "m" repeatedly updates using `hash-update`. Despite the very different collections used as keys, they all update the same value (instead of creating new keys).

The example above implies that `hash-unordered-coll` generates the same key for different collection types provided

they have the same content in any order:

```
(=
  (hash-unordered-coll [1 2 3]) ; ❶
  (hash-unordered-coll [3 2 1])
  (hash-unordered-coll #{1 2 3}))
;; true

(=
  (hash-ordered-coll [1 2 3]) ; ❷
  (hash-ordered-coll [3 2 1])
  (hash-ordered-coll #{1 2 3}))
;; false
```

❶We can verify that `hash-unordered-coll` generates the same hash number for diversely ordered collections.

❷If we need ordering to determine different hashing numbers, we can use `hash-ordered-coll`.

`hash-ordered-coll` and `hash-unordered-coll` work for collections implementing `java.lang.Iterable` interface. Iterable interfaces are the big majority both in Clojure and Java. Unfortunately, if a collection is not iterable, we can't use the hashing functions for collections seen so far.

`java.util.HashMap` is a common case of not iterable collection:

```
(import 'java.util.HashMap)

(hash-unordered-coll (HashMap. {:a 1 :b 2})) ; ❶
;; ClassCastException java.util.HashMap cannot b
```

❶We can't use `hash-unordered-coll` directly on a `java.util.HashMap` type.

In the next example we are going to design a Clojure-compatible hashing function that works on `java.util.HashMap`, enabling us to compare hashing on a mix of Clojure and Java maps. We can do this by iterating the `hash-map` and sum the `hash` for each pair of key-values. There is however one last problem we need to deal with, something that Clojure implements for us with `hash-unordered-coll` that we need to replicate.

Hashing algorithms often suffer from a problem related to how much of the bits in the hash change when altering the input. A good hashing algorithm produces an "avalanche effect" when each change in the input determines at least half of the bits in the output to change (possibly all, although this is not always possible in practice). The avalanche effect is better achieved with one last step that "mixes" the bits to maximize the changes.

Since we are going to implement our own hashing algorithm, we also need to explicitly call `mix-collection-hash` as the last step:

```
(defn hash-java-map [^java.util.Map m]
  (let [iter (.. m entrySet iterator)] ; ❶
    (loop [ret 0 cnt 0]
      (if (.hasNext iter)
        (let [^java.util.Map$Entry item (.next i
              kv [(.getKey item) (.getValue item
          (recur
            (unchecked-add ret ^int (hash kv)) ;
            (unchecked-inc cnt)))
        (.intValue ^Long (mix-collection-hash re

(= (hash (HashMap. {1 2 3 4})) ; ❺
   (hash {1 2 3 4}))
;; false

(= (hash-java-map (HashMap. {1 2 3 4})) ; ❻
   (hash {1 2 3 4}))
;; true
```

❶To iterate the `java.util.HashMap` we need to go through its `EntrySet` first, which is an iterable object.
❷An `Iterator` object is stateful and advances each time we call `.next` on it.
❸Note that to `hash` a `java.util.HashMap$Entry` we need its key and value components to form a vector.
❹After summing all hashed map pairs, we call `mix-collection-hash` to ensure a good avalanche effect.
❺`hash` called on `java.util.HashMap` results in a different number than the same called on `clojure.lang.PersistentArrayMap`.
❻But if we use `hash-java-map` we enable Clojure-style hashing on Java map objects.

Performance considerations and implementation details

=> `O(n)` **linear in the number of items to hash**

The `hash` function on collection objects is linear in the number of items to hash. Collections or scalar types with a custom `hashCode()` implementation might be subject to different complexity costs, as `hash` delegates to them if they are present. In terms of performance analysis, `hash` uses the Murmur3 algorithm which is generally very efficient.

[123] A nice summary on the topic of hashing in Clojure is available on the <<groups.google.com/d/msg/clojure-dev/lWXYrjaDuIc/WE_LUtll7VgJ,Clojure mailing list>> written by Mark Engelberg

6.7 CLOJURE.DATA/DIFF

`function` `since 1.3`

Listing 6.7. → *Nested Differences, Structural Comparison*

```
(diff [a b])
```

`diff` builds on top of Clojure equality operators to retrieve differences between arbitrarily nested data structures. While `=` would just return `false` if the arguments are different, `diff` also describes in which way they are:

```
(require '[clojure.data :refer [diff]]) ; ❶

(diff {:a "1" :b "2"} {:b "2" :c "4"}) ; ❷

;; ({:a "1"}  ; ❸
;;  {:c "4"}  ; ❹
;;  {:b "2"}) ; ❺
```

❶Note that `diff` is not part of the `clojure.core` namespace and thus is not available by default.
❷ `diff` takes two arguments and returns a sequence of 3 elements.
❸The first item in the result is what is present only in the first argument and not present in the second.
❹Likewise, the second item in the result is what is present only in the second argument but not in the first.
❺The last and final item in the results is what is common between the arguments, if any.

`diff` works across all Clojure data structures and scalar types with some limitations that are going to be illustrated in this chapter.

Contract

input

- "a" and "b" are the two mandatory arguments. They can be of any type including `nil`.

notable exceptions

- `clojure.lang.ArityException`: when not exactly 2 arguments are given.

output

A sequential collection (list or vector) of 3 elements at index 0, 1 or 2. The resulting triplet contains:

- `[a b nil]` when "a" and "b" don't have anything in common or when the types of "a" and "b" are not compatible (see below).
- `[only-in-a only-in-b common-items]` when "a" and "b" have something in common and they have a compatible types.

For collections, assuming "a" is of type A and "b" of type B, A and B are compatible if:

- They are both Java arrays (such that `(.isArray (class a))` and `(.isArray (class b))` are both `true`).
- They are both `java.util.Set` (such that `(instance? java.util.Set a)` and `(instance? java.util.Set b)` are both `true`.
- They are both `java.util.List` (such that `(instance? java.util.List a)` and `(instance? java.util.List b)` are both `true`. This is what makes lists and vectors compatible.

- They are both `java.util.Map` (such that `(instance? java.util.Map a)` and `(instance? java.util.Map b)` are both `true`.

For scalars (any other type that is not a container), `diff` follows = single equal compatibility rules.

Once established that "a" and "b" are compatible for `diff`, then the result contains:

- At index 0: an element of type A that contains all items (and sub-items) only present in "a" but not in "b". When "a" and "b" don't have anything in common, the index-0 element contains "a" itself.
- At index 1: an element of type B that contains all items (and sub-items) only present in "b" and not in "a". When "a" and "b" don't have anything in common, the index-1 element contains "b" itself.
- At index 2: the elements common to both "a" and "b" as a list or vector, or a single `nil` in case they don't have anything in common.

The example section contains example of the most interesting `diff` applications.

Warning

Any interleaving `nil` occurrence in the resulting triplet should be ignored, as it could be the result of `diff` internal processing and not an actual occurrence of `nil` in the input arguments. `diff` is thus not well equipped to handle input with explicit `nil` or empty collections, as it would be problematic to tell them apart from missing elements in the resulting triplet.

Examples

Let's start with a list of small examples to show how `diff` behaves in relation to the type categories illustrated in the contract section. Remember to require `clojure.data` if you want to use `diff` like the examples below:

```
(diff 1.0 1)                ; ❶
;; [1.0 1 nil]

(diff [1 "x" 3 4]           ; ❷
      '(1 "y" 3 5))
;; [[nil "x" nil 4]
;;  [nil "y" nil 5]
;;  [1 nil 3]]
```

```
(diff {:a "a" :b {:c "c"}} ; ❸
      {:a 1 :b {:c 2}})
;; ({:a "a" :b {:c "c"}}
;;  {:a 1 :b {:c 2}}
;;  nil)

(diff [1 {:a [1 2] :b {:c "c"}}]            ; ❹
      [1 {:a [1 3] :b {:c "c" :d "d"}}])
;; [[nil {:a [nil 2]}]
;;  [nil {:a [nil 3] :b {:d "d"}}]
;;  [1 {:a [1] :b {:c "c"}}]]

(diff (int-array [1 2 3])                   ; ❺
      (int-array [1 4 3]))
;; [[nil 2] [nil 4] [1 nil 3]]

(diff #{:a :c :b} #{:c :b :a})              ; ❻
;; [nil nil #{:a :b :c}]

(diff {"x" 42} (sorted-map :x 42))          ; ❼
;; java.lang.ClassCastException
```

❶Following `(= 1.0 1)` `false` semantic.
❷ `nil` appears in the collection as part of `diff` internal processing. What matters is that "x" and 4 are only in the first argument, "y" and 5 are only in the second and 1-3 are in common. Note that vectors and lists are compatible arguments.
❸The two maps have nothing in common.
❹If the value of a map key is another collection, `diff` shows which elements are in common going inside the nested level.
❺ `diff` working with Java array.
❻Sets are working as expected.
❼A known bug afflicting `diff` with sorted-map [124].

`diff` is a powerful tool to compare nested data structures and obtain an immediate feedback about how they differ. Services written in Clojure, for example, often produce JSON or EDN output with arbitrary nesting of data structures. If we wanted to implement changes to the service and we wanted to be sure not to introduce any regression, we could compare the output of the new service with the old and check for differences. Some of them might be expected, others aren't.

In the following example, a service is returning metadata about Clojure libraries and their dependencies. Here's a sample response from the live service for project "prj1":

```
(def orig
  {:defproject :prj1
   :description "the prj"
   :url "https://theurl"
   :license {:name "EPL"
             :url "http://epl-v10.html"}
   :dependencies {:dep1 "1.6.0"
                  :dep2 "1.0.13"
                  :dep6 "1.7.5"}
   :profiles {:uberjar {:main 'some.core :aot "a
              :dev {:dependencies {:dep8 "1.6.3"
                    :plugins {:dep9 "3.1.1" :dep
```

We now make the same request to the new service, that is just different code using the same database/infrastructure:

```
(def new-service
  {:defproject :prj1
   :description "the prj"
   :url "https://theurl"
   :license {:name "EPL"
             :url "http://epl-v10.html"}
   :dependencies {:dep1 "1.6.0"
                  :dep2 "1.0.13"
                  :dep6 "1.7.5"}
   :profiles {:uberjar {:main 'some.core :aot :a
```

```
                :dev {:dependencies {:dep8 "1.6.1"
                      :plugins {:dep9 "3.1.1" :dep
```

They apparently look the same, but how can we be sure? To make our life easier, we'd like to use some automation to extract all the paths where the two data structures differ, without any nesting or actual values. `diff` can do the heavy lifting, we just need to build on top of it:

```
(require '[clojure.data :refer [diff]])

(defn walk-diff [d path] ; ❶
  (if (map? d)
    (map #(walk-diff (% d) (conj path %)) (keys d
    path))

(defn flatten-paths [paths] ; ❸
  (->> paths
       (tree-seq seq? identity) ; ❹
       (filter vector?)))

(defn diff-to-path [orig other] ; ❺
  (let [d (diff orig other)]
    (flatten-paths
      (walk-diff (first d) []))))
```

❶ `walk-diff` is a function that knows how to parse the results coming back from `diff`. We are interested in creating a path of hash-map keys in the form of a vector, like `[:a :b :c]` for each difference that was found by `diff`. To do so we need to walk `diff` results recursively and go deeper every time a new hash-map is found.

❷ If what is presented as argument `d` is a hash-map, we know there are differences and we follow through each key to find how deep they are. We call `walk-diff` recursively passing as the second argument the path found to far.

❸ `flatten-paths` helps us cleaning the final output, removing any unnecessary nesting of lists that contain a single vector path in them. This is necessary because `walk-diff` is recursively generating nested list for each map invocation.

❹ tree-seq is another great resource in the Clojure standard library. `tree-seq` transforms an arbitrarily nested sequence into a tree of which it returns a depth-first walk. We can use it here to produce a tree where the nodes are the vector paths we want to filter out.

❺ `diff-to-path` is our entry point. It takes an `orig` and `other` arguments to compare with `diff`. We take the first result in the `diff` triplet (taking the second would be the same) and pass that through `walk-diff`. As explained

before, `walk-diff` output needs cleanup from all the cluttering nested lists surrounding the paths.

Let's now have a look at how everything works together:

```
(first (diff orig new-service))             ; ❶
;; {:profiles
;;   {:dev
;;     {:dependencies
;;       {:dep8 "1.6.3"}}
;;     :uberjar {:aot "all"}}}

(diff-to-path orig new-service)             ; ❷
;; ([:profiles :dev :dependencies :dep8]
;;  [:profiles :uberjar :aot])

(get-in orig [:profiles :dev :dependencies :dep8
;; "1.6.3"
```

❶From the output of `diff` we can see that there are indeed differences. It could take sometime to find out what's different in the original output though, especially if there are many more differences.
❷ `diff-to-path` produces an alternative view at the differences. We can quickly see that there are 2 of them and where they are located in the input.
❸Here's how we can get-in on one paths to see what the different value is.

See also

`diff` is the most sophisticated option of all the functions contained in this chapter. But it might be overkill if you need a simple operator or equality predicate. All the following alternatives are lower abstraction level and more specialized:

- = is what `diff` is based on. Use `=` to test simple conditions and where deeply nested equality is not the point.
- compare offers a way to verify multiple comparable conditions at once.
- == is equality for numbers and it should be used in all those cases where numbers are primarily involved.

Other functions in the standard library could be used to "navigate" data structures if `diff` is not doing what you need:

- clojure.walk/walk can iterate over tree-like data structures. The behavior to execute when a node is found can be easily customized.
- tree-seq was seen in action in this chapter. It doesn't provide a way to execute behavior when a node is found, but it produces a depth-first walk that can be later processed.

Performance considerations and implementation details

=> `O(n)` **Worst case with "n" total number of elements**

In the worst case, `diff` performs a complete traversal of the input, visiting each node of the data structure at least once and potentially resulting in the worst case scenario O(n) (where "n" is the total number of elements any level deep). For this reason you might encounter performance penalties to compare big data structures that in average have few nodes in common. If "diffing" is the core part of your system and you also need speed, you might want to specialize your algorithms a little more. For all other general cases (like testing or occasional searching for differences) `diff` perfectly fits the purpose.

`diff` consumes the stack to perform the traversal. In the average case of a well balanced input (the input presents an average depth that is distributed across all branches) the space consumed will be roughly `log(n)`. We can verify how deep the input data structure has to be in order to completely consume the stack with the following (depending on hardware and JVM settings, the test might return different result):

```
(require 'clojure.data)

(defn generate [n]    ; ❶
  (reduce (fn [m e]
    (assoc-in m (range e) {e e}))
    {}
    (range 1 n)))

(defn blow [depth]    ; ❷
  (doseq [n (range depth 100 -50)]
    (let [a (generate n)
          b (generate (inc n))]
      (try
        (clojure.data/diff a b)
        (catch StackOverflowError soe
          (println "StackOverflow at" n "deep.")

(blow 700)
;; StackOverflow at 700 deep.
```

```
  ;; StackOverflow at 650 deep.
  ;; StackOverflow at 600 deep.
  ;; StackOverflow at 550 deep.
  ;; StackOverflow at 500 deep.
  ;; ... from here diff starts working ; ❸
  ;; correctly from the bottom of the stack
```

❶ `generate` generates a nested map n-levels deep. For example `(generate 3)` generates the hash-map `{0 {1 {2 2}}}`.

❷ `blow` repeatedly calls `diff` with gradually shallower maps (in steps of 50 levels shallower each time), waiting for a point in which the `StackOverflowError` stops appearing. The generated map `a` and `b` have a difference in their very last nested map, forcing `diff` to walk the entire structure to find out.

❸Since `diff` first walk the structure to find the first place where `a-branch` is different from the `b-branch` using only `=` single equal, the stack is the first thing that is consumed and it throws exception immediately. As soon as the data structure is small enough to stay inside the stack, the actual `diff` computation starts and it might take minutes to end.

As you can see, you need very deep data structures in order to start suffering from stack size problems (between 450 and 500

levels deep) and there is also a good probability you'll go out of heap space before reaching the end of the stack. Fortunately, real-life data structures are unlikely to be that deep and, if they are, you possibly have design problems to solve first.

[124] For this bug afflicting diff on sorted-map see clojure.atlassian.net/browse/CLJS-1709

6.8 SUMMARY

This chapter is about comparison and equality, two important aspects in the design of a programming language. =>> is the most flexible and used function, but there are also specialized versions ==>> for numbers and compare for sorting and Java interop. Hashing is related to equality when comparing keys in hash tables. Clojure has a custom hash implementation which is optimized for the heavy use of collections as keys in data structures.

In the next chapter we are moving away from describing the basics of the language to see something specific to Clojure: reducers and transducers.

7REDUCERS AND TRANSDUCERS

Reducers and transducers are consecutive refinements of the same functional abstraction. They are both based on reduce to provide improved collection processing. The focus is on parallelism in the case of reducers, while transducers mainly focus on reuse. They also bring to the table other improvements like better composability and improved performance. Transducers improve over reducers in terms of functional abstraction, but reducers offers extended parallelism. Here's a summary of their main characteristics:

- Transducers can compose over the existing library functions like map or filter. Reducers require alternative implementations of the same functions.
- Transducers and reducers both compose with comp.
- Reducers can run in parallel using fold (although this works only on vectors, maps and r/foldcat objects). Transducers can still run in parallel on top of reducers with limitations around usable transducers (only stateless transducers work reliably in parallel).

- Both transducers and reducers apply multiple composed reducing functions during a "single pass" on the input collection. This differs from standard collection processing where each operation (like `clojure.core/map` or `clojure.core/filter`) produces intermediate results.
- Both transducers and reducers are open for extension: specific collection types can define their own folding or transducing behaviors.
- Transducers can be used lazily while reducers always consume their input eagerly.

This chapter touches on these aspects and illustrate the different functions involved. Some examples showcasing extensibility of both reducers and transducers are included in call-out sections. These are:

- Creating your own foldable collection (in fold).
- Designing a parallel reducer (in reducer).
- Creating a custom transducer (in transduce).

7.1 REDUCERS

Reducers were introduced in Clojure 1.5. Reducers implementation can be found in the

`clojure.core.reducers` namespace (which needs to be required before use). Reducers contain a wrapper layer on top of the Java fork-join framework (a model for parallelism introduced in Java 1.7). Reducers also contain a set of collection processing functions that have the same name as the ones in core: `map`, `filter`, `reduce` to name a few. Compared to the ones in `clojure.core`, they create a deferred "recipe" for processing that is executed when calling reduce:

```
(require '[clojure.core.reducers :as r])      ; (

(def map-inc (r/map inc))                     ; (
(def filter-odd (r/filter odd?))              ; (
(def compose-all (comp map-inc filter-odd))   ; (
(def apply-to-input (compose-all (range 10))) ; (

(reduce + apply-to-input)                     ; (
;; 30
```

❶We need to require the related namespace before using reducers.

❷ `r/map` can be used without specifying an input collection. It returns a "reducer" function that can be further composed with other reducers.

❸Same goes for `r/filter` and the other sequence-like operations available in the namespace.

❹comp is used here to compose the two operations.

❺If we now invoke the reducer on an input collection, we obtain a reducible collection. The computation hasn't performed yet. The reducible collection provides an implementation of the `clojure.core.reducers/CollFold` protocol used by `reduce`.

❻We can finally submit the computation to `reduce`. The combination of `inc` and `odd?` is applied at once while iterating the collection.

Reducers also introduce some new vocabulary that the reader should be aware of:

- A "reducible" collection is a collection that provides `reduce` with a custom implementation. If the collection implements the `clojure.core.reducers/CollFold` protocol, `reduce` delegates the iteration to the collection itself instead of using the generic mechanism. For example, `(range 10)` is a reducible (sequential) collection. `(r/map inc (range 10))` is also reducible collection although it does not exhibit other typical properties of a collection.
- A "reducing function" is a function of 2 arguments that can be used in a `reduce` operation (for example, `+`).
- A "reducer" is a function that when invoked on a reducible collection, returns a "reducible transforming collection". For example `(r/map inc (range 10))` is a reducible transforming collection because a `reduce` operation on this collection also produces a transformation (`inc`).

The reducers namespace introduces the following reducer functions:

`map`, `reduce`, `take`, `mapcat`, `cat`, `take-while`, `remove`, `drop`, `flatten`, `filter`

Their semantic is the same as the related functions in `clojure.core`, so the rest of the chapter is mainly dedicated to those functions specific to reducers: `fold`, `reducer`, `monoid`, `folder`, `foldcat`, `cat` and `append!`. To avoid confusion with the same functions in `clojure.core`, such functions are often prefixed with `r/` (which is a conventional alias for `clojure.core.reducers`).

7.1.1 fold

`function` `since 1.5`

Listing 7.1. → *Parallel processing, Reduce-Combine, Fork-Join*

```
(fold
  ([reducef coll])
  ([combinef reducef coll] )
  ([n combinef reducef coll]))
```

In its simplest form, `fold` takes a reducing function (a function supporting at least 2 arguments) and a collection. If the input collection supports parallel folding (currently vectors, maps and foldcat objects), it splits the input collection into chunks of roughly the same size and executes the reducing function on each partition in parallel (and on multiple CPU cores when possible). It then combines the results back into a single output:

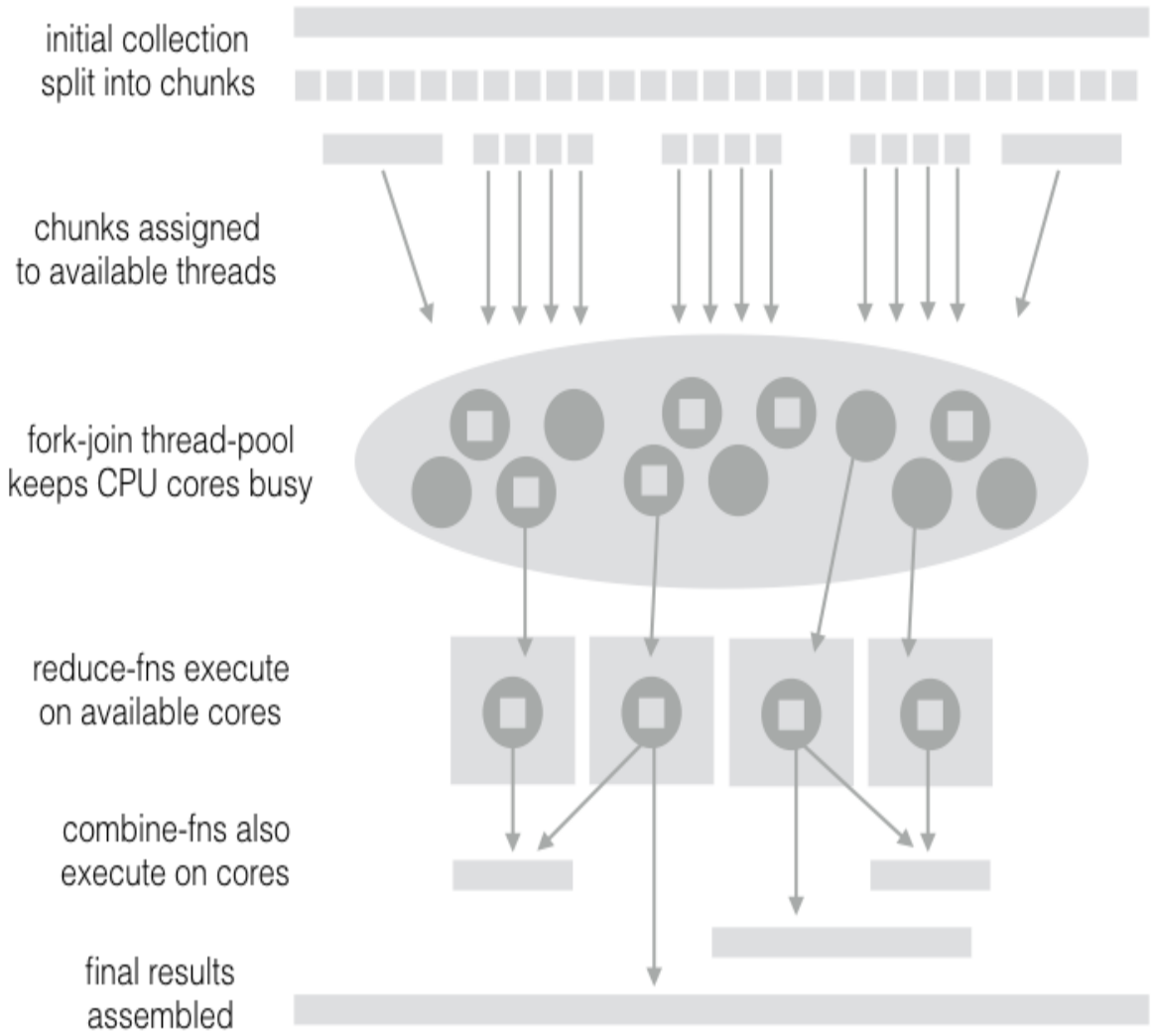
```
(require '[clojure.core.reducers :as r]) ; ❶
(r/fold + (into [] (range 1000000))) ; ❷
;; 499999500000
```

❶Reducers are bundled with Clojure, but they need to be required before use.

❷ `fold` splits the 1 million elements vector into chunks of roughly 512 each (the default). Chunks are then sent to the fork-join thread pool for parallel execution where they are reduced by `+`. The chunks are subsequently combined again with `+`.

`fold` offers parallelism based on "divide and conquer": chunks of work are created and computation happens in parallel while, at the same time, finished tasks are combined back into the final result. The following diagram illustrates the journey of a collection going through a fold operation:

Figure 7.1. How the fork-join model to reduce-combine in parallel.



An important mechanism that `fold` implements (the diagram can't show this clearly without being confusing) is work-stealing. After `fold` sends a chunk to the Java fork-join framework, each worker could further splits the work into smaller pieces, generating a mix of smaller and larger chunks. When free, a worker can "steal" work from another [125]. Work-stealing improves over basic thread-pooling, especially for less

predictable jobs keeping one or more threads unexpectedly busy.

Contract

input

The contract is different based on the presence of the optional "combinef" function and whether the input collection is a map:

- "reducef" is mandatory argument. It must be a function supporting at least a 2 arguments (and a 0 argument call when "combinef" is not provided). The 2 arguments call implements the canonical reduce contract receiving an accumulator and the current element. The 0 arguments call is used to establish the seed for the result, similarly to the "init" argument in reduce. When no "combinef" is provided, the 0-arity is invoked once for each chunk to establish the seed for the reduction. "reducef" is also used in place of "combinef" when the combination function is not provided. In this case "reducef" must be associative, as the chunks can be re-combined in any order.

- "combinef" is optional and when present it must allow 0 or 2 arguments. "combinef" needs to be associative to allow chunks to be combined in any order. The 2 arguments call is used to concatenate chunks back into the final result. When "combinef" is present, "reducef" 0-arity is never called and "combinef" is called instead.
- "n" is the approximate size of the smallest computational chunk. The default is 512.
- "coll" can be of any sequential type, empty or `nil`. If "coll" isn't a vector, hash-map or `clojure.core.reducers.Cat` object (see r/foldcat to know more), `fold` falls back on sequential `reduce` instead of going parallel. When "coll" is a `hash-map` both "reducef" and "combinef" are invoked with 3 arguments instead of 2, as per reduce-kv contract.

notable exceptions

- `IllegalArgumentException` is raised for the few unsupported collection types. This could happen for example when "coll" is a transient or a Java collection like `java.util.HashMap`. There are good reasons to exclude thread-unsafe mutable collections that would be subject to concurrency otherwise. Other thread-safe Java collections (like `java.util.concurrent.ConcurrentHashMap`) could be made "foldable" as we are going to explore in the extended example).

output

- Returns the result of invoking `(reducef)` or `(combinef)` with no arguments when "coll" is `nil` or contains just one element.
- Returns the result of applying "reducef" to the next item in the collection. Then again "reducef" applied to the previous result and the next item and so on, up to the last item in the collection. If "combinef" is present, then the partial accumulations are merged back using "combinef". The last result of applying "reducef" (or "combinef") is returned.

Examples

`fold` enables parallelism on top of the reduce-combine model. Many types of computations benefit from (or they can be adapted to) fold-like operations. reduce based data pipelines are a good candidate, like the word-count example we saw in the reduce section. In that example, we used a sequential `count-occurrences` function to count the frequency of words in a large text. We could rewrite the example to use `fold` like this:

```
(require '[clojure.core.reducers :as r])

(defn count-occurrences [coll]
  (r/fold
    (r/monoid #(merge-with + %1 %2) (constantly
    (fn [m [k cnt]] (assoc m k (+ cnt (get m k 0
    (r/map #(vector % 1) (into [] coll))))

(defn word-count [s]
  (count-occurrences (.split #"\s+" s)))

(def war-and-peace "https://tinyurl.com/wandpeac
(def book (slurp war-and-peace))

(def freqs (word-count book))
(freqs "Andrew")
;; 700
```

❶r/monoid is a helper function to create a function suitable for "combinef". The first argument for `r/monoid` is the merge function to use when to pieces are combined together. We want to sum the counts for the same word, something we can do with merge-with.

❷"reducef" needs to assoc every word to the results map "m". Two cases are possible: the word already exists and the count gets incremented or the word doesn't exist and 0 is used as the initial count.

❸"coll" needs to be a vector so we make sure the input is transformed with into. The transformation of each line includes the creation of a tuple (vector of 2 items) with the word and the number 1. We use `r/map` from the reducers library for this, so the transformation is deferred to parallel execution.

`fold` also works natively on maps. We could use `freqs` produced before as a new input for another `fold` operation. We could for example see the relationship between the first letter of a word and its frequency in the book.

The following example groups words by their initial letter and then calculates their average frequency. This operation is a good candidate for parallel `fold`, since the input contains thousands of keys (one for each word found in the input text):

```
(defn group-by-initial [freqs]
  (r/fold
    (r/monoid #(merge-with into %1 %2) (constant
    (fn [m k v]
      (let [c (Character/toLowerCase (first k))]
        (assoc m c (conj (get m c []) v))))
    freqs))

(defn update-vals [m f]
  (reduce-kv (fn [m k v] (assoc m k (f v))) {} m

(defn avg-by-initial [by-initial]
  (update-vals by-initial #(/ (reduce + 0. %) (c

(defn most-frequent-by-initial [freqs]
  (->> freqs
    group-by-initial
    avg-by-initial
    (sort-by second >)
    (take 5)))

(most-frequent-by-initial freqs)

;; ([\t 41.06891634980989]
;;  [\o 33.68537074148296]
;;  [\h 28.92705882352941]
```

```
  ;;  [\w 26.611111111111111]
  ;;  [\a 26.54355400696864])
```

❶ `group-by-initial` uses `fold` expecting a hash-map from strings to numbers. The output is a much smaller map from letters to vectors. Number of keys in this map, is equal to the number of letters in the alphabet (assuming the text is large enough and we filtered out numbers and symbols). The letter "a" in this map contains something like [700, 389, 23, 33, 44] which are the occurrences of each word in the book starting with the letter "a".

❷ The combining function is assembled using `r/monoid`. The initial value for each reducing operation is the empty map `{}`. Partial results are combined together by key merging their vector values together into a single vector.

❸ The reducing function takes three parameters: a map of partial results "m", the current key "k" and the current value "v". Similarly, to count word frequencies, we fetch a potentially existent key (using an empty vector as default value) and conj that into the vector of values "v". The key is the initial letter of each word found in the input map.

❹ `update-vals` takes a map and a function "f" of one parameter. It then applies "f" to every value in the map using reduce-kv.

❺ `avg-by-initial` replace each vector value in a map with the average of the numbers found in it.

❻ `most-frequent-by-initial` orchestrates the functions seen so far to extract the top-most frequent words by initial.

❼ `freqs` is the result of the word count from the previously in the example.

After running `most-frequent-by-initial` we can see that the letter "t" is on average the most used at the beginning of a word, closely followed by "o", "h", "w" and "a". This indicates that words starting with the letter "t" are on average the most repeated throughout the book (while some other word not starting with "t" might be, on absolute, the most frequent).

Creating your own `fold`

`fold` is a protocol-based extensible mechanism. Most of the Clojure collections provide a basic sequential folding mechanism based on `reduce` with the exception of vectors, maps and foldcat objects that are equipped with a parallel reduce-and-combine algorithm. Classes like `java.util.HashMap` don't have a proper `fold` and there are good reasons connected to the danger of exposing a mutable data structure to potentially parallel threads. Other thread-safe classes like `java.util.concurrent.ConcurrentHashMap` could be foldable, which is the subject of this section. What we are going to see for `java.util.concurrent.ConcurrentHashMap` can be easily extended to other collections (provided they support concurrent access).

To drive our example, let's use a large `ConcurrentHashMap` of integers (keys) into integers (values) and some expensive function to apply to all the keys. A trivial transformation on values like `inc` or `str` would be probably overkill for `fold` parallelism so we we are going to use the Leibniz formula to approximate "Pi"

instead (we already encountered this formula while talking about filterv). We would like to execute the transformation on each key in parallel.

The design of the parallel execution is as follows: instead of splitting the values into chunks, we are going to split the keys. Values corresponding to each partitioning are transformed in parallel by separate threads. No clashing would normally happen (as keys are unique), but fork-join is a work stealing algorithm so a partition could be routed to a thread where another partition has been assigned, generating an overlap. This is the reason why we need `java.util.concurrent.ConcurrentHashMap` instead of a plain `java.util.HashMap`.

```
(import 'java.util.concurrent.ConcurrentHashMap)
(require '[clojure.core.reducers :as r])

(defn pi [n]                          ; ❶
  "Pi Leibniz formula approx."
  (->> (range)
       (filter odd?)
       (take n)
       (map / (cycle [1 -1]))
       (reduce +)
       (* 4.0)))

(defn large-map [i j]                 ; ❷
  (into {}
    (map vector (range i) (repeat j))))

(defn combinef [init]                 ; ❸
  (fn
    ([] init)
    ([m _] m)))

(defn reducef [^java.util.Map m k]  ; ❹
  (doto m
    (.put k (pi (.get m k)))))
```

```
(def a-large-map (ConcurrentHashMap. (large-map 100000 100)))

(dorun                                  ; ❺
  (r/fold
    (combinef a-large-map)
    reducef
    a-large-map))
;; IllegalArgumentException No implementation of method: :kv-redu
```

❶ `pi` calculates an approximation of the π value. The greater the number "n" the better the approximation. Relatively small numbers in the order of the hundreds generate an expensive computation.

❷ `large-map` serves the purpose of creating a large `ConcurrentHashMap` to be used in our example. The map keys are increasing integers while the values is always the same.

❸ `combinef` with no arguments returns the base map, the one all threads should update concurrently. There is no need of concatenation as the updates happen on the same mutable `ConcurrentHashMap` instance. So `combinef` with two arguments just returns one of the two (they are the same object). `combinef` could be effectively replaced by `(constantly m)`.

❹ `reducef` replaces an existing key with the calculated "pi". Note the use of doto so Java operations like `.put` which would otherwise return `nil` return the map itself.

❺ `fold` is unsuccessful, as it searches for a suitable implementation of `reduce-kv` which is not found.

We are facing the first problem: `fold` fails because two polymorphic dispatches are missing: `fold` doesn't have a specific parallel version for `java.util.concurrent.ConcurrentHashMap`, so it routes the call to `reduce-kv`. `reduce-kv` also fails because there is an implementation for Clojure `hash-map` but not for Java

`ConcurrentHashMap`. As a first step, we could provide a `reduce-kv` version which removes the error, but this solution is not enough to run the transformations in parallel:

```
(extend-protocol                    ; ❶
  clojure.core.protocols/IKVReduce
  java.util.concurrent.ConcurrentHashMap
  (kv-reduce [m f _]
    (reduce (fn [amap [k v]] (f amap k)) m m)))

(time                               ; ❷
  (dorun
    (r/fold
      (combinef a-large-map)
      reducef
      a-large-map)))
;; "Elapsed time: 41113.49182 msecs"

(.get a-large-map 8190)             ; ❸
;; 3.131592903558553
```

❶ We can add a type to a protocol by using extend-protocol. Our `reduce-kv` doesn't need the value because we are mutating the Java `ConcurrentHashMap` in place.

❷ `fold` now run correctly. We need dorun to prevent the map to be printed on screen. We also printed a reasonably good estimate of the time elapsed for the operation to finish, which is above 40 secs.

❸ To be sure that `a-large-map` has effectively been updated, we check the random key "8190". It contains an approximation of "pi", as expected.

Although we provided a suitable `reduce-kv` implementation, `java.util.concurrent.ConcurrentHashMap` doesn't have a proper parallel `fold` yet. Similarly to `reduce-kv`, we need to provide a `fold` implementation by extending the correct protocol. The idea is to split the key set instead of the map and have each thread operate in parallel to process the given subset:

```
(defn foldmap [m n combinef reducef]      ; ❶
  (#'r/foldvec
    (into [] (keys m))
    n
    combinef
    reducef))
```

```
(extend-protocol r/CollFold                ; ❷
  java.util.concurrent.ConcurrentHashMap
  (coll-fold
    [m n combinef reducef]
    (foldmap m n combinef reducef)))

(def a-large-map (ConcurrentHashMap. (large-map

(time                                      ; ❸
  (dorun
    (into {}
      (r/fold
        (combinef a-large-map)
        reducef
        a-large-map))))
"Elapsed time: 430.96208 msecs"
```

❶ `foldmap` implements the parallel strategy for `java.util.concurrent.ConcurrentHashMap`. It delegates to `foldvec` in `reducers` namespace with the keys coming from the map, effectively reusing vectors parallelism.

❷ We instruct `CollFold` protocol to use `foldmap` when `fold` is presented with a `java.util.concurrent.HashMap` instance.

❸ After recreating the large map (remember how it is mutated after each execution) we try `fold` again resulting in the expected performance boost (from over 40 seconds for the sequential case down to 430 millisecond). We also take care of transforming the `ConcurrentHashMap` returned by `fold` back into a persistent data structure for later use.

After extending `CollFold` protocol from the `clojure.core.reducers` namespace, we can see that `fold` effectively runs the update of the map in parallel, cutting the execution time consistently. As a comparison, this is the same operation performed on a persistent `hash-map` which is parallel by default:

```
(def a-large-map (large-map 100000 100))

(time
  (dorun
```

```
      (r/fold
        (r/monoid merge (constantly {}))
        (fn [m k v] (assoc m k (pi v)))
        a-large-map)))
 ;; "Elapsed time: 17977.183154 msecs" ; ❶
```

❶We can see that despite Clojure `hash-map` is parallel enabled, the fact that it is a persistent data structure is playing against fast concurrent updates. This is not a weakness in Clojure data structure as they are designed with a completely different goal in mind.

See also

- pmap also applies a transformation function to an input sequence in parallel. `fold` and `pmap` have commonalities but they differ in the computational model. `pmap` supports laziness and has a variable number of workers (dependent on the collection chunk size plus the number of available cores plus 2). However, before moving on to the next chunk in the sequence, `pmap` has to wait for all workers in the current chunk to finish. Less predictable operations (those keeping a worker busy more than usual), effectively prevent `pmap` full concurrency. `fold`, on the other hand, allows a free worker to help a busy one dealing with a longer than expected request. As a rule of thumb, prefer `pmap` to enable lazy processing on predictable tasks, but use `fold` in less predictable scenarios where laziness is less important.

Performance considerations and implementation details

=> `O(n)` **linear**

`fold` is implemented to recursively split a collection into chunks and send them to the fork-join framework, effectively building a tree in `O(log n)` passes. However, each chunk is subject to a linear reduce that dominates the logarithmic

traversal: the bigger the initial collection, the more are the calls to the reducing function, making it a linear behavior overall.

Orchestration of parallel threads has a cost that should be taken into account when executing operations in parallel: like pmap, `fold` performs optimally for non-trivial transformations on potentially large dataset. The following simple operation for example, results in a performance degradation when executing in parallel:

```
(require '[criterium.core :refer [quick-bench]])
(require '[clojure.core.reducers :as r])

(let [not-so-big-data (into [] (range 1000))]
  (quick-bench (reduce + not-so-big-data)))
;; Execution time mean : 11.481952 µs

(let [not-so-big-data (into [] (range 1000))]
  (quick-bench (r/fold + not-so-big-data)))
;; Execution time mean : 32.683242 µs
```

As the collection gets bigger, the computation more complicated and the available cores increase, `fold` starts to outperform a similar sequential operation. The potential performance boost

is still not enough to grant the need for a `fold`, since other variables come into place such as memory requirements.

`fold` is designed to be an eager operation, as the chunks of input are further segmented by each worker to allow an effective work-steal algorithm. `fold` operations like the examples in this chapter need to load the entire dataset in memory before starting execution (or as part of the execution). When `fold` produces results which are substantially smaller than the input, there are ways to prevent the entire dataset to load in memory, for example by indexing it on disk (or a database) and include in the reducing function the necessary IO to load the data. This approach is used for example in the Iota library [126] which scans large files to index their rows and use that as the input collection for `fold`.

7.1.2 reducer and folder

`function` `since 1.5`

Listing 7.2. → *Transforming Functions, Reducing Functions, Collection Reduce, Folding*

```
  (reducer [coll xf])
  (folder [coll xf])
```

Both `reducer` and `folder` takes a collection and a function of one argument. They enhance their input collection with a custom reduce implementation (and additionally fold in the case of `folder`), as directed by the "xf" argument. Here's an example of a collection enhanced by `reducer` and one by `folder`:

```
(require '[clojure.core.reducers :as r])    ; ❶

(defn divisible-by-10 [current-reducing-fn] ; ❷
  (fn [acc el]
    (if (zero? (mod el 10))
      (current-reducing-fn acc el)
      acc)))

(into []                                    ; ❸
  (r/reducer
    (range 100)
    divisible-by-10))
;; [0 10 20 30 40 50 60 70 80 90]

(r/fold                                     ; ❹
  (r/monoid merge (constantly {}))
  (fn [m k v] (assoc m k (+ 3 v)))
  (r/folder
    (zipmap (range 100) (range 100))
```

```
      (fn [rf] (fn [m k v] (if (zero? (mod k 10))
  ;; {0 3, 70 73, 20 23, 60 63, 50 53, 40 43, 90 9
```

❶Both `reducer` and `folder` lives inside the `reducers` namespace. You need to require the namespace before use.

❷ `divisible-by-10` is an example of transformation on a reducing function. `reducer` transforms the input collection using `divisible-by-10` as the new reducing behavior. `divisible-by-10` verify if the current element is divisible by 10 and applies the current reducing function only in that case.

❸into is used here to show how the collection is now transformed. `into` is implemented on top of `reduce` hence why the transformation takes place. `divisible-by-10` has the same effect of filtering the input collection.

❹ `folder` works similarly for fold. This is demonstrated using a hash-map as input. `folder` is instrumenting the reducing function so only keys which are multiple of 10 are passing through. Another reducing function is present to increment every value by 3. The two are eventually concatenated together.

The input function "xf" in both `reducer` and `folder` gets a chance to intercept the current call to the original reducing

function and potentially alter the results. `reducer` and `folder` are useful in the definition of custom reducers (and they are used extensively inside the implementation of reducers themselves). The following table is a summary of the available reducers and their foldable behavior:

Table 7.1. Available reducers in the reducers namespace

Name	Description	Foldable?
`r/map`	Applies f to all elements in coll.	Yes.
`r/mapcat`	Applies f to all elements and concatenate the results.	Yes.
`r/filter`	Returns elements from coll where the predicate is `true`.	Yes.
`r/remove`	Removes elements from coll where the predicate is `true`.	Yes.
`r/flatten`	Eliminates collection nesting.	Yes.
`r/cat`	Not technically a reducer, but a combining function. It produces a `foldable`.	Yes.

`r/take-while`	Stops reduction when predicate is `true`.	No.
`r/take`	Stops reduction after consuming n values.	No.
`r/drop`	Removes the first n elements from coll.	No.

The table shows us that apart from `r/take-while`, `r/take` and `r/drop` all other standard reducers are foldable. The practical effects are that if you are using any of the three non-foldable reducers, you could prevent parallelism during a `fold`. Please refer to the call-out section after the examples to see how non-foldable reducers can also be enabled in parallel contexts.

Contract

input

- "coll" is any collection supported by seq, which excludes transients and deprecated structs.

- "xf" is a function of 1 argument returning a function of 2 arguments. "x" stands for "transforming" while "r" means "reducing". `reducer` invokes "xf" in the context of a `reduce` call, passing the original reducing function. "xf" returns a function of 2 arguments as per reduce contract. `reducer` replaces the original reducing function with the new reducing function returned by "xf".

output

- `reducer` returns "coll" enhanced with additional behavior as dictated by the "xf" argument that will apply in the context of a `reduce` call.
- `folder` applies the same changes as `reducer`, but also includes enhancing the input collection in the context of a `fold` operation. `folder` only intercepts "reducef" behavior, not "combinef" (please see fold contract).

Examples

dedupe is a function in the standard library that removes consecutive occurrences of the same item in a collection. We want to create a version of `reducer-dedupe` which plays

nicely with the other transforming functions in the `clojure.core.reducers` namespace. To do so, we are going to use `reducer` to wrap the given collection with some additional behavior:

```
(require '[clojure.core.reducers :as r])

(defn reducer-dedupe [coll]
  (r/reducer coll                       ; ❶
    (fn [rf]
      (let [prev (volatile! ::none)]    ; ❷
        (fn [acc el]
          (let [v @prev]
            (vreset! prev el)
            (if (= v el)                ; ❸
              acc
              (rf acc el)))))))))

(->> (range 10)                         ; ❹
     (r/map range)
     (r/mapcat conj)
     (r/filter odd?)
     reducer-dedupe
     (into []))

;; [1 3 1 3 1 3 5 1 3 5 1 3 5 7 1 3 5 7]
```

❶ `reducer` wraps the incoming collection and indicates how the reduction process should be altered passing a function of one argument "rf" (the original reducing function, for example conj is what into is using).

❷ Our `reducer-dedupe` needs to remember the previous element at each invocation of the reducing function. We need to store state and since the state is local to the function that wraps over the reduction we can use volatile! (an atom would also work, but it would introduce additional complexity regarding thread isolation in a concurrent context).

❸ Each reducer in the chain, including `reducer-dedupe`, decides what to do with the next transformations. In the case of `reducer-dedupe`, the next transformation happens only if the current element is not a duplicate.

❹ We can now use `reducer-dedupe` similarly to the other reducer functions.

In the previous example, `reducer` was used to enhance the input collection. `reducer` does not provide a `fold` implementation, so our `reducer-dedupe` would prevent parallelism (without any warning) for vectors or maps when they are the input of a `fold` operation:

```
(->> (range 1500)
     (into [])
```

```
      (r/map
        #(do (println      ; ❶
           (str (Thread/currentThread))) %))
      (r/map range)
      (r/mapcat conj)
      (r/filter odd?)
      reducer-dedupe
      (r/fold +))
;; Thread[main,5,main]    ; ❷
;; Thread[main,5,main]
;; ....
;; 280338192
```

❶ `r/map` function prints the thread signature for each element in the collection, thus showing on which thread the reduction is happening.

❷The prints on the screen are coming from the main thread, confirming that `reducer-dedupe` is preventing parallel fold.

Before attempting to implement a parallel `reducer-dedupe`, the reader must be aware that not all reducers are amenable for straightforward parallelization. There are several reasons to consider when designing a parallel reducer (or transducer, as they are based on the same design):

- **Associativity**. The sequential semantic of a reducer might not be directly applicable in a parallel context. Dropping 10 items from a collection, for instance, is not the same as dropping 5 elements from the head of the collection and 5 elements from the tail. Alternative semantics can be defined, for example deciding to drop "n" element for each chunk, but this results in radically different results from the same operation executed sequentially. Similar considerations apply to `r/take` or `r/take-while`.
- **Thread-safe stateful reducers**. State inside a reducer needs to be thread-aware when running in parallel. When the reducer runs as part of a parallel `fold` instruction, the state is subject to concurrent access by multiple fork-join threads.

Our `reducer-dedupe` reducer, is not amenable for parallelization without designing it specifically for concurrent access. Although without warnings, `fold` refuses to enable parallelism unless all involved reducers are parallel-enabled. The same behavior does not repeat with transducers (which are based on a different implementation technique not involving type extension), hence `r/fold` would allow parallel folding of stateful transducers with potentially inconsistent results. Please refer to the extended example below for more information.

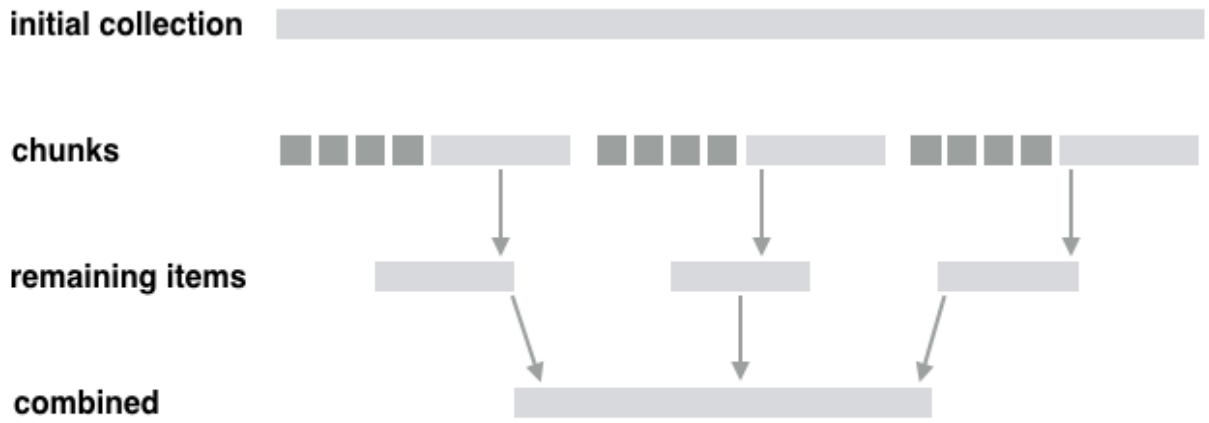
Note

Reducers like `reducer-dedupe` implemented in our example are also called "stateful", because they need to propagate information between invocations of the reducing function (the reducers namespace already contains `r/take` and `r/drop` stateful reducers). Stateful reducers typically define a local variable of type volatile! or an atom.

Designing a parallel reducer

This is an extended example section about reducers and parallelism. Our goal is to implement a version of `r/drop` that propagates parallelism when used in a `fold`. The new `pdrop` reducer (for parallel-drop), is designed to remove the same number of items from each chunk in parallel. This behavior is radically different from the definition of drop from the standard library: results are different based on the chunk size, not just the input collection (the more the chunks, the more the times the same amount of "n" elements gets removed). After deciding that this is a desirable behavior, the following diagram shows how a parallel `drop` would work:

Figure 7.2. Parallel drop of elements from each chunk.



The darker squares in the diagram are the target for removal from the head of each partitioned collection. We could implement the same behavior sequentially with the following:

```
(->> (vec (range 1600))     ; ❶
     (partition 200)
     (mapcat #(drop 10 %))
     (reduce +))
;; 1222840
```

❶A collection of 1600 numbers is split into 8 partitions of 200 items each. Standard drop is used to remove the first 10 items on each partition. The numbers are finally summed up together.

Our first attempt naturally follows the idea to to use `folder` instead of `reducer` to wrap around the `drop` operation. That enables parallelism straight away, but it comes with surprising result:

```
(require '[clojure.core.reducers :as r])
(import 'java.util.concurrent.atomic.AtomicInteg
(set! *warn-on-reflection* true)

(defn pdrop  ; ❶
  [n coll]
  (r/folder
    coll (fn [rf]
      (let [nv (volatile! n)]
        (fn
          ([result input]
           (let [n @nv]
             (vswap! nv dec)
             (if (pos? n)
               result
               (rf result input)))))))))

(distinct    ; ❷
  (for [i (range 1000)]
    (->> (vec (range 1600))
         (pdrop 10)
         (r/fold 200 + +))))
;; (1279155 1271155 1277571 ...)
```

❶ `pdrop` is a custom reducer using `r/folder` to define a specific foldable behavior. When executed, `pdrop` doesn't propagate reduction for the first "n" elements, effectively ignoring them in the final result.

❷ We try now to `fold` over the 1600 numbers defining a 200 chunk size. To show the inconsistency, a for repeats the same operation 1000 times. As we can see, the result is not only different from the expected 1222840, but also changes randomly.

Parallel enabled `pdrop` is returning inconsistent results. The reason for this is how the volatile! closes over the reducing function. Each fork-join task is created around standard reduce, which in turn is using our enhanced reducing function. The problem is that each `reduce` task sees the same volatile "nv" and each task could run on a separate thread. Depending on which thread reads "nv" and when, we are dropping less or more items than expected. Even assuming we could use isolated counters on each thread, work-stealing could migrate a chunk to another thread with a different counter condition.

One solution is to initialize state at every `reduce` invocation, instead of reducer creation. To do this, we need to create our own folding algorithm (a very similar version to the one found in the standard library) and a revised version of `pdrop` which

suspends the creation of the state until the execution point inside the fork-join task. The only change necessary to the current `r/foldvec` reduce-combine algorithm is to "unwrap" the additional function created around "reducef" before use:

```
(defn stateful-foldvec
  [v n combinef reducef]
  (cond
    (empty? v) (combinef)
    (<= (count v) n) (reduce (reducef) (combinef
    :else
    (let [split (quot (count v) 2)
          v1 (subvec v 0 split)
          v2 (subvec v split (count v))
          fc (fn [child] #(stateful-foldvec chil
      (#'r/fjinvoke
        #(let [f1 (fc v1)
               t2 (#'r/fjtask (fc v2))]
           (#'r/fjfork t2)
           (combinef (f1) (#'r/fjjoin t2)))))))))

(defn pdrop
  [dropn coll]
  (reify                          ; ❷
    r/CollFold
    (coll-fold [this n combinef reducef]
      (stateful-foldvec coll n combinef
```

```
          (fn []                        ; ❸
            (let [nv (volatile! dropn)]
              (fn
                [result input]
                (let [n @nv]
                  (vswap! nv dec)
                  (if (pos? n)
                    result
                    (reducef result input))))))))

  (distinct                             ; ❹
    (for [i (range 1000)]
      (->> (vec (range 1600))
           (pdrop 10)
           (r/fold 200 + +))))

  ;; (1222840)
```

❶ `stateful-foldvec` is copied from the private function `foldvec` inside the reducers namespace. There is only one small change when `reduce` is invoked on a chunk, where "reducef" is wrapped in round parenthesis to initialize the reducing function and remove the additional wrapping function.

❷ `pdrop` implements its own reify for the `CollFold` protocol. `prdop` instructs `fold` to call the new `stateful-foldvec`. The transformation of the reducing function happens when passing the last argument to `stateful-foldvec`.

❸ The reducing function is wrapped in a "thunk" (a lambda of no arguments with the only goal of delaying evaluation). The thunk is unwrapped at execution time by `stateful-foldvec`.

❹ We can see now the expected single result.

The idea for stateful reducer parallelization exposed above, can be extended to transducers by wrapping state initialization in a similar way. The need to also change the standard `r/foldvec` function from reducers remains, but we don't need to include that in the new transducer implementation:

```
(defn drop-xform [n]
 (fn [rf]
  (fn []                          ; ❶
```

```
    (let [nv (volatile! n)]
     (fn
      ([] (rf))
      ([result] (rf result))
      ([result input]
       (let [n @nv]
        (vswap! nv dec)
        (if (pos? n)
         result
         (rf result input)))))))))

(defn stateful-folder [coll]  ; ❷
  (reify r/CollFold
    (coll-fold [this n combinef reducef]
      (stateful-foldvec coll n combinef reducef)))

(distinct                     ; ❸
 (for [i (range 1000)]
  (r/fold 200
   +
   ((drop 10) +)
   (vec (range 1600)))))
;; (1279155 1271155 1267155 1275155 1275145 ...

(distinct                     ; ❹
 (for [i (range 1000)]
  (r/fold 200
   +
```

```
    ((drop-xform 10) +)
    (stateful-folder (vec (range 1600))))))
;; (1222840)
```

❶The new version of drop transducer is the same as the one in the standard library, except for the layer of indirection introduced just before the state is initialized. This is just a lambda function with no arguments that prevents evaluation of the volatile! instance.

❷To prevent `r/fold` from using standard vectors parallelization, we wrap the vector instance with a reify call that swaps the base implementation for our new `stateful-foldvec`.

❸We want to compare the differences between using `drop` transducer without any modification and our version. As you can see here, `drop` transducer used in a parallel context shows multiple inconsistent results.

❹By contrast, the parallel-enhanced `drop-xform` shows the expected result consistently.

See also

- reify is the main mechanism used by `reducer` and `folder`. Use `reify` on the target collection in case you need additional control over the implementation of the `clojure.core.protocols/CollReduce` or `clojure.core.reducers/CollFold` protocol.

Performance considerations and implementation details

=> `O(1)` **constant time**

`reducer` and `folder` are implemented in term of reify. This is a one-off transformation of the input collection that doesn't change with the input size and therefore is constant time.

As discussed throughout the chapter, `reducer` and `folder` produce similar results, with the addition that `folder` also provides enhancement over `fold` operations. `r/take-while`, `r/take` and `r/drop` do not use `folder`, thus preventing parallelism in certain conditions, for example:

```
(time (->> (range 50000)
           (into [])
           (r/map range)
           (r/mapcat conj)
           (r/drop 0)                    ; ❶
           (r/filter odd?)
```

```
        (r/fold +)))
  ;; "Elapsed time: 45516.963356 msecs"
  ;; 10416041675000

  (time (->> (range 50000)
        (into [])
        (r/map range)
        (r/mapcat conj)
        (r/filter odd?)
        (r/fold +)))
  ;; "Elapsed time: 9190.562896 msecs" ; ❷
  ;; 10416041675000
```

❶ `r/drop` is used in this reducer chain without actually dropping any element, just to show the effects on computation. The elapsed is around 45 seconds.

❷The same operation (except for `r/drop`) now executes in parallel, greatly reducing execution time to below 10 seconds.

`fold` operation using `r/take-while`, `r/take`, `r/drop` or any custom reducer using `reducer` to `reify` the input

collection, would appear to work normally but the reader should be aware that there is no parallelism in that case.

7.1.3 monoid

`function` `since 1.5`

Listing 7.3. → *Arity Handling, Function Completion, Initial Fold*

```
  (monoid [op ctor])
```

`monoid` is a higher order function in the `clojure.core.reducers` namespace. It wraps a function of two arguments so it can additionally provide a result when called with no arguments. Such case is typical for the initial value of a reducing function in a fold context:

```
  (require '[clojure.core.reducers :as r])      ; (

  (r/fold
    (r/monoid str (constantly "Concatenate "))  ; (
    ["th" "is " "str" "ing"])
  ;; "Concatenate this string"
```

❶We need to require the `reducers` namespace.
❷`monoid` is used here as reducing function for `r/fold`. Differently from normal reduce, `r/fold` calls the zero-arguments arity of the reducing function if no initial value is provided.

Normal reduce has an additional parameter to pass an initial value for reduction. fold, on the other hand, doesn't offer this option (having many parameters already). If the reducing function for a `fold` doesn't provide the zero-argument call, `monoid` offers a quick way to fix the problem without the need to use an anonymous function.

Contract

input

- "op" must be a function accepting two arguments and is required argument.
- "ctor" must be a function accepting a zero arguments call and is required argument.

notable exceptions

- `ArityException` or `ClassCastException` are typically seen when "ctor" is given as a value (e.g. a number, or empty vector) instead of a function of no arguments:

```
(r/fold (r/monoid + 0) (range 10))
;; ClassCastException java.lang.Long cannot be c
```

To prevent this from happening, remember to use something like constantly to wrap around the constant value.

output

Returns a function of 2 arities that can be called with zero or two arguments. When invoked with no arguments, it returns the result of invoking `(ctor)`. When two arguments are present, they are passed to "op" in order. This is equivalent to `(op a b)` if "a" and "b" are the arguments.

Examples

`monoid` is mainly used to build the "reducef" or "combinef" argument for fold. When processing a hash-map, for instance, fold often requires an empty hash-map to start with:

```
(r/fold (r/monoid merge (constantly {}))  ; ❶
        (fn [m k v] (assoc m k (str v)))
        (zipmap (range 10) (range 10)))

;; {0 "0", 7 "7", 1 "1", 4 "4",
    6 "6", 3 "3", 2 "2", 9 "9",
    5 "5", 8 "8"}
```

❶ `monoid` is used here to create a function to combine partial results with merge. Each `reduce` operation that fold performs on a chunk, is going to use the zero-argument arity provided by `monoid` (the empty map).

Please also check out fold for another example of `monoid` used to calculate the word frequencies for some long text.

Monoid: what about the name?

Monoids are a well known concept in algebra [127]. They describe the properties of a set in the presence of a binary-associative operation and an identity element. Natural numbers with "+" and "0" as identity element are a monoid. To prove this, we should demonstrate that rules for monoids hold true:

- **Identity element**: there is an unique element in the set that, when used in the binary operator, it returns the other argument untouched.
- **Associativity**: applications of the binary operation to different subsets are order independent.

Let’s verify that natural numbers with "+" (binary operator) and "0" (identity element) are indeed a monoid:

```
(+ 99 0)              ; ❶
;; 99

(= (+ (+ 1 2) 3)     ; ❷
   (+ 1 (+ 2 3)))
;; true
```

❶When "0" identity element is used, the other argument is returned without changes. This is tested with one random number, but it holds true for all other natural numbers by definition of zero addition.

❷We have here a subset of the natural numbers formed by "1,2,3". The application of "+" to "1,2" and "3" is the same as "2,3" first and "1" next.

`r/monoid` has been named this way considering its applicative context. The zero-arguments arity of the reducing function in fold provides a way to bootstrap reduce with an initial element. The initial value can optionally be the identity element for the reducing function (like "0" for "+"), thus providing an identity transformation for the first element.

`r/monoid` name reminds standard library users that the binary operation in `fold` should be associative (especially when the reducing function is also used as concatenation), as it can be potentially parallel and execute in any order.

See also

- completing has a similar goal to `monoid`. `completing` is used with custom transducers to provide additional arities to the reducing function. The single arity call in transducers is used to signal the end of the reduction. `completing` provides a quick way to create all the required arities around the main reducing function.
- fold as explained throughout the chapter, is the main use case for `monoid` to create the combine function.

Performance considerations and implementation details

=> `O(1)` **constant time**

`monoid` implementation is very simple and it just invokes its argument in a new function. There are no specific performance considerations.

7.1.4 foldcat, cat and append!

`function` `since 1.5`

Listing 7.4. → *Concatenation, Mutable Accumulator*

```
(foldcat [coll])
(cat ([])
     ([ctor])
     ([left right]))
(append! [acc el])
```

`r/foldcat`, `r/cat` and `r/append!` are related function in the `clojure.core.reducers` namespace (please note that `cat` here is different from clojure.core/cat the transducer, hence it's prefixed with the initial of the reducers namespace). They can be used independently, but they are designed with a specific optimization in mind: avoid frequent updates of temporary persistent data structures.

Immutable and persistent vectors or maps are typically created by functions like conj or assoc. Each chunk during a `fold`

operation could produce one of such persistent data structures. Once produced, the same data structures are combined, again producing ephemeral objects that are never consumed after the `fold` operation returns (see the diagram of the combine-reduce model in fold to know more).

To solve the issue, `r/append!` allows incremental build of mutable data structures while `r/cat` appends them to a tree. The orchestration of both effects is achieved by `r/foldcat` which just uses `r/cat` as "combinef" and `r/append!` as "reducef":

```
(require '[clojure.core.reducers :as r])
(def input (r/map inc (into [] (range 1000))))

(take 5 (r/fold r/cat r/append! input)) ; ❶
;; (1 2 3 4 5)

(take 5 (r/foldcat input))              ; ❷
;; (1 2 3 4 5)
```

❶This example shows using `r/cat` and `r/append!` explicitly with fold.

❷The second example is equivalent to the first, showing the use of `r/foldcat` to achieve the same result.

`r/foldcat` returns the root of the chunks produced by `fold` as a network of `Cat` objects (note the uppercase "c" denoting a class name instead of the function). `Cat` nodes are "counted" (they support the `clojure.lang.Counted` interface and can be counted in constant time), reducible and foldable, so they can be used efficiently as further input of a `reduce` or a `fold`.

Contract

input

`r/foldcat`

- "coll" as per fold contract.

`r/cat`

- Without arguments, it produces an empty `java.util.ArrayList` instance. When `r/cat` is used as "combinef" argument of a `fold` operation, the empty `ArrayList` is used as initial element for "reducef" reducing function.
- "ctor" overrides the choice of the empty argument call. "ctor" must be a function of no arguments.

- "left" and "right" can be of any type supporting count which includes `nil`. When `r/cat` is used as "combinef" in a `fold`, they can be the result of the reducing function (leaves), or the result of another `r/cat` operation (nodes).

`r/append!`

- "acc" can be any sub-type of `java.util.Collection`. The Java method `.add` is used internally by `r/append!`. When used as a reducing function, "acc" represents the accumulator.
- "el" can be any type. `r/append!` calls `.add` using "el" as the element to add to the accumulator.

output

- `r/foldcat` behaves similarly to `fold`, but the type of the result is either: a `java.util.ArrayList` when the size of the input is below the size of the requested chunk (512 by default) or a `clojure.core.reducers.Cat` object for larger collections. The `Cat` type represent the node of a binary tree with left and right children. The more chunks processed by `fold`, the deeper the tree.

- `r/cat` with no arguments returns an empty `ArrayList`. With one argument returns a new function which overrides the no argument behavior using the result of calling "ctor" without arguments. With two not-null arguments, it returns a new `clojure.core.reducers.Cat` object with count equal to the sum of the counts of the "left" and "right" objects.
- `r/append!` returns the result of calling the `java.util.Collection/add` method on "acc" using "el" as the argument.

Examples

`r/foldcat` uses `r/cat` and `r/append!` internally to generate results. The following example shows how we could process words from a large text using reducers and `r/foldcat`:

```
(require '[clojure.core.reducers :as r])
(require '[clojure.string :as s])

(def text                                ; ❶
  (-> "https://tinyurl.com/wandpeace"
      slurp
      s/split-lines))
```

```
(def r-word                                  ; ❷
  (comp
    (r/map #(vector % (count %)))
    (r/map s/lower-case)
    (r/remove s/blank?)
    (r/map #(re-find #"\w+" %))
    (r/mapcat #(s/split % #"\s+"))))

(def words (r/foldcat (r-word text)))   ; ❸

(take 5 words)
;; (["the" 3] ["project" 7] ["gutenberg" 9] ["eb
```

❶ We fetch "War and Peace", a large book, from Project Gutenberg (a collection of literature classics). We need to split the file into lines to create the initial vector for `r/foldcat`.

❷ `r-word` is a composition of reducers dedicated to text processing and cleanup. They are applied bottom-up: lines are split into words, non-alphabetic characters and empty strings are removed, words are converted lower-case and finally a pair is formed for each word with its length.

❸ `r/foldcat` accepts the collection of lines wrapped in the reducers call.

If we inspect the previous results we can see that "words" is not a normal collection (it would have been an `ArrayList` for a file with less than 512 words):

```
(type words)    ; ❶
;; clojure.core.reducers.Cat
(.count words)  ; ❷
;; 565985
(.left words)   ; ❸
;; #object[clojure.core.reducers.Cat 0x28e8dde3
(.right words)  ; ❹
;; #object[clojure.core.reducers.Cat 0x1f6c9cd8
```

❶ `r/foldcat` returned an instance of `clojure.core.reducers.Cat`, a deftype definition inside the reducers namespace. This represents the root of a binary tree.

❷ `clojure.core.reducers.Cat` objects have 3 fields: "count", "left" and "right". "count" is the number of words after processing.

❸When we invoke the Java method "left" we can see a similar object down the left branch. Please note that "left" is not a Clojure function and the implementation of the internals of `Cat` objects could change in future versions of Clojure. Don't use in production code!

❹Similarly when we invoke "right" we can walk the right branch of the root node.

If we walk the tree all the way down to a leaf, we can find a `java.util.ArrayList` instance created by invoking `r/append!` on each chunk:

```
(loop [root words cnt 0]
  (if (< (count root) 512) ; ❶
    (str (type root) " " (count root) " elems, d
    (recur (.left root) (inc cnt))))
```

```
;; "class java.util.ArrayList 321 elems, depth:8
```

❶512 is the default size of a chunk of computation in a fold operation. When the chunk size is below that threshold, we know we are in front of a leaf in the tree.

❷In this case we also know that the `ArrayList` that was found at that leaf, has 321 words in it.

By knowing the depth of the binary tree, we also know the approximate number of created nodes, corresponding to how many chunks of work were necessary to process the initial vector of lines. The number of nodes at the lowest level in a binary tree is 2^k, where k is the last level of the tree (counting from 0). In our example approximately $2^8 = 256$ splits were created.

After looking at the words in the result, it's easy to see that there are many duplicates:

```
(count (distinct (seq words))) ; ❶
;; 17200
```

❶Note the use of seq on the `Cat` tree instance returned by `r/foldcat`. A `Cat` object supports count but doesn't support nth (and many other sequential operations).

There are 17200 distinct words out of 565985, showing that the big majority of the words returned is a duplicate. We can use the property of `java.util.HashSet` to get rid of the duplicates. `r/cat` has a single argument call that allows to swap the internal implementation of the mutable data structure, as long as it exposes an `.add` method:

```
(import '[java.util HashSet])

(def words
  (r/fold
    (r/cat #(HashSet.))   ; ❶
    r/append!
    (r-word text)))

(count words)       ; ❷
;; 185561
```

❶ `r/cat` accepts a function of no arguments. The function will be used to initialize the data structure to use to initialize the reduction on each parallel chunk. We can pass the `HashSet` constructor here.

❷ We count the result to see if the words are now distinct.

We see a surprising number when trying to count the result. This number is lower than the total count of words, but nowhere near 17200. The reason is that although sets contain unique elements before concatenation, they are concatenated as sequences potentially introducing duplicates. To fix this problem, we can walk the tree merging individual `HashSet` back into a set, getting rid of duplicates in the process:

```
(defn distinct-words [words]
  (letfn [(walk [root res]   ; ❶
    (cond
      (instance? clojure.core.reducers.Cat root)
      (do (walk (.left root) res) (walk (.right
      (instance? java.util.HashSet root)
      (doto res (.addAll root))
      :else res))]
  (into #{} (walk words (HashSet.)))))  ; ❷
```

```
  (count (distinct-words words)) ; ❸
  ;; 17200
```

❶ `walk` is a recursive function that merges the content of each leaf into a new `HashSet` instance. It starts a new recursion for each left and right branch (when we are on a `Cat` node).

❷ The function encapsulate the mutable part of the merge, only returning a persistent data structure as the last step.

❸ The number of distinct words is now the same that we found using a sequential distinct.

By iterating the tree structure produced by `r/cat` we can continue using a mutable data structure to incrementally build results. Alternatively, `r/cat` is also reducible and foldable, which means that we can further use reduce or even r/fold on the results, producing another parallel computation:

```
  (reduce + (r/map last words))                    ; ❶
  ;; 1105590

  (defn letter-frequency [words]
    (let [res (r/fold
                (r/cat #(StringBuilder.))          ; ❷
                #(doto %1 (.append %2))
                (r/map first words))]
```

```
     (frequencies res)))                          ; ❸

  (take 5 (sort-by last > (letter-frequency words)
  ;; ([\e 144858] [\n 82295] [\i 80318] [\s 78874]
```

❶ `reduce` can be used just fine on the result of `r/foldcat` or `r/fold` with `r/cat` as combine function. Here we are summing up the length of all the words in the set.

❷ In this `r/fold` invocation, we are using a custom `r/cat` constructor and a custom reducing function (instead of the standard `r/append!`). Each chunk is processed in parallel to create a `StringBuilder` instance, which is a fast way to concatenate large number of strings together.

❸ The results are iterated using the sequential interface provided by `clojure.core.reducers.Cat` to create a map of frequencies. This operation and following sort-by are executing outside the `fold` operation and thus sequentially.

The choice regarding processing parallel chunks accumulating results in mutable data structures should be guided by performance measurements. There are many parameters impacting on speed, like size of the chunk, amount of processing requested and final use of the produced output. These concerns are better addressed in the performance section at the end of the chapter.

See also

- fold is the core engine upon which `r/foldcat`, `r/cat` and `r/append!` operates. The details covered in `fold` are fundamental to understand what is covered in this chapter. Use of plain `fold` compared to `r/foldcat` allows for additional flexibility on the choice of combination and reducing function. Use `r/foldcat` when the provided `ArrayList` building block is sufficient to cover the given use case. Use `r/fold` with specific `r/cat` initializer to have additional control over the mutable data structure.
- cat is a concatenation transducer based on reduce. `cat` is much more general use than `r/cat` which is tied to reducers and folding. Use `cat` if your goal is to flatten nested collections.
- concat is used to iterate the tree produced by `r/fold` sequentially. `concat` is more general purpose than `r/cat` alone and can be used to merge collections together.

Performance considerations and implementation details

=> `O(n)` **(foldcat)**

For a general discussion about `r/foldcat` performance characteristics, please read fold performance section. `r/foldcat` is an application of `r/fold` that does not change its performance profile. The use of `r/cat` or `r/append!` in isolation, is constant time.

The most interesting aspect in terms of performance of `r/foldcat` compared to a plain `r/fold`, is the speed improvement related to the use of mutable data structures:

```
(require '[criterium.core :refer [quick-bench]])

(quick-bench (doall (r/foldcat (r-word text))))
;; Execution time mean : 166.526116 ms

(quick-bench (doall (r/fold concat conj (r-word
;; Execution time mean : 659.501099 ms
```

❶Please refer to the beginning of the chapter for the definition of `r-word` reducer and `text` initial collection of lines.
❷A similar computation based on standard conj shows a relevant increase in computation time.

If the problem you are solving includes a large input data-set, non-trivial processing steps and still produces a collection,

chances are that `r/foldcat` is outperforming plain `r/fold`. Moreover, if `r/foldcat` output needs additional processing, it can be processed again in parallel using additional `fold` operations.

[125] The Fork-Join model for parallel computation is a complicated subject that can't be illustrated in this book. If you want to know more, please read the following paper by Doug Lea, the author of Fork-join in Java: gee.cs.oswego.edu/dl/papers/fj.pdf

[126] The Iota library README explains how to use the library: github.com/thebusby/iota

[127] Please see the Wikipedia page for an introduction to the topic: en.wikipedia.org/wiki/Monoid

7.2 TRANSDUCERS

Transducers are a recently introduced Clojure feature which has an impact on many standard library functions, including introducing new dedicated ones like transduce, eduction, completing or cat. The impact of transducers on already existent functions usually consists on adding a new arity that returns a specific transducer type. The following list is a summary of all the transducers enabled functions currently available in the standard library and a brief description of their

use in this context. They are illustrated in deeper details later in the chapter:

- transduce: applies a reducing function and related transducer chain to a collection.
- completing: completes a binary function with the required calls so it can be invoked as a transducer.
- eduction: applies a transducer chain to a collection producing a lazy sequence of transformed elements.
- sequence: similar to `eduction` with additional caching if the sequence is iterated multiple times.
- into: copy elements from one collection type to another, optionally transforming them with a transducer chain.
- cat: is a transducer in the standard library. It concatenates each (collection) item input into a collection output.

All the following collection processing functions, when invoked without their collection argument, return a transducer:

- map: returns a transducer that applies a transformation to each element.
- map-indexed: like `map`, but the produced transducer also includes the index for each item.

- mapcat: returns a transducer that concatenates a transformation of each item into the final results.
- filter: returns a transducer that applies the reducing function or not, based on a predicate.
- remove: similar to `filter`, but inverting the meaning of the predicate.
- take: produces a transducer that terminates the reduction after the given number of elements.
- take-while: produces a transducer that terminates the reduction when a predicate returns `true`.
- take-nth: produces a transducer that collects each "nth" element in the collection.
- drop: produces a transducer that doesn't invoke the reduction for the first "n" elements.
- drop-while: produces a transducer that doesn't invoke the reduction until a predicate returns `true`.
- replace: produces a transducer that replaces every elements in the input collection following the given substitution map.
- partition-by: produces a transducer that splits the input collection every time a given function applied to each element returns a different value.

- partition-all: like `partition-by`, but allows partitions to have less than the requested number of elements at the end.
- keep: produces a transducer that transforms each element and keep those that are not `nil` after the transformation.
- keep-indexed: like `keep`, but the produced transducers also includes the index for each item.
- distinct: produces a transducer that removes all duplicates from the output of the preceding transducer.
- interpose: produces a transducer that alternates input items with the given separator.
- dedupe: like `distinct` but the produced transducer only removes contiguous duplicates, allowing repetitions if there is something separating them.
- random-sample: produces a transducer that let each item through based on the given probability value.

7.2.1 transduce

`function` `since 1.7`

Listing 7.5. → *Transducers, Reducers, Reduce, Collection Processing*

```
(transduce
  ([xform f coll])
  ([xform f init coll]))
```

`transduce` is one of the main entry points into the transducers abstraction. It works similarly to reduce, but it also accepts a composition of transforming reducing functions (the so called transducers) as the parameter "xform". The following example shows the same operation performed with `reduce` and `transduce` to see how the compare:

```
(reduce + (map inc (filter odd? (range 10))))
;; 30

(transduce (comp (filter odd?) (map inc)) + (ran
;; 30
```

❶reduce is used to sum 10 numbers after incrementing them and just keeping the odd ones.

❷The same operation is now executed with `transduce`.

The similarity with `reduce` is evident and deliberate, as the two operations are an expression of the same form of iteration. The main difference is that `transduce` isolates transforming operations (like `map` or `filter`) along the other arguments in the parameter list. This design has a lot of interesting consequences, such as enabling the same transformation chain to be reused in other contexts (for example the `core.async` library).

Contract

input

- "xform" is a function following the transducers semantic and is a mandatory argument. "xform" is invoked with the reducing function "f" and should return another function supporting at least two arities: a single argument call to be invoked at the end of the reduction and a two arity call for the actual reduction. An optional zero arguments call is currently unused by `transduce` but could be in the future.
- "f" is a reducing function of two arguments, receiving the results so far and the next item in "coll". It is a mandatory argument.

- "init" is optional. When present, it is used as the first accumulator value in the reducing process, similarly to reduce.
- "coll" is any collection supported by reduce. transients and scalars (like numbers, keywords and so on) are not supported. Anything else is, including `nil`, Java Iterable and arrays.

notable exceptions

- `NullPointerException` when "xform" or "f" is `nil`.
- `ArityException` could happen if the given "xform" doesn't support some of the required arities. Unless you're using custom transducers, you shouldn't be concerned about this. If you are using custom transducers, please check completing.

output

- returns: the result of applying the reducing function, along with any other transformer reducing functions, to "coll". When "coll" is `nil`, returns "init" or the result of invoking `(f)` without arguments. In both cases, the result also depends on the single argument arity of any transducer in the chain. A transducer could in fact alter the final result of the reduction (for example like partition-all that flushes the last partial partition if necessary).

Examples

Most of the time `transduce` can replace reduce, provided it is possible to rearrange transformations of the input (if any) to a transducer chain with comp. The following example illustrates the point by showing how to implement the Egyptian multiplication algorithm [128]. Ancient Egyptians didn't use times tables to multiply numbers, but they worked out the multiplication by decomposing the number by power of twos:

```
(defn egypt-mult [x y]
    (->> (map vector                    ; ❶
           (iterate #(quot % 2) x)
           (iterate #(* % 2) y))
      (take-while #(pos? (first %)))
      (filter #(odd? (first %)))
```

```
        (map second)
        (reduce +)))                          ; ❷

  (egypt-mult 640 10)
  ;; 6400
```

❶The computation begins by generating pairs of numbers where the first is increasingly halved while the second increasingly doubled.

❷After stopping at the first zero occurrence, we filter the odd numbers. The final step is to use reduce to sum up the second element in the pair.

The `egypt-mult` function is a potential candidate for `transduce`: it contains a call to `reduce` as last call and some processing of the initial input. Despite the promising shape of the operation, `transduce` does not support multiple input collections, so on a first attempt we need to leave the initial building of pairs outside `transduce`:

```
  (defn egypt-mult [x y]
    (transduce                                ; ❶
      (comp                                   ; ❷
        (take-while #(pos? (first %)))
        (filter #(odd? (first %)))
```

```
       (map second))
     +
     (map vector                          ; ❸
          (iterate #(quot % 2) x)
          (iterate #(* % 2) y))))

(egypt-mult 640 10)
;; 6400
```

❶The threading macro and the final `reduce` operation have been replaced by a `tranduce` call.

❷comp groups together all the preprocessing steps except for the initial creation of pairs.

❸The creation of the sequence of pairs happens through map and is not part of the transducers composition.

`egypt-mult` is now using a transducer chain, but this is happening after forming the pairs of numbers. It would be nice to include all processing inside `transduce`, as this is creating an intermediate sequence that would be nice to avoid. Our only hope is that there is a way to express the formation of pairs with an alternative design so it can be included in `transduce`. Although there is no guarantee that such alternative design exists, we can in this case express the formation of pairs as interleaving of two sequences followed by grouping.

The next call-out section shows how this can be done, including creating custom transducers if the standard library doesn't provide one.

Create a custom transducer

In the previous section, we have seen an example of `transduce` involving the Egyptian multiplication method. In translating the method into a version using `transduce` we mentioned that the example was only partially adaptable because `transduce` does not allow for multiple collections as input. There is however a different way to express the same algorithm which is completely transducible if we use interleave and partition-all:

```
(defn egypt-mult [x y]
    (->> (interleave                    ; ❶
           (iterate #(quot % 2) x)
           (iterate #(* % 2) y))
      (partition-all 2)
      (take-while #(pos? (first %)))
      (filter #(odd? (first %)))
      (map second)
      (reduce +)))

(egypt-mult 640 10)
;; 6400
```

❶The only change made to `egypt-mult` is to replace the `(map vector)` expression with interleave followed by partition-all.

The new algorithm is very similar to the original one, except for isolating the interleaving from the pair creation. This allows us to design the reduction on top of one sequence, while the other

is part of the creation of the transducer chain. There is a problem though: there is no `interleave` transducer in the standard library. We can create our own `interleave-xform` transducer as follows:

```
(defn interleave-xform                          ; ❶
  [coll]
  (fn [rf]
    (let [fillers (volatile! (seq coll))]       ; ❷
      (fn
        ([] (rf))
        ([result] (rf result))
        ([result input]
         (if-let [[filler] @fillers]            ; ❸
           (let [step (rf result input)]
             (if (reduced? step)                ; ❹
               step
               (do
                 (vswap! fillers next)          ; ❺
                 (rf step filler))))
           (reduced result)))))))               ; ❻
```

❶ `interleave-xform` is modeled on the same semantic of the interleave function in the standard library: it interleaves elements up to the end of the shortest sequence. `interleave-xform` contains all the required arities: no arguments, single argument and two arguments.

❷ `interleave-xform` assumes the interleaving is coming from a collection passed while creating the transducer. The other is the transducing collection. We need to keep track of the remaining items in the sequence as we consume them, so the rest of them is stored in a volatile! instance.

❸During the reducing step we verify to have at least one more element to interleave before allowing the reduction. Note the use of if-let and destructuring on the first element of the content of the volatile instance.

❹As any good transducer "citizen", we need to check whether another transducer along the chain has required the end of the reduction. In that case we obey without propagating any further reducing step.

❺If instead we are not at the end of the reduction and we have more elements to interleave, we can proceed to update our volatile state and call the next transducer using the "filler" element coming from the internal state. Note that at this point, this is the second time we invoke "rf": the first one for

the normal reducing step, the second is an additional reducing step for the interleaving.

❻ In case we don't have any more items to interleave, we end the reduction using reduced. This prevents `nil` elements to appear in the final output, exactly the same as normal `interleave`.

With `interleave-xform`, we can express Egyptian multiplication as follows:

```
(defn egypt-mult [x y]
  (transduce
    (comp
      (interleave-xform (iterate #(* % 2) y))
      (partition-all 2)
      (take-while #(pos? (first %)))
      (filter #(odd? (first %)))
      (map second))
    +
    (iterate #(quot % 2) x)))

(egypt-mult 4 5)
;; 20
```

❶What used to appear as the second iteration of increasingly doubling numbers in the `(map vector)` form is now considered the interleaving sequence that we pass when creating the transducer.

❷The other iteration with increasingly halved numbers is now the normal input for `transduce`. The two sequences are interleaved together and partitioned into vectors as part of the transducing step.

See also

- reduce is the core abstraction for the semantic of `transduce`. reduce is still necessary for all those cases where is not possible to reformulate the algorithm with `transduce`. There are also reasons related to laziness that prevent the implementation of some algorithms with `transduce` (please see sequence regarding this aspect in relation to transducers).
- fold is another reduce-like operation available in Clojure that enables parallelism (although with some restrictions).

- completing is an utility function often see in conjunction with `transduce`. It completes (as implied by the name) the two missing arities if you already have a transforming reducing function of two arguments. This happens more often when you are creating your own transducers, as the transducers in the standard library already provide all the relevant arities.

Performance considerations and implementation details

=> `O(n)` **Linear in the number "n" of items in the input collection**

`transduce` is implemented on top of reduce, so the same performance aspects applies. `transduce` is linear number of steps based on the length of the input and is not lazy. Memory allocation and other performance aspects depend largely on the reducing function and transducer chain but they are not dependent on `transduce` itself.

One of the innovative features of transducers in terms of performance is that they avoid the generation of intermediate sequences while processing the input ahead of reduction. It follows that reductions requiring additional processing before

`reduce` are more likely to receive a performance boost. The following benchmark demonstrates this fact on a simple operation:

```
(require '[criterium.core :refer [quick-bench]])

(let [coll (into [] (range 1000))] ; ❶
  (quick-bench (reduce + (filter odd? (map inc c

;; Execution time mean : 39.507721 µs

(let [coll (into [] (range 1000))] ; ❷
  (quick-bench (transduce (comp (map inc) (filte

;; Execution time mean : 16.090126 µs
```

❶ `reduce` is used to sum a collection of numbers that have been incremented and filtered.

❷The same operation is performed with `transduce` which allows processing to happen at the same time the collection is iterated for reduction.

You shouldn't hesitate to use `transduce` if your algorithm can be easily rewritten in terms of composition of transducers. The Egyptian multiplication example in this chapter should be

considered border line: it requires a new design and a custom transducer which is additional complexity that needs to be justified in terms of the performance improvements it brings.

7.2.2 eduction

`function` `since 1.7`

Listing 7.6. → *Transducers, Non-caching Sequence, Transforming Sequence*

```
  (eduction [& xforms])
```

`eduction` takes any number of transducers (with or without explicit comp composition) and a collection and applies the transducers chain to each element in the collection:

```
  (take 2 (eduction (filter odd?) (map inc) (range
  ;; (2 4)
```

❶Note that despite using an infinite sequence, `eduction` works lazily and returns with the requested elements.

Compared to other transducers-aware functions (sequence, transduce or into), `eduction` is the only one supporting

multiple transducers without requiring explicit comp. Differently from sequence though, `eduction` starts a brand new loop for each sequential operation, including running transducers again. `eduction` is lazy but not strictly, as it produces enough of the sequence to satisfy the request in chunks of 32 items [129].

Contract

input

- 0 arguments: always return empty list.
- 1 argument: when a single argument is present this is assumed to be a sequential collection (all those supported by sequence). In this case the collection is transformed into a sequence and iterated without transformations. It accepts `nil` as a single argument.
- 2 or more arguments: all arguments except the last are considered transducer forms, while the last is a collection.

notable exceptions

- `NullPointerException` is thrown if any of the xform transducers is `nil`.

- `IllegalArgumentException` is thrown for arguments in xform position which are not transducers.

output

The output changes based on the presence of `nil` and number of arguments:

- Returns `()` (the empty list) if no arguments are present, or passing `nil` as an argument.
- Returns the sequential version of "coll" if a single argument is present and is supported by seq.
- Returns the transformed iteration of "coll" as directed by the transducers composition in any other case.

Examples

The result of `eduction` is a rare example of lazy sequence-like collection that does not cache its results. As the composition of transducers return a value for each item in the input, no chain of cons-cells is created to contain the resulting values. Standard lazy sequences cache their results so repeated access to the same item is possible without repeating computation (removing repetition of potential side effects).

`eduction` becomes faster than sequence when the final goal is to reduce the output, an operation that does not require to cache results in any way. The following example illustrates the effect of caching:

```
(let [input (sequence (map #(do (print ".") %))
      odds (filter odd? input)
      evens (filter even? input)]
    (if (> (first odds) (first evens))
      (println "ok")
      (println "ko")))
;; ..........ok       ; ❷
```

❶We use a side effecting println to see caching effects.
❷Note the presence of 10 dots printed in output.

The example shows that even when the same sequence requires multiple passes (to filter odds and then evens in this case), it only produces the initial computation once (the dot only prints once for each item). To achieve this result, `sequence` produces a chain of cached values amenable for subsequent access. Now observe the same example after replacing sequence with `eduction`:

```
(let [input (eduction (map #(do (print ".") %))
      odds (filter odd? input)
      evens (filter even? input)]
    (if (> (first odds) (first evens))
      (println "ok")
      (println "ko")))
;; ....................ok ; ❷
```

❶Compared to the previous example, we only changed `sequence` with `eduction`.

❷You can now count 20 dots printed in output.

The presence of 20 dots in the last example with `eduction` shows that the source sequence needs a new pass of computation each time we call filter on it.

Here's a more interesting example. We are going to design a `best-product` function that applies transformations to a large collection of financial products (kept small for the purpose of this example). The function accepts parameters influencing the composition of transducers. But first let's have a look at the input data:

```
(def data
  [{:fee-attributes [49 8 13 38 100]
```

```
     :product {:visible true
               :online true
               :name "Switcher AA126"
               :company-id 183
               :part-repayment true
               :min-loan-amount 5000
               :max-loan-amount 1175000
               :fixed true}
     :created-at 1504556932728}
    {:fee-attributes [11 90 79 7992]
     :product {:visible true
               :online true
               :name "Green Professional"
               :company-id 44
               :part-repayment true
               :min-loan-amount 25000
               :max-loan-amount 3000000
               :floating true}
     :created-at 15045569334789}
    {:fee-attributes [21 12 20 15 92]
     :product {:visible true
               :online true
               :name "Fixed intrinsic"
               :company-id 44
               :part-repayment true
               :min-loan-amount 50000
               :max-loan-amount 1000000
```

```
                :floating true}
      :created-at 1504556936983}])
```

Next we define two groups of transducers. The first `prepare-data` shapes the data in a slightly different form than the raw input, while `filter-data` filters them based on user arguments. Several helper functions are also present to improve readability:

```
(import 'java.util.Date)

(defn- merge-into [k ks]    ; ❶
  (map (fn [m]
    (merge (m k) (select-keys m ks)))))

(defn- update-at [k f]
  (map (fn [m]
    (update m k f))))

(defn- if-key [k]
  (filter (fn [m]
    (if k (m k) true))))

(defn if-equal [k v]
  (filter (fn [m]
    (if v (= (m k) v) true))))
```

```
(defn if-range [k-min k-max v]
  (filter (fn [m]
    (if v (<= (m k-min) v (m k-max)) true))))

(def prepare-data           ; ❷
  (comp
    (merge-into :product [:fee-attributes :creat
    (update-at :created-at #(Date. %))))

(defn filter-data [params]  ; ❸
  (comp
    (if-key :visible)
    (if-key (params :rate))
    (if-equal :company-id (params :company-id))
    (if-key (params :repayment-method))
    (if-range :min-loan-amount
              :max-loan-amount
              (params :loan-amount))))

(defn xform [params]        ; ❹
  (comp
    prepare-data
    (filter-data params)))
```

❶ `merge-into`, `if-key`, `update-at`, `if-equal` and `if-range` are small wrappers around `map` or `filter` transducers. By extracting them away from comp and giving them names, we can keep the transducers chain compact and expressive.

❷ `prepare-data` is the first composition of transducers. It contains anything related to transform or re-shape the input data. For example, we show here how to transform epoch time into `java.util.Date` objects and we want all keys to appear in the product hash-map.

❸ `filter-data` contains a composition of filters on the input data. Most of them are based on the search parameters, but other (like "visibility") are not. We can for example only show those products that can fulfill the given loan amount request.

❹ All transducers so far are assembled into a final `xform` composition.

`xform` is a composition of transducers which includes a preparation and a filter phase. To return the list of results we are going to use `eduction` instead of `sequence`. This is possible because of the reducing nature of the computation, going from the large list to a single or few products. If we had to retain all transformed products to access them multiple times, then sequence would be a better choice. Here's how the `best-product` function could be implemented:

```
(defn- best-fee [p1 p2]                ; ❶
  (if (< (peek (:fee-attributes p1))
         (peek (:fee-attributes p2)))
    p1 p2))

(defn best-product [params data best-fn]
  (reduce
    best-fn
    (eduction (xform params) data)))  ; ❷

(best-product
  {:repayment-method :part-repayment
   :loan-amount 500000}
  data
  best-fee)

;; {:name "Fixed intrinsic",
;;  :fee-attributes [21 12 20 15 92],
;;  :company-id 44,
;;  :floating true,
;;  :part-repayment true,
;;  :online true,
;;  :max-loan-amount 1000000,
;;  :visible true,
;;  :min-loan-amount 50000,
;;  :created-at #inst "2446-10-10T12:49:29.839-0
```

❶ Somewhat arbitrarily, we decide that the best product is the one offering the minimum last fee attribute.
❷ `eduction` also takes an already composed transducer as argument. We use the parameters from the request to prepare a different transducer configuration. You can see that the `eduction` output is the input for the enclosing `reduce` operation.

`eduction` lack of caching generally offers better performance than `sequence` in reducing contexts. It also enables the option of saving the equivalent of a "delayed reduce recipe" that can be used at some later point. With reference to the previous example, we could define a few eductions for the most popular input parameters:

```
(def best-part-repayment ; ❶
  (eduction (xform {:repayment-method :part-repa

(def best-fixed ; ❷
  (eduction (xform {:rate :fixed}) data))

(:name (reduce best-fee best-part-repayment)) ;
;; "Fixed intrinsic"
```

```
  (:name (reduce best-fee best-fixed))
  ;; "Switcher AA126"
```

❶ `best-part-repayment` represents the computation and the data necessary to retrieve which product has the lowest fee that allows "part repayment" as repayment method. The computation is defined, but doesn't run yet. This is similar to a delayed `reduce` operation.

❷ We also define another reduction `best-fixed` which retrieves with the cheapest fixed rate product.

❸ By using `reduce` we force the `eduction` and retrieve a result.

The eductions defined in the previous example are created at compile time and reused throughout the lifetime of the application additionally improving performance.

See also

- sequence is `eduction` close sibling (in the context of transducers). Use sequence when you plan to use the produced output multiple times, for example assigning it to a local binding. Use `eduction` when there is no plan to perform multiple scans of the output collection, saving on unnecessary caching. See also the performance section for some additional information.
- transduce allows the creation of a transforming reducing function, using the transducers composition to drive the computation on each item. Prefer transduce when there is no need to delay the computation, or to exploit some form of parallelism using fold.

Performance considerations and implementation details

=> `O(n)` **Linear in "n" number of items in the input collection**

`eduction` performs linearly, iterating through the items to perform transformations as dictated by the transducer chain. `eduction` is lazy, consuming just enough items as requested upstream. The laziness in `eduction` is chunked: new items are

processed in chunks of 32 elements each. Once the 32th element is reached, all others up to the 64th are processed and so on.

As explained several times throughout the chapter, `eduction` does not cache results. To understand the meaning of caching, we can use a special side-effecting counting transducer, so we can see how many times transducers are invoked:

```
(def cnt1 (atom 0))
(let [res (eduction (map #(do (swap! cnt1 inc) %
  (conj (rest res) (first res)) ; ❶
  @cnt1)
;; 20

(def cnt2 (atom 0))
(let [res (sequence (map #(do (swap! cnt2 inc) %
  (conj (rest res) (first res)) ; ❷
  @cnt2)
;; 10
```

❶first and rest have the effect of forcing the `eduction` to re-execute, at least up to the chunk size necessary to fulfill the request. Although the input collection contains 10 elements only, it appears that we are iterating them twice, as printed by "@cnt1" counter.

❷We now repeat the same operation on a sequence, which shows the caching behavior by not executing the computation again. The counter "@cnt2" correctly shows only 10 evaluations.

`eduction` approach has benefits on memory allocation at the price of possible re-evaluations. Since results are not cached, there is no "holding on to the head" problem either:

```
(defn busy-mem []
  (str (/ (-
    (.. Runtime getRuntime totalMemory)
    (.. Runtime getRuntime freeMemory))
  1024. 1024.) " Mb"))

(System/gc) (busy-mem)  ; ❶
;; 5.574 Mb

(def s1 (eduction (map inc) (range 1e7)))
(last s1)
```

```
(System/gc) (busy-mem)  ; ❷
;; 7.615 Mb

(def s2 (sequence (map inc) (range 1e7)))
(last s2)

(System/gc) (busy-mem)  ; ❸
;; 304.5126 Mb
```

❶We start a new REPL with `-Xmx512M` option to allocate max 512MB of heap size. Just after starting, we ask the JVM for garbage collection and measure the currently used heap size, which is around 5Mb.

❷We now execute an `eduction` pass on 10 million elements taking only the last. This is going to process the entire sequence moving the iterator from the first item down to the last. We keep the `eduction` alive by storing the head of the sequence "s1", but by the time we call the garbage collector and measure the memory again, we can see that the heap size only increased a couple of Mb.

❸If we try the same using sequence instead, we can see that memory remains allocated even after calling the garbage collector. `sequence` is caching all elements and by holding the head "s2" all items remain in memory.

The way `eduction` is implemented is by building a `java.lang.Iterable` object every time `eduction` is called. Any operation that requires access to the sequence is going to receive a brand new `java.lang.Iterator`, supporting the typical `hasNext()` followed by `next()` semantic. The iterator can only move forward and once it moves past an item, that item is eligible for garbage collection. Differently from normal sequences, the head of the sequence is not holding the rest of the sequence alive through caching. In the previous example, "s1" is pointing at the first chunk of the `eduction` while everything else is already eligible for garbage collection.

7.2.3 completing

`function` `since 1.7`

Listing 7.7. → *Transducers, Arity Completion, Helper Function*

```
(completing
  ([f])
  ([f cf]))
```

`completing` provides (or replace) the single arity call in a reducing function. This is useful when working with transduce:

```
(transduce (map inc) - 0 (range 10))
;; 55

(transduce (map inc) (completing -) 0 (range 10))
;; -55
```

❶transduce invokes subtraction `-` with the results of the reduction as the last step. Subtract with a single argument `(--55)` negates the already negative output, resulting in the apparently wrong result.

❷ `completing` wraps over subtraction single arity using the identity as a default, which leaves the result untouched.

There are three possible problems related to the single arity call that can arise using a reducing function with `transduce`:

1. The single arity version of the reducing function is generating unwanted changes. This is the case we've seen in the previous example.
2. The single arity version of the reducing function is non existent. In this case we want to provide one to avoid exceptions.
3. We want to provide a specific wrap-up behavior for `transduce`, so when the reduction is complete we still have control of the very last step.

Based on the list of scenarios above, `completing` is a handy function that completes or fixes the reducing function for `transduce`.

Contract

input

- "f" is a reducing function. It is expected to support a zero-arity call and the standard two-arity call with an accumulator/item pair. It is mandatory argument.
- "cf" is a "closing function", the function that should be called at the end of the reduction process. "cf" must support at least one argument. "cf" is optional argument. When "cf" is not provided identity is used.

output

- returns: a function accepting zero, one and two arguments. With no arguments, it returns the result of invoking "f" with no argument. With a single argument, it returns the result of invoking "cf" (or `identity` when omitted) with that argument. Finally, it invokes "f" with two arguments when invoked with two arguments.

Examples

The following example illustrates the way invocations flow through a transducer chain when invoked with `transduce`. `identity-xform` is a custom transducer that works like the `identity` function with additional printing when a dynamic variable is set to `true`. By executing the transducer with tracing enabled, we can follow the invocations on screen:

```
(def ^:dynamic *debug* false)

(defn- print-if [s]                          ; ❶
  (when *debug* (print (str s " "))))

(defn- identity-xform                        ; ❷
  ([]
    (fn [rf]
      (fn
        ([] (print-if "#0") (rf))
        ([acc] (print-if "#1") (rf acc))
        ([acc item] (print-if "#2") (rf acc item
  ([x] x))

(defn- completing-debug [f]                  ; ❸
  (completing f
    #(do (print-if "#done!") %)))
```

```
  (binding [*debug* true]                    ; ❹
    (transduce
      (comp (map inc) (identity-xform))
      (completing-debug +)
      [1 2 3]))

  ;; #2 #2 #2 #1 #done! 9
```

❶ `print-if` conditionally print to the standard output based on the dynamic variable `*debug*`.

❷ `identity-xform` custom transducer prints a different message for each provided arity. It does not touch the results in any other way. We need this custom transducer to see what happens when the single-arity call is invoked at the end of the reduction.

❸Similarly, `completing-debug` wraps `completing` to print a message on screen. We want to verify when the function generated by `completing` is called.

❹We put everything together when calling `transduce`. `*debug*` is set to `true` so we can see the related messages on the screen. The transducing chain increments each item and the pass them through the `identity-xform` custom transducer.

We can see from the results of the example that, after the two-argument call is called for each element in the input collection, the single-arity of each transducers is called (we only see messages from `identity-xform` but map transducer single-arity is called right before that). The very last step is performed by `complteing-debug` which prints the "#done!" message, just before the result is returned.

> Warning
>
> Note that the 0-arity call is not currently used by any transducing context but it could be in the future. Although transducers are not using it yet, other contexts might be, for example fold: it is possible to use a stateless transducer with `fold`, but the reducing function needs to provide an initialization value when called with no arguments. `(map inc)` for example, generates 0 with "+" enabling `fold` use: `(r/fold ((map inc) +) (range 10))`.

Noting that the 1-arity version of reducing function is called last, we can create a function to calculate the average of numbers based on `completing`. The following is the same

example found in filter but rewritten to take advantage of transducers:

```
(def events                                    ; ❶
  (apply concat (repeat
    [{:device "AX31F" :owner "heathrow"
      :date "2016-11-19T14:14:35.360Z"
      :payload {:temperature 62.0
                :wind-speed 22
                :solar-radiation 470.2
                :humidity 38
                :rain-accumulation 2}}
     {:device "AX31F" :owner "heathrow"
      :date "2016-11-19T14:15:38.360Z"
      :payload {:wind-speed 17
                :solar-radiation 200.2
                :humidity 46
                :rain-accumulation 12}}
     {:device "AX31F" :owner "heathrow"
      :date "2016-11-19T14:16:35.360Z"
      :payload {:temperature 63.0
                :wind-speed 18
                :humidity 38
                :rain-accumulation 2}}])))

(defn average [k n]                            ; ❷
  (transduce
    (comp
```

```
        (map (comp k :payload))
        (remove nil?)
        (take n))
      (completing + #(/ % n))
      events))

  (average :temperature 10)
  ;; 62.5
  (average :solar-radiation 60)
  ;; 335.2000000000004
```

❶ `events` is a simulation of an infinite series of readings from weather sensors connected to a central server. We just concatenate the same 3 events to generate an infinite stream.

❷ `average` contains a call to `transduce` and handles two parameters to change what data are returned. After selecting the specific sensor to read, we remove potentially empty results and take the requested amount of information. `completing` is adding a last step to + that calculates the average.

The need for `completing`

`completing` exists mainly because some functions that work properly with standard reduce, would potentially return different results when used with `transduce`. The root cause of this behavior are stateful transducers. Some stateful

transducers store partial results and then wait for a signal that the end of the reduction has been reached (see partition-all for instance).

There is nothing to indicate when the last item in the collection is passing through the transducer chain. In addition to that, any transducer can request the end of the reduction using reduced. For these reasons, transduce is designed to signal the end of the reduction by using the single arity call of the function sitting at the top of the transducer chain. The following example shows the steps happening internally to a `transduce` call:

```
(def xform                                ; ❶
  (comp (map inc)
        (partition-all 3)
        cat))

(def xform-reductor                       ; ❷
  (xform
    (completing +
      #(do (print "#done! ") %))))

(xform-reductor 0 0)                      ; ❸
;; 0
(xform-reductor 0 0)                      ; ❹
;; 0
(xform-reductor 0)                        ; ❺
;; #done! 2
```

❶A transducer chain is assembled as usual. Notice that partition-all is a stateful transducer with an internal buffer length 3. The cat transducer that follows unwraps any inner lists created by `partition-all`.

❷The transducer chain is invoked with `+`, preparing it for execution.

❸The first invocation of `xform-reductor` simulates what happens when 0 is the first element in the input collection. It gets incremented by the time it reaches `partition-all`, which stores the number in the internal buffer because it has not reached size 3 yet. `0` is returned, because `+` is yet to be invoked.

❹After calling `xform-reductor` with 0 again, `partition-all` contains [1,1] in its internal buffer.

❺We now simulate the end of the reduction calling `xform-reductor` with a single argument.

The example shows what happens inside `transduce` when invoked with a collection with 2 zeros in it: `[0 0]`. First the transducers chain is invoked with `+`, which instantiates all the transducing functions. It is named `xform-reductor` and `partition-all` initializes its internal state at this point. `xform-reductor` is a fully fledged reducing function, which increments each item, stores it temporarily in `partition-all` (up to 3) and sums up this item with the previous results.

Without the last step, `(xform-reductor 0)`, the result would be just "0". This is because `patition-all` doesn't have a chance to empty the internal buffer. `transduce` conveniently invokes the last step for us, making sure tear-down of transducers state happens consistently.

In summary:

- Transducers need signaling that the reduction is over.
- The reducing function (`+` plus in our example) sits at the end of the transducer chain and the last transducer (`cat` in our example) is unaware of what follows.
- The single-arity invocation rule makes sure that any state is cleared up properly at the end of the reduction.
- The reducing function at the end of the chain is affected by the same rule and needs to be prepared for it.
- `completing` is a convenient helper to fix any possible problem affecting the reducing function in this sense.

See also

- monoid has some similarities with `completing` but they are made for different purposes and cannot be used interchangeably. `monoid` compensates for the lack of a zero-arity call in a reducing function, while `completing` is mainly dedicated to the single-arity call.
- transduce is the main use case for `completing`.

Performance considerations and implementation details

=> `O(1)` **function generation**

`completing` generates a new function in constant time. There is nothing particularly relevant in terms of performance or implementation details.

7.2.4 cat

`function` `since 1.7`

Listing 7.8. → *Transducers, Concatenation, Flattening*

```
  (cat [rf])
```

`cat` is the only pure transducer (which is not connected to an already existent sequence function) present in the standard library. While there are many other transducers that can be obtained calling a sequence processing function (usually without arguments), `cat` is the only one that can be used directly:

```
  (eduction
    (map range)
    (partition-all 2)
```

```
   cat
   cat                ; ❶
   (range 5))

 ;; (0 0 1 0 1 2 0 1 2 3)
```

❶Note the use of `cat` without any wrapping parenthesis. This is just an effect of `eduction` which combines transducers through comp internally.

`cat` assumes the preceding transducer (or combination thereof) is producing a sequential result. `cat` iterates through the inner sequential step to remove one layer of sequential wrapping.

Contract

input

- "rf" is a reducing function and is a mandatory argument. Like in the case of other transducer, "rf" needs to support zero, one or two arguments calls. There is no need to provide "rf" explicitly, as this is usually performed by transduce-aware functions like `transduce`.

output

- returns: the result of applying "rf" to the initial value and the first element of the input, then the result to the second element and so on (see reduce. In the context of a transducing chain (a composition of transducers), `cat` iterates the previous sequential output applying the given reducing function, effectively removing the sequential layer.

Examples

The following example illustrates the relationship of `cat` with reduce:

```
((cat +) 0 (range 10))  ; ❶
;; 45

(reduce + 0 (range 10)) ; ❷
;; 45
```

❶ We can invoke `cat` with a reducing function such as +. Normally this is not the way `cat` is used, which is in the context of transducer composition.

❷ `cat` works like calling `reduce` once a reducing function has been assigned to it.

The previous example shows that `cat`, once instructed with a reducing function like `+` (plus), expects the input to be sequential. Differently from `reduce`, `cat` is a proper transducer and can be used to flatten back the inner collections produced by an upstream step:

```
(def team
  ["jake" "ross" "trevor" "ella"])

(def week
  ["mon" "tue" "wed" "thu" "fri" "sat" "sun"])

(defn rotate [xs]   ; ❶
  (sequence cat (repeat xs)))

(def rota           ; ❷
  (sequence
    (map vector)
    (rotate team)
    (rotate week)))

(last (take 8 rota))
;; ["ella" "mon"]
```

❶ `rotate` is a function that creates infinite sequence of repeating elements starting from a finite enumeration. We could for example repeat the days of the week or the members of a team. repeat creates a new sequence to enclose the infinite repetition of the input, but we don't want the inner wrapping layer to appear. `cat` is used here to eliminate the inner collections forming a flat sequence.

❷ `rota` shows how to assign a team member to each day of the week, even when the two enumerations have different length. The `rota` function can be used to decide who is in charge of a specific aspect of the project each day.

`cat` and `mapcat`

Looking back at the examples in this chapter, you can see how `cat` is effectively just concatenation logic, while sequence (or other transducer-aware iteration mechanism) handles the iteration. Before the introduction of `cat` in Clojure 1.7, the closest idiom to solve similar problems, was to use mapcat with identity:

```
(def seasons ["spring" "summer" "autumn" "winter"])
(nth (mapcat identity (repeat seasons)) 10) ; ❶
;; autumn

(nth (eduction cat (repeat seasons)) 10)    ; ❷
;; "autumn"
```

❶ In this case we only need concatenation but we don't need transformation. For this reason the "mapping" part of map is given identity as transforming function.

❷ `identity` can be removed completely by using `cat` only.

`cat` is essentially the "cat" part of `mapcat`. Since the iteration mechanism is now implemented by transducers, we don't have to necessarily carry over the mapping and just use `cat` alone when appropriate.

`mapcat` (which is also transducer-ready) is still useful when flattening includes a transformation step. For instance:

```
(take 10 (eduction (mapcat range) (range))) ; ❶
;; (0 0 1 0 1 2 0 1 2 3)
```

❶ The use of a range on top of another range with `mapcat` produces a series of increasingly bigger enumerations all starting at zero.

See also

- mapcat adds to `cat` the transformation step ahead of concatenation. If your algorithm always requires a transformation, prefer `mapcat` instead of `cat` over `map`. `mapcat` is also necessary when transducers are not an option (this has often to do with different laziness guarantees offered by transducers compared to normal sequential processing).
- concat lazily concatenates collections together.

- flatten is a stronger form of concatenation which includes removing also any layer of nested collection, not just the first layer. `cat` and `mapcat` will only affect the first level of nesting, leaving inner collections untouched. Use `flatten` when you want to remove any level of nested collections.
- r/cat in reducers is a specific concatenation operation specifically created to perform fold efficiently.
- r/foldcat combines `r/fold` and `r/cat` together. Like `r/cat` this is a specific operation used in reducers only.

Performance considerations and implementation details

=> `O(1)` **function generation**

=> `O(n)` **runtime**

`cat` is a function generator. The generation is a constant time operation that does not have relevant impact on performance optimization. In terms of runtime performance profile, `cat` is linear in the number of elements created by the upstream steps (same as `reduce` on which `cat` implementation is based).

[128] Please see the following Wikipedia article to know more about Egyptian multiplication en.wikipedia.org/wiki/Ancient_Egyptian_multiplication

[129] The reason for the chunk size 32 is because `eduction` returns a `java.lang.Iterable` which is iterated in chunk of 32 items by all sequence-aware functions in Clojure

7.3 REDUCED, REDUCED?, ENSURE-REDUCED, UNREDUCED

`function`

`since 1.5 (reduced and reduced?)`

`since 1.7 (ensure-reduced and unreduced)`

Listing 7.9. → *Reduce, Reducers, Transducers*

```
(reduced [x])
(reduced? [x])
(ensure-reduced [x])
(unreduced [x])
```

The four functions in this section control the presence of a wrapping `clojure.lang.Reduced` marker object around other Clojure objects. The `reduced` signaling mechanism

allows reducing functions to impose a premature end of the reduction:

```
(reduce
  (fn [acc el]           ; ❶
    (if (> el 5)
      (reduced acc)      ; ❷
      (+ acc el)))
  (range 10))

;; 15
```

❶The custom reducing function passed to reduce includes a condition on the incoming element "el". It only reduces with + if the element is less than 5.

❷When the element is more than 5, a `reduced` value wrapping the result so far is returned.

When reduce finds an element wrapped in a `Reduced` instance, it stops recurring over the remaining elements in the input. Both reducers and transducers take advantage of the signaling mechanism to signal a premature end of the reduction (for example, take-while immediately wraps the results so far in a `reduced` so other reducers/transducers

down the line have a chance to avoid computationally intensive operations).

The wrapped `reduced` instance supports the `IDeref` interface, allowing for easy inspection using the `@` (at) operator:

```
(def r (reduced "string"))

(reduced? r)
;; true

r
;; #object[Reduced 0x3e7d {:status :ready :val "

@r  ; ❶
;; "string"
```

❶A `reduced` object supports the `IDeref` interface similar to atoms or other reference objects.

Contract

input

- "x" is requested argument in all cases. It can either be an object already wrapped in a `clojure.lang.Reduced` instance or a plain object of any type.

notable exceptions

- `ClassCastException` is the typical exception encountered when a wrapped `reduced` object is used forgetting to dereferencing it first.

output

- `reduced` : returns the input object "x" wrapped in a new instance of `clojure.lang.Reduced` .
- `reduced?` : returns `true` if "x" is a `reduced` object, `false` otherwise.
- `ensure-reduced` : is like `reduced` , but does not wrap again "x" when "x" is already a `reduced` object.
- `unreduced` : retrieve the dereferenced version of "x" when "x" is a `reduced` object.

Note

there are 3 ways to look inside the content of a `reduced` object: given "r" is `reduced`, then: `@r`, `(deref r)` and `(unreduce r)` are equivalent.

Examples

With `reduced` we can provide an alternative way to drop elements from a collection after some condition is met. drop-while implements a similar concept, but if we are interested in the first element "after" the condition is met, we need an additional first call. The `reduced` option is not only shorter but likely faster; depending on the type of the input collection, the iteration could be already optimized for `reduce`:

Let's compare the two forms in the following example [130]:

```
(def random-vectors ; ❶
  (repeatedly #(vec (drop (rand-int 10) (range 1(

(first ; ❷
  (drop-while
    #(>= 3 (count %))
    random-vectors))
;; [2 3 4 5 6 7 8 9]
```

```
(reduce ; ❸
  #(when (> (count %2) 3) (reduced %2))
  random-vectors)
;; [4 5 6 7 8 9]
```

❶ `random-vectors` produces an infinite stream of vectors of different sizes, from 0 up to 10 items.

❷There are several options to solve the problem of returning the first vector longer that 3 items. One of them is to use sequential functions like drop-while to remove items from the head of the sequence until the predicate is `true`. At that point we need to call first to get the first item from the remaining vectors in the list.

❸We can use reduce and reduced to achieve a similar effect and stop the underlying loop when we find the first item that satisfies the predicate. `reduced` stops the computation and exit `reduce` at that point, removing the need to take the first element from the remaining ones.

A "moving average" is a time-based series of average values over time. It's useful to show snapshot results over time, without necessarily waiting for other values that might still be in transit. A perfect example is the typical plot for a stock price where each dot on the chart is an average value of the previous

(daily or otherwise) [131]. The idea is to use `reduce` in a blocking fashion, start processing values as they arrive, effectively suspending the loop. We need a few tools for this to happen:

- `java.util.concurrent.LinkedBlockingQueue` is a type queue where push-pop operations block waiting for elements to be in the queue. We can use it as a channel to feed new values in.
- We also need a "signal" to mark the end of the event stream, so the `reduce` operation can unblock and return.

To provide the given design, we are going to need to create and check for reduced elements during the reducing process. A `reduced` signal is sent by the client down the queue, to signal the end of the events for that session. The `reduced` signal is interpreted in the reducing function and propagated to `reduce` implementation to stop the loop (in normal conditions, the signal would be the end of the input sequence, but we don't have that in this case):

```
(import 'java.util.concurrent.LinkedBlockingQueu
(def values (LinkedBlockingQueue. 1))         ; 
```

```
(defn value-seq []                                ; (
  (lazy-seq
    (cons (.take values) (value-seq))))

(defn moving-average [[cnt sum avg] x]            ; (
  (let [new-cnt (inc cnt)
        new-sum (+ sum (unreduced x))
        new-avg (/ new-sum (double new-cnt))
        res [new-cnt new-sum new-avg]]
    (println res)
    (if (reduced? x)
      (reduced res)
      res)))

(defn start []                                    ; (
  (let [out *out*]
    (.start (Thread.
      #(binding [*out* out]
        (println "Done:"
          (reduce
            moving-average
            [0 0 0]
            (value-seq)))))))))

(start)
(.offer values 10)
;; [1 10 10.0]
(.offer values 10)
```

```
;; [2 20 10.0]
(.offer values 50)
;; [3 70 23.333333333333332]
(.offer values (reduced 20))
;; [4 90 22.5]
;; Done: [4 90 22.5]
```

❶ A linked queue of size one indicates that as soon as the first element is added, no more push into the queue are allowed until that element is consumed. This determines reduce to block waiting for another element to be available.

❷ `value-seq` creates a lazy sequence on top of of the queue, a sequential interface that `reduce` can understand and use without knowing the internals of a blocking queue.

❸ `moving-average` is the reducing function. The function is unaware of the queue-based implementation and could work just fine in a normal `reduce` call. The function assumes that, at any point, the argument "x" could be `reduced`. In that case the end of the reduction signal is sent to `reduce` implementation by wrapping the result returned in a `reduced` call.

❹ The `start` function starts the reduction in a separate thread. The main thread is used for the client to send values to the queue, while the background thread runs the (otherwise blocking) reduction. Note that we redirect the standard output of the new thread to the current one, so we can see messages at the REPL. The call to `reduce` is blocking and only returns when a `reduced` element is sent down the queue.

In the example, the interaction starts by forking a new thread containing the background (blocking) `reduce` call. The current

thread holds a reference to the queue that can be used to send values to `reduce`. As soon as we offer a value wrapped by `reduced` the thread stops and exit, showing the last available average. The vector contains a triplet with the count of elements seen so far, their sum and finally their average.

From the example, we can see how `reduced`, `reduced?` and `unreduced` are used:

- To transform a normal element into a `reduced` one, use `reduced`.
- To verify if an element is `reduced`, use `reduced?`.
- To use an element regardless of it being `reduced` or not, use `unreduced`. Using the dereference form with `@` in this case would cause an exception without checking with `reduced?` first.

Transducers and reduced

`ensure-reduced` is the only function left out from the example. `ensure-reduced` is especially used when multiple functions are involved in the reduction and each one has equal opportunity to stop the loop.

In that case we want to be able to mark something as `reduced` without necessarily checking if it was already. `ensure-reduced` has been introduced at the same time as transducers because transducers chains are one of the main clients of `ensure-reduced`: a transducer would like to be able to signal the end of reduction regardless of other transducers in the chain and `ensure-reduced` avoids the conditional form.

An example of `ensure-reduced` can be found in take transducer: after taking the given amount of element, `take` signals the end of the reduction using `ensure-reduced`.

See also

- reduce is the main recipient of `reduced` objects. Worth understanding a bit more about `reduce` if you need to work with the functions in this section.

Performance considerations and implementation details

=> `O(1)` **function generation**

`reduced`, `reduced?`, `unreduced` and `ensure-reduced` have very little impact on computation. The `reduced` object which is created in constant time contains a reference to the original object, which adds almost nothing to memory footprint.

[130] Thanks to Max Penet (@mpenet on the Slack Clojurians group chat) for suggesting the inclusion of this comparison in the book

[131] Please read the Wikipedia article on moving averages available at en.wikipedia.org/wiki/Moving_average

7.4 SUMMARY

Reducers and transducers are powerful tools. They have been added incrementally to the standard library to integrate with

the already existing primitives for sequential processing. `reducers` offers parallelism through the fork-join framework, compensating for the limitations of pmap with less predictable tasks. `transducers` offer a model of composition that scales better than sequential processing for larger data transformations.

PART 3. DATA STRUCTURES

This part of the book is dedicated to the functions operating on data structures, which usually involves one or more collections used as argument. Data structures and the functions operating on them, are a fundamental part of Clojure (as well as other programming languages). The book uses the term data structure to identify a container for data and to distinguish them from "scalars" or "singletons", which are atomic data that cannot be divided any further.

Clojure implements many types of data structures and also interoperates well with those provided by Java. Clojure data structures are usually immutable and persistent (there are exceptions that are well marked thanks to strong conventions). When a data structure is "mutated" in Clojure, it produces what appears to be new one, but internally they share most of the data. For all practical purposes, clients of the data structure don’t need to know if there is structural sharing or how it works, as Clojure takes care of all the implementation details.

The functions operating on data structures have been broadly categorized as follows:

collections

A collection is the most generic categorization. The term "collection" has been used throughout the book to describe the input of the most generic functions, those operating consistently on a broad spectrum of data structures. Collections are essentially all built-in data structures in Clojure. The term has also been used to describe Java collections, although in that case "Java" is always present to avoid confusion.

sequences

A sequence is another broad categorization which describes a specific abstract data type [132]. A sequence is better described in terms of its properties, that we are going to see in the two chapters dedicated to them. Many data structures which are not sequential (offering direct lookup of an element) can be iterated to conform to the sequential interface. The group of functions related to the sequential interface is the largest.

lists

A list is a persistent data type that supports the sequential interface. Lists add new elements at the front of the list and require a full scan to access the last element from the list. Lists don’t appear in their own chapter but as part of the chapter about sequences.

Queues

A `queue` is a concrete data type implementing a first-in first-out semantic. Queues are sequential but they are not natively a sequence.

vectors

A vector is a persistent collection offering direct access through an index. Vectors are "associative", as each item in the vector is implicitly associated to its integer index. New items are added at the end of the vector. The functions in this chapter are specific vectors operations.

sets

A set represents a collection of unique and unsorted values. The chapter describes typical mathematical functions available on sets like union or intersection, plus other related functions. Sets are also available in a variant that maintains ordering based on comparison of values.

maps

A map is another kind of associative data structure, offering direct lookup by a key (not necessarily an integer like vectors).

Maps come in different flavors: array-maps maintain insertion order, do not require hashing, but are only suitable for small maps. Hash-maps are the most versatile, offering direct lookup through hashing of their keys. Sorted maps maintain ordering of their elements using a comparator.

Needless to say this categorization is neither perfect or definitive. Many functions overlapping in scope and range, have been included with the data structure they are most frequently used with.

Collection initializers

You might have noticed a pattern learning about data structures and their functions: Clojure usually offers a constructor taking any number of items (like `vector`, `hash-set` etc.) and another to build from an existing collection (like `vec`, `set`). The following table contains the most common constructors for the different collection types:

Table III.1. A summary of the initializers available for the different collection types

Type	Init from items	Init from coll	literal
vector	`vector`	`vec`	`[]`
list	`list`	`list*`	`()`
set	`hash-set`	`set`	`#{}`
map	`hash-map`	`zipmap`	`{}`
sorted-map	`sorted-map`	`n.a.`	`n.a.`
sorted-set	`sorted-set`	`n.a.`	`n.a.`
array-map	`array-map`	`zipmap`	`{}`
subvector	`n.a.`	`subvec`	`n.a.`
vector-of	`vector-of`	`n.a.`	`n.a.`
queue	`n.a.`	`n.a.`	`n.a.`

We can observe the following:

- array-map and hash-map share the same literal, as the transition between the two types is automatic and based on size. * `queue` is a short name for `clojure.lang.PersistentQueue`. The only way to build a `queue` is through the `(clojure.lang.PersistentQueue/EMPTY)` empty instance, then using into to add elements. See also peek for some relevant examples.

[132] An abstract data type (or ADT) describes the behavior of a data structure without relationship to its specific implementation. An overview of the most common ADTs is available at en.wikipedia.org/wiki/Abstract_data_type

8COLLECTIONS

This chapter groups together the functions that can be used consistently across different data structures. Their consistency is expressed by one of the following properties or combination thereof:

- Their performance is consistent across different collections with little or no appreciable difference.
- Even if there are performance differences, their use is idiomatic across different data structures.
- The function gets the work done so quickly that the trade-off of using it against the "wrong" data structure is acceptable.
- The function has been designed to be polymorphic and to operate on different collections.

The chapter is further divided into additional sub-chapters. We are going to first have a look at the basics functions to create, count or otherwise access a collection in the most general way. Next are the functions designed to be polymorphic. Finally, the last sub-chapter is dedicated to general purpose functions like

grouping, sorting, partitioning and others that are very common in every day use.

8.1 BASICS

8.1.1 into

`function` `since 1.0`

Listing 8.1. → *Collection copy, Transducers*

```
(into
  ([])
  ([to])
  ([to from])
  ([to xform from]))
```

`into` is a frequently used function in the Clojure standard library. `into` conjoins all items of a source collection "from" into a destination collection "to":

```
(into #{} (range 10))
;; #{0 7 1 4 6 3 2 9 5 8} ; ❶

(into [:g :x :d] [1 5 9])
;; [:g :x :d 1 9 5]       ; ❷
```

❶ `into` conjoins the content of a sequence into an empty Set.
❷ `into` conjoins the content of a vector into another which already contains a few elements.

The output collection can be (and most frequently is) of a different type than the source collection "from". `into` can be also used to apply a transducer chain "xform" while conjoining the data:

```
(into (vector-of :int) (comp (map inc) (filter e
;; [2 4 6 8 10] ; ❶
```

❶ `into` applies a transducer chain while conjoining the content of a sequence into a vector of primitives.

Contract

input

- "from" can be any seqable collection or `nil`.

- "to" should be such that `(coll? to)` is `true`: lists, vectors, sets, Queues or maps are all allowable target collections. The following non-persistent data structures are instead not supported: transients, arrays or generic Java containers. "to" can also be `nil` and in this case a default list is used as the target collection.
- "xform" is a transducers composition (or a single transducer) and is optional argument.

notable exceptions

- `ClassCastException: "X" cannot be cast to java.util.Map$Entry` appears when "to" is a collection of type map and the "from" input is not in a form that conj can process for maps. In this case the "from" collection should be structured using pairs or other maps, for example:

```
(into {} [[:a "1"] [:b "2"]]) ; ❶
;; {:a "1", :b "2"}

(into {} [{:a "1"} {:b "2"}]) ; ❷
;; {:a "1", :b "2"}
```

❶The example illustrates the correct format the input collection should present to `into` when the target collection is a map.
❷This second form is equivalent to the previous.

output

- The "to" collection object, with additional elements added from the input collection, if any. The elements are added to the "to" collection following conj semantic.

Examples

`into` can be used in those cases where the application requires the features of a specific data structure but the input is given in a different one. The reader is invited to check the for chapter for example. The Game of Life implementation in that chapter, uses a Set as the main data structure to enable easy checking of the presence of a cell in the set. But for returns a sequence, so `into` is used to transform the results back into a set for further computation. Similarly, the following `maintain` function applies sequence processing (like map or filter) making sure that the output remains the same as the input collection:

```
(defn maintain [fx f coll]  ; ❶
  (into (empty coll) (fx f coll)))

(->> #{1 2 3 4 5} ; ❷
  (maintain map inc)
  (maintain filter odd?))
;; #{3 5}

(->> {:a 1 :b 2 :c 5} ; ❸
  (maintain filter (comp odd? last)))
;; {:a 1 :c 5}
```

❶ `maintain` delegates processing of the collection to the input "fx" and "f" functions. It then uses empty to create an empty collection of the same type as the input collection. `into` is finally used to copy the processed content before returning.

❷ A Set gets incremented and even numbers are removed. The standard calls to map and filter would return a lazy-sequence instead.

❸ Another example using a map.

We could easily extend `maintain` to deal with the optional presence of a transducer chain. To show also other interesting data types in the Clojure standard library, we are going to use a queue as the input collection:

```
(import 'clojure.lang.PersistentQueue)

(def xform  ; ❶
  (comp
    (map dec)
    (drop-while neg?)
    (filter even?)))

(defn queue [& items] ; ❷
  (reduce conj (PersistentQueue/EMPTY) items))

(defn maintain ; ❸
  ([fx f coll]
    (into (empty coll) (fx f coll)))
  ([xform coll]
    (into (empty coll) xform coll)))

(def input-queue
  (queue -10 -9 -8 -5 -2 0 1 3 4 6 8 9))

(def transformed-queue  ; ❹
  (maintain xform input-queue))

(peek transformed-queue) ; ❺
;; 0
```

```
  (into [] transformed-queue) ; ❻
  ;; [0 2 8]
```

❶ `xform` is a simple transducer chain which assumes numbers as input.

❷ `queue` is a queue initializer. It uses reduce and conj to build a `PersistentQueue` type.

❸ `maintain` is now allowing an additional arity which takes just a collection and a transducer.

❹ `maintain` is used as before, this time just passing the transducer chain.

❺The output queue can be used as expected, with peek.

❻ `into` is used one last time to move between datatypes and inspect the content of the queue as a vector.

`into` and metadata

Metadata in Clojure are mainly dedicated to application (or library) implementers who need to decorate Clojure data structures with additional data (literally, data about data, hence metadata). Metadata enables decoration of a data structure without introducing special symbols that would otherwise be visible to all collection processing operation. For example, we could add a time-stamp value to a collection like this:

```
  (def v (with-meta [1 2 3] {:ts (System/currentTimeMillis)})) ; ❶
  (meta v) ; ❷
  ;; {:ts 1490773173006}
```

❶with-meta attach metadata information to a data structure. The map can contain any set of key-values.

❷meta is used to read metadata information out.

The standard library functions and macros usually play nicely with metadata, carrying them over as one would expect. `into` is taking the approach of preserving any metadata already present in the "to" collection, discarding what is instead part of the "from" collection:

```
(defn sign [c] ; ❶
  (with-meta c {:signature (apply str c)}))

(meta (into (sign [1 2 3]) (sign (range 10)))) ;
;; {:signature "123"}
```

❶The function `sign` attaches metadata to the incoming collection "c". The metadata values is a string conversion of the content in the input collection.

❷A simple experiment shows that the metadata resulting from calling `into` are coming from the target collection.

See also

- into-array is the equivalent operation when the target collection to return is a native array. `into` doesn't support arrays.

- conj can be used similarly to `into` when the input is not a collection but just a single item. `into` is based on the semantic of conj in relation to types and behavior.

Performance considerations and implementation details

=> `O(n)` **Linear in n number of items in the input collection**

`into` implementation is based on reduce, which is the dominant component in the performance profile. If the target collection implements `clojure.lang.IEditableCollection`, `into` copies the element in a mutable transient before transforming it back into a persistent collection. For large editable collections (such as vectors, sets, array-maps and maps) the speed-up is noticeable.

The following code executes two series of benchmarks, one using a vector as target, the other using a list (that doesn't support transients). Each benchmark uses an increasingly bigger collection and the results are plotted using the `clj-xchart` library [133]:

```
(require '[criterium.core :refer [quick-benchmar
(require '[com.hypirion.clj-xchart :as c]) ; ❶
```

```
(defmacro b [expr] ; ❷
  `(first (:mean (quick-benchmark ~expr {}))))

(defn sample [c] ; ❸
  (for [n (range 100000 1e6 100000)]
    (b (into c (range n)))))

(c/view ; ❹
  (c/xy-chart
    {"(list)"   [(sample '()) (range 100000 1e6
     "(vector)" [(sample [])  (range 100000 1e6
```

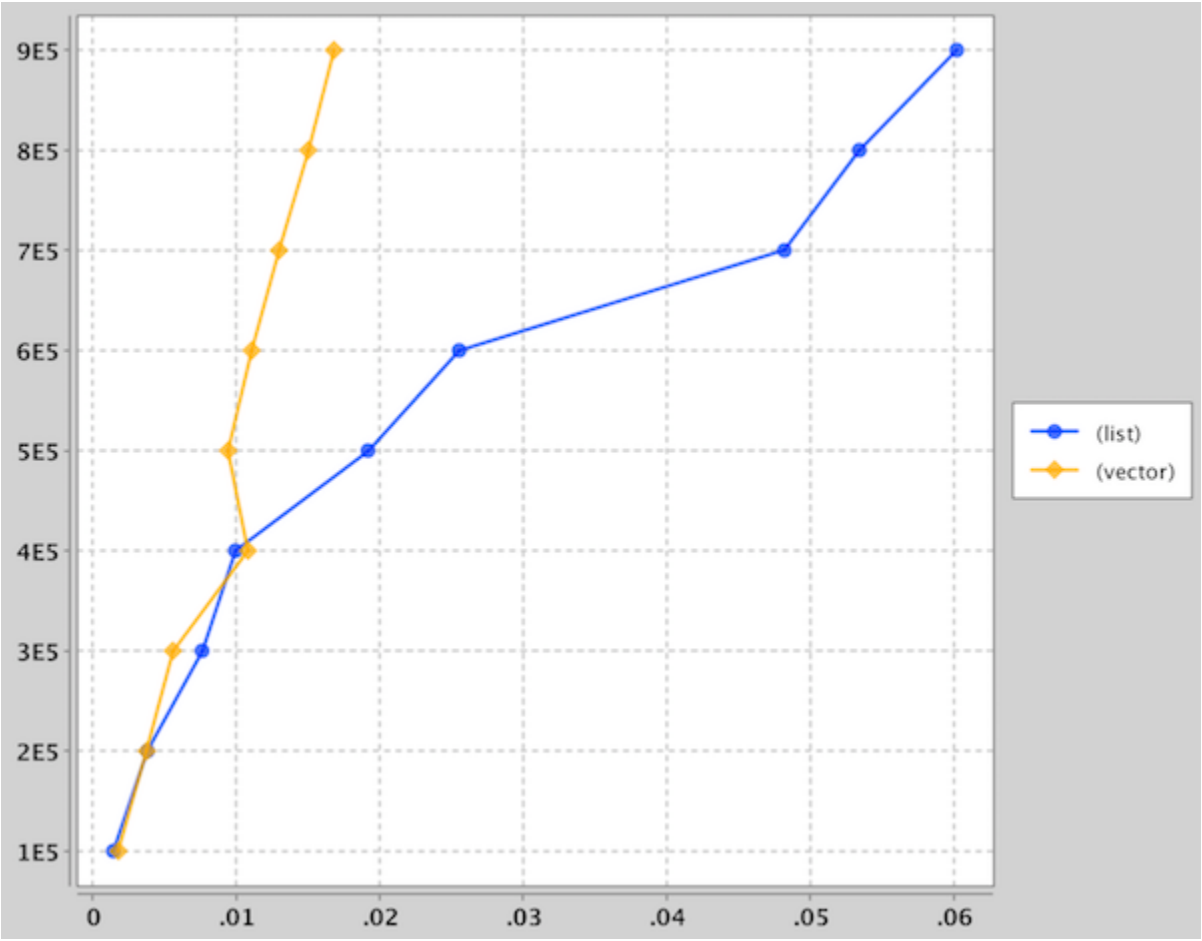
❶To require clj-xchart, make sure it is available on the classpath.

❷This macro extracts the relevant mean execution time from the map of metrics returned by Criterium.

❸`sample` takes the target collection as input and executes 10 loops with increasingly bigger inputs.

❹`view` and `xy-chart` are the only primitives necessary from clj-xchart.

Figure 8.1. `into` used with vectors compared to `into` used with a list. The x-axis is time in seconds.



The graph shows that `into` behavior is approximately linear (the lines are not perfectly straight because there are only 10 samples). But after the first few samples, the version with vector becomes faster, especially for larger collections.

The following benchmark shows a comparison between `apply merge` and `into {}` to merge multiple maps into a single one:

```
  (require '[criterium.core :refer [quick-bench]])
```

```
(let [maps (map #(apply hash-map  %) (partition-
  (quick-bench (apply merge maps)))
;; Execution time mean : 6.890232 µs

(let [maps (map #(apply hash-map  %) (partition-
  (quick-bench (into {} maps)))
;; Execution time mean : 6.126766 µs
```

❶ The benchmark creates a sequence of 10 maps of 5 keys each. The keys are distinct, so the resulting map will contain the original total of 100 keys.

❷ The benchmark reveals that there is not much difference between the two.

Both forms are idiomatic Clojure and they perform roughly the same, but `into {}` seems slightly better in terms of conveying the overall meaning of the operation with the visual aid of the map literal "{}".

8.1.2 count

`function` `since 1.0`

Listing 8.2. → *Size, Collections, Counting*

```
(count [coll])
```

`count` takes a collection as input and returns the number of elements it contains. `count` is an important operation in every programming language and Clojure puts specific care in making sure it works across all relevant data structures in the best possible way. Using `count` is simple:

```
(count [1 2 3])
;; 3
```

`count` eventually asks the input collection to count itself, a design decision that allows each data structures to provide the most efficient implementation. See the performance section for more information about how this decision affects speed.

Contract

input

- "coll" can be any collection type, record or struct. `(count nil)` is 0.

notable exceptions

- `UnsupportedOperationException` : when "coll" cannot be counted, for example for scalars like keywords, symbols, numbers and so on.

- `ArithmeticException` : if the number of elements in the collection is beyond `Integer/MAX_VALUE` , for example: `(count (range (inc Integer/MAX_VALUE)))` .

output

- returns: a `java.lang.Integer` representing the number of elements in "coll". The definition of "element" is type specific: it can be a tuple as in the case of maps or the number of fields in a record.

Examples

The need for counting is ubiquitous in programming and algorithmic. Here's a broad categorization of problems involving `count` in one form or another:

- Determining the largest/smallest of a set or group of items.
- Math formulas like the average, min or max.
- Determining "progress" over a process made of n-steps.
- Batching/splitting of items to a certain size.
- Determining frequencies, number of repetitions.
- Retry of requests some number of times.

One simple use of `count` can be found when checking consistency of command line arguments. The following example show a `copy` command implementation that requires at least 2 arguments (and optionally a third) to know which file to copy and where:

```
(require '[clojure.java.io :as io])

(defn- print-usage [] ; ❶
  (println "Usage: copy 'file-name' 'to-location

(defn- copy ; ❷
  ([in out]
    (copy in out "./"))
  ([in out dir]
    (io/copy (io/file (str dir in)) (io/file out

(defn -main [& args] ; ❸
  (cond
    (< (count args) 2) (print-usage)
    (= 2 (count args)) (copy (first args) (secon
    (> (count args) 2) (copy (first args) (secon

(-main "project.clj" "/tmp/copy1.clj") ; ❹
;; nil
```

```
  (-main "copy1.clj" "/tmp/copy2.clj" "/tmp/") ; ❺
  ;; nil
```

❶We print the expected usage of the command through this helper function.

❷The core logic is implemented in the `copy` function which contains 2 arities. The function excepts 2 or 3 arguments.

❸The entry-point function in Clojure, using the conventional "-" suffix. Clojure boostrap sequence can be instructed to search for this main signature function. Here we are using a cond instruction to `count` arguments.

❹For simplicity we are calling the `-main` function directly, otherwise we would have to compile and invoke the generated class requiring several other steps that are system specific. This version with 2 arguments assumes the current folder as the "origin" from where the input file should be loaded. Current folder in Java usually means the folder from where the JVM is invoked.

❺In this second version, we pass the third parameter. The additional "dir" param is used as the "origin" instead of the current folder.

How are lazy-sequence counted?

There are two main aspects involved in counting:

1. "pass over" each element in the collection to increment a counter.
2. "marking" of an element as already counted so it cannot be counted twice.

For data structures going through a phase of construction, counting can be done as each element enters the collection, resulting in a very fast `count` operation not requiring any additional enumeration (e.g. vectors).

Lazy-sequences are instead assembled on-demand at the time of consumption. It follows that there is no way for Clojure to hide the cost of counting at build-time, forcing a walk over the sequence to find where it ends. This is the primary reason why `count` on lazy-sequences is a linear operation. Please see the performance section for additional details.

See also

- counted? verifies if the given object supports the `clojure.lang.Counted` interface, which for Clojure collections means constant time `count`. Other notable Java types like strings and arrays are not `counted` but still constant time operations.

Performance considerations and implementation details

=> `O(1)` **Best case**

=> `O(n)` **Worst case (lazy sequences)**

`count` delegates the counting logic to the target collection whenever possible. In most of the cases, `count` is a fast constant time operation because the counting actually happens while the collection is constructed.

The following table summarizes `count` time complexity for different collection types:

Table 8.1. Summary of `count` performance by collection type.

Type	Counted?	Example	C
list	`true`	`(count '())`	(
vector	`true`	`(count [])`	(
queue	`true`	`(count (PersistentQueue/EMPTY))`	(
set	`true`	`(count #{})`	(
map	`true`	`(count {})`	(
sorted-map	`true`	`(count (sorted-map))`	(
sorted-set	`true`	`(count (sorted-set))`	(
array-map	`true`	`(count (array-map))`	(

arrays	`false`	`(count (int-array [1]))`	`(`
strings	`false`	`(count "abc")`	`(`
long-range	`true`	`(count (range 10))`	`(`
range	`false`	`(count (range 1e3))`	`(`
lazy sequence	`false`	`(count (for [i [1]] i)`	`(`
chunked	`false`	`(count (seq (for [i [1]] i)))`	`(`
seq sets	`false`	`(count (seq #{1}))`	`(`
seq maps	`false`	`(count (seq #{1}))`	`(`
seq sorted-set	`false`	`(count (seq (sorted-set 1))`	`(`

seq sorted-map	`false`	`(count (seq (sorted-map 1))`	(
seq vector	`true`	`(count (seq [1]))`	(
seq array-map	`true`	`(count (seq (array-map 1 2))`	(

The table describe a very good situation overall, with most collection types counting in constant time. There are a few notable linear time exceptions:

- lazy-collections and chunked sequences [134]. Note that `count` also realizes the lazy sequence when counting.
- long-range behaves differently from range created on other types (doubles for instance). `count` on a long-range is constant time, while it's not for other types.

- seq called on sets, maps, sorted-sets and sorted-maps are also linear-time (while the same on vectors or array-maps is instead constant time).

If counting is a necessary step for code that should also be as fast as possible, you should pay particular attention at lazy-sequences that are built as intermediate results after using seq as they often force a linear-time `count` operation.

8.1.3 nth

`function` `since 1.0`

Listing 8.3. → *Data Access, Indexed Lookup*

```
(nth
  ([coll index])
  ([coll index not-found]))
```

`nth` returns the value found at the specified index from the collection passed as argument:

```
(let [coll [0 1 2 3 4]]
  (nth coll 2))
;; 2
```

`nth` throws error in case the index doesn't exist (instead of returning `nil` (as get would instead do for sequential access). `nth` doesn't throw exception when a default value is provided:

```
(nth [] 2) ; ❶
;; IndexOutOfBoundsException

(nth [] 2 0) ; ❷
;; 0
```

❶ `nth` throws exception when the requested index is not available.

❷ The same operation returns successfully when a default value is provided.

`nth` is specifically designed for data structures offering random access lookup like vectors, sub-vectors, primitive vectors and transient vectors. It also handles native arrays and Java arrays with the same efficiency. `nth` can makes access to sequences although, in this case, less efficiently. It does not work at all on associative data structures (e.g. maps, sets and related ordered variants) for which a dedicate alternative exists (get).

Contract

input

- "coll" can be any collection type, excluding maps, sets (both ordered, unordered or created as transients).
- "index" can be any type of positive number up to `Integer/MAX_VALUE`. "index" is truncated to an integer if contains decimals: `(nth [1 2 3] 3/4)` is equivalent to `(nth [1 2 3] 0)`. `nth` using `Double/NaN` as "index" always returns the first element (when present).
- "not-found" is optional and can be of any type. "not-found" is returned when the requested index is not present.
- `nil` is accepted as a degenerated collection type. `nth` on `nil` returns `nil` unless "not-found" is provided.

notable exceptions

- `UnsupportedOperationException` is thrown when "coll" does not support any kind of indexed access.
- `IndexOutOfBoundsException` when the requested index is outside from the currently requested index.

- `IllegalArgumentException` when the given "index" is out of the integer range `(> "index" Integer/MAX_VALUE)` is `true`.

output

`nth` returns:

- The value found at "index", when available.
- In the absence of the requested "index", the "not-found" parameter (when present).
- `nil` when "coll" is `nil` and no default value is given.

Examples

Many data structures can be implemented on top of arrays. Hash-tables, for example, could store an object at the index generated by hashing it [135]. The next example shows how we could implement a basic hash-table on top of vectors.

The core concept in this data structure is the calculation of the hash number for each key-value pair. We can use the hash function to retrieve such number given the key and use the number to store the item at the index in the vector.

One problem we have with this solution is the pre-allocation of potentially large vectors (even for a single element). This is because the hash could be any number between -2^{31} and $2^{31}-1$.

Note

The reader should consider using a proper hash-map for any practical purposes. The implementation given here should give a good idea of the challenges related to creating robust data structures.

To reduce memory consumption, we are going to limit the hash to 2^{16} positive integers at the price of increasing the probability of collisions (which will still be relatively low and sufficient for the example):

```
(defn to-hash [n] ; ❶
  (bit-and (hash n) 0xFFFF))

(defn grow [upto ht] ; ❷
  (if (> upto (count ht))
    (let [t (transient ht)]
      (dotimes [i (- upto (count ht))] (conj! t
      (persistent! t))
    ht))
```

```
(defn assign [ht kvs] ; ❸
  (let [t (transient ht)]
    (doseq [[k v] kvs] (assoc! t k v))
    (persistent! t)))

(defn with-hashed-keys [args] ; ❹
  (map (fn [[k v]] (vector (to-hash k) v)) (part

(defn put [ht & args] ; ❺
  (cond
    (odd? (count args)) (throw (IllegalArgumentE
    (zero? (count args)) ht
    :else (let [kvs (with-hashed-keys args)
                ht (grow (apply max (map first k
          (assign ht kvs))))

(defn hashtable [& args] ; ❻
  (apply put [] args))

(defn fetch [ht k] ; ❼
  (nth ht (to-hash k)))
```

❶ The hashing function builds on top of Clojure own hash function. We need to restrict hash domain so it doesn't go beyond 2^{16} and it doesn't return negative numbers (which are not valid for vector indexing). We can use bit-and to apply both constraints at once.

❷ The `grow` function decides if we need to allocate more space in the vector holding the hash-table by checking the highest hash value from the entering keys. If one of the keys require an index in the vector that is beyond its length, the vector is conj-ed all the additional times required. The transient-persistent idiom is used to gain some speed in the process.

❸ The `assign` function assumes the vector holding the hash-table to be already of the right size. It uses again the transient-persistent idiom to write many key-value pairs efficiently.

❹ `with-hashed-keys` is a helper function to deal with transforming a list of arguments into a list of pairs where the first element (the key) is the hashed version.

❺ `put` is used both during initial construction and following updates. It first checks the arguments and then proceeds to potentially grow the hash-table (if there is not enough room for one of the arguments) and assigning the new elements.

❻ `hashtable` is used to build a new hash-table and it just calls `put` with an empty vector.

❼ `fetch` will make direct access to the hash-table to retrieve a specific value based on the hashed key.

Note the use of `nth` to fetch elements efficiently from vectorized hash-table. The hash-table implementation can be used as follow:

```
(def ht (hashtable :a 1 :b 2)) ; ❶

(fetch (put ht :c 3) :b) ; ❷
;; 2
```

❶Creates a hashtable with a few keys in it.

❷Shows access to the `:b` key via `fetch` and how to use `put` to add items to it.

See also

- get has a similar approach to `nth` with a specific focus to associative data structures. Although get can be used with vectors, there is no specific need to do that when the input is known to be a vector.
- vectors can also be used as a function with identical results to `nth`: `([1 2 3] 1)`. Vectors as a function can be handy and perform quite well but they don't support a default value.

Performance considerations and implementation details

=> `O(1)` **Best case**

=> `O(n)` **Worst case (mainly sequences)**

`nth` implementation depends on the type it is applied to. The `clojure.lang.Indexed` interface in Clojure indicates if a specific collection provides a constant time `nth` implementation. The following table summarizes `nth` time complexity for some important data structures:

Table 8.2. Summary of `nth` performance by collection type.

Type	Indexed?	Example	Complexity
vector	`true`	`(nth [1 2 3] 0)`	`O(1)`
subvector	`true`	`(nth (subvec [1 2 3] 0) 0)`	`O(1)`
vector-of	`true`	`(nth (vector-of :int 1 2 3) 0)`	`O(1)`
transient	`true`	`(nth (transient [1 2 3]) 0)`	`O(1)`
list	`false`	`(nth '(1 2 3) 0)`	`O(n)`
queue	`false`	`(nth queue 0)`	`O(n)`

long-range	`false`	`(nth (range 10) 0)`	`O(n)`
range	`false`	`(nth (range 1e3) 0)`	`O(n)`
arrays	`false`	`(nth (int-array [1]) 0)`	`O(1)`
strings	`false`	`(nth "abc" 0)`	`O(1)`
ArrayList	`false`	`(nth (ArrayList. (range 3)) 0)`	`O(1)`

The table shows 3 main groups:

1. Data structures (like vectors) supporting `clojure.lang.Indexed` interface. This class includes all types of vectors and it provides constant access time.

2. Sequential and lazy sequences not supporting `clojure.lang.Indexed` interface (like lists or ranges). `nth` performs linearly in this case.
3. Array and array based Java collections which are not `clojure.lang.Indexed` but are still constant time access.

The fact that `nth` is optimized for indexed lookup is clear looking at the table suggesting that `nth` should almost exclusively be used with vectors.

8.1.4 empty

`function` `since 1.0`

Listing 8.4. → *Collection Creation*

```
(empty [coll])
```

`empty` is a simple function that creates an empty collection of the same type as the input argument:

```
(empty (range 10))
;; ()

(empty (frequencies [1 1 2 3]))
;; {}
```

`empty` eventually delegates to the input collection the creation of an empty instance. Almost all collections (when `(coll? coll)` is `true`) provide this facility. Check the rest of the section for a few situations where `empty` doesn't work as expected (or throws exception).

Contract

input

- "coll" should be a collection type (`(coll? coll)` is `true`) but it can be of any other type (including `nil`).

notable exceptions

- `UnsupportedOperationException` when "coll" is a collection but `empty` is not supported. This only happens with records:

```
(defrecord Address [number street town])
(def home (Address. 12 "High st" "Alberta"))
(coll? home)
;; true
(empty home)
;; UnsupportedOperationException Can't create em
```

output

The type and value returned by `empty` changes based on the input collection. The following table shows the result of empty invoked on the most common Clojure and Java collections:

Table 8.3. Summary of the results of invoking `empty` on the

most common collection types.

Type	coll?	Example	Result
vector	`true`	`(empty [1 2 3])`	`[]`
subvector	`true`	`(empty (subvec [1 2 3] 0))`	`[]`
vector-of	`true`	`(empty (vector-of :int 1 2 3))`	`[]` (primitive)
list	`true`	`(empty '(1 2 3))`	`()`
queue	`true`	`(empty queue)`	(queue instance)
long-range	`true`	`(empty (range 10))`	`()`
transient	`false`	`(empty (transient [1 2`	`nil`

		`3]))`	
arrays	`false`	`(empty (int-array [1]))`	`nil`
strings	`false`	`(empty "abc")`	`nil`
ArrayList	`false`	`(empty (ArrayList. (range 3)))`	`nil`
Record	`true`	`(defrecord Test []) (empty (Test.))`	Exception
MapEntry	`true`	`(empty (first {:a 1}))`	`nil`
`nil`	`false`	`(empty nil)`	`nil`

Note that `queue` in the table means `(PersistentQueue/EMPTY)` constructor. The lower part of the table starting from transients shows some interesting results:

- transients do have an empty representation, but they are conventionally created only by the transient function only. This is a way to enforce use of transients (which are mutable data structures) only in explicit contexts. Allowing other functions to return transients would escape this form of control. Similarly goes for other mutable data structures like arrays and Java arrays.
- records have an identity of which attributes are an integral part. Creating an empty record from another would essentially create a different type. This breaks `empty` contract.
- A `MapEntry` is an implementation detail of how Clojure maps are iterated as sequential structures. A `MapEntry` only exists to contain a pair of key-value and it would not make sense to have an empty one.
- `nil` (along with other scalar types) does not throw exception but instead returns `nil`.

Examples

One of main goal of `empty` is to avoid conditionals to verify the type of a collection, so another can be created of the same type. To see an example of this, the reader is invited to check into,

where `empty` was used to prevent an input collection to be transformed into another type.

The following example shows what kind of conditional logic is replaced when using `empty`:

```
(let [coll [1 2 3]]   ; ❶
  (cond
    (list? coll) '()
    (map? coll) {}
    (vector? coll) []
    :else "not found"))
;; []

(empty [1 2 3])       ; ❷
;; []
```

❶cond checks the type of coll to decide which empty collection to return. Note that this is not an exhaustive list and would fail short using something like `(range 10)` as input "coll". `empty` is indeed implementing much more than this illustrative example.

❷ `empty` effectively "hide" the conditional avoiding the cumbersome task of listing all the collection types.

Creating new nodes while maintaining the original type is a typical problem when "walking" collections. Walking is typical of nested data structures and often involves changes that maintain the original structure. The following example, shows how we could iterate through an arbitrarily nested data structure so a function can be applied to certain elements. We expect the output to have the same structure of the input, including nodes and collections of the same type:

```
(defn walk [data pred f]            ; ❶
  (letfn [(walk-c [d] (map          ; ❷
            (fn [k] (walk k pred f)) d))
          (walk-m [d] (reduce-kv    ; ❸
            (fn [m k v] (assoc m k (walk v pre
  (cond
    (map? data) (walk-m data)
    (coll? data) (into (empty data) (walk-c data
    :else (if (pred data) (f data) data))))

(def coll {:a [1 "a" [] {:c "z"} [1 2]] :av 1N})

(walk coll (every-pred number? odd?) inc) ; ❺
;; {:a [2 "a" [] {:c "z"} [2 2]] :av 2N}
```

❶The logic to walk an arbitrarily nested collection is implemented by the recursive function `walk`. The function body is a cond instruction that differentiate between maps, other collection types and not collections. The need for distinguish between maps and collections is because they need to be iterated and assembled in a slightly different way.

❷`walk-c` is the local function dedicated to iterate collections. The function maps over the items of nested collection to apply `walk` again to each element.

❸Similarly `walk-m` is the function dedicated to iterate and rebuild hash-maps. Clojure maps when iterated like collections produce a sequence of tuples containing key-value pairs. So `walk-c` can't be used as is when a nested collection is of map type.

❹After dealing with hash-maps in the previous branch of cond, we can now make good use of `empty` to make sure that whatever is produce by `walk-c` (very likely a lazy-sequence) is always transformed back into the original collection type (for instance a vector).

❺We now invoke `walk` asking to increment all odd numbers any level deep.

See also

- into is often seen in conjunction with `empty` to build a specific type of collection.
- empty? are predicate functions to check if a collection contains elements. `(empty? (empty coll))` is `true` by definition.

Performance considerations and implementation details

=> `O(1)` **Constant time**

`empty` is a very simple function. The implementation is a single line to dispatch the call to the `empty` method present on all classes implementing the `clojure.lang.IPersistentCollection` interface. When the input argument doesn't implement the collection interface, `empty` simply return `nil`. The generation of the empty collection is extremely fast, usually in the order of few nanoseconds. For this reason `empty` is not particularly interesting in performance analysis.

8.1.5 every?, not-every?, some and not-any?

`function` `since 1.0`

Listing 8.5. → *Collection processing, Boolean logic*

```
(every? [pred coll])
(not-every? [pred coll])
(some [pred coll])
(not-any? [pred coll])
```

`every?`, `not-every?`, `some` and `not-any?` are four related functions that takes a predicate and a collection as arguments. They apply the predicate to the elements in the collection and combine the results using boolean logic operators like `AND`, `OR` and `NOT`.

`every?` for example, returns `true` if the predicate is `true` for each element in the collection (equivalent to the `AND` concatenation `(and (pred c1) (pred c2) (pred cN))` where `c1`, `c2`, .., `cN` are the items in "coll"):

```
(every? pos? [1 2 3 4]) ; ❶
;; true
```

❶All items in the collection are positive.

`not-every?`, `some` and `not-any?` operate similarly by combining predicate results with `NOT AND`, `OR` and `NOT OR`

respectively:

```
(not-every? neg? [-1 -2 0 -3]) ; ❶
;; true

(some neg? [-1 -2 0 -3]) ; ❷
;; true

(not-any? neg? [1 2 0 3]) ; ❸
;; true
```

❶ `not-every?` returns `true` because not all of the numbers in the list are negative.
❷ `some` returns `true` because at least one item in the collection is negative.
❸ `not-any?` returns `true` because it's false that there is at least one negative.

It's worth noticing the following relationships:

- `not-every?` is equivalent to negate the result of `every?` on the same form.
- `not-any?` is equivalent to negate the result of `some` on the same form.

- `every?` and `not-every?` answer the questions related to all of the items in the collection satisfying the predicate.
- `some` and `not-any?` answer questions related to at least one item in the collection satisfying the predicate.

Contract

input

- "pred" is a function of one argument returning any type. Even in case the predicate returns something different from a boolean, the returned value will be evaluated as `true` or `false` following Clojure conventions.
- "coll" must be a collection supporting the use of the seq function. Almost all of the Clojure collections are accepted by seq (notable exceptions are transients and the now deprecated strucs). Other Java types are also compatible, like arrays and other types from the `java.util.Collection` family.

notable exceptions

- `IllegalArgumentException` when the collection does not support the `ISeq` sequential interface (in the standard library, the only example are transients).

output

- `every?`, `not-every?` and `not-any?` returns either `true` or `false` (this fact is also evident from the naming convention of the question mark appended at the end of the name).
- `some` returns the first predicate result that that is not `nil`.

Examples

Functions like `every?` or `some` (and their complementary "not" variants) have some idiomatic usages in conjunction with other functions in the standard library. Their use avoids additional function calls, sometimes allowing more readable forms. For example, the following search by phone number (or any other unique attribute) can be written without filter or first:

```
(def address-book
  [{:phone "664 223 8971" :name "Jack"}
   {:phone "222 231 9008" :name "Sarah"}
```

```
    {:phone "343 928 9911" :name "Oliver"}])

(->> address-book                              ; ❶
  (filter #(= "222 231 9008" (:phone %)))
  first)
;; {:phone "222 231 9008" :name "Sarah"}

(some #(when (= "222 231 9008" (:phone %)) %) ad
;; {:phone "222 231 9008" :name "Sarah"}
```

❶ The threaded form filters the address book for the given phone number. It results in a sequence of the filtered items that we know contains a single entry.

❷ `some` is used to filter out the first item in the sequence that results in the predicate being not `nil`. We don't need the additional `first` call to unwrap the single returned result.

`every?` has also a few idiomatic uses in conjunction with other functions in the standard library. The following example shows how we can verify if all collections in a list have at least one element each:

```
(every? seq (list [:g] [:a :b]  [] [:c])) ; ❶
;; false
```

❶seq returns `nil` when the argument is an empty collection. `every?` returns `false` because at least one element in the input is `nil`.

In the next example we are using a Set as a function to check if all items in a collection are also contained in a set. The function `bingo?` checks if all numbers in the card are contained in today's numbers by using the `drawn` Set:

```
(def drawn #{4 38 20 16 87})

(def cards [[37 2 94 4 38] [20 16 87 19 1] [87 2

(defn bingo? [card] ; ❶
 (every? drawn card))

(map bingo? cards) ; ❷
;;(false false true)
```

❶ `bingo?` wraps the use of `every?` applied to a card.

❷Several cards contain elements from today's `drawn`, but only one card (the last in the list) contains all the extracted numbers.

Despite achieving similar goals, `some` is slightly different from the other functions in this section. The main differences are:

- It returns the result of applying the predicate to an element directly (there is no boolean transformation).
- It never returns `false`. If "el" is an element in "coll" and the result of `(pred el)` is `false`, `some` doesn't stop and continues with the next item.
- It returns `true` only when `true` is an element of the input collection (and the predicate returns `true`).
- It returns `nil` only if the predicate evaluated `nil` or `false` for all elements in the input.

The following example shows how we could use `some` to achieve two effects at once: pick the first of a valid set of options and second transform values using a hash-map:

```
(def prizes {"AB334XC" "2 Weeks in Mexico" ; ❶
             "QA187ZA" "Vespa Scooter"
             "EF133KX" "Gold jewelry set"
             "RE395GG" "65 inches Tv set"
             "DF784RW" "Bicycle"})

(defn win [tickets]                        ; ❷
  (or (some prizes tickets) "Sorry, try again"))
```

```
  (win ["TA818GS" "RE395GG" "JJ148XN"])     ; ❸
  ;; "65 inches Tv set"
  (win ["MP357SQ" "MB263DK" "HF359PB"])
  ;; "Sorry, try again"
  (win ["MP357SQ" "MB263DK" "QA187ZA"])
  ;; "Vespa Scooter"
```

❶The `prizes` hash-map contains a selection of ticket numbers and their associated prizes.

❷In the `win` function we can check all the tickets owned by a person. Using `some` we are constraining a potential double winner (because they have more than one winning ticket) to only get the prize for the first winning ticket. The translation from the ticket number to the associated prize happens by using the hash-map as a predicate function.

❸ `win` receives lists of tickets to verify if there are winners and what prize they get. If `some` returns `nil` it means there was no winning ticket in the list. We can check this condition with or. When no prize was found, the default message is printed.

every? and the vacuous truth

You might be surprised by the following:

```
(every? pos? []) ; ❶
;; true
```

```
(every? neg? []) ; ❷
;; true
```

❶This result shows that every element in an empty collection is a positive number

❷The second result shows the exact opposite: every element in the empty collection is negative.

It might seem an illogical or contradicting result, but it's a commonly accepted fact in mathematics that every statement about the empty set is `true`. The reason is that it's not possible to give counter-examples to any predicate applied to the empty set, so the answer is assigned by convention. See en.wikipedia.org/wiki/Vacuous_truth to know more.

See also

- every-pred is very similar to `every?` but it allows multiple predicates, combining all permutations with a logical "AND". Use `every-pred` if you need more than one predicate.
- some? should not be confused with `some` (without the question mark). `some?` is the equivalent of `(not (nil? x))` to verify if something is not `nil`.

Performance considerations and implementation details

=> `O(n)` **worst case, n number of items in the collection**

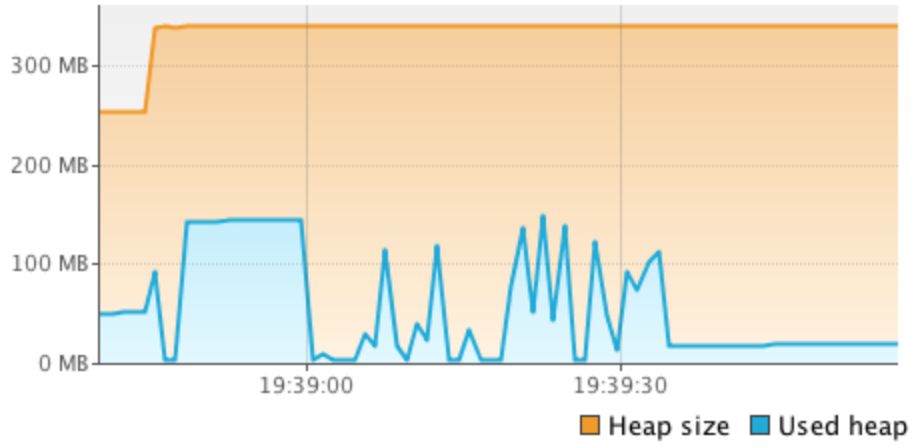
`every?`, `not-every?`, `some` and `not-any?` have similar performance profile. In the worst case, they needs to check the predicate against every item in the collection, producing a linear dependency. The best case scenario happens when the collection is empty.

The general case implementation is based on a tail-recursive recur. There is no need to hold on to head of the collection, so the implementation is memory-friendly. The following produces 1 billion iterations and takes some time to execute (depending on the hardware configuration):

```
(every? pos? (range 1 (long 1e9)))
```

Despite the amount of `long` integers created, the memory profile remains stable with a minimal garbage collection effort. The following image was taken during the execution of the line above and shows VisualVM [136] inspection of the JVM memory:

Figure 8.2. VisualVM showing JVM memory profile while running the example.



At "19:39:00" the execution kicks off after a small warm-up. The blue line representing the used heap is growing and released a few times until the evaluation returns at about "19:39:30". As the reader can see the used heap is quite small compared to the 330MB currently allocated (the orange line could grow up to 512MB maximum allowed size before going out of memory) and is gracefully claimed back a few times by the garbage collector.

8.1.6 empty? and not-empty

`function` `since 1.0`

Listing 8.6. → *Collections, Content verification*

```
(empty? [coll])
(not-empty [coll])
```

`empty?` and `not-empty` are simple functions to verify if a collection is empty or the opposite:

```
(empty? [])
;; true

(not-empty [])
;; nil

(empty? [1 2 3])
;; false
```

`not-empty` is not strictly the opposite of `empty?` as the missing question mark indicates that `not-empty` returns `nil` or "coll" itself to indicate logical true or false (instead of boolean types).

Contract

input

- "coll" can be a collection type, `nil`, string, array or generic Java iterable (e.g. `(instance? java.lang.Iterable coll)` is `true`). transients are not supported.

notable exceptions

- `IllegalArgumentException` when Clojure has no way to transform "coll" into a sequence. Apart from scalars (which are not collections, like numbers, symbols, keywords etc.), the exception is thrown with transients and the now deprecated structs.

output

`empty?` returns:

- `true` when "coll" is empty, or "coll" is an empty string, or `nil`.
- `false` otherwise.

`not-empty` returns:

- `nil` when "coll" is empty, or "coll" is an empty string, or `nil`.
- "coll" otherwise.

Note

A collection containing `nil` is not an empty collection! `(empty? [nil])` is therefore `false`.

Examples

`empty?` and `not-empty` are handy functions to compose a few idiomatic forms. For example the following snippet removes all empty strings (including `nil`) from a collection, but not strings with at least one character or beyond:

```
(remove empty? [nil "a" "" "nil" "" "b"]) ; ❶
;; ("a" "nil" "b") ; ❷
```

❶remove is used with `empty?` as a predicate.

❷Note that the string "nil" should not be confused with the `nil` constant.

The following example can be used to transform a string into a number. The same code without making use of `not-empty` would require a condition with multiple branches or throw exception. By using `not-empty`, we get rid of corner cases like `nil` or empty strings:

```
(defn digit? [s]
  (every? #(Character/isDigit %) s)) ; ❶

(defn to-num [s]
  (and
    (not-empty s)
    (digit? s)
    (Long/valueOf s))) ; ❷

(to-num nil) ; ❸
;; nil
(to-num "")
;; nil
(to-num "a")
;; false

(when-let [n (to-num "12")] (* 2 n)) ; ❹
;; 24
(when-let [n (to-num "12A")] (* 2 n))
;; nil
```

❶ `digit?` verifies if a string contains only numbers. Note that every? returns empty list when "s" is empty and `true` when "s" is `nil`.

❷and is short-circuiting, so as soon as one of the expression returns `nil`, there are no further evaluations. By using `not-empty` at the top of the chain, we make sure that following evaluations are not getting `nil` or empty string.

❸A few example using `to-num` showing what happens when the string is not properly a number. No exceptions are thrown and either `nil` or `false` are the only possible outputs.

❹ `to-num` behavior can be used in conjunction with when-let so we can proceed to treat "n" as a number only after the binding has been set.

Note how the use of `not-empty` with when-let achieves quite a lot of features in a small amount of code:

```
(def coll [1 2 3])

(when-let [c (not-empty coll)] (pop c)) ; ❶
;; [1 2]

(when-let [c (seq coll)] (pop c)) ; ❷
;; ClassCastException
```

❶ `not-empty` ensures the collection is not empty while `when-let` only assigns the local binding based on the results of `not-empty`. The body is then evaluated as a consequence with the original input collection.

❷ Comparison between `not-empty` and seq. `seq` is roughly equivalent to `not-empty` with an important caveat: `seq` transforms the collection into a sequential iterator which is not compatible with the following pop invocation.

See also

- clojure.string/blank? is a function dedicated to check for empty strings. `blank?` is more robust than `not-empty` because it considers white spaces. For example, if a string contains only spaces: `(clojure.string/blank? " ")` returns `true` while `(empty? " ")` returns `false`. If string processing is the main goal, the functions in `clojure.string` are in general a better fit.

- seq is widely used in the standard library to iterate over sequential collections. `seq` can also be used to check if a collection is not empty with `(seq coll)`. The main difference between `not-empty` and `seq` to check for empty collections is that `not-empty` returns the collection unaltered, while `seq` returns the sequential iterator over the input collection. This factor is important if `seq` or `not-empty` are then used in a if-let or when-let and the collection is further processed.

Performance considerations and implementation details

=> `O(1)` **Constant time**

Checking if a collection is empty with `empty?` is a fairly fast operation. `empty?` takes the same time independently from the number of elements in the collection. Although the use of `empty?` should not incur in performance penalties, if you need to check for emptiness in a tight loop there are ways to make it faster, for example using count or delegating the check to Java `isEmpty()` method. The following benchmark compares the approaches on a vector:

```
(require '[criterium.core :refer [quick-bench]])

(let [v (vec (range 1000))] (quick-bench (empty?
;; Execution time mean : 18.222054 ns

(let [v (vec (range 1000))] (quick-bench (zero?
;; Execution time mean : 7.026762 ns

(let [v (vec (range 1000))]
  (quick-bench (.isEmpty ^java.util.Collection v
;; Execution time mean : 4.663188 ns
```

❶ `empty?` is used on the vector "v".

❷ The same vector is verified using count to see if equals zero. This is roughly 50% faster.

❸ The Java method `isEmpty()` is present on all Clojure collections. In order to avoid the use of reflection, type hints should be used. This last approach is around 50% faster than `count`.

`empty?` and `not-empty` implementation is based on seq. This has been the subject of many discussions in the past [137]. If, on one hand, `seq` is not as fast as `count` to determine if a collection is empty, on the other it allows lazy-sequences to stay (almost) lazy: `seq` (and thus `empty?`) needs to realize only the

first element (or chunk). The following example illustrates the point:

```
(empty? (map #(do (println "realizing" %) %) (ra
;; realizing 0
;; realizing 1
;; [..]
;; realizing 31
;; false

(zero? (count (map #(do (println "realizing" %) 
;; realizing 0
;; realizing 1
;; [..]
;; realizing 99
;; false
```

❶ `empty?` is used to check if a lazy sequence is empty. The lazy sequence contains a side effect that prints a message on the screen. We can see 32 messages printed before the result is returned, the outcome of realizing the first "chunk" of a lazy (chunked) sequence.

❷ When `count` is used on the same input, the entire lazy sequence is realized.

[133] The clj-xchart library can be used to plot charts quite easily: github.com/hyPiRion/clj-xchart/

[134] Chunked sequences are a by-product of a performance optimization that Clojure implements behind the scenes.

[135] Hashing is the action of applying an hashing function to an object. The hashing function always converts the object to a number.

[136] VisualVM is a free JVM profiler that can be downloaded from visualvm.github.io

[137] Of the several threads on the mailing list about checking for empty collections with seq, this is one of the most articulated: groups.google.com/forum/#!topic/clojure/yW1Xw1dllJ8

8.2 POLYMORPHIC

This chapter contains a selection of those functions lacking a strong association with a specific collection type. For this reason, their behavior can change drastically depending on the input.

There is no perfect rule to decide what function should be in this section. The following are a few reasons why conj, get and contains? have been selected, while other functions are described elsewhere:

- Functions like assoc have similar polymorphic capabilities: `assoc` can be used on vectors as well as maps with a very different semantic. Despite this fact, `assoc` is predominantly used on maps and has a strong association with them. It would be weird for a chapter dedicated to maps not to contain `assoc`, hence why it's not appearing here.
- Functions like nth also works on many types (excluding hash-maps and sets). But the behavior of `nth` is the same when used on lists, vectors, lazy sequences and so on. `nth` can be considered polymorphic, but there is no drastically different behavior you should be aware of (except for the performance profile).

Note

More in general, the reader should see the classification implemented by the book as a tool to better visualize the content of the standard library and to help remember its content by association, naming or meaning.

8.2.1 conj

`function` `since 1.0`

Listing 8.7. → *Collection Conjoining, manipulation, inserting*

```
(conj
  ([coll x])
  ([coll x & xs]))
```

`conj` (an abbreviation for conjoining) inserts one or more elements into an existent collection. It takes the collection as the first argument:

```
(conj [1 "a" :c] \x) ; ❶
;; [1 "a" :c \x]

(conj (range 3) 99)  ; ❷
```

```
  ;; (99 0 1 2)

  (conj {:a 1} [:b 2]) ; ❸
  ;; {:a 1 :b 2}
```

❶The letter `\x` appears at the end of the input vector.
❷The number "99" appears at the beginning of the sequence.
❸The vector `[:b 2]` is interpreted by `conj` as a key-value pair for the input hash-map.

The example shows `conj` polymorphic behavior in respect of: where the element gets added (at the beginning or at the end) or some special input format (for example maps require vectors of two items). In general, `conj` delegates the accepting collection to perform the most efficient insertion.

Contract

input

- "coll" can be any collection type, but arrays and Java collections (such as `java.util.ArrayList`) are not supported. When "coll" is `nil` the emtpy list `()` is used by default.
- "x & xs" are one or more items to be added to "coll". They can be `nil`.

"x & xs" can be of any type for lists, vectors, sub-vectors, sets and sequences in general. There are restrictions for other types of "coll":

- If "coll" is a primitive type vector of type "T", then "x" needs to be of a type compatible with "T".
  Compatibility can be verified using one of Clojure type conversion operator (like int, char and so on) applied to "x" into the type "T", for example: `(int \a)` is a valid operation, so a (vector-of :char) can accept integers (only between 0 and 65535, which is the allowed range for `java.lang.Character`, an unsigned 16-bit value).

- If "coll" is a map type (excluding sorted), then "x" can be:
  1. A vector of two elements.
  2. An object implementing the `java.util.Entry` interface (one case being `clojure.lang.MapEntry` object obtained when sequencing with `(first {:a 1})`).
  3. Another map (like in `(conj {:a 1} {:b 2 :c 3})`).
  4. A record definition (for example `(defrecord Point [x y])`).
  5. An empty sequential collection like `#{}`, `""` or `()`. Note that list pairs are not supported, so `(conj {} '(1 2))` throws exception.
- If "coll" is a sorted set or sorted map then "x" needs to follow the compare compatibility rules between the type of the elements in "coll". For maps, the rule applies to keys only.

notable exceptions

- `IllegalArgumentException` when "coll" is a map type and "x" is a collection that is not supported by seq.

output

- returns: the result of adding "x" and any additional argument to "coll". The element will appear at the beginning of "coll" (for lists and lazy sequences), at the end of "coll" (for the vector family of types) or undetermined (for unsorted collections like maps or sets).

Examples

The most frequent use of `conj` is to insert a single item to vectors or sequences. But as seen in the contract, there are more ways to use `conj` for different collection types. The following examples show some of the most interesting cases:

```
(conj () 1 2 3)                ; ❶
;; (3 2 1)

(conj {:a 1 :b 2} {:c 3 :d 4})  ; ❷
;; {:a 1, :b 2, :c 3, :d 4}

(defrecord Person [name age])
(defrecord Address [number street zip])
(defrecord Phone [mobile work])

(conj                           ; ❸
```

```
    (Person. "Jake" "38")
    (Address. 18 "High Street" 60160)
    (Phone. "801-506-213" "299-12-213-22"))

  ;; #user.Person{:name "Jake", :age "38", :number
                  :street "High Street",
                  :zip 60160, :mobile "801-506-213
                  :work "299-12-213-22"}

  (def q (conj (clojure.lang.PersistentQueue/EMPTY
  (peek q)
  ;; 1
```

❶Multiple elements can be added in a single invocation.

❷This use of `conj` is equivalent to merge. The second map in the arguments is iterated sequentially and each key-value pair added to the first map argument.

❸Since records implement map semantic, we can use `conj` to merge multiple records together.

❹ `conj` is certainly an idiomatic way to push elements into a `PersistentQueue`.

During recursion, `conj` can be used to incrementally build results. The following example shows a function that writes text snippets to disk. The function expects a vector as input

containing titles and snippets as strings. The function writes to disk and outputs the list of the created files:

```
(require '[clojure.java.io :as io])

(defn- fname [dir path]          ; ❶
  (str dir "/" path ".clj"))

(defn write [examples root]      ; ❷
  (loop [[title forms :as more] examples
         files []]
    (if title
      (let [dir (str root "/" title)
            paths (map-indexed #(vector (fname d
        (doseq [[path content] paths]
          (io/make-parents (io/file path))
          (spit path content))
        (recur (nnext more) (apply conj files pa
      (map first files))))

(def examples
  ["add" ["(+ 1 1)" "(+ 1 2 2)" "(apply + (range
   "sub" ["(map - [1 2 3])" "(- 1)"]
   "mul" ["(*)" "(fn sq [x] (* x x))"]
   "div" ["(/ 1 2)" "(/ 1 0.)"]])

(write examples "/tmp")          ; ❼
```

```
;; ("/tmp/add/0.clj"
;;  "/tmp/add/1.clj"
;;  "/tmp/add/2.clj"
;;  "/tmp/sub/0.clj"
;;  "/tmp/sub/1.clj"
;;  "/tmp/mul/0.clj"
;;  "/tmp/mul/1.clj"
;;  "/tmp/div/0.clj"
;;  "/tmp/div/1.clj")
```

❶ `fname` joins together a folder path and a file name with extension "clj".

❷ `write` is a recursive function over the list of input examples. It writes snippets to disk accumulating their absolute path at each iteration. The accumulation process is a good candidate for `conj`.

❸map-indexed creates a list of tuples of number, path and content. doseq is destructuring on the output of `map-indexed` a couple of lines below.

❹The first side-effect is to create one or more folders corresponding to the path the file should be written to.

❺The second side-effect creates one file for each snippet, naming the file with a sequential number.

❻The function recurs stepping to the next 2 elements in the input using nnext. While invoking recur, results are pushed via `conj` to the accumulating vector of results.

❼We can see the list of files that were created after invoking the `write` function.

conj and cons

It is often a problem for Clojure beginners to understand the reason for the existence of two functions with a very similar name (and also very similar goal) like `conj` and cons. `cons` is a function of Lisp heritage [138] related to the concept of linking new elements to a list by creating an additional "cell" to hold the new item and the rest of the list. Clojure implements a similar design in which a new element wraps the existent list, so during iteration the last element added appears first. `cons` exists in

Clojure with the specific goal of creating a new "cons cell", so it makes sense for lists or sequences only. `cons` also works on vectors, but the vector is first transformed into a sequence, completely ignoring vectors peculiarities.

In order for Clojure to supports both sequences and vector-like collections (vector, maps and sets are all based on the same <<en.wikipedia.org/wiki/Hash_array_mapped_trie,hash array trie>> data structure), a function `conj` was designed to decouple the insertion logic from the collection implementation. `conj` delegates the input collection to add a new element in the best possible way. For sequences, it simply delegates back to `cons` logic, but for vectors it adds the new element at the end of the vector where a "tail buffer" is available (see vector for the implementation details). The pattern to delegate the input argument for the best way to implement an operation is a common theme in the Clojure standard library.

See also

- cons works on sequences (lists, ranges or lazy) and sequential objects (thus including vectors, sets, maps and many more). `conj` delegates to the internal `cons` method for list and sequences, so using `conj` is the best choice in the general case. `cons` has also other applications including building custom lazy-sequences.
- conj! is a special `conj` operation dedicated to transients. A transient is a special collection state in which a normally immutable data structure can mutate. A special set of functions (ending with an exclamation mark) is dedicated to transients.

- into, similarly to `conj`, adds elements from a collection to another. Prefer `into` for bulk-insertion of many items, as `into` is optimized for this kind of operation. Prefer `conj` to gradually build a collection one item at a time.
- assoc inserts a new value at some index. For maps the index is the key, for vectors it is the zero-based ordinal position in the sequence of items. `conj` on vectors will simply add to the end of the vector, with `assoc` you have more control about where the new element will appear (completely replacing the old one). Use `assoc` when the goal is to insert and replace an element at a specific position/key.

Performance considerations and implementation details

=> `O(1)` **(Near) Constant time (unsorted collections)**

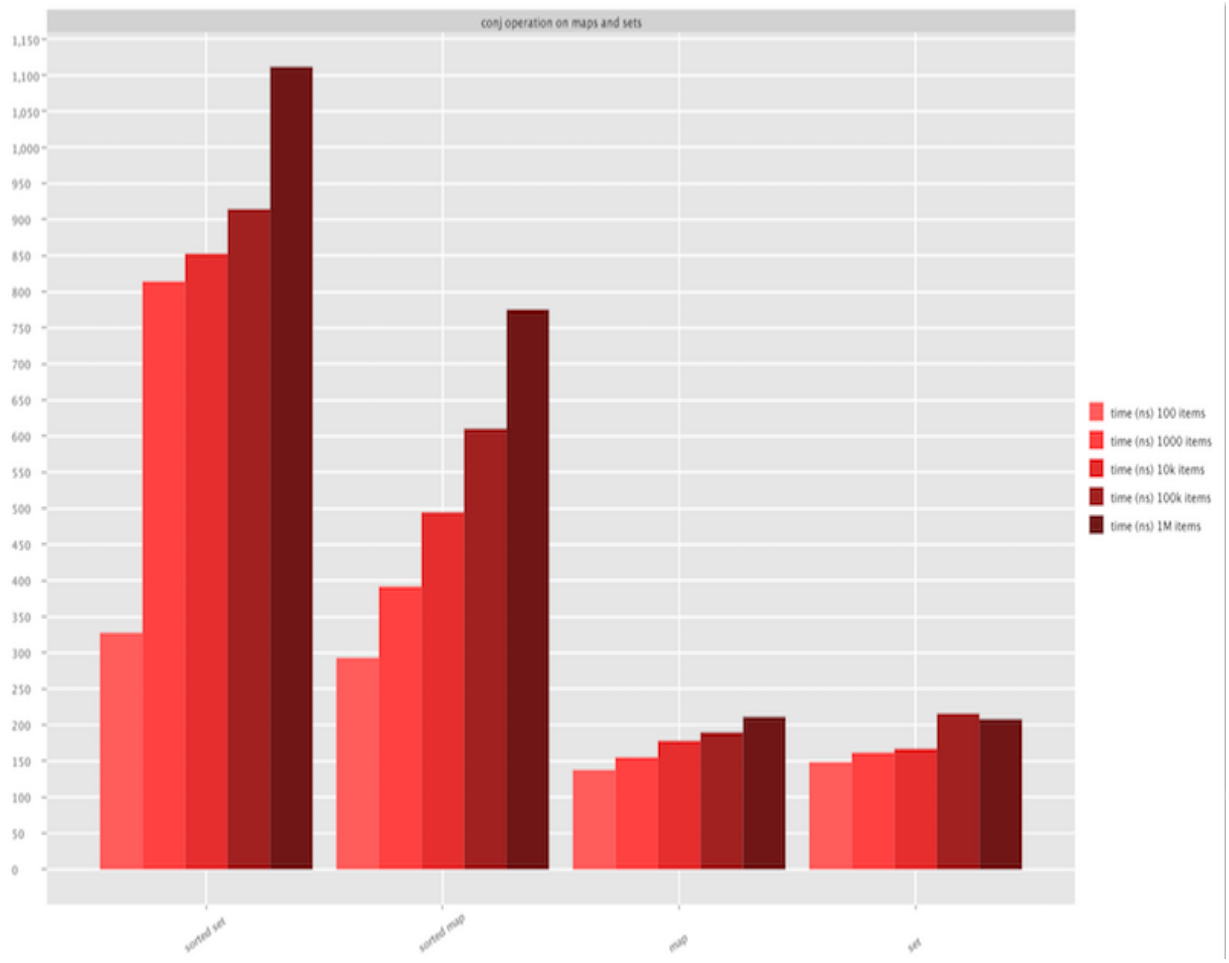
=> `O(log n)` **Logarithmic time (sorted collections)**

`conj` delegates the input collection to find the best way to insert a new element. `conj` implementation for maps and sets (ordered or not) is a very different operation than the one for sequential collections (vectors, lists, ranges and so on), where

the item can simply be appended without worrying about the existing content.

The following chart shows a comparison between maps and sets (ordered or unordered). `conj` has been used in all cases to add a non-existent keys into an increasingly bigger data structure (up to 1 million keys). For sorted collections, the new key has been selected to be roughly in the middle (using integer types) of the existing keys.

Figure 8.3. Efficiency of `conj` for sets and maps, ordered and unordered.

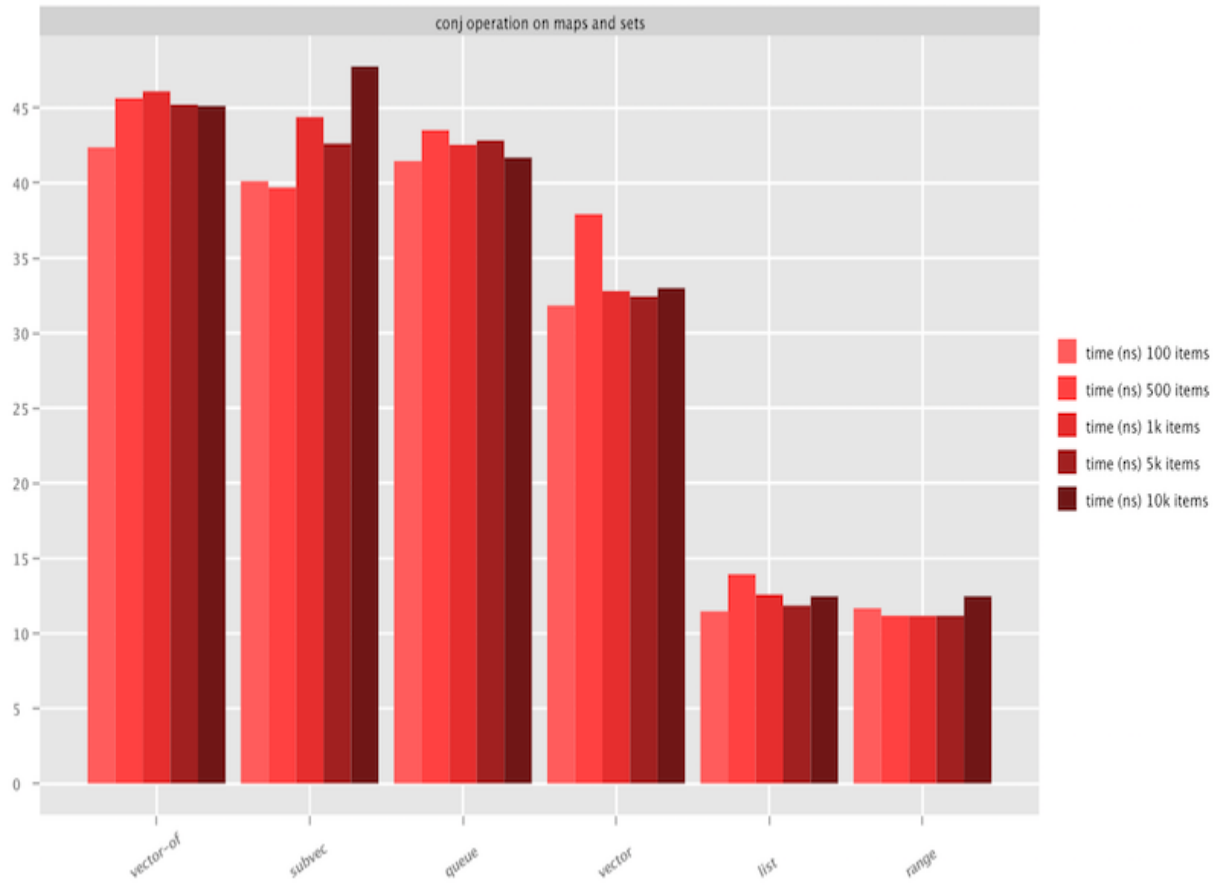


`conj` is `O(log N)` for Sorted maps and sorted sets. This is remarkable feature for an ordered collection. `conj` on unsorted maps (or sets) is instead near `O(1)` constant time (more precisely `O(log 32 N)` which is irrelevant in most practical cases).

The next chart repeats similar benchmark on some of the most used sequential types. As expected, `conj` performs faster on these types because there is no need to check on the existence

of the element, or to prepare space for the new key-pair in a specific place in the underlying data structure.

Figure 8.4. Efficiency of `conj` for sequential data structures.



The very fast results on lists and ranges is primarily because `conj` just creates a cons-cell, a small object that is linked to the rest of the collection. The reader should also keep in mind that

`conj` is not designed for bulk-inserts of many elements (for which other functions like into are better suited).

8.2.2 get

`function` `since 1.0`

Listing 8.8. → *Collection Access, Associative Lookup*

```
(get
  ([map key])
  ([map key not-found]))
```

In the group of the functions that make access to collections (the others being nth and find), `get` is specifically dedicated to maps (although it works on other types too):

```
(get {:a "a" :b "b"} :a)
;; "a"
```

`get` design is to avoid throwing exception, preferring `nil` when the collection type is not supported:

```
(get (list 1 2 3) 1)
;; nil
```

`get` offers a third argument which is returned when the element is not found or the collection is not supported:

```
(def colls [[1 2 3] {:a 1 :b 2} '(1 2 3)])
(def ks [-1 :z 0])
(def messages ["not found" "not found" "not supp

(map get colls ks messages) ; ❶
;; ("not found" "not found" "not supported")
```

❶We use map variable arity to call `get` with groups of items from each collection at at the same time.

This, along with returning `nil` instead of exception, makes `get` quite flexible for handling mixed type input:

Contract

input

The first argument called "map" is not limited to map types. It can also be:

- Set (ordered but not transient).
- A record created with defrecord.

- Classes implementing the `java.util.Map` interface (such as `java.util.HashMap` object instances).

Other arguments are:

- "key" can be any object. When the "map" argument is a `vector`, then "key" is restricted to positive integers up to `(Integer/MAX_VALUE)`. When "key" is beyond that range, the result of `get` can be difficult to predict, as "key" will be truncated to an integer with potential loss of precision.
- "not-found" is optional and can be of any type. "not-found" is returned when the requested index is not present.
- `nil` is accepted as a degenerated collection type. `get` of `nil` always returns `nil` unless "not-found" is provided.

When "key" is a number, "map" can additionally be:

- vector (including sub-vectors and native vector).
- A native Java array (such as those created with make-array or int-array).
- A string.

notable exceptions

- `ClassCastException` is thrown when "coll" is a sorted-map or sorted-set and "key" is not compatible with the content of "coll". To be compatible, the type of "key" and content of "coll" needs to be the same or have a suitable comparator. See the examples for additional information.

output

`get` returns a value which has a different meaning depending on the type of "coll":

- The value at key "key" when coll is a hash-map, sorted-map, array-map or their transient variation (with the exception of a sorted map that cannot be transient).
- "key" when coll is a Set, transient set or sorted-set and "key" is one of the elements in the set.
- The item at index "key" when "coll" is a vector, transient vector, sub-vector or native vector. "key" is assumed numeric in this case.
- The value at field "key" in a record instance, similarly to map access.
- In the absence of the requested index, value or item, the "not-found" parameter (when present).

- `nil` when "coll" is `nil` and no default value is given.
- `nil` in all other cases.

Examples

`get` contract contains a few exceptional situations worth highlighting as examples. The first has to do with sorted collections. When elements are added to a sorted collection, types are constrained by the existence of a suitable comparator to compare them. The reason for the constraint comes from the binary tree search algorithm implemented by sorted collections in Clojure. The search descends the right or the left branch based on comparison with the target object. The ordered tree allows O(logN) average search time at the price of some reduced flexibility:

```
(get (sorted-map :a 1 :b 2) :c "not-found")
;; "not-found"

(get (sorted-map :a 1 :b 2) "c" "not-found")
;; ClassCastException clojure.lang.Keyword canno
```

❶ `get` is searching for a non-existent key. The default value is thus returned.
❷ Another case of a non existent key, but with a type that cannot be compared to the keywords in the map.

The second interesting case in `get` contract has to do with transient collections. `get` can correctly access a transient map or vector, but cannot do the same on transient sets. This is a bug that should be fixed in the next Clojure releases [139]:

```
(get (transient {:a 1 :b 2}) :a)    ; ❶
;; 1

(get (transient [1 2 3]) 0)         ; ❷
;; 1

(get (transient #{0 1 2}) 1)        ; ❸
;; nil
```

❶ `get` is used on a transient map.
❷ `get` is used on a transient vector.
❸ `get` used on a transient set is unable to find an existing element.

The last surprising behavior happens on numerical keys exceeding the maximum value allowed for integers. `get`

implementation makes use of a lossy integer truncation with `(Number/intValue)` that can return unexpected results. Numerical keys are allowed for vectors, strings and arrays:

```
(+ 2 (* 2 (Integer/MAX_VALUE)))  ; ❶
;; 4294967296

(.intValue 4294967296)
;; 0

(get ["a" "b" "c"] 4294967296)   ; ❷
;; "a"

(get "abcd" 4294967296)
;; \a

(get (int-array [0 1 2]) 4294967296)
;; 0
```

❶ A sufficiently large number can mistakenly return valid indexes when coerced into an integer.

❷ `get` is used on vectors, strings and arrays using a sufficiently large number. The expectation would be for `get` to return `nil` instead.

`get` is the more resilient of the group of functions dedicated to element lookup in a collection. It works on every type (even scalars) at the cost of returning `nil` instead of throwing exception. It also accepts a `nil` collection as input, making it a good candidate when the target collection could potentially be `nil`:

```
(def mixed-bag [{1 "a"} [0 2 4] nil "abba" 3 '((

(map #(get % 1) mixed-bag) ; ❶
;; ("a" 2 nil \b nil nil)
```

❶ `get` works in almost all situations.

`get` also accepts objects implementing the `java.util.Map` interface, which is typical for a some Java interoperability scenarios. For instance, Java methods like `System/getProperties` or `System/getenv` are useful to gather information about the running environment and they return Java maps. The following example shows how we can search for interesting environment properties using `get`:

```
(defn select-matching [m k]
  (let [regex (re-pattern (str ".*" k ".*"))]
    (->> (keys m)
```

```
          (filter #(re-find regex (.toLowerCase %
          (reduce #(assoc %1 (keyword %2) (get m %

(defn search [k]
  (merge (select-matching (System/getProperties)
         (select-matching (System/getenv) k)))

(search "user")

;; {:USER "reborg"
;;  :user.country "GB"
;;  :user.language "en"
;;  :user.name "reborg"}
```

❶ `select-matching` searches for the given key (or portion of it) inside a Java map.
❷ The regular expression is built from the given key name and used to filter the matching keys regardless of the case.
❸ The following reduce operation builds a new Clojure map with the matching keys. The use of `get` is essential to access the Java map for the related value.
❹ `search` wraps access to the system properties and environment, merging them together before the actual selection.
❺ We can see the result of searching for "user", producing a Clojure map containing the matching keys. The content could be different on other systems.

The many ways to access maps in Clojure

`get` is possibly the most formal way to access a map in Clojure. Unless you care specifically about `get` handling of `nil` or Java interoperability, developers prefer others (usually more concise) ways. Those are illustrated below.

map as invokable function

"Map as a function" is one of the feature distinguishing Clojure from other languages. It works by using the map as the first item in a list, so it can be interpreted as a function call along with its parameters:

```
({:a 1 :b 2} :b) ; ❶
;; 2
```

```
({:a 1 :b 2} :c "Not found") ; ❷
;; "Not found"
```

❶Using the hash-map `{:a 1 :b 2}` as a function to access the `:b` key.

❷The optional second argument is returned when the key is not found.

hash-maps implement the `clojure.lang.IFn` interface, thus defining an `invoke` method that is used when the map appears in a callable position. It supports a second argument to be used as a default when the key is not found, exactly like the `get` function. `get` and "map as a function" even share the same implementation, both ending up calling the method `valAt()` from the `clojure.lang.ILookup` interface.

Keywords as invokable functions

keywords are also invokable functions similarly to maps. They accept a map as their first argument (and also other type of associative data structures like vectors). The keyword will then lookup inside the map keys for an instance of itself:

```
(:b {:a 1 :b 2}) ; ❶
;; 2
```

```
(:c {:a 1 :b 2} "Not found") ; ❷
;; "Not found"
```

❶keywords behave like maps when in callable position.
❷Like `get` they also support an optional second argument to return when the key is not found.

Keywords also implement the `clojure.lang.IFn` interface and delegate to the same `valAt()` method.

Java interop

Clojure maps are also instances of `java.util.Map`, so you can also access them with the `get(Object key)` Java method:

```
(.get {:a 1 :b 2} :b) ; ❶
;; 2

(.getOrDefault {:a 1 :b 2} :c "Not found") ; ❷
;; "Not found"
```

❶Note the "." dot before "get". It is syntactic sugar to invoke the Java method "get" on the object `{"a 1 "b 2}`.

❷Unlikely `get`, "map as a function" and "keyword as a function", `java.util.Map` interface doesn't have an overload of the `get()` method taking the optional second argument. There is intstead a specific method `getOrDefault()`.

Find

find is similar to the other methods seen so far but wrapping results in a `java.util.Map.Entry` object (which Clojure extends in its own `clojure.lang.IMapEntry` interface). Apart from wrapping the final result in a newly created map entry object, it shares the same implementation as `get`:

```
(find {:a 1 :b 2} :b) ; ❶
;; [:b 2]

(type (find {:a 1 :b 2} :b)) ; ❷
;; clojure.lang.MapEntry
```

❶Making access to the key in a map using find.

❷find does not support the optional second argument for default values. We can see that the returned type, which is printed like a vector, is instead a `MapEntry` object.

Choosing between one of the possible ways to access a key in a Map is a matter of how maps are used by different applications. Performances is less of a concern in this case, as Map access is overall a very fast operation independently from the method used.

See also

- find is similar to `get`, but it returns the entry map tuple (a vector of two element) instead of just the value. If you need to use the value but maintain the relationship to its key, `find` is the perfect choice.
- select-keys can access multiple keys at once and returns a map with those key-value pairs. Use `select-keys` to pick multiple values at once with their corresponding keys.

- nth accesses an element by index. `get` works on vectors too, but `nth` is specifically dedicated to that goal. `get` actually uses `nth` when the argument is a vector. Unless you are interested in `get` flexibility with `nil`, prefer using `nth` with vectors.
- get-in allows fetching values from withing nested maps (or the different kind of collections supported by `get`) without the need of nesting `get` invocations:

```
(def m {:a "a" :b [:x :y :z]})

(get (get m :b) 0)  ; ❶
;; :x

(get-in m [:b 0])  ; ❷
;; x
```

❶ `get` of `get` is used to access the vector at key ":b".
❷The same element in the vector can be reached with `get-in`

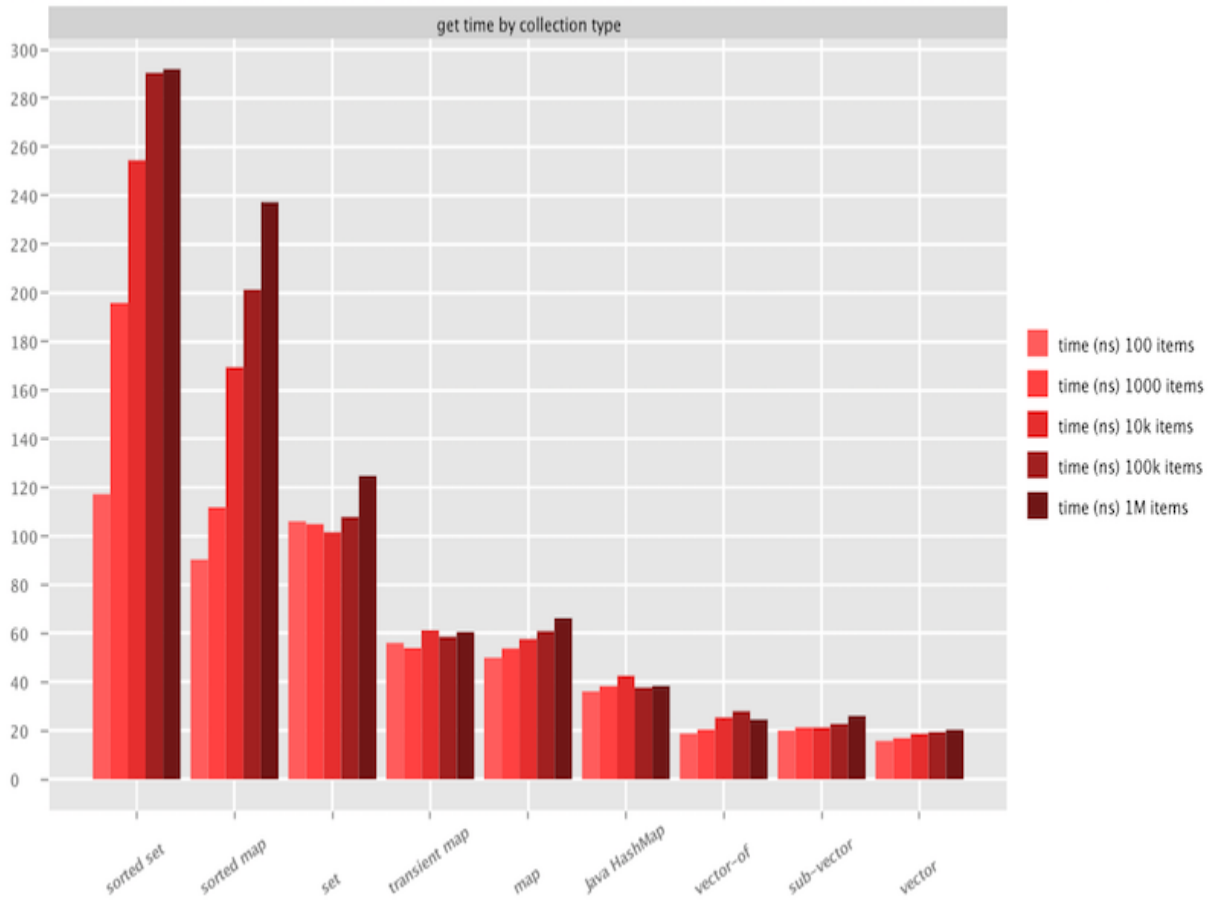
Performance considerations and implementation details

=> `O(1)` **Constant time (best case, unsorted collections)**

=> `O(log n)` **Logarithmic time (worst case, sorted collections)**

`get` performance is dependent on the input collection type. The following benchmark chart shows `get` used against different types of collections at increasingly bigger sizes. The benchmark is obtained by creating a map of randomized keyword keys (for example: `:12376, :47882` etc). The target key is roughly in the middle of the available values.

Figure 8.5. Access to several collection types using `get`.



Overall `get` is a fast operation that shouldn't raise particular concerns (times are in nanoseconds on a recent laptop). Sorted

collections are penalized, because access requires to compare the key at each branch of a balanced tree [140]. Also note how for sorted collection, `get` is `O(log N)` operation while is roughly constant time for other collections (it's not perfect constant time because persistent collections are implemented on top of a very shallow tree). Following sorted collections, sets are roughly 2 times slower than maps. Following that, vectors are 2 times faster for `get` access.

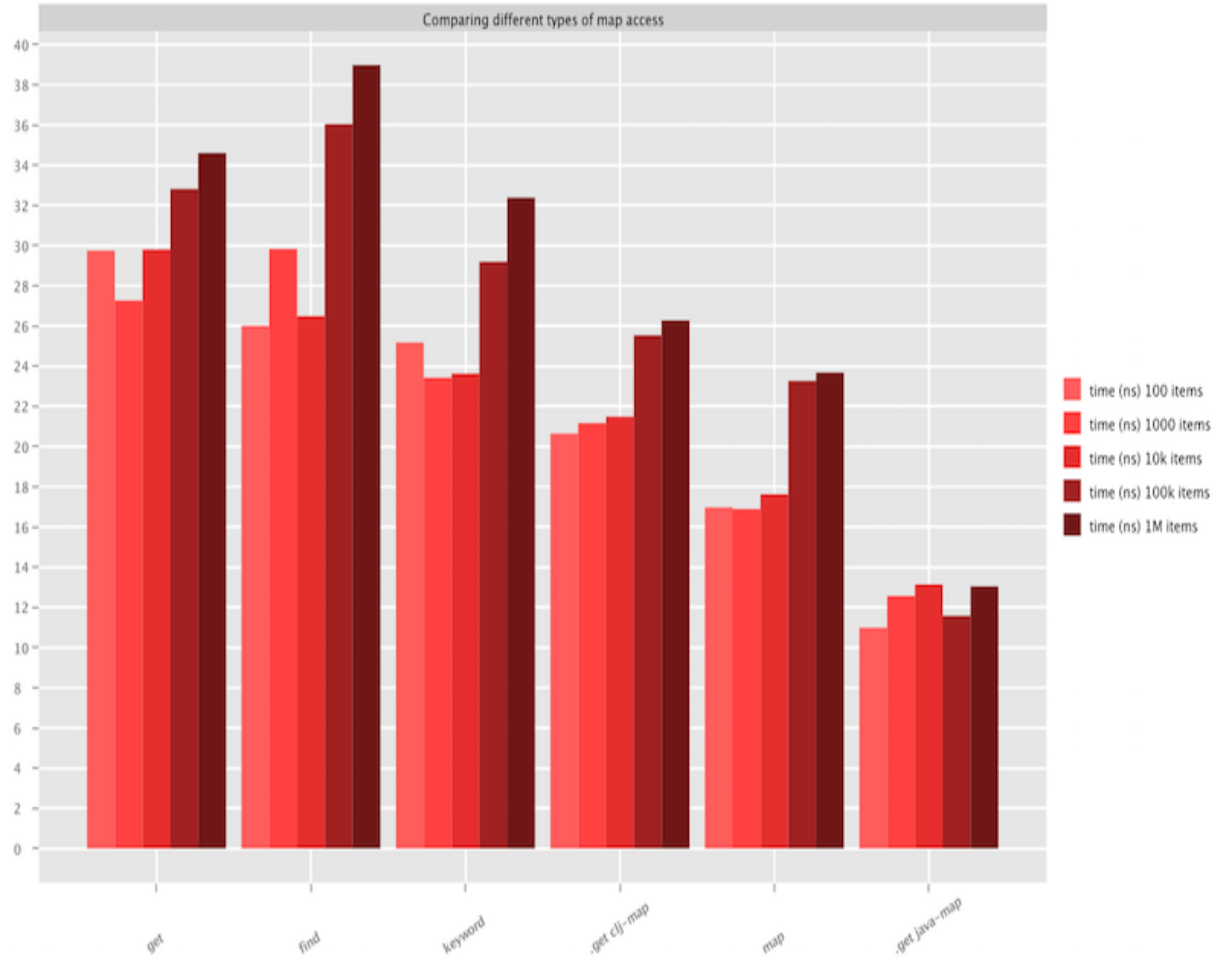
In terms of `get` compared to other ways to access maps (we illustrated the different ways to access a map before in the chapter) please see the following chart.

The chart compares 6 different ways to access a key in a map at increasingly bigger sizes (up to 1 million keys). Please note again that times are nanoseconds and overall we are talking about very fast operations. The bars in the chart, left to right, are:

- "get" is showing `get` access to the map. It scores around 30 ns average access.

- "find" is using the function find and despite the creation of the map-entry object, it performs roughly the same as `get`. This can be explained with the missing check for `nil` and the default value in find compared to `get`.
- "keyword" is using the key itself to access the map. It is 10% faster than `get`.
- ".get clj-map" is using the Java interop ".get" Java method to access a persistent hash-map in Clojure.
- "map" is using the map itself as a function to access the key.
- ".get java-map" is again using Java interoperability, but instead of accessing a Clojure map, it's showing what happens when we access a native mutable `java.util.HashMap` object.

Figure 8.6. Comparing different ways to access a map.



The chart essentially shows how good Clojure is performing against mutable Java HashMaps (for read-access). It also shows that using a map object directly as a function on keys is one of the best choices both for readability and speed. `get` remains a very good option when we want to take care of potentially `nil` maps without using an explicit condition or we want to access Java HashMaps.

8.2.3 contains?

`function` `since 1.0`

Listing 8.9. → *Collection Search, Item Inclusion*

```
  (contains? [coll key])
```

`contains?` verifies the presence of a "key" (element or index) in a collection supporting direct access lookup. Based on the kind of lookup implementation, it returns `true` or `false` if an element is found which is equal to the key, or an element is found at the index indicated by the key. Here's the most common use:

```
  (contains? {:a "a" :b "b"} :b) ; ❶
  ;; true

  (contains? #{:x :y :z} :z) ; ❷
  ;; true

  (contains? [:a :b :c] 1) ; ❸
  ;; true
```

❶A key equals to ":b" is found in the hash-map.
❷An element equals to ":z" is found in the set.
❸An element is present at index "1" in a vector.

Other types are supported, although their use is less common. The contract section goes into deeper details.

Contract

input

`contains?` main goal is to check for the presence of an element inside an "associative" data structures. In Clojure "associative" is a broad category including maps, vectors and by relationship with maps, records. sets are not implementing the `clojure.lang.Associative` interface, but they are supported by `contains?`. The following is an exhaustive list of all supported collections for the "coll" argument and related restrictions:

- map (sorted but not transient).
- Set (sorted but not transient).
- vector (including sub-vectors and native vector).
- A record created with defrecord.
- Classes implementing the `java.util.Map` interface (such as `java.util.HashMap` object instances).

- Classes implementing the `java.util.Set` interface (such as `java.util.HashSet` object instances).
- When "key" is a number, "coll" can additionally be a string or a native Java array.
- `nil` is accepted as a degenerated collection type.

When "coll" is a `vector` type, "key" can be any positive integer up to 2^{32}. When "key" is beyond that range, the result of `contains?` depends on the result of truncating key into an integer (which can be lossy). For instance:

```
(def power-2-32 (long (Math/pow 2 32))) ; ❶

(contains? [1 2 3] power-2-32) ; ❷
;; true
```

❶ `power-2-32` is 4294967296. It is beyond Java integer size of 32 bits.

❷ `contains?` internally makes sure "key" is an integer by truncating it with: `(.intValue 4294967296)` which equals 0.

notable exceptions

- `IllegalArgumentException` when `contains?` is not supported on a specific collection type. Noteworthy examples of collection throwing exception with `contains?` are `clojure.lang.PersistentQueue`, `java.util.ArrayList` and transients in general.
- `ClassCastException` is thrown when "coll" is a sorted-map or sorted-set and "key" is not compatible with the content of "coll".

output

`contains?` has a different meaning depending on the type of "coll". `contains?` returns `true` when:

- The "key" is present when "coll" is a hash-map, sorted-map, array-map or a Java map type.
- The "key" is in the set, when "coll" is a Set or sorted-set or a Java set type.
- The index "key" is present when "coll" is a vector, sub-vector, native vector or string. "key" must be numeric in this case.
- The index "key" is present when "coll" is a native Java array.

- There is a field "key" in a record instance, similarly to map access.

`contains?` returns `false` (or exception) when "coll" is `nil` and for all other types.

Examples

`contains?` distinguishes between the presence of a key with a `nil` value and the absence of a key in a hash-map. Other functions like get would return an unexpected result when used as conditions:

```
(def m {:a 1 :b nil :c 3}) ; ❶

(if (get m :b) "Key found" "Key not found") ; ❷
;; Key not found

(if (contains? m :b) "Key found" "Key not found"
;; Key found

(if-not (= ::none (get m :b ::none)) "Key found"
;; Key found
```

❶The map "m" contains a key ":b" with value `nil`.

❷Using get to check for the presence of the key would give the wrong result, as `nil` here means the value `nil` not the absence of the ":b" key.

❸ `contains?` is a better choice to check for the presence of a key.

❹To achieve the same effect with `get` with need a sentinel value `::none` as default to differentiate from a potential `nil`.

Expanding on the issue, let's have a look at the output of a group of electronic devices. The device presents a "nil" to indicate that the sensor did not send a response when it was requested. The following example shows how we could use `contains?` to verify the presence of a `nil` "key" in a Clojure hash-set:

```
(def sensor-read                          ; ❶
  [{:id "AR2" :location 2 :status "ok"}
   {:id "EF8" :location 2 :status "ok"}
   nil
   {:id "RR2" :location 1 :status "ok"}
   nil
   {:id "GT4" :location 1 :status "ok"}
   {:id "YR3" :location 4 :status "ok"}])
```

```
(defn problems? [sensors]
  (contains? (into #{} sensors) nil))   ; ❷

(defn raise-on-error [sensors]
  (if (problems? sensors)
    (throw (RuntimeException.
      "At least one sensor is malfunctioning"))
    :ok))

(raise-on-error sensor-read)            ; ❸
;; RuntimeException At least one sensor is malfu
```

❶ `sensor-read` is an example of a vector containing sensor data as maps. Two sensors returned no data resulting in a `nil`.

❷ `contains?` is used to build a predicate function `problems?` that can be used to verify the presence of `nil` in the set.

❸ In case of failed read of at least one sensor, it correctly throws an exception.

Note that the common approach to use a Set as a function passing `nil` as argument would not work in this case:

```
((into #{} sensor-read) nil) ; ❶
;; nil
```

❶Checking for the presence of `nil` in a set, using the set as a function generate an ambiguous result.

The `nil` returned in this case is ambiguous: it could refer to the fact that `nil` is a matching element as well as `nil` is not present in the set. `contains?` is the right function for this problem.

The many meanings of `contains?`

`contains?` often generates confusion when people approach it for the first time [141]. The confusion arises mainly from the meaning of searching a key in a vector (compared to a set or a map). The key "4" used in the example below, represents a possible index in the vector. The vector contains elements at indexes 0 to 3, hence `contains?` is returning `false` in this case:

```
(contains? [1 2 3 4] 4) ; ❶
;; false

(contains? [:a :b :c :d] :a) ; ❷
;; false
```

❶ `contains?` used on a vector requires the use of an integer as the second argument.

❷Array indexes should be in the integer range. When a type other than integer is used, `contains?` on a vector always returns `false`.

`contains?` on vectors only works in the presence of an integer as the second argument. Other types are accepted but

they always return `false`, adding to the confusion of certain expressions where the element is clearly present in the vector.

Clojure beginners expect `contains?` on vector to perform a linear search of the element in the vector instead of checking the presence of the element at the index and many discussions happened in the past around the option of changing name to `contains?` or to introduce a different function to scan collections for elements.

`contains?` is deliberately designed the way it is to prevent use in those contexts where it could generates a linear search. Other options are available in Clojure to perform linear scans of elements which are more explicit (for example some or `.contains` Java interop). This prevents abuse of standard library functions and data structures that would result in less than optimal performance.

See also

- some can be used to perform linear scans on vectors and other sequential collections: `(some #{:a} [:a :b :c])`. There are two restrictions compared to `contains?`: `some` needs a predicate, so the element to search usually end up inside a Set. The other problem is that it can't be used to search for the `nil` element. This can be fixed using equality as predicate: `(some #(= val %) coll)` works when `val` is `nil`.
- `.contains` does not belong to the standard library but is a similar Java method. Many Clojure data structures support the `java.util.Collection` interface with the exception of maps. `.contains` also works on strings and allows to search for substrings:

```
(.contains [:a :b :c :d] :a) ; ❶
;; true

(.contains {:a 1} :a) ; ❷
;; IllegalArgumentException No matching method f

(.contains "somelongstring" "long") ; ❸
;; true
```

❶Compared to `contains?`, the Java interop version `.contains` verifies the presence of the element in the collection (not the index).
❷ `.contains` does not work on hash-maps, but it works on hash-sets (ordered, unordered or transients).
❸On strings, `.contains` verify the presence of a substrings.

Performance considerations and implementation details

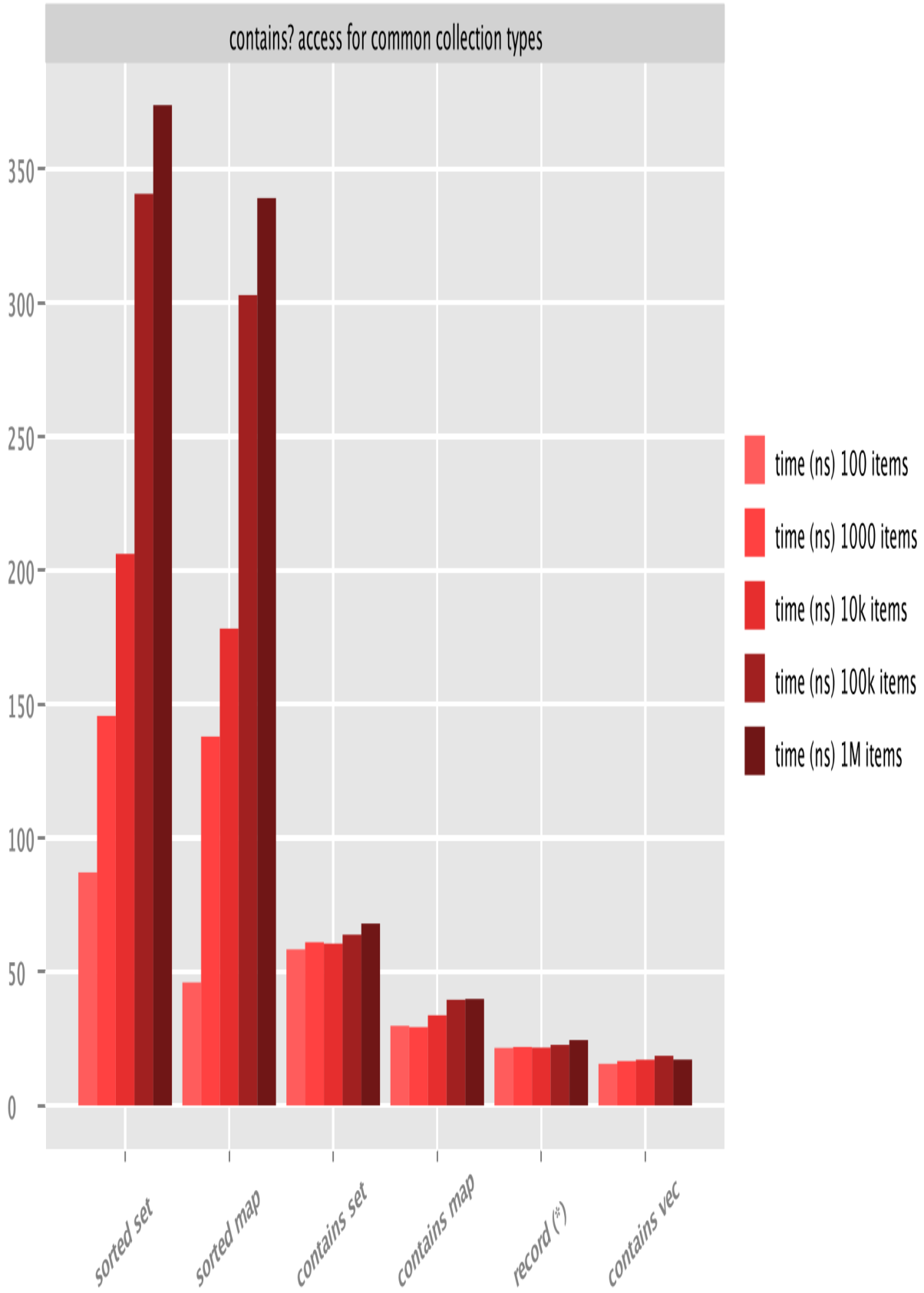
=> `O(1)` **Constant time (practical)**

=> `O(log N)` **Real**

`contains?` is designed around associative data structures (collections supporting direct lookup by index). Maps, sets and vectors are implemented as a variant of hash array mapped trie [142] with a slow increasing `O(log 32 N)` profile, while sorted collections are binary trees which is again logarithmic but with a different constant factor `(O log 2 N)`. The algorithmic class is in both cases logarithmic.

However, we refer to `contains?` performance as constant time in practice (a consideration valid for many other Clojure functions). We can verify the assumption in the following chart that shows that `contains?` performs in logarithmic time with different constant factors [143]:

Figure 8.7. `contains?` performs roughly constant time for all practical purposes.


contains? access for common collection types
350
300
250
200
150
100
50
0
sorted set
sorted map
contains set
contains map
record (*)
contains vec
time (ns) 100 items
time (ns) 1000 items
time (ns) 10k items
time (ns) 100k items
time (ns) 1M items

The chart shows that all collection types are slowly increasing in average access time as the number of items in the collection increases. Considered the scale, it's much easier to see it for sorted collections than for vectors. In terms of implementation, `contains?` is predominantly written in the Java side of Clojure where a simple entry point dispatches the call to the correct data type. For vectors, the implementation just checks if the requested index is within the length of the vector.

[138] `consel` was present in the early Lisp papers, then abbreviated to simply "cons"

[139] The problem with some transients being not compatible with collection operations is described in the following Jira ticket: clojure.atlassian.net/browse/CLJ-700

[140] Red-black trees are used in Clojure to implement sorted collections. Please see the following Wikipedia entry for more information: en.wikipedia.org/wiki/Red–black_tree

[141] This long thread from the mailing list summarizes many of the concerns related to `contains?`: groups.google.com/d/msg/clojure/bSrSb61u-_8/3AmJbVYOrzwJ

[142] HAMT (Hash Array Mapped Trie) is a data structure first presented by Phil Bagwell. See: lampwww.epfl.ch/papers/idealhashtrees.pdf

[143] Please note that records do not support more than 255 fields, so a fixed size record has been used throughout the test.

8.3 GENERAL PURPOSE

This chapter contains a collection of generally useful functions that can be used with different collections. Although they might have minor performance issues with some collection types, the code they produced is concise and they are powerful enough for most situations. Extensive examples and idiomatic usage has been illustrated for each function. Performance charts by

collection type have been added to those functions where it is still important to understand their trade-offs.

8.3.1 rand-nth

`function` `since 1.2`

Listing 8.10. → *Random Selection, Collections*

```
  (rand-nth [coll])
```

`rand-nth` selects a random element from a collection (excluding maps and sets) and returns it:

```
  (rand-nth (range 10)) ; ❶
  ;; 2

  (rand-nth "abcdefghijklmnopqrstuvwxyz") ; ❷
  ;; \b
```

❶ `rand-nth` is used to retrieve a random single digit number.

❷ `rand-nth` returns any of the element in the collection. In this case it returned "\b", but the result might be different on your computer.

The "randomness" provided by `rand-nth` as the same provided rand, a pseudo-number generator based on the standard Java library. For many practical purposes, `rand-nth` is sufficiently random and equally distribute the selection probability between the elements of the input collections. For more information about the limit of `rand-nth` please refer to the call-out section in rand.

Contract

`rand-nth` (as implied by the name) is built on top of nth and the collection needs to be "counted" in order to prevent exceeding the largest available index (please refer to count for more information about counted collections). A combination of the restrictions seen for nth and count applies to `rand-nth` as follows:

input

- "coll" can be any collection type excluding maps, sets (the exclusion extends to ordered, unordered or transients maps and sets). `rand-nth` also works for `java.util.ArrayList` or native arrays.
- Empty collections are not accepted as input.

- `nil` is accepted as a degenerated collection type and always returns `nil` as output.

notable exceptions

- `IndexOutOfBoundsException` : when the input collection is empty.
- `UnsupportedOperationException` : when "coll" is not a sequential collection, for example for scalar values (longs, chars) or unsupported data types like maps and sets.
- `ArithmeticException` : if the number of elements in the collection is beyond `(Integer/MAX_VALUE)` .

output

- A randomly selected element from the input collection.
- `nil` is returned when "coll" is `nil` or when `nil` was present in the collection and was selected.

Examples

`rand-nth` could be used to retrieve a random choice from typical enumerations like the sides of a dice, or a coin toss:

```
(defn roll-dice [] ; ❶
  (rand-nth [1 2 3 4 5 6]))

(defn flip-coin [] ; ❷
  (rand-nth ["heads" "tails"]))
```

❶Simple utility function `roll-dice` to return an equally probably number between 1 and 6.
❷Equally simple helper `flip-coin` to return "heads or "tails" with 50% probability each.

`rand-nth` is often found in games to avoid repetitiveness. It was used for example in the implementation of the "rock paper scissor" game in let to randomize the computer choice.

The following example shows a way to generate proverbs given a grammar. Although the grammar rules are very simple, it can still generate some realistic result:

```
(def article   ["A" "The" "A" "All"])
(def adjective ["nearer" "best" "better" "darkes
                "good" "bad" "hard" "long" "sha
(def subject   ["fool" "wise" "penny" "change" "
                "family" "proof" "necessity" "e
                "honesty" "no one" "everyone" "
(def action    ["is" "is not" "are" "are not" "h
```

```
(def ending    ["dying." "a dangerous thing." "a
                "stronger than words." "those w

(def grammar ; ❶
  [article adjective subject action ending])

(defn to-sentence [grammar] ; ❷
  (->> grammar
       (map rand-nth) ; ❸
       (interpose " ")
       (apply str)))

(defn generate ; ❹
  ([] (generate 1))
  ([n]
   (repeatedly n #(to-sentence grammar))))

(generate 5)

;; ("A bad experience is stronger than words."
;;  "A hard change are not stronger than words."
;;  "The nearer honesty are not nothing."
;;  "A good penny create stronger than words."
;;  "A good everyone is not a lot of noise.")
```

❶ `grammar` contains the recipe to assemble a sentence. Each part is a vector containing a selection of strings to randomly select from.
❷ `to-sentence` takes a grammar and proceeds to assemble the final string by joining all the parts together.
❸ `rand-nth` is used to pick a random choice for each part in the sentence. A better grammar would define weights by which each token is related to others.
❹ `generate` can be used to produce multiple proverbs using repeatedly.

As you can see, some generated sentences make more sense than others. Considering the amount of required code, this is still remarkable result. For anything more sophisticated, there are other more powerful and complicated techniques (see for example Markov Chains en.wikipedia.org/wiki/Markov_chain).

See also

- rand-int offers a mechanism to generate random integers within a range. The number can then be used to access an element in a collection at that index, which is essentially what `rand-nth` does. Use rand-int if you need to have control over the index generation.

- shuffle returns a random permutation of the entire collection not just one element. Use `shuffle` when the plan is making multiple sequential requests of random elements. The "shuffled" collection can then be iterated without the risk of retrieving the same element twice (something multiple calls to `rand-nth` would eventually generate).

Performance considerations and implementation details

=> `O(1)` **Best case**

=> `O(n)` **Worst case (some sequences)**

`rand-nth` has a simple implementation based on count, nth and rand-int. The performance profile of both `count` and `nth` is constant time for some collection types with a few linear time exceptions.

As with `nth`, `rand-nth` should be used mainly with vectors and avoided with sequences. The reader is invited to see nth section on performance for a complete explanation.

Given the presence of `count` in the implementation, please note that `rand-nth` realizes the entire sequence (when lazy),

even when the selected element is at the beginning of the input collection:

```
(def n (rand-nth (map #(do (println ".") %) (ran
;; prints 100 dots

(def n (nth (map #(do (println ".") %) (range 10
;; prints 32 to 100 dots
```

❶When `rand-nth` is used to pick an element at random from a lazy sequences, the entire sequence is realized. Also note that this is a linear operation that depends on the size of the input collection.

❷Using a combination of `rand-int` and `nth` we can avoid to realize the entire sequence on average. This doesn't eliminate the case: the entire sequence can still be fully realized when the selected element appears at the end.

If `rand-nth` needs to be used repeatedly on large lazy sequences, consider using combination of nth and rand-int to

avoid realizing the entire sequence when it's not strictly required.

8.3.2 shuffle

`function` `since 1.2`

Listing 8.11. → *Random Selection, Random Permutation, Collection Shuffle*

```
  (shuffle [coll])
```

`shuffle` takes a collection and returns a vector which contains a random permutation of its elements:

```
  (shuffle (shuffle (range 1 10)))
  ;; [9 3 2 1 5 7 6 8 4]
```

The algorithm used is the Fisher-Yates shuffling [144] shipped with the Java JDK. `java.util.Collections/shuffle` operates on `java.util.Collection` objects only, so almost all Clojure collections are supported with a few exceptions. Please refer to the contract section for additional information.

Contract

input

- "coll" can be any collection type excluding maps (hash-maps, sorted-maps, records), transients and native arrays.
- `nil` is not supported.

notable exceptions

- `ClassCastException` if the collection does not support the `java.util.Collection` interface.
- `NullPointerException` when `nil` is passed as an argument.

output

- returns: a vector containing a random permutation of the elements from the input collection.

Examples

One typical need for `shuffle` randomization is related to handling a spike of requests suddenly hitting an application. When that happens, we want to scale up the number of instances to handle the load. If such applications are configured to connect to internal services (for example the authentication

service), we don't want all instances to wake up at once and hit the same service together (generating a similar overloading effect downstream).

In this case `shuffle` can be useful to avoid the implicit ordering generated by configuration values, for example the list of hosts dedicated to authentication. The list of hosts is usually part of a configuration file and is read at start up before receiving the first request. `shuffle` can be used before the first call to spread the load across all the available servers.

one option to distribute the load across all available servers is using round-robin, a well known algorithm for resource allocation [145]. the following example shows how we could implement a round-robin approach where the first server is selected at random:

```
(defn round-robin [f hosts]
  (let [hosts (shuffle hosts) ; ❶
        idx (atom 0)]         ; ❷
    (fn []
      (f (nth hosts @idx))    ; ❸
      (reset! idx (mod (inc @idx) (count hosts))

(defn request [host & [path]] ; ❺
  (println "calling" (format "http://%s/%s" host
```

```
(def hosts ["10.100.89.42" "10.100.86.57" "10.10
(def get-host (round-robin request hosts)) ; ❻

(get-host) ; ❼
;; calling http://10.100.23.12/index.html
;; 1
```

❶ `round-robin` prepares internal state as part of the initial let block. One step of the initialization process consists of shuffling the list of hosts passed as arguments. This prevents other clients in a similar initialization state all to start requesting the first host in the list.

❷ The other part of the initialization contains the index of the host the next request should be made to.

❸ The request is made by invoking `f` on the host at the current index.

❹ Finally, the index is moved forward one element in the collection of hosts. mod makes sure we restart from the first host every time we reach the end of the list.

❺ `request` is the generic function to use to make requests. In a real scenario we would probably make actual http requests. We are printing to the standard output instead.

❻ `get-host` is assigned the function returned by `round-robin`. It can be now used by invoking it without arguments.

❼ Calling `get-host` prints the result on screen and returns the index in the collection of hosts to use the next request. The host is picked at random and it will be different if we re-initialize `get-host` var again.

The functional shuffle

Shuffling a collection of n elements is the problem of picking one of the n! (n-exponential) possible permutations. The naive solution that generates all the n!

permutations would quickly become impossible to handle (`O(n!)` is one of the worst complexities found in practical problems).

The common (imperative) approach is to use mutation of a data structure in place. There are many options, including one based on sorting (not extremely efficient at `O(n Log n)` complexity) and another that swaps randomly selected elements in place (this is the style adopted by the Fisher-Yates algorithm) which is linear.

A functional approach (not based on mutation) also exists [146], but its implementation is not as efficient as the imperative version (still featuring a pretty good `O(n Log n)` complexity).

Clojure pragmatic approach prefers the efficient implementation provided by Java, even if based on mutation (which is, by the way, isolated from the caller). There would be a small benefit in adding a functional approach to shuffling into the Clojure standard library, if not for the intellectual exercise.

See also

- rand are available to access randomly generated numbers.
- rand-nth randomly selects an item from a collection.

Performance considerations and implementation details

=> `O(n)` _Linear in n number of elements

As explained during the introduction, `shuffle` uses the implementation provided by the JDK which executes a single pass on the input collection, resulting in a linear behavior.

The input collection needs to be copied in a `java.util.ArrayList` so lazy sequences need to be completely traversed. A pure Clojure solution is possible but wouldn't remove the problem with lazy sequences, as the input collection needs to be counted and count is not lazy.

8.3.3 random-sample

`function` `since 1.7`

Listing 8.12. → *Random Selection*

```
(random-sample
  ([prob])
  ([prob coll]))
```

`random-sample` performs a random selection of elements from the collection passed as input. A number between 0 and 1 is used to control the probability that the item will be selected in the output. 0.5 can be used for example to set a 50% probability:

```
(random-sample 0.5 (range 10)) ; ❶
;; (1 2 7) ; ❷
```

❶ `random-sample` with 0.5 (50%) probability is used on a sequence of 10 items. Each element has 50% chances to appear in the output.
❷Results could differ when the same form is evaluated again.

Note that 50% does not mean that half of the elements will appear in the output, but "up to" 50% of the elements will definitely be.

When no input collection is provided, `random-sample` returns a transducer with the same characteristics. The following example simulates a scenario in which a coin is flipped repeatedly some number of times between 0 and "n":

```
(defn x-flip [n] ; ❶
  (comp (take n) (random-sample 0.5)))

(def head-tail-stream ; ❷
        (interleave (repeat "head") (repeat "tai

(defn flip-up-to [n] ; ❸
  (into [] (x-flip n) head-tail-stream))

(flip-up-to 10)
;; ["head" "head" "tail" "head" "tail" "head" "t
```

❶ `x-flip` is a function returning a transducer. The transducer applies a selection with 50% probability to the input sequence. It then limits the number of results to return.
❷ `head-tail-stream` produces an infinite sequence of alternating head-tail strings.
❸ `flip-up-to` applies the transducer to the infinite stream of head-tail strings.

Contract

input

- "prob" can be any number, although only the range between 0 and 1 is meaningful to calculate the probability. Any number below zero is considered 0%, while any number above 1 is considered 100%.
- "coll" is an optional collection input.

notable exceptions

- `ClassCastException` is raised when "prob" is not a number.

output

- returns: a lazy sequence of randomly selected items from "coll". Each item has probability "prob" to appear in the output.

Examples

`random-sample` can be used to implement a simple password generator. One fact to take into account when using `random-sample` is that the probability passed as argument influences the similarity of the output with the input. Observe the following:

```
(take 10 (random-sample 0.01 (cycle (range 10))))
;; (1 7 4 9 4 9 1 9 9 5)
(take 10 (random-sample 0.99 (cycle (range 10))))
;;(0 1 2 3 4 5 6 7 8 9)
```

❶A very low probability of 0.01 prevents many elements to be selected for the output, so the same range from 0 to 9 needs to be cycled several times before accumulating 10 elements.
❷A probability close to 1 on the other hand, produces a sequence that very closely mimic the input.

Using a low probability for `random-sample` requires a longer input sequence to produce items in the output. If we use cycle

we can repeat the same input range until `random-sample` picks enough elements for the output. We can use this recipe to create a random password generator:

```
(def letters (map char (range (int \a) (inc (int
(def LETTERS (map #(Character/toUpperCase %) let
(def symbols "!@£$%^&*()_+=-±§}{][|><?")
(def numbers (range 10))
(def alphabet (concat letters LETTERS symbols nu

(defn generate-password [n]
  (->> (cycle alphabet)     ; ❷
       (random-sample 0.01) ; ❸
       (take n)
       (apply str)))

(generate-password 10)
;; "C3pu@Y6Xhm"
```

❶The `alphabet` is the concatenation of all symbols, numbers and letters (upper and lower case).

❷cycle is used to create an infinite concatenation of the alphabet to itself.

❸ `random-sample` is used here with a low probability to create a sufficiently random sequence.

Controlling the sample size

`random-sample` might not be the best option when we need a specific sample size "k", as the size of the output sample depends on the probability factor. Other algorithms, like Reservoir Sampling [147] are a better choice. The following is a Clojure implementation of Algorithm "R" that could be used in place of `random-sample` to extract a random subset "k" from an input collection:

```
(defn random-subset [k s]
  (loop [cnt 0 res [] [head & others] s]
    (if head
      (if (< cnt k)
        (recur (inc cnt) (conj res head) others)
        (let [idx (rand-int cnt)]
          (if (< idx k)
            (recur (inc cnt) (assoc res idx head) others)
            (recur (inc cnt) res others))))
    res)))

(random-subset 5 (range 10000)) ; ❶
;; [8972 1623 1387 5184 3490]
```

❶Using Algorithm "R" we can be sure that the output sample contains the required amount of elements.

Algorithm R performs linearly and provides a `O(k)` constant memory allocation.

See also

- rand are the primitives used in many other functions in the standard library to deal with randomness. They generate random numbers that can be used as indexes or as a general source of randomness.
- rand-nth extracts a random element from an indexed collection. Use `rand-nth` when you are interested in a single random element from the input collection.
- shuffle returns a random permutation of the input collection. Use `shuffle` when you are interested in all elements.

Performance considerations and implementation details

=> `O(n)` **steps (worst case)**

=> `O(n)` **memory (worst case)**

`random-sample` is lazy, so the input sequence is evaluated when the results are actually used. How much of the input sequence is evaluated and retained in memory depends on:

1. The probability input parameter.
2. How much of the resulting sample is evaluated.

Assuming that we use the entire output sample, then `random-sample` needs to walk the entire input sequence, producing the

worst scenario of `O(n)` steps. `random-sample` does not retain the head of the sequence, so how much is loaded into memory is a function of the probability factor. A factor of 1.0 or above means all elements will appear in the output sample, hence the entire input sequence is loaded in to memory. If the output sample is not completely consumed (using for example take), we can limit the amount of steps required to walk the sequence, although this behavior is not completely deterministic. We can see why this is the case by quickly glancing into the implementation:

```
  (filter (fn [_] (< (rand) prob)) coll) ; ❶
```

❶ `random-sample` has a simple implementation making use of rand to decide when an element should be included or not.

Since there is no guarantee about how many iterations filter needs to find the next `true` predicate, we can't be sure about the number of steps required when we use take to sample a specific number of elements (potentially `O(n)` in the worst

case). Please refer to the call-out section about "Controlling the input size" for alternatives.

8.3.4 frequencies

`function` `since 1.2`

Listing 8.13. → *Counting, Statistics, Element Frequencies*

```
(frequencies [coll])
```

`frequencies` counts the repetition of the same item in a collection and returns the results as a map from the item (the key) into a count (the value):

```
(frequencies [:a :b :b :c :c :d]) ; ❶
;; {:a 1, :b 2, :c 2, :d 1}
```

❶ `frequencies` used to count the repetition of elements in a vector. We can see 2 occurrences of `:b` and `:c`. All other elements are unique.

`frequencies` is a handy tool for quickly calculate the distribution of a group with just a few line of code, as we are going to illustrate in the following sections.

Contract

input

- "coll" can be a collection (or `nil`) and is mandatory. All collection types are accepted (including defrecord) with the exception of transients and the deprecated defstructs.

notable exceptions

- `IllegalArgumentException`: when it's not possible to iterate the input sequentially. This happens for transients or scalars (numbers, keywords etc).

output

- Returns an unordered map. The map contains all items from the input collection as keys and a `long` number as a value. Each value represents the count of distinct elements found in "coll".
- An empty map `{}` when `nil` is passed as argument.

Examples

`frequencies` definition of "distinct" is based on single equal (=) semantic, allowing some types to be "bucketed" with the

same key. In almost all cases this is the most logical behavior, but worth remembering a few special cases:

```
(frequencies ['() [] (clojure.lang.PersistentQue
;; {() 3}

(frequencies [(byte 1) (short 1) (int 1) (long 1
;; {1 5}
```

❶Lists, vectors and queues all belong to the same equality category (please see = for more information).

❷Similarly, the same number expressed in all integer types is considered the same number.

As explained in the introduction, `frequencies` is useful to perform simple statistics. The following, for example, returns the top most frequently used words for a large book:

```
(require '[clojure.string :refer [split lower-ca

(defn freq-used-words [s]
  (->> (split (lower-case s) #"\s+")   ; ❶
       frequencies                     ; ❷
       (sort-by last >)                ; ❸
       (take 5)))                      ; ❹
```

```
(def war-and-peace "https://tinyurl.com/wandpeac
(def book (slurp war-and-peace))       ; ❺

(freq-used-words book)                 ; ❻
;; (["the" 34258] ["and" 21396] ["to" 16500] ["o
```

❶The content of the book as a string is piped through ->>. split takes the regular expression `#"\s+"` to split the text apart into single words. We then lower-case the string to remove unwanted differences.

❷ `frequencies` is used as is to calculate the word counts.

❸We then need to order the result by the highest frequency descending. It is done using sort-by and a suitable comparator.

❹The last step is to just take the top-most 5 words to show as results.

❺slurp can be used on local file paths as well as remote URLs. Be sure to have a working Internet connection before trying the example.

❻The result shows the most common articles and cnojunctions are at the top of the list.

Parallel frequencies

Counting distinct items is associative (it doesn't matter which order the keys are added together), so it is relatively easy to transform it into a parallel operation. Let's

revisit the word-count example using reducers instead of `frequencies`. The general design now consists of a merging operation (to bring together results coming from different cores) and a reducing operation to be used on each single core:

```
(require '[clojure.core.reducers :refer [fold]])
(require '[clojure.string :refer [blank? split split-lines lower-

(defn reducef [freqs line]  ; ❶
  (if (blank? line)
    freqs
    (let [words (split (lower-case line) #"\s+")]
      (reduce #(update %1 %2 (fnil inc 0)) freqs words))))

(defn combinef              ; ❷
  ([] {})
  ([m1 m2] (merge-with + m1 m2)))

(def war-and-peace "https://tinyurl.com/wandpeace")
(def book (slurp war-and-peace))

(defn freq-used-words [s]  ; ❸
  (->> (split-lines s)
       (fold 512 combinef reducef)
       (sort-by last >)
       (take 5)))

(freq-used-words book)
;; (["the" 34258] ["and" 21396] ["to" 16500] ["of" 14904] ["a" 10
```

❶ `reducef` is the reducing function used by fold. This will be used to reduce words on each processing unit. We designed the reducing function to push part of the computation ahead of `frequencies` down to a parallel chunks: splitting and lower-casing are now parallel as well.

❷ `combinef` is used by `fold` to merge processed chunks back together. This is essentially merge-with with an additional no arguments arity to provide a starting empty map `{}`.

❸ The main data pipeline is very similar to the sequential one. We additional take care of splitting the large text into lines ahead of the parallel part of the computation. This is what allows additional processing (such as word splitting and casing) to happen in parallel. `fold` is used here with the standard chunk size of 512, a quantity that could be set differently based on the size of the input text and number of processing cores.

The parallel word-count is only marginally faster than the sequential version, showing that the fork-join coordination cost should be taken into account for a simple computation like counting words on this file size and degree of parallelism (it was tested on 4 cores). For other scenarios, for example if more computation is required on the raw string or more cores are present, the parallel version could be visibly faster than the sequential one.

See also

- reduce is the fundamental construct used by `frequencies` to iterate over the input. Please refer to `frequencies` sources for any optimization needed to the reducing function, if that applies to your problem.
- group-by also produces a map as a result of a grouping rule. Use `group-by` when you want to split a collection based on some logic and have a key to access each group.
- partition-by produces a grouping but not as a map. The returned lazy sequence contains other sequences grouping the initial input based on a function passed as input. Use `partition-by` when the grouping needs to be iterated rather than accessed through a key.

Performance considerations and implementation details

=> `O(n log(n))` **steps**

The performance profile of `frequencies` is essentially linear with an additional shallow $O(\log_{32}(n))$ for the map access (tree-like persistent data structure) returning the results.

In terms of memory space, `frequencies` depends on the level of duplication present in the input. We go from having all input items in memory as keys (in the case of no duplicates `O(n)`) to constant space allocation for an input of the same repeating element.

The implementation uses transients internally to build the results, providing the fastest possible behavior.

8.3.5 sort and sort-by

`function` `since 1.0`

Listing 8.14. → *Sorting, Collection Sorting, Ordered Sequence*

```
(sort
  ([coll])
  ([^java.util.Comparator comp coll]))

(sort-by
  ([keyfn coll])
  ([keyfn ^java.util.Comparator comp coll]))
```

`sort` and `sort-by` are sorting functions in the Clojure standard library. They take an input collection containing items

of comparable type and return a sorted version based on a comparator (compare is the default if none is given):

```
(sort [:a :z :h :e :w]) ; ❶
;; (:a :e :h :w :z)

(sort > [18 43 3 0 9]) ; ❷
;; (43 18 9 3 0)
```

❶ `sort` is used to compare keywords using the default comparator.

❷To invert the ordering, we pass a different comparator. In this case of a collection of numbers, we can use any function taking 2 numbers and returning a boolean. However, comparison operators like `<`, `>`, `<=` or `>=` are the most common.

`sort-by` adds the option of passing a function to be invoked on each item before passing it to the comparator. The additional function allows `sort-by` to pre-process items or to transform their types before comparison:

```
(sort-by :age [{:age 65} {:age 13} {:age 8}]) ;
;; ({:age 8} {:age 13} {:age 65})
```

```
  (sort-by str [:f "s" \c 'u]) ; ❷
  ;; (:f \c "s" u)
```

❶ `sort-by` can be used to sort on a specific key of a map (instead of the entire map for example).

❷ str has been used to transform otherwise incompatible types into strings before comparing them. The transformation does not appear in the final result. Note that `(sort [:f "s" \c 'u])` would throw an exception in this case.

Contract

input

"coll" is valid for `sort` or `sort-by` when it is supported by seq (transients and scalars in general are not supported by `seq` while anything else is, including Java Iterable and arrays). The items inside "coll" are sortable when they are `nil`, identical? or belong to the following categories:

- They are numbers such that `(instance? java.lang.Number x)` is `true` for every "x" in "coll".

- They are comparable such that `(instance? java.util.Comparable x)` is `true` for every "x" in "coll".
- They provide a specific implementation of `compareTo()`. `compareTo()` is a method required by the `java.util.Comparable` interface.

"comp", as declared by the type hint in the function signature, must support the `java.util.Comparator` interface which implies the presence of a `int compare(Object o1, Object o2)` method available for execution. All Clojure functions implements such interface:

```
(map #(instance? java.util.Comparator %) [< > <=
;; (true true true true true)

(map #(instance? java.util.Comparator %) [+ - st
;; (true true true true)
```

❶Typical comparison operators can be used with `sort` as they implement the `java.util.Comparator` interface.

❷A little less obvious is the fact that all functions (more in specific, those extending the abstract class `clojure.lang.AFn`) are also comparators.

"keyfn" is mandatory argument for `sort-by`. It needs to be a function of a single argument that can accept the items in the input collection.

notable exceptions

- `ClassCastException` happens when an item is unable to compare to another. The exception usually reports what types are involved, for example: `java.lang.Long` cannot be cast to `java.lang.String`. To fix the problem, make sure that the item in the collection are compatible with the input rules explained above.
- `IllegalArgumentException` is thrown for incompatible collection types. transients and scalars are not accepted as collection.

output

- Returns: a sequence of ordered items following the given (or default) comparator logic.
- An empty list when the input is empty or `nil`.

Warning

When the input is a native array, `sort` and `sort-by` permanently mutate their input performing the sorting in place. Any other reference to the native array input will be affected as a side effect. For instance: `(let [a (to-array [3 2 1])] (sort a) (seq a))` will print `(1 2 3)` instead of `(3 2 1)` like for other persistent data structures.

Examples

Sorting is core operation in computer science and subject to specific research. It's essential for the solution of many problems and this book already used `sort` and `sort-by` in several examples. The user is invited to review the following:

- While discussing juxt, it was demonstrated how to compose multiple sorting criteria together, for example to sort a table of people records by `:age` and within that ordering, by `:sex` or other keys.

- Searching for the "best performing items" often involves sorting in descending order to extract the top "n" elements. We've seen this in calculating levenshtein distance of a string in the memoize chapter, or with letfn to find the biggest files by number of lines.
- In compare (a function strictly related to `sort`), we saw how to extend a custom record to support the comparable interface and then using `sort` on a collection of custom types.

Clojure's `<`, `>`, `<=` and `>=` can be used with `sort` or `sort-by` and collection of numbers:

```
(sort > (range 10)) ; ❶
;; (9 8 7 6 5 4 3 2 1 0)

(sort-by last >= [[1 2] [5 4] [3 4]]) ; ❷
;; ([3 6] [5 4] [1 2])
```

❶ `sort` takes an optional comparator argument before the input collection. Clojure extends functions so they can be used as comparators.

❷Similarly, `sort-by` accepts a custom comparator before the input collection. Equal element remains unsorted.

In the following extended example, we're going to build a parallel (and lazy) merge-sort on top of `sort` and `sort-by`. Basic `sort` and `sort-by` are eager: they need to load the collection into memory to perform reordering. This is good in many cases, but if the dataset doesn't fit in memory we need to operate differently.

Merge-sort is a popular sorting algorithm based on "divide and conquer" paradigm [148]: the idea is to split the initial collection, sort the smaller chunks and merge everything back in order:

- Take advantage of Clojure reducers to split and parallelize the processing of the initial collection.
- On each parallel thread, we are going to fetch a chunk of data from some external source, sort it and store it to disk.
- By having a relatively small amount "n" of concurrent threads, we can be sure that only "n" chunks of data are actively loaded into memory, never loading the entire dataset at once.

The algorithm has two separate phases: the first is about splitting the large input into smaller pieces, process them in parallel and store the ordered chunks to disk. The second part knows how lazily merge the ordered chunks so they appear as a

single sequence to the caller. Here's how we could go about implementing the first phase:

```
(require '[clojure.java.io :as io])
(require '[clojure.core.reducers :as r])

(defn- save-chunk! [data]             ; ❶
  (let [file (java.io.File/createTempFile "merge
    (with-open [fw (io/writer file)]
      (binding [*out* fw]
        (pr data)
        file))))

(defprotocol DataProvider             ; ❷
  (fetch-ids [id-range]))

(defn- process-leaf [id-range sortf]  ; ❸
  (-> (fetch-ids id-range)
      sortf
      save-chunk!
      vector))

(defrecord IdRange [from to]          ; ❹
  r/CollFold
  (coll-fold [{:keys [from to] :as id-range} n m
    (if (<= (- to from) n)            ; ❺
      (process-leaf id-range sortf)
      (let [half (+ from (quot (- to from) 2))
```

```
            r1 (IdRange. from half)
            r2 (IdRange. half to)
            fc (fn [id-range] #(r/fold n mergef
        (#'r/fjinvoke
          #(let [f1 (fc r1)
                 t2 (#'r/fjtask (fc r2))]
             (#'r/fjfork t2)
             (mergef (f1) (#'r/fjjoin t2)))))))

(extend-type IdRange ; ❻
  DataProvider
  (fetch-ids [id-range]
    (shuffle (range (:from id-range) (:to id-ran

(map (memfn getName) (r/fold concat sort (IdRang
;; ("mergesort-5429651713147139838.tmp"
;;  "mergesort-34393859464214136.tmp"
;;  "mergesort-82203554072858026.tmp"
;;  "mergesort-45086828924488256O4.tmp")
```

❶ `save-chunk!` is dedicated to the creation of a temporary file and storing of a sorted collection to disk. The exclamation mark is there to remember the side effect of calling this function: it stores on disk and returns a file handle.

❷ The `DataProvider` protocol is dedicated to potential client of our algorithm. Fetching the data from some external source is the only portion of the business logic that doesn't depend on merge-sort, so it's a good idea to prominently extract that out and offer an easy way to plug-in different data logic. The way this is done is further below through the use of extend-type.

❸ `process-leaf` is the core part of the algorithm, the computation that happens on each thread in parallel. It collects the operations that we need to perform on each chunk of data: fetch the IDs, sorting the resulting data, save the results.

❹ The reducers library offers different entry points. We decided to encapsulate the logic about how fold is going to behave in a new data type. `IdRange` is a record of two keys: `from` and `to`. They represent an integer range, something typically offered by databases as primary key for a table. The `defrecord` could be different to reflect a different system to store data which is not based on integers to uniquely identify records. What is important is that there is way to express a

partition of the full dataset without loading the actual data. An unique identifier is normally present in most systems. The second important aspect of the defrecord definition is extending the `CollFold` protocol from the `reducers` namespace. By extending `CollFold` we can use an `IdRange` type as the last parameter of a fold call and have the call to be routed to our custom implementation.

❺ The split happens with this if condition. If the chunk size is below the threshold (the chunk size determines how much data is loaded in memory in parallel before storing it to disk) we proceed with the sort operation. If the chunk is still too big, we fork-join the task of performing the same operation on the newly produced splits. Functions like `fjinvoke` or `fjfork` are the lowest Clojure primitives interacting with the Java side of the fork-join framework.

❻ Now that the `IdRange` type is defined, we can extend it to support the `fetch-ids` operation. This is performed on each `IdRange` object inside `process-leaf`. `fetch-ids`, when invoked on a `IdRange` object, delegates to the implementation specified with this `extend-type` instruction.

❼ `fold`, invoked on an `IdRange` instance, returns a list of file handles containing the sorted chunks.

Now that we are able to split, fetch and store sorted chunks of data, the second phase consists of merging back the files together without loading them all at once. Since the chunks are ordered, we can just look at the first item in each chunk to know which should come first. At every iteration, we proceed to lazily load the next chunk of data, as implemented by `sort-all`:

```
(defn sort-all                ; ❶
  ([colls]
   (sort-all compare colls))
  ([cmp colls]
   (lazy-seq
     (if (some identity (map first colls))
       (let [[[win & lose] & xs] (sort-by first
         (cons win (sort-all cmp (if lose (conj

(defn- load-chunk [fname]
  (read-string (slurp fname)))

(defn psort                   ; ❸
  ([id-range]
   (psort compare id-range))
  ([cmp id-range]
   (->> (r/fold 10000 concat (partial sort cmp)
        (map load-chunk)
        (sort-all cmp))))
```

```
(take 10 (psort (IdRange. 0 10000))) ; ❹
;; (0 1 2 3 4 5 6 7 8 9)
```

❶ `sort-all` assumes a collection of pre-sorted chunks (also collections) as input. It then iterates lazily the first item in each coll, searching the smaller/biggest elements and gradually forming a lazy sequence. Destructuring is quite helpful here to remove many occurrences of `first` and `rest`.

❷ We use `sort-by` on each iteration to find the next ordered element between all collections.

❸ `psort` is the main entry point (it means "parallel sort"). It offers a few defaults and prepares the call to `fold`. When the list of file handles is available, it lazily loads the file from disk and calls `sort-all`. Only enough of the content of each file is loaded into memory if we take from `psort`.

❹ Finally, we can invoke `psort` and see the results as expected.

Unless the client realizes the entire sequence, `psort` is never going to load the entire dataset in memory. `psort` additionally allows some configuration to take place, for example to change the comparator or provide specific logic to fetch the data.

See also

- compare is the default Clojure comparator. It returns a negative, zero or positive number based on comparing the two arguments.
- A custom predicate can be transformed into a comparator with comparator.
- sorted-set and sorted-map can be used to build ordering incrementally, as the elements arrive. They also accept a custom comparator.

Performance considerations and implementation details

=> `O(n log n)` **steps**

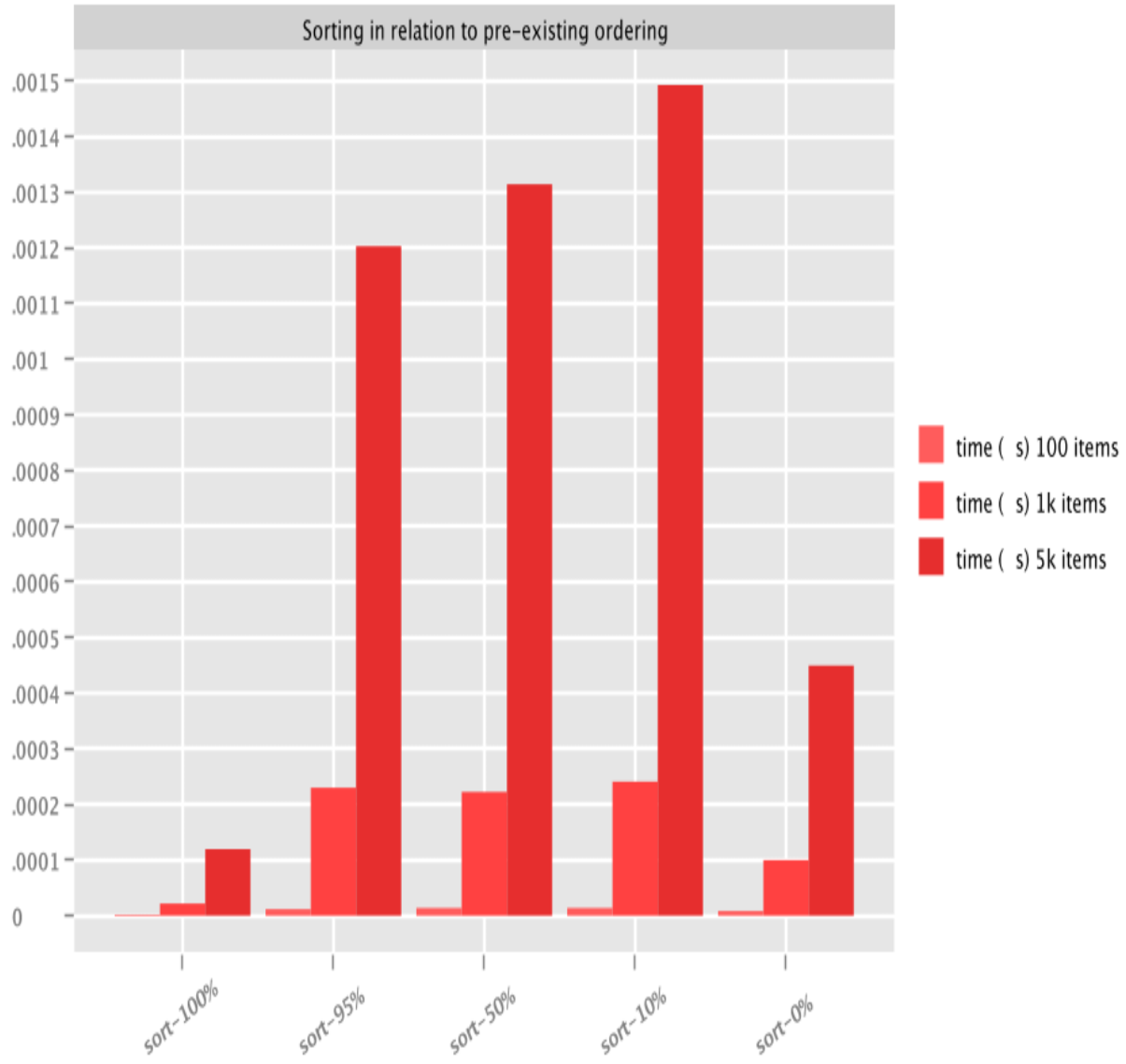
=> `O(n)` **memory**

The presence of `sort` or `sort-by` in a function should be reviewed for performance especially in the case of large datasets. For small collections this should not be a problem, but over a certain threshold (the ballpark number is around 100k entries) the linear logarithmic behavior of `sort` can result in a noticeable slowdown.

`sort` and `sort-by` are based on Timsort [149], the algorithm implemented by Java's `java.util.Arrays::sort()`. Timsort is a flexible sorting algorithm with `O(n log n)` worst case scenario and `O(n)` memory allocation.

Timsort looks for the presence of already sorted segments in the input collection. If the segment is below a specific threshold size, it uses insertion sort to bring the size up and then merge-sort to merge all the ordered segments. The dependency on already sorted segments is visible in the following chart, where `sort` runs against arrays of different sizes and level of pre-ordering:

Figure 8.8. `sort` is used to sort arrays with different level of ordering.



The chart shows that `sort` is only marginally faster on a collection which is 95% already sorted compared to a collection 10% sorted. Surprisingly, `sort` is faster when the collection is completely unsorted (alternating ascending/descending

contiguous pairs) compared to 95% sorted (but this last case is not common in real life scenarios).

8.3.6 group-by

`function` `since 1.2`

Listing 8.15. → *Collection Processing, Key Grouping*

```
  (group-by [f coll])
```

`group-by` groups the elements of an input collection based on the result of a function. The result identifies a key in the output hash-map while a vector contains the related values:

```
  (group-by first ["John" "Rob" "Emma" "Rachel" "J
  ;; {\J ["John" "Jim"], \R ["Rob" "Rachel"], \E [
```

❶first is called on each element in the collection, returning the first letter of each name. The letter is used as the key entry in the resulting map. If two items have the same initial, they are grouped together in the same `vector`.

Contract

input

- "f" is a required function argument. "f" is invoked with each item in "coll" and can return any result. It can be `nil` when "coll" is also `nil` (or empty).
- "coll" is also required argument. It can be `nil` or empty. If "coll" is not `nil`, "coll" needs to implement the `Seqable` interface such that `(instance? clojure.lang.Seqable coll)` returns `true` (transients are not supported).

notable exceptions

- `NullPointerException` when "f" is `nil` and "coll" is not empty.

output

- A `hash-map` containing the result of invoking "f" on each item in "coll" (as keys) and the grouping of the item in "coll" (as values). As a consequence, if a key exist in the map then its value must be a vector containing at least one element.

Examples

`group-by` is a flexible function with a broad application scope. We can use `group-by` to create dictionary-like data structures out of plain sequences, using the grouping function to decide how the values should aggregate. In conjunction with juxt, `group-by` allows to further restrict the grouping rules. The use of `juxt` determines the creation of a composite key vector:

```
(group-by (juxt odd? (comp count str)) (range 20
;; {[false 1] [0 2 4 6 8]
;;  [true 1]  [1 3 5 7 9]
;;  [false 2] [10 12 14 16 18]
;;  [true 2]  [11 13 15 17 19]}
```

❶An example of additional grouping constraint through the use of juxt.

As you can see, juxt forms vectors of results based on the functions passed as input. The range of 20 is split by odd/even numbers and then again based on the count of digits they have.

Let's use `group-by` now to search for anagrams. Anagrams are permutations of the same group of letters, which in our case represents the key once ordered:

```
(def dict (slurp "/usr/share/dict/words"))  ; ❶

(->> dict
  (re-seq #"\S+")   ; ❷
  (group-by sort)   ; ❸
  (sort-by (comp count second) >) ; ❹
  (map second)      ; ❺
  first)

;; ["caret" "carte" "cater" "crate" "creat" "cre
```

❶"/usr/share/dict/words" is a file present in most of Unix based systems. If you don't have one on your system, you can use the plain text version of War and Peace available from tinyurl.com/wandpeace (a Github link), or any other sufficiently large file of words.

❷The first step is to split the large string into each words. We can use re-seq to achieve the goal using a regular expression.

❸After the list of words is created, we feed it to `group-by` using sort. This creates an ordered list of the characters in each word, allowing `group-by` to see which ones is formed by the same letters.

❹Using `sort-by`, we can sort by the count of grouped words, starting from the larger group.

❺Now we eliminate the key and just keep the list of words (the second in each vector pair). The first list of 9 anagrams is visible in the output.

We can now extend the previous example to also force the presence of the letter "x" in a word using juxt. The example only needs a couple of changes:

```
(def dict (slurp "/usr/share/dict/words"))

(->> dict
  (re-seq #"\S+")
```

```
    (group-by (juxt #(some #{\x \X} %) sort))   ; ❶
    (filter ffirst)                             ; ❷
    (sort-by (comp count second) >)
    (map second)
    (take 3))

  ;; (["extra" "retax" "taxer"]
  ;; ["examinate" "exanimate" "metaxenia"]
  ;; ["axon" "noxa" "oxan"])
```

❶Along with sort, we use juxt to require additional rules for a word to enter an anagram group. some is used with a set literal as a predicate to verify if the letter "x" exists in the word.

❷We also need to eliminate all keys where the "x" component was not found. When a map is used as the input for filter it decomposes into a sequence of vector pairs, where the first element is the key (which is again a vector pair containing the result of the `some` operation in first position). By taking the ffirst we are taking the first of the first item from the key. If that is `nil` then the word doesn't contain the letter "x".

See also

- partition-by does not produce a map, but it creates nested sequences inside the input collection based on the changing results of a function. `partition-by` creates a sequential grouping, where nested parenthesis separate the original items without a key. Use `partition-by` when you don't group access by key or when you need laziness.
- frequencies also returns a map, but where the original input items are the key and the values are the number of their repetitions.

Performance considerations and implementation details

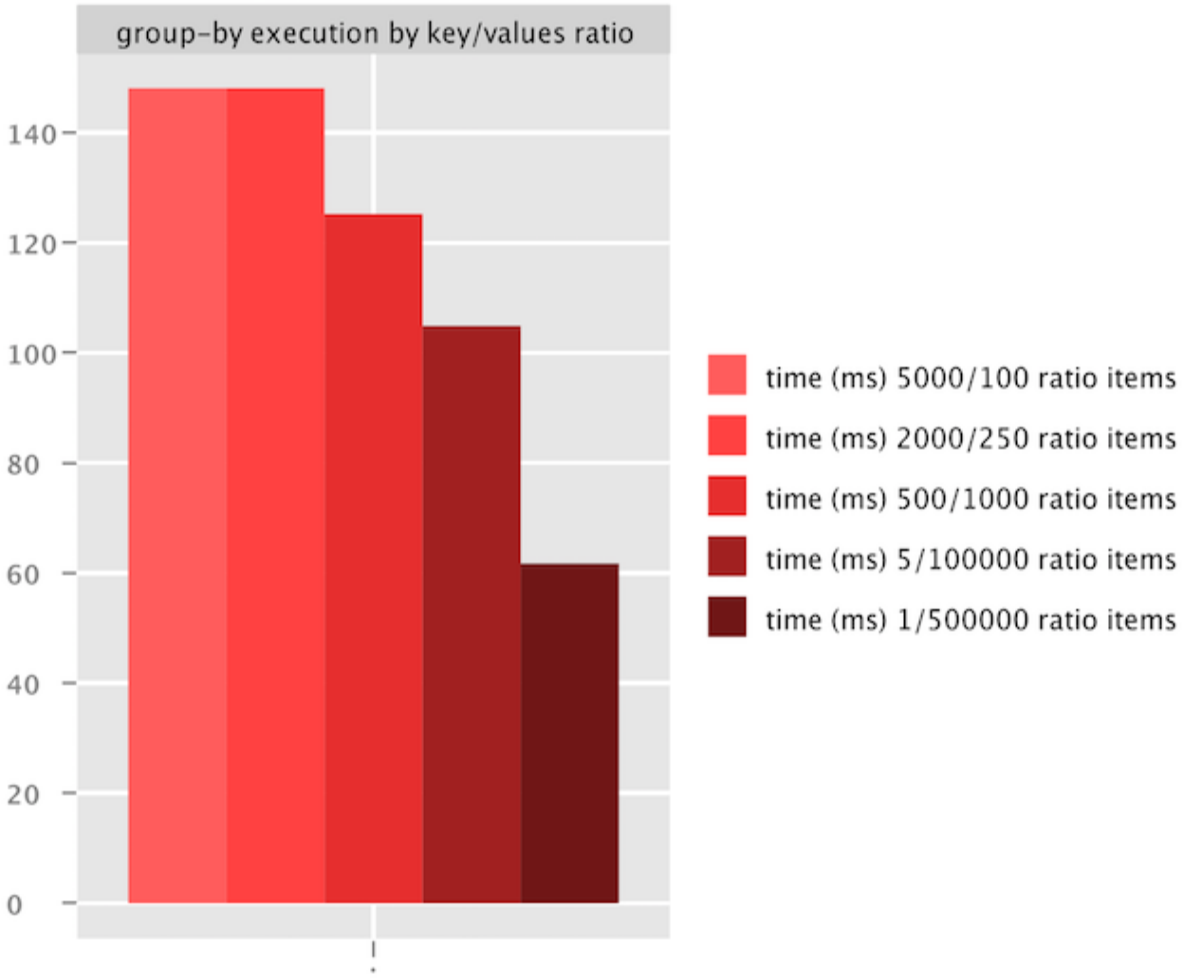
=> `O(n)` **Linear with the size of the input**

`group-by` eagerly consumes all items from the input collection while producing the output map. The larger the input, the more steps necessary, creating a linear behavior. `group-by` is based on reduce, producing a very similar performance profile.

As mentioned throughout the chapter, `group-by` is not lazy and eagerly consumes the input sequence by storing it in memory. The use of `group-by` on very large collection is possible but depending on the memory settings, it is prone to out of memory errors.

The following chart shows how `group-by` speed increases as the number of keys decreases in the result map:

Figure 8.9. `group-by` performance by number key collisions. The chart shows group-by performance for a collection with the same input size that changes in the number of items that can be grouped. The ratio "x/y" indicates the number of keys "x" in the final map containing "y" items each.



The chart shows that appending items to an already existing vector is faster than introducing new keys to the map, although

in relative terms, the speedup is not huge (going from 150 to 50 microseconds).

8.3.7 replace

`function` `since 1.0`

Listing 8.16. → *Collection processing, Selective Replacement*

```
(replace
  ([smap])
  ([smap coll]))
```

`replace` performs a selective replacement of the elements of the input collection as configured by a dictionary of substitutions. If the current item in the collection corresponds to one the key in the dictionary, it gets replaced with the value at that key:

```
(replace {:a "a" :b "b"} [:a 1 2 :b 3 4]) ; ❶
;; ["a" 1 2 "b" 3 4]
```

❶ `replace` takes a dictionary of substitutions and replace keys appearing in the input collection with the corresponding values. Note that a vector is returned if the input is a vector.

`replace` can also be used as a transducer, as in the following example:

```
(transduce
  (comp (replace {"0" 0})
        (map inc))        ; ❶
  +
  ["0" 1 2 "0" 10 11])
;; 30
```

❶ `replace` works as a transducer version by just omitting the collection argument.

Contract

input

- "smap" is an associative data structure (a data structure supporting access by key or index) and it's mandatory argument. Type supported are vectors (including transient vectors, subvectors, vector of primitives), maps (including transient hash-maps, array maps, Sorted maps and Java `HashMap` types). "smaps" can also be `nil` or empty.
- "coll" is a collection and is optional argument. When not provided, `replace` returns a transducer. Almost all collection types are accepted with few exceptions. "coll" can also be `nil` or empty.

notable exceptions

- `IllegalArgumentException` when "smap" is not associative, that is `(associative? smap)` returns `false`.
- `IllegalArgumentException` when it's not possible to obtain a sequential version of the "coll" (most notably, transients).

output

- returns: "coll" with "smap" substitutions. `nil` when "coll" is `nil`. The return type is a vector when "coll" is a vector, or a sequence otherwise.

Examples

The most common type of substitution dictionary for `replace` is a map. It also accepts a vector (another associative data structure). In this case it uses the index of the vector to match the element to replace:

```
(replace [:a :b :c] (range 10)) ; ❶
;; (:a :b :c 3 4 5 6 7 8 9)
```

❶vectors can be used as containers for substitutions. Each item in the vector is indexed by its position, creating a relationship equivalent to the map: `{0 :a 1 :b 2 :c}`

It's also possible to replace key-value pairs in maps, although is less common operation:

```
(def user {:name "jack" :city "London" :id 123})

(defn entry [k v] (clojure.lang.MapEntry/create

(def sub {(entry :city "London") [:postcode "WD1

(into {} (replace sub user))
;; {:name "jack", :postcode "WD12", :id 123}
```

❶Instead of searching for a key, we need to search for the entire map, including the value. When a map is iterated sequentially, it returns a list of `MapEntry` objects, which is what we need to match against. There is no Clojure function to create a `MapEntry`, but we can call the `create` method to the same extent.

❷The dictionary of substitutions contains `MapEntry` objects as keys and vector pairs as values.

❸Once replaced with `replace` we need to turn the sequential list of vector pairs back into a map with into.

`replace` could be used to implement a simple textual substitution system. An input string contains special placeholders that the system can identify and replace from a list of known substitutions:

```
(def text "You provided the following: user {usr
(def sub {"{usr}" "'rb075'" "{pwd}" "'xfrDDjsk'"

(transduce
  (comp
    (replace sub)                    ; ❶
    (interpose " "))
  str
  (clojure.string/split text #"\s"))  ; ❷
```

```
;; "You provided the following: user 'rb075' pas
```

❶Placeholders are represented by curly braces enclosing an identifier. Once the text has been split into words, they are isolated as a vector of sub-strings. The `replace` can be the first transducer to be applied in the chain, followed by interpose to restore the missing spaces.

❷We use string/split to split the string into a vector of sub-strings ready for processing.

See also

- string/replace is a function with the same name in the `clojure.string` namespace. It provides regular expressions based textual replacement for strings. Use `clojure.string/replace` if replacements are easy to describe with a regular expression and the input is text.
- clojure.walk/prewalk-replace works similarly to `replace`, but additionally walks nested data structures to apply substitutions.

- reduce-kv, is another way to transform a map into another. It gives more power than `replace` to select the right key-value pair to substitute and the actual substitution semantic. Prefer `reduce-kv` to `replace` on maps for all non-trivial transformations.

Performance considerations and implementation details

=> `O(n)` **linear**

`replace` needs to fully iterate the input to replace matching elements, so the number of computation steps increases linearly with the length of the input collection.

`replace` also needs to perform a lookup for each element in the collection. Since the lookup is almost constant time (`O(log32N)`), there shouldn't be any visible degradation, unless huge dictionaries are involved:

```
(require '[criterium.core :refer [quick-bench]])

(defn large-map [i] (into {} (map vector (range
(def big-map (large-map 2e6))

(let [v (into [] (range 1e6))]
  (quick-bench (replace {:small "map"} v))
```

```
    (quick-bench (replace big-map v)))

  ;; Execution time mean : 48.013222 ms
  ;; Execution time mean : 690.137260 ms
```

❶With the help of `large-map` function, we create a map with 1 million keys.

❷We use a small map and a large map to invoke `replace` using the same input collection of 1 million items. As expected, the size of the dictionary is also influencing the results, although dictionaries of 1 million keys are not common.

Now let's have a look at the difference in using a vector or a sequence as input:

```
  (let [s (range 1e6)                    ; ❶
        v (into [] s)]
    (quick-bench (doall (replace {:small "map"} s)
    (quick-bench (replace {:small "map"} v)))

  ;; Execution time mean : 51.061897 ms ; ❷
  ;; Execution time mean : 47.768686 ms
```

❶In this benchmark, we have a look at the differences in performance when we feed `replace` with a sequence or a vector. `replace` has two different implementations for them.
❷ `replace` is just a bit slower on sequences and at the same time it offers laziness.

Finally, let's see how the transducer performs:

```
(let [s (range 1000000)]                    ; ❶
  (quick-bench (doall (replace {:small "map"} s)
  (quick-bench (doall (sequence (replace {:small

(let [s (range 1000000)]
  (quick-bench (doall (map inc s)))
  (quick-bench (doall (sequence (map inc) s))))

;; Execution time mean : 67.827412 ms    ; ❷
;; Execution time mean : 104.787416 ms
```

❶In this benchmark, we compare `replace` and the `replace` transducer. We need to remember to doall on the resulting lazy sequence to fully realize the results.
❷We can see that the transducer version adds some more time to the iteration.

We can see that the transducer version is almost twice as slow than the normal version. The reader in this case should consider that the real advantage of transducers is when they are combined to perform multiple transformations at once without generating intermediate sequences.

8.3.8 reverse

`function` `since 1.0`

Listing 8.17. → *Collection Processing, Order Inversion*

```
  (reverse [coll])
```

`reverse`, as the name suggests, returns an inverted list of the elements in a collection:

```
  (reverse [9 0 8 6 7 5 1 2 4 3])      ; ❶
  ;; (3 4 2 1 5 7 6 8 0 9)
```

❶ `reverse` takes a collection as input and returns the elements in the input collection in reverse order.

While other sequential operations seen so far produce a lazy sequence, `reverse` is a rare example of function producing a

`clojure.lang.PersistentList` data structure:

```
(type (reverse [1 2 3])) ; ❶
;; clojure.lang.PersistentList
```

❶The type returned by `reverse` is always a Clojure `list`.

Contract

input

- "coll" is mandatory and can be of any type supported by seq.

notable exceptions

- `IllegalArgumentException` if you try to reverse something that does not offer a sequential version (a transient for instance).

output

- returns: a persistent list containing the input items in reverse order.

Examples

Clojure beginners tend to use the idiom sort then `reverse` to put a collection in reverse order. This is very inefficient. We can instead use `sort` with comparator:

```
(reverse (sort (shuffle (range 10)))) ; ❶
;; (9 8 7 6 5 4 3 2 1 0)

(sort > (shuffle (range 10))) ; ❷
;; (9 8 7 6 5 4 3 2 1 0)

(sort #(compare %2 %1) (shuffle (map str (range
;; ("9" "8" "7" "6" "5" "4" "3" "2" "1" "0")
```

❶An inefficient use of `reverse` to order a collection starting from the bigger element. This is definitely possible, but sort supports a custom comparator to provide a precise ordering.
❷We could use the more efficient comparator ">" with `sort`.
❸If the input is not numeric, we can create a custom comparator. Strings are comparable, so compare works with them directly.

We encountered `reverse` already while searching palindromic sequences in DNA strands (the reader is invited to check again in range). In the next example, we are going to see another common operation in bioinformatics called "reverse-

complement" which is at the core of the DNA transcription mechanism [150]:

```
(def DNA "CTATCTTTTAATCGGTTCTTGCAGTGAGATACATTCCA

(->> DNA
  reverse       ; ❶
  (replace {\A \T \T \A \C \G \G \C}) ; ❷
  (apply str))  ; ❸

;; "AAGTCGGGCATGTGGAATGTATCTCACTGCAAGAACCGATTAAA
```

❶The string is first reversed, creating a sequence of single characters.

❷replace swaps the nucleotides following the provided dictionary of substitutions.

❸The string is finally concatenated back together.

The proposed solution takes advantage of the sequential nature of strings to decompose the input into single letters, inverting the sequence, applying substitutions and then composing the result back into a string. Although the solution is not the most efficient (we are going to see a faster version in rseq), it's definitely simple and readable.

See also

- rseq invert a sequence in constant time, but only those implementing the `clojure.lang.Reversible` interface (essentially vectors, sets and maps).
- sort can be used instead of `reverse` if you also need to order the collection before reading it in reverse. The reverse ordering can be obtained while sorting the collection, without the need of an additional `reverse` step.

Performance considerations and implementation details

=> `O(n)` **linear in n**

`reverse` works by pushing each input item in a persistent list: the first item goes in first, then it "cons-es" the second and so on, obtaining the reverse effect typical of "cons-ed" lists ("cons-ed" is used colloquially in Lisp to identify a linked list built by consecutively invoking cons on the input). Thus `reverse` is not a lazy operation:

```
(first (reverse (map #(do (print % "") %) (range
;; 0 1 2 3 4...98 99 99
```

❶A short demonstration that `reverse` is not lazy.

Please note that it's not possible to have a sub-linear (less than `O(n)`) implementation of `reverse`. In-place `reverse` implementation on mutable data structures can achieve a `O(n/2)` at most. rseq achieve constant time by creating a lazy reverse indexing of the input. However it becomes linear as soon as the sequence is consumed. Let's compare the two approaches in the case of a fully consumed reverse sequence:

```
(require '[criterium.core :refer [quick-bench]])

(let [s (range 1e6)
      v (into [] s)]
  (quick-bench (reverse s))          ; ❶
  (quick-bench (reverse v))          ; ❷
  (quick-bench (doall (rseq v))))    ; ❸

;; Execution time mean : 10.520128 ms
;; Execution time mean : 13.818802 ms
;; Execution time mean : 11.555540 ms
```

❶ The first benchmark measures `reverse` on a long range, a typical case with lazy sequences.
❷ In the second benchmark, we use again `reverse` on a vector instead, a data structure that is more suitable for rseq.
❸ Finally we compare with rseq. Note that we now need to doall the reverse sequence.

The benchmark on a collection of 1 million items shows that the results (for fully realized results) are very similar between `reverse` and `rseq`. However, when a reversible input is available, rseq remains the best choice to consume smaller parts of the reversed sequence.

[144] Please refer to en.wikipedia.org/wiki/Fisher–Yates_shuffle for additional information about how the shuffling algorithm works

[145] here a more specific explanation of the round-robin algorithm: en.wikipedia.org/wiki/round-robin_dns

[146] A post on `comp.lang.functional` explains how purely functional shuffle works. It is available here: okmij.org/ftp/Haskell/perfect-shuffle.txt

[147] Reservoir Sampling is a family of algorithms dedicated to randomly choosing a sample from an arbitrarily large input. Algorithm "R" is the most common example of such algorithms. Please see en.wikipedia.org/wiki/Reservoir_sampling for more information

[148] As usual Wikipedia has a very detailed article about Merge-Sort available at en.wikipedia.org/wiki/Merge_sort

[149] Timsort was implemented first for the Python language and subsequently adopted by Java. Please see en.wikipedia.org/wiki/Timsort for more information

[150] We can't add too much details in the book regarding DNA transcription. However, the principle are clearly explained in this Wikipedia entry: en.wikipedia.org/wiki/Complementarity_(molecular_biology)#DNA_and_RNA_base_pair_complementarity

8.4 TRAVERSING

Data structures in Clojure are even more fundamental than other languages. Clojure applications not only use data but they are designed on top of them. As a consequence, arbitrarily nested and multi-typed data structures are common, especially as part of the data exchange between distributed systems.

Nested data models naturally as a tree. Let's take the following nested data structure:

```
{:t 'x ; ❶
   :n [{:t 'y :n [{:t 'x :n false}
                  {:t 'k :n [{:t 'h :n :halt}]}]
       {:t 'y :n "2011/01/01"}
       {:t 'h :n [{:t 'x :n 90.11}]}]}
```

❶An arbitrarily nested map which includes vectors and other data types.

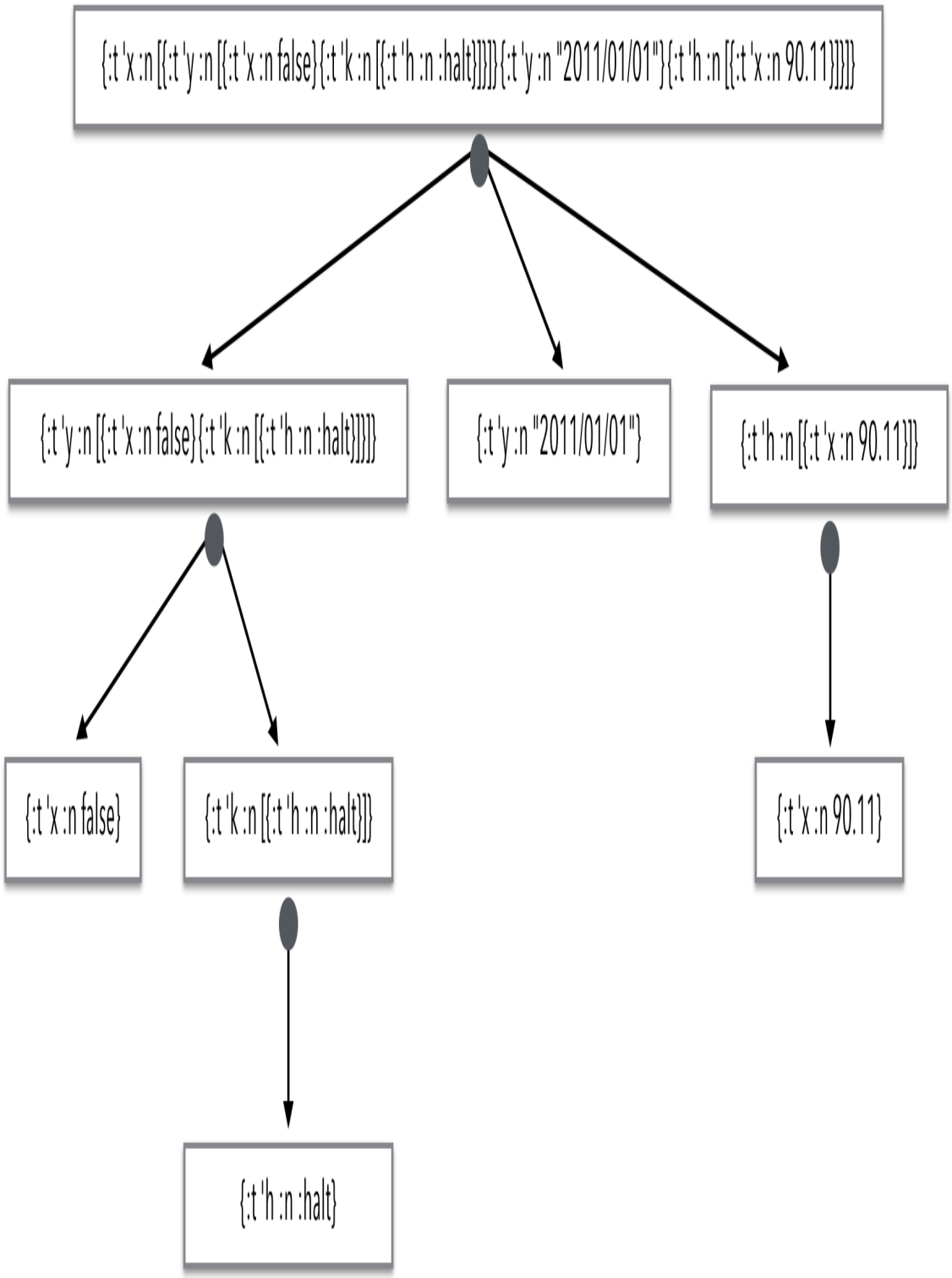
There isn't a single best way to model this data as a tree. We could for example establish the following convention:

- The presence of a vector indicates branching: the item containing the vector becomes a parent node and the items inside the vector are children nodes.

- The value at ":n" decides if there is additional branching.
- If the ":n" key does not contain a vector, the entire map at that level is a terminal node (also referred as "leaf" node).
- Anything else other than the ":n" key are "data" belonging to the node.

The tree formed by feeding the example data to the convention described above would look like the following picture:

Figure 8.10. A possible tree rendition of a nested data structure.


{:t 'x :n [{:t 'y :n [{:t 'x :n false}{:t 'k :n [{:t 'h :n :halt}]}]}{:t 'y :n "2011/01/01"}{:t 'h :n [{:t 'x :n 90.11}]}]}
{:t 'y :n [{:t 'x :n false}{:t 'k :n [{:t 'h :n :halt}]}]}
{:t 'y :n "2011/01/01"}
{:t 'h :n [{:t 'x :n 90.11}]}
{:t 'x :n false}
{:t 'k :n [{:t 'h :n :halt}]}
{:t 'x :n 90.11}
{:t 'h :n :halt}

A common operation to perform on trees is called **traversal**. Traversing is performed by visiting each node exactly once. While visiting the node, we can read its content or perform operations on the data. The traversal is said to be **depth-first** if we visit the children of a node before visiting the other nodes at the same level. The traversal is instead **breadth-first** if we visit the nodes at the same level before going down to visit children at deeper levels.

Algorithms for traversal usually allows one (or more) user-supplied functions to execute on each node. The traversal is said to be **pre-order** if the function executes on the parent before descending into the children (before "going down" for depth-first traversal or "going right" for breadth first). It is called **post-order** traversal if the function executes on the children before the parent (while going "back up" for depth-first traversal or "going left" for breadth-first traversal).

The main difference between pre-order and post-order traversal is that with pre-order with have a chance to influence the traversal itself by altering nodes before descending into them. With post-order we can model reduction-like operations by reducing the branch to a "value". Both styles have very interesting uses.

Traversal operations on nested data structures are so common that Clojure provides different ways to handle them: we've already seen tree-seq for lazy sequential access and this chapter is going to describe two additional traversal models: `clojure.walk` and `clojure.zip`. Compared to the other functions in this chapter, we are going to abandon the formal structure adopted so far in the book for a more free tutorial-like form.

8.4.1 walk, prewalk-demo and postwalk-demo

`walk`, `prewalk-demo` and `postwalk-demo` are useful functions to understand traversal of arbitrarily nested data structures (so it's a good idea to look at them before using prewalk or postwalk).

`walk` implements a type-dependent iteration on the children of a node. Based on the collection type, `walk` knows how to apply an "inner" and an "outer" function to transform the content. The inner function applies on each item of the collection, while the outer function applies on end result:

```
(require '[clojure.walk :as walk]) ; ❶

(defn inner [x] (println "inner on" x) x)
(defn outer [x] (println "outer on" x) x)
```

```
(walk/walk inner outer [1 [2] #{:a 1} 4]) ; ❷
;; inner on 1
;; inner on [2]
;; inner on #{1 :a}
;; inner on 4
;; outer on [1 [2] #{1 :a} 4]
;; [1 [2] #{1 :a} 4]
```

❶All functions in this section require explicit require of the `clojure.walk` namespace.

❷"inner" and "outer" are identity functions that additionally prints their argument. After feeding them to `walk/walk` we can see "inner" evaluates on each item from the input while "outer" executes just once at the end.

Note

`clojure.walk/walk` is not particularly interesting on its own, because it's not recursive. It is however the fundamental polymorphic step for all other `clojure.walk` functions. We are going to see how `clojure.walk/walk` helps traversal when talking about prewalk and postwalk.

`walk`, with similar printing functions, is used by `prewalk-demo` and `postwalk-demo` to demonstrate the traversal ordering of `prewalk` and `postwalk` on a deeply nested data structure:

```
(require '[clojure.walk :refer [prewalk-demo pos

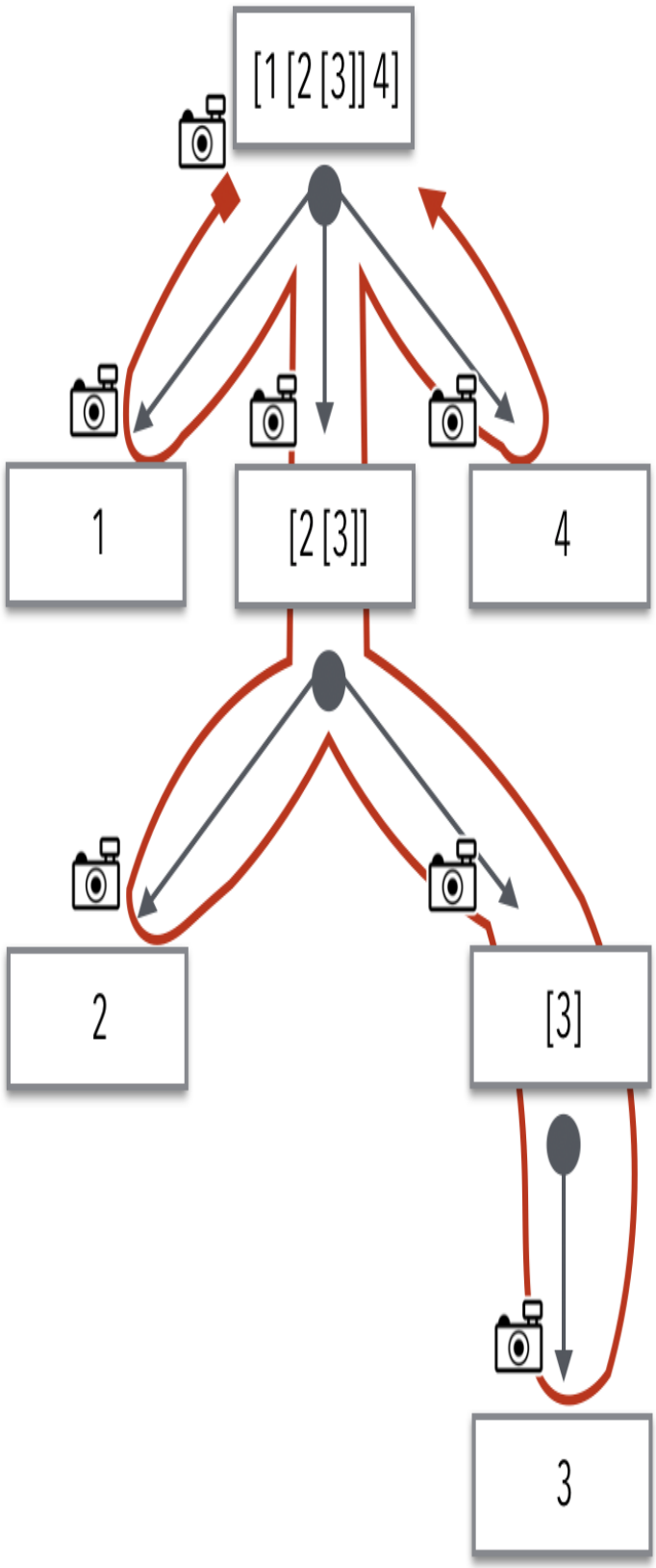
(prewalk-demo [1 [2 [3]] 4]) ; ❶
;; Walked: [1 [2 [3]] 4]
;; Walked: 1
;; Walked: [2 [3]]
;; Walked: 2
;; Walked: [3]
;; Walked: 3
;; Walked: 4
;; [1 [2 [3]] 4]
```

❶ `prewalk-demo` executes a depth-first pre-order traversal of an arbitrarily nested data structure using a printing function for illustrative purposes.

With `prewalk-demo`, we see the effect of traversing the data structure depth-first, printing a debug message ahead of visiting each node and while "moving down" the traversal. The

traversal path formed by the nested vector in the example is illustrated by the following picture:

Figure 8.11. Depth-first, pre-order traversal of a simple tree. The continuous line shows the traversal path, while the little cameras represent the call to the visiting function.


[1 [2 [3]] 4]
1
[2 [3]]
4
2
[3]
3

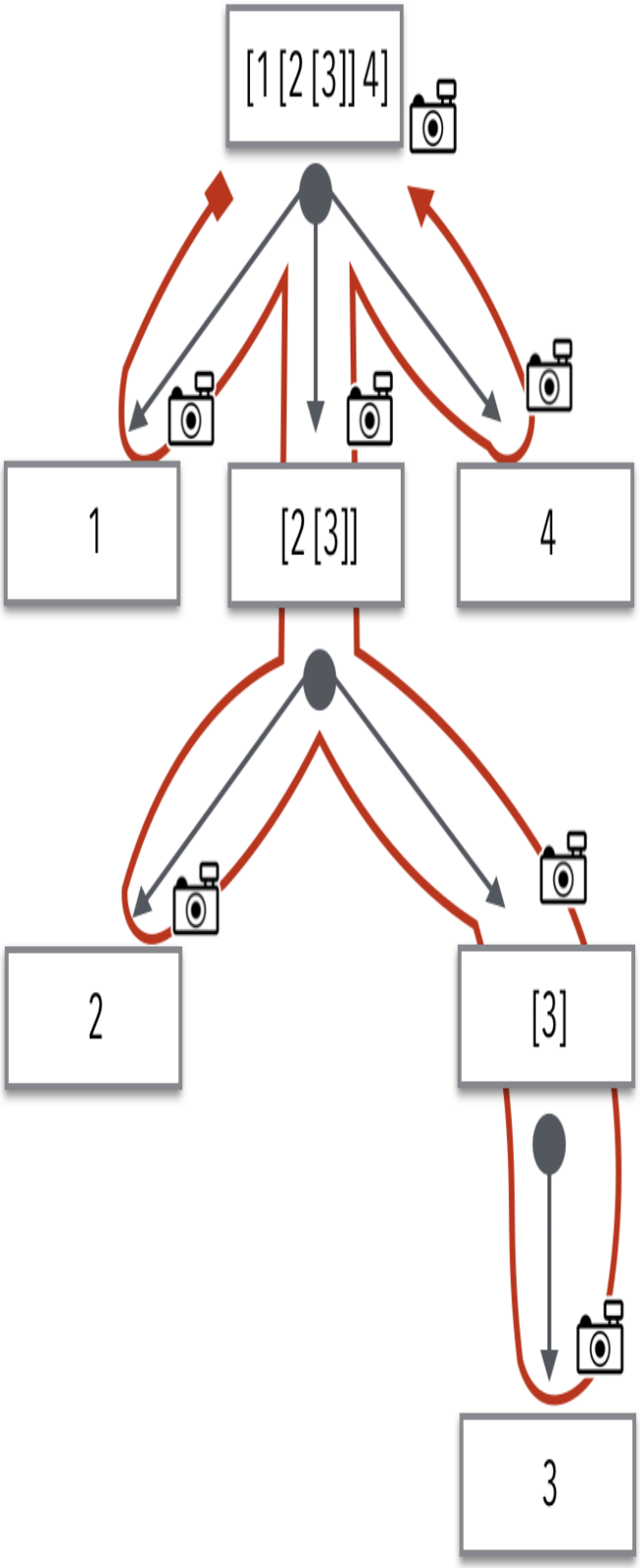
Similarly, `postwalk-demo` shows the effect of traversing the tree depth-first but only printing the node when ascending from a visited node:

```
(postwalk-demo [1 [2 [3]] 4]) ; ❶
;; Walked: 1
;; Walked: 2
;; Walked: 3
;; Walked: [3]
;; Walked: [2 [3]]
;; Walked: 4
;; Walked: [1 [2 [3]] 4]
;; [1 [2 [3]] 4]
```

❶ `postwalk-demo` executes a depth-first post-order traversal of an arbitrarily nested data structure printing a message each time is coming back up after visiting a node.

Differently from `prewalk-demo`, `postwalk-demo` prints the message coming back from each node and only after reaching the bottom of a branch, as illustrated by the next diagram:

Figure 8.12. Depth-first, post-order traversal of a simple tree. The continuous line shows the traversal path, while the little cameras represent the call to the visiting function.


[1 [2 [3]] 4]
1
[2 [3]]
4
2
[3]
3

There are good reasons to execute the visiting function ahead or after visiting each node: in the pre-order case, we have a chance to alter the traversal path by changing elements in the node. With a post-order visit we can process the output of the traversal, for example "reducing" the tree. We are going to see examples of both in the following section.

8.4.2 prewalk and postwalk

`function` `since 1.1`

Listing 8.18. → *tree traversal, data transformation*

```
  (postwalk [f form])
  (prewalk [f form])
```

`prewalk` and `postwalk` perform a depth-traversal of an arbitrarily nested data structure. They both take a function to execute on each nested item (collection or not). The function executes before descending into an inner collection in the case of `prewalk` (pre-order traversal) while it executes after reaching an innermost item in the case of post-order traversal (see also walk, prewalk-demo and postwalk-demo).

Contract

input

- "f" is a function of one argument. The function evaluates for each nested collection or other type in "form". It's mandatory argument.
- "form" can be of any type (with a couple of exceptions, see below) including `nil`. If "form" is a collection, then "form" is iterated recursively and "f" called on each item in turn. It's a mandatory argument.

notable exceptions

- `UnsupportedOperationException` is possible if "form" is recognized as Clojure collection type but the type does not implement all the necessary functions. One rare case in the standard library is bean which produces a map-like representation of an object but doesn't provide an `empty` method.

output

- The output type for both `prewalk` and `postwalk` depends mainly on the transformations operated by "f". In general usage, the output is a collection of the same type as the input.

Examples

`prewalk` executes a depth-first traversal of an arbitrarily nested collection, calling a function "f" on each item before descending into any nested item (also see prewalk-demo for more information):

```
(require '[clojure.walk :refer [prewalk]]) ; ❶

(prewalk #(do (println %) %) [1 [2 [3]]]) ; ❷

;; [1 [2 [3]]] ; ❸
;; 1           ; ❹
;; [2 [3]]     ; ❺
;; 2
;; [3]
;; 3           ; ❻
;; [1 [2 [3]]] ; ❼
```

❶Remember to require the necessary functions from the `clojure.walk` namespace.

❷We call `prewalk` with a simple function that prints its argument. The input collection contains 3 levels of nesting with an item each.

❸ `prewalk` calls "f" on the input "as is" as the first step of the computation.

❹The input is a collection, so `prewalk` process its content recursively. `prewalk` calls "f" on the first item in the collection, which prints the number "1".

❺ `prewalk` then calls "f" on the second item, the vector `[2 [3]]`. The second item is also a collection, so `prewalk` iterates each item in turn.

❻The number "3" is the last item in the innermost vector `[3]` and there are no other collection or items to iterate.

❼The result of `prewalk` is the same as the input, because our "f" just returned the item unaltered.

In the following example, we are going to use `prewalk` to prevent processing of a large branch in a deeply nested data structure. If a node is of type "pipeline" we don't want to execute any ":action" in the current or nested nodes:

```
(def data ; ❶
  {:type "workflow"
```

```
    :action '(do (println "flowchart") :done) ; ❷
    :nodes [{:type "flowchart"
             :action '(do (println "flowchart") :
             :nodes [{:type "workflow"
                      :action nil
                      :nodes false}]}
            {:type "routine"
             :action '(do (println "routine") :do
             :nodes [{:type "delimiter"
                      :action '(println "delimite
                      :nodes "2011/01/01"}]}
            {:type "pipeline"
             :action '(do (println "pipeline") :d
             :nodes [{:type "workflow"
                      :action '(Thread/sleep 1000
                      :nodes 90.11}]}
            {:type "delimiter"
             :action '(do (println "pipeline") :d
             :nodes [{:type "workflow"
                      :nodes 90.11}]}]})

(defn- step [node]          ; ❹
  (if (= "pipeline" (:type node))
    (dissoc node :nodes)
    (do
      (eval (:action node))
      node)))
```

```
  (time (prewalk step data))  ; ❺
  ;; flowchart
  ;; flowchart
  ;; routine
  ;; delimiter
  ;; pipeline
  ;; "Elapsed time: 4.098095 msecs"
  ;; {:type "workflow", :action (do (println "flow
```

❶ `data` is a small section of a much larger data structure that contains nodes that are very expensive to process. We still want to process the data, but we want to skip any wasteful processing.

❷ Each node contains a `:type`, `:action` and `:nodes` keys. The `:action` needs evaluation but we don't want to evaluate any "pipeline" action, including sub-nodes. When the action evaluates, the keyword ":done" appears in the output data.

❸ To show that `prewalk` is not processing the entire tree, a `Thread/sleep` call adds a 10 seconds delay if evaluated.

❹ The `step` function contains the necessary logic. If a node is of type ":pipeline", no action gets evaluated. The nested nodes are removed to prevent any further evaluation and they won't appear in the output.

❺ Calling `time` on `prewalk` immediately reveals that there is no 10 seconds wait. At the same time, other actions are evaluated as confirmed by the printouts of the different node types. Finally, the output data is the same as the input but "pipeline" nodes have disappeared and evaluated action are replaced with ":done".

If we used `postwalk` in the example above, we would see that `prewalk` and `postwalk` produce the same output but not the same side effects:

```
(require '[clojure.walk :refer [prewalk postwalk

(time (= (prewalk step data) (postwalk step data

;; flowchart ; ❷
;; flowchart
;; routine
;; delimiter
;; pipeline

;; flowchart
;; delimiter
;; routine
;; pipeline
;; flowchart

;; "Elapsed time: 10012.562208 msecs" ; ❸
;; true
```

❶This equivalence demonstrates that `prewalk` and `postwalk` produce the same result. However, side effects ordering (if any) and computational cost are changing.

❷The printouts corresponds to the type of the nodes, `prewalk` first and `postwalk` next. As you can see the ordering is different.

❸Another clue that `postwalk` is unable to prevent evaluation of sub-nodes comes from the 10 seconds necessary to return the result.

While `prewalk` is useful to reason about the structure of nested data ahead of processing, `postwalk` is perfect to process branches ahead of the parent node. A typical case is representing an expression as a tree where nodes are the operators and branches the operands. The operator cannot process the operands until they are of the correct type (for example numbers) but this requires to process the operands first (the equivalent of evaluating arguments before passing into a function). To illustrate the point, let's take the formula to calculate the compound interest that we saw talking about map and its representation as data:

```
(defn compound-interest ; ❶
  [rate loan-amount period]
  (* loan-amount
```

```
      (Math/pow
        (inc (/ rate 100. 12))
        (* 12 period))))

(defn compound-interest-data ; ❷
  [rate loan-amount period]
  {:function *
   :children
   [loan-amount
    {:function #(Math/pow %1 %2)
     :children [{:function inc
                 :children [{:function /
                             :children [rate 100
                {:function *
                 :children [12 period]}]}]}])
```

❶ `compound-interest` is a formula that calculates the total cost of a loan at the given yearly rate and period (please see the first example in map for more details). Code is data: in this case nested lists are interpreted by Clojure as function calls.

❷ `compound-interest-data` is the same function expressed using a different data structure made by maps and vectors.

Our goal is to evaluate the expression syntax defined by `compound-interest-data`. We can do that elegantly with

`postwalk` (but not `prewalk`):

```
(defn evaluate [node] ; ❶
  (if-let [f (:function node)]
    (apply f (:children node))
    node))

(postwalk evaluate (compound-interest-data 7.2 5
;; 5771.936460924754
```

❶ `evaluate` is a function of a node. If the node contains a function, then the function gets invoked on the children of the node. The operation succeeds only if the children are also evaluated, which happens only if we evaluate the nodes starting from the leaves. Note that the result of calling apply replaces the node.

❷ We can see how much you are required to pay a loan of 5000$ with an annual interested of the 7.2% which is payed back in 2 years.

`prewalk`, in this case, would be a difficult choice. `prewalk` would call `evaluate` while descending each node, when children are not evaluated yet. `postwalk` instead calls evaluate on leaves first, then on nodes while ascending back to

the root, which is exactly the expectation to apply a function to its (evaluated) values.

See also

- clojure.zip is a different way to traverse nested data structures while maintaining the traversal state. Use zippers to create a relationship between the state of the program and the state of the traversal and, in general, take complete ownership of the traversal policies.
- tree-seq flattens a depth-first pre-ordered traversal into a lazy sequence for additional processing. Use `tree-seq` to leverage laziness, for example to stop traversal when a specific node is found: `(some pred? (tree-seq coll? identity coll))`. `tree-seq` removes the parent-children relationship from the output, which is not ideal for any post-order processing.
- prewalk-replace or postwalk-replace offers a simpler approach if you are only interested in changing or replacing nodes in the input.

Performance considerations and implementation details

=> `O(n)` **linear in the number of nodes**

`prewalk` and `postwalk` perform linearly with the number of nodes in the input, both in terms of computation steps as well as memory. When applicable, "pruning branches" during pre-order traversal reduces the number of nodes to visit and improves overall performance.

`prewalk` and `postwalk` are eager operations that consume considerable amount of resources especially on large and complicated data structures, a factor to consider in real-life scenarios. There is also a slim probability to encounter a stack overflow with very deep data structures. However this scenario is not common in normal applications.

8.4.3 prewalk-replace and postwalk-replace

Replacing items in a deeply nested data structure is a common operation that Clojure provides as a dedicated function. `prewalk-replace` and `postwalk-replace` are functions built on top of prewalk and postwalk that perform a depth-first traversing of an arbitrarily nested data structure. When the current node matches a key present in a dictionary of substitutions, the item is replaced with the value corresponding to the key:

```
(require '[clojure.walk :refer [prewalk-replace

(def data  ; ❷
  [[1 2]
   [3 :a [5 [6 7 :b [] 9] 10 [11 :c]]]
   [:d 14]])

(prewalk-replace {:a "A" :b "B" :c "C" :d "D"} da
;; [[1 2] [3 "A" [5 [6 7 "B" [] 9] 10 [11 "C"]]]
```

❶ `prewalk-replace` and `postwalk-replace` are functions declared in the `clojure.walk` namespace.

❷ The example data is a vector of nested vectors. Some items that are not vectors are numbers, others are keywords.

❸ The map of substitutions contains some lower-case keyword as keys, with a corresponding upper-case string letter as value. `prewalk-replace` traverses the data while replacing the matching items.

We can generalize the concept of "dictionary of substitutions" with any data structure supporting contains? that can be used as a function of one argument. array-map or hash-map is a natural choice, but a vector works as well:

```
(def ^:const greek ; ❶
       '[α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ

(prewalk-replace greek data) ; ❷
;; [[β γ] [δ :a [ζ [η θ :b [] κ] λ [μ :c]]] [:d
```

❶ `greek` is a vector containing the Greek letters in lower-case. Each index in the vector associates with the given greek letter, forming a dictionary where keys are the number 0-23.

❷ `prewalk-replace` verifies if `greek` contains a value using the items in `data` as keys. If the item is a number between 0 and 23 it performs the substitution.

While `prewalk-replace` applies the substitution before descending into the data, `postwalk-replace` only applies the substitution after visiting a leaf that can't be iterated any further. This is similar to the difference we've seen between prewalk and postwalk. For example, in the following problem involving boolean expressions, we can simplify the input formula using `postwalk-replace`:

```
(require '[clojure.walk :refer [postwalk-replace

(def formula      ; ❶
  '(and (and a1 a2)
```

```
         (or (and a16 a3) (or a5 a8)
             (and (and a11 a9) (or a4 a8)))
         (and (or a5 a13) (and a4 a6)
              (and (or a9 (and a10 a11))
                   (and a12 a15)
                   (or (and a1 a4) a14
                       (and a15 a16))))))

(def ands          ; ❷
  '{(and true true) true   (and true false) fals
    (and false true) false (and false false) fal

(def ors
  '{(or true true) true  (or true false) true
    (or false true) true (or false false) false}

(def var-map       ; ❸
  '{a1 false a2 true a3 false a4 false
    a5 true a6 true a7 false a8 true
    a9 false a10 false a11 true a12 false
    a13 true a14 true a15 true a16 false})

(def transformed-formula  ; ❹
  (postwalk-replace (merge var-map ands ors) for

transformed-formula
;; (and
;;   false
```

```
  ;;   (or false true false)
  ;;   (and true false
  ;;     (and false false
  ;;       (or false true false))))
```

❶ `formula` is a nested list of "and" and "or" boolean operators concatenating 16 variables (from `a1` to `a16`). The formula needs substitutions of the variables before it can evaluate to `true` or `false`. However, before evaluation we have a chance to reduce the size of the formula using truth tables.

❷ `ands` and `ors` contain the and and or table of truth as key-value pairs in a map.

❸ `var-map` is a possible combination of values for the variables from `a1` to `a16`.

❹ `postwalk-replace` is given the concatenation of the truth tables with the variable substitution map. The substitution of variables happens during a post-order traversal: leaf nodes processing happens first, giving an opportunity to `postwalk-replace` to also use the truth tables.

The `transformed-formula` contains less nodes than the original while maintaining the original meaning. `prewalk-replace` would not be able to simplify the formula after

replacing the variables, as you can test by replacing `postwalk-replace` with `prewalk-replace` in the same example.

8.4.4 clojure.zip

`clojure.zip` is a namespace part of the standard library that contains an implementation of the zipper data structure [151]. A zipper represents a location inside a tree: the zipper can "move" around, retrieve nodes, or perform functional changes (where the original input never actually mutates).

When using zippers, it might be useful to think in terms of "modes". After creating a zipper we are in "editing mode": we can move around, retrieve nodes and perform changes. After performing any relevant operation, we call the function `root` to retrieve the resulting data (which includes all changes). Calling `root` exits editing mode: to re-enter editing we need to create a new zipper.

While in editing mode with a zipper, we have the option of retrieving nodes or locations: a node is pure data, the same you would expect from using normal Clojure functions. Locations are instead new positions that effectively move the focus of the zipper elsewhere.

Compared to clojure.walk functions, zippers separate the concept of traversal from processing. With `clojure.walk`, a depth-first traversal is the only option to perform any kind of operation. With zippers, the traversal algorithm is not part of the contract (although a depth-first traversal is optionally offered using `zip/next` or `zip/prev`) and we can pick a different (or partial) traversal quite easily.

The following is a summary of the zipper functions including brief explanation of their goal. We are going into more details in the following sections:

- **Building functions**: a zipper can be created with the generic `zipper` function, or we can use `seq-zip`, `xml-zip` or `vector-zip` to create one starting from a sequential, xml or vector input, respectively. `make-node` creates a new single node that can be added to the zipper while editing.
- **Location functions**: `up`, `down`, `right` and `left` move the zipper in one of the possible directions. `rightmost` and `leftmost` will jump many times `right` or `left` to reach the most distant sibling in the respective direction.

- **Retrieving functions**: `node` retrieves the data at the current location. `children`, `lefts` and `rights` retrieves data from below, left or right of the current location. `branch?` answers the question if the current location is a branch node (and implicitly if it's a leaf). Finally, `path` returns the list of nodes necessary to reach the current location of the zipper.
- **Update functions**: `replace` applies a substitution of the current node with a new one. `edit` is similar, but it takes a function of the current node to produce the next. `insert-left`, `insert-right` and `append-child` add a new in one of the respective directions.
- **Traversal functions**: zippers come with a built in depth-first traversal facility that can start from any location. `next` retrieves the next location, depth-first. `prev` retrieves the previous location in reverse depth-first order. After reaching the end of the traversal, `end?` returns `true` after reaching the end of the traversal.

Building Zippers

Clojure offers options to create zippers out of nested vectors (including subvectors and primitive vectors), lists (including

other native sequential types) and XML (as returned by clojure.xml/parse):

```
(require '[clojure.zip :as zip]) ; ❶
(require '[clojure.xml :as xml])
(require '[clojure.java.io :as io])

(def vzip ; ❷
  (zip/vector-zip
    [(subvec [1 2 2] 0 2)
     [3 4 [5 10 (vector-of :int 11 12)]]
     [13 14]]))

(def szip ; ❸
  (zip/seq-zip
    (list
      (range 2)
      (take 2 (cycle [1 2 3]))
      '(3 4 (5 10))
      (cons 1 '(0 2 3)))))

(def xzip  ; ❹
  (zip/xml-zip
    (->
        "<b>
            <a>3764882</a>
            <c>80.12389</c>
            <f>
```

```
                  <f1>77488</f1>
                  <f2>1921.89</f2>
                </f>
              </b>"
          .getBytes io/input-stream xml/parse)))
```

❶Clojure zippers live in the `clojure.zip` namespace, that needs to be explicitly required.

❷The first example shows how to create a zipper from a vector, including other vector-like types provided by Clojure.

❸Similarly, we can create a zipper starting from some type sequences (those implementing `clojure.lang.ISeq` which are the native sequences list, cons and sequence generators such as range or cycle).

❹ `zip/xml-zip` creates zippers out of XML documents. The document needs to be in the format provided by the output of `clojure.xml/parse` which in turns requires an input-stream.

The zipper object returned by `vector-zip`, `seq-zip` or `xml-zip`, is a tuple (vector or two items) that contains the data at the current location (initially, the entire input) and a map that describes the surrounding nodes (initially `nil`):

```
vzip ; ❶
;; [[[1 2] [3 4 [5 10 [11 12]]] [13 14]]
;;  nil]
```

❶Exploring the content of a freshly created zipper.

The recipe that describes how to navigate a specific type of data structure is embedded as metadata as part of the zipper:

```
(pprint (meta vzip)) ; ❶
;; {:zip/branch? ; ❷
;;  "clojure.core$vector_QMARK___4369@23802cfd"]
;;  :zip/children ; ❸
;;  "clojure.core$seq__4357@265a07d7"],
;;  :zip/make-node ; ❹
;;  "clojure.zip$vector_zip$fn__7605@4cae7ff1"]}
```

❶The recipe that tells the zipper how to traverse the data structure is embedded as metadata.
❷The `zip/branch?` key is similar to the "branch?" parameter in tree-seq and contains a function that tells the zipper how to distinguish a branch from a leaf. If `(branch? item)` returns `true` for any item in the input then the zipper knows that the item can be descended further.
❸The `zip/children` key contains a function to retrieve the children from a branch.
❹Finally, `:zip/make-node` contains the function that is used to create new node when needed.

The default zipper builders cover a few interesting cases and they were created at the time when XML was the common data exchange format. For general use, you'll likely need to the most general zipper function. But before creating a custom zipper, we need to introduce a few more primitives to change and retrieve locations.

Location Functions

The "location" of a zipper is the vector of two items returned right after construction and after calling a location function. The location contains a copy of the original input destructured to represent the current position in the data structure. A

location function is a function that changes such destructuring to represent another position. Let's see how `zip/down` changes the location:

```
(pprint vzip)

;; [[[1 2] [3 4 [5 10 [11 12]]] [13 14]] ; ❶
;;  nil]

(pprint (zip/down vzip))

;; [[1 2] ; ❷
;;  {:l [],
;;   :pnodes [[[1 2] [3 4 [5 10 [11 12]]] [13 14
;;   :ppath nil,
;;   :r ([3 4 [5 10 [11 12]]] [13 14])}]

(pprint (zip/rightmost (zip/down vzip)))
;; [[13 14] ; ❸
;;  {:l [[1 2] [3 4 [5 10 [11 12]]]],
;;   :pnodes [[[1 2] [3 4 [5 10 [11 12]]] [13 14
;;   :ppath nil,
;;   :r nil}]
```

❶ This is the original `vzip` instance as built by the `zip/vector-zip` constructor. The input data is intact as the first item in the vector.

❷ After calling `zip/down` the focus of the original data becomes `[1 2]` while the rest appears as part of the map in the second item of the tuple. The map contains keys for `:l` left nodes (there are no left nodes at this level), `:r` right nodes (we have the 2 nodes `[3 4 [5 10 [11 12]]]` and `[13 14]`) and `:pnodes` parent nodes looking up from where we came from.

❸ The call to `zip/rightmost` is now related to the location `[1 2]`. It moves the location all the way to the right most node available at that point of the traversal, which is `[13 14]`.

As you can see, each location function preserves the information required to move to other directions in the data structure without the need of any additional state. On reaching the edge of the data in any direction, location functions return `nil`:

```
(-> vzip zip/down zip/down zip/down) ; ❶
;; nil
```

❶ On reaching a leaf (in this case the number 1) the next request to move down result in a `nil` returned. This signals that we reached the edge of the data and we cannot move further in that direction. Also note that it's quite idiomatic to compose location functions using the `->` macro.

Looking inside locations is useful to understand how zippers work, but it's not the way they are actually used. After the desired location has been reached, there are dedicated functions to access the current or neighboring nodes that we are going to see in the next section.

Retrieving Functions

One of the most used function after moving the zipper to a specific location is `zip/node`, which retrieves the node at the current location as pure data:

```
(-> xzip zip/down zip/node) ; ❶
;; {:tag :a, :attrs nil, :content ["3764882"]}
```

❶ `zip/node` returns the data corresponding to a specific location.

We could also "look around" starting from a location and retrieves nodes which are to the right, left or below the current node:

```
(-> xzip zip/down zip/children) ; ❶
;; ("3764882")

(-> xzip zip/down zip/lefts) ; ❷
;; nil

(-> xzip zip/down zip/rights) ; ❸
;; ({:tag :c, :attrs nil, :content ["80.12389"]}
;;  {:tag :f, ;;   :attrs nil,
;;   :content
;;   [{:tag :f1, :attrs nil, :content ["77488"]}
;;    {:tag :f2, :attrs nil, :content ["1921.89"
```

❶ `zip/children` takes a location and returns a list of children nodes. In this case the only children node is available traversing the `:content` key of the xml map structure.

❷There are no `zip/lefts` nodes available starting at the current location. `zip/lefts` returns `nil`.

❸There are 2 `zip/rights` nodes to the right of the current location.

Using the `zip/path` function, we can "look up" from the current location and retrieve the trail of parent nodes descended so far. For example, if we move through the sequential zipper created at the start of the section down to the "(5 10)" node we can collect parents traveled so far with `zip/path`:

```
(zip/node szip) ; ❶
;; ((0 1) (1 2) (3 4 (5 10)) (1 0 2 3))

(-> szip ; ❷
    zip/down zip/right zip/right
    zip/down zip/rightmost
    zip/down
    zip/path)

;; [((0 1) (1 2) (3 4 (5 10)) (1 0 2 3)) ; ❸
;;  (3 4 (5 10))
;;  (5 10)]
```

❶ It’s useful to print the root of the sequential zipper to follow the traversal.

❷ We pass the sequential zipper "szip" through locations to reach the "(5 10)" node. At the end of the chain we call `zip/path` to collect all visited nodes so far.

❸ The result of `zip/path` only includes the nodes `zip/down` was called on.

Note

`zip/path` does not return a traversal of all visited node. As you can see from the example above, nodes like "(0 1)" and "(1 2)" were also part of the visit, but they are not collected from `zip/path`. We are going to see how to achieve proper traversal with `zip/next` below.

Creating custom zippers

Now that we have seen how to move between locations, we can go back to the problem of creating a zipper on top of any data structure. We’ve seen that the built-in constructors for vectors, lists and XML files are useful but limited to those specific use cases.

A more typical scenario is data transformations to and from JSON files, which involves mixing vectors and maps with arbitrary nesting. The following fragment of a much larger data structure is the result of parsing a JSON document:

```
(def document ; ❶
  {:tag :balance
   :meta {:class "bold"}
   :node
   [{:tag :accountId
     :meta nil
     :node [3764882]}
    {:tag :lastAccess
     :meta nil
     :node ["2011/01/01"]}
    {:tag :currentBalance
     :meta {:class "red"}
     :node [{:tag :checking
             :meta nil
             :node [90.11]}]}]})
```

❶The `document` presented here is a fragment from a much larger data structure originally transmitted as JSON file. A node is represented by a map with `:tag`, `:meta` and `:node` keys. If the value at the `:node` key is a collection of maps, then those represent the children of the node.

There is no built-in zipper constructor for these kind of data, but we are going to take inspiration from the very similar `xml-zip` implementation to build our own:

```
(defn custom-zip [root]
  (zip/zipper ; ❶
    #(some-> % :node first map?) ; ❷
    (comp seq :node) ; ❸
    (fn [node children] ; ❹
      (assoc node :node (vec children)))
    root)) ; ❺

(def czip (custom-zip document)) ; ❻
(-> czip zip/down zip/rightmost zip/down zip/nod
;; {:tag :checking, :meta nil, :node [90.11]}
```

❶We can use `zip/zipper` to create a generic constructor for custom data structures.

❷some-> is a good choice to compose together the conditions that determine if the passed argument is a branch. It needs to contain a `:node` key, the value at that node has a `first` element and that element is a `map?`.

❸The second argument is the "children" function. The function embeds the logic to extract a sequence out of the nodes.

❹We also need to tell how to assemble a new node given the node and a collection of children, although we are not using it in this specific example.

❺The final argument is the input data structure.

❻We use `custom-zipper` as usual to create a new zipper and navigate to the deepest node.

Update functions

Clojure offers a few functions to change, insert or delete nodes in a zipper. `replace` overwrites the node at the current location without looking at its current content, while `edit` requires a function of the old node to the new. `remove` simply deletes the current node:

```
(-> vzip zip/down zip/rightmost (zip/replace :re
;; [[1 2] [3 4 [5 10 [11 12]]] :replaced]
```

```
(-> vzip zip/down zip/rightmost (zip/edit conj 1
;; [[1 2] [3 4 [5 10 [11 12]]] [13 14 15]]
(-> vzip zip/down zip/rightmost zip/remove zip/r
;; [[1 2] [3 4 [5 10 [11 12]]]]
```

❶ `vzip` is the vector zipper created at the beginning of the zipper section. In this first example we can see how to change the location to "[13 14]" and replace its content with the `:replaced` keyword.

❷ If we use `zip/edit` instead of `zip/replace` at the same location, conj receives "[13 14]" as the first argument to which it adds "15".

❸ In the last example, the node is completely removed from the output. Note that we used `zip/root` to unwind directly to the root without using `zip/up`.

Note that the current location remains unchanged for all update functions except `zip/remove`. After removing a node, the current location becomes the location of the node that comes before the one that was removed in depth-first traversal order:

```
(-> vzip zip/down zip/rightmost zip/remove zip/n
;; 12
```

```
  (zip/node vzip)
  ;; [[1 2]
  ;;  [3 4 [5 10 [11 12]]] <-- location jump on 12
  ;;  [13 14]] <-- removes here
```

❶ `zip/remove` remove the current node, in this case "[13 14]" and jump to the previous location in depth traversal order, in this case "12". Note that the location could jump to a completely different branch, like in this case.

Zipper also offers a few options to add nodes: `insert-left` and `insert-right` add a new node to the left or to the of the current location, respectively:

```
  (-> vzip zip/down zip/rightmost (zip/insert-left
  ;; [[1 2] [3 4 [5 10 [11 12]]] INS [13 14]]
  (-> vzip zip/down zip/rightmost (zip/insert-righ
  ;; [[1 2] [3 4 [5 10 [11 12]]] [13 14] INS]
  (-> vzip zip/down zip/rightmost (zip/insert-chil
  ;; [[1 2] [3 4 [5 10 [11 12]]] [INS 13 14]]
  (-> vzip zip/down zip/rightmost zip/down (zip/in
  ;; Exception called children on a leaf node
```

❶For all 3 examples, the location moves do the node "[13 14]" before any insert operation. In this first case, we add a node "INS" to the left and at the same level of the current location.
❷With `zip/insert-right` the new element is added to the right.
❸ `insert-child` adds a new element as the leftmost item in the collection of existing children.
❹ `insert-child` does not automatically promote a leaf node to a branch. If the node is a leaf, it throws exception.

`insert-child` and `append-child` are similar operations for branch nodes only. `insert-child` adds a new children node of the current location as the leftmost node, while `append-node` appends the new node as the rightmost:

```
(-> vzip zip/down zip/rightmost (zip/insert-chil
;; [[1 2] [3 4 [5 10 [11 12]]] [INS 13 14]]

(-> vzip zip/down zip/rightmost (zip/append-chil
;; [[1 2] [3 4 [5 10 [11 12]]] [13 14 INS]]

(-> vzip zip/down zip/rightmost zip/down (zip/in
;; Exception called children on a leaf node
```

❶ `insert-child` adds a new element as the leftmost item in the collection of existing children.
❷ `append-child` adds a new element as the rightmost item instead.
❸ Neither `insert-child` nor `append-child` automatically promote a leaf node to a branch. If the node is a leaf, the operation throws exception.

`make-node` is useful to create new branch node that can be part of an existing zipper without necessarily knowing how nodes are assembled together. For example, we could use `make-node` to write a function to remove the first children from a node as follow:

```
(defn remove-child [loc] ; ❶
  (zip/replace loc (zip/make-node loc (zip/node

(-> vzip zip/down zip/rightmost remove-child zip
;; [[1 2] [3 4 [5 10 [11 12]]] [14]]

(-> vzip zip/down zip/rightmost remove-child rem
;; [[1 2] [3 4 [5 10 [11 12]]] []]
```

❶ `make-node` takes a location and a collection of children. The details related to the internals of the zipper comes as part of the metadata at the location.

❷ `remove-child` is used similarly to the rest of the zipper interface. A call to `remove-children` removes the first children node at that location.

❸ We can call `remove-child` repeatedly until there are no more children to remove, leaving the node empty.

Traversal Functions

The zipper namespace contains a few functions that move the current location in predefined directions following a depth-first traversal path. `zip/next` and `zip/prev` moves the location to the next or previous depth-first location, respectively:

```
(-> vzip zip/next zip/node) ; ❶
;; [1 2]

(-> vzip zip/next zip/next zip/node)
;; 1

(-> vzip zip/next zip/next zip/next zip/node)
;; 2
```

```
(-> vzip zip/next zip/next zip/next zip/next zip/
;; [3 4 [5 10 [11 12]]]
```

❶ `vzip` is the vector zipper that was defined at the beginning of the chapter. We can follow the result of repeated invocations of `zip/next` descending to the first node "[1 2]", visiting its elements and finally moving up to the next node.

If we want to perform the traversal of all the available nodes, we can repeatedly call `zip/next` on the result of the previous invocations with iterate. Note the use of `zip/end?` to decide when to stop the traversal:

```
(->> vzip
  (iterate zip/next) ; ❶
  (take-while (complement zip/end?)) ; ❷
  (map zip/node)) ; ❸

;; ([[1 2] [3 4 [5 [6 7 8 [] 9] 10 [11 12]]] [13
;;  [1 2]
;;  1 2
;;  [3 4 [5 [6 7 8 [] 9] 10 [11 12]]]
;;  3 4
;;  [5 [6 7 8 [] 9] 10 [11 12]]
;;  5
;;  [6 7 8 [] 9]
```

```
  ;;  6 7 8 [] 9 10
  ;;  [11 12]
  ;;  11 12
  ;;  [13 14]
  ;;  13 14)
```

❶iterate repeatedly calls `zip/next` on the location returned during the previous invocation (initially the vector zipper "vzip" which is the first location).

❷There is a specific `zip/end?` predicate of a location that returns `true` when the location is the last available in the traversal.

❸All collected locations so far needs to be translated into simple nodes to visualize their content.

It is quite natural to compare `clojure.zip` with `clojure.walk`. The key differences are:

- The zipper functions, except `zip/next` and `zip/prev`, do not mandate a specific traversal algorithm. `clojure.walk` only works in conjunction with a depth-first traversal (with option for pre-order or post-order).

- `clojure.walk` does not have a traversal state. As a consequence, we would need to use some mutable state (for example an atom) to collect a list of nodes during traversal. Zippers works naturally for both use cases: collecting nodes in a flat list or preserving the original nesting.

The result of the previous traversal using iterate was a lazy list of the visited nodes. In the next example we approach the problem eagerly using loop-recur to perform transformations while maintaining the original nested structure:

```
(defn zip-walk [f z] ; ❶
  (if (zip/end? z)
    (zip/root z)
    (recur f (zip/next (f z)))))

(zip-walk  ; ❷
  #(if (zip/branch? %) % (zip/edit % * 2))
  (zip/vector-zip [1 2 [3 4]]))
;; [2 4 [6 8]]
```

❶ `zip-walk` is analogous to clojure.walk/prewalk. It takes a function of a location and a zipper and traverse the zipper applying the function to each node.

❷The function we pass to `zip-walk` takes a zipper location and returns a location. In this case we perform the operation only if the node is not a branch.

There is no going back after traversal

Please be aware that once a zipper traversal completes with `zip/next`, there is no turning going back or moving location:

```
(def zipper-end ; ❶
  (-> (zip/vector-zip [1 2]) zip/next zip/next zip/next))

(zip/end? zipper-end) ; ❷
;; true

(zip/prev zipper-end) ; ❸
;; nil
```

❶The simple vector "[1 2]" produces an equivalently simple zipper.

❷After calling `zip/next` 3 times, we are at the end of the traversal.

❸After reaching the end of the traversal, the location cannot be used for any further navigation, including going back the traversal path with `zip/prev`.

If the traversal does not reach the end instead, then `zip/prev` or any other location change is possible. Also note that `zip/prev` does not have the same behavior on reaching the

root node: `zip/next` after `zip/prev` on the root location works as expected.

This concludes our description of the zipper functions and the chapter on collections. We are going to see a more specialized version of collection called "sequence" in the next chapter.

[151] Gerard Huet introduced a formalization of zippers in his paper from 1993: gallium.inria.fr/%7ehuet/PUBLIC/zip.pdf

8.5 SUMMARY

In this chapter we illustrated functions dedicated to collection processing, a large class of functions which includes general purpose facilities such as counting, iterating, access and traversal. A collection is the most general kind of container in Clojure and in the next two chapters we are going to describe a more specialized data abstraction called a "sequence". For the most part, Clojure programmers can ignore the difference between them as the standard library takes care of gracefully transitioning between the two. Additionally, most collection functions extend to Java collections and arrays for seamless interoperability.

9SEQUENCES

A Clojure sequence is an abstract data type. An abstract data type (or ADT) describes the behavior of a data structure without mandating a specific implementation. The following are the main properties of the abstraction:

- As the name implies it's iterated sequentially: you cannot access the nth element without first accessing the nth -1 (and there are no gaps).
- It works like a stateless cursor: the iteration can only move forward.
- It's persistent and immutable: like all other core data structures, sequences cannot be altered once created, but changes are possible in terms of a new sequence based on the previous (with structural sharing).

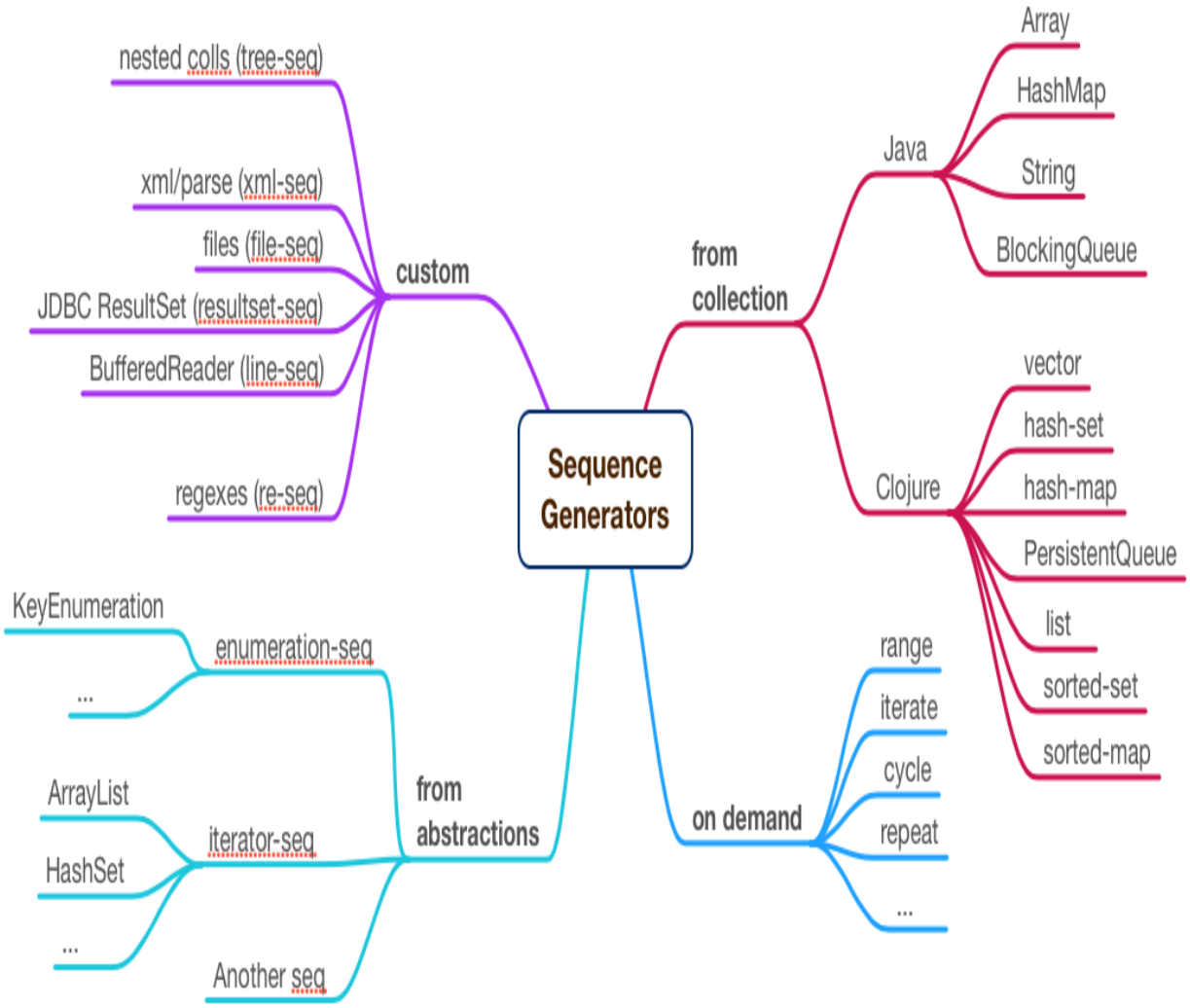
Optionally, sequences also support the following features (although they are not part of the contract):

- They are commonly (but not necessarily) lazy: the next element is produced only if that element is requested.

- They are also cached: the first access to the sequence elements produces a cached version of each item. Subsequent access to the sequence does not require further computation.
- They often apply "chunking" to improve performance. Chunking consists of processing a few more elements than requested, assuming the caller will soon move forward and access the rest of the sequence.

Clojure makes large use of sequences and as a consequence there are many functions devoted to them. The book dedicates two chapters to the topic, one about the way sequences are produced and another about their processing. The following diagram shows producers and consumers of sequences in the standard library.

Figure 9.1. The different built-in facilities to create sequences.



The next chapter is about sequence producers. There are essentially 4 ways to create them:

1. "Seqable" collections are collections supporting the sequential interface. A sequential view is produced by calling seq on them directly, or implicitly through one of the many processing functions (using seq internally).

2. On-demand generation: the data doesn't exist before consuming the sequence, but it is generated as soon as requested. Functions like range are a perfect example: the list of numbers does not exist until it is requested. range (and other similar functions) describes a recipe to produce the data, but it's not the data itself.
3. Custom generation: a sequence is built (often through the use of lazy-seq) on top of some source of data that is not necessarily structured or available in memory.
4. Native sequence: Clojure offers two concrete data structures implementing the sequential interface natively: consed lists and persistent lists.

9.1 SEQUENTIAL COLLECTIONS

seq generates a sequence starting from an enabled collection type. Almost all Clojure collections implement the `clojure.lang.Seqable` interface, which produces a conformed view of a collection for sequential access. Apart from the standard forward-only iteration produced by seq, Clojure also provides other flavors of sequential access:

- rseq creates a reverse sequence, a sequence that reads backwards from the last element.

- subseq and rsubseq produce a sequence from parts of a sorted collection, such as sorted-set or sorted-map.
- seque creates a blocking sequence, a sequence backed by a blocking queue that can potentially block if the consumer gets ahead of the producer.
- pmap produces a sequence while applying a transformation on the items in parallel.

We are now going to see the different types of sequential generation in detail.

9.1.1 seq and sequence

`function` `since 1.0`

Listing 9.1. → *Sequence Creation, Sequence Coercion, Transducers*

```
(seq [coll])

(sequence
  ([coll])
  ([xform coll])
  ([xform coll & colls]))
```

`seq` and `sequence` enable sequential behavior on top of existing collections. The collection is ultimately responsible for sending the data over, but the end result has to conform to a Clojure "sequence", a persistent and immutable data structure. Based on the target collection, the sequence is additionally cached or chunked (see the beginning of the chapter for a brief explanation of these features).

Despite being of huge importance for Clojure internals (all sequence functions call `seq` in one way or another), explicit use of `seq` has just a few idiomatic uses. `seq` can be used for example to check if a collection contains at least one element:

```
(def coll [])

(if (seq coll) :full :empty)    ; ❶
;; :empty

(if (empty? coll) :empty :full) ; ❷
;; :empty

(if (not-empty coll) :full :empty) ; ❸
;; :empty
```

❶ An idiomatic use of `seq` is to offer an uniform way to check if a collection contains at least one element. `seq` of an empty collection returns `nil` (and not the empty collection), which is what enables this conditional to work properly.

❷ Assuming we are happy to invert the order of the conditional branches (giving prominence to the fact that the collection is empty in the first place), we could use emtpy? instead.

❸ Finally, if we agree to negate the conditional form, we can also use not-empty.

`sequence` has additional features. Used with a single collection argument, it works similarly to `seq` with the only difference being the treatment of the empty collections:

```
(seq nil)
;; nil
(sequence nil)  ; ❶
;; ()
(seq [])
;; nil
(sequence [])   ; ❷
;; ()
```

❶ `seq` returns `nil` when invoked on a `nil` collection. `sequence` returns a new empty list instead.
❷ `seq` returns `nil` on an empty collection, while `sequence` returns an empty list.

After the addition of transducers to the standard library, `sequence` also offers the possibility to apply a transducer:

```
(sequence (map str) [1 2 3] [:a :b :c]) ; ❶
;; ("1:a" "2:b" "3:c")
```

❶ `sequence` accepts a transducer (or composition thereof) and a variable number of collections.

`sequence` is also the only transducer-aware function to support multiple collection inputs:

```
(sequence (map *) (range 10) (range 10)) ; ❶
;; (0 1 4 9 16 25 36 49 64 81)
```

❶When multiple collections are present, the first transducer receives a transformation call with 2 (or more) parameters. In this case the mapping function "*" receives two parameters to generate the square of a number.

Contract

input

- "coll & colls" are compatible collection types, including Clojure collections (excluding transients), other sequences, strings, arrays or Java iterables. `seq` requires one collection argument that can be empty or `nil`. `sequence` allows for a variable number of empty or `nil` collections, but at least one must be present.
- "xform" is a function following the transducers semantic. `sequence` is the only transducer function supporting multiple collection inputs. If `sequence` receives two or more "colls" arguments, then the transducer "xform" receives two or more arguments as well.

notable exceptions

- `IllegalArgumentException` is thrown for unsupported collection types.

output

- returns: a sequence representing the sequential view over the input collection. If one or more transducers are present, `sequence` applies the transducer chain to each element returned in the output sequence. When more than one collection is present, the output stops after reaching the shortest input.

Examples

`seq` has a few idiomatic uses. We saw in the introduction that it can be used to verify if a collection is not empty. This is useful property, for example during recursion, to gradually consume a collection. Here's the general mechanism implemented to reverse a generic collection input:

```
(defn rev [coll]
  (loop [xs (seq coll) done ()] ; ❶
    (if (seq xs)                ; ❷
      (recur
        (rest xs)               ; ❸
        (cons (first xs) done)) ; ❹
      done)))

(rev [8 9 10 3 7 2 0 0])        ; ❺
;; (0 0 2 7 3 10 9 8)
```

❶To be absolutely sure we can operate on "coll" through the sequential interface, we call `seq` when we first initialize the loop-recur construct. This has the effect of throwing an exception at the earliest possible point. If the `rev` function only operates in the context of Clojure data structures, explicit `seq` is not usually necessary.

❷ `seq` is used to check if a collection is not empty. We could also ask if the collection is empty? and reverse the if statement, but depending on algorithmic emphasis and programming style, it's good to have both options always available.

❸We can definitely call rest now, as we are forcing a `seq` "xs" conversion at the beginning of the loop.

❹We are using cons on "done", a local name initially bound to the empty list. lists support the sequential interface natively without a sequential adapter.

❺Collections like vectors are not native sequences, but Clojure adapts them easily by walking their internal structure.

In a similar fashion, `seq` could be used as a predicate to verify that all collections in a list contains at list one item:

```
(every? seq [#{} [:a] "hey" nil {:a 1}]) ; ❶
;; false
```

❶ `seq` can be used as a predicate for operations involving a list of collections.

After the introduction of transducers, `sequence` is now available for a brand new set of applications. Similarly to `seq`, `sequence` produces a sequential view on top of the input collection. Additionally, `sequence` applies a transformation on each item using the provided transducer chain.

In the following example, we are going to parse some unstructured input to extract the information we need. We need to connect to a device which returns data as a two-dimensional representation similar to the grid on a spreadsheet. The output also contains interleaving rows that we don't need. Here's an example of the data we would like to extract and how it appears when connecting to the device:

```
;; == example data pattern == ; ❶
;; Wireless MXD CXP  ; header: kind & codes
;; ClassA 34.97 34.5 ; metric: name & measures
;; ClassT 11.7 11.4  ; metric: name & measures
;; ClassH 0.7 0.4    ; metric: name & measures

(def device-output ; ❷
  [["Communication services version 2"]
   ["Radio controlled:" "Enabled"]
```

```
     ["Ack on transmission" "Enabled" ""]
     ["TypeA"]
     ["East" "North" "South" "West"]
     ["10.0" "11.0" "12.0" "13.0"]
     ["Wireless" "MXD" ""]
     ["ClassA" "34.97" "" "34.5"]
     ["ClassB" "11.7" "11.4"]
     ["Unreadable line"]
     ["North" "South" "East" "West"]
     ["10.0" "11.0" "12.0" "13.0"]
     ["Wired" "QXD"]
     ["ClassA" "34.97" "33.6" "34.5"]
     ["ClassC" "11.0" "11.4"]])
```

❶The example shows the kind of data pattern we are searching for: it contains an header followed by several lines of numerical metrics.

❷This is the output we receive from the device. The interesting data appears in the output interleaved by additional "noise" we want to remove.

The approach we follow is to read the device output top to bottom and gradually create groups of rows. We keep the group only if it conforms to the interesting pattern of data. We can use the following predicates to check if a group of lines is something we are interested in or not:

```
(defn measure? [measure] ; ❶
  (and
    measure
    (re-matches #"[0-9\.]*" measure)))

(defn metric? [[name & measures]] ; ❷
  (and
    name
    (re-matches #"Class\D" name)
    (every? measure? measures)))

(defn header? [[kind & [code]]] ; ❸
  (and
    (#{"Wireless" "Wired"} kind)
    (#{"MXD" "QXD" "CXP"} code)))

(defn pattern? [[header & metrics]] ; ❹
  (and
    (header? header)
    (every? metric? metrics)))

(pattern? [["Wireless" "MXD" ""] ; ❺
           ["ClassA" "34.97" "" "34.5"]
           ["ClassB" "11.7" "11.4"]])
;; true
```

❶A `measure?` is a string representing a decimal number. The regular expression is very simple for the purpose of this example.
❷A `metric?` is a list containing a name and any number of measures. The name must start with "Class" and followed by a letter.
❸A `header?` is a list of strings. It should start with a "kind" (either "Wireless" or "Wired") and followed by a "code".
❹A `pattern?` matches the entire specification. It checks that the first line is a valid header and what follows are metrics.
❺We can try the predicate on a test specification.

Now that we have implemented the predicates to recognize the interesting pattern of data, it's time to process the raw feed from the device. We proceed by iterating a range from 0 up to the number of input lines and then use nthrest to gradually remove them from the top. This generates a list of all ordered subsets of the input. We know that some subset of the input could be the pattern of data we are interesting in. `sequence` comes handy to process this sequence using transducers:

```
(defn all-except-first [lines]  ; ❶
  #(nthrest lines %))

(def if-header-or-metric        ; ❷
```

```
    #(take-while (some-fn header? metric?) %))

  (defn filter-pattern [lines]  ; ❸
    (sequence
      (comp
        (map (all-except-first lines))
        (keep if-header-or-metric)
        (filter pattern?))
      (range (count lines))))

  (filter-pattern device-output)

  ;; ((["Wireless" "MXD" ""]
  ;;   ["ClassA" "34.97" "" "34.5"]
  ;;   ["ClassB" "11.7" "11.4"])
  ;;  (["Wired" "QXD"]
  ;;   ["ClassA" "34.97" "33.6" "34.5"]
  ;;   ["ClassC" "11.0" "11.4"]))
```

❶The first transducer takes all lines except the first "n", where "n" comes from the iteration of a range input.

❷The second transducer only keeps lines in a subset that are either headers or metrics, which are the only line types we are interested in.

❸ `filter-pattern` assembles all transducers together.

When we finally test `filter-pattern` on the raw input coming from the device, we can see that it correctly assembles the patterns of data we are searching for.

What is the role of a sequence in Clojure?

A sequence is, for all practical purposes, a collection. Differently from other collections, a sequence does not implement a container for its data (apart from caching values), but receives data from some source. `seq` and `sequence` receive their data from other collections, but there are other ways to obtain data: data for a sequence can be generated on demand by built-in functions (for example range), asking a custom producer (using lazy-seq) or other objects (with resultset-seq , xml-seq , tree-seq etc.)

Another important aspect is that a sequence is lazy by construction. The mechanism to access the next element (from the sequence down to the concealed collection) does not actually operate until elements are requested upstream. This is particularly effective for data generators, as data are nowhere in memory until the element is requested.

A sequence also offers a basic caching mechanism. The sequence can be assigned to a let block and accessed multiple times and the underlying collection or generator doesn't perform any additional work (producing consistent results as far as any involved function don't have side effects).

See also

- list creates a concrete collection that is also a sequence natively.

- lazy-seq offers a way to create a custom data generator for a sequence.
- iterator-seq returns a sequential view over a `java.util.Iterator` instance. Many Clojure and Java collections are accessible through the `Iterator` interface.

Performance considerations and implementation details

=> `O(n)` **steps (fully evaluated)**

=> `O(n)` **memory (worst case)**

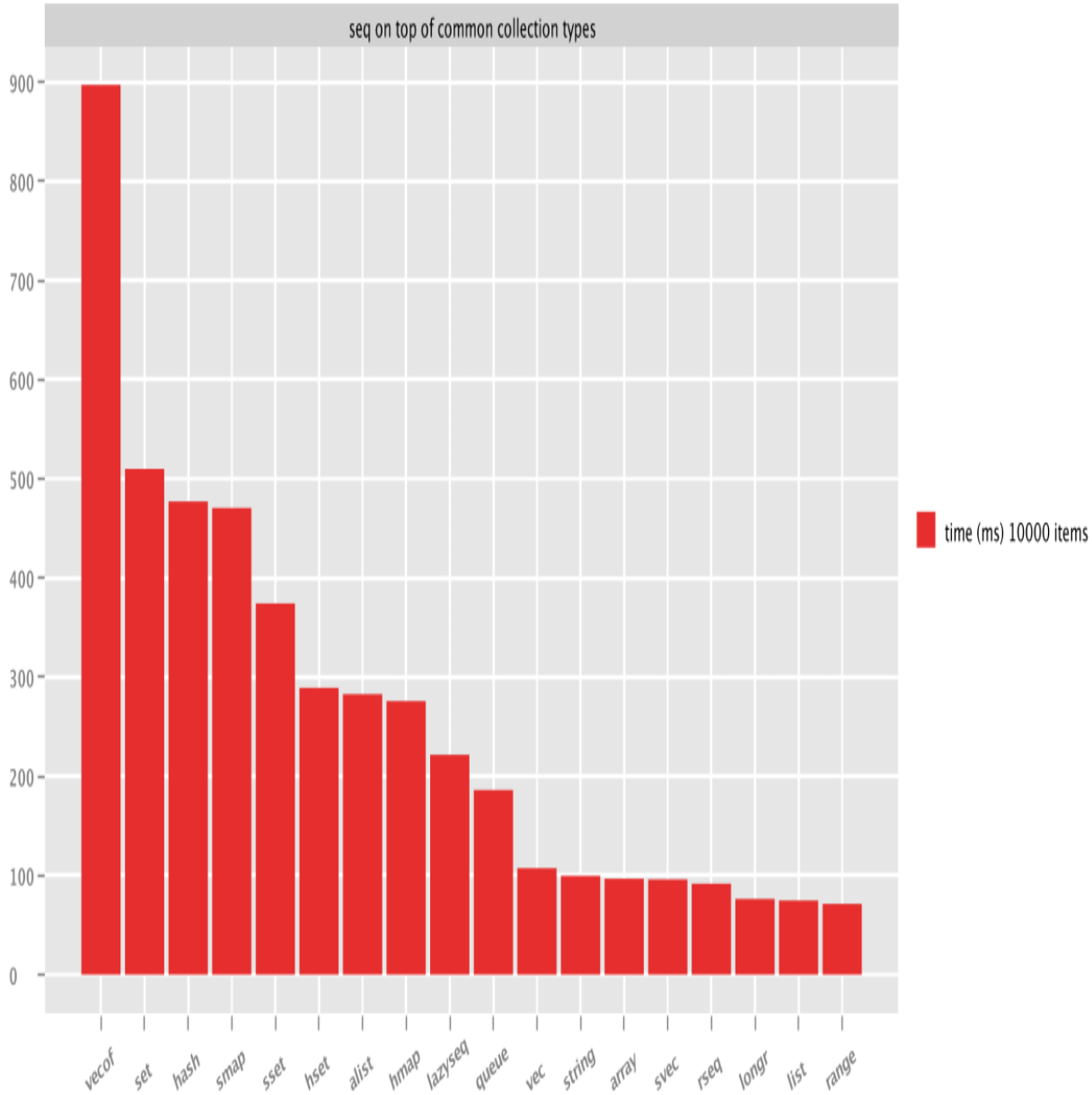
`seq` and `sequence` depend on the length of the fully consumed sequential collection (or generator). Values are cached, so when fully consumed a copy of the input collection is created by the sequence generation giving linear behavior in memory space.

From the implementation perspective, `seq` and `sequence` require the input type to provide the sequential interface. The input has the opportunity to provide an optimal algorithm to build a sequence of its elements.

The following chart shows `(doall (seq coll))` called on the most common sequential types containing the same 10000

items. vector-of is the worst performing, followed by Set and hash-map and their ordered variants.

Figure 9.2. `seq` called on the most common sequential types "hset" and "hmap" are abbreviations for `hash-set` and `hash-map` respectively.



On the fast side of the chart, we find native sequential collections and generators like ranges and long ranges. Other types of vectors are also performing well. For many of the types benchmarked in the chart the sequential transformation is of relative importance since their main goal is to offer direct lookup access.

Caching is an important factor in performance, especially if expensive computations are involved. If reuse is necessary, the sequence can be closed over (for example with a let binding) and re-used efficiently. We could go ahead and extend the parser output example seen previously by adding a side-effecting transducer to see if the same message appears more than once:

```
(defn filter-pattern [lines]  ; ❸
  (sequence
    (comp
      (map (all-except-first lines))
      (keep if-header-or-metric)
      (filter pattern?)
      (map #(do (println "executing xducers") %)
    (range (count lines))))

(let [groups (filter-pattern device-output)] ; ❷
  [(dorun (seq groups))
```

```
    (dorun (first groups))
    (dorun (last groups))])

;; executing xducers ; ❸
;; executing xducers
;; [nil nil nil]
```

❶The side effecting transducer is a map transducer printing to standard output and returning the input without any modification.

❷`groups` is the local name for the results of the `sequence` call. We are then creating another sequence on top, accessing the first and the last element.

❸We can see two printouts corresponding to the groups that were found in the input. Despite first and last are both calling `seq` on their input (effectively creating a new sequence on top of `groups`) no other printouts are visible, showing that the transducer chain is never invoked again. eduction prevents this behavior for those cases where transducers are expected to execute again.

Laziness and caching have implications when the wrapped collection is mutable (like a Java collection) and it mutates after the sequential view is created. Here's an example showing the effect of mutation on the sequential view:

```
(import '[java.util ArrayList])

(let [a (ArrayList. [:o :o :o]) s (seq a)] ; ❶
  [(.set a 0 :x) (first s) (.get a 0)])    ; ❷
;; [:o :x :x]

(let [a (ArrayList. [:o :o :o]) s (seq a)] ; ❸
  [(first s) (.set a 0 :x) (first s) (.get a 0)]
;; [:o :o :o :x]
```

❶The sequence is created here, when `seq` is invoked on a `java.util.ArrayList` instance.

❷The `ArrayList` is mutated to now contain a different item in the first place. This happens before we ever called first on the sequence "s" created previously, so the new value gets printed not the old.

❸We now make access to the first element ahead of the mutation, so we get the current ":o" content first. After mutating the `ArrayList` the first element of the sequence is unchanged, because the value was cached.

A final remark about the use of `sequence` with transducers from the performance perspective. `sequence` implementation for transducers uses a buffer mechanism to temporarily park each transformed item and return it as the sequence is

consumed. For trivial transducer chains, normal sequential generation is faster than transducer sequential generation:

```
(require '[criterium.core :refer [bench quick-be

(let [xs (range 500000)]
  (bench (last (filter odd? (map inc xs))))) ; ❷
;; Execution time mean : 26.944707 ms

(let [xs (range 500000)]
  (bench (last (sequence (comp (map inc) (filter
;; Execution time mean : 37.773642 ms
```

❶We use the Criterium library to measure performance in the most accurate way.

❷A range of 500.000 items is completely evaluated by accessing the last element. The processing involved is very simple.

❸We operate similarly with `sequence` and a transducer chain. We can see a slight performance degradation.

When considering if to use the transducer version of `sequence` or plain sequence generation, laziness definitely plays a role. In order to see some performance degradation we need to fully consume a relatively large range as seen in the benchmark: you can confidently use `sequence` with

transducers for medium size input or non-trivial transducers chains.

9.1.2 rseq

`function` `since 1.0`

Listing 9.2. → *Sequence, Reverse, Fast Vector Reversal*

```
  (rseq rev)
```

`rseq` creates a reversed sequential view on top of a collection. The collection needs to know how to produce such a view to work with `rseq` (it needs to implement the `clojure.lang.Reversible` interface).

One of the main use of `rseq` is to provide a constant time reverse version for vectors, Sorted maps and sorted sets which would otherwise be forced into a linear sequence scan. `rseq` returns a "reversed view" of the input data structure, so when elements are iterated they are returned in inverted order:

```
  (rseq [:b :a :c :d]) ; ❶
  ;; (:d :c :a :b)
```

```
(rseq (sorted-map :d 0 :b 3 :a 2)) ; ❷
;; ([:d 0] [:b 3] [:a 2])
```

❶ `rseq` used on a vector returns a reversed sequence.

❷ `rseq` returns a sequence of the entries in the sorted map in reverse.

As we can see, results print inside round parentheses. This correctly indicates that the return type is sequential (vectors, sorted maps and sorted sets return a specific `rseq` wrapper that implements the sequential interface):

```
(conj (rseq [1 2 3]) :a) ; ❶
;; (:a 3 2 1)
```

❶ `conj` into the reversed vector inserts to the "head" instead of the tail position (as it would be the case with vectors).

`rseq` on vectors returns effectively a sequence. For this reason beware that operations like peek or nth are not optimized for `rseq` output even when the input data structure is a vector.

Warning

`rseq` differs from `reverse` in its treatment of the empty collection and `nil`. `(reverse nil)` and `(reverse [])` returns `nil`, but `(rseq nil)` leads to a `NullPointerException` being thrown.

Contract

input

- "rev" is the only mandatory argument. "rev" must be a collection implementing the `clojure.lang.Reversible` interface. An helper can be used to check if this is true for "rev": `(reversible? rev)` should return `true`. Currently only vectors created with vector, vector-of or subvec are reversible. Additionally, Sorted maps and sorted sets are also reversible.

notable exceptions

- `ClassCastException` when "reversible" doesn't implement `clojure.lang.Reversible`. Vectors and sorted map/set are compatible with `rseq`.

- `NullPointerException` if "reversible" is `nil`.

output

`rseq` returns:

- A sequence of the elements in "rev" in their reverse order when "reversible" is non-empty.
- If "reversible" is a non-empty sorted map, a sequence of the key value pairs in the map in reverse key order.
- `nil` when "reversible" is an empty vector, sorted map or sorted set.

Examples

We've seen palindromes already while talking about range. Palindromes have many applications, for example they are used in genetics to find enzymes enabled locations in DNA strands [152]. DNA consists of strings of nucleotides. By convention the nucleotides are called A, T, G and C. A and T complement each other, as do C and G.

A sequence of DNA nucleotides is a palindrome if it's equal to its reverse complement. Let's see how we can use `rseq` to write a program to find palindromes in DNA sequences:

```
(defn complement-dna [nucl]                    ; ❶
  ({\a \t \t \a \c \g \g \c} nucl))

(defn is-palindrome? [dna]
  (= (map complement-dna dna) (rseq dna)))  ; ❷

(defn find-palindromes [dna]                   ; ❸
  (for [i (range (count dna))
        j (range (inc i) (count dna))
        :when (is-palindrome? (subvec dna i (inc
        [i j]))

(mapv complement-dna [\a \c \c \t \a \g \g \t])
;; => [\t \g \g \a \t \c \c \a]

(is-palindrome? [\a])
;; => false

(is-palindrome? [\a \c \c \t \a \g \g \t])
;; => true

(find-palindromes [\a \c \g \t])
;; => ([0 3] [1 2])
```

❶ We use a map to find the complement of each nucleotide.
❷ `is-palindrome?` checks if a DNA sequence is palindromic. It calculates the complement and reverses the sequence using `rseq`. The result is compared to the input sequence.
❸ We use for list comprehension to iterate over all sub-vectors of DNA consisting of two or more nucleotides. `find-palindromes` returns a lazy sequence of pairs specifying the start and the end of each palindrome (if any).

The algorithm above is an elegant example that demonstrates the use of `rseq` and avoid reversing the input. However, it wasn't designed with production scale in mind and you should look into more sophisticated techniques for real-life problems [153].

See also

- seq is used to return a sequence on a collection without reverting its content.
- reverse returns a reverse sequence on all seqable collections, not just vectors.

Performance considerations and implementation details

=> `O(1)` **Time, sequence creation**

=> `O(n)` **Time, fully consumed**

=> `O(n)` **Space, fully consumed**

A simple `rseq` invocation performs in constant time as it doesn't actually iterate the input. If the resulting sequence is iterated, then `rseq` performance is linear in the number of elements in the collection. `rseq`, like seq, caches values in memory, consuming memory linearly in the number of items.

The obvious comparison is with reverse. Let's visit again the `palindrome` example that was shown in the "Performance Considerations" section of range. In that example, we wanted to find if a very long DNA sequence was a palindrome or not. The example considered two solutions: one comparing items starting from the middle, the other using reverse and then comparing with the original input. The reverse-based solution is compared here to a solution based on `rseq`:

```
(require '[criterium.core :refer [quick-bench]])

(defn complement-dna [nucleotide] ; ❶
  ({\a \t \t \a \c \g \g \c} nucleotide))

(defn random-dna [n] ; ❷
  (repeatedly n #(rand-nth [\a \c \g \t])))
```

```
(defn palindrome-reverse? [dna] ; ❸
  (= (map complement-dna dna) (reverse dna)))

(defn palindrome-rseq? [dna] ; ❹
  (= (map complement-dna dna) (rseq dna)))

(let [dna (random-dna 1e4)]
  (quick-bench (palindrome-reverse? dna)))

;; Execution time mean : 834.510161 µs

(let [dna (vec (random-dna 1e4))]
  (quick-bench (palindrome-rseq? dna)))

;; Execution time mean : 2.940745 µs ; ❺

(let [dna (apply concat (repeat 1e4 [\a \c \c \t
  (quick-bench (palindrome-reverse? dna)))

;; Execution time mean : 12.991438 ms

(let [dna (vec (apply concat (repeat 1e4 [\a \c
  (quick-bench (palindrome-rseq? dna)))

;; Execution time mean : 11.238614 ms ; ❻
```

❶ This is the same function to complement nucleotides shown in the example.
❷ `random-dna` creates random DNA sequences of length "n".
❸ `palindrome-reverse?` is the reverse-based version for comparing the DNA sequence. Note that this version doesn't need a vector as input.
❹ `palindrome-rseq?` uses `rseq` to check for palindromes. Note that this version requires a reversible input (such as a vector).
❺ We can see that a random sequence (with a low probability of being a palindrome) can be spotted much faster with `rseq` than `reverse` (more than two orders of magnitude).
❻ If we use a sequence of similar length that is a palindrome instead (the worst case for comparing) we can see almost no difference.

The solution based on `rseq` is more than two orders of magnitude faster than a similar solution based on reverse. The result is achieved considering that random sequences have a very low probability of being a palindrome and equality

returns `false` pretty soon after the start of the iteration, without the need to fully realize the reversed sequence.

9.1.3 subseq and rsubseq

`function` `since 1.0`

Listing 9.3. → *Sequential selection, Searching, Range selection*

```
(subseq
  ([sc test key])
  ([sc start-test start-key end-test end-key]))

(rsubseq
  ([sc test key])
  ([sc start-test start-key end-test end-key]))
```

`subseq` and `rsubseq` create a sequence out of the elements enclosed by a lower/upper bound in a sorted collection:

```
(subseq (apply sorted-set (range 10)) > 2 < 8) ;
;; (3 4 5 6 7)

(rsubseq (apply sorted-map (range 10)) <= 5)   ;
;; ([4 5] [2 3] [0 1])
```

❶ `subseq` produces a sequence of all elements above 2 and below 8.
❷ `rsubseq` produces a reverse sequence of all the key-value pairs where the key is below or equals to 5.

`rsubseq` differs from `subseq` in the order the sequence is generated: from the first matching element (`subseq`) or from the last (`rsubseq`). `subseq` and `rsubseq` implicitly require the input collection to support a notion of ordering, which restricts the possible input types to sorted-set and sorted-map.

Contract

input

- "sc" is a sorted collection implementing the `clojure.lang.Sorted` interface. There are currently 2 concrete implementations in the standard library: sorted-set and sorted-map.
- "test" can be one of the four comparators: `<` , `⇐` , `>` or `>=` .
- "key" type needs to be comparable with the content of "sc". In most practical situations it means that "key" has the same type of the keys in "sc".

- "start-test", "start-key", "end-test" and "end-key" are of the same type as "test" and "key" respectively. The different name is necessary just when both bounds are present in the function call.

notable exceptions

- `ClassCastException` is thrown when "key" is not comparable with the keys in "sc" or when "sc" is not a sorted collection.

output

- `subseq` : returns the forward sequence of elements enclosed between the lower/upper bound (when both present). When the upper bound is not present, the start/end of the sequence delimits the upper/lower bound implicitly.
- `rsubseq` : works like `subseq` but inverts the order in which the elements are returned in the output sequence.

Examples

`subseq` and `rsubseq` can be used to perform searches of elements above or below a certain key. For example, here's how

to answer the question "What is the smallest/biggest element for which the key is more/less than x":

```
(defn smallest>  [coll x] (first (subseq coll > x
(defn smallest>= [coll x] (first (subseq coll >=
(defn greatest<  [coll x] (first (rsubseq coll <
(defn greatest<= [coll x] (first (rsubseq coll <=

(def coll (sorted-map "a" 5 "f" 23 "z" 12 "g" 1

(smallest>  coll "f") ;; ["g" 1]
(smallest>= coll "f") ;; ["f" 23]
(greatest<  coll "f") ;; ["b" 0]
(greatest<= coll "f") ;; ["f" 23]
```

❶ `smallest>` implementation retrieve the upper sequence beyond the given boundary. The first element is the smallest after the target.

❷ `greatest<` uses `rsubseq` to avoid taking the last from the resulting sequence. Access the last of a sequence is usually not a great idea in terms of performance and we have a straightforward way to avoid it.

The following example shows how to implement auto-completion of words given the first few letters. We can load a

dictionary in a sorted-set as part of the application bootstrap and use it to quickly select a range of words to complete what the user types.

```
(require '[clojure.string :refer [split]])

(def dict
  (into (sorted-set)    ; ❶
    (split (slurp "/usr/share/dict/words") #"\s+

(defn complete [w dict] ; ❷
  (take 4 (subseq dict >= w)))

(map #(complete % dict) ["c" "cl" "clo" "clos" "
;; (("c" "ca" "caam" "caama")
;;  ("clabber" "clabbery" "clachan" "clack")
;;  ("cloaca" "cloacal" "cloacaline" "cloacean")
;;  ("closable" "close" "closecross" "closed")
;;  ("closure" "clot" "clotbur" "clote"))
```

❶ We build a dictionary starting from a list of words (in this case the Unix standard dictionary location). The dictionary is created as a sorted-set.

❷ `complete` takes a word or more likely the first few letters and returns the first 4 words after the fragment from the dictionary.

❸ Here you can see a simulation of the user typing, progressively trying to spell "closure". You can see that the word is first in line at the 5th letter (and it could appear before if we took longer auto-completion lists).

Red-black Trees

Ideal Hash Trees usage in Clojure is well documented in articles and presentations (Phil Bagwell's HAMT tree idea has been adapted to be the basis of persistent data structures in Clojure, you read the original paper here lampwww.epfl.ch/papers/idealhashtrees.pdf). But Clojure also makes practical use of other interesting data structures like Red-black Trees, a type of self-balancing tree which is at the basis of sorted sets and sorted hash maps [154].

In Red-black tree nodes, a bit is dedicated to identify a color (black or red by convention) which helps maintaining the tree balanced during insertion. Inspiration for the current Clojure Red-black tree implementation comes from Okasaki [155] and it was partially described on the Clojure mailing list [156].

Apart from direct use of sorted-set and sorted-map for basic ordering, `subseq` and `rsubseq` are the only functions in the standard library to make explicit use of `clojure.lang.PersistentTreeMap` methods. The reason is the `O(log n)` access guarantee (compared to the linear access in a sequence) to reach the requested

element from where to start generating the sequence. Even assuming an ordered vector or sequence, it would take linear time to reach the requested element with other data structures.

See also

- subs retrieve a sub-string from another larger string.
- subvec creates a sub-vector given a start and end index.
- drop and take can be used with their variant to isolate a portion of a sequence.
- rseq can be used to generate a sequence from the element of a collection in reverse.
- sort is used to order the content of a collection.
- sorted-set and sorted-map store their content ordered by a comparator.

Performance considerations and implementation details

=> `O(log n)` **steps, average**

=> `O(n)` **steps, worst case**

=> `O(n)` **space, average**

`subseq` and `rsubseq` leverage binary tree searching to perform efficient searches. More specifically, they both provide `O(log n)` guarantee in returning the first element matching

the predicate. Any further generation of a sequence after finding the entry point depends in `n` the length of the requested sequence (although this is a worst case when the generated output is fully consumed).

The following example performs a search to find the smallest item appearing after a threshold value "x", showing that `subseq` is much faster when the collection is a `sorted-set` (compared to sorting items in a sequence):

```
(require '[criterium.core :refer [quick-bench]])

(def items (shuffle (range 1e5)))

(let [x 5000 xs (sort items)]
  (quick-bench (first (drop-while #(>= x %) xs))
;; Execution time mean : 88.201576 µs

(let [x 5000 ss (into (sorted-set) items)]
  (quick-bench (first (subseq ss > x)))) ; ❷
;; Execution time mean : 0.767148269 µs
```

❶Before we can search the threshold value, we need to explicitly sort the sequence of "items". We then proceed to drop-while until we reach the threshold and then return the first element. This is a linear scan of the sequence.
❷The same "items" are used to create a sorted-set. We use `subseq` to access the smallest item after reaching "x".

The example shows an evident advantage of `subseq` for these kind of operations (from 88 microseconds to less than a microsecond). What the example does not show though, is the time required to create the sorted-set compared to the creation of the sorted sequence. There is a trade-off to consider when designing an algorithm around `subseq`, which is the way the sorted collection is created and evolved during the lifetime of the application. Performance of `subseq` and `rsubseq` are definitely good for use cases like suggesting words from a dictionary (and optionally adding more words after the initial creation) like presented in this chapter.

9.1.4 seque

`function` `since 1.0`

Listing 9.4. → *Sequence generation, Buffered sequence*

```
(seque
  ([s])
  ([n-or-q s]))
```

`seque` (pronounced "seek") creates an in-memory queue on top of a producer sequence (the name is indeed a mnemonic for "sequence on a queue"). `seque` is used by wrapping over the producer sequence:

```
(seque (range 10)) ; ❶
;; (0 1 2 3 4 5 6 7 8 9)
```

❶ `seque` used on a range produces another lazy sequence with the same content.

Behind the scenes, `seque` creates an asynchronous buffer capable of background computation. `seque` is useful to coordinate consumers operating at different speed than producers, for example to process a sequence fed by a physical device, distributed service or some intensive computation.

Contract

input

- "s" is any collection supporting the sequential interface (seq is going to be called on "s" to produce a sequential view) and it's mandatory argument. It can be empty or `nil`.
- "n-or-q" is optional. When present it can be a positive integer (up to `Integer/MAX_VALUE`) or an object supporting the `java.util.concurrent.BlockingQueue` interface. Although there are several `BlockingQueue` implementations available in Java, `seque` is not compatible with some of them (the `BlockingQueue` should also be ordered, finite and able to accept `nil` as an element). For this reason it's advisable to just use `java.util.concurrent.LinkedBlockingQueue` as queue implementation.

notable exceptions

- `IllegalArgumentException` if "n" is 0, negative.
- `NullPointerException` is thrown in case "n" is `nil`.

output

- `seque` produces the sequence obtained by iterating the elements from "s", possibly parking up to "n" (default 100) items from "s" if the generated sequence is not consumed promptly.
- An empty sequence if "s" is empty or `nil`.

Examples

To understand how `seque` works, let's first explore a simple example without `seque`. The following listing shows a fast producing input sequence created with range and a slow consumer simulated with a `Thread/sleep`. When the fast producer is attached into the slow consumer we see the following:

```
(defn fast-producer [n] ; ❶
  (->> (into () (range n))
       (map #(do (println "produce" %) %))))

(defn slow-consumer [xs] ; ❷
  (keep
    #(do
      (println "consume" %)
      (Thread/sleep 2000))
```

```
      xs))

(slow-consumer (fast-producer 5)) ; ❸

;; produce 4
;; consume 4 ; ❹
;; produce 3
;; consume 3
;; produce 2
;; consume 2
;; produce 1
;; consume 1
;; produce 0
;; consume 0
```

❶ `fast-producer` is a function creating a list, which supports the sequence interface natively (specifically for this example, it doesn't use any optimization that would generate a confusing output). map was added to print each produced element.

❷ `slow-consumer` takes a sequence "xs" as input and simulates some lengthy computation. keep is used here to map over each item and to suppress the `nil` output that would otherwise produce confusion in the printout.

❸ After calling the slow consumer using the fast producer input, we can see each item producing a "produce-consume" pair of lines every 2 seconds.

❹ Each "consume" printout happens after 2 seconds from the related "produce".

We can now add `seque` between producer and consumer. `seque` creates an in-memory buffer to help reducing the need for the producer to wait for the consumer:

```
(slow-consumer (seque (fast-producer 5))) ; ❶

;; produce 4
;; produce 3
;; produce 2
;; produce 1
```

```
  ;; produce 0 ; ❷
  ;; consume 4
  ;; consume 3
  ;; consume 2
  ;; consume 1
  ;; consume 0
```

❶The only addition to the previous example is the `seque` call wrapping the fast producer.

❷ `fast-producer` is now able to move forward without waiting. `slow-consumer` starts catching up after around 2 seconds, slowly consuming items from the input sequence but without a dependency on the fast producer, which has now the opportunity to park resources or do some other work.

Here's a similar example with opposite roles. A slow producer is attached to a fast consumer and `seque` is between them:

```
  (defn slow-producer [n] ; ❶
    (->> (into () (range n))
         (map
           #(do
              (println "produce" %)
              (Thread/sleep 2000) %))))

  (defn fast-consumer [xs] ; ❷
```

```
    (map #(do (println "consume" %) %) xs))

  (first (fast-consumer (seque (slow-producer 5))))

  produce 4 ; ❹
  produce 3
  produce 2
  consume 4
  4
  produce 1
  produce 0
```

❶ `slow-producer` now contains the simulated `Thread/sleep` of 2 seconds.

❷ `fast-consumer` is now free to just map over the input sequence without stopping.

❸We added `seque` between the slow producer and the fast consumer. Note how we only take the first element from the resulting processing pipeline.

❹Each "produce" waits around 2 seconds before moving on to the next.

When `seque` is present between a slow producer and a fast consumer, it allows the producer to work "n" items ahead in the background, even when we only request the first item.



Warning

Note that although similar to sequence "chunking" (an internal optimization that allows sequences to compute some number of items ahead), `seque` operates independently from it and even on sequences that are not necessarily chunked.

A mechanism like `seque` could be used to implement a form of look-ahead pagination. While the consumer is busy with the first results, the computation could continue in the background to prepare the next "n" results:

```
(defn by-type [ext]     ; ❶
  (fn [^String fname]
    (.endsWith fname ext)))

(defn lazy-scan []      ; ❷
  (->> (java.io.File. "/")
    file-seq
    (map (memfn getPath))
    (filter (by-type ".txt"))
    (seque 50)))

(defn go []
  (loop [results (partition 5 (lazy-scan))] ; ❸
```

```
      (println (with-out-str (clojure.pprint/write
      (println "more?")
      (when (= "y" (read-line))
        (recur (rest results)))))

  (go)

  ("/usr/local/Homebrew/docs/robots.txt" ; ❹
   "/usr/local/Homebrew/LICENSE.txt"
   "/usr/local/var/homebrew/linked/z3/todo.txt"
   "/usr/local/var/homebrew/linked/z3/LICENSE.txt"
   "/usr/local/var/homebrew/linked/z3/share/z3/exa
  ;; more?
```

❶ `by-type` is a function to build a predicate for filter below. It returns `true` if the file name ends with the given extension.

❷ `lazy-scan` creates a sequence of files starting from the home folder and only with the given extension. file-seq is providing the initial lazy sequence by following the file system down each available folder. We are asking `seque` to look ahead 50 items on this sequence. Note that even if the folder contains many files or subfolders, the producers only produces some amount of items ahead of time, without triggering an entire file system scan.

❸ This loop asks the user to input "y" to see the next page of results, or any other letter to stop. While the first 5 items are displayed, `seque` is searching for the next 50 in the background.

❹ The list of results could appear different on a different machine.

`seque` also allows to use a custom queue. We could use this feature to print a message every time the buffer is full, an useful information to understand the best buffer size based on the relative speed of the consumer. The buffer indicator runs from a different thread and prints how many items are in the queue every second:

```
(import '[java.util.concurrent LinkedBlockingQue

(def q (LinkedBlockingQueue. 2000)) ; ❶

(defn counter [] ; ❷
  (let [out *out*]
    (future
      (binding [*out* out]
        (dotimes [n 50]
          (Thread/sleep 1000)
          (println "buffer" (.size q)))))))

(defn lazy-scan [] ; ❸
  (->> (java.io.File. "/")
    file-seq
    (map (memfn getPath))
    (filter (by-type ".txt"))
    (seque q)))

(counter) ; ❹
;; #object[clojure.core$future_call$reify__8454 (
;; buffer 0
;; buffer 0
;; buffer 0

(go)

;; ("/usr/local/Homebrew/docs/robots.txt" ; ❺
```

```
;;  "/usr/local/Homebrew/LICENSE.txt"
;;  "/usr/local/var/homebrew/linked/z3/todo.txt"
;;  "/usr/local/var/homebrew/linked/z3/LICENSE.t
;;  "/usr/local/var/homebrew/linked/z3/share/z3/
;; more?

;; buffer 544 ; ❻
;; buffer 745
;; buffer 745
;; buffer 749
;; buffer 749
;; ...
;; buffer 2000
;; buffer 2000
;; ...
```

❶ The blocking queue is stored in a var. To show that `seque` is working in the background, we make a much larger buffer of 2000 files so there is time to print the increasing size of the buffer.

❷ `counter` starts a future that wakes up every second to print the size of the buffer. This happens 50 times before exiting, which is enough to see progress while typing instructions at the REPL.

❸ `lazy-scan` is the same as before except when we build `seque`. Instead of passing the buffer size, we pass the queue instance directly.

❹ We start the `counter` first and we can see it printing "buffer 0" every second.

❺ As soon as we invoke the `(go)` function, we can see the first page of results (this could be different on different machines).

❻ The `counter` continues in the background, showing progress while the buffer is filling up from 0 to 2000 items. If we wait long enough without doing anything, we can see it printing 2000 continuously, sign that the buffer is full and no more I/O is continuing on disk.

By tweaking the buffer-size we have a way to deliver "back-pressure" to the producing sequence [157]. If the producing sequence is lazy (like our case) the file scan pauses when the

buffer is full waiting for pagination to presents all the 2000 results.

A bit of history: `seque` original design

Like other functions in the standard library, `seque` was designed as a collaboration between early Clojure adopters and Rich Hickey. A first discussion about creating a sequence to handle asynchronous XML processing was posted by Chris Houser on the Clojure mailing list in 2008 (groups.google.com/d/msg/clojure/5_TeVm7dtuo/lunp2qAyAFkJ). After a few refinements on the #Clojure IRC channel (web.archive.org/web/http://clojure-log.n01se.net/date/2008-06-30.html#10:44c) `seque` was born.

The version of `seque` now part of the standard library is designed differently from that first idea around XML parsing. The current `seque`, for instance, is not capable of handling asynchronous callbacks (a type of problem that is now solved elegantly with `core.async`).

The following is an attempt at solving the original problem of building a sequence from data coming from a queue. The queue can be shared across and passed to callbacks:

```
(import '[java.util.concurrent LinkedBlockingQueue])
(def q (LinkedBlockingQueue. 5))  ; ❶
(def sentinel (Object.))          ; ❷

(defmacro start [& body]          ; ❸
  `(let [out# *out*]
    (future
      (binding [*out* out#] ~@body))))

(defn producer [^LinkedBlockingQueue q items] ; ❹
  (start
```

```
    (loop [[x & xs :as items] items]
      (Thread/sleep 1000)
      (let [x (or x sentinel)]
        (println "adding" x)
        (if (.offer q x)
          (when-not (identical? x sentinel) (recur xs))
          (recur (or items sentinel)))))))

(defn seque2 [^LinkedBlockingQueue q] ; ❺
  (lazy-seq
    (let [x (.take q)]
      (cons
        (if (identical? x sentinel) nil x)
        (seque2 q)))))

(defn consumer [q]                ; ❻
  @(start
    (map
      #(do
         (println "consume" %)
         (Thread/sleep 1000))
      (seque2 q))))
```

❶ An instance of `LinkedBlockingQueue` is shared using a var. Note max queue size is set to 5, a small number to help us understanding the example. In a real scenario, this number would have to be adjusted to the memory capacity and relative speed between consumers/producers.

❷ `sentinel` is a sentinel object used by the producer to signal that there are no more items available. A "sentinel" is an useful pattern to mix control signals with data. Consumer and producer needs to agree on the sentinel object ahead of time, which needs to be different from any instance of the data.

❸ The macro `start` wraps future to make sure the created thread uses the main standard output. We need a separate thread to be able to print to the same standard output for demonstration purposes.

❹ The callback is simulated by the `producer` function. The main idea is to give the queue reference to any callback that needs to place items in it. `producer` simulates the generation of items from a fixed "items" argument in a separate thread. In a real scenario, new items appear as calls to the callback with an "event" argument.

❺ `seque2` creates a lazy sequence on top of a blocking queue, stopping only when the `sentinel` is received. Laziness is created with the standard lazy-seq and cons pattern.

❻ Finally, a `consumer` is started in another thread. The `consumer` uses `seque2` to create a sequential view on the blocking queue.

The consumer and the producer in this example are slowed down on purpose, so we can play with them. Here's what happens when we start the producer:

```
(producer q (range 8)) ; ❶
;; adding 0
;; adding 1
;; adding 2
;; adding 3
```

```
;; adding 4
;; adding 5
;; adding 5
;; adding 5

(take 3 (consumer q)) ; ❷
;; consume 0
;; adding 5
;; consume 1
;; adding 6
;; consume 2
;; adding 7

(take 5 (consumer q)) ; ❸
;; consume 3
;; consume 4
;; consume 5
;; consume 6
;; consume 7
```

❶The `producer` sends 8 items to the queue at 1 second intervals. On reaching the 5th number, the queue is at capacity and the same item is retried indefinitely.
❷The `consumer` takes 3 items out of the queue. As soon as this happens, the `producer` can send all the remaining items down the queue. The `producer` sends the sentinel object down the queue as well.
❸We now takes the other 5 items out of the sequence. When the `consumer` sees the sentinel, the sequence terminates and the control returns to the main thread. Any additional requests to take items from the `consumer` sequence would now block, waiting for more items to be available.

Although this is a possible idea to solve the problem of building a sequence on a blocking queue, libraries like `core.async` (github.com/clojure/core.async) provides a robust solution to the problem, which should be evaluated before rolling a custom solution like the one presented above.

See also

- sequence can create a sequential view on a `LinkedBlockingQueue` following a standard, non-blocking, approach.

- lazy-seq is the fundamental building block for `seque`.
  Is worth revisiting the mechanism by which `lazy-seq` allows the creation of lazy sequences to understand `seque`.

Performance considerations and implementation details

=> `O(n)` **Worst case, time and memory**

`seque` follows general sequential processing rules: in case of a fully consumed output, `seque` needs to process all items from the input sequence producing a typical linear behavior.
Laziness helps avoiding the worst case scenario when the input is not fully consumed.

Lazy sequences cache results. If the head of the output is retained and fully consumed, `seque` produces a sequence that contains a full copy of the input (this is of course the worst case scenario). The size of the buffer can also contribute linearly to memory occupation: the larger the buffer the more items can potentially reside in memory, especially when the consumer is not fast enough.

The implementation of `seque` is an early example of agent:

- An agent iterates the input sequence in a loop. It processes enough of the sequence that can fit in a `LinkedBlockingQueue` instance. The remaining of the sequence is stored as the new state of the `agent` and the task exits.
- While the `agent` is filling the queue, the main thread of computation is building a lazy sequence off the queue. Each time the loop is able to take an item off the queue, another "fill" requests is sent to the `agent`. Each task sent to the `agent` resumes from the remaining part of the input sequence.
- Any error produces a retry from the previous state.
- Sentinels objects are used to signal the end of the input which propagates to the lazy sequence that eventually returns `nil`.

9.1.5 pmap, pcalls and pvalues

`function: pmap and pcalls` `macro: pvalues` `since 1.0`

Listing 9.5. → *Parallel computation, Lazy sequences*

```
  (pmap
    ([f coll])
```

```
    ([f coll & colls]))

  (pcalls [& fns])

  (pvalues [& exprs])
```

`pmap`, `pcalls` and `pvalues` build a lazy sequence as the result of processing a set of expressions in parallel (using futures). Both `pcalls` and `pvalues` build on top of `pmap`. `pmap` has a similar interface to map, but transformations apply to the input in parallel:

```
  (pmap + (range 10) (range 10)) ; ❶
  ;; (0 2 4 6 8 10 12 14 16 18)
```

❶ `pmap` has a similar interface to map but the input transformations happen in parallel.

`pcalls` builds on top of `pmap` accepting any number of functions as input. It then creates a lazy sequence from the results of calling the functions without arguments. `pcalls` is a good solution for side effecting parallel transformations:

```
  (pcalls          ; ❶
    (constantly "Function")
```

```
    #(System/currentTimeMillis)
    #(println "side-effect"))
  ;; side-effect  ; ❷
  ;; ("Function" 1553770187108 nil)
```

❶ `pcalls` expects functions of no arguments, presumably for side effects.

❷ A side effecting function produces the string "side-effect" on screen, while the others produces an output different from `nil`.

`pvalues` is a macro also building on top of `pmap`. It takes any number of expressions that are evaluated in parallel to generate a sequence of the results:

```
(pvalues ; ❶
  (+ 1 1)
  (Math/sqrt 2)
  (str "last" " " "item"))

;; (2 1.4142135623730951 "last item") ; ❷
```

❶ `pvalues` is a macro, allowing deferred and parallel evaluation of the expressions in the input.
❷ The result is the lazy sequence of evaluations of the expressions in the input.

All the functions, `pmap`, `pcalls` and `pvalues`, produce a lazy sequential output which corresponds to the ordered evaluation of the items in the input.

Contract

input

- "f" in `pmap` is mandatory argument. "f" must be a function of one or more arguments. The number of arguments corresponds to the number of input collections.
- "coll" in `pmap` is mandatory argument. "coll" needs to provide a sequential view, such that `(instance? clojure.lang.Seqable coll)` is `true`.
- "colls" in `pmap` means that any number of additional collection is accepted after "coll". The number of collections determines the required arity for "f".
- "fns" in `pcalls` is zero or more functions of no arguments.

- "exprs" in `pvalues` is any number of valid Clojure expressions, including an empty list of expression arguments.

notable exceptions

- `ArityException` when invoking `pmap` without at least one "coll". Note: there is no transducer version of `pmap`.

output

- `pmap` returns the lazy sequence containing the result of applying "f" to all the elements in the input collection. Please see map for general considerations about the presence of multiple collections.
- `pcalls` returns the lazy sequence containing the result of invoking each input function without arguments.
- `pvalues` returns the lazy sequence containing the evaluation of all expression arguments.

Examples

`pmap` achieves easy and immediate parallelism with the same map interface. However, there are good reasons to avoid

replacing every use of `map` with `pmap`:

- The computational cost of the transformation function (or the arguments in case of `pcalls` / `pvalues`) should be substantial. In that case, there are good chances the thread orchestration cost is going to outweigh any performance benefit.
- `pmap` produces the output in the same order as the input by performing ordered batches of parallel computation. If one input produces significantly more work than another in the batch, `pmap` needs to wait before moving on to the next batch. The presence of a longer computation in a batch decreases the level of parallelism (see the performance section for an example).

Even taking into account the constraints above, a sufficiently large application usually contains some part of the code justifying the need for `pmap`. One of such case is when processing large datasets, for example the documents resulting from querying ElasticSearch [158] or some other service. The reader is invited to review a few solutions already using `pmap` illustrated by the book:

- xml-seq contains an example processing of large documents that uses `pmap` to speedup processing. Each document is roughly the same length and the transformation is not trivial.
- partition also contains an interesting example of `pmap` using aggregation to group smaller tasks into larger ones.

In those cases where it makes sense to use `pmap`, depending on the type of transformation "f" and the input size, we can control the thread orchestration by using partution-all. By grouping items, we create partitions of sequential processing that could improve performance:

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (range 10000)]       ; ❶
  (quick-bench
    (last (map eval xs))))
;; Execution time mean : 23.182619 ms

(let [xs (range 10000)]       ; ❷
  (quick-bench
    (last (pmap eval xs))))
;; Execution time mean : 19.001539 ms
```

```
(let [xs (range 10000)]       ; ❸
  (quick-bench
    (last (last
      (pmap
        #(map eval %)
        (partition-all 1000 xs))))))
;; Execution time mean : 3.208768 ms
```

❶"xs" is a relatively large sequence. We use eval to simulate a non-trivial computation. Calling `eval` on a number just produce the same number. We can see that it takes around 23ms on average to process the input sequence.

❷We decide to give `pmap` a go, assuming `eval` is expensive enough to justify the thread orchestration cost. This is true, but the advantage is minimal, taking around 19ms on average.

❸By partitioning the input we trade some of the parallelism in exchange for a reduced thread overhead that in this case is definitely paying off.

In cases where `pmap` advantage seems minimal compared to the sequential case, it's worth testing if partitioning the input produces positive effects. However, the reader should always remember to carefully benchmark such assumptions.

Understanding `pmap`

One frequent question about `pmap` is how many threads are actually working in parallel. We can't control `pmap` parallelism directly, but we have some control over the chunk size of the input sequence (the other option being increasing the number of CPU cores). The easiest case to understand is to assume no chunking in the input sequence:

```
(defn dechunk [xs] ; ❶
  (lazy-seq
    (when-first [x xs]
      (cons x
        (dechunk (rest xs))))))

(defn f [x] ; ❷
  (Thread/sleep (+ (* 10 x) 500))
  (println (str "done-" x))
  x)

(def s (pmap f (dechunk (range 100)))) ; ❸
;; done-0
;; done-1

(first s) ; ❹
0
;; done-2
;; done-3
;; done-4
;; done-5
;; done-6

(take 2 s) ; ❺
(0 1)
;; done-7
```

❶ `dechunk` creates a lazy sequence on top of another, removing any chunking for consumers upstream.

❷ We call `pmap` using a tracing function that prints a message after a sleep period that slightly increases on each call. The increasing sleep time gives time to println to flush the entire string to standard output, so we can see each message appearing on a different line (they would interleave otherwise).

❸ Interestingly, two threads start at definition time, even if we don't consume elements from the sequence. This is a byproduct of how `pmap` implementation destructures input internally. In general, this shouldn't be a problem, unless you're searching for maximum laziness.

❹ As soon as we take the first item, `pmap` goes ahead with the computation. The expression was evaluated on a 4 cores machine. `pmap` is designed to stay `(+ 2 N-cores)` ahead of the requested item. The requested item is the first, 2 items have been already computed before, there are 4 more items evaluated.

❺ From this point onward, following requests move the head of the computation forward one element, starting a new future each time.

If the input sequence is not chunked, `pmap` stays `(+ 2 N-cores)` ahead of the requested item, providing readily cached results for incoming requests. If we suddenly request the last element, `pmap` guarantees it would never go beyond `(+ 2 N-cores)` concurrent requests. The situation changes with chunked sequences, as the next request for an item might result in the entire chunk getting realized:

```
(def s (pmap f (range 1000))) ; ❶
;; done-0
;; ...
;; done-31
```

```
  (first s) ; ❷
  0

  (first (drop 26 s)) ; ❸
  ;; done-32
  ;; ...
  ;; done-63
```

❶The call to `dechunk` was removed from the creation of the sequence "s" with `pmap`. range produces a chunked sequence with chunk size 32. As soon as the expression is evaluated, 32 futures start the computation for the first chunk.

❷The first element has already been computed, as well as the following `(+ 2 N-cores)`, so nothing prints this time.

❸In order to see the next chunk of 32 threads, we need to drop at least `(- chunk-size 2 N-cores)` from the sequence (that number is 26 when the number of cores is 4).

The following rules can be used to understand how many threads `pmap` runs at once (assuming tasks are roughly the same computational cost). The min level correspond to the situation where the consumer is slower than the producer, while the max level is when the consumer is faster than the producer:

- When the sequence is not chunked (for example subvec) the min parallelism is 1 and the max parallelism is `(+ 2 N-cores)`. Example: with 12 cores, `(doall (pmap #(Thread/sleep %) (subvec (into [] (range 1000)) 0 999)))` keeps 12+2 threads busy.
- In case of chunked sequences (vast majority are size 32), the min parallelism is `(min chunk-size (+ 2 n-cores))`, while the max amount is equal to `(+ chunk-size 2 N-cores)`. Example: with 12 cores, `(doall (pmap #(Thread/sleep %) (range 1000)))` keeps 12+2+32 threads busy.

With those rules in mind, by changing the chunk size, we can get any grade of parallelism:

```
(defn re-chunk [n xs] ; ❶
  (lazy-seq
    (when-let [s (seq (take n xs))]
      (let [cb (chunk-buffer n)]
        (doseq [x s] (chunk-append cb x))
        (chunk-cons (chunk cb) (re-chunk n (drop

(def s (pmap f (re-chunk 1000 (range 1000)))) ;
```

❶ `re-chunk` takes a (possibly already chunked) sequence and produces another chunked sequence with a different chunk-size.

❷ This `pmap` definition starts 1000 concurrent threads straight away. Use with caution!

Sequences with a custom chunk size are rare but possible. If your application implements one, it uses `pmap` and the chunk size is in the thousands, there is some possibility to saturate the unbounded future's thread pool and lose control of the JVM. Something to keep in mind.

See also

- fold is the main entry point into a different model of parallel computation called "fork-join". `fold` is designed to handle some variance in computational complexity thanks to an algorithm called "work-stealing".
- future is the threading primitive used by `pmap` to send computations off to parallel threads.

Performance considerations and implementation details

=> `O(n)` **linear(size of the input)**

`pmap` computational cost is linear with the size of the input. As other lazy functions, assuming we fully consume the input, memory occupation is going to be linear in the size of the input. The only case memory allocation becomes `O(1)` is when we don't hold onto the head of the sequence. Please see map for more information about this aspect.

`pmap` futures orchestration has a cost. When the transformations (or expressions) are trivial, the cost of orchestrating the threads outweighs the performance benefits of parallel computation. The following typical example, shows the degradation introduced on a trivial computation by misusing `pmap`:

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (range 100000)] (quick-bench (last (map
;; Execution time mean : 4.651943 ms

(let [xs (range 100000)] (quick-bench (last (pma
;; Execution time mean : 325.748151 ms
```

❶This simple transformation increments each number in a large sequence.
❷The equivalent with `pmap` is roughly 100 times slower.

The situation does not necessarily improve with aggregation when the computation is too trivial. However, if we repeat the same example with aggregation, we can see how `pmap` is now only 2 times slower:

```
(let [xs (partition-all 1000 (range 100000))]
  (quick-bench
    (into [] (comp cat (map inc)) xs))) ; ❶
;; Execution time mean : 6.553814 ms

(let [xs (partition-all 1000 (range 100000))]
  (quick-bench
    (into [] cat (pmap #(map inc %) xs)))) ; ❷
;; Execution time mean : 13.539197 ms
```

❶ `patition-all` groups input items into inner sequences of 1000 items each. We can use into with transducers to process each item in the inner sequence an remove inner groups with cat.

❷ `pmap` is now only 2 times slower.

Another aspect to consider is uniformity of the computational cost across the input. The following example distributes 10 long computation across an input sequence of 320 items. The size of

the input and the distribution of the long tasks exacerbate the dependency of `pmap` on chunked sequences:

```
(def xs (map #(if (zero? (mod % 32)) 1000 1) (ra

(time (dorun (map #(Thread/sleep %) xs))) ; ❷
;; "Elapsed time: 10019.599748 msecs"

(time (dorun (pmap #(Thread/sleep %) xs))) ; ❸
;; "Elapsed time: 10024.762327 msecs"
```

❶The input sequence "xs" contains 320 repetition of the number 1, but every 32 items the 1 was replaced by 1000. This means that "xs" contains exactly 1 occurrence of the number 1000 for each chunk of size 32.

❷We first run normal map with a function waiting the number of milliseconds as indicated by the input item. As expected, the sequential execution lasts roughly 10 seconds.

❸The second run uses `pmap` instead of `map`. The execution lasts again 10 seconds despite the parallelism.

The example above is the worst possible scenario for `pmap` that executes exactly like the sequential case. But if we push multiple occurrences of the number 1000 inside the same

chunk (the highest the number the slowest the task), we give `pmap` the opportunity to execute them in parallel:

```
(time (dorun (pmap #(Thread/sleep %) (sort xs)))
;; "Elapsed time: 1028.686387 msecs"
```

❶The only change compared to the previous example is sorting the input.

After sorting the input we achieve the effect of compacting all long running tasks inside the same chunk, showing that uniformity is key for `pmap`.

[152] en.wikipedia.org/wiki/Palindromic_sequence

[153] On the topic of efficient DNA processing, please see www.ncbi.nlm.nih.gov/pmc/articles/PMC3602881/

[154] Please refer to this general introduction to Red-black trees available on Wikipedia: en.wikipedia.org/wiki/Red–black_tree

[155] "Purely Functional Data Structures" is an important book in functional programming that describes ways to implement the most common data structures persistently, that is, with structural sharing preserving older versions

[156] See groups.google.com/forum/#!msg/clojure/CqV6smX3R0o/_Znnim

boYjQJ

[157] Back-pressure is an important concept in event driven systems, where we want downstream components to be able to limit upstream producers. This talk from Zach Tellman www.youtube.com/watch?v=1bNOO3xxMc0 does a wonderful job introducing the key concepts.

[158] ElasticSearch is a popular document store. Queries to ElasticSearch can retrieve potentially large list of documents in JSON format.

9.2 ABSTRACT GENERATORS

This section groups together functions that generate sequences without being driven by a source of sequential data. They use an algorithmic recipe instead, such as calling a function or using a fixed set of items. range (which is about generating sequence of numbers) should be included in this set, but it's so widely used that it was given prominence in the chapter about the basic constructs.

Here's a basic summary:

- repeatedly calls a function without arguments to obtain the next element for the sequence.

- iterate calls a function and use the results to invoke the same function again to obtain the next element for the sequence.
- repeat repeats the same input items to produce a sequence.
- cycle repeats the same input items in order to produce a flattened sequence.

Implementation notes

Generators like iterate, repeat, cycle and range have a dedicated Java implementation as part of the Clojure internals (while repeatedly is the only one written in pure Clojure). There are good performance reasons to implement them in Java (especially to provide a fast path for reducing and transducing), but as an exercise in functional design, here's how they would be implemented in Clojure with cons and lazy-seq:

```
(defn repeatedly* [f] ; ❶
  (lazy-seq (cons (f) (repeatedly* f))))

(defn iterate* [f x] ; ❷
  (lazy-seq (cons x (iterate* f (f x)))))

(defn repeat* [x] ; ❸
  (lazy-seq (cons x (repeat* x))))

(defn cycle* [coll] ; ❹
  ((fn step [[x & xs]]
     (lazy-seq
       (if x
         (cons x (step xs))
         (cycle* coll)))) coll))
```

```
(defn range* [n] ; ❺
  ((fn step [x]
     (lazy-seq
       (when (< x n)
         (cons x (step (inc x)))))) 0))
```

❶ `repeatedly*` is the only function from this set already implemented in Clojure. The implementation has been copied over as a model for the others. We can see that in order to create the sequence we need to "cons" `(f)` on each iteration.

❷ `iterate*` treats the function differently, as we want the result of each `(f x)` to become the input for the next iteration. `(f x)` is invoked on iteration and we "cons" the value that was returned before iterating again.

❸ `repeat*` does not use a function. We just "cons" the input over and over at each iteration while generating the lazy sequence.

❹ `cycle*` works similar to `repeat*` but the fact that the input collection needs to be iterated requires an additional inner loop wrapped by the `step` function. The outer recursion starts with the `step` function being invoked with "coll" the initialization parameter. After destructuring the input we verify if we have more items coming from "coll" and in that case we "cons" the first of them into the sequence. In case we don't have more items, we start a new outer cycle using "coll" again.

❺ `range*` presented here is a simplified version of the function in core. Similarly to `cycle*` we need use an inner recursion to maintain an incrementing counter. Each `step` recursion we "cons" the current number into the sequence and call `step` again incrementing it. We are done when the counter is equal to "n".

What follows is a more formal treatment for each of the functions in this section.

9.2.1 repeatedly

`function` `since 1.x`

Listing 9.6. → *Function application, Sequence generation*

```
(repeatedly
  ([f])
  ([n f]))
```

`repeatedly` generates an infinite lazy sequence by calling the same function with no arguments and collecting the results:

```
(take 3 (repeatedly rand)) ; ❶
;; (0.2416205627046507 0.8326807316362209 0.9275
```

❶rand returns a random `double` between 0 and 1. `repeatedly` calls `rand` each time producing a different number which is collected as a lazy sequence. Remember to use take to avoid printing infinite numbers on screen.

`repeatedly` is useful to create sequences from functions with side effects. With a pure function (a function returning the same output given the same input) it would return a repetition of the same item over and over (and repeat already exists for that use case).

`repeatedly` also takes a number "n" of repetition to perform before stopping:

```
(repeatedly 3 #(if (> (rand) 0.5) true false)) ;;
;; (true false true)
```

❶An example showcasing the number "n" of repetition to produce. The side effecting function converts random numbers into sequence of `true` or `false`.

Contract

input

- "f" is a function of no arguments and possibly side-effecting to return different results on each invocation.
- "n" is optional. When present, it is expected to be a positive number or zero. `double` numbers are rounded up to the nearest integer.

notable exceptions

- `NullPointerException` if "f" or "n" is `nil`.

output

`repeatedly` returns the sequence generated by calling "f" with no arguments "n" or an infinite amount of times if "n" is

not present.

Examples

`repeatedly` is used throughout the book for random data generation. One interesting example is about the generation of proverbs while discussing rand-nth. Other examples generate streams of random numbers, but we could also generate strings or keywords with gensym:

```
(zipmap (map keyword (repeatedly gensym)) (range
;; {:G__52 0, :G__53 1, :G__54 2, :G__55 3, :G__
```

❶We use `repeatedly` and gensym to generate more interesting keys for a map.

Another example of side effecting functions are futures. A future takes a form body without evaluating it and sends the form to a separate thread for evaluation. Using this knowledge, we can build an infinite lazy sequence of workers. When we need more workers, we take from the sequence and start the concurrent computation. The resulting sequence contains the results of the computation:

```
(import '[java.util.concurrent ConcurrentLinkedQ
(def q (ConcurrentLinkedQueue. (range 1000))) ; 
(def ^:const parallel 5)

(defn task [job]    ; ❷
  (Thread/sleep (rand-int 2000))
  (println "Work done on" job)
  (inc job))

(def workers        ; ❸
  (repeatedly
    #(let [out *out*]
       (future
         (binding [*out* out]
           (when-let [item (.poll q)]
             (task item)))))))

(defn run [workers] ; ❹
  (println "-> starting" parallel "new workers")
  (let [done? #(> (reduce + (remove nil? %)) 30)
        futures (doall (take parallel workers))
        results (mapv deref futures)]
    (cond
      (done? results) results
      (.isEmpty q) (println "Empty.")
      :esle (recur (drop parallel workers)))))

(run workers)               ; ❾
```

```
;; -> starting 5 new workers
;; Work done on 0
;; Work done on 1
;; Work done on 2
;; Work done on 3
;; Work done on 4
;; -> starting 5 new workers
;; Work done on 5
;; Work done on 6
;; Work done on 7
;; Work done on 8
;; Work done on 9
;; [6 7 8 9 10]
```

❶ We need a concurrent data structures to hold the input for the workers. Workers compete to take an item from the data structure to produce a result. `java.util.concurrent.ConcurrentLinkedQueue` is a good choice, as in a real life scenario we could push more input into this queue while the workers are running. For this example we push integers into the queue.

❷ The `task` is the core job of each worker. The example simplifies the task by just waiting some number of milliseconds, printing a message and returning the incremented integer.

❸ `workers` builds an infinite sequence of workers ready to work on tasks. The function passed to `repeatedly` needs to establish a binding between the standard output of the main thread and the one in the future. This is done here to show messages on screen. The body of the future invocation takes one item from the queue and produce a result.

❹ `run` orchestrates how many workers should be put at work and when we should stop.

❺ `done?` is a predicate that decides if the current set of results satisfies the global condition to stop and return the results we found. In this example we check if the sum of the results is above 30 to finish the computation.

❻Each iteration we take and realize some numbers of futures (5 in this example, but it would likely be an higher number of concurrent thread in a real life scenario). `(doall (take parallel workers))` realizes the first 5 elements from the infinite sequence returned by `repeatedly`, which means 5 concurrent threads will be at work. doall is necessary to actually kick-off the threads as the `take` operation would be lazy without.

❼mapv and deref ensures the futures are done before checking the results (an equivalent of a "join" operation to wait all threads).

❽If we are done with the results, we return them so they can be inspected. If the queue is empty, we stop the recursion and return no results. If we can proceed, we recur after dropping the first 5 workers we just used.

❾We can see no output messages before running the loop, confirming that the entire design is fully lazy and no threads are in flight before we start.

After calling `run` we can see batches of 5 workers printing messages. After reaching the exit condition, `run` prints the batch that produced the right result.

See also

- repeat produces a sequence starting from a single repeating value or expression, instead of a function of no arguments.
- dotimes does not produce a sequence, but executes the body the given amount of time for side effects. Use `dotimes` when the computation is purely for side effect and there are no results to be collected.
- iterate passes the result of the previous invocation as the input for the next one. Use `iterate` when there is a relationship between the results of each invocation.
- rand, future or atom are typical target for `repeatedly`, typically appearing as part of the function passed as argument.
- lazy-seq is `repeatedly` building block. General considerations about laziness discussed in `lazy-seq` applies to `repeatedly` as well.

Performance considerations and implementation details

=> `O(n)` **linear (worst case, fully consumed)**

=> `O(n)` **memory (worst case, retaining head)**

`repeatedly` generates the sequence in constant time, as it performs no actual iteration until a consumer requests items. As items are requested, `repeatedly` creates a lazy sequence that caches results. The memory footprints is thus linear in the number of items requested (assuming we maintain a reference to the head of the sequence).

Other considerations about laziness (like the risk involved holding the head of the sequence or recursive nesting of lazy sequences) are similar to those discussed in the lazy-seq chapter. `repeatedly` generates a potentially infinite sequence of items that can be consumed and removed from memory as far as we don't hold the head of the sequence. This test was executed imposing the JVM a deliberately small heap size to quickly prove the point:

```
(let [rands (repeatedly 1e7 rand)] (first rands)
;; 0.5900717554100915
(let [rands (repeatedly 1e7 rand)] (last rands)
;; OutOfMemoryError GC overhead limit exceeded
```

❶We generate a large sequence of random numbers. When we access the last item, there is nothing else before the end of the form that needs to consume the sequence, so the items in the sequence can be iterated and discarded, giving a change to the garbage collector to remove them from memory.

❷The second example asks again for the last item, but we also want to access the first at the end of the evaluation of the form. The sequence cannot be garbage collected and needs to stay in memory to satisfy this last request. At the same time, the previous last request completely evaluated the sequence, caching all random numbers in memory. We can see the typical message of `OutOfMemoryError` because the garbage collector is unable to free memory fast enough for other items to be cached.

As mentioned at the beginning of the generators section, `repeatedly` does not have a fast reducing/transducing path like the other generators. There are the following consequences:

1. When invoking reduce or any transducing functions (like transduce or into) the iteration happens through the lazy sequence without optimizations (while other generators have custom reducing paths).

2. Results while reducing are cached, so multiple reduce calls on the same sequence are also cached.

The last point about caching during reduction is important for a function like `repeatedly` designed to work with side effects: if results were not cached, you could see different results each call. Other generators working with pure functions can instead skip caching and gain speed during reduce or transduce.

9.2.2 iterate

`function` `since 1.0`

Listing 9.7. → *Recursive function call, Sequence generation, Laziness*

```
(iterate f x)
```

`iterate` takes a function "f" and an initial parameter "x" and invokes "f" over "x". The result of the invocation is used as the new parameter to invoke "f" again, which returns the next result and so on, gradually building a sequence of the results:

```
(take 10 (iterate inc 0)) ; ❶
;; (0 1 2 3 4 5 6 7 8 9)
```

❶ `iterate` is used here to simulate range. 0 is the explicit start of the sequence, then `(inc 0)` produces 1, `(inc 1)` produces 2 and so on, until we take 10 elements.

One of the main use case for `iterate` is the generation of arbitrarily complex lazy sequences to be used as input into other sequential processing.

Contract

input

- "f" is a function of one argument and is a mandatory argument. Compared to repeatedly, "f" is expected to be free of side effects.
- "x" is the argument for "f" and can be of any type. It is also a required argument.

notable exceptions

- `NullPointerException` when "f" is `nil`.
- `ClassCastException` when "f" is not a function (does not implement `clojure.lang.IFn`).

output

- returns: the infinite lazy sequence of `x`, followed by `(f x)`, `(f (f x))` and so on. `iterate` returns a `clojure.lang.Iterate` object which is a type of sequence (implementing the `clojure.lang.ISeq` interface).

Examples

One of the classic example of `iterate` is the function producing the Fibonacci numbers. The series start with a fixed "0,1" pair and then numbers that follow are the sum of the previous:

```
(def fibo
  (iterate ; ❶
    (fn [[x y]] [y (+' x y)])
    [0 1]))

(take 10 (map first fibo)) ; ❷
;; (0 1 1 2 3 5 8 13 21 34)
```

❶The design of the function follows quite straightforward from the definition of the series. The initial argument for `iterate` is the given pair [0 1]. The iteration shifts the pair forward by using the sum of the elements as the new item in the next pair. Note the use of `+'` to enable automatic promotion to `clojure.lang.BigInt`. This is useful for Fibonacci numbers that grow quite quickly.

❷The vector pair in the sequence carries all the information needed for the next iteration, but to extract the actual results we need to only take the first element.

`iterate` shines with series where the previous element has a relationship with the next, like Fibonacci or the inverse tangent series at the basis of the Leibniz approximation of Pi [159]. We saw the formula already in filterv in its sequential version. `iterate` has a custom reduce implementation and we want to take advantage of that by rewriting the formula to use transduce:

```
(defn calculate-pi [precision] ; ❶
  (transduce
    (comp
      (map #(/ 4 %))
      (take-while #(> (Math/abs %) precision)))
    +
```

```
      (iterate #(* ((if (pos? %) + -) % 2) -1) 1.0

  (calculate-pi 1e-6) ; ❷
  ;; 3.141592153589724
```

❶ `calculate-pi` is a rewrite of the function presented in filterv to calculate the approximation of Pi. The transformation of the formula to use transduce is straightforward: the source becomes the target for `iterate` and processing is now part of the composition of transducers.

❷ We can see how to calculate an approximation of Pi. Decimals are correct up to the 6th decimal digit. Note that asking for additional precision requires exponentially more time.

`iterate` can be generalized to any process in which state depends on the previous state, not necessarily simple numbers. We already created a Game of Life implementation when talking about for, but we never used it for a full simulation. The reader is invited to review that implementation, but for the purpose of iterating states of the Game of Life we are only interested in the `next-gen` function.

`next-gen` takes height and width of the grid where cells live and an initial Set of alive cells. It returns the next state of also

as a `set`. We are going to `iterate` some amount of `next-gen` states and print them:

```
;; please see "next-gen" from the "for" chapter.

(defn grid [h w cells] ; ❶
  (letfn [(concats [& strs]
            (apply str (apply concat strs)))
          (edge [w]
            (concats
              " "
              (repeat (* 2 w) "-")
              " \n"))
          (row [h w cells]
            (concats
              "|"
              (for [x (range w) :let [y h]]
                (if (cells [x y]) "<>" "  "))
              "|\n"))]
    (concats
      (edge w)
      (for [y (range h) :let [x w]]
        (row y x cells))
      (edge w))))

(defn life [height width init] ; ❷
  (iterate (partial next-gen height width) init)
```

```
(def pulsar-init ; ❸
  #{[2 4] [2 5] [2 6] [2 10] [2 11] [2 12]
    [4 2] [4 7] [4 9] [4 14]
    [5 2] [5 7] [5 9] [5 14]
    [6 2] [6 7] [6 9] [6 14]
    [7 4] [7 5] [7 6] [7 10] [7 11] [7 12]
    [9 4] [9 5] [9 6] [9 10] [9 11] [9 12]
    [10 2] [10 7] [10 9] [10 14]
    [11 2] [11 7] [11 9] [11 14]
    [12 2] [12 7] [12 9] [12 14]
    [14 4] [14 5] [14 6] [14 10] [14 11] [14 12]}

(defn pulsar [] ; ❹
  (let [height 17 width 17 init pulsar-init]
    (doseq [state (take 3 (life height width ini
      (println (grid height width state)))))
```

❶The `grid` function contains all the code necessary to format the life grid. A living cell is printed with "<>" else the cell is left blank. The function also takes care of printing horizontal and vertical edges to enclose the grid.

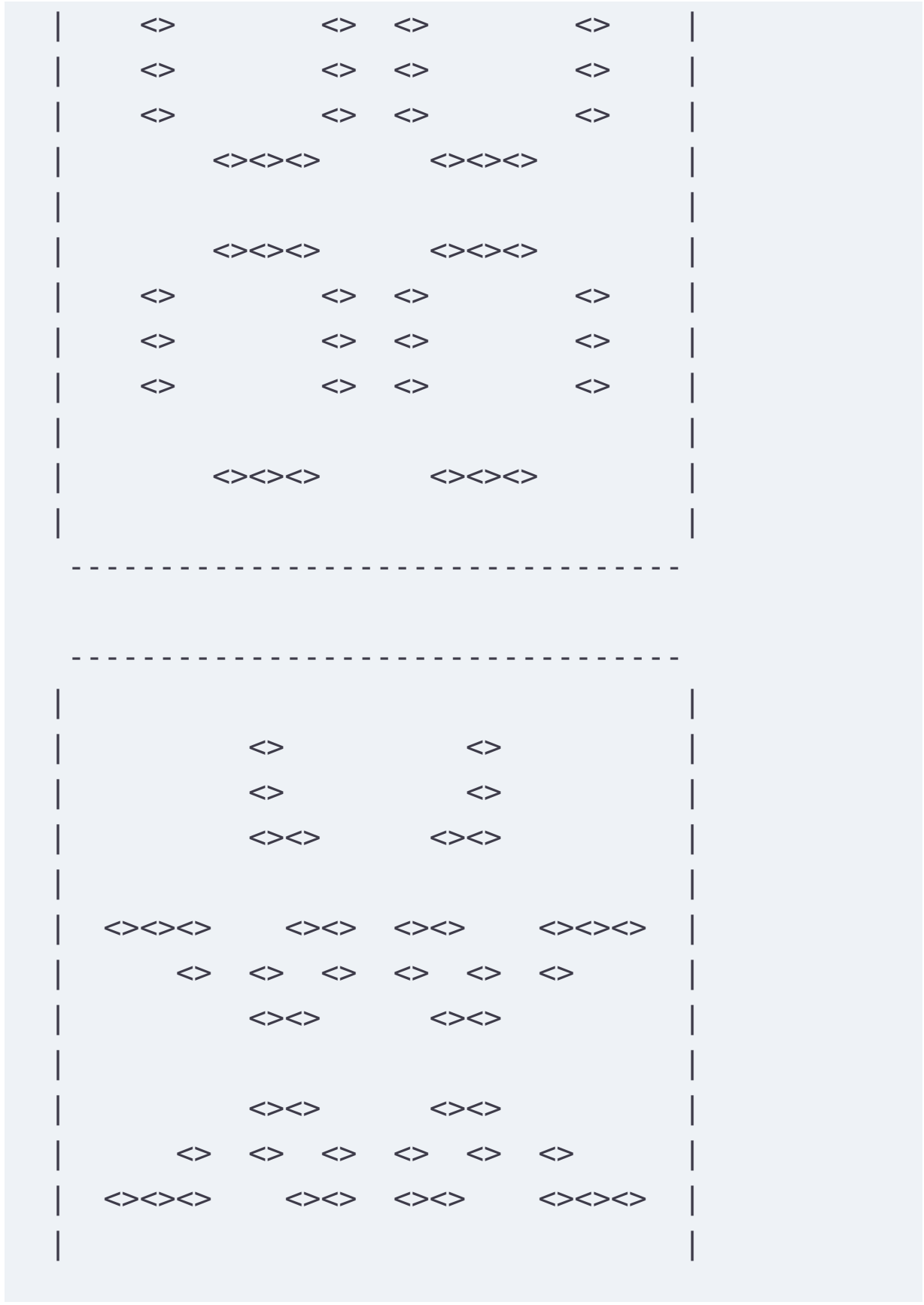
❷ `life` contains a call to `iterate` to create an infinite sequence of Game of Life states given an initial one. "height" and "width" are necessary to `next-gen` to calculate neighbors `next-gen` function is visible in the example section of the for chapter.

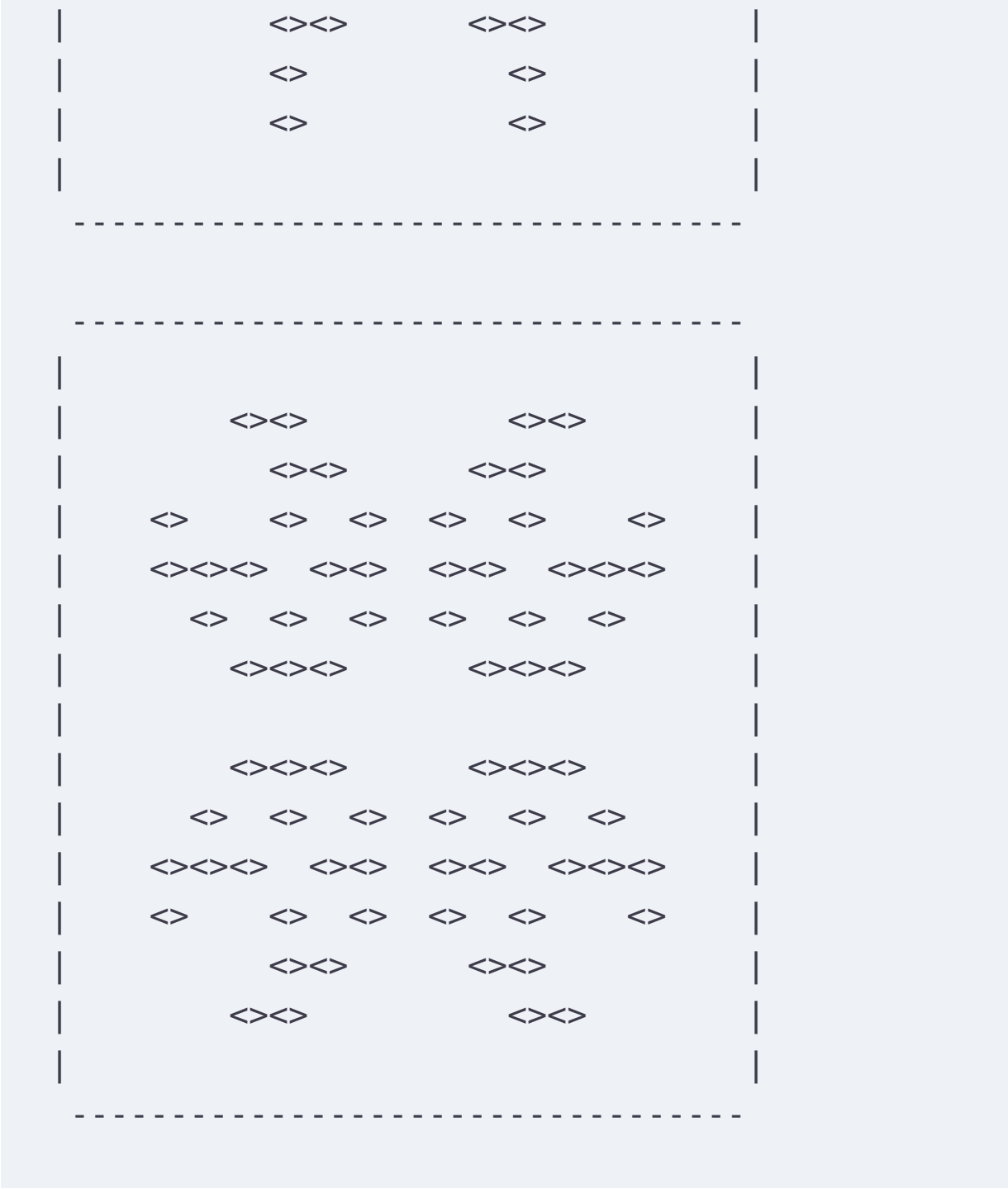
❸A <www.ericweisstein.com/encyclopedias/life/Pulsar.html,Pulsar is a period 3 oscillator that creates a nice shape. We can give any of the 3 states as the initial population. The initialization is a Set of live cells.

❹To print a Pulsar we need a large enough grid of 17x17 cells. We use doseq to take 3 states (after which the printout restart from the initial state).

The output produced by `pulsar` is visible below:

```
(pulsar)
 ---------------------------------
|                                 |
|        <><><>       <><><>     |
|                                 |
```





See also

- repeat produces a sequence starting from a single repeating value or expression, instead of a function invoked on the previous arguments.
- repeatedly produces a sequence by invoking the same functions without arguments (primarily for side effects).
- dotimes produces the side effects (if any) of evaluating its body the given amount of times.

Performance considerations and implementation details

=> `O(n)` **linear (worst case, fully consumed)**

=> `O(n)` **memory (worst case, retaining head)**

Like other lazy sequence generators, `iterate` produces the lazy sequence in constant time. Consuming the lazy sequence produced by `iterate` is linear in the number of elements requested. Memory follows the same linear behavior assuming the worst case in which the head can't be garbage collected and the lazy sequence is consumed.

As quickly reminded at the beginning of the generators section, `iterate` is implemented in Java. One of the advantages of the Java implementation is better control of the reducing context. reduce delegates the specific data structures the details on how

to reduce itself. The default reduce is sequential: the reduction is done by repeated next calls until the end of the sequence. But `iterate` implementation provides a custom reduce implementation that can also be used with transducer. The advantage is visible in the following benchmark:

```
(require '[criterium.core :refer [quick-bench]])

(defn iterate* [f x]
  (lazy-seq (cons x (iterate* f (f x)))))

(quick-bench (into [] (take 1e6) (iterate* inc 0)))
;; Execution time mean : 97.414648 ms

(quick-bench (into [] (take 1e6) (iterate inc 0)))
;; Execution time mean : 44.920465 ms
```

❶ `iterates*` is a straightforward implementation using lazy-seq and cons. `iterate*` shows how a pure Clojure implementation would look like.

❷ The first benchmark measures our `iterate*` implementation based on lazy sequences. into uses reduce which in this case is not optimized.

❸ The second benchmark repeats the same operation on the standard optimized `iterate`.

Note that the custom reduce implementation of `iterate` lacks the typical sequential caching. In this respect `iterate` is similar to eduction:

```
(let [itr (iterate* #(do (println "eval" %) (inc
      v1 (into [] (take 2) itr)
      v2 (into [] (comp (drop 2) (take 2)) itr)]
      (into v1 v2))
;; eval 0
;; eval 1
;; eval 2
;; eval 3
;; [0 1 2 3]

(let [itr (iterate #(do (println "eval" %) (inc 
      v1 (into [] (take 2) itr)
      v2 (into [] (comp (drop 2) (take 2)) itr)]
      (into v1 v2))
;; eval 0
;; eval 0
;; eval 1
;; eval 2
;; [0 1 2 3]
```

❶The first example shows `iterate*` (an implementation of `iterate` using cons and lazy-seq) being used twice after creation in two separate into invocations to create a vector and applying a take-drop transducer combination. The function used to `iterate` is also printing each incremented number.

❷In the second example, core `iterate` has been used instead for the same exact operation.

We can see how `iterate*` caches evaluations of items (and also evaluates an additional item ahead, to check for the end of the sequence condition before creating the lazy lazy-seq instance). The standard `iterate` version produces multiple prints of the same item evaluation, showing that there is no caching. This is especially important to understand when `iterate` is used with side-effecting functions, as there is no guarantee about how many times `iterate` will call the function.

9.2.3 repeat and cycle

`function` `since 1.0`

Listing 9.8. → *Values repetition, Cycling, Sequential generation*

```
(repeat
  ([x])
  ([n x]))

(cycle
  [coll])
```

`repeat` and `cycle` have a similar goal, generating a sequence by performing some repetition of the input. `repeat` takes a single value "x" and produces a lazy sequence by repeating it infinite (or "n") times. `cycle` takes instead a collection of values and produces a sequence by repeating the content of the collection in a cycle:

```
(take 5 (repeat (+ 1 1))) ; ❶
;; (2 2 2 2 2)

(take 5 (cycle [1 2 3])) ; ❷
;; (1 2 3 1 2)
```

❶ `repeat` follows normal evaluation rules and evaluates the expression passed as an argument. The number is then repeated indefinitely creating an infinite sequence. We take some amount of items from the sequence to show the result.
❷ `cycle` uses the given collection as the source of repetition, taking the items in the collection in a cycle and producing a new sequence out of them.

`repeat` also accepts a number of elements to limit the length of the sequence:

```
(repeat 5 1) ; ❶
;; (1 1 1 1 1)
```

❶Repeating "1" only 5 times.

Contract

input

- "x" can be any expression or value including `nil`. It is a mandatory argument.
- "n" is optional. When present it can be positive number. If 0 or negative, `repeat` produces an empty clojure.lang.PersistentList. If "n" is a double, it is truncated to the nearest integer.

- "coll" is a sequential collection (a collection that can produce a sequence following seq contract>>) or `nil`.

notable exceptions

- `NullPointerException` when "n" is `nil`.

output

- `repeat` generates a sequence by repeating the single value "x" infinite (or "n" times). The type of the result is a sequence-compatible `clojure.lang.Repeat` object.
- `cycle` generates a sequence by cycling through a collection of values infinite times. The type of the result is a sequence-compatible `clojure.lang.Cycle` object. It produces an empty clojure.lang.PersistentList when the sequence is empty.

Examples

`repeat` and `cycle` are flexible tools for sequential processing. For example, `repeat` can be used as the second input for map to transform a collection of words:

```
(defn bang [sentence] ; ❶
  (map str (.split #"\s+" sentence) (repeat "!")))

(bang "Add exclamation each word")
;; ("Add!" "exclamation!" "each!" "word!")
```

❶The `bang` function splits a string into separate words. `repeat` is used to generate an infinite sequence of exclamation marks which can be adapted to the sequence of words as the input for map.

The second parameter "n" can be used to limit the sequence length, for example to calculate $x^{\overline{y}}$ (the power of "x" to exponent "y") with:

```
(defn pow [x y] (reduce * (repeat y x))) ; ❶
(pow 2 3)
;; 8
```

❶We use `repeat` to create a sequence of multiplication factors that we can reduce with "*" to obtain the final result.

The following example implements a tally mark system [160]:

```
(defn to-tally [n] ; ❶
  (apply str (concat
     (repeat (quot n 5) "卌")
     (repeat (mod n 5) "|"))))

(defn new-tally [] ; ❷
  (let [cnt (atom 0)]
    (fn []
      (to-tally (swap! cnt inc)))))

(def t (new-tally))

(t)     ; ❸
;; "|"
(t)
;; "||"
(t)
;; "|||"
(t)
;; "||||"
(t)
;; "卌"

(repeatedly 5 t) ; ❹
;; ("卌|" "卌||" "卌|||" "卌||||" "卌卌")
```

❶The implementation of the tally system is a concatenation of the constituent characters based on their ordinal number. The special UTF-8 symbol 𝍌 (U+534C) is used to simulate the horizontal strikethrough every 4 vertical lines. `repeat` is used twice, one for the repetition of the strikethrough and another for the reminder after the last strikethrough.

❷A new tally is created by closing over an atom state. This simulates the appearance of a new tally sign each time we call the generated function.

❸Each invocation of the tally function `t` returns the next state of the tally, up to the point where the strikethrough is used.

❹We can optionally use repeatedly to generate many of them in a sequence.

`cycle` can be used to adapt a short collection to fit a larger sequential view. We've seen a few example already in the book that the reader is invited to review:

- `cycle` was used in trampoline to build an infinite sequence of states for the changing colors of a traffic light. There are only three colors, but they've been adapted by `cycle` to simulate infinite traffic light changes over time.

- We've used `cycle` to generate an arbitrary long password using a finite alphabet of symbols in random-sample.
- The Leibniz formula approximation for Pi also uses `cycle` in the "creating your own fold" call-out.

As of idiomatic use of `cycle` the following should be preferred instead of `concat`:

```
(take 10 (apply concat (repeat [1 2 3]))) ; ❶
;; (1 2 3 1 2 3 1 2 3 1)

(take 10 (cycle [1 2 3])) ; ❷
;; (1 2 3 1 2 3 1 2 3 1)
```

❶ `concat` is used with `repeat` over a collection of elements, producing a nested sequence of [1 2 3] vectors.
❷This is exactly the use case for `cycle`.

See also

- iterate is a flexible form of repetition to generate a lazy sequence with a function that decides the next item based on the previous. `repeat`, for instance, can be written in terms of `iterate` as: `(iterate identity x)`.

- repeatedly has a specific focus on side effecting functions to produce the output sequence.
- constantly produces a function not a sequence. The function can be passed any number of arguments and always return the same result.
- dotimes iterates the body and produce `nil`. The body is evaluated for side effects only.

Performance considerations and implementation details

=> `O(n)` **linear (worst case, fully consumed)**

=> `O(n)` **memory (worst case, retaining head)**

Both functions creates a lazy sequence in constant time, as no items are consumed. Consuming items from `repeat` or `cycle` is linear in the number of requested items. Memory is consumed linearly only when the head is retained resulting in the worst case scenario.

Like range or iterate, `repeat` and `cycle` have a fast reduce path. When `repeat` or `cycle` are used directly as the source for reduction (including transducers) they don't use the default sequential `reduce`. Applications reducing over relatively large sequences created with `repeat` or `cycle` can take advantage of transducers especially when transformations are involved:

```
(require '[criterium.core :refer [quick-bench]])

(quick-bench (reduce + (take 1e6 (map * (range)
;; Execution time mean : 193.403844 ms

(quick-bench (transduce (comp (map-indexed *) (t
;; Execution time mean : 80.234017 ms
```

❶The initial map operation with two sequences as input produces the series of alternating sign integer numbers `(0 1 -2 3 -4 5 -6 7 -8 9)` etc. and calculate the sum of the first million numbers.

❷The transduce version of the same operation uses the generation of positive numbers by map-indexed and keeps `cycle` as the only source. We can see an approximate 50% speed gain.

[159] The Leibniz formula is well described in the dedicated Wikipedia page en.wikipedia.org/wiki/Leibniz_formula_for_π

[160] Tally marks are simple numeral systems with a single symbol. The amount of symbol is the total count while their grouping is a visual tool to help counting large numbers. See the Wikipedia entry to know more: en.wikipedia.org/wiki/Tally_marks

9.3 OTHER GENERATORS

Generators create sequences starting from objects that are not necessarily designed to contain data (unlike vectors, hash-maps or lists or other built-in collections). This category contains:

- Ephemeral transport objects (such as a Java `BufferedReader`).
- Iteration abstractions (like iterators or enumerables).
- Arbitrarily nested compound objects (like the result of parsing an XML document or listing the content of folders on a file system).

Some of these mechanism are quite generic (like tree-seq), others are very specific (like building a sequence out of a database `ResultSet`).

9.3.1 lazy-seq

`macro` `since 1.0`

Listing 9.9. → *Lazy evaluation, Caching, Sequence generation*

```
(lazy-seq [& body])
```

`lazy-seq` is a macro that returns a sequence from a generic sequential input:

```
(lazy-seq '(1 2 3)) ; ❶
; (1 2 3)

(type (lazy-seq [1 2 3])) ; ❷
;; clojure.lang.LazySeq

(seq? (lazy-seq [1 2 3])) ; ❸
;; true

(lazy-seq 1 2 [3]) ; ❹
;; (3)
```

❶The REPL automatically prints the evaluation of the last expression. When the value is sequential, like in this case, it prints as a list.

❷We can try another sequential input, such as a vector. We can see that the type is a special `clojure.lang.LazySeq` object.

❸The object returned by `lazy-seq` is a sequence conforming to the `clojure.lang.ISeq` interface.

❹Note that you can pass a variable number of arguments. They are implicitly treated in a do block.

The reader at this point might be wondering what's the purpose of wrapping a sequential object (something that can be turned into a sequence) in a sequence. There are two reasons. The main purpose of `lazy-seq` as a macro is to delay the evaluation of the input:

```
(def output (lazy-seq (println "evaluated") '(1 2
;; #'user/output

(first output) ; ❷
;; evaluated
;; 1
```

❶Note that we added a side effecting `println` as part of the arguments. `lazy-seq` prevents the evaluation of the input: we don't see the message when we declare the output.

❷As soon as we access `output`, for example to fetch the first item, the body evaluates.

The second goal of `lazy-seq` is to cache the result of evaluating the input. When the same `lazy-seq` form evaluates again, the result comes from the internal cache:

```
(defn trace [x] (println "evaluating" x) x) ; ❶

(def output (lazy-seq (list (trace 1) 2 3))) ; ❷

(first output) ; ❸
;; evaluating 1
;; 1

(first output) ; ❹
;; 1
```

❶ `trace` is a simple debugging function that prints the argument before returning it without any changes.
❷ We use list to produce a list of 3 numbers. The first number is wrapped within the `trace` function. On creation of the `lazy-seq` nothing is printed on screen.
❸ Evaluating the first element produces the message as well.
❹ The second time we access the output, we only see the number 1 and no "evaluating" string. `lazy-seq` evaluated the body once and then cached the result of evaluating the expression.

The fundamental properties of `lazy-seq` are not particularly interesting when used in isolation, but it's a fundamental building block to produce lazy sequences in pair with cons. We can chain `lazy-seq` objects together to delay the evaluation of the items in a sequence (the fundamental aspect of laziness). Please see the example section for the details.

Contract

input

- "& body" is a variable length argument. The arguments are implicitly wrapped in a do block. The result of evaluating the `do` block expression needs to be sequential (any collection implementing the `clojure.lang.Seqable` interface).

notable exceptions

`IllegalArgumentException` when "body" does not evaluate to a sequential collection, as per seq semantic. Note that given the implicit do argument, the last expression needs to return a sequential collection:

```
(lazy-seq 1 2 3) ; ❶
;; IllegalArgumentException Don't know how to cr
```

❶ `lazy-seq` accepts any number of arguments that are treated as an implicit `do` block. The result of the evaluation of the `do` block needs to return a sequential collection. In this case, "3" is not sequential.

output

- returns: a sequence representing the sequential view over the input "body". Returns an empty sequence if there are no arguments or if the arguments is an empty collection. `lazy-seq` never returns `nil`.

Examples

A sequential operation, like transforming each item in a sequence, can become lazy by interleaving a delaying `lazy-seq` to each transformation step. The pattern consisting of using `lazy-seq` and cons in a recursive function, is the canonical way to generate a lazy sequence in Clojure (and it's used pervasively throughout the standard library). Please compare the following functions to transform the input "coll" in a list (a custom version of map):

```
(defn eager-map [f coll] ; ❶
  (when-first [x coll]
    (println "iteration")
    (cons (f x)
          (eager-map f (rest coll)))))

(def eager-out (eager-map str '(0 1 2))) ; ❷
;; iteration
;; iteration
;; iteration
```

```
(defn lazy-map [f coll] ; ❸
  (lazy-seq
    (when-first [x coll]
      (println "iteration")
      (cons (f x)
            (lazy-map f (rest coll))))))

(def lazy-out (lazy-map str '(0 1 2))) ; ❹
(first lazy-out) ; ❺
;; "iteration"
;; "0"
```

❶ `eager-map` is a recursive function creating a list of cons "pairs". Each iteration produces a cons object by pairing the current transformed item and the computation of the rest of the list.

❷ We can see that even without using the output of `eager-map` we already fully evaluated the recursion including performing all transformations. The newly transformed sequence of cons is already existing in memory right after evaluation.

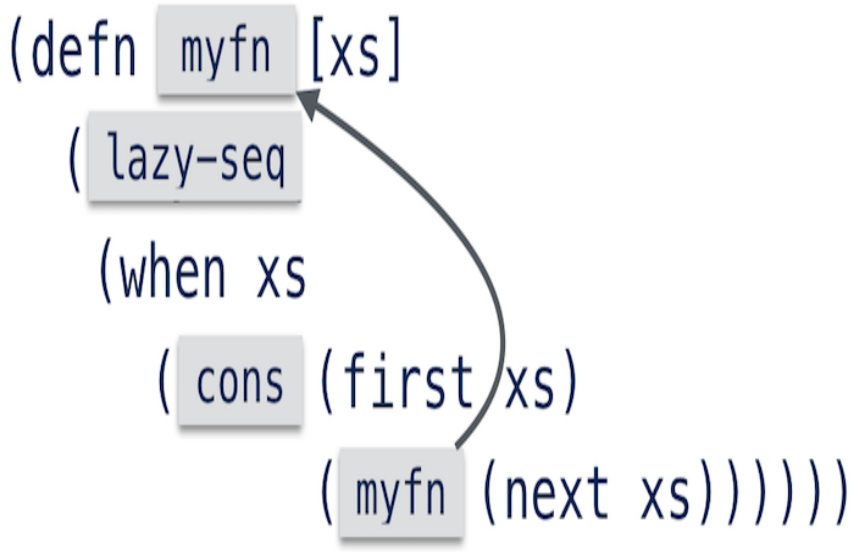
❸ `lazy-map` wraps the outer form in a `lazy-seq` call. This is the only change compared to `eager-map`. cons produces a sequence type which is accepted by `lazy-seq` (along with persistent list, cons is the other native sequential Clojure type).

❹ The function call is evaluated but it does not produce any output.

❺ Additionally, we can see that asking a single element produces a single recursion loop. This is because the next `lazy-seq` wrapper does not force another recursion unless explicitly requested.

The following diagram shows the general idea of the pattern: the fundamental aspect is the presence of the wrapping `lazy-seq` call before the body of the function `myfn`, which can then be called recursively at any point.

Figure 9.3. The general recursive pattern to build lazy sequences.



We can apply the same pattern to build lazy sequential abstractions on top of a disparate range of data producers (the standard library does this extensively). Data producers can be concrete collections, services or abstract generators.

The following naive implementation of the Sieve of Eratosthenes, uses a natural number generator to return an infinite sequence of prime numbers [161]:

```
(defn sieve [n]
  (letfn [(divisor-of? [m] #(zero? (rem % m))) ;
```

```
            (step [[x & xs]]          ; ❷
              (lazy-seq (cons x       ; ❸
                (step (remove (divisor-of? x) xs))
    (take n (step (nnext (range))))))

(sieve 10)
;; (2 3 5 7 11 13 17 19 23 29)
```

❶ `divisor-of?` predicate takes the dividend number "m" and returns a function that can be used to find divisors of "m". `((divisor-of? 4) 2)` is for example `true`.

❷The `step` function contains the recursive action. Each `step` corresponds to an iteration during recursion. The `step` wraps `lazy-seq` around its body and contains the logic to calculate the next number. The initial destructuring into "x" and "xs" helps removing the occurrence of `first` and `rest` later on.

❸The basic recurring step consists of taking the current number "x" at the beginning of the list and cons it into the rest of the computation. "x" is already a prime number because the rest of the calculation removes all its divisors from the list of all natural numbers.

❹The beginning of the recursion prepares an infinite list of positive integers starting from 2 (using nnext is equivalent at calling next twice).

One interesting effect of using `lazy-seq` is transforming a stack-consuming algorithm into heap-consuming. Let's macroexpand the `lazy-map` implementation from the previous example:

```
(macroexpand ; ❶
  '(lazy-seq
     (when coll
       (cons
         (f (first coll))
         (lazy-map f (next coll))))))

;; (new clojure.lang.LazySeq
;;   (fn* []
;;     (when coll
;;       (cons
;;         (f (first coll))
;;         (lazy-map f (next coll))))))
```

❶We call macroexpand on the body definition of the `lazy-map` function as seen in the examples before in the chapter.

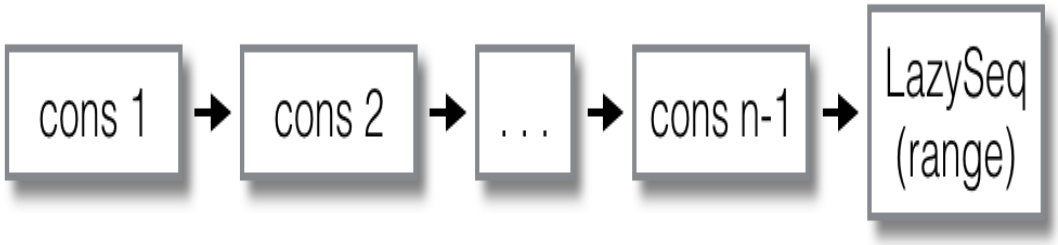
The expansion shows that `lazy-seq` is a Java object constructor accepting a function object as parameter. The function object has no parameters and when invoked it evaluates the content of the body. The recursive call to `lazy-`

`map` that appears inside the body of the function does not live on the stack, because it returns the next `clojure.lang.LazyMap` object immediately. The object contains a promise for computation at some later time and is parked on the heap (the default residence for object allocation).

When a consumer pulls an item from the lazy sequence, the outer `LazySeq` object evaluates and caches its value. At the same time, the recursion produces the next promise for computation. As the consumer asks additional items, the recursive action produces more promises for computation. If nothing is holding the head of the sequence, the first outer `LazySeq` can be garbage collected and the entire sequence never resides in memory at once.

Simple recursion with `lazy-seq` produces linear, heap-consuming lazy sequences like the one shown in this diagram:

Figure 9.4. Recursive use of `lazy-seq` produces a concatenation of cons objects with a final `LazySeq` holding the unrealized rest of the infinite sequence.



You need to be careful when nesting recursive lazy sequence generators. This happens quite often as part of designing typical algorithms for sequential processing. The problem is that the nesting could end up consuming the stack even when you don't expect it. The following `lazy-bomb` function illustrates the problem:

```
(defn lazy-bomb [[x & xs]]
  (letfn [(step [[y & ys]]
            (lazy-seq
              (when y
                (cons y (step ys)))))] ; ❶
    (lazy-seq
      (when x
```

```
          (cons x (lazy-bomb (step xs)))))))) ; ❷

  (last (lazy-bomb (range 10000))) ; ❸
  ;; StackOverflowError
```

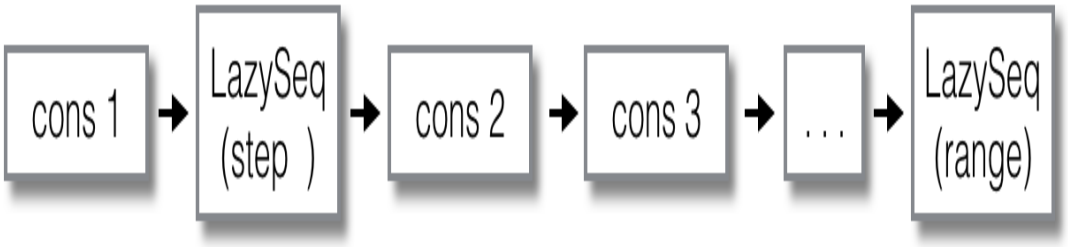
❶ `lazy-bomb` contains a `step` function which contains another recursion using the typical `lazy-seq` pattern. The `step` doesn't do anything apart from destructuring and rebuilding the input with an intermediate recursive call.

❷ `lazy-bomb` main body is a similar recursion which takes each item from the input and cons it into a call to the inner `step` function.

❸ `lazy-bomb` generates stack overflow for modestly large inputs of a few thousands items.

The `step` function in `lazy-bomb` follows the standard `lazy-seq` pattern but produces a stack overflow error when we would expect an heap consuming recursion. The problem is in the interleaving of the outer recursion of `lazy-bomb` and the inner recursion of `step`. The structure of the algorithm is such that `lazy-bomb` always return an unrealized sequence as the target for the first `cons`, like illustrated in the following diagram.

Figure 9.5. Nested use of `lazy-seq` can produce a sequence with intermediate unrealized steps.



In order to satisfy requests for more items, the unrealized lazy sequence appearing at the head needs to be traversed to get to the range generator. The farther away we request the item, the longer is the traversal to get to the generator. The traversal is on

the stack because it's part of evaluating the "misplaced" `lazy-seq`.

Unfortunately, nested `lazy-seq` recursions can be hidden away behind other innocents functions making them difficult to see. If you look carefully at the `sieve` function presented before, it contains such accidental nesting disguised as a remove call. We can rework the `sieve` function to inline the `remove` call and make that explicit:

```
(defn sieve [n]
  (letfn [(remove-step [x [y & ys]] ; ❶
            (lazy-seq
              (when y
                (if (zero? (rem y x)) ; ❷
                  (remove-step x ys)
                  (cons y (remove-step x ys)))))))
          (sieve-step [[x & xs]] ; ❸
            (lazy-seq
              (cons x (sieve-step (remove-step x
    (take n (sieve-step (nnext (range))))))

(sieve 10) ; ❹
;; (2 3 5 7 11 13 17 19 23 29)
```

```
  (sieve 10000) ; ❺
  ;; StackOverflowError
```

❶ The previous remove call has been replaced with a `remove-step` local function, which is just the implementation of `remove` with a fixed predicate.

❷ `divisor-of?` appears here as part of the if condition, which is the general rule for removing items: if the next number is divisor of the current prime, ignore and iterate the next.

❸ Also the recurring call to `sieve` was extracted into a new `sieve-step`. The relationship between the outer `sieve-step` and inner `remove-step` is now explicit.

❹ We can see that this is generating prime numbers like before.

❺ For large numbers, this `sieve` implementation goes into stack overflow.

The `sieve` function described so far suffers from the `lazy-seq` nesting problem, producing a stack overflow for relatively small numbers. There are several alternatives to consider, including reformulating the algorithm to be tail recursive. In doing so, we have an opportunity to look at the accumulated list of prime numbers so far and use that knowledge to reduce the searching space for the next prime [162]:

- We could search for odd numbers only, as no even number could ever be a prime.
- We can concentrate on just the last found prime number onward.
- We can check for prime factors up to the square root of the prime candidate.

The following `sieve` generates a vector using the suggestions above:

```
(defn sieve [n] ; ❶
  (letfn [(odds-from [n] ; ❷
            (iterate #(+ 2 %) (if (odd? n) (+ 2 
          (divisor? [p] ; ❸
            #(zero? (rem p %)))
          (cross-upto [n primes] ; ❹
            (take-while #(<= (* % %) n) primes))
    (loop [cnt (dec n) primes [2]] ; ❺
      (if (pos? cnt)
        (recur (dec cnt)
          (conj primes
            (first (drop-while ; ❻
                #(some (divisor? %) (cross-upto 
                (odds-from (peek primes))))))
        primes))))
```

```
  (peek (sieve 10000)) ; ❼
  ;; 104729
```

❶ `sieve` starts with a few function definitions for internal use before entering the main loop-recur recursion.

❷ `odds-from` starts a range of odd numbers starting from the first available after "n".

❸ `divisor?` returns a predicate to check if a number is divisor of "p".

❹ `cross-upto` takes a list of prime numbers and returns them up to first that when squared goes beyond the given candidate prime "n". The "cross" evokes the similar operation in the Sieve of Eratosthenes.

❺ The main loop starts by setting the counter and initial vector of primes (which always starts from 2).

❻ The next prime is the first after we dropped candidates using a predicate function to cross the relevant divisors.

❼ The result of calling `sieve` to see the 10000th prime number is not resulting in a stack overflow.

Although the algorithm presented above is not consuming the stack, is far from being efficient. Asking for a million primes takes a considerable amount of time and the results are not lazy

consuming linear memory. Improving on such algorithm is possible but beyond the goals of this book.

See also

- seq produces a lazy sequence out of collection supporting the sequential interface.
- concat creates a lazy sequence out of the concatenation of 2 or more collections.

Performance considerations and implementation details

=> `O(1)` **macro execution**

=> `O(1)` **memory**

The `lazy-seq` macro builds a Java object and related fn closure in constant time. `lazy-seq` caching is responsible for the linear `O(n)` memory footprint of most of the standard library sequence processing functions. At the same time caching is at the base of lazy sequences power and flexibility.

Internally `lazy-seq` takes great deal of attention to clear temporary variables that are necessary for evaluation right after caching results (assignment of `null` to variables after use is called "locals clearing" and happens both inside the `LazySeq`

object and the generated fn, which is an interesting case of `^{:once true}` metadata).

`lazy-seq` locals clearing feature is subject to sporadic `NullPointerException` when an error occurs during evaluation (a known bug, see clojure.atlassian.net/browse/CLJ-2069):

```
(defn squares [x] ; ❶
  (cons (* x x) (lazy-seq (squares (* x x)))))

(def sq2 (squares 2))

(take 5 sq2) ; ❷
;; (4 16 256 65536 4294967296)

(take 6 sq2) ; ❸
;; ArithmeticException integer overflow

(take 6 sq2) ; ❹
;; NullPointerException
```

❶ `squares` generates a sequence of squares of the previous numbers.
❷ The sequence starting at 2 already grows pretty fast, quickly reaching `Long/MAX_VALUE` after the 5th item.
❸ Multiplication fails to produce larger numbers than `long` capacity, which is expected.
❹ But following calls produce an unexpected `NullPointerException`.

The example above shows that locals clearing removed references to local bindings before the `lazy-seq` step was able to cache the result. Additional requests for the same item results in the `LazySeq` object being unable to cache the result instead of showing the previous error.

9.3.2 tree-seq

`function` `since 1.0`

Listing 9.10. → *Depth first walk, Sequence generation*

```
  (tree-seq [branch? children root])
```

`tree-seq` is a general mechanism to traverse an arbitrarily nested data structure and produce a lazy sequence of the

visited nodes:

```
(defn pretty-print [x] ; ❶
  (println (with-out-str (clojure.pprint/write x

(pretty-print ; ❷
  (tree-seq vector? identity [[1 2 [3 [[4 5] [] (

;; ([[1 2 [3 [[4 5] [] 6]]]]
;;  [1 2 [3 [[4 5] [] 6]]]
;;  1
;;  2
;;  [3 [[4 5] [] 6]]
;;  3
;;  [[4 5] [] 6]
;;  [4 5]
;;  4
;;  5
;;  []
;;  6)
```

❶ `pretty-print` is a helper function to format the result of `tree-seq` to read on multiple lines.

❷ After instructing `tree-seq` on how to recognize a branch, it returns the lazy sequence of visited nodes in depth-first traversal order.

`tree-seq` requires the following information to work properly:

1. How to distinguish a branch node: when a branch is found, `tree-seq` iterates its content potentially following down into other branches.
2. How to iterate the content of a branch in case it's not sequential.
3. How to pre-process nodes before moving further.

`tree-seq` is the lazy equivalent of clojure.walk. By producing a lazy depth-first traversal, it can process large data structures that don't fit into memory (paying attention at not holding the head of the sequence).

Contract

input

- "branch?" is a predicate function returning logical `true` or `false`. It's invoked on each node to understand if it's a branch or not.
- "children" is a function of one argument. It's invoked on a branch to obtain its sequential view.
- "root" is root object from which `lazy-seq` starts the traversal. It can be `nil` or an empty collection.

notable exceptions

- `NullPointerException` when either "branch?" or "children" are `nil`.

output

- `tree-seq` returns the lazy sequence of nodes visited during a depth-first traversal starting at "root".

Examples

`tree-seq` is useful to traverse deeply nested data structures to process interesting nodes. In the following example, we can see how to collect all positive values from a nested vector:

```
(defn collect [pred? branch?]     ; ❶
  (fn [children]
    (filter
      (fn [node]
        (or (branch? node) (pred? node)))
    children)))

(defn collect-if [pred? root]     ; ❷
  (let [branch? vector?
        children (collect pred? branch?)]
  (->> root
```

```
      (tree-seq branch? children)
      (remove branch?))))           ; ❸

  (collect-if pos? [[1] [-2 4 [-3 [4] 5 8] -6 7]])
  ;; (1 4 4 5 8 7)
```

❶ `collect` is a function of 2 predicates, "pred?" and "branch?". "pred?" is used to process nodes that are not branches (in our example, everything that is not a vector). "branch?" is used to understand which node is a branch. `collect` returns a function of a collection of children nodes. This function decides which nodes should belong to the final result.

❷ `collect-if` prepares the call to `tree-seq`. It defines the meaning of what is a "branch?" and how to process "children".

❸ Note that we need to remove branches from the results if we only care about terminal nodes.

❹ We can see that the traversal returns positive nodes in depth-first order.

Here's another interesting example. The following `tree-seq` traversal creates a sequence of files starting from the root "/" of the local file system. It's worth noting that thanks to laziness this example runs quickly, as there is no full disk scan to perform (unless we consume the entire sequence):

```
(import java.io.File)

(take 5
  (tree-seq
    (memfn ^File isDirectory) ; ❶
    (comp seq (memfn ^File listFiles)) ; ❷
    (File. "/"))) ; ❸

;; (#object[java.io.File 0x527f7b54 "/"] ; ❹
;;  #object[java.io.File 0x27049765 "/home"]
;;  #object[java.io.File 0x6e76794f "/usr"]
;;  #object[java.io.File 0x5f1781e5 "/usr/bin"]
;;  #object[java.io.File 0x769be488 "/usr/bin/uu
```

❶ `tree-seq` invokes the `File/isDirectory` method for each file. The presence of a directory induces `tree-seq` to descend its content.

❷ `File/listFiles` is used by `tree-seq` for each `File` object representing a directory. When the item is a directory, `listFiles` produces an array of file objects (`nil` otherwise). `seq` transforms the array into sequential content.

❸ The root object is a file object representing the beginning of the iteration.

❹ Note that the output presented here might be different on other system.

In the following example, we want to process a nested document which contains a mix of vectors and hash-maps. Such data structure is typically the result of parsing JSON or similar exchange formats:

```
(def document ; ❶
  {:tag :balance
   :meta {:class "bold"}
   :node
   [{:tag :accountId
     :meta nil
     :node [3764882]}
    {:tag :lastAccess
     :meta nil
     :node ["2011/01/01"]}
    {:tag :currentBalance
     :meta {:class "red"}
     :node [{:tag :checking
             :meta nil
             :node [90.11]}]}]})

(def branch? (every-pred map? :node)) ; ❷

(def document-seq ; ❸
  (tree-seq
    branch?
    :node
```

```
    document))

(remove branch? document-seq) ; ❹
;; (3764882 "2011/01/01" 90.11)

(keep :meta document-seq) ; ❺
;; ({:class "bold"} {:class "red"})
```

❶The document implements branching through maps and vectors. If a node is a map type and contains a `:node` key, then the children are available as the value at that key. Terminal nodes are either strings or numbers. The document seems to follow this convention top to bottom.

❷The `branch?` predicate asserts that a node is a branch if it's a Clojure map and it contains the `:node` key.

❸`document-seq` stores the lazy evaluation of the document sequence in a var. The "children" is simply the `:node` keyword.

❹As seen before, we remove branch nodes from the final result to concentrate on simple values.

❺Other kind of filtering are also possible, for example to show all the `:meta` values.

An eager `tree-seq`

The `tree-seq` implementation in the standard library consists of a lazy recursive walk to generate only as much output as requested. For those scenarios where the output is fully consumed, we can achieve better performance by giving away laziness as follows:

```
(defn eager-tree-seq [branch? children root]
  (letfn [(step [res root] ; ❶
            (let [res (conj! res root)]
              (if (branch? root)
                (reduce step res (children root))
                res)))]
    (persistent! (step (transient []) root)))) ; ❷
```

❶The `step` function uses recursion by reducing the children of the current node. When we reach the bottom of a branch, it returns the results so far without any further reduction.

❷We can introduce a mutable transient to collect the results more efficiently.

The new `eager-tree-seq` function gives away laziness to gain some speed. Please check the performance section to see a full comparison between the lazy and the eager version of `tree-seq`.

See also

- clojure.walk/walk performs a depth-first traversal of arbitrarily nested data structures. The "branch?" predicate for `clojure.walk/walk` is implicitly `true` for all the most common collection types. `clojure.walk/walk` maintains the original nesting of the input, instead of creating a flattened sequence of nodes.
- A zipper is another option to traverse a deeply nested data structure. `zippers` are the most flexible option, as they separate the traversal logic (which is fixed to depth-first for both `tree-seq` and `clojure.walk/walk`) from the traversal state.

Performance considerations and implementation details

=> `O(n)` **linear in number of nodes (worst case)**

=> `O(n)` **memory (worst case)**

`tree-seq` is linear in the number of nodes. `tree-seq` is lazy, so it consumes memory linearly with the number of nodes but only the output is full consumed.

Let's now compare the standard `tree-seq` with the eager solution presented before. In the next example we parse a large

XML document (around 450 KB), but the benchmark only applies to its traversal:

```
(require '[clojure.xml :refer [parse]])
(require '[criterium.core :refer [quick-bench]])

(def document
  (parse "https://nvd.nist.gov/feeds/xml/cve/mis

(let [branch? (complement string?) ; ❶
      children (comp seq :content)]
  (quick-bench (dorun (tree-seq branch? children

;; Execution time mean : 2304.531 µs

(let [branch? (complement string?) ; ❷
      children (comp seq :content)]
  (quick-bench (doall (eager-tree-seq branch? ch

;; Execution time mean : 437.484386 µs
```

❶The first benchmark uses standard `tree-seq`. We need dorun to fully realized the sequence.

❷This is the same benchmark using the `eager-tree-seq` seen before. This function produces a vector instead of a sequence and does not require `dorun`.

The eager version is roughly 5 times faster. This doesn't mean that `lazy-seq` is necessarily slow. There are many factors influencing the choice between a slower but lazy function and a faster, but memory consuming eager version. If you are planning to access just the initial part of the output and potentially move toward the end, then `lazy-seq` is still the best choice. If your application requires maximum speed and the input tree is reasonably sized, prefer an eager version. An eager application of the file system scan seen previously, could easily consume the entire memory for instance.

9.3.3 file-seq

`function` `since 1.0`

Listing 9.11. → *File search, Lazy sequence, Tree Traversal*

```
  (file-seq [dir])
```

`file-seq` takes a `java.io.File` object and returns a lazy sequence of the `java.io.File` objects for all contained files and sub-folders:

```
  (require '[clojure.java.io :as io])

```

```
(count (file-seq (io/file "/usr/share/man"))) ; ❶
;; 16727

(->> (io/file "/etc") ; ❷
     (file-seq)
     (map (memfn getPath))
     (take 6))

;; ("/etc"
;;  "/etc/afpovertcp.cfg"
;;  "/etc/aliases"
;;  "/etc/aliases.db"
;;  "/etc/apache2"
;;  "/etc/apache2/extra")
```

❶"/usr/share/man" is a typical Unix location for command manuals. On this system there are 16727 files and folders, as `file-seq` returns both.

❷"/etc" is another standard folder on Unix systems. `file-seq` returns a sequence of `java.io.File` objects from which we can extract the full path as a string with `getPath`. memfn is used here to create a Java-interop anonymous function.

`file-seq` performs a depth-first file traversal: if the next file is a folder, `file-seq` descend into the folder immediately before traversing the other files at the same level. The example

above shows that `file-seq` descends into "apache2" as soon as it is found ("apache2/extract" appears right after).

Contract

input

- "dir" is a mandatory argument of type `java.io.File`.

notable exceptions

- `ClassCastException` if "dir" is not a `java.io.File`.
- `NullPointerException` if "dir" is `nil`.

output

- returns: a depth-first traversal of all files and folders under "dir" as a lazy sequence.

Examples

Let's start by illustrating the behavior of `file-seq` in the presence of a few special `java.io.File` objects. Some of them produce a valid (abstract) path, which results in apparently empty folders:

```
(def work-dir (file-seq (java.io.File. ".")))
(def abstract-path (file-seq (java.io.File. "")))
(def non-existent (file-seq (java.io.File. "NONE

(.getAbsolutePath (first work-dir)) ; ❶
;; "/Users/reborg/book/."
(count work-dir)
;; 5110

(.getAbsolutePath (first abstract-path))  ; ❷
;; "/Users/reborg/book"
(count abstract-path)
;; 1

(.getAbsolutePath (first non-existent))  ; ❸
;; "/Users/reborg/book/NONE"
(count abstract-path)
;; 1
```

❶The "." is the standard representation for the current folder, which is the folder the JVM process was started from.

❷The empty string is accepted as a valid path, but it is referred as "abstract path" as it is not a physical path. It brings potential inconsistencies and should be avoided. We can see that it prints the current folder, but that's the current folder name plus the empty abstract path folder, which is still a non existent folder.

❸A malformed path (a random string "NONE") shows a non existent folder appended to the working directory. It has the same effect as the empty abstract path.

The following example shows a simplified "grep" utility. "grep" is a common Unix command to verify the presence of a string inside a file. Our "grep" offers the possibility to search specific file extensions starting from the working folder:

```
(require '[clojure.string :as str])

(defn grep-by-type [q ext] ; ❶
  (sequence
    (comp
       (remove (memfn isDirectory)) ; ❷
       (map (memfn getAbsolutePath)) ; ❸
       (filter #(= ext (last (str/split % #"\.")
       (filter #(str/includes? (slurp %) q))) ;
```

```
      (file-seq (java.io.File. "."))))

  (grep-by-type "file-seq" "adoc")
  ;; ("/Users/reborg/book/./manuscript/08-Sequenti
```

❶ `grep-by-type` processes results as a pipeline of transducers transformations.

❷We don't want folders to appear in the result, so we remove them.

❸A `java.io.File` object contains many property we don't need in this example. We take the absolute file path as a string from each file object.

❹This step keeps only file matching the given extension. We assume that the extension comes after the last dot in the file path.

❺Finally, we load the content of the file to search for the given query "q". slurp is an easy option, but other approaches like line-seq might be better to just load enough of the file in memory to find the first match.

See also

- clojure.java.io/file is the recommended way to create file objects in Clojure.

- tree-seq is the generic mechanism used by `file-seq` to perform the folder traversal. Have a look at how `file-seq` is implemented if you need to perform the file traversal in a specific way.

Performance considerations and implementation details

=> `O(n)` **linear worst case**

=> `O(n)` **memory worst case**

`file-seq` implements a lazy depth-first traversal based on `tree-seq`. When fully evaluating the output, `file-seq` needs to perform one `isDirectory` check each file and one `listFiles` operation each directory. At the same time, `file-seq` creates a `java.io.File` object for each item in the sequence. The memory footprint is linear in the number of file objects, as is the number of steps to create the final sequence.

Laziness allows to partially consume `file-seq` results without the need to perform a full scan (a potentially lengthy operation

on a large file system). Please refer to tree-seq for additional performance considerations.

9.3.4 xml-seq

`function` `since 1.0`

Listing 9.12. → *XML traversal, Lazy sequence*

```
(xml-seq [root])
```

XML is a hierarchical markup language footenote:[For additional details on the XML specification, please visit the Wikipedia page at en.wikipedia.org/wiki/XML]. XML nodes can be nested, have attributes and carry content. XML is textual, so the hierarchical properties are encoded through the use of open/closing tags.

Numerous parsers exist for XML and Clojure offers one through the clojure.xml/parse function. `xml-seq` is designed to traverse the output of `clojure.xml/parse`:

```
(require '[clojure.java.io :as io])
(require '[clojure.xml :as xml])

(def balance
```

```
    "<balance>
      <accountId>3764882</accountId>
      <currentBalance>80.12389</currentBalance>
      <contract>
        <contractId>77488</contractId>
        <currentBalance>1921.89</currentBalance>
      </contract>
    </balance>")

  (def xml (-> balance .getBytes io/input-stream xm

  (filter (comp string? first :content) (xml-seq xm

  ;; ({:tag :accountId, :attrs nil, :content ["3764
      {:tag :currentBalance, :attrs nil, :content
      {:tag :contractId, :attrs nil, :content ["774
      {:tag :currentBalance, :attrs nil, :content
```

❶For illustration purposes, we are going to use a small XML fragment encoded directly as a string. `xml/parse` requires that we convert the string into an input-stream before parsing.

❷The output sequence produced by `xml-seq` contains branch nodes, those with a `:content` key which refers to other nodes. Here we are just interested in the terminal nodes of the XML structure, those with a `:content` key that contains strings.

`xml-seq` was added to the standard library in a period where XML was the lingua franca for inter-process communication. Nowadays, other formats are commonly used, but `xml-seq` remains an effective approach to basic XML processing.

Contract

input

- "root" should be consistent with the format produced by clojure.xml/parse. A `nil` root produces a `nil` result.

output

- returns: a `clojure.lang.LazySeq` lazy sequence object which contains branches and values from the XML input in depth-first traversal order.

Examples

An RSS feed is an XML document with a specific format. It is used as one of the standard for feed readers, a class of applications dedicated to aggregate and manage information coming from different online sources. RSS feed are widely used in news websites and in the next example we want to build a small application to search a list of feeds.

Let's parse the raw XML using clojure.xml/parse to produce a hierarchical view of the content in the feed. With `xml-seq` we can then process the sequential view of all the nodes to extract interesting information (which would be buried several level deep in the xml tree otherwise). We could orchestrate processing of nodes to extract the title for each article using a transducer chain. The following retrieves breaking news from a few popular RSS feeds:

```
(require '[clojure.java.io :as io])
(require '[clojure.xml :as xml])

(def feeds
```

```
 [[:guardian "https://git.io/guardian-world-rss-x
  [:wash-post "https://git.io/washpost-rss-xml"]
  [:nytimes "https://git.io/nyt-world-rss-xml"]
  [:wsj "https://git.io/wsj-rss-xml"]
  [:reuters "https://git.io/reuters-rss-xml"]])

(defn search-news [q [feed url]]
  (let [content (comp first :content)]
    [feed
     (sequence
       (comp
         (filter (comp string? content))   ; ❶
         (filter (comp #{:title} :tag))    ; ❷
         (filter #(re-find q (content %))) ; ❸
         (map content))
       (xml-seq (xml/parse url)))]))

(pmap (partial search-news #"(?i)climate") feeds

;; ([:guardian ()]
;;  [:wash-post ()]
;;  [:nytimes
;;   ("Economic Giants Are Restarting.
;;     Here's What It Means for Climate Change."
;;  [:wsj ()]
;;  [:reuters ()])
```

❶The first thing to do is to keep nodes with a content attribute of type string.
❷There are many types of terminal nodes, including those with only metadata, links and so on. The nodes with a title are selected next.
❸It's time to match against the given regular expression and only keep those nodes matching the query.
❹Each feed generates a potentially expensive http call. pmap is an easy choice to achieve better performance by processing the feeds in parallel.

The output from the previous example could be different based on the type of news in the feeds. At the time the example was added to the book it found only one news related to "climate".

See also

- clojure.xml/parse is a mandatory step before calling `xml-seq`. You could parse the xml using other tools and still be able to call `xml-seq` by making sure the output from the parser is compliant with the required format.
- tree-seq implements a generic depth-first traversal for nested data structures. `xml-seq` is based on `tree-seq` as other functions in this chapter.

Performance considerations and implementation details

=> `O(n)` **time worst case**

=> `O(n)` **memory worst case**

`xml-seq` is another example of tree traversal making use of `tree-seq` and as a consequence it has a similar performance profile. Consuming the entire output sequence forces a complete traversal to reach all nodes, producing a linear behavior in the number of nodes in the input XML.

`xml-seq` is lazy, traversing just enough of input to produced the requested output. However, note that `clojure.xml/parse` loads the entire XML content in memory regardless. `xml-seq` is only lazy in terms of the traversal but not for parsing the XML.

9.3.5 re-seq

`function` `since 1.0`

Listing 9.13. → *Regular expressions, Sequence generation*

```
  (re-seq [re s])
```

`re-seq` creates a lazy sequence from the matching instances of a regular expression in a string:

```
(re-seq #"\d+" "This sentence has 2 numbers and
;; ("2" "6")
```

❶ `re-seq` creates a sequence of numbers from this sentence. The numbers are still in their original format as strings.

Contract

input

- "re" is an object of type `java.util.regex.Pattern`. Clojure has a reader literal `#"regex"` to create a `Pattern` instance, or you can use the extended form with Java interop.
- "s" is a `java.lang.CharSequence`. It is normally a string, but it could be another object implementing the same interface. For example:

```
(def sb (doto (StringBuilder.)
              (.append "23")
              (.append "aa 42")))
```

```
(re-seq #"\d+" sb) ; ❶
;; ("23" "42")
```

❶ `re-seq` also accepts a `java.lang.StringBuilder` as input, not just strings.

notable exceptions

- `NullPointerException` if either "re" or "s" are nil.

output

- returns: a sequence of the matching instances of "re" in the string "s". This could be a list of strings or vector of strings, depending on the presence of matching groups. `nil` is returned when "s" is the empty string.

Examples

Strings are inherently sequential in Clojure, producing a sequence of `java.lang.Character` objects forming the string. `re-seq` can be used to produce a sequential list of strings instead of characters:

```
(seq "hello") ; ❶
;; (\h \e \l \l \o)
```

```
(map (memfn toUpperCase) (map str "hello")) ; ❷
;; ("H" "E" "L" "L" "O")

(map (memfn toUpperCase) (re-seq #"\w" "hello"))
;; ("H" "E" "L" "L" "O")
```

❶The result of transforming a string in a sequence with seq is a lazy sequence of `java.lang.Character` objects.

❷Note that we need to turn each `Character` into a `String` before applying `toUpperCase`.

❸ `re-seq` can be used to split the string into single letter which are already converted into `java.lang.Character` without using str.

`re-seq` returns matching instances with individual matching groups if they are present. In the following example we match and destructure a list of repeating names and phone numbers:

```
(def signed-up ; ❶
  "Jack 221 610-5007 (call after 9pm),
   Anna 221 433-4185,
   Charlie 661 471-3948,
   Hugo 661 653-4480 (busy on Sun),
   Jane 661 773-8656,
   Ron 555 515-0158")
```

```
(let [people (re-seq #"(\w+) (\d{3}) \d{3}-\d{4}
  {:names (map second people) :area (map last pe

;; {:names ("Jack" "Anna" "Charlie" "Hugo" "Jane
;;  :area ("221" "221" "661" "661" "661" "555")}
```

❶ This is a sample string of a potentially longer list of people who signed up to teach programming to a group of kids. The text contains the name, phone and an optional note regarding availability.

❷ We can destructure the text using a regular expression, because name and phone numbers appear with the same pattern throughout the file. `re-seq` is given the pattern to search for and the string. Note the round parenthesis in the pattern: we want to be able to isolate specific portion of the matching substring (this is part of standard regex format).

❸ `re-seq` returns a vector containing the matching string and any groups within. We can use the list of vectors to group the information we need.

In the next example we are taking advantage of `re-seq` laziness on some large text. The text contains a fairly long list of 1 million digits of Pi [163]:

```
(def pi-digits
  (slurp "https://tinyurl.com/pi-digits")) ; ❶

(def pi-seq                ; ❷
  (sequence
    (comp
      cat                  ; ❸
      (map int)            ; ❹
      (map #(mod % 48)))   ; ❺
    (re-seq #"\d{10}" pi-digits)))

(take 20 pi-seq)
;; (1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6)
```

❶ `pi-digits` contains a text version of the "EBook of Pi", a book containing Pi digits up to 1 million places. The text contains predominantly digits, but includes also an introduction and a specific space separated format for the digits.

❷ We can produce the sequence of matching Pi digits from the book by matching them by groups of 10, which is how they are formatted in the book.

❸ cat transducer takes each 10 digits string, splits them into single numbers and concatenates them.

❹ The `java.lang.Character` instance coerced into an int produces the index at which it is stored in the ASCII table.

❺ Since "0" is 48 in the ASCII table, the modulo operation has the effect of translating the ASCII entry index into the actual number.

Thanks to `re-seq` laziness, we don't need to match the entire book to retrieve the first 20 digits, saving us some computation steps in case we don't want to consume all digits. However, note that we are still forced into loading the entire book in memory before starting the computation. In the next extended example we are going to see how to fix that.

Lazier than lazy

`re-seq` is lazy in building the sequence of matched patterns out of an in-memory string. If the string is large enough, we might not be able to use `re-seq` simply because the string cannot load into memory.

The following `restream-seq` function is a rework of `re-seq` to use an input-stream instead of a string. We are going to use `restream-seq` to build the list of digits of Pi, as in the example discussed before in this this chapter:

```
(import '[java.io InputStream]
        '[java.net URL]
        '[java.util Scanner]
        '[java.util.regex Pattern])

(defn restream-seq [^Pattern re ^InputStream is] ; ❶
  (let [s (Scanner. is "UTF-8")]
    ((fn step []
       (if-let [token (.findInLine s re)]
         (cons token (lazy-seq (step)))
         (when (.hasNextLine s) (.nextLine s) (step)))))))

(defn pi-seq [is] ; ❷
  (sequence
    (comp
      cat
      (map int)
      (map #(mod % 48)))
    (restream-seq #"\d{10}" is)))

(def pi-digits (URL. "https://tinyurl.com/pi-digits"))

(with-open [is (.openStream pi-digits)] ; ❸
  (doall (take 20 (pi-seq is))))
;; (1 4 1 5 9 2 6 5 3 5 8 9 7 9 3 2 3 8 4 6)
```

❶ `restream-seq` implementation is based on a `java.util.Scanner`, a Java class that can search through a stream for consecutive tokens like regular expressions. The scanner is built on top of the `java.io.InputStream` instance. The lazy sequence is built by recursively moving down lines in the input stream. Each matching toke start a new recursion which builds the final lazy sequence.

❷ The processing pipeline we used before has been adapted to accept the input-stream and use the new `restream-seq` function.

❸ We need to wrap any generation of the Pi digits in a with-open call, so we are sure to properly close the stream once we are done.

The sequence generation downloads just enough HTTP request to satisfy the number of digits to print, preventing the entire book to reside in memory all at once.

See also

- re-find are other functions dedicated to regular expression matching that don't create a sequence. Use `re-seq` if you are interested in a sequential view of the matching pattern in a string.

Performance considerations and implementation details

=> `O(n)` **time worst case**

=> `O(n)` **memory worst case**

`re-seq` required steps depend on the regular expression. The more matches are generated the longer the sequence when

fully consumed. In general, `re-seq` is linear in the number of matched patterns in the input string. Like other functions in this chapter, `re-seq` is lazy and builds just enough of the sequence to satisfy the request upstream.

The presence of a regular expression in any performance sensitive algorithm should be carefully considered. This is not related to `re-seq` in particular or a bad implementation of regular expressions. The problem is that regular expressions, while being powerful and expressive, are designed to solve a broad category of problems and it's relatively easy to produce them inefficiently.

For simple matching of portion of a string, please consider the following alternatives if speed is a concern:

- `(.startsWith "Some string" "Some")` matches for the string "Some" at the beginning of the string "Some string".
- `(.contains "Some string" "e s")` matches for "a e" in any portion of the string "Some string".

- `(.endsWith "Some string" "ing")` matches for "ing" at the end of the string "Some string".

9.3.6 line-seq

`function` `since 1.0`

Listing 9.14. → *Sequential reading, Text processing, Sequence generation*

```
  (line-seq [rdr])
```

`line-seq` creates a sequence of lines from a stream of characters. A new line item in the sequence is created for each line termination marker found in the input:

```
  (require '[clojure.java.io :refer [reader]]); ❶

  (with-open [r (reader "https://tinyurl.com/pi-di
    (count (line-seq r))) ; ❸
  ;; 29301
```

❶ We need to create an instance of `java.io.Readable` to work with `line-seq`. The `clojure.java.io` namespace contains the necessary functions.

❷ We are going to use the with-open macro quite often in this section, because we must always ensure the reader resources are correctly released after we are done processing the input.

❸ `line-seq` is used here to count the number of lines from a large text.

Contract

input

- "rdr" stands for "reader". `line-seq` expects an instance of `java.io.BufferedReader`, a commonly used Java type to process newline separated files.

notable exceptions

- `NullPointerException` if "rdr" is `nil`.
- `ClassCastException` if "rdr" is not a `java.io.BufferedReader` instance.

output

- returns: the sequence of lines found by reading the input.

Examples

`line-seq` is useful to process large textual files without loading them completely in memory. This can happen in two ways:

1. The lazy sequence is never fully realized, stopping processing after a few items.
2. The lazy sequence is fully realized, but processing only accesses the item to retain some partial information (for example the count or other statistics). In doing so the head of the sequence is not retained and each item can be garbage collected as we process the rest of the input.

In the next example we are making access to the top 1 million Alexa entries (Alexa is a company providing web analytics) which is a relatively large (15 MB) archive containing the most popular website by traffic. The archive is compressed, but we can uncompress it while processing the content line by line. We want to know what is the top ranking ".me" domain:

```
(import '[java.net URL])
(import '[java.util.zip ZipInputStream])
```

```
(require '[clojure.java.io :as io])

(def alexa "http://s3.amazonaws.com/alexa-static/

(defn zip-reader [url] ; ❶
 (-> (URL. url)
     .openConnection
     .getInputStream
     ZipInputStream.
     (doto .getNextEntry)
     io/reader))

(defn domain [^String line] ; ❷
  (some-> line (.split "\\.") last))

(defn first-of-domain [ext] ; ❸
  (with-open [r (zip-reader alexa)]
    (some #(when (= ext (domain %)) %) (line-seq

(first-of-domain "me") ; ❹
;; "246,line.me"
```

❶ `zip-reader` creates a `java.io.BufferedReader` starting from an URL. As you can see, Clojure does a great job creating straightforward and readable code out of many objects required to open the reader. One important detail is that while opening a zip archive you have to position the input stream at the beginning of the next entry with `getNextEntry`. In our case it's relatively easy since the archive contains a single entry.

❷ `domain` takes a line and extract the domain of the website it contains.

❸ `first-of-domain` takes an extension as parameter. It then access the sequence in the context of a with-open block. some is used to take enough sequence until the predicate first returns a match, which is the corresponding line containing the domain.

❹ We can see that the highest traffic ".me" website is "https://line.me/", a website for sending free SMS messages. This result could be different at some other time.

The search for the first matching domain executes relatively fast, which signals the URL is not completely downloaded. The next example extends the previous by ranking the list by the most frequent domain:

```
(defn top-10-domains-by-traffic [archive]
 (with-open [r (zip-reader archive)]
  (->> (line-seq r)
       (map domain)
       frequencies        ; ❶
       (sort-by last >)
       (take 10))))

(top-10-domains-by-traffic alexa)
;; ["com" 487682] ["org" 50189] ["ru" 43619]
;; ["net" 42955] ["de" 36887] ["br" 20192]
;; ["uk" 18828] ["ir" 16915] ["pl" 16730] ["it"
```

❶ We use the same `zip-reader` function to operate on the sequence in the context of a with-open call. This ensures any resource is correctly closed at the end of the computation. `line-seq` starts generating the lazy sequence from which we extract the `domain` extension and then pass the entire sequence to frequencies.

The second part of the example uses frequencies, an eager function that scans the entire input to populate its counters. In doing so, it doesn't hold on the head of the sequence. The use of `line-seq` allows for very large files to be processed in memory assuming the collected information (keys and numbers

in the case of `frequencies`) can be stored in memory. Anything else that is not retained in the final results, can be safely garbage collected while processing is still ongoing.

See also

- slurp downloads the content of a URL or local file into memory as a single string. It's usually a straightforward choice for configuration files or other small files.
- split-lines can be used to split a large string into lines producing a vector. It is not lazy, so it should be used when the memory footprints it generates is predictable.

Performance considerations and implementation details

=> `O(n)` **time worst case**

=> `O(n)` **memory worst case**

Like other functions producing a sequence `line-seq` is linear in the number of lines contained in the input. Similarly for memory: the more items consumed from the output the more items collected in memory (sequences are lazy and cached). By avoiding retaining the head of the sequence, we can fully

consume even large sequences without exhausting the entire memory. Laziness plays a role in performance whenever the sequence is not completely consumed, in which case the performance profile is sub-linear in both time and memory.

The implementation of `line-seq` is quite straightforward: a lazy sequence is produced on top of the `BufferedReader` instance for every line available, ending when the buffer produces a `nil` signaling the end of the stream.

9.3.7 resultset-seq

`function` `since 1.0`

Listing 9.15. → *Database, Table rows, Resultset, Sequence generation*

```
  (defn resultset-seq [rs])
```

`resultset-seq` generates a sequence from a `java.sql.ResultSet` object. A `ResultSet` is typically the result of invoking a SQL query on a relational database, although other implementations exist [164]. A `ResultSet` models a table in the database: a sequence of rows with each value representing the respective column attribute.

The creation of a proper `ResultSet` requires the presence of a JDBC compliant database driver. For illustration purposes though, we are going to use a reified `ResultSet` object which simulates the interaction with a database driver:

```
(import '[java.sql ResultSet ResultSetMetaData])

(defn db-driver [attrs] ; ❶
  (reify
    ResultSet
    (getMetaData [_]
      (reify
        ResultSetMetaData
        (getColumnCount [_] (count attrs))
        (getColumnLabel [_ idx] (nth attrs (dec
    (next [_] true)
    (close [_])
    (^Object getObject [_ ^int idx] (rand-int 10

(take 3 (resultset-seq (db-driver ["id" "count"]
;; {:id 886, :count 433}
;; {:id 211, :count 431}
;; {:id 51,  :count 939})
```

❶ `db-driver` creates a dynamic instance of a `ResultSet` implementation with reify. It contains the implementation of the functions used by `resultset-seq` and a small mechanism to produce semi-realistic results. Note that `next` implementation always returns `true`.

❷ After creating an instance of the `ResultSet` stub, we can invoke `resultset-seq` directly on it. We always need to remember to take a finite amount of elements from the infinite sequence of results.

Contract

input

- "rs" must be an instance of `java.sql.ResultSet` and is required argument.

notable exceptions

- `NullPointerException` if "rs" is `nil`.
- `ClassCastException` if "rs" is not a `java.sql.ResultSet` instance.

output

- returns: a sequence of (now deprecated) Clojure struct types. Each `struct` contains the keyword rendering of the name of column in the database, followed by the value of that record at that key.

> Note
> Although structs are now deprecated in favor of defrecords, you can access them the same as normal hash-maps. Their use in `resultset-seq` does not require any other specific knowledge.

Examples

The following example shows a basic JDBC interaction with a database and the way `resultset-seq` wraps the results. The example requires the SQLite driver for Java (available from github.com/xerial/sqlite-jdbc) in the classpath of the running process. We are going to use SQLite configured as an in-memory database:

```
(import '[java.sql DriverManager ResultSet])

(defn create-sample-data [stmt] ; ❶
  (.executeUpdate stmt "drop table if exists per
  (.executeUpdate stmt "create table person (id
```

```
    (.executeUpdate stmt "insert into person value
    (.executeUpdate stmt "insert into person value

  (with-open [conn (DriverManager/getConnection "j
              stmt (.createStatement conn)]
      (create-sample-data stmt)
      (->> (.executeQuery stmt "SELECT * FROM pers
           resultset-seq
           doall)) ; ❹

  ;; ({:id 1, :name "leo"}
  ;;  {:id 2, :name "yui"})
```

❶ `create-sample-data` remove and recreate a table "person" with an id and a name.

❷It is necessary to close connections and statements to release resources after we are done with the results. We can use with-open to close them automatically.

❸After creating a connection object and a statement, we are ready to seed data and execute the query, which return a `ResultSet` instance.

❹We need to completely realize the lazy `ResultSet` with doall before leaving the `with-open` context. If we don't use `doall` we are going to see just the initial results.

Note

When the sequence creation happens inside a try-finally block that closes the connection, make sure that all necessary operations happen inside the block. If portion of the unrealized sequence escapes the block, you might incur in a "connection already closed" exception. One solution to the problem (that removes the benefits of laziness though) is to use doall as shown in the example. Another laziness-friendly option is to pass the processing function into the body of try-finally block.

`resultset-seq` fulfills its full potential when:

- The database driver streams the results from the server (instead of bulk loading them into memory).
- The results are too large to load into memory at once, but we are able to process them incrementally.

The SQLite driver streams results lazily by default (the same is not true for other popular drivers like MySql). If you are dealing with large results and want to process them lazily, you need to be sure the driver supports streaming capabilities.

See also

- Ad-hoc solutions to the problem of iterating a JDBC `ResultSet` can be created with lazy-seq and cons, the building blocks for Clojure lazy sequences (`resultset-seq` included). For any other standard iteration, consider using `resultset-seq`.

Performance considerations and implementation details

=> `O(n)` **time worst case**

=> `O(n)` **memory worst case**

Like other functions in this chapter, `resultset-seq` is linear in the number of rows inside the `ResultSet` object.

Laziness is especially important for `resultset-seq`. Databases are designed to store amounts of data beyond the memory capacity of a single machine. With `resultset-seq` we are able to process large datasets (for example using reduce) assuming the processing function doesn't need the entire

dataset in memory and does not hold on the head of the sequence.

9.3.8 iterator-seq and enumeration-seq

`function` `since 1.0`

Listing 9.16. → *Java iterators, Java enumerators, Sequence generation*

```
(iterator-seq [iter])
(enumeration-seq [e])
```

`iterator-seq` and `enumeration-seq` are functions generating a sequence starting from two standard Java interfaces: `java.util.Enumeration` was the original way to iterate Java collections and `java.util.Iterator` was introduced later as an improvement. Both interfaces are still in use, so Clojure contains a way to create a sequential view from both:

```
(import '[java.util Collections])

(def an-iterator (.iterator [1 2 3])) ; ❶
(def an-enumeration (Collections/enumeration [1
```

```
  (iterator-seq an-iterator) ; ❸
  ;; (1 2 3)
  (enumeration-seq an-enumeration) ; ❹
  ;; (1 2 3)
```

❶The `java.util.Iterator::iterator` method is present in most of the Clojure and Java collections. We can see here how to invoke it on a Clojure PersistentVector.

❷ `Enumeration` objects are more difficult to find, as the interface was gradually abandoned in favor of iterators. There are still plenty of examples for backward-compatibility in the JDK. The method `java.util.Collections::enumeration` can be used to extract an enumeration from any collection supporting an iterator.

❸ `iterator-seq` is used here to generate a sequence out of the iterator object.

❹You can see that the output of `enumeration-seq` is the same as `iterator-seq`.

Contract

input

- "iter" must implement the `java.util.Iterator` interface.
- "e" must implement the `java.util.Enumeration` interface.

notable exceptions

- `NullPointerException` when "iter" or "e" are `nil`.

output

- returns: a sequence generated by iterating the `Iterator` or `Enumeration` object. It returns `nil` in case of iterators or enumerations coming from empty collections.

Examples

The large majority of Java and Clojure collections support the `Iterator` interface. Collections don't implement the interface directly, but provide an `iterator()` method to get a fresh `Iterator` object for every new iteration. While seq knows how to produce a sequence using the `iterator()` method, `iterator-seq` remains for those cases in which the `Iterator` is the only object available (for example as the return type from another function call).

Java 8 introduced `java.util.stream.Stream`, a new interface to support a more functional style of collection processing in Java. The `Stream` supports the iterator interface, so we can use `iterator-seq` to generate a sequence:

```
(->> "Clojure is the best language"
  (.splitAsStream #"\s+") ; ❶
  .iterator ; ❷
  iterator-seq) ; ❸

;; ("Clojure" "is" "the" "best" "language")
```

❶ `splitAsStream` is available on regular expression patterns and can be applied to a string. In this case the regular expression returns any group of 1 or more non-space characters.

❷ The stream is not sequential by default and it does not implement the `Iterable` interface. Calling seq would throw an exception. We can instead call `iterator` explicitly to retrieve the iterator for this stream.

❸ `iterator-seq` knows how to translate an iterator into a sequence.

Warning

`iterator-seq` produces a cached sequence. After processing an item from the iterator source, that item gets cached by the generated sequence, effectively creating an immutable view of the iterator at that point in time. Contrary to Clojure design principles, the `Iterator` interface even includes a `remove` method that allows clients to remove objects from the source object of the iterator! These changes, if any, are not visible from `iterator-seq` after the output sequence has been generated. The reader should remember this behavior when wrapping Java classes that reuse the same iterator instance.

There are still a few objects in the Java standard library that offers an `Enumeration` instead of an `Iteration` view. One case is `java.lang.concurrent.ConcurrentHashMap`, a widely used concurrent collection. The following example shows how we could use `ConcurrentHashMap` to implement a parallel version of distinct. The kind of data structure we would actually need is a `ConcurrentHashSet` which is not in the JDK [165]:

```
(import '[java.util.concurrent ConcurrentHashMap
(require '[clojure.core.reducers :refer [fold]])

(defn parallel-distinct [v]
  (let [m (ConcurrentHashMap.)
        combinef (fn ([] m) ([_ _]))
        reducef (fn [^ConcurrentHashMap m k] (.pu
    (fold combinef reducef v) ; ❶
    (enumeration-seq (.keys m)))) ; ❷

(defn many-repeating-numbers [n]
  (into [] (take n (apply concat (repeat (range

(parallel-distinct (many-repeating-numbers 1e6))
;; (0 1 2 3 4 5 6 7 8 9)
```

❶reducers/fold is used on top of a mutable, but concurrent, data structure. This is why `combinef` always seed the initial reduction with the reference to the map "m". There is also no `combinef` step, as there are no chunks to concatenate.
❷The last step converts the mutable hash-map (an implementation details of how `parallel-distinct` works) back into an immutable sequence using `enumeration-seq` on the keys of the map.
❸After producing a long vector of repeating numbers, we can see that `parallel-distinct` returns them without duplication.

See also

- sequence generates sequences from objects implementing the `java.util.Iterable` interface. When that is not available but there is another way to produce a `java.util.Iterator` object, use `iterator-seq` instead.
- lazy-seq is the main mechanism used to create sequences out of iterable or enumerable objects.

Performance considerations and implementation details

=> `O(n)` **time, worst case**

=> `O(n)` **memory, worst case**

`iteration-seq` and `enumeration-seq` have to perform the iteration down to the last element when fully consumed, producing a linear behavior in the number of items in the original collection.

Both functions are lazy but `iteration-seq` evaluates the input iterator in chunks of 32 items each, while `enumeration-seq` is fully lazy:

```
(import 'java.util.Collections)

(defn dbg-coll [n] ; ❶
  (let [xs (into () (range n 0 -1))]
    (map #(do (print % ", ") %) xs)))

(first (iterator-seq (.iterator (dbg-coll 100))))
;; 1 , 2 , ... ,  32 , 33 , 1

(first (enumeration-seq (Collections/enumeration
;; 1 , 1
```

❶ `dbg-coll` creates a new list of the given size, wrapping it with console logs for each element evaluated.

❷ We can extract an `Iterator` from a Clojure sequence with `.iterator` as they all implement `Iterable`. We can see that `iterator-seq` pulls 32+1 items out of the input sequence: 32 items is the size of the chunk that is evaluated when we call first and 1 item is evaluated from the next chunk to check if there is more input.

❸ `enumeration-seq` is fully lazy without any chunking.

> Note
>
> the reason for `iteration-seq` to provide a chunking behavior is a by-product of recent work related to enabling sequence with transducers, which is in turn driven by performance. In general, the chunking behavior of sequences is often a trade-off between performances and full laziness.

9.3.9 concat and lazy-cat

`function and macro` `since 1.0`

Listing 9.17. → *Collection concatenation, Sequence generation*

```
(concat
  ([])
  ([x])
  ([x y])
  ([x y & zs]))

(lazy-cat [& colls])
```

`concat` and `lazy-cat` generate a sequence by concatenating zero or more sequential collections together. Items from the first collection are added first, followed by the second collection and so on, until all input is consumed:

```
(concat [1 2 3] () {:a 1} "hi" #{5}) ; ❶
;; (1 2 3 [:a 1] \h \i 5)
```

❶ `concat` is used here to concatenate several types of sequential collections. It produces a lazy-seq.

`lazy-cat` is a macro build on top of `concat` that wraps the input collection into lazy-seq before passing it to `concat`. The additional layer of protection, enables lazy concatenation without arguments evaluation. We can see the strong

relationship of `lazy-cat` with `concat` by Macroexpanding a simple form:

```
(macroexpand '(lazy-cat [1 2 3] (range))) ; ❶
;; (concat (lazy-seq [1 2 3]) (lazy-seq (range))
```

❶The macroexpansion of `lazy-cat` shows the use of `concat` with argument wrapping into lazy-seq.

Laziness is the most interesting aspect of both `concat` and `lazy-cat` and we are going to see how it can be used in the example section.

Contract

input

- With no arguments, `concat` returns the empty sequence `()`.
- "x" single argument is accepted by `concat`. It needs to be of sequential type (such that `(instance? java.lang.Seqable x)` is `true`) or `nil`. With a single argument `concat` behaves similarly to lazy-seq producing a lazy sequence from "x".
- "x", "y" and "zs" can be sequential collections or `nil`.

notable exceptions

- `IllegalArgumentException` if any of the input is not sequential as per seq contract.

output

- returns: the lazy sequence generated by concatenating the content of "x", "y" and "zs" (if any) or empty sequence otherwise.

Examples

`concat` is useful to create an uniform view over different sources, each one producing an independent collection or lazy sequence. Here's for example an `identifier` function that produces an unique object identifier which includes all implemented classes and interfaces:

```
(defn identifier [x]
  (let [classname #(.getName ^Class %) ; ❶
        split #(.split % "\\.")
        typex (type x)]
    (apply str
      (interpose "-"
        (concat
          (split (classname typex)) ; ❷
```

```
            (mapcat (comp split classname) (supers

  (identifier #"regex") ; ❹
  ;; "java-util-regex-Pattern-java-io-Serializable
```

❶The let block defines two helper functions: `classname` to get the name of a class as a string and `split` to split a string at each "." dot position.

❷One source of names is the package and class name.

❸Another source of names is derived from processing supers of the same class. We `concat` those together before interposing them with a dash "-" sign.

❹You can try `identifier` with an object like a `java.util.regex.Pattern` or a Clojure vector (which produces a much longer list).

When the list of source collections is only known at runtime, `concat` can be used with apply to concatenate all arguments:

```
(def sold-icecreams ; ❶
  [[:strawberry :banana :vanilla]
   '(:vanilla :chocolate)
   #{:hazelnut :pistachio}
   [:vanilla :hazelnut]
   [:peach :strawberry]])
```

```
(defn next-day-quantities [sold-icecreams]
  (->> (apply concat sold-icecreams) ; ❷
    frequencies
    (sort-by second >)))

(next-day-quantities sold-icecreams) ; ❸
;; ([:vanilla 3] [:strawberry 2] [:hazelnut 2]
;;  [:banana 1] [:chocolate 1] [:pistachio 1] [:
```

❶In this example, we receive a list of today's sold ice creams. The list is simplified in length and structure, reporting only the group of flavors each ice cream contained.

❷We want to be able to see all flavors together, so we can calculate how much ingredients we need to stock for the next day. `apply concat` is an useful idiom to concatenate all lists together.

❸We can see which flavors are most requested and stock accordingly.

Perhaps the most interesting aspect in both `concat` and `lazy-cat` is laziness. Both of them concatenates just enough of the input to satisfy the consumer request:

```
(defn trace [x] (println "evaluating" x) x) ; ❶

(def l1 (map trace (list 1 2 3))) ; ❷
(def l2 (map trace (list 3 4 5)))

(def l1+l2 (concat l1 l2)) ; ❸
(first l1+l2) ; ❹
;; evaluating 1
;; 1
```

❶ `trace` is a simple function that prints its argument before returning it.
❷ `l1` and `l2` are lazy sequences built with map.
❸Nothing is printed when calling `concat`
❹Access to the first element only realize enough of the concatenation to return the first element.

Additionally, `lazy-cat` does not evaluate arguments, which is useful to concatenate non-lazy collections with different creation costs. We could for example build a lazy sequence of items increasingly harder to fetch. If the majority of consumers only consume the first few items, they will be served immediately, while others (the minority) might have to wait longer.

```
(time (first (concat (vec (range 10)) (vec (rang
;; "Elapsed time: 1032.928937 msecs"

(time (first (lazy-cat (vec (range 10)) (vec (ra
;; "Elapsed time: 0.313782 msecs"
```

❶ `concat` is used on a small vector and a much larger one. We only want the first element, but we incur in the cost of creating the large vector anyway.

❷ `lazy-cat` defers evaluation of arguments until the last possible moment. Since we only look at the first element, the large vector is never materialized.

We could leverage laziness to produce "padding" for strings: we want to fill a string with spaces to the right until it reaches a given width. The following example shows how we could use `concat` to draw a rectangle on screen to enclose a given sentence:

```
(require '[clojure.string :as s])

(defn padder [width] ; ❶
  #(take width (concat % (repeat " "))))

(defn line [width] ; ❷
```

```
    (apply str (repeat (+ 2 width) "-")))

  (defn quote-sentence [sentence width]
    (transduce ; ❸
      (comp
        (map (padder width))
        (map #(apply str %))
        (map #(str "|" % "|\n")))
      (completing str #(str % (line width)))
      (str (line width) "\n")
      (s/split sentence #"\s+")))

  (println (quote-sentence "Clojure is my favorite
  ;; --------------
  ;; |Clojure     |
  ;; |is          |
  ;; |my          |
  ;; |favorite    |
  ;; |language    |
  ;; --------------
```

❶ `padder` creates a padding function for a given width. After being created it can be used as a transformation function for map. We use `lazy-cat` on the input string (which is sequential) and the infinite repetition of the space character. Since the infinite sequence appears at the bottom, we can take as much padding as we need without worrying about an upper bound.

❷ `line` is a function that creates line made of dashes ready for display.

❸transduce composes the lines forming the drawing together. The horizontal header is initial argument for the reducing function, while the closing footer is created as the single-argument call to the reducing function. We use completing to compose the finalization and the reducing function str.

❹We can see how to use `quote-sentence` to draw a rectangle around a given sentence.

Using `concat` to build results incrementally

`concat` can be used in a recursive loop to build results when intermediate steps are sequences. But there are potential problems related to the way `concat` calls are nested:

```
(defn get-batch [id] ; ❶
  (repeat id id))

(defn step
  ([n] (step n ()))
```

```
   ([n res]
     (if (pos? n)
       (recur (dec n) (concat res (get-batch n))) ; ❷
       res)))

(step 4) ; ❸
;; (4 4 4 4 3 3 3 2 2 1)

(first (step 10000)) ; ❹
;; StackOverflowError
```

❶ `get-batch` simulates some computation to retrieve a list of items. In a real scenario this could be a database query.

❷ `concat` concatenates the last results in front of the current batch. The operation is recursive until we reach the amount of desired step.

❸Calling step with a small number results in a flat sequence.

❹But large enough steps produce an unexpected `StackOverflowError`.

The `StackOverflowError` is surprising as we are using loop-recur, a construct that doesn't consume the stack. The problem is not recurring from tail position, but the nested `concat` calls that are gradually building up on the stack.

Each iteration, `concat` produces a new result wrapped in a `lazy-seq` object. The chain of `lazy-seq` segments grows up to the point that the traversal needs too many stack frames [166]. A quick solution is to break the `lazy-seq` nesting by inverting the order of the `concat` arguments. Please note that this also changes the order of the results :

```
(defn step
  ([n] (step n ()))
  ([n res]
    (if (pos? n)
      (recur (dec n) (concat (get-batch n) res))
      res)))

(step 4) ; ❷
;; (1 2 2 3 3 3 4 4 4 4)

(last (step 10000))
10000
```

❶ `res` is now at the bottom of the `concat` call, with the lazy sequence produced by repeat first.

❷ We can see that results are now inverted, as the recursion builds up from the last `get-bach` call pushing previous batch to the head of the final sequence.

The new version of the `step` function traverses sequences instead of lazy-seq closures to produce results, generating a heap-consuming algorithm. However, the results have now a different ordering which might not be an option in some situations.

See also

- mapcat is preferred choice when the concatenation is preceded by a transformation. For example:

```
(apply concat (map rest [[1 2 3] [4 5 6]])) ; ❶
;; (2 3 5 6)

(mapcat rest [[1 2 3] [4 5 6]]) ; ❷
;; (2 3 5 6)
```

❶map is followed by `concat` to eliminate inner nesting of sequences.
❷mapcat produces the same result more concisely.

- cat is the transducer version of `concat`.

Performance considerations and implementation details

=> `O(1)` **constant time (best case)**

=> `O(n)` **worst case generated sequence**

`concat` and `lazy-cat` return in constant time, as they stop at the first unevaluated lazy-seq thunk waiting for consumers to pull additional elements out of the sequence. Memory

consumption can go up to the sum of items in the input, with worst case scenario of `O(n)` if the output is fully consumed.

The implementation of `concat` (on which `lazy-cat` is based) is a recursive lazy-seq-cons pattern on the first argument, followed by the second (and possibly others). If the sequence is chunked then the recursion happens by chunk of items instead of a single one.

The "examples" section already illustrated some of the advantages of laziness for concatenation, with `lazy-cat` allowing additional control over the evaluation of non-lazy collections. `lazy-cat` also enables recursive definitions. The following `fibs` definition contains a lazy sequence of Fibonacci numbers (the series of numbers which are the sum of the previous two). The definition is certainly elegant, but it holds on the head of the sequence by design (it is assigned to a var):

```
(def fibs (lazy-cat [0 1] (map +' fibs (rest fibs

(take 10 fibs) ; ❷
;; (0 1 1 2 3 5 8 13 21 34)
```

```
  (last (take 1e6 fibs)) ; ❸
  ;; java.lang.OutOfMemoryError: Java heap space
```

❶Note how `fibs` is just about to be defined, but is used already in the definition itself. This is possible because `lazy-seq` is a macro and does not evaluate its argument. `concat` would not work in this case.

❷We can see that `fibs` works as expected for small numbers.

❸If we try to access the 1 millionth Fibonacci number we incur in a `OutOfMemoryError` (depending also on JVM settings).

Assigning a lazy sequence created with `lazy-cat` directly to a var is potentially dangerous. This is because while we iterate further down the sequence the `fibs` var holds a reference to the head. It is however a typical textbook example to feature laziness and recursion.

[161] The Sieve of Eratosthenes is possibly one of the most instructional algorithms to study the effect of laziness. The naive version presented here is far from being the best algorithm to find prime numbers, but it's relatively simple to understand. This Wikipedia page describes the enhancement to the basic form as well as links to other algorithms: en.wikipedia.org/wiki/Sieve_of_Eratosthenes#Algorithmic_complexity

[162] The problem of efficiently generating prime numbers is vast and fascinating: if you want to know more, this paper about the "Genuine Sieve of Eratosthenes" is worth reading: www.cs.hmc.edu/~oneill/papers/Sieve-JFP.pdf

[163] Supercomputers are now enabling many more digits to be calculated, see en.wikipedia.org/wiki/Pi#Modern_quest_for_more_digits

[164] JDBC, the Java Data-Base Connection framework, is one of the well known Java features. For an overview of the framework and how to work with it, please have a look at the Java Tutorial docs.oracle.com/javase/tutorial/jdbc/basics/index.html

[165] There is no `ConcurrentHashSet` in the Java standard library, but it's possible to obtain a `ConcurrentHashMap` backed `KeySet` that fulfills a similar role. See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html#newKeySet-- for more

[166] Stuart Sierra wrote an article about the same problem at stuartsierra.com/2015/04/26/clojure-donts-concat

9.4 LISTS

The term "list" is subject to overlapping definitions. In Clojure, a list is a concrete data type (`clojure.lang.PersistentList`), in the same way vectors or maps are. The list function is also

the builder for the same data type. Lists are also sequences as they implement the abstraction directly, but they are not technically sequence generators because they are the sequence themselves. Lists are fundamental, starting from the fact that evaluating a Clojure file creates a `list` which eventually feeds the compiler.

cons is, along with list, the other concrete data type extending `clojure.lang.ASeq` directly (other collections have an adapter class for it). cons is also the builder function with the same name similarly to list. The two types, list and cons, are closely related and some functions treat them interchangeably or transparently (thanks to supporting the same sequential interface). cons and list shares the same system of building up chains of linked cells to create the sequential effect. list supports more features than cons, for example a list can be counted in constant time or reduced with an optimized algorithm.

9.4.1 list

`function` `since 1.0`

Listing 9.18. → *Collections, Data type, Sequences*

```
(list [& args])
```

The `list` function creates a new `clojure.lang.PersistentList` data type from the given items:

```
(list 1 2 3 4 5) ; ❶
;; (1 2 3 4 5)
```

❶Creating a `list` with 5 numbers.

The `list` constructor is fundamental in the language: Clojure code that appears as text is first transformed into lists and symbols, then macro-expanded and eventually compiled to bytecode. Functional arguments are also processed as lists. It's not a coincidence that `list` is the first function defined in the standard library at the top of the `core` namespace [167].

Despite their extensive use in the language itself, lists are less flexible compared to lazy sequences, vectors, sets or hash-maps for everyday Clojure programming. There are however a few use cases which we are going to explore in the example section.

Contract

input

- "args" are zero or more arguments including `nil`.

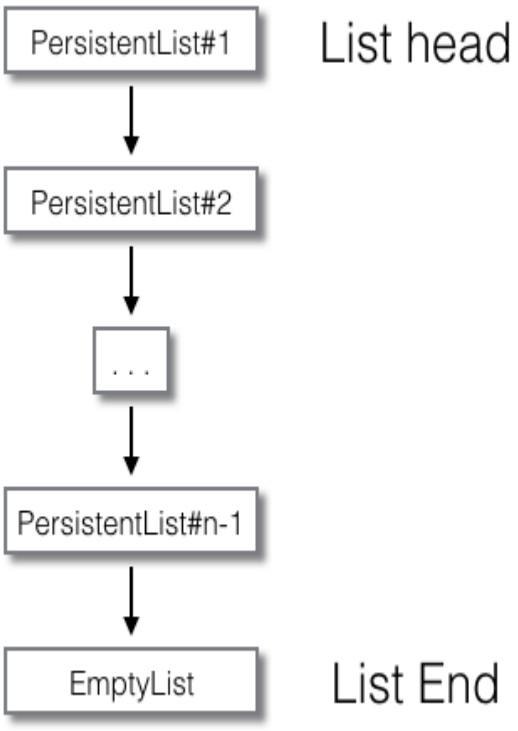
output

- returns: a `clojure.lang.PersistentList` object containing the arguments, or empty list if no arguments.

Examples

Lists are created by "linking" elements to one another. The last item to enter the list is added to the head (the left of the list when printed) and pointed at the previous head forming a chain. The following diagram shows the constituent of a typical `list`:

Figure 9.6. A `list` formed from a chain of `PersistentList` objects. The `EmptyList` is a specialized version of `PersistentList` to handle the tail position.



conj can be used to push elements into a `list`. Each item is pushed at the head of the list, which makes the list appear backward when printed:

```
(-> () (conj 1)) ; ❶
;; (1)

(-> () (conj 1) (conj 2))
;; (2 1)
```

```
(-> () (conj 1) (conj 2) (conj 3)) ; ❷
;; (3 2 1)
```

❶The versatile conj understands how to push elements to a `list`, as well as pushing elements to many other data types.
❷Arguments pushed to the list appears backward when we print the list.

The fact that `list` concatenates new elements to the head, can be used with into (which repeatedly uses conj) to reverse the content of another collection:

```
(defn rev [coll] (into () coll)) ; ❶

(rev (range 10)) ; ❷
;; (9 8 7 6 5 4 3 2 1 0)
```

❶Creating a `list` by pushing items into an empty list. The produced `list` now prints backward, because into uses conj to push new elements to the head of the list.
❷The expression is using `rev` to reverse the content of a range of 10 elements.

A list is a good choice to implement a stack (last-in first-out queues are also called stacks) as it supports the peek-pop

interface. In the following example we are going to use a `list` to build a stack to find the sequence of the nearest smaller values [168].

To understand how the nearest smaller value search works, let's have a look at a small input first. Given a list `(8 11 4 12 5 6)` the sequence of the nearest smaller values is `(8 4 4 5)`:

- "8" has no previous number, so there is nothing to add to results.
- "11" has a previous smaller value, so "8" is added to the results.
- "4" has two previous values, but none is smaller.
- "12" has three smaller previous numbers. "4" is the nearest and is added to the output.
- "5" closest smaller value is "4" so another "4" appears in the output.
- "6" nearest and smaller value is "5".

Note how, once found that "y" is bigger than "x" (for example y="12" and x="5" in the previous list), we can exclude all elements before "y" that are bigger than "y" (for example, we don't need to compare "5" with "11"). We are going to use a

stack to keep track of the visited elements, implicitly giving us an opportunity to skip items for the next iteration:

```
(defn stack [] ()) ; ❶
(defn push [x stack] (conj stack x)) ; ❷

(defn nearest-smaller [xs] ; ❸
  (letfn [(step [xs st]
            (lazy-seq
              (when-first [x xs]
                (loop [st st] ; ❹
                  (if-let [s (peek st)]
                    (if (< s x)
                      (cons s (step (rest xs) (pu
                      (recur (pop st)))
                    (step (rest xs) (push x st))
    (step xs (stack))))

(nearest-smaller [0 8 4 12 2 10 6 14 1 9 5 13 3
;; (0 0 4 0 2 2 6 0 1 1 5 1 3 3 7)
```

❶ `stack` is "syntactic sugar" to create a new `list`. It effectively renames `list` to `stack` helping us dealing with the abstraction properly.

❷ Similarly, the primitive conj operation has been renamed `push` to enforce the proper use of a stack.

❸ `nearest-smaller` general setup follows the typical lazy-seq pattern: the `step` inner function generates the lazy sequence after receiving the input sequence and an empty stack. As soon as when-first returns `nil` the generation ends. What characterizes `nearest-smaller` is the presence of an inner loop-recur constructs to iterate the content of the stack.

❹ The inner (not lazy) loop performs the iteration of the stack content at that point during recursion. We are searching the stack for the first smaller number compared to the current head of the input. If we find one, it goes into the generated output sequence straight away. If we can't find one, we search the next top of the stack and so on.

❺ Every time we reach the bottom of the stack without a smaller item, we `step` over the next iteration without cons. Note how the recursion into generating the output always push "x" to "st" (the current stack view), which means the current head of the list is positioned at the top of the stack.

The outer `step` recursion and the inner `loop` are nested in `nearest-smaller`. The presence of nested loops typically

indicates $O(n^{\overline{m}})$ behavior (where m is the level of nesting and n is the length of the input), but not in this case. Each item is pushed and popped from the stack at most once, effectively limiting the number of operations for the inner loop to a constant factor.

The many `list` meanings

`list` in Clojure has a very specific meaning compared to lists in general computer science. The Clojure `list` function is a constructor for `PersistentList` class, a general purpose data structure.

The "list" abstract data type instead, defines an iterable data structure that has ordered access. Typical implementations are the single or double linked lists found in many languages. Clojure `list` is an example of single linked list. Compared to arrays, lists are dynamic as they can grow and shrink in size at run-time.

Note

cons into a `list` is possible but produces another list with type `clojure.lang.Cons` instead of `clojure.lang.PersistentList`. From that point forward the list forms an hybrid between the two cell types and any subsequent operation (such as into or conj) produces a `clojure.lang.Cons` type that is not reducible nor counted. conj should always be the first choice to add element to a collection, especially for `list` where cell type can be easily mixed without noticing.

See also

- vector, compared to `list`, offers direct lookup of elements by index.
- conj is the primary tool to push elements into a `list` after construction.
- cons is also a form of list (for which list*is the constructor, note the "star" at the end of the name). cons chains are not generally used as data structure as they offer very limited flexibility (they are neither counted nor reducible). cons main use case is as building blocks for lazy sequences.

- seq generates a lazy sequence that apart from laziness behave similarly to `list`.

Performance considerations and implementation details

=> `O(n)` **linear in number of args**

A `list` is created by chaining `clojure.lang.PersistentList` cell objects. Access to an element is linear with worst case `O(n)` when fetching the last element. It follows that lists perform best when operations happen on the head. Full scan searches or access by index suffers from linear behavior and should be used sparingly.

Lists are counted and reducible. Counted means that during conj operations they update an internal counter that can be later retrieved in constant time. Being reducible is about offering a special implementation to reduce. We can see why this matters in the following benchmark:

```
(require '[criterium.core :refer [quick-bench]])

(defn alist [n] ; ❶
  (into (list) (range n)))

(defn acons [n] ; ❷
```

```
    (reduce #(cons %2 %1) () (range n)))

  (let [l1 (alist 1e5)] (quick-bench (reduce + l1)))
  (let [l2 (acons 1e5)] (quick-bench (reduce + l2)))

  (let [l1 (alist 1e5)] (quick-bench (count l1)))
  (let [l2 (acons 1e5)] (quick-bench (count l2)))
```

❶ `alist creates a `list` using into.

❷ `acons needs to cons into an initial empty `list` to create a linked list of cons cells. The dedicated constructor list*only accepts a limited number of cells.

❸reduce on a `list` is twice as fast as the same operation on a list of cons cells.

❹count is 2 orders of magnitude faster on a `list` compared to a cons-list of the same length. Note that we go from nanoseconds to milliseconds. count on a cons-list is a linear operation.

Lists support the sequence interface without the need for an adapter. At the same time, it prevents `list` to provide chunked behavior like ranges or vectors. Worth remembering that `lists` are sequential but not lazy: their creation already implies the evaluation of all the elements.

Since one of the practical use case for lists is to implement a stack, let's compare them to vectors. The following `check` function is used to verify balanced parenthesis (the example is presented fully in peek). All we need to do to use a different kind of stack is to pass a different `stack` parameter to the function:

```
(require '[clojure.set :refer [map-invert]])

(def push conj)
(def brackets {\[ \] \( \) \{ \}})

(defn check [form stack] ; ❶
  (reduce (fn [q x]
    (cond
      (brackets x) (push q x)
      ((map-invert brackets) x)
      (if (= (brackets (peek q)) x)
        (pop q)
        (throw
          (ex-info
            (str "Unmatched delimiter " x) {})))
      :else q)) stack form))

(check "(let [a (inc 1]) (+ a 2))" ())
;; ExceptionInfo Unmatched delimiter ]
```

```
  (check "(let [a (inc 1)] (+ a 2))" ())
  ;;()
```

❶All we need to do to change stack implementation is passing something different as `stack` parameter to the `check` function.

We are now going to compare a vector stack and `list` stack for the `check` function:

```
  (require '[criterium.core :refer [quick-bench]])

  (def small  (str (seq (take 100  (iterate list (
  (def large  (str (seq (take 1000  (iterate list

  (quick-bench (check small ())) ;; 3.24 ms
  (quick-bench (check small [])) ;; 4.80 ms

  (quick-bench (check large ())) ;; 317 ms
  (quick-bench (check large [])) ;; 386 ms
```

❶ `small` and `large` contain similar patterns of nested parenthesis, for example "". The pattern repeats at different depths up to the given maximum as directed by take, forcing deeper stacks and stressing the different stack implementations.

We can see that `list` outperforms vector implementing a stack, although not by a big margin.

9.4.2 cons and list*

`function` `since 1.0`

Listing 9.19. → *Collections, Data type, Sequences*

```
(cons [x seq])

(list*
  ([args])
  ([a args])
  ([a b args])
  ([a b c args])
  ([a b c d & more]))
```

`cons` creates a new `clojure.lang.Cons` data structure by linking the given element to a sequential tail:

```
(cons :a [1 2 3]) ; ❶
;; '(:a 1 2 3)
```

❶ `cons` takes the element to be added and another sequential data structure, joining them together in a new sequential view.

The output of `cons` is itself sequential and can be used for another `cons` forming gradually longer linked lists of cons cells:

```
(cons 1 (cons 2 (cons 3 (cons 4 ())))) ; ❶
;; (1 2 3 4)
```

❶ `cons` invocations are nested to form a linked list.

Beyond the first few items, `list*` can be used to create longer `cons` chains and avoid repetition:

```
(list* 1 2 3 4 5 ()) ; ❶
;; (1 2 3 4 5)
```

❶ `list*` is used to create a linked list of cons cells by repeatedly applying `cons` on each element in the input. Similarly to `cons`, the last element needs to be sequential.

List of `cons` cells are rarely used to create large data structures (Clojure for example uses `list*` internally to compose arguments into a single list). `cons` is used primarily as the building block of lazy sequences.

Note

Despite the name, `list*` creates a `clojure.lang.Cons` not a `clojure.lang.PersistentList` (as list does). `cons*` would likely be a better name.

Contract

input

- "x" can be any type and is a required for `cons`.
- "seq" is a sequential collection (as per seq contract) or `nil`.
- "a", "b", "c" and "d" for `list*` can be of any type. "a" is the last item being added to the resulting `cons` list.
- "args" in `list*` indicates that the last argument is different from the other and it's required to be sequential or `nil`.

- "more" in `list*` allows any number of arguments but the last needs to be a sequential collection or `nil`.

notable exceptions

- `IllegalArgumentException` is thrown when "seq" is not a sequential. This happens when inverting the arguments, which is typically the result of confusing conj with `cons`.

output

`cons` returns:

- The `clojure.lang.Cons` instance containing "x" as the first element and "seq" as the rest.
- When "seq" is `nil` it returns `clojure.lang.PersistentList` instead of `clojure.lang.Cons`.

`list*` returns:

- A linked list composed by `clojure.lang.Cons` cell objects and "args" as the last element.

Examples

Creating a cons-list longer than a few items is possible but not encouraged, as their use as collection is penalized by general performance considerations. If you are thinking of using apply to create longer chains, keep in mind that the end result might not be a pure cons-list:

```
(def l (apply list* -2 -1 (range 10) ())) ; ❶
;; (-2 -1 0 1 2 3 4 5 6 7 8 9)

(type (next l)) ; ❷
;; clojure.lang.Cons

(type (nnext l)) ; ❸
;; clojure.lang.LongRange
```

❶ `nil` or `()` are both valid options as `list*` last element.
❷ If we start to inspect the types of the elements, we can see they are `clojure.lang.Cons` as expected.
❸ But at the point the range was concatenated in, there is the entire a `clojure.lang.LongRange` instead of a `cons` object.

To create a pure cons-based list, we can use reduce:

```
(def l (reduce #(cons %2 %1) () (range 9 -3 -1)))
;; (-2 -1 0 1 2 3 4 5 6 7 8 9)
```

```
(type (nthrest l 10)) ; ❷
;; clojure.lang.Cons
```

❶Using reduce to repeatedly invoking `cons`.

❷Going deeper into the elements of the list shows that they are all `clojure.lang.Cons` objects.

As discussed in the introduction, one of the main use case for `cons` is to build lazy sequences. Let's review the typical sequence generation scenario and focus on the use of `cons`:

```
(defn lazy-loop [xs] ; ❶
  (lazy-seq
    (when-first [x xs]
      (cons x           ; ❷
        (lazy-loop (rest xs))))))

(last (lazy-loop (range 100000))) ; ❸
;; 99999
```

❶ `lazy-loop` generates a lazy sequence from its input without any transformation. One side effect of the apparently useless loop, is the removal of chunks when iterating chunked sequences.

❷ After checking if there are more items to process, it pushes the current item into the next recursion with `cons`.

❸ We can access elements that are far away from the head without consuming the stack, even without tail recursion.

`cons` is designed to handle a sequential tail (not another `clojure.lang.PersistentList` like in the list case). For this reason `cons` does not necessarily evaluate its content allowing for the typical suspension of evaluation produced by lazy-seq. As a counter example, the following attempt to create a lazy sequence using `PersistentList` is stack consuming:

```
(defn lazy-loop [xs]
  (lazy-seq
    (when-first [x xs]
      (conj ; ❶
        (lazy-loop (rest xs))
        x))))

(last (lazy-loop (range 100000))) ; ❷
;; StackOverflowError
```

❶Compared to the previous example, we are using conj instead of `cons`. Note that we had to invert the argument order.

❷This time, lazy-seq is unable to hold the body unevaluated as an effect of conj a `PersistentList` not having an option for a sequential tail.



Note

Cons cells in Clojure are different from cons cells in other Lisps. In most Lisps, a "cons" holds pointers to arbitrary objects. In those Lisps, you could use cons cells to build trees for instance. A Clojure `cons` takes an arbitrary Object as head but only allows an `ISeq` as tail.

See also

- conj understands all Clojure types including `list`, invoking the right "append" semantic based on the type of the input collection. Use `cons` to create lazy sequences (or small throw-away lists) and if you are absolutely certain that conj won't work in your case.
- lazy-seq understands `cons` to which is intimately connected for the generation of lazy-sequences.

Performance considerations and implementation details

=> `O(1)` **constant time, cons**

=> `O(n)` **steps, list***

`cons` creates an instance of `clojure.lang.Cons` in constant time. The second argument can be any sequential object, including another `clojure.lang.Cons` instance. The tail is only evaluated if necessary, so `cons` can be considered lazy in regards of the second argument evaluation.

In terms of performance, the natural comparison is with list. This was already done in the list chapter so the reader is invited to review that part.

`list*` needs to iterate through the arguments to build the linked list, so its behavior is linear in the number of items in the input.

[167] `core.clj` is main standard library file in the Clojure codebase:
github.com/clojure/clojure/blob/master/src/clj/clojure/core.clj
[168] Nearest smaller value search is an optimization used in many algorithms, for instance merge sort. Please see the Wikipedia entry for an overview:
en.wikipedia.org/wiki/All_nearest_smaller_values

9.5 SUMMARY

This chapter is about how sequence are created. The most common way is by iterating an existing collection sequentially, but sequences can also be generated from a computational recipe or external data. The chapter also includes two data structures which implements the sequence interface directly: list and cons. In the next chapter we are going to explore the functions in the standard library that process sequences.

10 SEQUENTIAL PROCESSING

This chapter is about functions and macros for sequential processing. These functions typically transform their input into a sequence (if it's not already) and produce another sequence. Although sequential functions can be used with any collection type offering a sequential interface, they tend to perform at their best with pure sequential input/output.

Sequential processing can be broadly categorized as follows:

- Splitting: isolate a consecutive portion of the sequence by index, number of items, or using a custom predicate. Functions like split belong to this group, and also drop, rest and their variations.
- Selection: retrieve specific items from the sequence but not necessarily as a consecutive selection, for example take-nth or filter.
- Transformation: apply a function to each item in the sequence to produce another sequence, like map, keep or mapcat.
- Combining: combination of multiple sequences to form another sequence, like interpose or concat.

- Chunking: process a sequence by groups of multiple elements instead of one at a time. Clojure provides a small API to create chunked sequences described in the chunk-cons section.

Some functions like first , map or filter also belong here, but they have given specific emphasis in the "Basic Constructs" chapter.

10.1 REST, NEXT, FNEXT, NNEXT, FFIRST, NFIRST AND BUTLAST

`functions` `since 1.0`

Listing 10.1. → *Seq-in-seq-out, Sequential processing*

```
(rest     [coll])
(next     [coll])
(butlast  [coll])
(nnext    [coll])
(nfirst   [coll])
(fnext    [coll])
(ffirst   [coll])
```

`rest` , `next` , `butlast` `nnext` and `nfirst` generate a lazy sequence after removing at most 1 element from either the

head or the tail of the input collection. `fnext` and `ffirst` return a single element instead.

The following table summarizes the functions in this section and their goals:

Table 10.1. A summary of the functions available to extract a portion of a sequence.

Name	Description	Input	Output
rest	"coll" but the first item, or empty list.	`'(0 1 2 3)`	`(1 2 3)`
next	"coll" but the first item, or `nil`.	`'(0 1 2 3)`	`(1 2 3)`
butlast	"coll" but the last item, or `nil`.	`'(0 1 2 3)`	`(0 1 2)`
nnext	Next of next of "coll".	`'(0 1 2 3)`	`(2 3)`
nfirst	Next of the first of "coll".	’0 1 2	`(1 2)`
fnext	First of the next of "coll".	`'(0 1 2 3)`	`1`
ffirst	First of the first of "coll".	`'0 1 2`	`0`

As you can see, the functions differ in a few aspect, like what they return if there are no more items (empty sequence or `nil`), which side of the input they operate on (beginning or end) or if they return a sequence or a single item.

`rest` and `next` (along with first) play an important role in recursive algorithm definitions over sequential inputs. Their combinations (`fnext`, `nnext`, `ffirst` and `nfirst`) spares a few keystrokes and parenthesis when necessary.

Contract

input

- "coll" can be any sequential input or `nil`. A collection is "seqable" when it provides a sequencing strategy for seq. Most of Clojure data structures are sequential as are the most important Java data structures.

notable exceptions

- `IllegalArgumentException` especially for `ffirst` and `nfirst` which assume nested data structures.

output

- `rest`, `next`, `butlast` and `nfirst` return a sequence. In case there are no items to return for the requested operation, `rest` returns an empty list while the others return `nil`.
- `ffirst`, `nnext` and `fnext` return a single item or `nil` if the operation results in no item being available.

Examples

By using juxt we can quickly understand the behavior of theses functions on `nil` and empty collection:

```
((juxt rest next butlast nfirst ffirst nnext fne
;; [() nil nil nil nil nil nil]

((juxt rest next butlast nfirst ffirst nnext fne
;; [() nil nil nil nil nil nil]
```

❶All functions can be called on `nil`. `rest` is the only one returning an empty list.

❷All functions also accept an empty collection as input producing exactly the same result.

As mentioned in the introduction, `rest` (or `next`) are part of the fundamental recursive idiom for sequences along with first:

```
(defn rest-loop [coll] ; ❶
  (loop [xs coll results []] ; ❷
    (if-let [xs (seq xs)] ; ❸
       (recur
          (rest xs) ; ❹
          (conj results (first xs)))
        results)))

(rest-loop (range 10))
;; [0 1 2 3 4 5 6 7 8 9]
```

❶ `rest-loop` iterates over the elements of the given collection and puts them in a vector without any transformation.

❷ The loop-recur construct defines the defaults. We bind "coll" to "xs" inside the loop, so we are free to consume it each iteration. "results" holds the gradual accumulation of the output.

❸ We need to check if there are elements in "xs" before starting another recursion. seq is used here to transform a potential empty list (what rest returns) in a `nil` so it can be used in the if-let condition.

❹ If there are more elements to process, the first one is added to the results and the rest of "xs" is used for the next recursion.

It should be now apparent why `next` is useful if we want to use the condition on the collection itself. Since `next` returns `nil` to signal the end of the input (instead of empty list), we can use the data structure itself as a logical boolean (a Clojure idiom known as "nil punning"). By doing so we can remove the if-let binding:

```
(defn next-loop [coll] ; ❶
  (loop [xs (seq coll) results []]
    (if xs ; ❷
      (recur
        (next xs) ; ❸
```

```
          (conj results (first xs)))
        results)))

(next-loop (range 10))
;; [0 1 2 3 4 5 6 7 8 9]
```

❶ `next-loop` is a rewrite of the previous `rest-loop` to take advantage of the nil-punning quality of `next`.

❷ The if condition now happens directly on "xs" which is the current view of the sequence. Note however that we had to add a one-off call to seq during `loop` binding of "xs" to handle the empty input case: `(next-loop [])` should result in `[]` and not `[nil]`.

❸ `next` is used instead of `rest`.

Note that the two functions `rest-loop` and `next-loop` are designed to fully consume their input, without specific concerns about laziness. If however the input was something extremely expensive to compute, then we might be interested in the difference between `rest` and `next` in terms of laziness. To illustrate the point, let's now create a lazy-seq recursive loop using `next`:

```
(defn lazy-expensive []   ; ❶
  (map #(do (println "thinking hard") %)
```

```
    (into () (range 10))))

  (defn lazy-loop [xs]      ; ❷
    (lazy-seq
      (when xs
        (cons
          (first xs)
          (lazy-loop (next xs))))))) ; ❸

  (first (lazy-loop (lazy-expensive))) ; ❹
  ;; thinking hard
  ;; thinking hard
  ;; 9
```

❶Our input is an expensive lazy sequence. `lazy-expensive` produces a side effect by printing on screen so we can see when something is produced.

❷ `lazy-loop` uses the recursive lazy-seq idiom to build a lazy sequence on top of the input. As author of the function we don't know what kind of input will be passed in, but we guarantee to the outside world that we are going to consume it lazily.

❸We decide to use the `next` looping style, taking advantage of `nil` punning in the when condition.

❹There are two prints on screen when we ask for the first element.

In a real life scenario, `lazy-expensive` could be producing side-effects like reading large files in memory. Laziness gives us an opportunity to just consume enough of the expensive computation when necessary, but in order to avoid consuming an additional (unwanted) item, we have to use `rest`:

```
(defn lazy-loop [xs] ; ❶
  (lazy-seq
    (when-first [x xs]
      (cons x
        (lazy-loop (rest xs))))))) ; ❷

(first (lazy-loop (lazy-expensive))) ; ❸
;; thinking hard
;; 9
```

❶ `lazy-loop` has been changed to accommodate for `rest` recursive style. We are using the handy when-first shortcut which expands into an assignment of `(first xs)` into the local binding "x".

❷We can now use `rest` instead of `next`.

❸The output is now fully lazy, without consuming additional items than the ones actually requested.

In the following example, we are going to use `butlast` to implement version of into that doesn't require the use of comp

to compose transducers (a similar mechanism is implemented for eduction in the standard library):

```
(defn into* [to & args] ; ❶
  (into to
    (apply comp (butlast args))  ; ❷
    (last args)))

(into* [] (range 10)) ; ❸
;; [0 1 2 3 4 5 6 7 8 9]
(into* [] (map inc) (range 10))
;; [1 2 3 4 5 6 7 8 9 10]
(into* [] (map inc) (filter odd?) (range 10))
;; [1 3 5 7 9]
```

❶Our `into*` is reusing into internally after processing the parameters. After the first parameter "to" there is a catch-all "args" that can optionally include transducers.

❷We isolate potential transducers in the arguments with `butlast`. We know that an origin collection is always required, so we can safely exclude the last argument. We can rely on into regarding parameters validation.

❸A few tests to verify that `into*` works as intended.

`ffirst` and `fnext` can be useful to implement a form of "look-ahead" behavior in a nested sequence, for example to pair

up incoming items or mark an item as "incomplete" when reaching the end of the sequence:

```
(def message [["A" 1 28] ["H" 37 82 11] ["N" 127

(defn process [message] ; ❷
  (lazy-seq
    (when-let [xs (seq message)]
      (let [e1 (ffirst xs) e2 (fnext xs)] ; ❸
        (cons (if (nil? e2)
                {:item e1 :succ :incomplete}
                {:item e1 :succ e2})
          (process (rest xs)))))))

(process message) ; ❹
;; ({:item "A", :succ ["H" 37 82 11]}
;;  {:item "H", :succ ["N" 127 0]}
;;  {:item "N", :succ :incomplete})
```

❶ The sample `message` presented here is a short vector, but the `process` function is designed to accept an (arbitrarily long) lazy sequence and return another lazy sequence.

❷ `process` generates a lazy sequence from the input message. We have an option to customize the recursion so we can check following elements in the sequence to decide what to do.

❸ `ffirst` and `fnext` are convenience functions to access the current head of the element and the next element.

❹ In the final output we can see that each element is potentially linked to the next and the last one is marked as "incomplete".

Why there is a `rest` and a `next`?

`next` was introduced at the time when lazy sequences were redesigned [169].

`rest` used to do what `next` does now: either the collection contained one more item or `nil`. To do so, at least the next item needed to be evaluated to see if there was more. Although this wasn't a problem in most cases, it created surprises at times when items got evaluated just because they were temporarily at the head of the sequence.

To prevent any form of eagerness, `rest` was implemented so it would never evaluate elements unless explicitly required (with seq). `next` was created to maintain the old `rest` behavior for those cases where laziness was not an issue.

See also

- first and last are other popular functions to access specific element of a sequence.
- second is equivalent to `fnext` to access the second element in a sequence. There is no "third" function or following ordinals.
- pop is the equivalent of `butlast` for vectors. `butlast`, in order to remove the last element, needs to walk the entire sequence achieving linear behavior (the worst case). Vectors are specifically optimized for tail access and `pop` is the correct way to get rid of the last element.

Performance considerations and implementation details

=> `O(1)` **(rest, next, fnext, nnext, ffirst, nfirst)**

=> `O(n)` **(butlast)**

The functions in this section are mostly constant time as they manipulate the head of the sequence only. `butlast` is the only case of linear behavior because it needs to walk the entire sequence to drop the last element.

Laziness differs between `rest` and `next` (including `fnext` and `nnext`). The difference is important for the generation of custom lazy sequences, where the choice between `rest` or

`next` determines the evaluation of one additional item. The unwanted evaluation is even more important when we think that the step size of a lazy sequence is not necessarily 1. The standard library contains chunked sequences for instance, which are evaluated by chunks of 32 items:

```
(defn counter [cnt] ; ❶
  (fn [x] (swap! cnt inc) x))

(defn not-chunked [f] ; ❷
  (let [cnt (atom 0)]
    (f (drop 31 (map (counter cnt) (into () (ran

(defn chunked [f] ; ❸
  (let [cnt (atom 0)]
    (f (drop 31 (map (counter cnt) (range 100)))

(not-chunked rest) ; 32
(not-chunked next) ; 33
(chunked rest) ; 32
(chunked next) ; 64
```

❶ `counter` creates a mapping function which closes over a mutable atom. We can use the state to count how many items are flowing through the sequence.

❷ `not-chunked` creates a non-chunked sequence by using a persistent list as the source for the map operation. map will call seq on list which doesn't use chunking. Note that the fact that the list is created on top of a chunked range is neutralized as soon as we use it build the list.

❸ `chunked` on the other hand, creates a chunked sequence by using the range directly. Both `not-chunked` and `chunked` drops 31 items before applying "f" to the resulting sequence.

The 4 results we see when executing `not-chunked` and `chunked` can be explained by the following:

1. We call `rest` on a non-chunked sequence after dropping 31 items. The item at index 31 is evaluated in order for `rest` to move forward. The total of items evaluated is 32 (they start from zero).
2. We call `next` on a non-chunked sequence, again dropping the first 31 items. The item at index 31 is evaluated for `next` to get past and the item at index 32 is evaluated to establish if to return `nil` or not. 33 items in total are evaluated.

3. We call `rest` on a chunked sequence after dropping 31 items. The item at index 31 needs to be evaluated to get past to the next chunk in the sequence, which is not evaluated yet. 32 items are evaluated in total, the entire content of the chunk from index 0 to 31.
4. Finally, we call `next` on the chunked sequence. We dropped 31 items, item at index 31 needs evaluation, as well as item at index 32 to verify the end of the sequence. But item at index 32 sits in the next chunk, therefore the second chunk is evaluated, resulting in a total of 64 items evaluated.

Chunked evaluation is a trade-off that gives up some laziness in exchange for the preemptive caching of additional items beyond what is actually requested. Given that 32 is the branching factor of persistent data structures, there is clearly a correlation: the evaluation of a chunk corresponds to the bulk array copy of a node in the hash array mapped trie (please refer to the introduction of the vectors chapter to know more about the implementation details of Clojure persistent data structures).

[169] The rationale for introducing lazy-seq and remove `lazy-cons` is described on this page of the main Clojure website: clojure.org/reference/lazy. The page is still there mainly for historical purposes.

10.2 DROP, DROP-WHILE, DROP-LAST, TAKE, TAKE-WHILE, TAKE-LAST, NTHREST, NTHNEXT

`functions`

`since 1.0 (drop,drop-while,drop-last,take,take-while,nthnext)`

`since 1.1 (take-last)`

`since 1.3 (nthrest)`

Listing 10.2. → *Seq-in-seq-out, Sequential processing, Sequential selection, Transducers*

```
  (drop ([n]) ([n coll]))
  (drop-while ([pred]) ([pred coll]))
  (drop-last ([coll]) ([n coll]))

  (take ([n]) ([n coll]))
  (take-while ([pred]) ([pred coll]))
  (take-last [n coll])

  (nthrest [coll n])
  (nthnext [coll n])
```

The functions in this section generate a sequence of contiguous elements from an input collection. They offer different parameters to control how the selection should happen:

- By number of items from the head or from the tail of the input (`drop`, `drop-last`, `take`, `take-last`, `nthrest`, `nthnext`)
- Using a predicate (`drop-while` and `take-while`)
- Keep the head, drop the tail (`take`, `take-while`, `drop-last`)
- Drop the head, keep the tail (`drop`, `drop-while`, `take-last`, `nthrest`, `nthnext`)

The following table summarizes the functions in this section and their goals:

Table 10.2. A summary of the functions available to extract a

portion of a sequence.

Name	Description	Transducible?	Fully lazy?
drop	Drops the first n elements.	Yes	Yes
drop-while	Drops items until the given predicate returns `false`.	Yes	Yes
drop-last	Drops the last n elements (default to 1).	No	Yes
take	Keeps the first n elements, drops the rest.	Yes	Yes
take-while	Keeps element until the given predicate returns `false`.	Yes	Yes

take-last	Keeps the last n elements (no default for n).	No	No
nthrest	Like `drop` with inverted arguments. Never returns `nil`.	No	No
nthnext	Like `nthrest` but returns `nil` if beyond input length.	No	No

As the reader can see from the table, some functions provide a transducer version while others (specifically `take-last`, `nthrest` and `nthnext`) are partially evaluating their arguments. Please look at the "Examples" section below for more details.

Contract

input

- "n" can be any number: positive, zero, negative, integer or floating point. It defaults to 1 for `drop-last`. It's required argument for `take-last`, `drop`, `take`, `nthrest` and `nthnext`. Negative "n" is equivalent to passing zero.
- "pred" is a function of one argument returning logical `true` or `false`. It is required argument for `drop-while` and `take-while`.
- "coll" can be any sequential input or `nil` (a collection is "seqable" when it provides a sequencing strategy for seq or is a sequence itself). It is optional input for the transducer-aware functions: `drop`, `drop-while`, `take` and `take-while`.

notable exceptions

- `IllegalArgumentException` when "coll" is not sequential (as per seq contract).

output

The functions in this section generate a sequence from an input collection (except when returning the transducer version, see below):

- `drop` returns a sequence with the first "n" items removed, or an empty sequence if "n" bigger than `(count coll)`. When only "n" is present, it returns a transducer that removes the first "n" elements when used in a transducers chain.
- `drop-while` removes those items from the head of "coll" for which `(pred item)` returns logical `true`. An alternative way to describe it is: `drop-while` drops elements from "coll" stopping at the first time `(pred item)` returns `false`. When "coll" is not provided, returns the transducer version of the function. Returns empty sequence when there are not enough items to satisfy the request.
- `drop-last` removes "n" items from the tail of the input. Removes the last item when "n" is 1. Returns empty sequence when there are not enough items to satisfy the request.
- `take` keeps the first "n" items, dropping the rest. When "coll" is not provided, returns a transducer version that keeps the first "n" element. Returns empty sequence when "n" is larger than `(count coll)`.

- `take-while` keeps those items for which `(pred item)` is `true`. When "coll" is not provided returns the transducer version of the function. Returns empty sequence when there are not enough item to satisfy the request.
- `take-last` keeps the last "n" elements starting from the tail of "coll". It returns empty sequence if "n" is larger than `(count coll)`.
- `nthrest` removes the first "n" items from "coll". It returns empty list when "n" is larger than `(count coll)`.
- `nthnext` is like `nthrest`, but returns `nil` when "n" is larger than `(count coll)`.

The functions returning a transducer when "coll" is not present are: `take`, `take-while`, `drop` and `drop-while`. The returned transducer follows the transducers contract and accept a reducing function to use in a transducing context.

Examples

`drop` and `take` are typically used when the number of items to drop or take is known. For example, if we wanted to process the remaining days in the current year (assuming today is December 25th):

```
(import '[java.util Calendar])
(def day-of-year (.get (Calendar/getInstance) (C

(drop day-of-year (range 1 366)) ; ❷
;; (360 361 362 363 364 365)
```

❶One way to obtain the number for the current day of the year.
❷Once we have the current day of the year, we can `drop` it from the range of the days in a year.

We can use `take` in a similar way, for example to extract information that always appear at the beginning of a collection. In the following example, a message hub contains messages from other applications. The message is encoded as a vector starting with an error code. We only want to process some messages and discard others based on the error code and the month the message was generated:

```
(def hub-sample ; ❶
 [[401 7  :mar "-0800" :GET 1.1 12846]
  [200 9  :mar "-0800" :GET 1.1 4523]
  [200 2  :mar "-0800" :GET 1.1 6291]
  [401 17 :mar "-0800" :GET 1.1 7352]
  [200 23 :mar "-0800" :GET 1.1 5253]
  [200 7  :mar "-0800" :GET 1.1 11382]
```

```
  [400 27 :mar "-0800" :GET 1.1 4924]
  [200 27 :mar "-0800" :GET 1.1 12851]])

(defn error-in-month? [code month] ; ❷
  (and (= (>= code 400) (= :mar month))))

(defn process-errors [hub-messages] ; ❸
  (filter #(let [[code _ month] (take 3 %)]
    (error-in-month? code month))
    hub-sample))

(process-errors hub-sample)
;; ([401 7 :mar "-0800" :GET 1.1 12846]
;;  [401 17 :mar "-0800" :GET 1.1 7352]
;;  [400 27 :mar "-0800" :GET 1.1 4924])
```

❶A small sample of the content from the message hub has been created for this example.

❷The `error-in-month?` function isolates error codes for a specific month.

❸ `process-error` contains the logic to iterate through the messages in the hub. It uses a filter function to isolate the interesting messages. The predicate of the `filter` only takes the first 3 items from each message, which include an error code, a day of the month (which is not used) and the name of the month this message belongs to. The error code and the month are sent to `error-in-month?` to decide if the message should be kept.

We can instead use a predicate function when there is a rule driving what should be taken/removed. The following example shows how we could implement a function to isolate contiguous items in a list and generate a lazy sequence of them:

```
(defn tokenize [pred xs]
  (lazy-seq
    (when-let [ys (seq (drop-while (complement p
      (cons (take-while pred ys) ; ❷
            (tokenize pred (drop-while pred ys))

(def digits '(1 4 1 5 9 2 6 4 3 5 8 9 3 2 6))
```

```
(tokenize odd? digits) ; ❹
;; ((1) (1 5 9) (3 5) (9 3))
```

❶ The first step involves `drop-while` to remove all the items we don't want from the head of the list. Once we are positioned on something we are interested in (the first odd number in this example) we start collecting results. Note how we need to use seq here, as `drop-while` returns an empty list when we are at the end of the input.

❷ `take-while` collects the items we want to group and isolate from the head of the sequence. Those are pushed with cons to the lazy sequence under construction.

❸ Before we start over, we need to `drop-while` all the items that we just collected, as we don't want to have them again for the remaining part of the input.

❹ We can use a list of digits to test the results. The resulting lazy sequence contains all the groups of contiguous odd digits present in the input.

`take`, `drop` and `take-while`, `drop-while` also provide a transducer version:

```
(transduce (comp (drop 3) (map inc)) + (range 10
;; 49
```

❶We can see the transducer version of `drop` used to remove the first 3 items before summing them up with `+`.

> Warning
> `take`, `drop`, `take-while` and `drop-while` are stateful transducers. Stateful transducers produce inconsistent results in a concurrent scenario. You should pay attention when using them with fold or core.async `pipeline` construct (see clojuredocs.org/clojure.core.async/pipeline and the transducer chapter for more information).

Laziness considerations

You need to pay attention to some of the functions in this section when dealing with infinite sequences:

```
(defn xs [] (map #(do (print ".") %) (iterate inc 0))) ; ❶

(def take-test (take 1e7 (xs))) ; ❷
;; #'user/take-test

(def time-bomb (drop 1e7 (xs))) ; ❸
#'user/time-bomb
```

❶ We setup a testing infinite lazy sequence with side effecting printing of the number of evaluated items. We can use it to verify the way a sequence processing function supports laziness.

❷ `take` from an infinite sequence does not produce any evaluation as expected and should not give surprises.

❸ `drop` behaves the same and does not evaluate any item at creation. You have to be careful though, as any further operation on "time-bomb" will evaluate `1e7` (10 million) elements.

There are also different degrees of laziness. Functions like `take-last` eagerly evaluates their input at creation time, so you need to be extra careful:

```
(def eager (take-last 1 (take 10 (xs)))) ; ❶
;; ..........#'user/eager

(def lazy-bomb (drop-last (xs))) ; ❷
;; #'user/lazy-bomb
```

❶ This call to `take-last` element would hang forever without first taking 10 elements out of the infinite sequence "xs". Note that 10 dots have been printed showing that `take-last` already evaluated the result.

❷ `drop-last` doesn't suffer the same problem, but it can create a `lazy-bomb` (like in this case) as any operation touching the sequence would incur in infinite evaluation.

`nthrest` and `nthnext` also evaluate eagerly, but they are lazier than `take-last`:

```
(def lazier (nthrest (xs) 3)) ; ❶
;; ...#'user/lazier
```

❶ We can see 3 dots printed, showing that `nthrest` evaluated the first 3 elements (those that are going to be discarded anyway).

One interesting effect of using eager evaluation with `nthrest` or `nthnext` is the possibility to execute side effects even when remaining elements in a sequence are never evaluated. In the next example, a service produces a sequence backed by temporary files connected to each item. Some application logic decides to drop elements from the sequence before returning it to a client. After the control returns from the service we don't know if the client is going to consume the rest of the sequence or not, but we certainly know that we can clean the files related to the items dropped from the sequence:

```
(require '[clojure.java.io :as io])
(import  '[java.io File])

(defn generate-file [id] ; ❶
```

```
    (let [file (File/createTempFile (str "temp" id
      (with-open [fw (io/writer file)]
        (binding [*out* fw]
          (pr id)
          file))))

  (defn fetch-clean [f] ; ❷
    (let [content (slurp f)]
      (println "Deleting file" (.getName f))
      (io/delete-file f)
      content))

  (defn service [] ; ❸
    (let [data (map #(generate-file %) (list 1 2 3
      (nthrest (map fetch-clean data) 2)))

  (def consumer (service)) ; ❹
  ;; Deleting file temp1-8176280320841013882.tmp
  ;; Deleting file temp2-6114428806665839159.tmp
```

❶ `generate-file` creates a temporary file with a number as content for testing purposes.
❷ `fetch-clean` is a processing function for the file. It loads the content and removes the file once done.
❸ `service` contains application logic that removes elements from the start of the sequence. We simplified this logic to the bare minimum, just to show the effect of `nthrest`.
❹ A `consumer` of the service may decide to consume the data straight away or wait for some other condition to happen. While this is happening, anything that was dropped from the sequence has already cleaned up, as we can see from the messages appearing regarding deleting 2 files.

By using `nthrest` instead of `drop`, we make sure that only the relevant files are still available for a future computation and eagerly dispose of the rest freeing up resources.

See also

- rest are similar functions to those in this section, which default "n" to 1 for most of their operations.
- subvec can be used to extract portions of a vector in a similar way to `drop` or `take` with better performance.

- pop should be used instead of `drop-last` on vectors, as `drop-last` would convert the vector to a sequence first.

Performance considerations and implementation details

=> `O(1)` **time and space, best case**

=> `O(n)` **step and space, worst case**

For most of the functions in this section, the worst case is fully consuming the input sequence determining a linear behavior. Here's the combinations of parameters forcing full evaluation of an input sequence "xs" of length "n":

```
(drop n xs) ; ❶
(drop-while (constantly false) xs) ; ❷
(drop-last xs) ; ❸
(take n xs) ; ❹
(take-while (constantly true) xs) ; ❺
(take-last 1 xs) ; ❻
(nthrest xs n) ; ❼
(nthnext xs n) ; ❽
```

❶ `drop` over the length of the input, fully evaluates the input but does not retain any element in `O(1)` space.
❷ Similarly for `drop-while` with an always `false` predicate.
❸ `drop-last`, with default "n" of 1, fully evaluates its input and hold on the head, determining `O(n)` space.
❹ `take` over the length of the sequence is equivalent to the sequence itself, plus another copy of its elements for the sequence just created by `take`. This operation is `O(n)` steps and `O(n)` space.
❺ Same happens to `take-while` with an always `true` predicate.
❻ `take-last` for a single element is `O(n)` steps and `O(1)` space.
❼ `nthrest` over the length of the input evaluates all but holds nothing in `O(1)` space.
❽ Same as `nthrest` for `nthnext`.

There are differences in the level of laziness offered by some functions. `take-last`, `nthrest` and `nthnext` for instance, evaluate results partially at creation time, as outlined in "Laziness considerations" above.

Using sequence processing functions on data other than sequential is not usually a good idea, as they need to be converted into sequences. When necessary equivalent

operations exist for vectors for example subvec. Here is how we would write `drop` and `take` in terms of `subvec`:

```
(defn dropv [n v] (subvec v n (count v))) ; ❶

(dropv 5 (vec (range 10)))
;; [5 6 7 8 9]

(defn takev [n v] (subvec v 0 n)) ; ❷

(takev 5 (vec (range 10)))
;; [0 1 2 3 4]
```

❶ `drop` written for vectors using subvec.

❷ `take` written for vectors using subvec.

10.3 KEEP AND KEEP-INDEXED

`function` `since 1.2`

Listing 10.3. → *Sequential transformations, Nil Filtering, Transducers*

```
(keep
  ([f])
  ([f coll]))
```

```
(keep-indexed
  ([f])
  ([f coll]))
```

`keep` applies a transformation to the elements of an input collection and additionally removes any generated `nil`:

```
(keep first [[1 2] [2] [] nil [0] [2]]) ; ❶
;; (1 2 0 2)
```

❶ `keep` in used to take the first from each of the collections in the input. Note that when first happens on an empty collection (or `nil`) the generated `nil` does not appear in the final output.

`keep` has certainly a lot in common with map including a `keep-indexed` version which also maintains an index to the current item:

```
(keep-indexed #(nthnext (repeat %2 %1) %1) [1 3 
;; ((0) (1 1) (2 2 2 2 2 2))
```

❶Similarly to map-indexed, `keep` has a `keep-indexed` version. We are using it here to generate a sequence of repetitions of positive integers. The input collection drives the number of repetition for each index (e.g. at index 2 in the vector we have number 8) and the transformation function drops "index" elements from the repetition (e.g. if the index is 2 and the number at that index is 8, it repeats the number 2 eight times and then remove 2 of them). When the number at index is the same the index (e.g. the numbers 4,5,6 in the vector) it produces a `nil` that is removed from the final result.

Unlike map, `keep` doesn't accept multiple collections arguments, but like map it provides a transducer version. Here's for example how we could replicate the functionality of random-sample:

```
(sequence (keep #(when (> 0.5 (rand)) %)) (range
;; (0 2 3 4 5 10 13 15 16 17 18 19)
```

❶ `keep` used to simulate random-sample.

Contract

input

- "f" is a function of one (`keep`) or two (`keep-indexed`) arguments. In the case of 2 arguments, the first is the index of the item passed as second argument. It is mandatory argument.
- "coll" is any sequential-aware collection following the seq contract. When "coll" is not present, it returns the transducer version.

notable exceptions

- `ArityException` when `map-indexed` is erroneously passed a function only accepting one argument.

output

- When "coll" is present: the lazy sequence generated by applying the transformation "f" to each element in "coll". If any application of "f" returns `nil` , `nil` is removed from the final output.
- When "coll" is not present, `keep` and `keep-indexed` return a transducer which accepts a reducing function as required by the transducers contract.

Examples

`keep` has been used a few times in the book. The reader is invited to review these examples:

- flatten has an example of retaining just the last item of the results of applying a regular expression with re-find.
- sequence contains a transducer example of `keep` to parse some structured output.
- seque contains an example of `keep` to remove unwanted `nil` from the output.

Idiomatically, `keep` is known as a more concise way to express `(remove nil? (map f coll))`:

```
(def dict {1 "one" 2 "two" 3 "three"}) ; ❶

(remove nil? (map dict [5 3 2])) ; ❷
;; ("three" "two")

(keep dict [5 3 2]) ; ❸
;; ("three" "two")
```

❶ `dict` is a dictionary which contains relationships between an integer id and a string.

❷ We can extract specific values using the `dict` as a function and putting interesting keys in a vector. But this introduces potential `nil` in the output for missing keys. So we are force to use `remove`.

❸ A better way to express the same computation is by using `keep`.

The following example shows how to implement a `first-index-of` function to return the position of an element in a sequence using `keep-indexed`:

```
(defn first-index-of [x coll] ; ❶
  (first (keep-indexed #(when (= %2 x) %1) coll)))

(first-index-of 2 (list 3 9 1 0 2 3 2)) ; ❷
;; 4

(first-index-of 11 (list 3 9 1 0 2 3 2)) ; ❸
;; nil
```

❶ The implementation of `first-index-of` takes advantage of the `nil` filtering provided by `keep`. `map-indexed` would produce a sequence of `nil` except for the matching item.

❷ By using `keep-indexed` the output contains a single element (or `nil`).

❸ Note that this is not a collection with a `nil` element, but just `nil` to indicate no match.

Extending `keep` to multiple collections

Unlike map, `keep` does not allow for multiple collections as input. We can extend `keep` to allow a variable number of collections and use it in those cases where the result should not contain `nil`:

```
(defn keep+ [f & colls] ; ❶
  (lazy-seq
    (let [ss (map seq colls)] ; ❷
      (when (every? identity ss) ; ❸
        (let [x (apply f (map first ss)) ; ❹
              rs (map rest ss)]
          (if (nil? x) ; ❺
            (apply keep+ f rs)
            (cons x (apply keep+ f rs))))))))

(keep+ #(and %1 %2 %3) ; ❻
  [1 2 nil 4]
  [5 nil 7 8]
  (range))
;; (0 3) ; ❼
```

❶The general design of the function follows the implementation of map. The idea is to iterate over all the first elements in each collection, apply the given function and decide when to cons results based on the presence of `nil`.

❷ `keep+` produces a lazy sequence and makes use of rest to prevent any unwanted evaluation. We then need to make sure that there is a least one more element in each collection by replacing any empty list with a `nil`.

❸With "ss" now containing either a list or `nil` (but no empty lists) we can use every> to make sure we can perform another iteration.

❹This is where we apply "f" to the first groups of items from each collection.

❺The logic for `keep` is in this condition, which goes into recursion with or without consing the next element into the output sequence.

❻An example use of `keep+` which answers the question: what indexes would produce not-nil values when applied to all input vectors?

❼The answer is 0 (would produce the values 1 and 5) and 3 (would produce 4 and 8).

See also

- map and map-indexed are similar to `keep` and `keep-indexed` without filtering for `nil`.
- filter can be used on top of any sequential processing to remove `nil` or other unwanted items.
- range can be used to generate an infinite list of positive integers to use as indexes in custom solutions not involving `keep-indexed`.

Performance considerations and implementation details

=> `O(n)` **steps worst case**

=> `O(n)` **space worst case**

`keep` and `keep-indexed` have very similar performance profile to map and other function to process sequences. When fully realized, the transformation applies to all elements producing a linear behavior.

As with other sequence processing functions, `keep` expects a sequential input and caches of the sequential output, producing linear behavior in space when the results are fully consumed.

Also see map and lazy-seq performance sections for more information.

10.4 MAPCAT

`function` `since 1.0`

Listing 10.4. → *Sequence transformation, Concatenation, Transducers*

```
(mapcat
  ([f])
  ([f & colls]))
```

`mapcat` functionality is well described by its name, the union of a map operation and the concatenation of the transformation produced by map. For the concatenation part to work correctly,

it is assumed that the transformation produces a sequential collection:

```
(map range [1 5 10]) ; ❶
;; ((0) (0 1 2 3 4) (0 1 2 3 4 5 6 7 8 9))

(mapcat range [1 5 10]) ; ❷
;; (0 0 1 2 3 4 0 1 2 3 4 5 6 7 8 9)
```

❶We map a range to a few numbers. This produces a sequence of increasingly longer inner sequences.
❷ `mapcat` removes the intermediate sequences.

`mapcat` accepts a variable number of collections. Here's an example of multiple collections used with the transducer version of `mapcat`:

```
(sequence (mapcat repeat) [1 2 3] ["!" "?" "$"])
;; ("!" "?" "?" "$" "$" "$")
```

❶ `mapcat` transducer is used with two collections input. The function repeat accepts two arguments: the first is the number of repetition (appearing in the first collection) and the second is the item to repeat (appearing in the second collection).

Contract

input

- "f" is a function of one or more arguments returning any sequential type (as described by seq contract) or `nil`. It is mandatory argument. The number of arguments accepted by "f" corresponds to the number of "colls" passed as input.
- "colls" is a variable number of 0 or more sequential collections (as described by seq contract). "colls" is optional.

notable exceptions

- `IllegalArgumentException` is typically thrown if the transformation "f" does not produce sequential output.

output

- returns: a lazy sequence of the concatenation of the transformations produced by applying "f" to each element in "coll". If multiple "colls" are present, "f" is applied to all first elements in "colls", then all second elements, and so on until reaching the end of the shortest collection. If no "colls" is provided, it returns `mapcat` transducer version.

Examples

`mapcat` with multiple collections can be used to isolate different ranges out of a larger set and merge them back together. Hexadecimal characters for instance, are defined by the range of numbers "0123456789" and the six letters "ABCDF". Both sets are available in the ASCII set at different indexes. The index range for numbers is 48-58 while the range for "ABCDEF" is 65-71:

```
(def hex?
  (set (sequence ; ❶
    (comp
      (mapcat range) ; ❷
      (map char))
    [48 65] ; ❸
    [58 71])))
```

```
  (every? hex? "CAFEBABE") ; ❹
  ;; true
```

❶ `hex?` function takes advantage of the fact that Set can be used as functions of one argument to decide if an element belongs to the set or not. We just need to def the set to the var `hex?` to create a properly working predicate.

❷ `mapcat` has to be the first transducer in the list, because it takes care of the input coming from multiple collections. It is followed by a `int` to `char` transformation.

❸ The 2 input vectors contain the index ranges for the characters we are interested in. All the lower bounds appears in the first vector and all the upper bounds are on the second vector. This is to allow range to receive them in the right order.

❹ We can use the predicate on a string with every?.

We can see `mapcat` in action in the following topological sort [170]. A typical problem in compilers, is to find the right order to load the modules in an application. The dependency of a module needs to load before the module itself. Similarly, if the dependency has other dependencies, they need to load first. We can solve the problem by sorting the direct graph formed by modules (nodes) and their dependencies (vertices).

The example uses a map `libs` of library names and their direct dependencies. We can use `mapcat` to extract the next layer of transitive dependencies using the map as a function. Let's have a look at this pattern in isolation first:

```
(def libs {:async [:analyzer.jvm] ; ❶
           :analyzer.jvm [:memoize :analyzer :re
           :memoize [:cache]
           :cache [:priority-map]
           :priority-map []
           :asm []})

(mapcat libs (:analyzer.jvm libs)) ; ❷
;; (:cache)
```

❶ `libs` is a map from each library name to their list of dependencies.

❷ We can use `libs` as a function for `mapcat` passing list of library names. Each key pulls a list of dependencies which are concatenated by `mapcat`. Here we can see what dependencies should be satisfied before we load `:analyzer.jvm`.

Using `mapcat` on any group of keys, we can see the first layer of transitive dependencies. To be sure all dependencies are

satisfied we need to iterate further until we find a library that has no dependencies. We can then walk the list of dependencies backward to read the order in which we should satisfy the dependencies:

```
(defn tp-sort [deps k]    ; ❶
  (loop [res () ks [k]]
    (if (empty? ks)     ; ❷
      (butlast res)
      (recur (apply conj res ks) ; ❸
             (mapcat deps ks)))))

(tp-sort libs :async) ; ❹
;; (:priority-map :cache :asm
;;  :reader :analyzer :memoize
;;  :analyzer.jvm)
```

❶ `tp-sort` implement a simplified form of topological sort (it's not detecting cycles). It contains a loop-recur which is used internally for recursion after preparing the initial arguments.
❷ The termination condition for recursion is an empty list of transitive keys dependencies, which means we've followed all dependencies.
❸ `mapcat` is used here to send the newly discovered layer of transitive dependencies to the next iteration. At the same time results are accumulated by conj into a list. The form with apply allows to treat the "ks" list like separated arguments to insert in the list, removing the unwanted nesting. Note that cons does not support a variable number of arguments.
❹ We ask `tp-sort` to find the order of dependencies to process so that the root module `:async` is satisfied.

Note that `tp-sort` is a generic function that works for modules, tasks, or any other form of direct acyclic graph (DAG). However, `tp-sort` could loop infinitely if the graph contains and cycle. Cycle detection is another interesting problem in graph theory that is left as an exercise to the reader.

See also

- map is used ahead of concat by `mapcat` to apply transformations.

- concat concatenates two or more sequential collections together.
- cat is the transducer version of concat.
- r/mapcat is the reducers version of `mapcat`.

Performance considerations and implementation details

=> `O(1)` **function generation**

=> `O(n)` **generated function**

`mapcat` is implemented on top concat and map for transformations and designed to be lazy. The performance profile is similar to other sequence processing functions, consuming their input in linear time (when fully evaluated).

`mapcat` argument processing is based on apply which is used to spread arguments for concat. For this reason `mapcat` is not completely lazy, always consuming the first four arguments:

```
(def a (mapcat range (map #(do (print ".") %) (i
;; ....
```

❶ `mapcat` is used here to concatenate different ranges of numbers. We can see 4 dots printed even if there is no consumer of the generated sequence.

If `mapcat` initial eagerness is a problem, the following `mapcat*` alternative removes the unwanted evaluation:

```
(defn mapcat* [f & colls]
  (letfn [(step [colls]
            (lazy-seq
              (when-first [c colls]
                (concat c (step (rest colls))))))]
    (step (apply map f colls)))) ; ❶

(def a (mapcat* range (map #(do (print ".") %) (
```

❶We can lazily apply the transformation to all collections with map. The laziness problem affects the second `apply` call that is on top of this first one in the core implementation.

❷Using the same example, no dots are printed this time, sign that `mapcat*` is fully lazy.

`mapcat` has also a transducer version that we can compare for sequential generation or reduction. We expect `mapcat` transducer to have an advantage over normal `mapcat`, as the

transducer version eliminates the need of the intermediate sequence generate internally by map. The following benchmark measures the different ways of producing output with `mapcat`:

```
(require '[criterium.core :refer [bench]]) ; ❶

(let [xs (range 1000)] (bench (last (mapcat rang

(let [xs (range 1000)] (bench (last (sequence (m
(let [xs (range 1000)] (bench (last (eduction (m

(let [xs (range 1000)] (bench (reduce + (mapcat
(let [xs (range 1000)] (bench (transduce (mapcat
```

❶As usual throughout the book, we are making use of the Criterium benchmarking library.

❷This is the basic `mapcat` generating a lazy sequence without using transducers. We take just the average time elapsed from the otherwise verbose Criterium output.

❸sequence and eduction are benchmarked next. They are considerably slower than basic `mapcat`.

❹Finally we can see a comparison between reduce and transduce using `mapcat`.

The transduce version is marginally faster than the reduce version, but sequence or eduction perform 3 times slower than plain `mapcat`. This can be explained with the additional complexity that sequence implementation has to deal with to enable transducers. The problem was discussed in the performance section of sequence in more detail. If performance are important and the transducer chain is not particularly complicated, the best option for sequential access is to use basic `mapcat`.

Alternatively, if laziness is not an issue, `mapcat` transducer can generate a vector using into with very good performance:

```
(let [xs (range 1000)] (bench (into [] (mapcat r
```

❶Creating a vector with a transducer chain using `mapcat` is faster than lazy sequential `mapcat`.

[170] Typical application of topological sorting is the ordering of the Java classpath so that classes are loaded only when all their dependencies are satisfied. The Wikipedia entry has more examples: en.wikipedia.org/wiki/Topological_sorting

10.5 INTERPOSE AND INTERLEAVE

`function` `since 1.0`

Listing 10.5. → *Sequence merging, Sequence processing, Transducers*

```
(interpose
  ([sep])
  ([sep coll]))

(interleave
  ([])
  ([c1])
  ([c1 c2])
  ([c1 c2 & colls]))
```

`interpose` and `interleave` produce a sequence by alternating elements coming from different inputs. `interpose` uses the first argument as interposing element for items in "coll":

```
(interpose :orange [:green :red :green :red]) ; (
;; (:green :orange :red :orange :green :orange :

(sequence (interpose :orange) [:green :red :greer
;; (:green :orange :red :orange :green :orange :
```

❶ `interpose` alternates the keyword ":orange" to each element in a vector. Note that there is no keyword ":orange" as the last element in the output sequence.

❷ `interpose` also has a transducer version that is showcased here.

`interleave` takes `interpose` a step further and offers a way to define the source of elements to alternate:

```
(interleave [:green :red :blue] [:yellow :magent
;; (:green :yellow :red :magenta :blue :cyan)
```

❶ `interleave` takes two or more sequential inputs to produce a lazy sequence of the alternating elements in them.

`interleave` stops alternating elements after reaching the shortest argument.

Contract

input

- "sep" is any expression or constant value including `nil`. It is required argument for `interpose`.

- "c1", "c2", "coll" or "colls" are collection arguments supporting the sequential interface (see sequence contract) or `nil`.

output

- Without a collection, `interpose` returns a transducer that alternates "sep" in the reducing step.
- With no arguments or if any argument is `nil`, `interleave` returns the empty list.
- With a single collection, `interleave` transforms the input into a lazy-seq without altering its content.
- With 2 or more "colls", `interleave` alternates items from each collection into a new output sequence stopping at the shortest input.

Examples

`interpose` and `interleave` are flexible functions for general sequential processing. `interpose`, for example, works well with string concatenation:

```
(def grocery ["apple" "banana" "mango" "other fr

(apply str (interpose ", " grocery)) ; ❶
;; "apple, banana, mango, other fruits"
```

```
(transduce (interpose ", ") str grocery) ; ❷
;; "apple, banana, mango, other fruits"
```

❶ `interpose` can merge words together using a separator. No separator appears at the end of the output sequence by default, which is a nice feature in this scenario (we don't want a trailing "," to appear at the end of the sentence).

❷The same operation in action in a transducer context. See the performance section for a comparison between the two forms.

There are numerous examples in the book that can be used as a starting point to observe how `interpose` or `interleave` are used in practice:

- random-sample features `interleave` laziness to alternate two infinite sequences of "head" and "tail" results for a coin toss simulation.
- rand-nth contains an example of `interpose` to generate a sentence from a list of words.
- partition contains an example of `interpose` used to format a SQL query for submission.
- transduce shows how we could create an `interleave` transducer.

The following example shows multiple passes of `interleave` to rotate the members of a single team first, then each team next in taking shifts for some kind of activity or competition:

```
(defn team [& names] ; ❶
  (apply interleave (map repeat names)))

(defn shifts [& teams] ; ❷
  (apply interleave teams))

(def a-team (team :john :rob :jessica))
(def b-team (team :arthur :giles))
(def c-team (team :paul :eva :donald :jake))

(take 10 (shifts a-team b-team)) ; ❸

;; (:john :arthur :rob :giles :jessica :arthur :
```

❶ We need a way to define a `team` taking a list of names. We want the list to repeat so we can cross the content with other teams. Each name is first repeated indefinitely, then interleaved with the others in the order they enter the function.

❷ Second we need to `interleave` members of each team. `interleave` would stop at the shortest of the team, but we made sure each team is instead a sequence of its members repeating ad infinitum. The output of `shifts` is again an infinite sequence of interleaving team members.

❸ After creating a few teams for testing, we can see that the generated sequence is an average of all the team members across all teams. Smaller teams are appearing more often and needs to do more work.

Inverse of `interleave`

`interleave` merges two sequences together by alternating their elements into a new sequence. We'd like to figure out a function to invert the process and produce the opposite effect.

We need at least one change though: `interleave` takes a variable number of collections as input, but we cannot have a "variable number of outputs" from a function in Clojure. The consequence is that our function is going to return a sequence of the originally interleaved sequences. Second, once we receive an interleaved input, we have lost the information about how many input sequences were interleaved in the first place. We need that number to be part of the input. Here's how we could approach the problem:

```
(defn untangle [n xs] ; ❶
  (letfn [(step [xs]  ; ❷
            (lazy-seq
              (cons
                (take-nth n xs) ; ❸
                (step (rest xs))))))] ; ❹
    (take n (step xs))))

(untangle 2 (interleave (range 3) (repeat 3 ".")))
;; ((0 1 2) ("." "." ".")) ; ❺
```

❶ `untangle` takes the output "xs" of a typical `interleave` call and the number "n" of original inputs.

❷The function uses an internal `step` that builds an infinite lazy sequence.

❸ `take-nth` is a good option for this problem, because it selects alternating elements from an input sequence. We can start by selecting the first series of alternating items at index (0,2,4..) to cons into the results. That's the first sequence that was interleaved.

❹We now shift one forward using `take-nth` again the next iteration.

❺Calling `untangle` with 2 interleaved sequences extract back the untangled sequences.

Another aspect to consider is laziness. `interleave` handles laziness accepting infinite sequences as input and creating a fully lazy output to consume. If the interleaved sequences from the input "xs" are infinite, we need to be careful to not realize them when they are returned:

```
(def infinite   ; ❶
  (interleave
    (iterate inc 1)
```

```
      (iterate dec 0)
      (iterate inc 1/2)))

(def untangled (untangle 3 infinite)) ; ❷

(take 10 (first untangled)) ; ❸
;; (1 2 3 4 5 6 7 8 9 10)

(take 10 (second untangled))
;; (0 -1 -2 -3 -4 -5 -6 -7 -8 -9)

(take 10 (last untangled))
;; (1/2 3/2 5/2 7/2 9/2 11/2 13/2 15/2 17/2 19/2
```

❶ `infinite` contains the interleaving of 3 infinite sequences.

❷ `untangled` contains the lazy sequence that untangles the interleaved sequences.

❸We can make access to the original sequences at the first , second and last place. They are infinite sequences, so we need to be careful to access a specific number of elements from each of them.

See also

- concat concatenates collections without interleaving of their elements.

- clojure.string/join performs string merging with an optional separator. Consider using `clojure.string/join` if you are not interested in laziness and only targeting formatting of strings.

Performance considerations and implementation details

=> `O(n)` **time, worst case**

=> `O(n)` **space, worst case**

`interpose` and `interleave` produce lazy sequences. Assuming they are completely evaluated on some input of length "n", then the worst case scenario is a linear dependency in both time and space. Be careful about holding onto the head of the sequence:

```
(let [s (interleave (range 1e7) (range 1e7))]
  (- (first s) (last s))) ; ❶
;; -9999999

(let [s (interleave (range 1e7) (range 1e7))]
  (- (last s) (first s))) ; ❷
;; likely OOM
```

❶ Access to the end of the sequence is the last operation performed. All elements cached so far can be garbage collected because nothing is holding an explicit reference to them.

❷ The end of the sequence is reached first thing, but the head of the sequence needs to stay around for first to access the first element after that. Depending on JVM settings, this can lead to massive work of the garbage collector that ends in out of memory error.

`interpose` is also available as a transducer. If laziness is important (no predictable plan to consume the entire output) and the sequence does not require much additional processing except for interposing elements, then basic `interpose` is probably the best choice:

```
(require '[clojure.string :as st])
(require '[criterium.core :refer [quick-bench]])

(def large-text "https://tinyurl.com/wandpeace")
(def lines (st/split-lines (slurp large-text)))

(quick-bench (last (eduction (interpose "|") lin
;; Execution time mean : 13.429075 ms
```

```
(quick-bench (last (interpose "|" lines))) ; ❸
;; Execution time mean : 3.717828 ms
```

❶For this benchmark we are using a large public domain book which is split into lines (thus eliminating the new line termination).

❷ `interpose` transducer runs as eduction and produce a non-cached sequence as result.

❸Basic `interpose` is faster in this case.

As soon as we require additional processing, the transducer version performs better than sequential:

```
(def xform ; ❶
  (comp
    (mapcat #(st/split % #"\s+"))
    (map st/upper-case)
    (remove #(re-find #"\d+" %))
    (interpose "|")))

(defn plainform [xs] ; ❷
  (->> xs
    (mapcat #(st/split % #"\s+"))
    (map st/upper-case)
    (remove #(re-find #"\d+" %))
    (interpose "|")))
```

```
(quick-bench (last (eduction xform lines)))
;; Execution time mean : 296.121150 ms

(quick-bench (last (plainform lines)))
;;  Execution time mean : 326.535851 ms
```

❶ `xform` performs additional processing on each line of the large text. After splitting lines into words, words are changed into upper-case, numbers are removed and finally they are separated with the pipe symbol.

❷ `plainform` is the nested sequential rendition of the same transducers chain.

The advantage, even in case of multiple processing step, is marginal, suggesting that there are few cases where `interpose` transducer is a better choice than basic `interpose` to produce a new sequential output.

Results are different if we give up laziness and reduce results, for example to generate a single string:

```
(quick-bench
  (str
    (reduce ; ❶
      #(.append ^StringBuilder %1 %2)
```

```
       (StringBuilder.)
       (interpose "|" lines))))
;; Execution time mean : 14.763760 ms

(quick-bench
  (transduce ; ❷
    (interpose "|")
    (completing #(.append ^StringBuilder %1 %2) 
    (StringBuilder.)
    lines))
;; Execution time mean : 9.631605 ms

(quick-bench (st/join "|" lines)) ; ❸
;; Execution time mean : 9.021710 ms
```

❶We use a mutable `StringBuilder` to create the string incrementally and avoid the creation of many intermediate strings. `interpose` is part of an initial sequential transformation. The sequence is the used as the input for reduce. The reducing function appends the string accumulated so far (including pipe separators) to a `StringBuilder` instance which is initially empty. We need to remember the type hints because the `StringBuilder` instance is passed as generic object into the reducing function and type information would be otherwise lost.

❷The same operation is implemented as `transduce`. Apart from using `interpose` transducer and a couple of small differences, the principle is the same.

❸A further comparison with clojure.string/join shows similar performance. `clojure.string/join` is definitely simpler if the only reason to process the input is to join into a final string.

We can see a speed improvement by using transduce instead of reduce (the equivalent operation for standard `interpose`). To summarize: if the main goal of using `interpose` is to completely evaluate the sequential output (giving up laziness) it's worth investigating the option offered by `interpose` as transducer. If laziness is still important, `interpose` transducer

can offer some advantage only when there are other sequential transformations in the same transducer chain.

10.6 PARTITION, PARTITION-ALL AND PARTITION-BY

`function` `since 1.0-1.2`

Listing 10.6. → *Lazy processing, Sequence splitting, Subsequence*

```
(partition
  ([n coll])
  ([n step coll])
  ([n step pad coll]))

(partition-all
  ([n])
  ([n coll])
  ([n step coll]))

(partition-by
  ([f])
  ([f coll]))
```

The three partitioning functions in this paragraph create subsequences (a lazy sequence containing other sequences)

from an input collection. `partition` and `partition-all` use a counter to decide when to split into the next subsequence:

```
(partition 3 (range 10))          ; ❶
;; ((0 1 2) (3 4 5) (6 7 8))

(partition-all 3 (range 10))      ; ❷
;; ((0 1 2) (3 4 5) (6 7 8) (9))
```

❶ `partition` is used to create subsequences of exactly 3 elements each.
❷ `partition-all` is more flexible and allows for the last subsequence to contain a different number of items.

The simple example already shows the main difference between `partition` and `partition-all`, with the former being strict about producing subsequences with the requested number of items and the latter allowing any "reminder" to end up in a last (potentially smaller) subsequence.

Instead of breaking the input by count, `partition-by` uses a function to decide where to split a new subsequence:

```
(partition-by count (map str [12 11 8 2 100 102
;; (("12" "11") ("8" "2") ("100" "102" "105") ("
```

❶A new subsequence is created every time the number of digits in the list of numbers changes.

There are a few differences between these functions that make them suitable for different problems. We are going to see what they are and how to use them in the contract and example sections.

Contract

INPUT

- "coll" is the input collection to partition. "coll" is mandatory for `partition` but not for `partition-by` or `partition-all`: they return a transducer in this case. "coll" must be compatible with seq to generate a sequence in case "coll" is not a sequence already.

- "n" establishes the max number of items in each partition. It's mandatory for `partition` or `partition-all` which are based on that, but not for `partition-by` which instead uses an user provided function to decide the split point. It can be negative (in which case an empty list is returned), 0 (which returns a potentially infinite sequence of empty lists) or positive (which is the most common case).
- "step" is only used by `partition`. It determines the distance (in terms of how many items apart) at which each subsequence should start. It works similarly to an offset, potentially repeating elements in multiple subsequences.
- "padding" is another collection only supported by `partition`. When there are remaining items from the input collection that cannot fit in the partitioning, `partition` can use the padding collection to fill the gaps and return the remaining items. It can be empty of `nil` and needs to be supported by seq.
- "f" is the mandatory argument for `partition-by`. "f" is a function of 1 argument returning any type. The returned values from "f" are compared and a new subsequence is created each time the value changes (as per equality contract).

NOTABLE EXCEPTIONS

Although not properly exceptions, it's possible to incur in infinite recursion:

```
(partition 3 0 (range 10))  ; ❶
;; WARNING infinite sequence of ((0 1 2) (0 1 2)
```

❶We are asking partition to return subsequences of 3 items each with a zero offset always restarting at "0" to create the next partition. Use take to limit the amount of results you need.

OUTPUT

- returns: a lazy sequence of partitions of the input collection "coll" as dictated by the given input parameters, or `nil` when "coll" is `nil`.

Examples

Let's start with a few examples to illustrates padding and offset:

```
(partition 3 3 (range 10)) ; ❶
;; ((0 1 2) (3 4 5) (6 7 8))
```

```
(partition 3 2 (range 10)) ; ❷
;; ((0 1 2) (2 3 4) (4 5 6) (6 7 8))
```

❶The default "step" is the same as the partition size "n": the next subsequence should start 3 elements apart from the beginning of the previous one. The default step can be omitted without altering the results.

❷We decreased the "step" and we can see that each next subsequence is now starting 2 elements apart from the beginning of the previous, even if this implies repeating the same item in different subsequences.

```
(partition 3 3 [:a :b :c] (range 10)) ; ❶
;; ((0 1 2) (3 4 5) (6 7 8) (9 :a :b))
```

❶The last subsequence, the one with the number "9" in it, is now appearing thanks to the padding collection that was also provided. The padding collection `[:a :b :c]` provides additional padding elements to compensate for the missing 2 elements in the last partition.

```
(partition-all 3 (range 10)) ; ❶
;; ((0 1 2) (3 4 5) (6 7 8) (9))
```

```
  (partition-all 3 2 (range 10)) ; ❷
  ;; ((0 1 2) (2 3 4) (4 5 6) (6 7 8) (8 9))
```

❶If we don't care about uneven partitions, `partition-all` achieve similar results to `partition` without padding.
❷ `partition-all` also allows an offset specification that works similarly to `partition` with any items left added to the last partition.

`partition-all` is often involved producing batches of requests. Batching is a technique that allows a group of operations to be seen as a single "execution unit". If an application needs to perform a large quantity of requests and each request is a costly operation (like a database call or an HTTP request), `partition-all` could be used to segment the requests and perform the operation on each group instead. Each group could also be assigned to a different thread for parallel execution, for example with pmap:

```
  (def records (map #(-> {:id % :data (str %)}) (r

  (defn log [query] (str (.substring query 0 70) "

  (defn insert-query [records]
    (let [->value (fn [{:keys [id data]}] (format
```

```
          rows (apply str (interpose "," (map ->va
     (log
       (str "INSERT INTO records (id, data) VALUE
            " ON DUPLICATE KEY UPDATE"))))

(println (pmap insert-query (partition-all 10 re
;; (INSERT INTO records (id, data) VALUES (0,0),
;;  INSERT INTO records (id, data) VALUES (10,10
;;  ;; [...]
;;  INSERT INTO records (id, data) VALUES (990,9
```

❶Records are a list of maps containing an id and a data key.

❷Instead of using a SQL library to execute the query for real, in this example we just print the query.

❸The `insert-query` function has the goal of formatting a batch-style SQL update statement. It takes a list of records as input and creates a string suitable for execution on a SQL database. The records are formatted to form a comma separated list of values. We can see portions of the query once is printed to the standard output.

❹Here's where `partition-all` allows to segment the large collection into smaller batches. Although calling `insert-query` once with the entire collection is possible, it would fail the entire update even for a single record failure. `partition-all` creates more manegeable partitions that can be re-tried in isolation. This is a good use case for pmap, as the SQL update function is non-trivial and we can process partition in parallel (assuming the records have distinct keys).

`partition-by` is used idiomatically with identity to transform the element itself into the partitioning function. This has the effect of splitting the input collection for each contiguous repetition of elements. For example, let's assume a sensor network reads the temperature every minute. To know how fast the temperature is changing we could use `partition`:

```
(def temps [42 42 42 42 43 43 43 44 44 44 45
            45 46 48 45 44 42 42 42 42 41 41])

(map count (partition-by identity temps))     ; (
;; (4 3 3 2 1 1 1 1 4 2)                      ; (
```

❶ `partition-by` used with identity has some interesting application in highlighting churning rates, such as how much the temperature is changing by minute.

❷ We can read the results as: after 4 minutes the temperature changes, then after 3 minutes changes again, then after 3 minutes and so on, until we see a different reading every minute, which means a sharp gradient change.

`partition-by` was also used for a simple sentiment analysis while demonstrating the indentity function. `partition-by` was used there with the assumption that sentiment are sometimes expressed in text as repetition of letters. The reader is invited to review the example to see how `partition-by` was used.

Finally, both `partition-all` and `partition-by` can be used with transducers:

```
(eduction                ; ❶
  (map range)
  (partition-all 2)
  (range 6))
;; ([() (0)]             ; ❷
;;  [(0 1) (0 1 2)]
;;  [(0 1 2 3) (0 1 2 3 4)])
```

❶eduction produces a cache-less sequence and takes any number of transducers as input (without the need for comp).
❷Note how in the case of transducers, the partition produced by `partition-all` or `partition-by` is not a lazy sequence but a vector.

A `partition-with` for other partition strategies

`partition` and `partition-all` offers a partition strategy based on the number of element ending up in each partition. `partition-by` is more general, but the partitioning is limited by the change of results of the partitioning function "f". Some class of problems would benefit from even a more general strategy that takes into account a previous and current element to decide when to split. How difficult it is to make the partitioning strategy even more general? The answer is: not that difficult. Let's have a look at how `partition-by` is implemented:

```
(defn partition-by [f coll]     ; ❶
 (lazy-seq
  (when-let [s (seq coll)]
    (let [fst (first s)
          fv (f fst)
```

```
            run (cons fst (take-while #(= fv (f %)) (next s)))] ; ❷
        (cons run (partition-by f (seq (drop (count run) s))))))))
```

❶ `partition-by` as implemented in core, but stripped of the transducer implementation.

❷The "recipe" for partitioning lives here. "coll" is accumulated with take-while until an `(f item)` is found that is different from `(f first-item)`. At that point we take the accumulation and we invoke `partition-by` recursively with the remaining elements.

To change the partitioning strategy, we can change the take-while predicate. Here's for example a predicate "f" taking two arguments: the current item and the next item in the collection. Based on returning `true` or `false` we create a new partition:

```
(defn partition-with [f coll]     ; ❶
 (lazy-seq
  (when-let [s (seq coll)]
    (let [prev (first s)
          run (cons prev (take-while #(f prev %)
      (cons run (partition-with f (seq (drop (co
```

❶The function was renamed `partition-with`.

❷The change to implement the different strategy is quite small. All we need to do is to pass the function "f" the first element and partition the following ones until "f" returns `true`. As soon as "f" returns `false` we recur with the remaining items like before.

The new `partition-with` can be used in time series, where a list of events is sent through as list of maps containing a time marker:

```
(import '[java.time Duration Instant]
        '[java.time.format DateTimeFormatter]
        '[java.time.temporal TemporalAccessor])

(def events [{:t "2017-05-04T13:08:57Z" :msg "ms
             {:t "2017-05-04T13:09:52Z" :msg "ms
             {:t "2017-05-04T13:11:03Z" :msg "ms
             {:t "2017-05-04T23:13:10Z" :msg "ms
             {:t "2017-05-04T23:13:23Z" :msg "ms

(defn ->inst [{t :t}]     ; ❷
  (Instant/from
    (.parse (DateTimeFormatter/ISO_INSTANT) t)))

(defn burst? [t1 t2]      ; ❸
  (let [diff (.getSeconds (Duration/between (->i
```

```
      (<= (Math/abs diff) 120)))

  (partition-with burst? events) ; ❹

  ;; (({:t "2017-05-04T13:08:57Z", :msg "msg1"}
  ;;   {:t "2017-05-04T13:09:52Z", :msg "msg2"})
  ;;  ({:t "2017-05-04T13:11:03Z", :msg "msg3"})
  ;;  ({:t "2017-05-04T23:13:10Z", :msg "msg4"}
  ;;   {:t "2017-05-04T23:13:23Z", :msg "msg5"}))
```

❶The events are entering our example system as a "stream", a lazy sequence attached to some IO facility. For illustration purposes, they are just a vector with a few items.

❷The function name `→inst` prefixed with an arrow `→` indicates a transformation from one format (time stamp as a string) into another (time stamp as `java.time.Instant` object).

❸The `burst?` function is a predicate of two instants. It returns `false` if their different is more than 120 seconds, indicating that the two events are too far apart to be considered part of the same group.

❹We use `partition-with` as illustrated before, using the `burst?` function as the predicate.

The result from `partition-with` contains a partitioning of the input events so that two or more events are together if their time difference is below 2 minutes. If more than 2 minutes, then the event is considered part of the following group.

See also

- mapcat and `partition` are often found together in a processing pipeline, because while `partition` introduces a new nested level of partitions, `mapcat` removes the level returning to a flat sequence.
- pmap is a parallel map implementation. It is often associated with `partition-all` to perform parallel batch processing.
- split-at are similar partitioning functions that splits the input sequence into 2 parts only. Use `split-at` or `split-with` if you are looking for a single split point in the sequence.

Performance considerations and implementation details

=> `O(n)` **linear**

The `partition` functions are implemented on top of a relatively simple recursion that starts every time a new

partition is created. The number of iterations is linked to the number of input items producing a linear behavior.

`partition` functions are lazy, producing just enough of the iterations as requested by the caller:

```
(first (partition 3 (map #(do (println %) %) (ra
;; 0
;; 1
;; 2
;; (0 1 2) ; ❶
```

❶ `partition`, `partition-all` and `partition-by` are lazy functions. As expected, we can see that only the items necessary to form the first partition are realized.

Laziness with transducers works differently and in general is more eager. Even when a single partition is requested, this transducer `partition-all` realizes n*(32+1) items. This is because how sequence works, always requesting at least 32 items from the result of the transducer, which translates in 32 partitions of n elements each, even when only one is requested:

```
(first
  (sequence
```

```
     (comp (map #(do (print % ",") %))   ; ❶
           (partition-all 100))          ; ❷
     (range)))
;; 0, 1, 2, ....., 3299,                ; ❸
;; [0 1 2 ... 98 99]
```

❶We use a transducer which is just printing a message and passing the element back as is.
❷ `partition-all` is used here as transducer.
❸The printing transducer shows 3299 items requested, which corresponds to the `(* 100 (+ 1 32))` 3300th element in the output.

As another example, compare the following plain `partition-by` and its transducer version. We create a partition that contains just the number "0" and another with all other positive numbers. We can request the first item from sequential `partition-by` but we can't do the same with the transducer version:

```
(partition-by pos? (range 10)) ; ❶
;; ((0) (1 2 3 4 5 6 7 8 9))

(first (partition-by pos? (range))) ; ❷
;; (0)
```

```
  (first (sequence (partition-by pos?) (range))) ;
  ;; WARNING: hangs
```

❶This example shows how partitioning numbers by `pos?` looks like: "0" is not positive and ends up alone in the first partition, while everything else goes into the partition of positive numbers.

❷We now remove the restriction on the size of the range of numbers, creating an infinite sequence. `partition-by` is strictly lazy and just evaluates enough to give the result.

❸The transducer version is eager internally and tries to realize the second (infinite) partition as well.

As you can see from the example, the implementation of `partition-all` becomes eager in case of transducers.

10.7 FLATTEN

`function` `since 1.2`

Listing 10.7. → *Nested Sequences, Collection Processing, Unwrapping*

```
(flatten [x])
```

`flatten` is a function that takes an arbitrarily nested data structure and returns the sequence formed by concatenating items from supported collections at any level of nesting:

```
(flatten [[1 2 [2 3] '(:x :y [nil []])]]) ; ❶
;; (1 2 2 3 :x :y nil)
```

❶Simple usage of `flatten` to remove "wrapped" elements into a flat sequence.

`flatten` only operates on collections for which `sequential?` returns `true`. These are predominantly lists and vectors (and others types described in the contract section).

Warning

`(flatten coll)` is not a replacement for `(mapcat identity coll)`. While this might seem obvious from the description of the function, liberal use of `flatten` with the assumption that "coll" has only one level of nesting can be dangerous, because it will also remove any other nesting. It's better to express intent explicitly with `mapcat` when the goal is just to concatenate the first level of nesting.

Contract

INPUT

- "x" is the only mandatory argument. It can be any type including `nil`. If all nested collections in "x" are compatible, `flatten` returns a linear sequence of their content. Supported types are shown in the following table:

Table 10.3. Summary of `flatten` supported types (`c.l.`

means `clojure.core` namespace)

type	supported?
`c.l.PersistentList`	yes
`c.l.PersistentVector`	yes
`c.l.PersistentVector$TransientVector`	no
`c.l.APersistentVector$SubVector`	yes
`clojure.core.Vec`	yes
`c.l.PersistentHashSet`	no
`c.l.PersistentArrayMap`	no
`c.l.PersistentTreeSet`	no
`c.l.PersistentTreeMap`	no
`c.l.LongRange`	yes

`c.l.LazySeq`	yes
`[I`	no
`c.l.PersistentQueue`	yes
`java.util.ArrayList`	no
`java.lang.String`	no
`c.l.Cons`	yes

NOTABLE EXCEPTIONS

None.

OUTPUT

- returns: a lazy sequence that concatenates all supported collections at any level of nesting.

Examples

Let's start by showing the behavior of `flatten` in the presence of unsupported types:

```
(import 'java.util.ArraList)

(flatten [[{:a 1} #{2 3} (doto (ArrayList.) (.ad

;; ({:a 1} #{3 2} [1 2])
```

❶Note that maps, sets or `ArrayList` are not supported and `flatten` does not traverse their content.

When the collection types are not supported, `flatten` skips their traversal and just passes them on as they are.

A typical nested data structure is the one resulting from macro-expansion. In the next example we are going to collect all Clojure (core) functions are used after expanding a macro. We know that core functions are generally prefixed with the `clojure.core` namespace. It is worth remembering that Clojure code is data (a famous feature of Lisp languages) and data types are just nested lists and symbols. We could use `flatten` to surface all the symbols from their nested position:

```
(require '[clojure.walk :as w])

(defn core-fns [form]
  (->> (w/macroexpand-all form)   ; ❶
       flatten                    ; ❷
       (map str)
       (map #(re-find #"clojure\.core/(.*)" %))
       (keep last)
       distinct))                 ; ❸

(core-fns                         ; ❹
  '(for [[head & others] coll
         :while #(< i %)
         :let [a (mod i 2)]]
    (when (zero? a)
      (doseq [item others]
        (print item)))))

;; ("seq" "chunked-seq?" "chunk-first" "int"
;;  "count" "chunk-buffer" "<" "first" "next"
;;  "chunk-append" "unchecked-inc" "chunk-rest"
;;  "chunk-cons" "chunk" "cons" "rest")
```

❶macroexpand-all is a function that given a form, invokes macroexpansion recursively until there are no more macros to expand. It then returns the expanded form. This is usually quite bigger than the original form depending on the usage and complexity of macros in it.
❷ `flatten` called on the expanded form returns all symbols after unwrapping all level of nesting.
❸After transforming symbols into strings and filtering `clojure.core` ones, we need to remove repeating function names.
❹We use a for loop as input.

The result shows what functions are used by an example for loop, although special forms like `let*`, `if`, `loop` or `recur` are not visible, because they lack qualification with the `clojure.core` namespace.

See also

- mapcat applies a function to each element in a collection. Assuming the transformation produces a layer of sequences, `mapcat` concatenates their content into a linear sequence. When used with identity, `mapcat` removes a single layer of nested collections.

- transducer cat and reducer cat apply a similar concept to `mapcat` to remove a single nesting from their input collection.

Performance considerations and implementation details

=> `O(n)` **linear**

`flatten` needs to traverse every collection and potentially unwrap their content. It follows that the number of steps to perform is linear in the amount of elements at any level in the input.

From the implementation perspective, `flatten` is built on top of tree-seq. It is then a matter of distinguishing supported collection types and only process their content.

`flatten` operates lazily and will only pull enough of input sequence to output the requested result:

```
(->>
  (range)   ; ❶
  (map range)
  (map-indexed vector)
  flatten
```

```
    (take 10))
  ;; (0 1 0 2 0 1 3 0 1 2)
```

❶The input source for this processing chain is the infinite sequence of integers produced by the range invocation. It can be used safely with flatten.

10.8 DISTINCT, DEDUPE AND DISTINCT?

`function`

`since 1.0 (distinct, distinct?)`

`since 1.7 (dedupe)`

Listing 10.8. → *Duplicates Detection, Collection Processing, Transducers*

```
  (distinct
    ([])
    ([coll]))

  (dedupe
    ([])
    ([coll]))

  (distinct?
```

```
    ([x])
    ([x y])
    ([x y & more]))
```

`distinct` and `dedupe` remove duplicates from an input collection, while `distinct?` just reports about their presence returning `true` or `false`. `distinct` and `distinct?` detects all duplicates in a collection (or list of arguments) while `dedupe` only removes contiguous repetitions:

```
(distinct [1 2 1 1 3 2 4 1])  ; ❶
;; (1 2 3 4)

(distinct? 1 2 3 2 4 1)       ; ❷
;; false

(dedupe [1 2 1 1 3 2 4 1])    ; ❸
;; (1 2 1 3 2 4 1)
```

❶ `distinct` removes duplicate items throughout the collection, independently from their relative position.

❷ `distinct?` detects duplicate arguments returning `true` or `false`.

❸ `dedupe` only removes contiguous repetitions of the same item, retaining those that are at least one element apart.

`distinct`, `dedupe` and `distinct?` use Clojure extended equality semantic, accepting both scalars (numbers, keywords, symbols and so on) as well as collections. Clojure equality uses compatibility groups to decide when two items are the same (please refer to = for an exhaustive explanation).

`distinct`, `dedupe` and `distinct?` are used quite frequently in Clojure, reflecting the presence of many problems in computer science dealing with duplicates (for example data compression [171].

Contract

There are small contract differences between the functions illustrated in this section. `distinct` does not accept sets or maps as argument (including the related Java types, `HashMap` and `HashSet`), while `dedupe` accepts them. In general, there little sense in feeding `maps` or `sets` to `distinct` or `dedupe` as they cannot have duplicates.

INPUT

`distinct` and `dedupe`:

- "coll" is optional and can be `nil`. When "coll" is not present, both functions return the related transducer. transients are not accepted.
- `distinct` does not allow sets or maps.
- `dedupe` works on all collection types excluding transients.

`distinct?`:

- "x", "y" and "more" is the typical signature of variadic functions like `distinct?`. "x", "y" and "more" can be any kind of Clojure forms, literal or `nil`. At least one argument is required.

NOTABLE EXCEPTIONS

- `distinct` throws `UnsupportedOperationException` when the input is a Set, map, `java.util.HashMap` or `java.util.HashSet`.

- `ArityException` when `distinct?` is invoked without arguments. There is a specific case which is less easy to detect when `distinct?` is used with apply: `(apply distinct? [])` produces the exception on empty collection, requiring a check like the following: `(and (seq []) (apply distinct? []))`.

OUTPUT

`distinct` and `dedupe`:

- A lazy sequence of the non repeated items in "coll". `dedupe` allows duplicates which are at least one item apart.
- An empty list when "coll" is `nil`.

`distinct?`:

- `true` when there are no duplicates independently from their position in the list of arguments.
- `false` where there is at least one duplicate item.

Examples

A voting system allows for a maximum of 5 votes to 3 distinct candidates. Users of the system might double vote for a candidate (either on purpose or by mistake) and we want to remove any additional vote for a candidate if it's coming from the same user:

```
(def votes [                    ; ❶
  {:id 14637 :vote 3 :secs 5}
  {:id 39212 :vote 4 :secs 9}
  {:id 39212 :vote 4 :secs 9}
  {:id 14637 :vote 2 :secs 43}
  {:id 39212 :vote 4 :secs 121}
  {:id 39212 :vote 4 :secs 121}
  {:id 45678 :vote 1 :secs 19}])

(->> votes          ; ❷
  (group-by :id)
  (reduce-kv
    (fn [m user votes]
      (assoc m user (distinct (map :vote votes))))
    {}))

;; {14637 (3 2), 39212 (4), 45678 (1)}
```

❶ Votes enter the system as a list of all the votes at the end of a competition. Here we show a little sample of a much larger list. Each ":id" is an user, followed by a ":vote" for a candidate as a number and finally the number of seconds elapsed since the beginning of the competition.

❷ After grouping votes by users using group-by we can see how many votes each candidate received from each user. We can process the map with reduce-kv and make sure that each value does not contain duplicates for a specific candidate using `distinct`.

The voting system is composed by a software service and an hardware button to express preferences. After analyzing the data, the voting system exposes a strange behavior. It is discovered that the hardware generates "burst of clicks" each time the user presses a button to vote, when it should only send one choice. We want to get rid of the unwanted clicks as early as possible, as they present a problem for the scalability of the system. Luckily for us, the problematic clicks happen a few milliseconds apart so we can clearly tell which votes are to be discarded. We can do this with `dedupe` ahead of the `group-by`:

```
(->> votes          ; ❶
  dedupe
```

```
    (group-by :id)
    (reduce-kv
      (fn [m user votes]
        (assoc m user (distinct (map :vote votes))))
      {}))

  ;; {14637 (3 2), 39212 (4), 45678 (1)}  ; ❷
```

❶We execute the same operation as before, but we get rid of contiguous duplicates first with `dedupe`.

❷The results are the same as before, as expected.

`distinct?` can be used as a predicate to find collections of distinct items. Clojure, for example, uses `distinct?` internally to find a suitable combination of hashes when implementing case. `case` needs to adapt test expressions so they can fit the switch-table of the relative JVM instruction, which requires distinct integer keys. When `case` test constants are generic objects, `case` calculates the hash of each object and then attempts several bit/mask combination to find a transformation that produces distinct keys:

```
(def max-mask-bits 13)              ; ❶

(defn- shift-mask [shift mask hash] ; ❷
```

```
  (-> hash
    (bit-shift-right shift)
    (bit-and mask)))

(defn- maybe-min-hash [hashes]      ; ❸
  (let [mask-bits (range 1 (inc max-mask-bits))
        shift-bits (range 0 31)
        masks (map #(dec (bit-shift-left 1 %)) ma
        shift-masks (for [mask masks shift shift
                          [shift mask])]
    (first
      (filter
        (fn [[s m]]
          (apply distinct?          ; ❹
            (map #(shift-mask s m %) hashes)))
        shift-masks))))

(maybe-min-hash
  (map (memfn hashCode) [:a :b :c :d]))   ; ❺

;; [1 3]

;; (case op :a "a" :b "b" :c "c" :d "d")  ; ❻

(map #(shift-mask 1 3 %)
  (map (memfn hashCode) [:a :b :c :d]))
;; (0 2 1 3)
```

❶ This specific `max-mask-bits` is because of the 32 bits size allowed for the JVM tableswitch instruction [172].

❷ `shift-mask` applies a shift of bits and the specific mask to a hash number. This is ultimately the transformation that we want to apply to each test case expression, but only if it's not producing duplicates.

❸ `maybe-min-hash` generates all possible permutations of a bit-shift and a bit-mask using a for loop. It then applies them to the given hashes resulting in a collection from which we only want the first combination that is not producing duplicates.

❹ `apply distinct?` takes the result of the shift-mask transformation and verify that is producing distinct results for each hashed test expressions of the `case` statement. We don't want the actual collection of distinct values, so we use the predicate to filter the shift-mask combination instead.

❺ We can see how to use `maybe-min-hash` on `[:a :b :c :d]`. The result "[1 3]" should be interpreted as: `bit-shift-right` each hash of "1" and then `bit-and` each hash with mask "3". This generates distinct keys to be used in the generated JVM instruction.

❻ This shows how the `case` expression for this example would look like. The 4 keywords used as text expression can be

encoded as the integer keys "0,2,1,3" with the shift-mask transformation "[1 3]".

As transducers, `distinct` and `dedupe` can be used invoking their zero-arity version:

```
(sequence
  (comp
    (map range)
    cat
    (distinct))          ; ❶
  (range 10))
;; (0 1 2 3 4 5 6 7 8)

(sequence
  (dedupe)               ; ❷
  [1 1 1 2 1 1 1 3 1 1])
;; (1 2 1 3 1)
```

❶ `distinct` as a transducer requires wrapping inside parenthesis, unlike cat which appears directly above.

❷ `dedupe` transducer removes contiguous duplicates like the sequential version.

Duplicates, sorting and sets

There is a relationship between searching for duplicates, sorting and sets. For instance, you can achieve `distinct` through a combination of sort and `dedupe`:

```
(def duplicates [8 1 2 1 1 7 3 3])

(distinct duplicates)       ; ❶
;; (8 1 2 7 3)

(dedupe (sort duplicates))  ; ❷
;; (1 2 3 7 8)
```

❶ `distinct`, as implemented in the standard library, maintains ordering of the original collection while removing duplicates.

❷ `sort` has the effect of grouping duplicates together in a way that `dedupe` can completely remove.

Both `distinct` and `dedupe` with `sort` remove all duplicates, but they return the same list of number in different order. There is also a data structure with very similar properties: a hash sets, by forcing uniqueness of their elements by design, can be used to remove duplicates:

```
(set duplicates)            ; ❶
;; #{7 1 3 2 8}
```

❶Set can be used to create a Clojure Set directly from another collection, producing no duplicates.

We've seen many ways of removing duplicates in this section. Which one to use depends on several factor:

- Constraints on the ordering of the output: if ordering of the initial collection is important, then `distinct` is the primary choice.
- Presence of transformations on the input collection: then the best choice is to use the transducer version of `distinct` or `dedupe`.
- Need for checking the presence of an element while removing duplicates: transforming the input in a hash-set offers performing lookups and duplicate removals at the same time.

See also

- sort was mentioned a few times. `sort` does not remove duplicates, but put the input collection in a condition where they can be managed. `dedupe` can be used in conjunction with `sort` to obtain a form of ordered `distinct`.
- Set produces a Clojure hash-set from an input collection, automatically removing duplicates in the process.

Performance considerations and implementation details

=> `distinct`: `O(n)` **steps** `O(n)` **memory (worst case)**

=> `dedupe`: `O(n)` **steps** `O(1)` **memory**

=> `distinct?`: `O(n)` **steps** `O(n)` **memory, n number of args**

`distinct`, `dedupe` and `distinct?` are implemented using a similar pattern which consists of maintaining a view of the currently seen items as a Clojure Set. This represents a trade-off between the `O(n^2)` steps required to scan all items against each others (which doesn't consume memory) and the additional memory required to maintain a copy of each item in a set (a worst case scenario when there are no duplicates).

With exception of `disctinct?` (which is designed for a relatively low number of arguments), both `distinct` and `dedupe` are lazy (or semi-lazy) operations:

```
(first (distinct (map #(do (print % ",") %) (ran
;; 0 ,0

(first (dedupe (map #(do (print % ",") %) (range
;; 0 ,1 ,2 ,3 ,..,32,0
```

❶ `distinct` is the laziest, consuming exactly the required amount of items to satisfy the request.
❷ `dedupe` is semi-lazy, as it is implemented on top of its transducer version which makes use of sequence and sequence always consumes the first 32+1 items.

The transducer version of `distinct` outperforms the standard sequential implementation at the price of slightly reduced (or removed in case of transduce) laziness:

```
(require '[criterium.core :refer [quick-bench]])

(defn with-dupes [n]
  (shuffle
    (into []
      (apply concat (take n (repeat (range n))))

(let [c (with-dupes 1000)]
  (quick-bench (doall (distinct c)))
  (quick-bench (doall (dedupe c)))
  (quick-bench (doall (sequence (distinct) c)))
  (quick-bench (doall (sequence (dedupe) c))))

;; Execution time mean : 271.592546 ms
;; Execution time mean : 102.599305 ms
```

```
  ;; Execution time mean : 93.271275 ms
  ;; Execution time mean : 105.953654 ms
```

❶The Criterium library that we used through out the book is available on GitHub.

❷ `with-dupes` is an helper function to create a collection with of n*n elements with duplicates.

❸The benchmark compares both `distinct` and `dedupe` in their standard and transducers version. Some of the Criterium output is omitted for clarity.

The results of the simple benchmark shows that `distinct` transducer version outperforms the standard version with a significant margin, while `dedupe` performance is roughly the same. A more precise benchmark should take into account the amount of duplicates present in the original input and their ordering, as they both influence the final result, especially in the case of `dedupe`.

[171] See en.wikipedia.org/wiki/Data_deduplication

[172] there is some complexity related to the way the JVM implements a fast lookup switch, if you want to know more this is a good starting point stackoverflow.com/questions/10287700/difference-between-jvms-lookupswitch-and-tableswitch

10.9 TAKE-NTH

`function` `since 1.0`

Listing 10.9. → *Filtering, Sequential indexing, Transducers*

```
(take-nth
  ([n])
  ([n coll]))
```

`take-nth` creates a lazy sequence using a selected set of elements from another collection. It includes the first element by default, while the others are identified by repeatedly dropping the same amount of items:

```
(take-nth 3 [0 1 2 3 4 5 6 7 8 9]) ; ❶
;; (0 3 6 9)
```

❶ `take-nth` selects "0" as the first element. Then it skips 3 items to reach the number "3" which is added to the output. The process repeats until reaching the end of the input.

`take-nth` implements a form of filtering by index which is useful in many situations, usually in conjunction with other

functions dedicated to sequential processing. `take-nth` is also available as a (stateful) transducer:

```
(into [] (take-nth 2) (range 10)) ; ❶
;; [0 2 4 6 8]
```

❶ `take-nth` without the collection arguments returns a transducer with similar capabilities.

Contract

input

- "n" is the number of elements to drop after the first to reach the next element to include in the output. `take-nth` requires a positive number greater than zero. Decimal number are possible but are automatically rounded.
- "coll" can be any sequential collection and is optional argument.

notable exceptions

- `ArithmeticException` divide by zero error: only on the transducer version for n = 0 and non empty "coll", such as `(into [] (take-nth 0) [1 2 3])`.

- `NullPointerException` for the transducer version, when "n" is `nil` and "coll" contains at least one element.

output

The transducer and the sequential version differs in the treatment of some corner cases. In both cases, with "n" positive integer, `take-nth` returns the sequence generated by taking the first element from "coll", dropping "n" - 1 elements, then the element at "n", then dropping "n" - 1 next elements and so on.

`take-nth` sequential version has the following output for the cases listed below:

- "n" `nil`: returns empty sequence.
- "n" zero or negative: returns an infinite list of of the first item in "coll".
- "n" decimal number: "n" is rounded up to the next integer (e.g. `(take-nth 1.1 [1 2 3])` is equivalent to `(take-nth 2 [1 2 3])`)

`take-nth` transducer has instead the following behavior:

- "n" `nil` and non-empty "coll": `NullPointerException`.

- "n" zero: `ArithmeticException`. This has been flagged as potential bug.
- "n" negative integer: same as positive integer.
- "n" decimal number: "n" is doubled (e.g. `(into [] (take-nth 2.5) (range 10))` is equivalent to `(into [] (take-nth 5) (range 10))`.



Warning

some of the corner cases shown in the output contract for the transducer version has been raised as bugs at the following link: clojure.atlassian.net/browse/CLJ-1665.

Examples

`take-nth` is a natural solution for the problem of generating the multiples of a number:

```
(defn mult-n [n] ; ❶
  (rest (take-nth n (range))))

(take 10 (mult-n 11)) ; ❷
;; (11 22 33 44 55 66 77 88 99 110)
```

```
(take 10 (mult-n 42))
;; (42 84 126 168 210 252 294 336 378 420)
```

❶ `mult-n` is a function that given a number, generates an infinite sequence of the its multiples. We use rest to drop the initial zero from the list.

❷ We can see how to generate multiples for the number "11" and "42".

`take-nth` is also useful when handling a variable number of key-value pair arguments. Here's for example a function to create a sparse vector, a vector that has zeros at every index except for those indicated by the arguments:

```
(defn sparsev [& kv]
  (let [idx (take-nth 2 kv) ; ❶
        xs  (take-nth 2 (next kv)) ; ❷
        items (zipmap idx xs)] ; ❸
    (reduce
      #(conj %1 (items %2 0)) ; ❹
      []
      (range 0 (inc (apply max idx))))))) ; ❺

(sparsev 1 4 3 7 21 8)
;; [0 4 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 8]
```

❶The first `take-nth` invocation is to select the indexes.

❷After shifting one ahead with next we can select the values.

❸zipmap takes care of creating a map out of the two list. The map is used later when building the output vector. Note that we don't need to think about a possible different length between keys and values: a single value is ignored by default.

❹We can use reduce to build the vector incrementally. If there are performance constraint we could use a transient vector. transduce is also a viable option.

❺We create a range of all possible indexes, 0 to the max index that we need to fill with other something than zero.

Implementing `drop-nth`

`take-nth` almost naturally calls for `drop-nth` with similar features but inverted meaning: generate the lazy sequence of items which is left after we remove each "nth" element in the input sequence. A starting point could be to look at `take-nth` sources and change the relevant parts:

```
(defn drop-nth [n coll] ; ❶
  (lazy-seq
    (when-let [s (seq coll)]
      (concat (take (dec n) (rest s)) ; ❷
              (drop-nth n (drop n s)))))) ; ❸

(drop-nth 3 (range 10))
;; (1 2 4 5 7 8)
```

❶Apart from the name change, the general design of the function stays the same as `take-nth`.

❷Instead of cons results, we need to concat them, because we select the multiple elements between the nth gaps instead of the nth element. We need to skip the first item which is in nth position and then take up to `nth - 1` more items.

❸The next iteration uses the sequence after dropping everything up to the next nth element.

Alternatively, we could use rem to see which items corresponds to which index and implement `drop-nth2` on top of keep-indexed:

```
(defn drop-nth2 [n coll] ; ❶
  (keep-indexed ; ❷
    #(when-not (zero? (rem %1 n)) %2) ; ❸
    coll))

(drop-nth2 3 (range 10))
;; (1 2 4 5 7 8)
```

❶ `drop-nth2` implemented using a different approach.

❷keep-indexed is needed because we are going to produce a `nil` each time we process an item whose index is a multiple of "n".

❸Using rem we can identify items that sits "n" elements apart.

The second approach using keep-indexed also has a straightforward transducer version which can be obtained by

removing "coll":

```
(defn xdrop-nth [n] ; ❶
  (keep-indexed
    #(when-not (zero? (rem %1 n)) %2)))

(sequence (xdrop-nth 3) (range 10))
;; (1 2 4 5 7 8)
```

❶For the transducer version, we can use keep-indexed in its transducer version.

`drop-nth2` is slightly more concise, but which is faster? We can have a look with Criterium:

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (range 1e5)] (quick-bench (last (drop-n
;; 13.511636 ms

(let [xs (range 1e5)] (quick-bench (last (drop-n
;; 4.586312 ms
```

`drop-nth2` is both smaller and faster.

See also

- filter removes elements from a sequence using a predicate instead of using the distance between the elements.
- split-at splits a sequence into two parts at the requested index.
- partition also splits a sequence into subsequences of the requested size.

Performance considerations and implementation details

=> `O(n)` **steps, worst case**

=> `O(n)` **space, worst case**

`take-nth` has a typical sequential processing performance profile. The output produced by `take-nth` is lazy, so the amount of requested elements determines how much work the function does, up to a maximum of "n" steps if "n" is the length of the input.

Memory footprints for the fully evaluated output is also linear in the length "n" of the input sequence. If `take-nth` is used in a reducing scenario (where the output size is smaller than the input), memory footprints can be reduced not holding on to the head of the generated output (see for example map

performance section for an example of holding onto the head of a sequence).

`take-nth` is also available as a transducer. The transducer implementation tends to be faster:

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (range 1000000)] (quick-bench (last (ta
;; Execution time mean : 75.020203 ms

(let [xs (range 1000000)] (quick-bench (last (se
;; Execution time mean : 69.482801 ms
```

❶ `take-nth` basic version is benchmarked against the transducer version in the same scenario of generating another sequence. Note the use of last to fully evaluate the sequence.

❷ We can see that the transducer version is slightly faster. This is in part due to the different implementation used for the transducer version of `take-nth`.

We can do a similar comparison in a reducing context:

```
(let [xs (range 1000000)] (quick-bench (reduce +
;; Execution time mean : 70.808658 ms

(let [xs (range 1000000)] (quick-bench (transduc
;; Execution time mean : 45.558123 ms
```

❶We use reduce to sum up the multiples of 2 up to 1 million elements.

❷The transducer version is faster than the plain reduce version.

We can see that the reducing context produces favorable results for the transducer version.

10.10 SPLIT-AT AND SPLIT-WITH

`function` `since 1.0`

Listing 10.10. → *Sequential selection, Splitting*

```
(split-at [n coll])
(split-with [pred coll])
```

`split-at` and `split-with` split a sequential input by size or by using a predicate:

```
(split-at 8 (range 10)) ; ❶
;; [(0 1 2 3 4 5 6 7) (8 9)]

(split-with (complement zero?) [1 4 5 0 3 2 0 1
;; [(1 4 5) (0 3 2 0 1 1 0)]
```

❶ `split-at` is used with a number representing the index at which we want the split to occur. In this case for example we are indicating we want a split when we reach the 8th element in the sequence.

❷ `split-with` works with a predicate in the same way as take-while, splitting the input as soon as the predicate returns `false` on an item. We can see that the split happens on the first appearance of a zero.

Differently from partition or partition-all, `split-at` and `split-with` are designed to return only two partitions.

Contract

input

- "n" can be any integer equal or greater than zero. Negative numbers are possible, but their effect is the same as "n" set to zero. Decimal point numbers are also possible and rounded up to the nearest integer. If "n" is larger than "coll" size, it returns a vector pair with "coll" and the empty list. It is required argument for `split-at`.
- "pred" is a function of one argument returning logical `true` or `false`. It is required argument for `split-with`.
- "coll" is any sequential collection (as per seq contract) or `nil`.

notable exceptions

- `IllegalArgumentException` when "coll" is not sequential (as per seq contract).

output

- `split-at` returns a vector of two elements: the first is a lazy sequence of the first "n" items in "coll" and the seconds are the remaining `(- (count coll) n)` items.

- `split-with` returns a vector of two elements: the first is the lazy sequence of elements up to the first `false` evaluation of "pred" and the second contains all the remaining items.

Examples

`split-at` and `split-with` output vector is commonly destrctured as part of a let binding:

```
(let [[head others] (split-at 8 (range 10))] ; ❶
  (+ (last head) (last others)))
;; 16
```

❶The vector results from `split-at` can be easily destructured, so the two parts can be processed independently.

Both functions work lazily on their input:

```
(take 10 (last (split-at 10 (range)))) ; ❶
;; (10 11 12 13 14 15 16 17 18 19)
```

❶ `split-at` is invoked on an infinite range. The second part of the split is still an infinite sequence.

We could use a Set as the predicate with `split-with` to split a collection based on existence of an item. Since the split happens on the first `false` evaluation of the predicate, we need to remember to complement the predicate:

```
(split-with (complement #{\a \e \i \o \u}) "hell
;; [(\h) (\e \l \l \o)]
```

❶We created a set literal containing vowels and used it as a predicate in a complement form. The expression returns the split of the word at the first occurrence of any vowel.

We could also split a sorted set or sorted map. `split-with` also works with normal sets or maps but the result is undetermined:

```
(defn k? [k] (complement #{k})) ; ❶
(defn kv? [k v] (complement #{[k v]}))

(split-with (k? 5) (apply sorted-set (range 10)))
;; [(0 1 2 3 4) (5 6 7 8 9)]

(split-with (kv? 4 5) (apply sorted-map (range 1
;; [([0 1] [2 3]) ([4 5] [6 7] [8 9])]
```

```
(split-with (k? 5) (set (range 10))) ; ❹
;; [(0 7 1 4 6 3 2 9) (5 8)]

(split-with (kv? 4 5) (apply hash-map (range 10)))
;; [([0 1]) ([4 5] [6 7] [2 3] [8 9])]
```

❶ `k?` and `kv?` are function to help generate a suitable predicate to use with `split-with`.

❷The first example show `split-with` used on a `sorted-set`. The result is intuitive and expected.

❸We can use `split-with` with a `sorted-map` as well. In this case the result contains key value pairs from the input map splitting at the requested key.

❹Splitting on a normal `set` is still possible, but the result of the split is undetermined.

❺Also splitting on a `hash-map` return an undetermined partitioning of key-value pairs.

Multiple splits with `split-by`

`split-with` splits the input collection the first time the predicate returns `false`. Any additional item producing the same result does not cause the sequence to split again. The following `split-by` function recursively calls `split-with` to further split the input each time the predicate switch from `true` to `false`:

```
(defn split-by [pred coll] ; ❶
  (lazy-seq
```

```
    (when-let [s (seq coll)] ; ❷
      (let [!pred (complement pred)
            [xs ys] (split-with !pred s)] ; ❸
        (if (seq xs) ; ❹
          (cons xs (split-by pred ys)) ; ❺
          (let [skip (take-while pred s) ; ❻
                others (drop-while pred s)
                [xs ys] (split-with !pred others)] ; ❼
            (cons (concat skip xs)
                  (split-by pred ys)))))))

(take 3 (split-by #(zero? (mod % 5)) (range))) ; ❽
;; ((0 1 2 3 4) (5 6 7 8 9) (10 11 12 13 14))
```

❶ `split-by` follows the typical lazy sequence organization pattern. There are two possible cons points before calling `split-by` recursively.

❷ This condition verifies the end of the input and terminates the recursion.

❸ During each recursion we move a virtual cursor forward in the input sequence and we can invoke `split-with` at that point.

❹ There are two possible outcomes: the sequence starts with a splitting point (or multiple of them in a sequence) or the splitting point is beyond the first element. `split-with` returns an empty "xs" sequence of the head elements in the case the split point is the first element.

❺ In case the split point is beyond the first element, "xs" is the first batch of results to cons into the results. We recur with the rest "ys".

❻ In case the split point is on the first element we need additional processing to discover the next split point without throwing away elements. To do so we take-while until the predicate becomes `false`, which reaches our next split point. We call these initial elements "skip" elements. "others" is anything beyond the new splitting point.

❼ We can now apply a new `split-with` on the "others" and proceed from there. What goes into the results with cons is the concatenation of the item we had to "skip" with the new group "xs". We then recur with the rest of items beyond the second split point.

❽ We can use `split-by` to partition a list of integers into multiple of 5. Note that partition-by can do something similar with the difference that splitting items are isolated in their own partition.

See also

- partition are generic partitioning functions with support for multiple splitting points.
- drop-while and take-while work on similar principles as `split-with`.

Performance considerations and implementation details

=> `O(n)` **steps, worst case**

=> `O(n)` **space, worst case**

`split-at` and `split-by` have a linear behavior in the worst case, subject to the amount of output actually consumed. Despite returning a vector, `split-at` and `split-by` are lazy functions similar to take-while or drop-while on which they are based. The reader is invited to review the performance section related to drop for additional details.

The implementation is quite simple in both cases and worth copying here for inspection:

```
(defn split-at [n coll] ; ❶
  [(take n coll) (drop n coll)])

(defn split-with [pred coll] ; ❷
  [(take-while pred coll) (drop-while pred coll)
```

❶How `split-at` is implemented in core.
❷Similarly for `split-with`, the implementation wraps a vector around drop-while and take-while calls.

10.11 WHEN-FIRST

`macro` `since 1.0`

Listing 10.11. → *Local binding, Sequential access, Lazy recursion*

```
(when-first [bindings & body])
```

`when-first` is a relatively simple macro to automate the operation of assigning a local binding to the first element of a sequence:

```
(when-first [x (range 10)] (str x)) ; ❶
;; 0

(when-first [x ()] (print "never gets here")) ;
;; nil
```

❶We use `when-first` to access the first element in a sequence.

❷If there is no first element, `when-first` returns `nil` and the body is not evaluated.

Listing 10.12. Contract

```
(when-first bindings <body>)

bindings :=> [<name> <value>]
```

input

- "bindings" is a vector of exactly 2 elements: the "name" of the local binding and its "value". "name" should be a valid Clojure symbol and "value" any sequential collection (as supported by seq).
- "body" can be any Clojure form. "name" local binding will be available for use in the body at compile-time (during macro-expansion).

notable exceptions

- `IllegalArgumentException` if "bindings" does not contain exactly 2 arguments or it is not a vector.

output

- returns: the result of evaluating "body" when "value" is a collection with at least one item, `nil` otherwise. "name" becomes available symbol during evaluation of the "body".

Examples

One good use of `when-first` is to improve readability of the idiomatic lazy recursive loop. The following `dechunk` function loops over the input to remove chunking from chunked sequences (for example ranges or vectors). Sequential chunking is normally useful feature, but there might be cases where we want full control over evaluation (for example dealing with costly or side-effecting inputs):

```
(first (map #(do (print ".") %) (range 100))) ; 
;; ................................0

(defn dechunk [xs] ; ❷
  (lazy-seq
    (when-first [x xs] ; ❸
      (cons x
        (dechunk (rest xs))))))
```

```
  (first (map #(do (print ".") %) (dechunk (range
  ;; .0
```

❶In this first experiment, we use a side-effecting map operation to print a "." dot each requested element. We can see that by asking the first element from a range, 32 dots are printed.

❷ `dechunk` implements a simple lazy loop that cons the first item of the input into the output sequence without any additional transformation.

❸By using `when-first`, we can avoid using first on "x" removing one set of parenthesis.

❹We can use `dechunk` in front of the chunked sequence to prevent the rest of the sequential computation to be performed in chunk of 32 items. The print of a single dot confirms that chunking has been removed. Please note that chunking was removed upstream of the range creation but `range` itself is still evaluating 32 items at a time.

`when-first` also avoids double evaluation of the input by reusing the transformation of the input into a sequence to extract the first item. Please check the performance section below for additional details on `when-first` evaluation policy.

See also

- first is used to access the first item of a sequential collection.
- when-let implement a similar mechanism to `when-first`, creating a local binding to a generic object (collection or not) if and only if it has been evaluated to `true`.

Performance considerations and implementation details

=> `O(1)` **Constant time**

`when-first` is a macro and its expansion happens independently from the size of the input in constant time. After expansion, `when-first` produces code that accesses the first element in a sequence, which is also constant time:

```
(clojure.pprint/pprint ; ❶
  (macroexpand
    '(when-first [x coll] (println x))))

;; (let* [temp_123 (seq coll)] ; ❷
;;  (when temp_123
;;   (let [xs_123 temp_123]
;;    (let [x (first xs_123)] ; ❸
;;     (println x)))))
```

❶We use pprint to format the macro-expansion into multiple lines. Namespaces have been removed from functions for clarity.

❷seq is called on "coll" with the effect of creating a sequential version (if it's not already a sequence). It generates `nil` when "coll" is empty.

❸If "temp_123" (a randomly generated symbol name) is not `nil`, it means we have some content to process. This is where we use first to bind the first element of the input to "x".

We can see from the macro expansion that "coll" is evaluated only once as the argument for seq. This feature could be important, for example if "coll" is not a cached sequence and we want to prevent multiple evaluations. The following `take-first` function illustrates the problem. The function produces a lazy sequence using the first element of the input collection. This initial version does not use `when-first`:

```
(defn take-first [coll] ; ❶
  (lazy-seq
    (when (seq coll) ; ❷
      (cons (first coll) ()))))

(take-first (sequence (map #(do (println "eval" 
;; eval 1
;; (1)
```

```
(take-first (eduction (map #(do (println "eval" %
;; eval 1
;; eval 1
;; (1)
```

❶ `take-first` is a simple function to create a lazy sequence containing only the first element from the input "coll".

❷sequence as a function can be used to verify if a sequence is empty. We don't want to push a `nil` in the generated output, so we first check to see if there is anything in "coll".

❸ `take-first` is invoked on an input produced with sequence. Note that we print "eval" for each item evaluation. We can see a single print of "eval" for number 1.

❹Now `take-first` is used on a sequence produced with eduction which is a non-caching sequence. We can see two prints for the evaluation of the number 1.

The provided implementation of `take-first` evaluates the input twice, once to check if it's empty and the second to get the first element. The only reason why we don't see double evaluation is because of the implicit caching provided by sequence. eduction makes the problem evident, because it does not cache evaluations of the input. Let's use `when-first` now:

```
(defn take-first [coll]
  (lazy-seq
    (when-first [x coll] ; ❶
      (cons x ()))))

(take-first (sequence (map #(do (println "eval" %
;; eval 1
;; (1)

(take-first (eduction (map #(do (println "eval" %
;; eval 1
;; (1)
```

❶The second version of `take-first` uses `when-first` which evaluates "coll" just once. It also removes one function call to first.

❷The new version of `take-first` not only produces the same results with a caching sequence but it's also more efficient. It is in fact avoiding going back to the cache to get the first item again.

❸The test with a non-caching sequence now shows that we are not evaluating the input twice.

10.12 CHUNK-CONS, CHUNK-FIRST, CHUNK-REST, CHUNK-NEXT, CHUNK-

BUFFER, CHUNK-APPEND AND CHUNK

`function` `since 1.1`

Listing 10.13. → *Chunked sequences*

```
(chunk-cons [chunk rest])
(chunk-first [s])
(chunk-rest [s])
(chunk-next [s])

(chunk-buffer [capacity])
(chunk-append [b x])
(chunk [b])
(chunked-seq? [s])
```

The functions in this section (now on `chunk-*` for brevity) are part of the chunking sequence abstraction. Chunking is a Clojure feature that allows data structures to enforce a specific fetching granularity during sequential iteration. Without chunking, a lazy sequence would always realize one item at a time advancing the related iterator one position forward in the collection. With chunking, the number of items to process can be more than one, even if they are not consumed straight away. Chunked items are parked in an intermediate iterator (the buffer) that provides elements until the end of the chunk. If

more items are requested upstream, another chunk is created and positioned in the buffer. The cycle repeats until the end of the input and all items consumed.

Chunking is mainly a performance optimization that leverages the internal data layout of some sequential collections: vectors, vector-of and ranges are among those that benefit the most from chunking. The `chunk-*` functions in this section allow other data sources to take advantage of chunking to improve their performance during sequential processing and can be divided into two groups:

1. `chunk-cons`, `chunk-first`, `chunk-rest` and `chunk-next` are almost drop-in replacements for cons construction in lazy sequences along with first , rest and next.
2. `chunk-buffer`, `chunk-append` and `chunk` serve the purpose of creating the intermediate (mutable) buffer and serve the buffer back to `chunk-cons`. Buffers should be used as internal artifacts for the sole purpose of processing chunked sequences.
3. `chunked-seq?` verifies if a sequence supports the chunked abstraction.

Warning

`chunk-*` functions were released as part of Clojure 1.1 and labeled as "implementation details" [173]. The release note also specifies that they were made public to allow experimentation. Many years later, chunking functions are used extensively in the standard library and despite being still undocumented, there is no signs of them being deprecated, changed or removed.

Contract

input

- "chunk" is an object of type `clojure.lang.IChunk` which is the returned type of `chunk` and input for `chunked-cons`.
- "rest" is any sequential collection (as per seq contract) not necessarily chunked.
- "s" indicates a chunked sequence, such that `(chunk-seq? s)` returns `true`.
- "capacity" is a positive integer (must be less than `Integer/MAX_VALUE`) that represents the size of the buffer.

- "b" must be an object of type `clojure.lang.ChunkBuffer` which is essentially a wrapper around a Java object array.
- "x" can be any object.

notable exceptions

- `NullPointerException` if "s" or "capacity" are `nil`.
- `NegativeArraySizeException` if "capacity" is less than 0.
- `ArrayIndexOutOfBoundsException` when attempting to `chunk-append` on a full `chunk-buffer`.
- `ClassCastException` trying to `chunk-first`, `chunk-rest` or `chunk-next` on something that is not a chunked sequence. Notably, `(chunk-rest ())` produces the error, because the empty list is sequential but not chunked.

output

Depending on the specific function:

- `chunk-cons` returns a `clojure.lang.ChunkedCons` object, similarly to the `clojure.lang.Cons` object returned by cons.
- `chunk-first` return the first chunk of a chunked sequence input.
- `chunk-rest` returns the rest of a chunked sequential collection after removing the first chunk, empty list otherwise.
- `chunk-next` returns the rest of a chunked sequential collection after removing the first chunk, `nil` otherwise.
- `chunk-buffer` returns a `clojure.lang.ChunkBuffer` of the given "capacity".
- `chunk-append` adds an element to a `clojure.lang.ChunkBuffer` instance, up to the available space in the buffer.
- `chunk` returns an object of type `clojure.lang.IChunk` given a `clojure.lang.ChunkBuffer` instance.

Examples

Before showing a complete example, it is useful to illustrate some aspect of chunking in isolation. Buffers are necessary to

process data in chunks, as `chunk-first` returns an (immutable) view to the internal state of the collection that cannot be processed directly. A buffer can be created and used as follows:

```
(def b (chunk-buffer 10)) ; ❶
(chunk-append b 0) ; ❷
(chunk-append b 1)
(chunk-append b 2)
(def first-chunk (chunk b)) ; ❸
(chunk-cons first-chunk ()) ; ❹
;; (0 1 2)
```

❶ `chunk-buffer` creates a mutable buffer object of size 10.
❷ We can append to the buffer up to 10 elements before it starts throwing `ArrayIndexOutOfBoundsException`.
❸ `chunk` transforms the temporary buffer into a new chunk.
❹ Chunks created from a buffer a ready to be used to generate a chunked sequence.

Please note that a buffer that was used to create a chunk becomes unusable:

```
(def b (chunk-buffer 10))
(chunk-append b 0)
(chunk b)
```

```
(chunk-append b 0) ; ❶
;; NullPointerException   clojure.lang.ChunkBuff
```

❶Once a buffer has been transformed into a chunk, any following attempt to `chunk-append` to the buffer fails with an exception. The buffer should be used once to create the buffer and then thrown away.

As explained in the introduction, `chunk-*` functions are used to process chunked sequences. The following `map-chunked` is a chunked version of map to transform items from a chunked sequence. `map-chunked` is very similar to the way map is implemented along with other functions like filter, keep. The approach used in `map-chunked` is general and can be used in other sequential functions requiring chunk-awareness:

```
(defn map-chunked [f coll]
  (lazy-seq
    (when-let [s (seq coll)] ; ❶
      (let [cf (chunk-first s) ; ❷
            b (chunk-buffer (count cf))] ; ❸
        (.reduce cf (fn [b x] (chunk-append b (f
        (chunk-cons (chunk b) (map-chunked f (ch
```

```
(take 10 (map-chunked inc (range 10000)))
;; (1 2 3 4 5 6 7 8 9 10)
```

❶The exit condition for this recursion is that there is at least one item in "coll".

❷We can proceed and take `chunk-first` from the chunked sequence.

❸A new `chunk-buffer` is created and assigned to a local binding.

❹Chunked objects such as range or vector are equipped with a `reduce` implementation that is accessible only through Java interop.

❺The buffer is converted into the corresponding chunked array instance which is used by `chunk-cons` to go into recursion and gradually build a lazy chunked sequence.

In the following example we are going to create our own chunked sequence. Data on a physical device (such as a disk or network) might be subject to hardware related constrains like a specific storage block size, or transmission packet size. On a file system for instance, data is usually organized in blocks to produce uniform allocation and predictable performance. We want to be able to read bytes from a file lazily and, at the same time, being able to choose the optimal chunk size for the device:

```
(import '[java.io FileInputStream InputStream])

(defn byte-seq [^InputStream is size]
  (let [ib (byte-array size)]
    ((fn step [] ; ❶
       (lazy-seq
         (let [n (.read is ib)] ; ❷
           (when (not= -1 n) ; ❸
             (let [cb (chunk-buffer size)] ; ❹
               (dotimes [i size] (chunk-append cb
               (chunk-cons (chunk cb) (step)))))
```

❶ `byte-seq` follows the typical lazy sequence generation pattern, enclosing the body of the `step` function in a lazy-seq call. Both "is" and "ib" (the input stream and input buffer respectively) are mutable objects and don't need to be passed as parameters to the inner `step`. So the `step` function is defined and immediately invoked (note the additional set of parenthesis).

❷ "ib" is a byte-array buffer that indicates the `.read` operation how many bytes to fetch (from the file system in this case). It is overwritten each iteration with fresh data.

❸ `.read` returns the number of bytes read. "-1" indicates the end of input, so we stop recursion unless there are more bytes to process.

❹ The second buffer "cb" is created to hold the content of the first buffer. This is redundant, but the two buffers have incompatible types and the input buffer cannot be used directly to create the `chunk-cons`.

❺ dotimes is used to transfer the content of the "ib" buffer into the "cb" buffer with `chunk-append`.

❻ `chunk-cons` is called as the last step before going into `step` recursion.

❼ When a `java.io.InputStream` class is involved in lazy processing, the `.close` operation needs to happen after the end of the computation. In this example, we read 20 bytes

using a 4096 buffer size. We need to process the bytes before the end of the with-open block which closes the input stream.

Please note that the `byte-seq` function above is useful to process unstructured bytes from an input stream lazily. Other approaches exist to load the content of a text file eagerly (such as slurp) or line by line (line-seq).

See also

- lazy-seq allows `chunk-cons` (and cons) to generate a sequence lazily.
- first and rest are the equivalent of `chunk-first`, `chunk-rest` for sequences not supporting chunking.

Performance considerations and implementation details

=> `O(1)` **steps, chunk handling**

=> `O(n)` **steps, chunk buffering**

The functions in this section are designed to work together. Overall, they process a sequence linearly in the number of items (or chunks if the input sequence is chunked). We can roughly divide performance considerations in two groups:

1. `chunk-cons`, `chunk-first`, `chunk-rest` and `chunk-next` operate in constant time to extract and produce a chunk.
2. `chunk-buffer`, `chunk-append` and `chunk` manage the intermediate buffer. Of the 3 functions, `chunk` is linear in the chunk size, while `chunk-buffer` and `chunk-append` are constant time operations.

`chunk-*` functions serve the purpose of modeling computation by chunks, greatly improving processing for those data structure where the shape of the data is important factor. By choosing the right chunk size, expensive fetching operation can be minimized while clients continue to work on an item by item basis.

To have an idea of how much they improve sequencing of vectors, we can compare the same operation on subvectors that don't support chunking (despite the fact that they share the same data structurally):

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (into [] (range 10000))] ; ❷
  (quick-bench (doall (map inc xs))))
;; Execution time mean : 330.650098 µs
```

```
  (let [xs (subvec (into [] (range 10000)) 0 9999)
    (quick-bench (doall (map inc xs))))
  ;; Execution time mean : 988.394350 µs
```

❶We use the Criterium library to properly execute the benchmarks.

❷A vector of 10k integers is created and iterated with map. As soon as map call seq internally, the vector returns a chunked iterator that knows how to load items from the internal data structures. doall is necessary to fully evaluate the sequence.

❸The same operation executes on a subvector. Subvectors don't provide a chunked iterator. We can see that `subvec` takes approximately 3 times more than `vector`.

[173] The full text of Clojure release 1.1 is visible here github.com/richhickey/clojure/blob/68aa96d832703f98f80b18cecc877e3b93bc5d26/changes.txt#L92 with this link pointing at chunked sequence functions

10.13 SUMMARY

Use of the sequential abstraction in the standard library is pervasive. Many algorithms are built on top sequential processing and often showcased as the quintessential Clojure example. This is certainly the reason why there are so many

functions dedicated to sequential processing and why Clojure developers should take the necessary time to understand them. In the last two chapters we've seen important functions dedicated to sequential creation and processing. For the remaining of part II, we are going to see all other concrete data structures such as vectors, sets and maps.

11 MAPS

Maps, along with sequences and vectors, are possibly the most flexible and used Clojure data structure. They support Clojure application design in several ways:

- Attaching "labels" to data. Each key in a map is a name for a value. Named values are easier to reason about in code and supported by other features of the language (such as destructuring.
- Supporting immutability and persistency in an efficient way (map uses the same HAMT Hash Array Mapped Trie data structure used by vectors [174]).
- Allowing lookup by key, including using the map itself as a function.
- The standard library contains many functions dedicated to map manipulation (such as assoc, merge, select-keys, etc.). The description of such functions is the topic of this chapter.

Clojure contains several kind of maps (or map-like objects) which are described using a mix of their constructors names and actual Java types (class names belong to the `clojure.lang` package unless otherwise specified). The types

in the following list implement `IPersistentMap`, which is the most specific map interface [175]:

- array-map is the default choice for smaller maps. The class name is `PersistentArrayMap`. `array-map` doesn't scale well for larger maps. For this reason, Clojure automatically promotes array-map to hash-map over a certain size threshold. Along with structs it also maintains insertion order during iteration.
- hash-map is the most flexible implementation. The class is `PerisestentHashMap`. It scales well for larger number of keys while maintaining good average performance. It does not retain insertion order, so when keys or values are requested, they are not necessarily returned in the same order they were added.
- sorted-map maintains internal ordering of key-value pairs thanks to a comparator. The type is `PeristentTreeMap`. Sorted maps come handy with certain use cases but perform slower than `hash-map`.
- records are dynamically generated classes with key-like attributes. They are not designed to scale beyond a few keys, as their main goal is to provide object-like feature such as inheritance of behavior (with protocols).

- structs are based on `PersistentHashMap` but they maintain an additional notion of minimal set of keys, or "structure" as a new `PersistentStructMap` class type. They have similar features to hash-map plus additional constraints. Their use is now discouraged after the introduction of `defrecord`.

Other Clojure collections exhibit map traits but they are not properly maps. This is because there are several interfaces contributing to maps final behavior. One of them is `clojure.lang.Associative` which is shared with other collections like vectors. As a consequence, `vectors` get a subset of map properties, such as the possibility to assoc or dissoc items. Another example are transient maps that support lookup with get, but expose a completely different `assoc!` (note the bang) function.

Note

the functions described in this chapter work on associative data structures (unless otherwise specified). `clojure.lang.Associative` is an interface currently implemented by the following: hash-map, sorted-map, array-map, records, structs, vector, sub-vectors and native-vectors.

The chapter further divides into the following sections:

- "Creating" contains map constructors for the different types of maps. The functions in this section can be used to create a new instance of any map types. Other functions like frequencies or group-by also create maps, but in the context of processing general collections.
- "Accessing" are functions dedicated to fetch a specific key or group of keys, returning their values with or without their associated key.
- "Processing" contains functions to alter the content of a map. All map types (with the exception of transient maps) are immutable (you can't mutate them in place) and persistent (each change generates a new version while older changes are still accessible).
- "Utility functions" contains other interesting functions to manipulate maps.

11.1 CREATING

11.1.1 hash-map

`function` `since 1.0`

Listing 11.1. → *Data Structure, Constructor, Key Lookup*

```
(hash-map
  ([])
  ([& keyvals]))
```

`hash-map` is the builder function for Clojure hash maps, a type of immutable data structure that supports direct access lookup by key (also called hash table in other languages):

```
(def phone-book
  (hash-map "Jack N"    "381-883-1312" ; ❶
            "Book Shop" "381-144-1256"
            "Lee J."    "411-742-0032"
            "Jack N"    "534-131-9922"))

(phone-book "Jack N") ; ❷
;; "534-131-9922"
```

❶ `hash-map` takes any number of key-value pairs. Note that "Jack N" is duplicated entry.

❷ The result of calling `hash-map` (the map itself) can be used as a function to lookup its content by key. The last value associated to the "Jack N" entry overwrites any previous entries associated with the same key.

Contract

input

- "keyvals" can be any (even) number of arguments including no arguments.

notable exceptions

- `IllegalArgumentException` is thrown when the number of arguments is not even.

output

- returns: a `clojure.lang.PersistentHashMap` instance containing the given key-value pairs, or empty. When the same key is present multiple times, the last key-value pair to enter the map overwrites the content of the previous. Note that metadata on the key (if any) are retained from the original key:

```
(def map-with-meta ; ❶
  (hash-map (with-meta 'k {:m 1}) 1
            (with-meta 'k {:m 2}) 2))

(map-with-meta 'k) ; ❷
;; 2
```

```
(-> map-with-meta ; ❸
    (find 'k)
    (first)
    (meta))

;; {:m 1}
```

❶A `hash-map` with a duplicate key "k". Note that each time we add the same key, we use different metadata.
❷Access to the key confirms the last value was overwritten.
❸Metadata on the key propagates from the original key.

Examples

`hash-map` and the syntax literal `{}` have similar goals. `{}` however, requires to specify key-value pairs at compile time, while `hash-map` allows to assemble arguments at run time, possibly from other collections:

```
(apply hash-map
  (mapcat vector
    (repeatedly #(rand-int 10))
    (range 4))) ; ❶

;; {1 3, 9 2, 8 0}
```

```
{(rand-int 10) 3 ; ❷
 (rand-int 10) 2
 (rand-int 10) 0}
;; Syntax error. Duplicate key: (rand-int 10)
```

❶The keys in this `hash-map` are randomly generated. If the generation creates a collision (like in this case) the map contains less keys than the expected 4 (as dictated by `(range 4)` in the expression).

❷An attempt to generate a similar map using the syntax literal `{}` fails because there is no evaluation of the key expression at compile time: `(rand-int 10)` enters the map as list with symbols, instead of evaluating into a random number.

One frequent case of key-value pairs are URL parameters. In the following example we have a relatively long URL and we would like to build a map of the parameters from the request:

```
(require '[clojure.string :as s])

(def long-url ; ❶
  (str "https://notifications.google.com/u/0/_"
       "/NotificationsOgbUi/data/batchexecute?"
       "f.sid=4896754370137081598&hl=en&soc-app=
       "soc-platform=1&soc-device=1&_reqid=53227
```

```
(defn split-pair [pair] ; ❷
  (let [[k v] (s/split pair #"=")]
    (if v
      [k v]
      [k nil])))

(defn params [url] ; ❸
  (as-> url x
    (s/split x #"\?")
    (last x)
    (s/split x #"\&")
    (mapcat split-pair x)
    (apply hash-map x)))

(params long-url) ; ❹
;; {"soc-device" "1"
;;  "_reqid" "53227"
;;   "soc-platform" "1"
;;   "f.sid" "48967543701370815 98"
;;   "rt" nil
;;   "soc-app" "208"
;;   "hl" "en"}
```

❶To avoid a very long line, the URL has been artificially split into separate strings. It's also easier to recognize keys and their values in this form.

❷ `split-pair` takes a string pair in the form "a=b" and returns a vector of `[a b]`. The only complexity to deal with is the potential absence of the value which is replaced with `nil`.

❸ `params` is organized to flow vertically making good use of as->. Each line executes a small bit of processing and the output becomes the input of the following form using the "x" placeholder. `hash-map` is the last part to apply.

❹The output shows what parameters are present in the URL and deals with potential missing values.

See also

- apply can be used in conjunction with `hash-map` to spread arguments from a collection instead of enumerating them explicitly.
- zipmap allows the creation of a map from two ordered collections, the first providing the keys the second the values.

- into offers another option to build a map starting from a list of pairs. Differently from `hash-map` which would require apply, into accepts a collection of key-value pairs directly.

Performance considerations and implementation details

=> `O(n)` **steps, with n number of pairs**

=> `O(n)` **space**

`hash-map` creates a new `clojure.lang.PersistentHashMap` instance and performs linearly in the number of arguments. Similarly, the new `hash-map` occupies memory space linearly in the number of distinct keys. Any following assoc operation is roughly constant time (`O(log32N)` to be precise). For very large maps (million of keys), assoc logarithmic profile becomes noticeable (please see assoc performance profile for more information).

The following benchmark compares `hash-map` with other solutions to create the same large map with 1 million keys:

```
(require '[criterium.core :refer [quick-bench]])
(import 'java.util.HashMap)
```

```
(let [pairs (into [] (range 2e6))] ; ❶
  (quick-bench (apply hash-map pairs)))
;; Execution time mean : 595.268066 ms

(let [pairs (into [] (map-indexed vector (range
  (quick-bench (into {} pairs)))
;; Execution time mean : 716.550233 ms

(let [m (HashMap. (apply hash-map (into [] (rang
  (quick-bench (into {} m)))
;; Execution time mean : 602.384550 ms

(let [ks (doall (range 1e6)) vs (doall (range 1e
  (quick-bench (zipmap ks vs)))
;; Execution time mean : 632.163418 ms
```

❶The first benchmark uses apply on a flat list of items which `hash-map` uses as key-value pairs.

❷In the second approach we create a list of pairs suitable for into. The generated map is the same as in the first example.

❸The third example assumes we have a Java `java.util.HashMap` created elsewhere. The `java.util.HashMap` instance to use in the benchmark is created from a Clojure map, but this is just for the benchmark.

❹The last benchmark uses zipmap.

We can see that there is no huge difference between `hash-map`, into or zipmap approaches, with `hash-map` showing a mild advantage over into.

If keys and values are given as vectors, we could use the following `zipmap*` inspired by zipmap. We are going to use transients internally with the assumption that relaxing immutability is going to improve on performance:

```
(defn zipmap* [v1 v2] ; ❶
  (let [cnt (count v1)]
    (loop [m (transient {}) idx 0]
      (if (< idx cnt)
        (recur (assoc! m (v1 idx) (v2 idx))
               (unchecked-inc idx))
        (persistent! m)))))

(let [v1 (into [] (range 1e6)) v2 (into [] (rang
  (quick-bench (zipmap* v1 v2)))
;; Execution time mean : 479.392200 ms
```

❶This version of `zipmap*` assumes arguments are vectors. With that constraint, we can make direct access lookup to the items and incrementally to build the final map. Notice how the map enters the loop as a transient to temporarily remove persistence. It is turned back into `persistent!` at the end.

❷We improve about 100ms by removing any sequential access to keys and values.

11.1.2 array-map

`function` `since 1.0`

Listing 11.2. → *Data Structure, Associative, Ordering*

```
(array-map
  ([])
  ([& keyvals]))
```

`array-map` creates a new `clojure.lang.PersistentArrayMap`, a form of associative data structure similar to a hash-map:

```
(def m (array-map :a 1 :b 2)) ; ❶
```

```
  (m :a) ; ❷
  ;; 1
```

❶ `array-map` has the same interface as `hash-map`, accepting an even number of parameters.

❷The map created by `array-map` can be used exactly like `hash-map`.

`array-map` maintains the same interface as `hash-map`, but it has a simpler linear implementation (compared to the tree-like implementation of `clojure.lang.PersistentHashMap`) that does not require hashing of keys. One peculiar aspect compared to hash-map or sorted-map is that `array-map` maintains insertion order during iteration.

Contract

input

- "keyvals" is an even list of arguments of any type, including no arguments.

notable exceptions

- `IllegalArgumentException` is thrown when the number of arguments is not even, showing which key is missing the respective value.

output

- returns: a `clojure.lang.PersistentArrayMap` containing the given "keyvals" in insertion order (when iterated). In case of duplicate keys, the value of the last key overrides the previous, while metadata on the key (if any) are retained from the original key (see hash-map contract for an example of metadata retention).

Examples

Clojure contains a mechanism of self-promotion for `array-map` into hash-map. It can be seen in action using map literals:

```
(type {0 1 2 3 4 5 6 7 8 9})
;; clojure.lang.PersistentArrayMap ; ❶

(type {0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
;; clojure.lang.PersistentHashMap ; ❷
```

❶A map with 5 keys is created as `PersistentArrayMap`.
❷As soon as the map grows beyond 10 keys it is automatically promoted to `PersistentHashMap`.

assoc (and functions like zipmap which are based on it) also promotes an `array-map` into `hash-map` if necessary:

```
(def an-array-map (apply array-map (range 200)))
(type an-array-map)
;; clojure.lang.PersistentArrayMap ; ❶

(def an-array-map? (assoc an-array-map :a :b))
(type an-array-map?)
;; clojure.lang.PersistentHashMap ; ❷

(def an-array-map! (dissoc an-array-map 0))
(type an-array-map!)
;; clojure.lang.PersistentArrayMap ; ❸
```

❶We start from an `array-map` with 100 keys. It is correctly reported as a `PersistentArrayMap` instance.
❷As soon as we assoc into `an-array-map` we get back a `PersistentHashMap` instance.
❸Note however that dissoc does not self-promote.

The reason for the self-promotion is that `array-map` is a simple map-like data structure that does not scale beyond a few hundreds entries. Even with this limitation, `array-map` has some applications related to its insertion order guarantee during iteration. We could for example aggregate the headers of a table as `array-map` with the idea to store its content as CSV file (CSV, or comma separated values, is a simple and ubiquitous plain text exchange format for tabular data):

- The key is the name of the column to be used when exporting data into CSV files.
- The value is a vector pair containing the mapping in the database column and a function that can be used to validate the corresponding data.
- The position of the key in the map is the position at which the corresponding column should appear.

Assuming we fetch data from a database as `query-results`, here's how it could work:

```
(def query-results ; ❶
  [{:date "01/05/2012 12:51" :surname "Black"
    :name "Mary" :title "Mrs" :n "20"
    :address "Hillbank St" :town "Kelso" :postco
   {:date "01/05/2012 17:02" :surname "Bowie"
    :name "Chris" :title "Miss" :n "44"
```

```
     :address "Hall Rd" :town "Sheffield" :postco
    {:date "01/05/2012 17:08" :surname "Burton"
     :name "John" :title "Mr" :n "41"
     :address "Warren Rd" :town "Yarmouth" :postc

(defn checkfn [predicate] ; ❷
  (fn [val]
    (let [res (predicate val)]
      (if (predicate val)
        val
        (throw (RuntimeException.
          (str "Error: '" val "' is not valid"))

(def customers-format ; ❸
  (array-map
    'TITLE  [:title (checkfn #{"Mrs" "Miss" "Mr"
    'FIRST  [:name (checkfn (comp some? seq))]
    'LAST   [:surname (checkfn (comp some? seq))
    'NUMBER [:n (checkfn #(re-find #"^\d+$" %))]
    'STREET [:address (checkfn (comp some? seq))
    'CITY   [:town (checkfn (comp some? seq))]
    'POST   [:postcode (checkfn #(re-find #"^\w{
    'JOINED [:date (checkfn #(re-find #"^\d{2}/\

(defn csv-str [coll] ; ❹
  (str (apply str (interpose "," coll)) "\n"))

(defn format-row [format] ; ❺
```

```
    (fn [row]
      (let [specs (map second format)
            data (map (fn [[column checkfn]]
                        (checkfn (row column))) sp
        (csv-str data))))

  (defn format-data [data format] ; ❻
    (let [headers (csv-str (keys format))
          body (map (format-row format) data)]
      (apply str headers (seq body))))

  (println (format-data query-results customers-fo
  ;; TITLE,FIRST,LAST,NUMBER,STREET,CITY,POST,JOIN
  ;; Mrs,Mary,Black,20,Hillbank St,Kelso,TD5 7JW,0
  ;; Miss,Chris,Bowie,44,Hall Rd,Sheffield,S5 7PW,
  ;; Mr,John,Burton,41,Warren Rd,Yarmouth,NR31 9AB
```

❶ `query-results` is a sample output for a typical database query about customers. Data are organized by maps with keys corresponding to the column of the corresponding table. These column are not necessarily in the format required for data export, for example CSV files.

❷ `checkfn` creates validation functions. It takes a function of 1 argument and creates a function that can be used for validation. The "predicate" is used to determine the validity of the input. If the input is not valid, it throws a `RuntimeException`.

❸ `customer-format` is an `array-map` representing the relationship between the internal database format and the external format. The keys have the right name and the right order. Values are vector of two elements: the first is the name of the column in the database, the second is a validation function created with `checkfn`. Note the use of symbols instead of keywords for keys: symbols print as they are, without ":" prefix that would need to be removed.

❹ `csv-str` takes a collection of values and composes them into a comma separated string ending in a new line.

❺The `format-row` function accepts a "format" parameter specification and uses it to create another function. The output function can be used with a single "row" parameter. The returned function takes 1 database row and produces a

single line comma separated string containing the selected data in the right order. The function is also responsible for invoking each column validation function.

❻ `format-data` transforms each record into the corresponding row, invoking `format-row` on each database record.

❼ We can see the output from `format-data` when all validations are successful.

Storing information in metadata

Clojure contains a feature called "metadata". Metadata allows "tagging" of some Clojure types with general information. This information is not normally visible, but is accessible with functions like meta and is used throughout Clojure internals to carry types and debugging descriptions.

We could for example attach the mapping information between the database columns and the desired external name to the key of a `hash-map` or `array-map`. If for example "JOINED" is the desired name for the external format, but ":created_at" is what is in the database, it makes sense to link them together. One option is certainly the one provided with the example (as the value of the map), but another is to use metadata on the key:

```
(def customers-format ; ❶
  (array-map
    (with-meta 'TITLE  {:db :title   }) (checkfn
    (with-meta 'FIRST  {:db :name    }) (checkfn
    (with-meta 'LAST   {:db :surname }) (checkfn
    (with-meta 'NUMBER {:db :n       }) (checkfn
    (with-meta 'STREET {:db :address }) (checkfn
    (with-meta 'CITY   {:db :town    }) (checkfn

(map (comp :db meta) (keys customers-format)) ;
;; (:title :name :surname :n :address :town)
```

❶We can attach metadata to symbols (but it can't be done on keywords) and remove the database column information from the values in the map.
❷meta gives access to metadata.

See also

- hash-map is the more robust and scalable version of `array-map`. `array-map` differs from `hash-map` in implementation details, performance profile and insertion order guarantee.

- sorted-map implements and maintains items ordering. Use `sorted-map` when you need a map-like data structure which is ordered (and persist order) by a comparator.
- assoc is used to incrementally add elements to an `array-map`. Remember that `array-map` can self-promote to `hash-map` after 10 keys (the exact number is an implementation detail, not a feature to rely on).

Performance considerations and implementation details

=> `O(n)` **steps, best case**

=> `O(n^2)` **steps, worst case**

`array-map`, despite the map-like interface, is based on arrays. Pairs are stored consecutively inside an array: one item is the key and the following is the value. Persistency (the property by which immutable data structures appear to mutate while allowing access to previous versions) is achieved by copying the array before using assoc to add a new pair. Access by key is linear instead of constant time, because the key is searched iterating from the start of the array. `array-map` performance

factors contribute to the behavior of functions like assoc trying to upgrade to `hash-map` as soon as possible.

The `array-map` function specifically, creates a new `clojure.lang.PersistentArrayMap` linearly in the number of arguments. Differently from `hash-map` though, `array-map` needs extra work to deal with duplicated keys. If a duplicate is found in the list of arguments, the previous key needs to be updated, generating a linear scan to search for it.

We can see the effects of key duplicates by creating an `array-map` with 500 keys and adding the last key an additional 500 times:

```
(require '[criterium.core :refer [quick-bench]])

(let [r1 (doall (concat (range 1000) (repeat 100
      r2 (doall (range 2000))] ; ❷
  [(quick-bench (apply array-map r1))
   (quick-bench (apply array-map r2))])

;; Execution time mean : 11.177368 ms
;; Execution time mean : 4.417755 ms
```

❶"r1" is a range of 2000 elements where the last 1000 are repeating the same number. The resulting map is going to contain 500 unique keys.
❷"r2" is a range of 2000 items without duplicates. The resulting map will contain 1000 keys.

The possible effect of duplicates is another reason to discourage the creation of large `array-maps`, although our benchmark simulates an unlikely situation.

11.1.3 sorted-map and sorted-map-by

`function` `since 1.0`

Listing 11.3. → *Data structure, Sorting, Associative*

```
(sorted-map [& keyvals])
(sorted-map-by [comparator & keyvals])
```

`sorted-map` is the constructor function for `clojure.lang.PersistentTreeMap`, a type of data structure similar to hash-map which maintains ordering of keys:

```
(sorted-map :c 3 :b 2 :a 1) ; ❶
;; {:a 1, :b 2, :c 3}
```

❶ `sorted-map` creates a map from an even list of key-value pairs and prints elements in order.

`sorted-map` uses the default comparator to maintain order. `sorted-map-by` can be used to pass a different comparator:

```
(sorted-map-by #(< (:age %1) (:age %2)) ; ❶
  {:age 35} ["J" "K"]
  {:age 13} ["Q" "R"]
  {:age 14} ["T" "V"])

;; {{:age 13} ["Q" "R"], {:age 14} ["T" "V"], {:
```

❶ `sorted-map-by` accepts a function of two arguments returning -1 (first argument less than second), 0 (arguments are equal) or 1 (first argument is more than second), which is the typical interface for a comparator.

Ordering requires that the keys in the map can be compared. It follows that key objects:

1. Need to support the `java.lang.Comparable` interface.
2. Keys must be of the same type of the first key in the map (or provide a `Comparable` interface that accepts different types).

Contract

input

- "keyvals" is a list of arguments of any type, including no arguments. Arguments need to be in pairs, so the list count must be even.
- "comparator" is a function of 2 arguments. The function should return a negative, 0 or positive number to indicate the first argument is less than, equal or more than the second argument respectively. It's a mandatory argument for `sorted-map-by`.

notable exceptions

- `IllegalArgumentException` is thrown when the number of arguments is not even, showing which key is missing the respective value.
- `ClassCastException` when the key cannot be cast to `java.lang.Comparable` or two keys are of a different type and there's no compatible `Comparable` interface.

output

- returns: a `clojure.lang.PersistentTreeMap` containing the given "keyvals" in the order determined by the default comaparator (`sorted-map`) or the given comparator (`sorted-map-by`). In case of duplicate keys, the value of the last key overrides the previous, while metadata on the key (if any) are retained from the original key.

Examples

Like for the other map types (hash-map and array-map), in case of duplicated keys `sorted-map` maintains the metadata (if any) from the first key and the value of the second (or later) key:

```
(defn timed [s] ; ❶
  (let [t (System/nanoTime)]
    (println "key" s "created at" t)
    (with-meta s {:created-at t})))

(def m (sorted-map (timed 'a) 1 (timed 'a) 2)) ;
;; key a created at 206892376620199
;; key a created at 206892376884656

(println m)  ; ❸
;; {a 2}
```

```
  (meta (ffirst m)) ; ❹
  ;; {:created-at 207021400730108}
```

❶The `timed` function takes a symbol "s" and returns the symbol after adding metadata containing the time of the creation. It also prints on standard output when the key was created.

❷We create a `sorted-map` using the `timed` function to insert keys. The second key is the same as the first.

❸We can see the value of the "a" key is the second value that was added.

❹But the metadata on the key is coming from the creation of the first key.

One interesting use of `sorted-map` is implementing priority queues [176]. Priority queues are at the base of important algorithms in computer science, such as searching for the optimal path in a graph. The A* (a-star) algorithm, for instance, can be implemented using a priority queue [177].

A* was devised in the late '60 to guide a robot around obstacles in the most optimal way and is still used nowadays in games and navigation software. A* requires a heuristic, an approximate distance to the destination used to filter out

unwanted paths. In the case of a car navigation system for example, the heuristic could be the straight line distance between two locations, which is easily obtainable from geospatial data.

We could implement the priority queue of the optimal paths using `sorted-map-by` with a composite vector key. Here's how it would look like after processing two paths departing from "origin" and going to "point 1" and "point 2":

```
(sorted-map-by compare ; ❶
  [4.5 "point 2"] [["origin"] 1.5] ; ❷
  [5.5 "point 3"] [["origin"] 2.5]) ; ❸
```

❶We create a priority queue with `sorted-map-by` using compare as comparator.

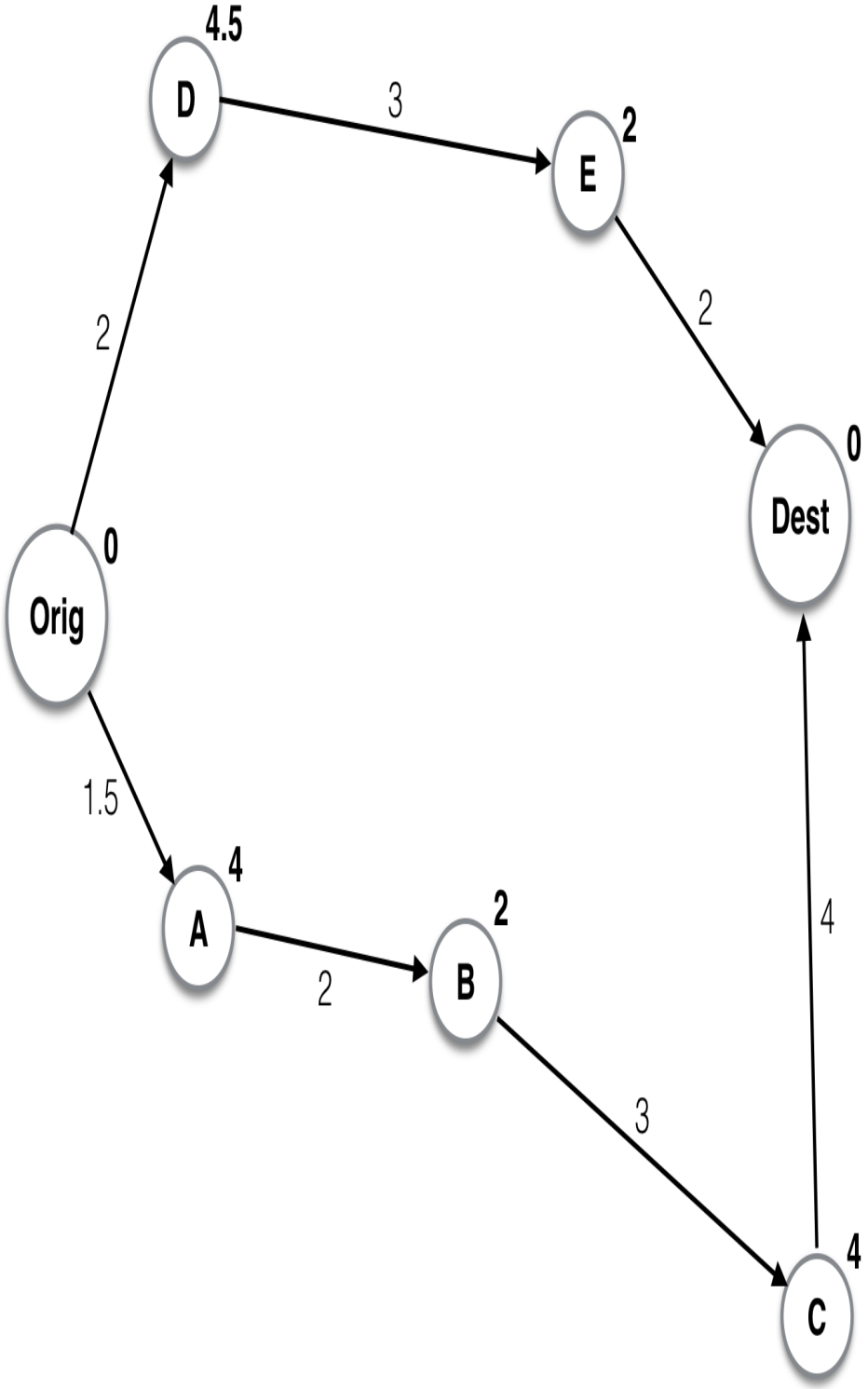
❷Keys are vectors containing the composite distance (physical path distance plus the heuristic) and the location this refers to ("point 2" or "point 3" for example).

❸The values in the map are also `vectors`: the first element is the `vector` of locations necessary to reach the key (in this case just "origin") and the second contains the distance from the last visited location to the current (for example the distance from "origin" to "point 2" is 1.5).

`sorted-map-by` in the example above shows how to use composite keys with sorted maps. vectors can be used as keys but they don't implement the `java.lang.Comparable` interface natively, requiring the explicit use of compare as comparator.

Before we can iterate locations, we need a suitable representation. Locations can be represented graphically as a cyclic directed graph like the one shown below:

Figure 11.1. Locations and routes connecting them. We want to move from "Orig" to "Dest" using the shortest path.


4.5
D
3
E
2
2
2
0
Dest
0
Orig
1.5
4
A
2
B
2
3
4
C
4

We could translate the graph using a map containing all locations as keys:

```
(def graph
  {:orig [{:a 1.5 :d 2} 0]  ; ❶
   :a    [{:orig 1.5 :b 2} 4]
   :b    [{:a 2 :c 3} 2]
   :c    [{:b 3 :dest 4} 4]
   :dest [{:c 4 :e 2} 0]
   :e    [{:dest 2 :d 3} 2]
   :d    [{:orig 2 :e 3} 4.5]})
```

❶ The value for each location key is a `vector` of two items. The first item is another map containing the connected locations and their distances from the key. The second element is the heuristic for that location.

The following implementation of the A* algorithm takes `graph` as input, an origin and a destination. The algorithm is implemented by three functions: `a*` is the main entry point, `walk` is a tail recursive function that traverses the graph and `discover` helps removing already visited nodes at each iteration (eliminating potentially infinite cycles). Each `walk` iteration selects the node that minimizes the heuristic plus the

actual distance from origin. At the same time, we `discover` new nodes to put in the priority queue for the next iteration:

```
(defn discover [node path visited]
  (let [walkable (first (graph node))   ; ❶
        seen (map last (keys visited))]
    (reduce dissoc walkable (conj seen (last pat

(defn walk [visited dest]
  (loop [visited visited]
    (let [[[score node :as current] [path total-
      (if (= dest node)
        (conj path dest)
        (recur
          (reduce-kv ; ❹
            (fn [m neighbour partial-distance]
              (let [d (+ total-distance partial-
                    score (+ d (last (graph neig
                (assoc m [score neighbour] [(con
            (dissoc visited current) ; ❻
            (discover node path visited)))))))

(defn a* [graph orig dest]
  (walk (sorted-map-by compare [0 orig] [[] 0]) 

(a* graph :orig :dest) ; ❽
;; [:orig :d :e :dest]
```

```
(a* graph :d :c)
;; [:d :e :dest :c]
```

❶In order to access all accessible nodes from the current location, we need to make access to the graph using the location "node" as key.

❷The repeated use of dissoc removes nodes that we've already seen.

❸destructuring is very important here to help reduce the amount of code we need to write. The first from `visited` is the location which currently minimizes the sum of the distance traveled so far with the heuristic (in this example the straight line distance to destination).

❹reduce-kv iterates over the list of traversable nodes starting from the priority map of visited locations. It also calculates aggregated metrics like the total distance from the origin.

❺When moving to a new location, we also need to build the path to get to it in case this location becomes the solution (the current location is the destination).

❻We need to remove the first node in the priority list before we start calculating again to avoid infinite recursion.

❼During initialization, we create a sorted map with `sorted-map-by` containing only "orig".

❽We can invoke `a*` to search for the best path from `:orig` to `:dest` or any other pair of locations.

Comparators and uniqueness of elements

As we are also going to see for sorted-set-by, the custom comparator for `sorted-map-by` needs to be carefully designed to work properly with all kind of keys. When keys are collections, for instance, there is a specific problem that might be difficult to detect. Observe the following example, where we are trying to create a `sorted-set` ordered by the inverse of the size of its vector keys:

```
(sorted-map-by
  #(compare (count %2) (count %1)) ; ❶
  [:a :b] 1 [:a] 2 [:b] 3)
  ;; {[:a :b] 1, [:a] 3}
```

❶A flawed custom comparator for `sorted-map-by`.

The first sign that the comparator is not working properly is the missing `[:b]` key in the resulting set. A second hint is that subsequent operations fail unpredictably:

```
(def ordered-by-count ; ❶
  (sorted-map-by
    #(compare (count %2) (count %1))
    [:a :b] 1 [:a] 2 [:b] 3))

(assoc ordered-by-count [:x] 4) ; ❷
;; {[:a :b] 1, [:a] 4}

(dissoc ordered-by-count [:x]) ; ❸
;; {[:a :b] 1}
```

❶ `ordered-by-count` is a `sorted-set-by` the inverse of the size of its vector keys.

❷ We try to assoc the key `[:x]` with value "4" but the keys does not appear in the resulting set.

❸ When we try to dissoc a non existent key, it results in removing an actual but different key.

The comparator does not work properly because it is used to check if a key is in the map as well as deciding in which relative order it should appear in the resulting `sorted-map`. The comparator does not take into account the first aspect, the fact that is going to be used to check if the key is already in the map:

```
(def flawed-comparator
  #(compare (count %2) (count %1)))

(flawed-comparator [:a] [:x]) ; ❶
;; 0
```

❶ This is conceptually how the comparator is used to verify a key `[:x]` is the same as another already existent key `[:a]` in the `sorted-set`. "0" here means the keys are the same, but this only means "same size" not "equal".

The comparator is called for each key already in the `sorted-set` and the new `[:x]` key to be inserted. Since they have the same size, it wrongly concludes `[:x]` is already in the set and should not be added. What is missing is an additional check if the keys are actually the same or not:

```
(def good-comparator ; ❶
  #(compare [(count %2) %1] [(count %1) %2]))

(good-comparator [:a] [:x]) ; ❷
-23
```

❶ A `good` comparator for compound keys should check both the specific custom ordering constraint as well as full key equality. By pairing these two aspects in a vector, we use vector equality to verify them at the same time.

❷ `good-comparator` now returns "-23" which is the difference in distance between the letter "a" and "x" in the alphabet. But what we care about from the point of view of compare is that this number is negative, indicating the two keys are not the equal and they should be both part of the `sorted-set`.

With the correct formulation of the custom comparator, we can see that `sorted-set-by` behaves as expected:

```
(def ordered-by-count
  (sorted-map-by
    #(compare [(count %2) %1] [(count %1) %2])
    [:a :b] 1 [:a] 2 [:b] 3))

(assoc ordered-by-count [:x] 4) ; ❶
;; {[:a :b] 1, [:a] 2, [:b] 3, [:x] 4}

(dissoc ordered-by-count [:x]) ; ❷
;; {[:a :b] 1, [:a] 2, [:b] 3}
```

❶After fixing the custom comparator, assoc of a new key behaves as expected as well as maintaining the ordering by size of the key (bigger vectors first).

❷dissoc also works as intended, not removing an element that should not be removed.

The reader is also invited to check the call-out section in sorted-set for additional considerations.

See also

- sorted-set are similar functions to create an ordered set instead of a map.
- subseq are used to generate a sequence starting from an element of a sorted map or set.

- hash-map or array-map are the other kind of persistent dictionary-like data structures available in Clojure.

Performance considerations and implementation details

=> `O(N)` **linear in the number of keys to insert**

`sorted-map` and `sorted-map-by` are implemented as persistent Red-Black trees, a solution providing `O(logN)` access , insertion and deletion guarantee [178]. The "N" in `O(logN)` refers to the number of keys in the `sorted-map` at the time of insertion of a new key. In building the `sorted-map` from scratch though, the number of steps is linear in the number of keys to insert.

In the following benchmarks we compare different ways to create a `sorted-map` with 1000 keys, with the last example using a Java `TreeMap` for comparison:

```
(require '[criterium.core :refer [quick-bench]])
(import '[java.util TreeMap])

(let [pairs (into [] (range 2e3))] ; ❶
  (quick-bench (apply sorted-map pairs)))
;; Execution time mean : 452.317814 µs
```

```
(let [pairs (into [] (map-indexed vector (range
  (quick-bench (into (sorted-map) pairs)))
;; Execution time mean : 469.676463 µs

(let [m (apply hash-map (into [] (range 2e3)))]
  (quick-bench (TreeMap. m)))
;; Execution time mean : 158.487373 µs
```

❶We use apply to spread a list of key-value pairs for the `sorted-map` constructor.

❷In this case we use into with the same number of keys.

❸The last case builds a similar but mutable Java data structure.

The benchmark shows almost no difference between apply and into. The mutable Java version, as expected, is roughly 3 times faster.

While discussing the `a*` algorithm, we used first to access the best location from the sorted map. Although first is in general a good choice, it adds the overhead of transforming the `sorted-map` into a sequence. The Java class implementing `sorted-map` contains two public methods `min` and `max` that are not exposed as part of the standard library. The following

benchmark shows a significant performance gain when we access `.min` using Java interop:

```
(require '[criterium.core :refer [quick-bench]])
(import '[clojure.lang PersistentTreeMap])

(let [m (apply sorted-map (range 10))] ; ❶
  (quick-bench (first m))) ; ❷
;; Execution time mean : 57.393946 ns

(let [m (apply sorted-map (range 10))]
  (quick-bench (.min ^PersistentTreeMap m))) ; ❸
;; Execution time mean : 6.234699 ns
```

❶ The first benchmark access the map using first.

❷

❸ In the second case we use `PersistentTreeMap::min()` method through Java interop. We can see an order of magnitude speed up.

It's common case to access a `sorted-map` to retrieve the first or the last element (especially when they are used as priority queues). In that case we can use `min` or `max` methods directly

remembering to type hint the map to avoid a very costly reflection lookup.

11.1.4 create-struct, defstruct, struct-map, struct and accessor

`functions (except defstruct)` `since 1.0`

Listing 11.4. → *Constrained Map, Guarded keys*

```
(create-struct [& keys])
(defstruct [name & keys])
(struct-map [s & inits])
(struct [s & vals])
(accessor [s key])
```

This group of four functions and one macro (`defstruct`), is the programming interface to the `struct` type in Clojure. A `struct` is an instance of `clojure.lang.PersistentStructMap` , a map-like type that embeds a structural definition: differently from a plain map, a `struct` always guarantees the presence of the defined set of keys.

`struct` is designed to remove repetition of keys when many map-like objects need to be created (for example to hold large

structured data after parsing them into Clojure). They were later superseded by defrecord that additionally offers some form of inheritance. `struct`, however, is not deprecated (there is no official "deprecation" label in the function body) and still in use in some parts of the Clojure standard library (for example xml processing, resultset-seq or `cl-format`). Along with array-maps and defrecord, `struct` is the only other map-type to maintain insertion order of keys.

Contract

input

- "keys" is the minimal set of keys that should be present in the `struct`.
- "name" defines a name for the `struct` which is assigned to a local var in the current namespace (by `defstruct`).
- "s" is a struct definition created with `create-struct` or `defstruct`. It is used by `struct-map`, `struct` and `accessor` to get access to a previously defined `struct`.
- "inits" are unstructured key-value pairs that can be passed to `struct-map` to initialize a new `struct` instance.

- `vals` is a list of values passed to `struct`. Values should match positionally in relation to the keys in the `struct` definition.
- `key` is used by `accessor` to create a function to access that key in a given `struct` instance.

notable exceptions

- `IllegalArgumentException` if too many "vals" are passed to `struct` given a definition that has less keys than the given values.
- `RuntimeException` trying to dissoc a key that is part of the struct definition.

output

- `create-struct` creates a new `PersistentStructMap$Def` definition object with the given "keys". At least one key is required.
- `defstruct` is the equivalent of assigning the definition created by `create-struct` to a var.

- `struct-map` creates a new `struct` instance based on a `struct` definition and a list "inits" of key value pairs. If "inits" does not contain the keys defined in the `struct`, then keys are assigned a default `nil` value. Other key-value pairs are accepted along with the keys from the definition.
- `struct` accepts a definition and a list of values. `struct` will try to match values positionally in relation to the key definitions. Missing values for keys result in a default value of `nil` being assigned. "vals" can be equal or less than the number of keys.
- `accessor` returns a function of one argument. The function accepts a `struct` instance and retrieves the value at the specific "key".

Examples

`struct` are map-like types with a minimal set of required keys. When using a `struct` we need to distinguish between their definition (the result of calling `create-struct`) and their instantiation (the result of calling `struct` or `struct-init` on a definition). Equality for instance, has a different meaning for definitions and instances:

```
(= (create-struct :x :y) ; ❶
   (create-struct :x :y))
;; false

(= (struct (create-struct :x :y) 1 2) ; ❷
   (struct (create-struct :x :y) 1 2))
;; true
```

❶In the first example, we compare two `struct` definitions. Despite they contain the same keys, the two definitions are independent.

❷The second example instantiate an actual `struct` object from the definition using the same values. Two instances with same keys and values are the same.

Additionally, definitions with different set of keys can generate equal `struct` instances by adding the missing keys:

```
(defstruct point-2d :x :y) ; ❶
(defstruct point-3d :x :y :z) ; ❷

(= (assoc (struct point-2d 1 2) :z 3) ; ❸
   (struct point-3d 1 2 3))
;; true
```

❶ `point-2d` is a `struct` definition containing 2 coordinates, ":x" and ":y".
❷ `point-3d` adds a third dimension ":z" to the definition.
❸ A point with only two dimensions can be the same as a three-dimensional point by adding the missing coordinate. Note that for the equivalence to work, the keys need to have the same name, values and order.

A `struct` is much simpler than the more powerful defrecord. One fundamental difference is that `create-struct` creates an anonymous `struct` definition, while defrecord creates a Java class as a side effect. defrecord generates a class by design: this allows powerful features such as inheritance of record types. At the same time, records are more complicated to use across namespaces (as they require an explicit `import` statement) and they are trickier to reload when their definition changes. The following example illustrates the difference:

```
(struct (create-struct :a :b :c) 1 2 3) ; ❶
;; {:a 1, :b 2, :c 3}

(type (defrecord abc [a b c])) ; ❷
;; java.lang.Class
```

```
  (abc. 1 2 3) ; ❸
  ;; #user.abc{:a 1, :b 2, :c 3}
```

❶The `struct` definition exists without necessarily assigning a name, so they can be created and instantiated directly.
❷defrecord on the other hand, requires a name and generates a `java.lang.Class` that cannot be used directly.
❸The class "abc" generated by defrecord, is now available in the "user" namespace and can be used to initialize a new record instance.

In the following example, we are going to make use of inlined `struct` definitions. A "waypoint" contains the coordinates of a point of interest on Earth. It contains a type, id, latitude and longitude. Here's the related `struct` definition:

```
  (require '[clojure.string :refer [split-lines sp

  (def waypoints "https://tinyurl.com/station-locs
  (def lines (-> waypoints slurp split-lines))

  (def waypoints
    (let [sdef (create-struct :type :lat :lon :id)
      (transduce
        (comp
          (map #(split % #"\s+"))
```

```
          (map #(apply struct sdef %))) ; ❸
        conj
        lines)))

  (first waypoints) ; ❹
  ;; {:type "VHF", :lat "0.000000", :lon "0.000000
```

❶This url points to a list of about 50000 point of interest (originally downloaded from ral.ucar.edu/maps/Station.loc). We can load its content and proceed to split the lines using `clojure.string/split-lines`.

❷The `struct` definition is created as part of the computation to parse the list with `create-struct`. Note that if necessary, the definition could be altered at runtime, for example to add a specific key looking at the ":type" of the waypoint.

❸Each line is split into four values which are applied to the `struct` definition.

❹We can see one example of waypoint that appears printed exactly like a normal map.

`struct-map` is the other way to instantiate a `struct`. It's useful when converting a normal hash-map into a `struct` with corresponding keys:

```
(defstruct waypoint :type :lat :lon :id) ; ❶

(def coordinates [ ; ❷
  {:alt 150 :lat "18.3112" :lon "3.1314" :id "XV
  {:alt 312 :lon "10.04883" :id "FFA" :type "XFV
  {:temp 78.3 :lat "23.7611" :id "XJP"}])

(defn to-waypoints [coords] ; ❸
  (map #(apply struct-map waypoint (mapcat ident

(to-waypoints coordinates) ; ❹

;; ({:type nil, :lat "18.3112", :lon "3.1314", :
;;  {:type "XFV", :lat nil, :lon "10.04883", :id
;;  {:type nil, :lat "23.7611", :lon nil, :id "X
```

❶ The `waypoint struct` is now defined using a namespace local var definition. The definition is visible in the current namespace (or other namespaces if required) from this point onward.

❷ We receive a vector of heterogeneous coordinates, sometimes with keys we need, sometimes with missing data.

❸ `to-waypoints` transforms a list of maps into waypoint `struct` objects. We need to use mapcat to flatten the content of the map into a plain list of key-value pairs without parenthesis. We can then use apply to feed the list into `struct-map`.

❹ The end result is a list of structs. What is different from before is that the presence of the minimal set of keys from the waypoint definition is guaranteed, although the corresponding value could be `nil`.

`accessor` optimizes frequent access to the field of a `struct` by skipping the typical hash-based access by using a faster array-index lookup. With reference to the previous example, we can define accessors to access waypoint instances as follows:

```
(def k-type (accessor waypoint :type)) ; ❶
(def k-lat (accessor waypoint :lat))
(def k-lon (accessor waypoint :lon))
(def k-id (accessor waypoint :id))
```

```
(def waypoints (to-waypoints coordinates))
(map k-id waypoints) ; ❷
;; ("XVA" "FFA" "XJP")
```

❶ `accessor` creates a function in the current namespace that accepts a `struct` instance. We define an `accessor` for each key in a waypoint.

See the performance section below to have an idea of the possible performance gain.

Lisp `defstruct`

`defstruct` specifies a set of supported keys similarly to how a Java class declares supported attributes. At the same time, `struct` creates an instance of such specification similarly to how objects represent instances of a class. As a consequence, it was common question on the Clojure mailing list how to design inheritance using `defstruct`, because of its class-like appearance. But `defstruct` doesn't offer any of the powerful features an object system provides, such as behavioral inheritance. One additional reason to think `struct` in terms of classes and objects, is the presence of the same keyword in Common Lisp:

```
;; Lisp code

(defstruct ; ❶
  (person (:constructor create-person (id name age)))
     id name age)
```

```
(create-person 1 "reborg" 40) ; ❷
;; #S(PERSON :ID 1 :NAME "reborg" :AGE 40)
```

❶ `defstruct` also exists in Common Lisp and predates CLOS (Common Lisp Object System) integration. We declare here a constructor for a "person" record with id, name and age.

❷The generated constructor can be used invoking it with the required arguments.

Common Lisp `defstruct` is designed to offer object oriented features like inheritance:

```
(defstruct
  (female (:include person)) ; ❶
     (gender "female" :type string))

(make-female :name "Debby")
;; #S(FEMALE :ID NIL :NAME "Debby" :AGE NIL :GEN
```

❶By including "person" we inherit attributes common to all people and at the same time we can define additional attributes such as "GENDER".

The similarities between Clojure `defstruct` and Common Lisp might explain why people coming to Clojure in the early days (especially from a Lisp background) were trying to use `struct` differently from what it was intended for (an optimization). Clojure introduction of `defrecord` and `defprotocol` a few

years later removed this ambiguity. defrecord also offers the same capabilities of `defstruct`: currently, there are very few use cases that require `struct` instead of defrecord and some of the differences have been highlighted in this chapter. This is the reason why the official Clojure documentation points the reader to defrecord instead of `defstruct`.

See also

- defrecord effectively replaces `defstruct` for the definition of strongly typed map-like structures. It additionally offers a form of inheritance and support for interface declaration. Prefer `defrecord` to emulate object oriented programming. Unless you need lightweight throwaway map-like objects for internal computations, defrecord is almost always the preferred choice.

Performance considerations and implementation details

=> `O(1)` **only accessor**

=> `O(n)` **other functions and macros**

Creating the `struct` definition is linear in the number of the requested keys and bounded by the highest number of elements

addressable in a Java array (`(Integer/MAX_VALUE)` or 2,147,483,647 keys). The creation of a `struct` instance given a definition, is also linear in the number of keys. `accessor` is the only function returning in constant time.

For practical use, let's verify how `struct` performs compared to defrecord. With reference to creating a large number of waypoints from a file, we can compare overall performance and memory overhead between `struct`, defrecord, array-map and hash-map:

```
(require '[clojure.java.io :refer [reader]])
(require '[clojure.string :refer [split-lines sp
(require '[criterium.core :refer [quick-bench]])

(def waypoints "https://tinyurl.com/station-locs
(def lines (->> waypoints slurp split-lines (map
(last lines)
;; ["ARP" "44.244823" "-84.179802" "Y31"] ; ❶

(defstruct w-struct :type :lat :lon :id) ; ❷
(defrecord w-record [type lat lon id])
(defn w-map [type lat lon id] {:type type :lat l
(defn w-hmap [type lat lon id] (hash-map :type t

(quick-bench (doall (map #(apply struct w-struct
;; Execution time mean : 8.602227  ms
```

```
(quick-bench (doall (map #(apply ->w-record %) li
;; Execution time mean : 6.172892  ms

(quick-bench (doall (map #(apply w-map %) lines)
;; Execution time mean : 7.157129  ms

(quick-bench (doall (map #(apply w-hmap %) lines
;; Execution time mean : 20.554723 ms
```

❶ `lines` contains a list of waypoints as vectors of strings. By asking the last we have fully realized the sequence.

❷ `w-struct`, `→w-record`, `w-map` and `w-hmap` are the 4 constructors that we need for the test. Note that `w-map` creates an `array-map` not a `hash-map`.

We can see that `defrecord` is the fastest of the group, with `array-map` and `struct` following. The creation of a hash-map has the worst performance. We can now benchmark access to the waypoints. Each type has a few ways to access values given a key. For `struct` we are going to create specific `accessor`:

```
(def points-struct (doall (map #(apply struct w-
(def points-record (doall (map #(apply ->w-recor
(def points-map    (doall (map #(apply w-map %)
```

```
(def points-hmap   (doall (map #(apply w-hmap %)

(def id (accessor w-struct :id))

(let [w (first points-struct)] (quick-bench (id
;; Execution time mean : 4.318680 ns

(let [w (first points-struct)] (quick-bench (:id
;; Execution time mean : 10.249677 ns

(let [w (first points-record)] (quick-bench (:id
;; Execution time mean : 4.626574 ns

(let [w (first points-map)] (quick-bench (:id w)
;; Execution time mean : 9.730 ns

(let [w (first points-hmap)] (quick-bench (:id w
;; Execution time mean : 14.216794 ns

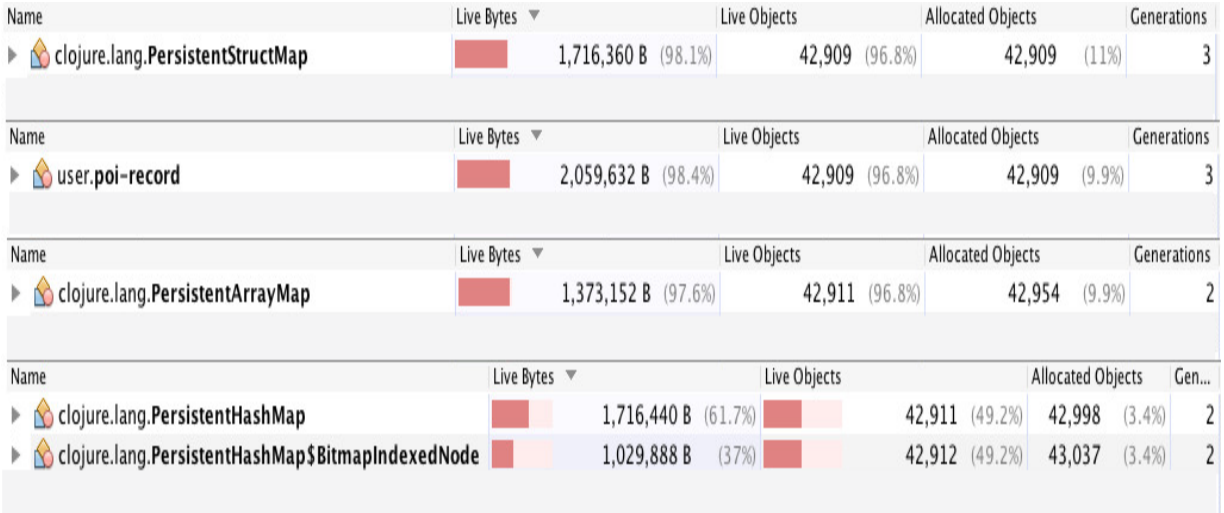
(let [^user.w-record w (first points-record)] (q
;; Execution time mean : 3.612035 ns
```

❶ We can see substantial equivalence between `defrecord` and `struct` when use the faster `accessor`.

❷ `defrecord` also generates a Java type that accepts direct access to the accessors methods on its class. We need to remember to give the compiler a type hint and the results are slightly faster.

Overall `defrecord` is faster at creating and accessing data compared to `defstruct`, with the additional Java interop option to achieve further speed-up. The following picture shows the memory profiling of the 4 types of struct-like types examined so far. The byte count has been obtained loading the same list of waypoints presented before on an empty JVM using memory profiling in VisualVM [179]:

Figure 11.2. A comparison of the memory allocation for the different type of map-like structures.



We can see roughly the same amount of 42,900 allocated objects, corresponding to the amount of waypoints in the file. `clojure.lang.PersistentStructMap` represents `defstruct` with 1,716,360 allocated bytes. `defstruct` is cheaper than the following `user.poi-record` `defrecord` 2,069,632 bytes. The cheapest of the group is array-map with a total of 1,373,152 allocated bytes. The last entry in the group is for `clojure.lang.PersistentHashMap`, the normal hash-map. In this case also `BitmapIndexedNode` instances are allocated, adding up to 2,746,328 bytes for the total memory allocation.

11.1.5 zipmap

`function` `since 1.0`

Listing 11.5. → *Map building, Keys and values*

```
  (zipmap [keys vals])
```

`zipmap` creates a new array-map (or hash-map depending on the number of keys) from a collection of keys and a collection of values. Key-value pairs are formed by making an ordered sequential access to both collection in inputs:

```
(zipmap [:a :b :c] [1 2 3]) ; ❶
;; {:a 1, :b 2, :c 3}

(type (zipmap (range 10) (range 10))) ; ❷
;; clojure.lang.PersistentHashMap
```

❶ `zipmap` accepts two ordered collections and generates a new `array-map`.

❷When the number of keys is sufficiently large, the output self-promotes to `hash-map`.

`zipmap` is useful function to create maps when keys and values are coming from the processing of separated collections.

Contract

input

- "keys" and "vals" are both sequential collection following seq contract for conversion into a sequence (when necessary). They are both required arguments but they can be empty or `nil`.

notable exceptions

- `IllegalArgumentException` if either "keys" or "vals" are not sequential.

output

- returns: the map formed by taking keys and values alternating from "keys" and "vals" respectively. If "keys" contains duplicates, the last key (and corresponding value) overwrites the previous (following assoc semantic). If the number of keys and values are different, `zipmap` stops after reaching the shortest between the two. If either argument is `nil` or empty, it returns an empty map.

Note

Please note that the actual type returned by `zipmap` can be either `clojure.lang.PersistentArrayMap` for smaller maps with less than 10 keys or `clojure.lang.PersistentHashMap` for 10 keys or more. This is consistent with the auto-promoting behavior of `hash-map` related functions such as assoc. Also note that the order in which the map is printed (or iterated) could be different from the order of the initial "keys" and "vals".

Examples

First of all, it's worth remembering that `zipmap` gives no guarantee that the generated map follows the order of the inputs:

```
(zipmap (range 10) (range 10)) ; ❶
;; {0 0, 7 7, 1 1, 4 4, 6 6, 3 3, 2 2, 9 9, 5 5,
```

❶The example shows that even if the keys and values are given in order, the map internal ordering when printed is not determined.

Despite the absence of ordering guarantee, the following expression is always true:

```
(let [m {:a 1 :b 2 :c 3 :d 4 :e 5}]
  (= m (zipmap (keys m) (vals m)))) ; ❶
;; true
```

❶We can always build an equivalent map by calling hash-based on an input map "m" and use `zipmap` to build a new map. The original map "m" and the newly created one are equivalent despite non deterministic ordering.

`zipmap` could be used to generate a map where all values are the same (for example initializing them to "1"):

```
(zipmap ["red" "blue" "green"] (repeat 1)) ; ❶
```

❶Infinite sequences are not a problem for `zipmap`. Here we can see that each color key gets a value of "1".

`zipmap` is useful to build a map when keys and values are not necessarily available at compile time. A typical case are record-oriented data such as a database result set or comma separated value files. In the following example we want to process a CSV file (plain text file that usually contains a header and many rows). We want each row of values to be transformed into a Clojure map using the headers as keys:

```
(require '[clojure.java.io :as io])
(require '[clojure.string :as s])

(def file-content ; ❶
"TITLE,FIRST,LAST,NUMBER,STREET,CITY,POST,JOINED
Mrs,Mary,Black,20,Hillbank St,Kelso,TD5 7JW,01/0
Miss,Chris,Bowie,44,Hall Rd,Sheffield,S5 7PW,01/
Mr,John,Burton,41,Warren Rd,Yarmouth,NR31 9AB,01

(defn split-line [line] ; ❷
  (s/split line #","))

(defn transform [data] ; ❸
  (let [lines (line-seq data)
        headers (split (first lines))]
    (eduction
      (map split)
      (map (partial zipmap headers)) ; ❹
      (rest lines))))
```

```
(with-open [data (io/reader (char-array file-con
  (doall (transform data)))

;; ({"TITLE" "Mrs", "FIRST" "Mary", "LAST" "Blac
;;   "STREET" "Hillbank St", "CITY" "Kelso", "PO
;;   "JOINED" "01/05/2012 12:51"}
;;  {"TITLE" "Miss", "FIRST" "Chris", "LAST" "Bo
;;   "STREET" "Hall Rd", "CITY" "Sheffield", "PO
;;   "JOINED" "01/05/2012 17:02"}
;;  {"TITLE" "Mr", "FIRST" "John", "LAST" "Burto
;;   "STREET" "Warren Rd", "CITY" "Yarmouth", "P
;;   "JOINED" "01/05/2012 17:08"})
```

❶A small portion of the CSV file is simulated in memory using a string.

❷The `split` function contains the logic to split a line strings into multiple strings removing all "," between them. The resulting sequence can be used as values for `zipmap`.

❸`transform` contains the transformation logic to transform "data" (assumed to be a `java.io.Reader` instance`) into a well formed map. The responsibility to close the reader is delegated to the caller.

❹A composition of transducers models the computation with eduction. `zipmap` is part of the last transducer applied to each list of values. The `headers` have been already assigned to a local binding and are ready to be used.

❺We simulate loading from a sample string instead of a file using char-array. To load the data from a file, we would need to change the binding "data" to `(io/reader "somefile.csv")` with no other changes.

See also

- hash-map is the standard map constructor taking any number of arguments (as key-value pairs). `hash-map` can be used to enumerate key-value pairs explicitly or when key and values are coming as a single collection.

- array-map is a specialized version of map that maintains insertion order. It works effectively for a small number of keys (like maps to pass around as argument to functions) but it suffers from inefficient linear access.
- sorted-map offer a way to create ordered maps by key using a comparator.
- into also generates maps provided the input collection has the right format (a collection of key-value pairs).

Performance considerations and implementation details

=> `O(n)` **Linear in the number of keys**

`zipmap` is implemented as loop-recur on the input, requiring n-steps to return the output, where "n" is the shortest between "keys" and "vals".

`zipmap` implementation doesn't use transients, a solution adopted by into for instance. We've already seen an improvement to `zipmap` discussing hash-map performance, in the assumption that "keys" and "vals" are vector. We can now show a faster `zipmap` implementation that works for all kind of arguments:

```
(require '[criterium.core :refer [quick-bench]])

(defn zipmap* [keys vals] ; ❶
  (loop [m (transient {})
         ks (seq keys)
         vs (seq vals)]
    (if (and ks vs)
      (recur (assoc! m (first ks) (first vs))
             (next ks)
             (next vs))
      (persistent! m))))

(let [s1 (range 1000) s2 (range 1000)] (quick-be
;; Execution time mean : 150.549758 µs

(let [s1 (range 1000) s2 (range 1000)] (quick-be
;; Execution time mean : 128.869009 µs
```

❶The new `zipmap*` is a slight modification of the existent implementation to transform the use a transient map.

❷We can see about 30% speed up when using `transient` in this example of 1000 keys.

This implementation of `zipmap*` using transients has been considered for Clojure core but never made it to a Clojure release [180].

[174] Please see the vector chapter for additional information on the implementation details of HAMT

[175] There are several interfaces contributing to define the map abstraction: `clojure.lang.ILookup` is the most general and defines the behavior of looking up a value from a key. `clojure.lang.Associative` adds the behavior of adding a value for a specific key. Finally, `clojure.lang.IPersistentMap` further refines the abstraction adding the option to `dissoc` keys (called `without` on the interface).

[176] A priority queue is a type of data abstraction similar to a queue where each item is also assigned a priority. The priority is used to decide which element is dequeued next. Please see the Wikipedia entry for more information: en.wikipedia.org/wiki/Priority_queue

[177] en.wikipedia.org/wiki/A*_search_algorithm

[178] Red-Black trees are a flexible data structure. Compared to plain binary trees, Red-Black trees offer self adjusting to avoid unbalanced branches. Please have a look at the Wikipedia article to know more en.wikipedia.org/wiki/Red–black_tree

[179] VisualVM is available for free at visualvm.github.io

[180] See: clojure.atlassian.net/projects/CLJ/issues/CLJ-1005

11.2 ACCESSING

11.2.1 keys and vals

Listing 11.6. → *Map access, Key list, Value list*

```
(keys [map])
(vals [map])
```

`keys` and `vals` are two utility functions to retrieve a sequence of keys or values from a map-like type (hash-map, array-map, sorted-map, struct-map or other objects implementing the `java.util.Map` interface):

```
(keys {:a 1 :b 2 :c 3 :d 4 :e 5 :f 6 :h 7 :i 8 :
;; (:e :c :j :h :b :d :f :i :a)

(vals (array-map :a 1 :b 2 :c 3 :d 4 :e 5 :f 6 :
;; (1 2 3 4 5 6 7 8 9)
```

❶ `keys` retrieves the sequence of the keys from a given map. Note that there is no order guarantee about the ordering of the returned keys as the input is of type hash-map.

❷ `vals` retrieves the sequence of values in a map. Note that the sequence is ordered as the input map is an array-map.

`keys` and `vals` don't guarantee ordering of the output which is dependent on the type of the input.

Contract

input

- "map" is the only argument and is mandatory. It can be one of the possible Clojure map types (implementing the `clojure.lang.IPersistentMap` interface), a Java map type (extending `java.util.Map`) or a collection of `java.util.Map$Entry`, a class representing a key-value pair. Empty collection or `nil` is also a possible argument.

notable exceptions

- `ClassCastException` is thrown when the "map" is not one of the allowed map types, or when the collection does not contain `java.util.Map$Entry` instances.

output

- `keys` returns a sequence of the keys in "map" with ordering dependent on the type of "map".
- `vals` returns a sequence of the values in "map" with ordering dependent on the type of "map".

Note

Although `keys` and `vals` could return results in undetermined order (depending on the input type), ordering is consistent when using the same input.

Examples

The contract allows calling `keys` or `vals` on all types of Clojure and Java maps. It also includes the option to use a collection of `java.util.Map$Entry` objects. `java.util.Map$Entry` is the result of processing a map as a sequence for example using map or filter on it:

```
(def filter-odd (filter (comp odd? second) {:a 1

(map type filter-odd)
;; (clojure.lang.MapEntry clojure.lang.MapEntry)
```

```
  (keys filter-odd) ; ❶
  ;; (:a :c)
```

❶We call filter on a map. The map is transformed into a sequence of `java.util.Map$Entry`. We can use `keys` to see which key has an odd number as value.

`keys` and `vals` could be used to extract meaningful information from a configuration map. Typically, a configuration map is read at startup to determine the behavior of the application. We could use it, for example, to implement a simple form of language processing. The following configuration map contains data about the n-grams (combination of words) used to evaluate the tone of a sentence and we are going to use it to measure how much emphasis is used in a sentence:

```
(def matchers     ; ❶
  {"next generation" 10
   "incredible" 10
   "revolution" 10
   "you love" 9
   "more robust" 9
   "additional benefits" 8
   "evolve over time" 8
```

```
   "brings" 7
   "perfect" 5
   "better solution" 7
   "now with" 6})

(defn avg-xf [rf] ; ❷
  (let [cnt (volatile! 0)]
    (fn
      ([] (rf))
      ([result]
       (rf (if (zero? @cnt)
             0.
             (float (/ result @cnt)))))
      ([result input]
       (vswap! cnt inc)
       (rf result input)))))

(defn score [text] ; ❸
  (transduce
    (comp
      (map #(re-find (re-pattern %) text))
      (keep #(matchers %))
      avg-xf)
    + (keys matchers))) ; ❹

(score
  "All-new XT600 brings all the features
   you love about XT300, now with a new design,
```

```
    improved sound and a lower price!")
;; 7.3333335

(score
  "We think this book is a perfect fit for the i
   or seasoned Clojure programmer who wants to u
  how a function (and ultimately Clojure) works"
;; 5.0
```

❶ `matchers` is a map of words into weights. The higher the weight, the more the fragment is important to determine emphasis in a sentence. In a real scenario, the map would be the result of applying natural language processing to some large text.

❷ `avg-xf` is an average transducer. "xf" is a typical suffix to name transducers. This special transducer needs to be the last in a composition chain that only processes numbers. Internally, `avg-xf` maintains a counter of how many items are added up to the final sum and it produces the average on the exit step.

❸ `score` contains the logic to calculate the total score given to a sentence. The text is searched for the fragments and each fragment weights differently based on the content of the `matchers` map.

❹The `keys` of the `matchers` map is the input for transduce. Each key is used in a regular expression and then again to access the weight.

❺A typical advertisement message scores around 7 on our simple emphasis metric.

❻A less emphatic message scores 5.0 points.

See also

- find searches for the given key in a map and returns the corresponding `java.util.Map$Entry` key-value pair object.
- key and val extract the key or the value from a `java.util.Map$Entry` instance (respectively).
- select-keys returns a map containing just the selected keys from another input map.

Performance considerations and implementation details

=> `O(1)` **Constant time**

`keys` and `vals` request the creation of an iterator on top of the content of the given map (or collection when applicable). No items are actually iterated until requested, producing a performance curve that is not dependent on the size of the input:

```
(def big-map (apply hash-map (range 1e7))) ; ❶

(time (first (keys big-map))) ; ❷
;; "Elapsed time: 0.055119 msecs"

(time (last (keys big-map))) ; ❸
;; "Elapsed time: 930.845288 msecs"
```

❶ `big-map` contains 5 million keys.
❷ Retrieving the first key from `keys` is inexpensive operation.
❸ We then force the iteration through all the keys of the map to access the last element. We can see that it takes much more processing time.

While the creation is constant time, the iteration of the sequence returned by `keys` or `vals` is linear in the number of keys present in the map.

11.2.2 find, key and val

`function` `since 1.0`

Listing 11.7. → *Searching, Key-value pair*

```
(find [map key])
(key [e])
(val [e])
```

`find` searches for a key in a map such as those created by hash-map, array-map, sorted-map, struct or objects implementing `java.util.Map` like defrecord:

```
(import 'java.util.HashMap)
(defrecord R [a b c])
```

```
(find (hash-map :a 1 :b 2) :a)
(find (array-map :a 1 :b 2) :a)
(find (sorted-map :a 1 :b 2) :a)
(find (struct (create-struct :a :b) 1 2) :a)
(find (HashMap. {:a 1 :b 2}) :a)
(find (R. 1 2 3) :a)

;; [:a 1] ; ❶
```

❶All the calls to `find` in the example produce the same result of `[:a 1]` when we search for the key `:a`.

`find` also accepts vectors, subvectors and native vectors. In this case `find` looks up an item at the given index:

```
(find [:a :b :c] 1) ; ❶
;; [1 :b]

(find (subvec [:x :a :b :c] 1 3) 1) ; ❷
;; [1 :b]

(find (vector-of :int 1 2 3) 1) ; ❸
;; [1 2]
```

❶ `find` used on a common vector. If an element is found at the index, a `Map$Entry` is returned containing both.
❷ `find` used on a subvector works similarly.
❸Finally, `find` can be used on native vectors built with vector-of.

All `find` examples so far have the return type in common: a `java.util.Map$Entry` instance composed by a key and a value. `key` and `val` are function dedicated to extract the key or the value from a `Map$Entry` without using Java interoperability:

```
(key (first {:a 1 :b 2})) ; ❶
;; :a

(key (find {:a 1 :b 2} :a)) ; ❷
;; :a

((juxt key val) (last (System/getenv))) ; ❸
;; ["JENV_LOADED" "1"]
```

❶ `key` can be used on the elements of the sequence produced by a map. Here we extract the first key-value pair and then the `key`.

❷ `find` similarly produce a `Map$Entry` instance that we can then access to retrieve they `key`.

❸ `System/getenv` returns a map of all the environment variables visible by the running Java virtual machine. We can access the last key-value pair directly using last but here we prefer to retrieve a vector instead of a `Map$Entry`, so we use juxt with `key` and `val`. Please note that the same example could return different results on different machines.

Contract

input

`find`:

- "map" can be an instance of `java.util.Map` (which includes Clojure map types) or an instance of `clojure.lang.Associative` (which includes the vector family of types). It is mandatory argument but it can be `nil` or an empty collection of the proper type.

- "key" can be any type. In case of a vector "key" represents one of the possible indexes.

`key` and `val`:

- "e" stands for "entry" and it must be of type `java.util.Map$Entry`.

notable exceptions

- `NullPointerException` if `key` or `val` are invoked with a `nil` argument.
- `ClassCastException` if "map" is a transient (only for Clojure version < 1.9).

output

`find` returns a `java.uti.Map$Entry` pair containing both the "key" and the value at that "key" if it was found, `nil` otherwise. If "map" is a vector, then any "key" types other than positive integers between 0 and $\overline{2^{32}}$ can generate unpredictable results (see the call-out section for an example).

`key` and `val` returns the key and the value found in "e" respectively.

Note

since Clojure 1.9 release, `find` also accepts transient maps or vectors as arguments returning a `java.util.Map$Entry` instance with the same semantic.

Examples

`find` works similarly to filter by returning the `java.util.Map$Entry` instance for a specific key. We could use it on a list of maps to isolate interesting key-value pairs:

```
(def records ; ❶
  [{(with-meta 'id {:tags [:expired]}) "1311" 'b
   {(with-meta 'id {:tags []}) "4902" 'b "Jane"
   {(with-meta 'id {:tags []}) "1201" 'b "John"

(def ids (keep #(find % 'id) records)) ; ❷

(-> ids first key meta :tags) ; ❸
;; [:expired]
```

❶ `records` is an example list of maps resulting from some data store. The `'id` key for each record additionally contains metadata.

❷ We could use `(map 'id records)` directly, but that would remove the key and the potentially useful metadata. Using find, we can extract the key-value pairs and decide to use the metadata later if we need.

❸ We can access the metadata attached to each key, for example the first one, using `key` and then meta.

`find` and GIGO

Clojure beginners are sometimes confused by the specific meaning of "find" compared to other languages. For instance, they could be surprised to find out that:

```
(find [:a :b :c :d] :c) ; ❶
;; nil
```

❶ The popular meaning of "find" goes against the intuition that this expression should return something different than `nil`.

When `find` is used on vectors, the "key" argument becomes one of the available indexes. As a consequence, "key" should be a positive integer to be meaningful. Passing any other type as an index is possible but would probably result in a `nil`.

There are several cases in the standard library where the documentation does not specify the result of using a function with types it wasn't designed for. There are good reasons for this, such as keeping the documentation compact and essential. A rule of thumb to read Clojure documentation is that if the function does not mention a specific case, that's likely a violation of the functional contract. This is referred to as "GIGO" (colloquially "garbage in, garbage out") and usually produces undetermined results. A glaring example of GIGO for `find` is the following:

```
(def power-2-32 (long (Math/pow 2 32))) ; ❶

(find [1 2 3] power-2-32) ; ❷
;; [4294967296 1]
```

❶ $2^{\overline{32}}$ - 1 corresponds to the maximum number that can fit a 32 integer in Java. To express $2^{\overline{32}}$ and beyond, it needs more than 32 bits. Integers are truncated to access array indexes, so $2^{\overline{32}}$ becomes 0.

❷ `find` effectively retrieves something at index $2^{\overline{32}}$: 0 is the result of truncating $2^{\overline{32}}$ to a Java integer.

Clojure is a language of many trade-offs and GIGO is one of them. Instead of proactively catching and warning about all

possible misuses of a function, Clojure delegates the caller to build a more robust contract barrier driven by actual use cases.

See also

- contains? works similarly to `find` but it returns `true` or `false` to indicate if the element is present or not. It also extends to other non associative types such as sets because it does not verify the presence of the element at a specific index (it wouldn't be possible on `sets` because they are unsorted).
- get is the most flexible and general compared to contains and `find` and works on a larger spectrum of types.

Performance considerations and implementation details

=> `O(log N)` **steps**

For all practical purposes `find` is a constant time operation. `find` is however `O(log 32 N)` for all supported types except `sorted-map`, which is `O(log 2 N)`. The algorithmic family is logarithmic in all cases, but `sorted-map` has a higher constant factor. We did similar considerations for contains? that the reader is invited to review.

`find` implementation is mostly delegated to the Java side of Clojure. `clojure.lang.RT/find` contains a type dispatch and the actual search is delegated to the specific collection implementation.

11.2.3 select-keys and get-in

`function` `since 1.0 (select-keys), 1.2 (get-in)`

Listing 11.8. → *Associative access, Map lookup, Vector lookup*

```
(select-keys map keyseq)

(get-in
  ([m ks])
  ([m ks not-found]))
```

`select-keys` and `get-in` are functions to access keys and values from maps:

```
(select-keys {:a 1 :b 2 :c 3} [:a :c]) ; ❶
;; {:a 1, :c 3}

(get-in {:a 1 :b {:c 3}} [:b :c]) ; ❷
;; 3
```

❶ `select-keys` retrieves keys and related values from a map returning a new map with the selected pairs, if any.

❷ `get-in` only retrieve values but it can follow an arbitrarily nested map multiple levels deep.

`select-keys` and `get-in` also work on vectors (although they are used most frequently on `maps`):

```
(select-keys [:a :b :c :d :e] [1 3]) ; ❶
;; {1 :b, 3 :d}

(get-in [:a :b :c [:d :e]] [3 1]) ; ❷
;; :e
```

❶ `select-keys` on a `vector` returns a map which contains integers as keys and values from the input `vector`. The keys are the indexes passed as input.

❷ `get-in` accepts integer coordinates and traverses a nested vector to retrieve the value at the index.

Contract

input

`select-keys`

- "map" can be any type of map (hash-map, array-map, sorted-map, struct-map, defrecord, Java `Map`) or vector (vector, subvectors, vector-of) or `nil`. The related transient versions are not supported.
- "keyseq" is any seq-able collection or `nil`.

`get-in`

- "m" should be an associative data structure (all `maps` and `vectors` including `transients`). `get-in` also works on sets (ordered or transient). Many other data structures are accepted, including the most used Java collections, although not all of them produce meaningful results.
- "ks" can be any type of sequential collection.
- "not-found" is used as a default when no value is found at key (otherwise `get-in` would simply return `nil` that could be ambiguous in case of the actual presence of `nil` at that key).

notable exceptions

- `IllegalArgumentException` is thrown by `select-keys` on unsupported types.

output

- `select-keys` always returns a map-type (array-map for smaller maps, hash-map when there are over 10 keys). The resulting map contains the matching keys from "keyseq", if any. It returns an empty map if either "map" or "keyseq" are `nil`.

- `get-in` returns the value found by accessing "m" using the keys in "ks", in order. Each key in "ks" extracts a value from a deeper level in "m", if any. Returns `nil` if no value was found or "not-found" if a default is present. Sequential collections like ranges or lists does not work with `get-in`:

```
(get-in '(0 1 2 3) [0]) ; ❶
;; nil
```

❶Accessing a list with `get-in` always returns `nil`, even when the corresponding nth operation works correctly.

Examples

Let's start by noticing the opposite behavior of `select-keys` compared to `get-in` when using an empty `vector` of keys:

```
(select-keys {:a 1 :b 2} []) ; ❶
;; {}

(get-in {:a 1 :b 2} []) ; ❷
;; {:a 1 :b 2}
```

❶ `select-keys` returns an empty map when an empty `vector` is used to select the keys.
❷ `get-in` returns the input instead.

`select-keys` is frequently used to extract a restricted set of key-value pairs from a bigger map:

```
(def large-input-map {:a 1 :b 2 :c 3 :d 4 :e 5})
(select-keys large-input-map [:a :c :e]) ; ❷
;; {:a 1, :c 3, :e 5}
```

❶We are assuming `large-input-map` defines a large map instance with thousands of keys.
❷ `select-keys` extracts a smaller and more manageable map with the only the keys we are interested in.

`select-keys` also preserves metadata:

```
(def m ^:original {:a 1 :b 2}) ; ❶

(meta m)
;; {:original true}

(meta (select-keys m [:a])) ; ❷
;; {:original true}
```

❶"m" is a map created using the literal notation "^:" to attach metadata. The notation implies that the `:original` key is automatically associated a `true` value.

❷The map returned by `select-keys` preserves the given metadata.

`select-keys` can be used with vectors, for example to extract letters from a word or sentence:

```
(let [word "hello"]
  (select-keys (vec word) (filter even? (range (
;; {0 \h, 2 \l, 4 \o}
```

Now let's have a look at how `get-in` can be used to extract values from deeply nested data structures (typically a mix of maps and vectors). Data in Json format, for instance, is used to encode information between data services and tends to be arbitrarily nested. In the following example we receive a list of financial products ordered by lowest legal fees. The list have been translated from Json (JavaScript Object Notation) to Edn (Extensible Data Notation) using one of the several library available and is ready for processing. Here's how input data look like:

```
(def products ; ❶
  [{:product
    {:legal-fee-added {:rate "2%" :period "month
     :company-name "Together"
     :fee-attributes [["Jan" 8] 99 50 13 38 62]
     :initial-rate 9.15
     :initial-term-label {:bank "provided" :form
     :created-at 1504556932727}}
   {:product
    {:legal-fee-added {:rate "4.2%" :period "yea
     :company-name "SGI"
     :fee-attributes [["Mar" 8] 99 50 13 38 62]
     :initial-rate 2.15
     :initial-term-label {:bank "provided" :form
     :created-at 1504556432722}}
   {:product
    {:legal-fee-added {:rate "2.6%" :period "mon
     :company-name "Together"
     :fee-attributes [["Jan" 8] 99 50 13 38 62]
     :initial-rate 5.5
     :initial-term-label {:bank "Chase" :form "A
     :created-at 1504556332211}}])
```

❶ `products` is a small sample of a much larger data feed. It contains an initial vector of products and each product is detailed with additional nested vectors and maps.

We can inspect any of the products for interesting metrics. We can use `get-in` assuming we know what index we want to access from the top level vector:

```
(defn rate-at [products idx] ; ❶
  (get-in products [idx :product :legal-fee-adde

(rate-at products 0)
;; "2%"

(rate-at products 1)
;; "4.2%"
```

❶ `rate-at` takes the vector of products and an index. `get-in` makes access to the element at index "idx" and then extract the "rate" information from the inner maps.

See also

- get does not offer access to nested data structures. Prefer `get` to `get-in` to access values that are not nested.
- keys retrieves the collection of keys from a map instance.

- zipmap creates a new map starting from a collection of keys and a collection of values.

Performance considerations and implementation details

=> `O(n)` **Linear**

Both `select-keys` and `get-in` have linear behavior in the number of selection keys. If you need to select a huge number of keys from an even larger map, you might consider the option of dissoc the unwanted keys instead of selecting the wanted ones.

`select-keys` implementation needs to build the output by gradually introducing keys. Currently it does not take advantage of transients (something which is captured already as enhancement for the next Clojure releases [181]). We could go ahead and produce such implementation and compare it against the current one:

```
(require '[criterium.core :refer [quick-bench]])

(defn select-keys2 [m keyseq] ; ❶
  (with-meta
    (transduce
      (keep #(find m %))
```

```
        (completing conj! persistent!)
        (transient {})
        keyseq)
      (meta m)))

  (let [m (apply hash-map (range 40))] ; ❷
    (quick-bench (select-keys m [0 2 4 6 8 10 12])
  ;;  Execution time mean : 773.492163 ns

  (let [m (apply hash-map (range 40))] ; ❸
    (quick-bench (select-keys2 m [0 2 4 6 8 10 12]
  ;; Execution time mean : 545.102979 ns
```

❶This version of `select-keys` is called `select-keys2` and is based on transduce. It iterates "keyseq" and uses each key to call find against the input map "m". Each entry is then added to the transient results using `conj!`. Results are finally transformed back into a persistent data structure using completing and `persistent!`.

❷The benchmark tests an average size map of 20 keys and a selection of 7 keys.

❸We can see that `select-keys2` has a some advantage over the standard version.

`select-keys2` is faster than normal `select-keys`, especially in the case of even larger maps with larger sets of

keys to select.

[181] clojure.atlassian.net/browse/CLJ-1789

11.3 PROCESSING

11.3.1 assoc, assoc-in and dissoc

`functions` `since 1.0`

Listing 11.9. → *Adding keys, Replacing values, Removing keys*

```
(assoc
 ([map key val])
 ([map key val & kvs]))

(assoc-in
  [map ks val])

(dissoc
  ([map])
  ([map key])
  ([map key & ks]))
```

`assoc`, `assoc-in` and `dissoc` are fundamental operations on maps (they also work on other associative data structures such as vectors).

`assoc` replaces the value of an existing key or inserts a new key if one doesn't exist:

```
(def m {:a "1" :b "2" :c "3"})

(assoc m :b "changed") ; ❶
;; {:a "1" :b "changed" :c "3"}

m ; ❷
;; {:a "1" :b "2" :c "3"}
```

❶ The effect of `assoc` in the presence of an existing key is to replace its value.

❷ As in all other persistent Clojure data structures, `assoc` returns a new instance of the input with changes on top, any references to the same input data structure remain unchanged.

`dissoc` removes one or more keys from the input collection:

```
(def m {:a "1" :b "2" :c "3"})

(dissoc m :a :c) ; ❶
;; {:b 2}
```

❶We use `dissoc` on the original var "m" which contains a reference to a previously created map.

Finally, `assoc-in` is a specialized version of `assoc` that knows how to operate at different level of nesting:

```
(def m {:a "1" :b "2" :c {:x1 {:x2 "z1"}}})

(assoc-in m [:c :x1 :x2] "z2") ; ❶
;; {:a "1" :b "2" :c {:x1 {:x2 "z2"}}}
```

❶The vector `[:c :x1 :x2]` identifies a descending path in the nested data structure "m". The value "z2" is swapped with "z1" after subsequent extractions of the nested maps from the input collection.

Contract

The first input argument "map" is common for the three functions. It's an associative data structure (such that `(associative? map)` is `true`). When `nil` it defaults to the empty map. It's mandatory argument.

assoc input

- "key": for maps, it can be any object while for vectors it has to be an integer. To match an existing key, "key" is compared using = equality semantic. It's mandatory argument.
- "val": is the value for "key". It's mandatory argument.
- "kvs": are any additional key-value pairs. They are optional but if present, they must be an even number of items.

assoc-in input

- "ks" is a sequence of keys (the documentation on the function reports the argument as `[k & ks]` to include destructuring). Each key is used to subsequently look-up at the value that was found at the previous key. When empty or `nil`, it is equivalent to `[nil]` (that is, it accesses a `nil` key at the first level of the input "map"). In case of nested vectors input, "ks" should be a non-empty sequence of integers.
- "val" is the value to be associated to "map".

dissoc input

- "key" can be any object. It’s used to match an existing key using equality comparison, similarly to `assoc`. It is optional argument.
- "ks" are any additional keys to remove from "map".

notable exceptions

- `UnsupportedOperationException` when "kvs" in `assoc-in` is not a sequence of keys. To pass a single key, wrap the key in a sequential collection.
- `IndexOutOfBoundsException` when "key" or any "ks" is more than n+1 where "n" is the max available index of the vector.
- `IllegalArgumentException`: "key" cannot be `nil` attempting to `assoc` on vectors. Similarly, "ks" cannot be an empty vector trying to `assoc-in` on vectors.
- `ClassCastException`: when "key" is not an integer for `assoc` or `dissoc` on vectors.

output

- `assoc` returns "map" with either one or more newly added keys or one or more changes to already existing keys.

- `assoc-in` uses keys in "ks" to descend one or more level of nesting before inserting a new key-value pair or updating an already existing key.
- `dissoc` removes zero or more key-values from "map". No key-value are removed when there are no matching keys or no "key" is passed.

Examples

`assoc` and `dissoc` are popular functions to process maps. They are often seen inside the -> thread first macro along with other map processing functions to form a transformation pipeline:

```
(def m {nil 0 :c 2})

(-> m                  ; ❶
  (assoc :a 1)
  (dissoc nil)
  (update :c inc)
  (merge {:b 2}))
;; {:c 3, :a 1, :b 2} ; ❷
```

❶"m" is a partially built map. We can see that each step executes some operation one the map and the vertical arrangement helps keeping track of the flow.
❷The processed map is returned without any specific key ordering.

`assoc` can be also used to gradually build a map with reduce. This can be useful when the values can be derived from the keys, for example to retrieve data using an "id":

```
(defn lookup [id] ; ❶
  {:index "backup"
   :bucket (rand-int (* 100 id))})

(def request [12 41 11]) ; ❷

(reduce (fn [m item] (assoc m item (lookup item)

;; {12 {:index "backup" :bucket 888}
;;  42 {:index "backup" :bucket 4058}
;;  11 {:index "backup" :bucket 355}}
```

❶ `lookup` simulates the interaction with a service or database to retrieve structured information by id.

❷ Likewise `request` contains a random selection of ids that in a real life application are probably coming from some user interaction.

❸ `reduce` accepts an empty map literal `{}` to start creating new results. `reduce` iterates on the content of the `request` to pass the content of the map so far and the next "item" to `assoc`. The key is the item itself, while the value comes from the `lookup` service.

`assoc-in` tends to be used with deeply nested data structures, especially those mixing maps and vectors that can be traversed by `assoc-in` independently from the nested type:

```
(def articles                    ; ❶
  [{:title "Another win for India"
    :date "2017-11-23"
    :ads [2 5 8]
    :author "John McKinley"}
   {:title "Hottest day of the year"
    :date "2018-08-15"
    :ads [1 3 5]
    :author "Emma Cribs"}
  {:title "Expected a rise in Bitcoin shares"
    :date "2018-12-11"
```

```
    :ads [2 4 6]
    :author "Zoe Eastwood"}])

(assoc-in articles [2 :ads 1] 3) ; ❷

;; [{:title "Another win for India"
;;   :date "2017-11-23"
;;   :ads [2 5 8]
;;   :author "John McKinley"}
;;  {:title "Hottest day of the year"
;;   :date "2018-08-15"
;;   :ads [1 3 5]
;;   :author "Emma Cribs"}
;; {:title "Expected a rise in Bitcoin shares"
;;   :date "2018-12-11"
;;   :ads [2 3 6]               ; ❸
;;   :author "Zoe Eastwood"}])
```

❶ `articles` is a simplified portion of a larger map which contains several level of nesting in the form of vectors (for items that need to be listed) or maps (for items that can be retrieved by key). The ":ads" key contains the position of the ads in the articles, for example after the 2nd, 5th and 8th paragraph.

❷ We want to alter the ads position for the last article, moving one ad up a position after the 3rd paragraph instead of the 4th.

❸ We can see the change from `[2 4 6]` to `[2 3 6]`.

It's worth remembering that `assoc` is also effective option to update vectors and change their content. We used `assoc` in identity for instance, to change the availability of cashiers in a queue modeled as a vector. `assoc` also works as an alternative to conj for those situations where an element could be either replaced or added to the vector:

```
(def pairs [[:f 1] [:t 0] [:r 2] [:w 0]]) ; ❶

(map (fn [[item index :as v]] ; ❷
  (assoc v index item)) pairs)

;; ([:f :f] [:t 0] [:r 2 :r] [:w 0]) ; ❸
```

❶The second element of each `pairs` is a number that can be used as an index for the following `assoc` operation.
❷Using destructuring, we access the first "item", the "index" and the entire pair as "v". We can then `assoc` the pair "v" using the "index" as key and the "item" as value.
❸The result shows the different transformations involved: when the index is 1 the pair is repeated, when the index is 0 nothing happens and when the index is 2 the pair becomes a triplet.

dissoc-in

There is no `dissoc-in` in the standard library, but in this extended section we are going to create one. Differently from `assoc`, which works quite naturally for both maps and vectors, we need a bit more work to `dissoc` a vector, especially to remove an item in the middle. Let's proceed step by step and solve `dissoc-in` for maps first:

```
(def m {:a {:b 2 :c {:d 4 :e 5}}}) ; ❶

(defn dissoc-in [m [k & ks]] ; ❷
  (if ks
    (assoc m k (dissoc-in (get m k) ks))
    (dissoc m k)))

(dissoc-in m [:a :c :d]) ; ❸
;; {:a {:b 2, :c {:e 5}}}
```

❶The input "m" contains several nested maps and we want to `dissoc` just the ":d" key.

❷ `dissoc-in` is similar to the actual `assoc-in` implementation. The strategy is to recur until we reach a level where only one key can be removed. This is where we can use `dissoc`. The modified map needs to be swapped in place of the old ones up the chain, so we `assoc` while moving back up.

❸The result confirms `dissoc-in` is working as expected.

An different and elegant version of `dissoc-in` that works for maps only, is the following:

```
(defn dissoc-in [m ks] ; ❶
  (update-in m (butlast ks) dissoc (last ks)))

(let [m {:a [0 1 2 {:d 4 :e [0 1 2]}]}]
  (dissoc-in m [:a 3 :e]))

;; {:a [0 1 2 {:d 4}]} ; ❷
```

❶This version of `dissoc-in` makes a smart use of of update-in with `dissoc`.

❷We can see that we were able to remove the ":e" key from the inner most map.

But this version of `dissoc-in` fails when the last `dissoc` executes on a vector instead of a map:

```
(let [m {:a [0 1 2 {:d 4 :e [0 1 2]}]}] ; ❶
  (dissoc-in m [:a 3 :e 0]))
;; ClassCastException clojure.lang.PersistentVec
;; cannot be cast to clojure.lang.IPersistentMap
```

❶When the last element to remove from is a vector, this version of `dissoc-in` fails.

To handle vectors similarly to `assoc-in`, we need to treat them differently. The section related to subvec also contains an example function called `remove-at` that can be used to remove an element at the given index in a vector. We can use `remove-at` to `dissoc` from a vector after checking the type of the collection:

```
(def m {:a [0 1 2 {:d 4 :e [0 1 2]}]})

(defn remove-at [v idx] ; ❶
  (into (subvec v 0 idx)
        (subvec v (inc idx) (count v))))

(defn dissoc-in [m [k & ks]]
  (if ks
    (assoc m k (dissoc-in (get m k) ks))
    (cond
      (map? m) (dissoc m k)
```

```
        (vector? m) (remove-at m k)
        :else m))) ; ❷

  (dissoc-in m [:a 3 :e 0]) ;❸
  ;; {:a [0 1 2 {:d 4, :e [1 2]}]}
```

❶ `remove-at` was introduced while discussing subvec. The reader is invited to review subvec if necessary but the mechanism is pretty simple: split the vector so that the unwanted element is left out then merge the vector back together.

❷The only change to `dissoc-in` introduces a condition to verify from what type we need to remove the key from.

❸The result confirms the general design of the function once we nest maps and vectors together.

We can see that the last solution solves all combinations of nesting.

See also

- `assoc!` is an `assoc` version specifically designed for transients.

- get-in and update-in are similar functions to `assoc-in` to handle reading/writing of nested data structures. Differently from `assoc-in`, update-in accepts a function of the old value to produce the new one.

Performance considerations and implementation details

=> `O(log32N)` **map, array-map, struct, vector**

=> `O(log2N)` **sorted-map**

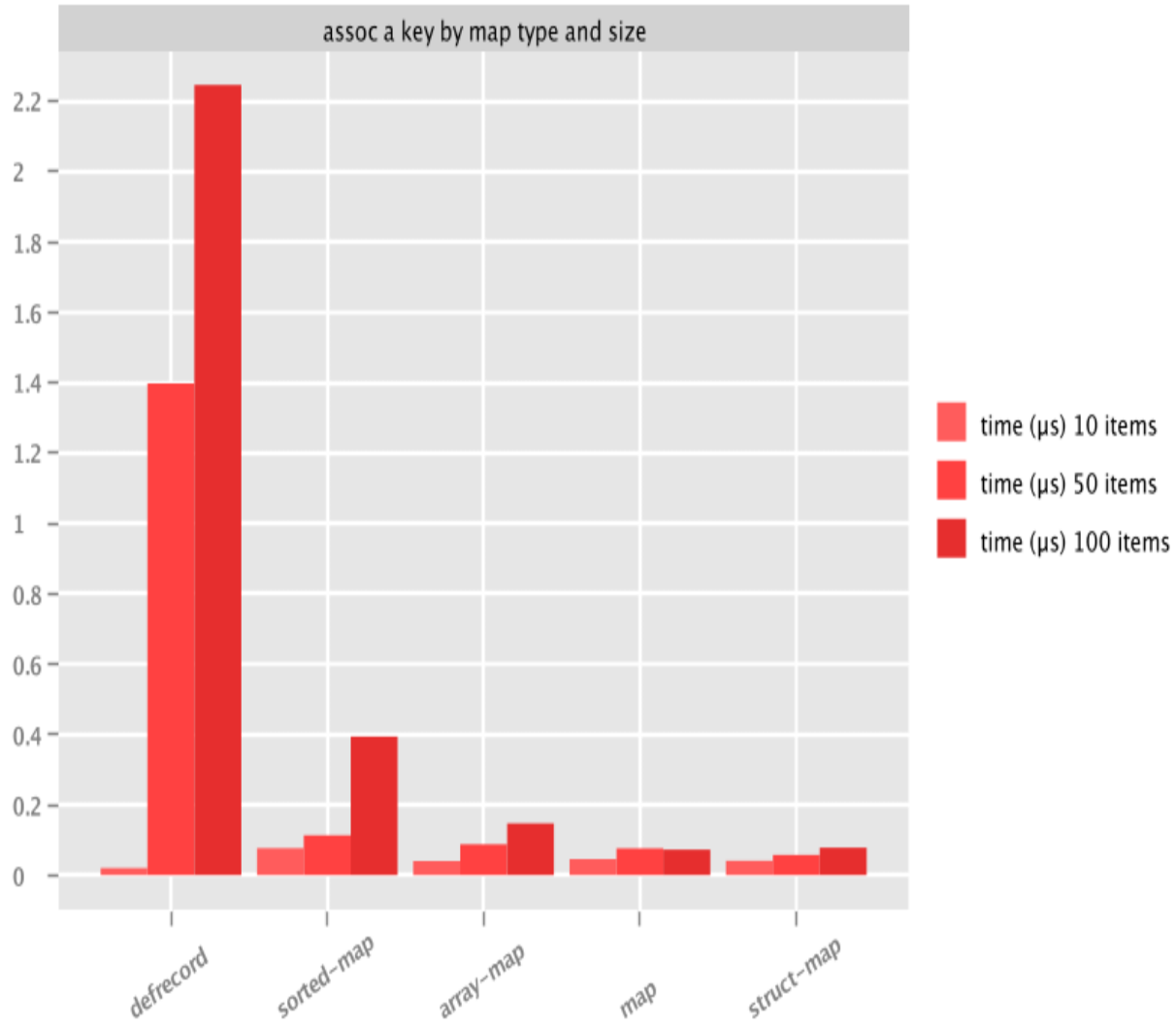
=> `O(n)` **defrecord**

Many Clojure persistent collections, such as maps and vectors are built on HAMTs (Hash Array Mapped Tries), a shallow bit-mapped tree data structure. Most common operation on HAMTs, like traversing or updating are `O(log32N)` with "N" the number of items. The only exception is sorted-map which is instead implemented as binary tree (more precisely, a self-adjusting variant called Red-Black tree). `assoc` for `sorted-map` is still logarithmic but with a constant factor of 2 instead of 32: in practice, there is small (large maps) or no difference (small maps).

`assoc-in` profile is different from `assoc`, because it also has a linear dependency on the length "ks" of the input which dominates the tree traversal. In practice, "ks" normally contains just a few items as it represents the level of nesting of the input.

In the following chart we are going to compare `assoc` on some supported data structures. The benchmark executes `assoc` on different map types sizes: 10 keys, 50 keys and 100 keys. The key is selected so it exists in the current structure, replacing the existing value:

Figure 11.3. `assoc` benchmark for different map types.



We can see that `defrecord` is an order of magnitude slower than other types. This can be explained remembering that `defrecord` builds its associative nature on top of Java class attributes: `assoc` is implemented by matching the required key in a condp condition resulting in a linear behavior. Since

normal usage of `defrecord` includes just a few keys, it shouldn't be a concern for performance.

11.3.2 update and update-in

`function` `since 1.7 (update), 1.0 (update-in)`

Listing 11.10. → *Associative update, Nested updates*

```
(update ; ❶
  ([m k f])
  ([m k f x])
  ([m k f x y])
  ([m k f x y z])
  ([m k f x y z & more]))

(update-in
  ([m ks f & args]))
```

❶The many arities offered by `update` are a performance optimization for the most frequent calls.

`update` and `update-in` are both designed to alter values inside associative data structures. Differently from `assoc` and `assoc-in`, they accept a function of the old value into the new one:

```
(update {:a 1 :b 2} :b inc) ; ❶
;; {:a 1, :b 3}

(update-in {:a 1 :b {:c 2}} [:b :c] inc) ; ❷
;; {:a 1, :b {:c 3}}
```

❶inc is a function of one argument. `update` invokes `int` with the current value of the ":b" key and replace it with the corresponding value increment.

❷ `update-in` takes a list of keys `[:b :c]` as input. Each key is used in turn to get nested associative collections.

Both functions are built on top of assoc, so similar considerations apply for the input types and performance.

Contract

- "m" is an associative data structure. It implies that `(associative? m)` is `true`. It's mandatory argument.

- "k": when "m" is a map type (hash-map, sorted-map, array-map, records, structs or native Java maps), "k" can be any object. When "m" is a vector type (vector, sub-vectors or native-vectors) "k" must be an integer below $\bar{2}32$. To match an existing key, "k" is compared using = equality semantic.
- "ks" is a sequence of keys. Each key follows the same contract as "k", applying to the type of associative data structure found at that level of nesting.
- "f" is a function from a generic object into another generic object. "f" is invoked with the value found at the relevant key.
- "x", "y", "z", "more" and "args" are additional arguments for the function "f" (other than the value found at key which is passed as the first argument).

notable exceptions

- `UnsupportedOperationException` happens if you forget to wrap "ks" in a list or vector when using `update-in`.
- `IllegalArgumentException` using a key that is not an integer to access a vector, including when the vector is nested with `update-in`.

output

- `update` returns a new version of the input "m" with the value "v" indicated by "k" replaced with the result of invoking `(f v)`. If "k" does not exist and "m" is a map, "k" is created and the result of `(f nil)` is used as the new value. If "m" is a vector, then "k" needs to be within 0 and `(dec (count m))`.
- `update-in` returns a new version of the input "m" with the value "v" indicated by "ks" (at any level of nesting) replaced with the result of invoking `(f v)`. "ks" is interpreted so the first key is used to get a value in "m", the second key is used to get a value in the previous value and so on, up to the last key. "ks" is subject to the same considerations as `update`.

Examples

Differently from `assoc`, `update` and `update-in` can be used to "upsert" new keys (update or insert instead of replace or insert). This model applies well to counters, or in general any update that requires the presence of the previous value. fnil is often seen in action with `update` to provide a default when there is no key:

```
(def words {"morning" 2 "bye" 1 "hi" 5 "gday" 2}

(defn insert-word [w words]
  (update words w (fnil inc 0))) ; ❷

(insert-word "hello" words)
;; {"morning" 2, "bye" 1, "hi" 5, "gday" 2, "hel
```

❶We have a list of words and their frequencies.

❷`update` is used to increment the counter of an already existing word or insert a new one. inc would fail with a `NullPointerException` in case the word is not in the list. By wrapping `inc` with `fnil` we prevent invoking the function with a `nil` value. We pick a default of "0" instead, which is then passed to `inc` to be incremented.

❸We can see "hello" appearing in the words map with a frequency of 1.

`update` on vectors follows a similar pattern, allowing the addition of a new element at the tail when "k" is equal to the length of the vector:

```
(update [:a :b :c] 3 (fnil keyword "d")) ; ❶
;; [:a :b :c :d]
```

```
(update [:a :b :c] 4 (fnil keyword "d")) ; ❷
;; IndexOutOfBoundsException
```

❶The index "3" is allowed for `update`, but it's out of bound for `[:a :b :c]` where the only available index are 0 (:a), 1 (:b) and 2 (:c).

❷The addition of a new item in the vector only works right after the last item. Trying to access past the size of the vector causes `IndexOutOfBoundsException`.

In the following example, "products" is a map with a key dedicated to store how many items are in stock. When a product is sold, we want to decrease the corresponding number:

```
(def products ; ❶
  {"A011" {:in-stock 10
           :name "Samsung G5"}
   "B032" {:in-stock 4
           :name "Apple iPhone"}
   "AE33" {:in-stock 13
           :name "Motorola N1"}})

(defn sale [products id] ; ❷
  (update-in
    products
```

```
      [id :in-stock]
      (fnil dec 2)))

  (get-in ; ❸
    (sale products "B032")
    ["B032" :in-stock])
  ;; 3
```

❶"products" is a sample version of a bigger list coming from a database. Products are located by key and the value contains the details, including how many items are still in stock.
❷We use `update-in` to access a product by key and then updating the ":in-stock" key decreasing its content by 1.
❸`get-in` is useful to focus on the newly updated `:in-stock` key.

`update` (but also `assoc` and their `*-in` variants) are often seen in conjunction with swap! to change the `atom` during a compare and swap (CAS) transaction [182]. With reference to the previous example, we could now allow concurrent sales:

```
  (def products ; ❶
    (atom {"A011" {:in-stock 10
                   :name "Samsung G5"}
           "B032" {:in-stock 4
                   :name "Apple iPhone"}
```

```
          "AE33" {:in-stock 13
                  :name "Motorola N1"}}))

(defn total-products [products] ; ❷
  (reduce + (map :in-stock (vals products))))

(total-products @products) ; ❸
;; 27

(defn sale! [products id] ; ❹
  (swap! products
         update-in
         [id :in-stock]
         (fnil dec 2))
    products)

(defn sale-simulation! [ids] ; ❺
  (dorun (pmap (partial sale! products) ids)))

(sale-simulation! ["B032" "AE33" "A011" "A011" "A
;; nil

(total-products @products) ; ❻
;; 21
```

❶ `products` is an in-memory view of the warehouse state. It wraps the state in atom to enable concurrency control.

❷ `total-products` is used to count how many products, in total, are still in stock. It uses reduce to count over all the ":in-stock" keys.

❸ We can see that we have 27 items in the warehouse before sales are taking place.

❹ `sale!` is similar to the function we defined before, but it is now side-effecting the state of the products. swap! has similar arguments to `update-in`: it takes a function of an old value into a new value, plus any additional arguments to be passed to `update-in` when it's executed. The effect is similar to the thread first macro ->: `products` is placed as the first argument of `update-in` and the vector of keys and `fnil` default follows.

❺ `sale-simulation!` simulates the interaction of multiple clients concurrently. We use pmap to start several sales in parallel. Note the conventional "!" bang symbol at the end of the function name to denote a side-effecting function.

❻ If we check the number of total products, we can see that is consistently selling the right amount of products. By using swap! we make sure that the operation always decrease the stock number by one, as the transaction repeats if the

operation ends and another thread was able to decrease the number at the same time.

See also

- fnil was mentioned a few times in the examples. It's not an alternative to `update`, but it works well to provide a default for missing keys.
- assoc produces a similar effect to `update` where the new value does not depend on the old. Same applies to `assoc-in` compared to `update-in`.
- get and get-in retrieve the value without changes.

Performance considerations and implementation details

=> `O(log32N)` **map, array-map, struct, vector**

=> `O(log2N)` **sorted-map**

=> `O(n)` **defrecord**

`update` and `update-in` are built on top of assoc. Supported types are the same, as well as the performance profile. The reader is invited to visit assoc's performance section for additional details.

Like assoc, `update` and `update-in` perform well on average map operations, also considering they don't operate on sequences preventing unwanted linear behavior.

11.3.3 merge and merge-with

`function` `since 1.0`

Listing 11.11. → *Map merging, Map processing*

```
(merge [& maps])

(merge-with [f & maps])
```

`merge` and `merge-with` are specifically dedicated to join one or more maps together:

```
(merge {:a 1 :b 2} {:c 3 :d 4}) ; ❶
;; {:a 1, :b 2, :c 3, :d 4}
```

❶A simple use of `merge` to join two maps (two keys each).

While `merge` just "replaces over" an existing key at destination in case of a clash, `merge-with` allows to specify what to do if the same key is found more than once:

```
(merge-with + {:a 1} {:b 2 :a 10}) ; ❶
;; {:a 11, :b 2}
```

❶ `merge-with` allows to decide what should be done in case the target key already exists at destination. In this case the current value of the key ":a" is added to the new value.

Contract

input

- "maps" is any number of map types (hash-map, sorted-map, array-map, records, structs but not native `java.util.HashMap`). The first map in maps is the "target" map and determines the return type. Sequential types (like vectors or lists) are also accepted, but they don't merge correctly, with one or more of them forming a nested levels. Types other than maps are therefore not supported.
- "f" is mandatory argument for `merge-with`. If a key already exists in the results, "f" is invoked with two arguments, the current value of the key and the new value of the key. The result of "f" is used in place of the old value.

notable exceptions

- `ClassCastException` when the target map is not a `clojure.lang.IPersistentCollection`.
- `IllegalArgumentException` when the target map is followed by elements that are not pairs, for example `(merge {} [1 2 3])`. This is because `merge` is attempting to transfer the sequential collection into the target map but there is a key ("3") that is missing a value.

output

- With no arguments or `nil` arguments: returns `nil`.
- With a single argument other than `nil`: returns the argument itself.
- In all other cases, `merge` attempts to transfer the content of `(rest maps)` into `(first maps)` copying each key-value pair. In case of conflicting keys, the value corresponding to the last key to enter the output overwrite the previous value.
- `merge-with` output is the same as `merge` but the key conflict resolution is handled by the custom function "f".

The output type is the same as the target `(first maps)`, with the following rules:

- When the target is an array-map the output can auto-promote to hash-map depending on the number of keys (usually beyond 10).
- When the target is a sorted-map, keys from `(rest maps)` need to be comparable with keys in `(first maps)`.

The following example clarifies the last rule:

```
(def sorted-map-of-keywords
  (sorted-map :z 3 :f 5 :c 4))

(def map-of-ints
  (hash-map 1 "a" 2 "b" 5 "c"))

(merge sorted-map-of-keywords map-of-ints) ; ❶
;; ClassCastException clojure.lang.Keyword canno

(def map-of-keywords
  (hash-map :a 1 :b 4 :e 4))

(merge sorted-map-of-keywords map-of-keywords) ;
;; {:a 1, :b 4, :c 4, :e 4, :f 5, :z 3}
```

❶ The interface contract of `sorted-map` (the first argument to `merge`) takes precedence over the other map types. We are unable to merge the maps in this example because an integer key cannot compare to a keyword key.

❷ We can successfully merge a `sorted-map` with a `hash-map` provided the key types are compatible (both keywords in this example).

Examples

Along with the examples presented below, the book shows the use of `merge` and `merge-with` in other chapters. The reader is invited to take a quick look at:

- fn contains an example of `merge` on an arbitrary number of input maps. apply can be used with `merge` if the input is in the form of a list of maps.
- ->> contains a similar example to merge parameters after parsing them from an URL.
- `merge` and `merge-with` are also typical in combining algorithms. fold contains several example of combining function based on `merge-with` using reducers/monoid like `(r/monoid merge (constantly {}))`.

If the values in the input are all vectors, we can use into to merge vectors belonging to the same key. This can be useful to group values together:

```
(let [m1 {:id [11] :colors ["red" "blue"]} ; ❶
      m2 {:id [10] :colors ["yellow"]}
      m3 {:id [31] :colors ["brown" "red"]}]
  (merge-with into m1 m2 m3))

;; {:id [11 10 31], ; ❷
;;  :colors ["red" "blue" "yellow" "brown" "red"
```

❶Note that ":id" and ":colors" for all maps "m1", "m2", "m3" needs to be vectors.

❷The result of `(merge-with into)` is a map with the same keys and the union of all values from all maps for that key.

The following `merge-into` function can be used to pull keys up from an arbitrarily nested map. In this example we receive some product data containing several levels of nested maps and vectors. We want to select some of the deeply nested keys and merge them at the top of the map:

```
(defn merge-into [k ks] ; ❶
  (fn [m]
```

```
    (merge
      (get m k {})
      (select-keys m ks))))

(def product-merge ; ❷
  (merge-into :product [:fee-attribute :created-

(def product ; ❸
  {:fee-attributes [49 8 13 38 62]
   :product {:visible false
             :online true
             :name "Switcher AA126"
             :company-id 183
             :part-repayment true
             :min-loan-amount 5000
             :max-loan-amount 1175000}
   :created-at 1504556932728})

(product-merge product) ; ❹
;; {:visible false,
;;  :online true,
;;  :name "Switcher AA126",
;;  :company-id 183,
;;  :part-repayment true,
;;  :min-loan-amount 5000,
;;  :max-loan-amount 1175000,
;;  :created-at 1504556932728}
```

❶ `merge-into` generates a function of the map to be transformed. The function merges the selection of keys "ks" into the top level key "k" of map "m".

❷ A function generated by `merge-into` is assigned to a var. We call this specialization `product-merge`.

❸ Here's an example of product data. The `:fee-attributes` and `:created-at` keys belong to the `:product` but they appear in the top layer.

❹ `product-merge` transforms the input into a new map which contains everything that was previously under `:product`, including `:fee-attributes` and `:created-at` (that were not).

In the next example we are going to use `merge-with` to implement addition on complex numbers. Complex numbers have a real and imaginary part that we can implement as keys in a record. `defrecord` provides a syntactically appealing form to implement complex numbers, including the option to expand the set of available operations in the future:

```
(defprotocol IComplex ; ❶
  (sum [c1 c2]))

(defrecord Complex [re im]
  IComplex
```

```
    (sum [c1 c2] (merge-with + c1 c2))) ; ❷

(sum (Complex. 2 5) (Complex. 1 3))
;; #user.Complex{:re 3, :im 8}
```

❶The protocol `IComplex` defines a suitable interface to describe complex numbers operations (the "I" suffix means "Interface").

❷The sum of a complex number defined as Clojure `defrecord` can be implemented using `merge-with`.

Multi-type merge

When merging maps, `merge` and `merge-with` offer two strategies to deal with different values for the same key: `merge` simply replaces the old value with the new value while `merge-with` accepts a function of the old and new value for additional customization. If the values in all maps are homogeneous (for example all values are vectors) then we can use `merge-with` with `into` to collapse all values into the same key. We saw an example of this above.

But what if values are not `vectors` or they are different types? We could write a custom function for `merge-with` with conj to store new values in a vector:

```
(let [m1 {:a 1 :b 2}
      m2 {:a 'a :b 'b}
      m3 {:a "a" :b "b"}]
  (merge-with (fn [v1 v2]
                (if (vector? v1) ; ❶
                  (conj v1 v2)
                  [v1 v2]))
```

```
               m1 m2 m3))

;; {:a [1 a "a"], :b [2 b "b"]} ; ❷
```

❶This is a condition to understand if we can accumulate on the old value (because it's already a vector?) or create a new vector to start accumulating.

The approach seems to work, but fails if the values are already vectors:

```
(let [m1 {:a [1 3] :b 2} ; ❶
      m2 {:a 'a :b 'b}
      m3 {:a "a" :b "b"}]
  (merge-with (fn [v1 v2]
                (if (vector? v1)
                  (conj v1 v2)
                  [v1 v2]))
              m1 m2 m3))

;; {:a [1 3 a "a"], :b [2 b "b"]} ; ❷
```

❶The input map "m1" contains a vector as value for the ":a" key.

❷The key ":a" in the results should contain the original `[1 3]` vector but the inner nesting has been erroneously removed.

To fix the special treatment of vectors values, we need a way to distinguish between "user values" and "special values". The wrapping vector that we add has a special meaning that should

not be confused with any type of user values. We could use metadata to mark our vectors differently from any other vector:

```
(let [m1 {:a [1 3] :b 2}
      m2 {:a 'a :b 'b}
      m3 {:a "a" :b "b"}]
  (merge-with (fn [v1 v2]
                (if (:multi (meta v1)) ; ❶
                  (conj v1 v2)
                  ^:multi [v1 v2]))    ; ❷
              m1 m2 m3))

;; {:a [[1 3] a "a"], :b [2 b "b"]} ; ❸
```

❶Instead of checking for the presence of a vector?, we check for the presence of custom metadata that only our function can create.

❷If the metadata are not present, we know for sure that the incoming value needs wrapping in a new vector with the additional metadata to distinguish it from any other `vector`.

❸We can see that the results are now the expected ones.

See also

- group-by can be used to collapse a list of collections into a map based on some feature of the data. While `group-by` is more about grouping by key, `merge` is about how values are combined. Compared to `merge-with`, `group-by` collects the entire input as values of the resulting map which can be unwanted feature for large inputs. With `merge-with` there is more flexibility to decide what part of the input ends up in the final result.

Performance considerations and implementation details

=> `O(n)` **Linear in "n" number of keys**

Both `merge` and `merge-with` need to cycle through the keys of the input. The number of processing steps increasing linearly with the number of keys in the input map (or aggregate keys in multiple maps inputs).

In absolute terms, `merge` can be costly on larger inputs. Some improvement is possible by implementing `merge` with transients:

```
(require '[criterium.core :as c])

(let [m1 (apply hash-map (range 2000))
```

```
        m2 (apply hash-map (range 1 2001))]
    (c/quick-bench (merge m1 m2)))           ; ❶
  ;; Execution time mean : 221.025373 µs

  (defn merge* [m & maps]
    (when (some identity maps)
      (persistent!
        (reduce conj! (transient (or m {})) maps))

  (let [m1 (apply hash-map (range 2000))
        m2 (apply hash-map (range 1 2001))]
    (c/quick-bench (merge* m1 m2)))          ; ❷
  ;; Execution time mean : 162.887879 µs
```

❶We create a `hash-map` with 1000 keys and another one of the same size but different keys, without overlap.

❷We can see a visible speed up by using transients.

We can use `transient` for `merge-with` with similar results:

```
  (let [m1 (apply hash-map (range 2000))
        m2 (apply hash-map (range 1 2001))]
    (c/quick-bench (merge-with + m1 m2))) ; ❶
  ;; Execution time mean : 304.863164 µs

  (defn merge-with* [f & maps] ; ❷
    (when (some identity maps)
```

```
     (letfn [(merge-entry [m [k v]]
               (assoc! m k
                 (if-not (= ::none (get m k ::non
                   (f (get m k) v)
                   v)))
             (merge-into [m1 m2] ; ❹
               (reduce merge-entry (transient (or
       (persistent! (reduce merge-into maps)))))

(let [m1 (apply hash-map (range 2000))
      m2 (apply hash-map (range 1 2001))]
  (c/quick-bench (merge-with* + m1 m2))) ; ❻
;; Execution time mean : 220.885976 µs
```

❶First of all, let's benchmark normal `merge-with` using 2 maps with 1000 non-overlapping keys.
❷transients require `assoc!` instead of normal assoc.
❸Note that we can't use contains? directly on `transient` because is not supported. We can use get instead, provided we use a sentinel value like `::none` to establish if the key exists (with a potential value of `nil`).
❹The inner function `merge-into` decides when to switch from normal `hash-map` to transient maps.
❺The call to `persistent!` happens at the end of the computation.
❻We can see that `merge-with*` is visibly faster than the standard `merge-with`.

11.3.4 reduce-kv

`function` `since 1.4`

Listing 11.12. → *Associative reduce, hash-map reduce*

```
  (reduce-kv [f init coll])
```

`reduce-kv` is a specialized version of reduce for associative data structures (the contract section is more precise about which types are allowed). Processing of maps and vectors

involves keys, not just values, so it makes sense to have a specialized version of `reduce` that behaves accordingly. Here's a comparison between the two styles:

```
(reduce ; ❶
  (fn [m [k v]] (assoc m k (inc v)))
  {}
  {:a 1 :b 2 :c 3})
;; {:a 2, :b 3, :c 4}

(reduce-kv ; ❷
  (fn [m k v] (assoc m k (inc v)))
  {}
  {:a 1 :b 2 :c 3})
;; {:a 2, :b 3, :c 4}
```

❶ Normal reduce for a map-type like array-map or hash-map requires a reducing function that understands the next item is an entry formed by a key and a value. We use destructuring to get them individually. Under the hood, `reduce` is forced into transforming the associative data structure into a sequence first.

❷ `reduce-kv` is dedicated to associative data structures, so the reducing function takes 3 arguments instead of 2: the accumulator, the key and the value. What is not visible though, is that `reduce-kv` takes a faster path to iterate the input.

`reduce-kv` allows the input data structure to provide a specific implementation through the dedicated `clojure.core.protocols/IKVReduce` protocol. Compatible Clojure data structures already implement `IKVReduce` and other associative types could extend the same abstraction. We are going to see how to extend `reduce-kv` to other map-like types in the examples.

Contract

input

- "f" needs to be a function of 3 arguments that is expected to return the accumulation of the results so far. It is mandatory argument.
- "init" is the value that is passed to "f" as first argument during the first call. This is usually an empty collection (not necessarily associative) that `reduce-kv` is supposed to fill with results.
- "coll" can be one of the supported types or `nil`. Supported types are those implementing `clojure.lang.IKVReduce` or `clojure.lang.IPersistentMap`. The former interface indicates a custom implementation that usually performs better. The following table contains a summary of what is supported and does not throw exception:

Table 11.1. Summary of the collection types supported by `reduce-kv` (including `nil` as an exceptional case).

Type	Function Involved	Interface
`PersistentArrayMap`	array-map	`IKVReduce`
`PersistentHashMap`	hash-map	`IKVReduce`
`PersistentTreeMap`	sorted-map	`IKVReduce`
`PersistentVector`	vector	`IKVReduce`
`PersistentStructMap`	struct	`IPersistentMap`
Record	defrecord	`IPersistentMap`
`nil`	`nil`	`nil`

Note

associative collections like subvectors or `java.util.Map` are not supported (while in general they are by other functions in this chapter).

---

notable exceptions

- `IllegalArgumentException` is thrown when there is no `reduce-kv` implementation for a specific type (for example list).

output

- returns: the result of applying "f" on "init" and the first item in "coll", followed by applying "f" again on the previous result and the second item in "coll" and so on, until there are no more items.

Examples

`reduce-kv` is useful to process maps, for example to update all values, keys or both. We could for example transforms keys representing environment variables into keywords following normal Clojure naming conventions:

```
(def env ; ❶
  {"TERM_PROGRAM" "iTerm.app"
   "SHELL" "/bin/bash"
   "COMMAND_MODE" "Unix2003"})

(defn transform [^String s] ; ❷
  (some-> s
    .toLowerCase
    (.replace "_" "-")
    keyword))

(reduce-kv ; ❸
  (fn [m k v] (assoc m (transform k) v)) {} env)

;; {:term-program "iTerm.app", ; ❹
;;  :shell "/bin/bash",
;;  :command-mode "Unix2003"}
```

❶ `env` contains a list of environment variables that conventionally are uppercased and underscored.

❷ The transformation we want to apply is a composition of `toLowerCase`, `replace` underscores with dashes and move to keyword. some-> is a good idea just in case the transformation receives `nil`, which generates `NullPointerException` with string manipulation functions. The type hint contribute to reinforce the expectations around the type of the input, as well as improving performance.

❸ The `reduce-kv` call is relatively straightforward. We are going to move the entries from the input map into another map transforming the keys while doing so.

❹ We can see the transformed keys as we would expect them formatted in standard Clojure code.

Environment variables like the ones from the example can be retrieved using the Java interoperability call `(System/getenv)`. The type returned by `(System/getenv)` is `java.util.Collections$UnmodifiableMap` for which there is no default reduce-kv implementation:

```
(reduce-kv
  (fn [m k v] (assoc m (transform k) v))
  {}
```

```
    (System/getenv)) ; ❶
  ;; IllegalArgumentException No implementation of
```

❶ `reduce-kv` doesn't have a specific implementation for the kind of Map returned by `(System/getenv)`. `transform` was defined in the previous example.

To solve the problem we need to provide a suitable implementation of `reduce-kv` for the kind of map returned by `(System/getenv)`. It turns out `java.util.Collections$UnmodifiableMap` extends `java.util.Map` so in solving the problem for environment variables, we also solve the problem for many other Java map types:

```
(import 'java.util.HashMap)

(extend-protocol clojure.core.protocols/IKVReduc
  java.util.Map ; ❷
  (kv-reduce [m f init]
    (let [iter (.. m entrySet iterator)] ; ❸
      (loop [ret init]
        (if (.hasNext iter) ; ❹
          (let [^java.util.Map$Entry kv (.next i
              (recur (f ret (.getKey kv) (.getVa
          ret)))))
```

```
(reduce-kv
  (fn [m k v] (assoc m (transform k) v))
  {}
  (System/getenv)) ; ❺

;; {:jenv-version "oracle64-1.8.0.121",
;;  :tmux "/private/tmp/tmux-502/default,2685,2"
;;  :term-program-version "3.1.5",
;;  :github-username "reborg"
;;  ...}

(reduce-kv
  (fn [m k v] (assoc m (transform k) v))
  {}
  (System/getProperties)) ; ❻

;; {:java.vm.version "25.121-b13",
;;  :java.specification.name "Java Platform API S
;;  :java.io.tmpdir "/var/folders/25/T/",
;;  :java.runtime.name "Java(TM) SE Runtime Envi
;;  ...}
```

❶The protocol dedicated to `reduce-kv` extensions is `clojure.core.protocols/IKVReduce`.

❷We want to extend the protocol to the `java.util.Map` type using the extend-protocol dedicated function.

❸The fastest way to iterate a Java Map is to get the `entrySet` (usually cached internally) which provides an `iterator` instance. The iterator returns all entries sequentially.

❹The main design of the iteration consists of a loop-recur instruction until the iterator `.hasNext` returns `true`. Each iteration we read the content of the entry invoking "f" with the result so far, the key and the value. The result of invoking "f" is conventionally the accumulation of the next result, which is used for the next iteration.

❺The invocation of `reduce-kv` on `(System/getenv)` now results in the polymorphic call to the relevant protocol extension, producing the expected results (truncated for brevity).

❻Similarly, other Java types implementing the `java.util.Map` interface are now returning results, like this call to `(System/getProperties)` for example.

`reduce-kv` and the `reduced` convention **

Another aspect to mention regarding `reduce-kv` is that, like reduce, it understands the reduced signaling system. This is an

useful mechanism to stop the reduction in case the needed results are already available, preventing any further processing:

```
(reduce-kv
  (fn [m k v]
    (if (> k 2)
      (reduced m) ; ❶
      (assoc m k v)))
    {}
    [:a :b :c :d :e])
;; {0 :a, 1 :b, 2 :c} ; ❷
```

❶We choose an arbitrary condition based on one of the keys. When the condition is `true`, we return a "reduced" result and we skip the related assoc operation, signaling the fact that we want to terminate the reduction. reduce knows how to interpret the signal and does not proceed any further.

❷As expected, the resulting map is missing keys above the number 2.

When implementing extensions to `reduce-kv` (similarly to reduce) we should always remember to obey the signaling convention and handle the reduced case. If we don't, then we might ignore the signal and waste additional computation. Our

`reduce-kv` extension for `java.util.Map`, for example, is not behaving correctly:

```
(import 'java.util.LinkedHashMap)

(reduce-kv
  (fn [m k v]
    (if (= k :abort)
      (reduced m) ; ❶
      (assoc m k v)))
  {}
  (LinkedHashMap. {:a 1 :abort true :c 3})) ; ❷

;; ClassCastException clojure.lang.Reduced canno
```

❶If the input contains a special key (here called ":abort") we'd like for the reduction process to stop. The fact that we are using a `LinkedHashMap` guarantees that the entries are retrieved in insertion order, so we expect that no other keys after ":abort" is found to appear in the results.

❷We can see that using reduced is not handled by our previous protocol extension. Instead of stopping after reaching a `reduced` item, we pass that to `assoc` which fails.

To solve the problem, we need to enhance the protocol extension for `java.util.Map` types to handle an element wrapped in a reduced object:

```
(extend-protocol clojure.core.protocols/IKVReduc
  java.util.Map
  (kv-reduce [m f init]
    (let [iter (.. m entrySet iterator)]
      (loop [ret init]
        (if (.hasNext iter)
          (let [^java.util.Map$Entry kv (.next i
                ret (f ret (.getKey kv) (.getVal
            (if (reduced? ret) ; ❶
              @ret
              (recur ret)))
          ret)))))

(reduce-kv
  (fn [m k v]
    (if (= k :abort)
      (reduced m)
      (assoc m k v)))
  {}
  (LinkedHashMap. {:a 1 :abort true :c 3}))

;; {:a 1} ; ❷
```

❶ We repeat the same protocol extension from before, but this time, we check to see if we are passed a reduced? element and in that case, stop recursion. We also need to unwrap the reduced item with the "@" (dereference) reader macro.
❷ We can see that the result only contains keys up to the point where the ":abort" request was found.

See also

- reduce is the model `reduce-kv` is inspired by. `reduce` should be considered for all other non associative data structures.
- reduced are the function implementing the signaling mechanism that was discussed also for `reduce-kv`.

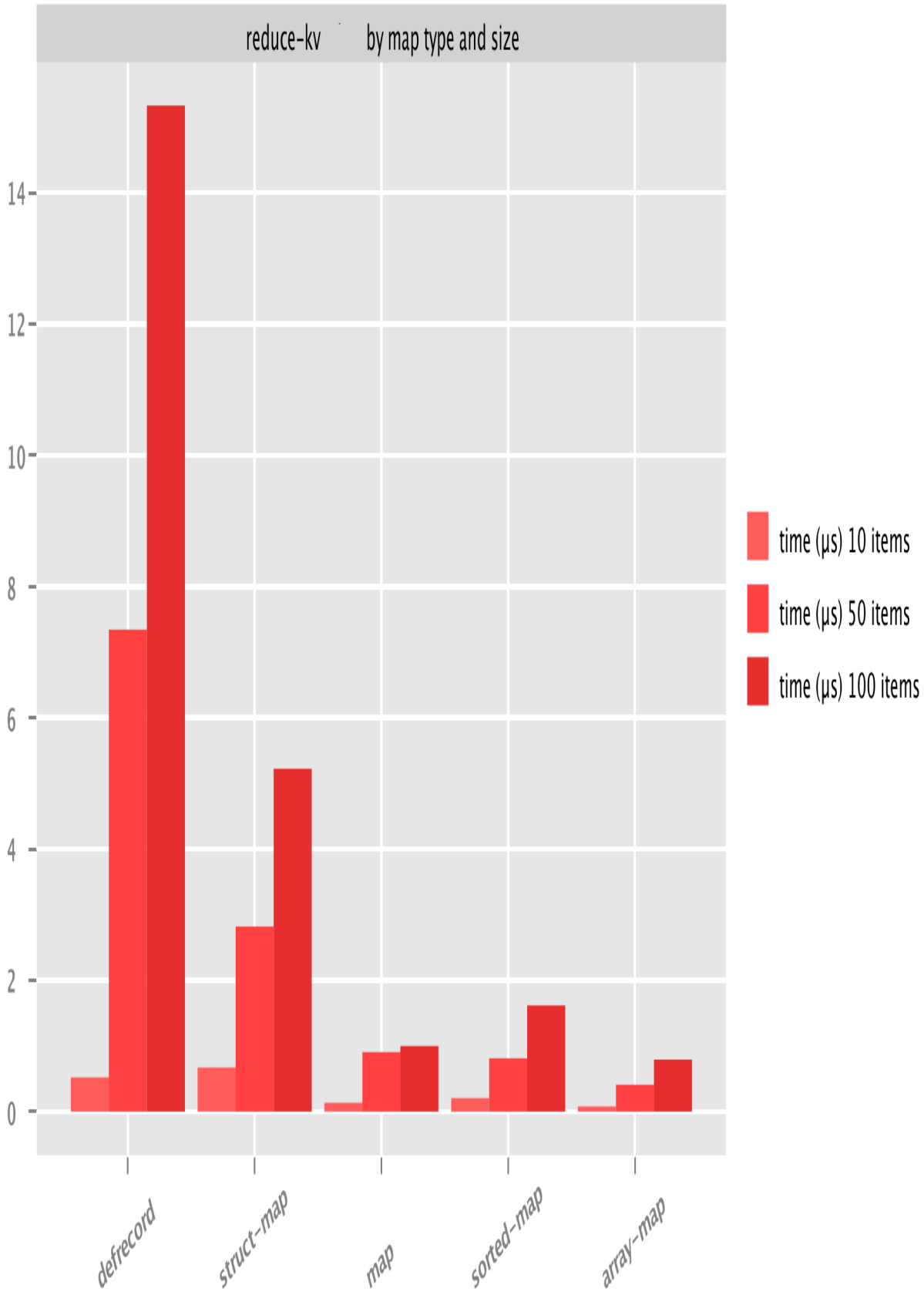
Performance considerations and implementation details

=> `O(n)` **linear in n number of entries**

Performance considerations for `reduce-kv` are very similar to reduce: `reduce-kv` performs linearly in the number of entries (key-value pairs in a map or items in case of vector). In absolute terms, the way reduction is performed depends on the type of input collection. The following chart shows how `reduce-kv` performs with compatible types and different input sizes. The

reducing function is `(constantly nil)` so no other time is considered other than the pure iteration.

Figure 11.4. reduce-kv benchmarks.


reduce-kv by map type and size
14
12
10
8
6
4
2
0
defrecord
struct-map
map
sorted-map
array-map
time (µs) 10 items
time (µs) 50 items
time (µs) 100 items

The chart shows that records are an order of magnitude slower (but they are also unlikely to contain that many keys). The fastest of the benchmark is array-map but the other types (except record) follow closely.

Memory allocation depends on the reducing function: the generation of a map from another map where all the keys are preserved is linear in space. `reduce-kv` (like `reduce`) is not lazy and unless interrupted by reduced, it processes all items independently from how many are consumed downstream.

[182] Compare and swap (CAS) is the semantic used by Clojure concurrency primitives: atom, ref and agent. It consists of attempting an unsynchronized mutating operation followed by a check on the original value. Only if the original value is still intact, the new value is committed to the `atom`.

11.4 UTILITY FUNCTIONS

A few functions that can be used with map types are not included in the core namespace. This doesn't mean that they should not be used, but just that they were created while solving other problems. The following functions can be used effectively on maps and have been added to this chapter accordingly:

- keywordize-keys and stringify-keys live in `clojure.walk` namespace. They transform the keys of an arbitrarily nested map into keywords or strings, respectively.
- rename-keys from `clojure.set` allows to rename the keys in a map.
- map-invert again from the `clojure.set` namespace, swaps keys and values in a map.

In this section we are giving a quick summary of their features and relevant information.

11.4.1 clojure.walk/keywordize-keys and clojure.walk/stringify-keys

`keywordize-keys` and `stringify-keys` accept a map input (all types implementing `clojure.lang.IPersistentMap` which are hash-map, array-map, sorted-map, records and sructs) and transform the type of the keys into `keyword` or `string` respectively:

```
(require '[clojure.walk :refer [keywordize-keys

(keywordize-keys {"a" 1 "b" 2}) ; ❷
;; {:a 1 :b 2}
```

```
(stringify-keys {:a 1 :b 2}) ; ❸
;; {"a" 1 "b" 2}
```

❶ `keywordize-keys` and `stringify-keys` need an explicit require instruction to become available in the current namespace.

❷The input `array-map` has strings as keys and they are transformed into keywords.

❸The input `array-map` has now keywords as keys and they are transformed into strings.

`keywordize-keys` and `stringify-keys` are restricted to only apply the transformation to `string` keys (`keywordize-keys`) or `keyword` keys (`stringify-keys`). Other key types result in no changes:

```
(keywordize-keys {1 "a" 2 "b"}) ; ❶
;; {1 "a", 2 "b"}

(stringify-keys {1 "a" 2 "b"}) ; ❷
```

❶There are no changes to the input map as the keys are not of type `string`.

❷Likewise `stringify-keys` is not transforming the keys if the key type is not `keyword`.

Perhaps the most interesting feature of `keywordize-keys` and `stringify-keys` is that they operate at any level of nesting. We could use `keywordize-keys` for example, to transform keys in a deeply nested data structures:

```
(require '[clojure.walk :refer [keywordize-keys]

(def products ; ❶
  [{"type" "Fixed"
    "bookings" [{"upto" 999 "flat" 249.0}]
    "enabled" false}
   {"type" "Variable"
    "bookings" [{"upto" 200 "flat" 20.0}]
    "enabled" true}])

(keywordize-keys products)

;; [{:type "Fixed" ; ❷
;;   :bookings [{:upto 999 :flat 249.0}]
;;   :enabled false}
;;  {:type "Variable"
;;   :bookings [{:upto 200 :flat 20.0}]
;;   :enabled true}]
```

❶ `products` contains several level of nesting of vectors and hash-maps.

❷ We can see that `keywordize-keys` walks the `products` recursively and keyword keys appear also in nested structures.



Warning

both functions accept all map types but they always return array-map (or hash-map for bigger inputs, following array-map auto-promoting rules).

11.4.2 clojure.set/rename-keys

`rename-keys` name is quite self-explanatory. Given an input map and a dictionary map it renames keys according to the content of the dictionary:

```
(require '[clojure.set :refer [rename-keys]]) ;

(rename-keys {:a 1 :b 2 :c 3} {:a "AA" :b "B1" :
;; {"AA" 1, "B1" 2, "X" 3}
```

❶ `rename-keys` is a public function inside the `clojure.set` namespace which is part of the standard library.
❷ Each matching key in the first map is replaced by the corresponding value in the second map.

`rename-keys` is an useful function for simple renaming of keys, for example while transforming one data format to another. The renaming is limited to the first level and does not loop over nested maps (if any). `rename-keys` has some restrictions dependent on the type of input map to consider. `rename-keys` always return `array-map` or `hash-map` independently from the input type.

```
(defrecord A [a b c])

(rename-keys (A. 1 2 3) {:a :y :b :z}) ; ❶
;; {:c 3, :y 1, :z 2}

(type *1) ; ❷
;; clojure.lang.PersistentArrayMap
```

❶ After creating a simple `record` "A" of 3 fields ":a", ":b" and ":c", we ask `rename-keys` to change some keys.
❷ The operation is successful but the type of the results is array-map and not a record.

We can use `rename-keys` on sorted-map provided we the replacements are the same type of the existing keys:

```
(rename-keys (sorted-map :a 1 :b 2 :c 3) {:a :z}
;; {:b 2, :c 3, :z 1}

(rename-keys (sorted-map :a 1 :b 2 :c 3) {:a 9})
;; ClassCastException clojure.lang.Keyword canno
```

❶ We use a `sorted-map` as input for `rename-keys`. The operation completes successfully and the returned type is again a `sorted-map`.

❷ We need to be careful and use the proper replacement type as the new key type needs to be comparable with the old.

Finally, a note on structs. Since `rename-keys` first removes the target key using dissoc, `structs` will throw error, as removing keys that are part of the definition is not allowed:

```
(rename-keys (struct (create-struct :a :b :c) 1
;; RuntimeException Can't remove struct key
```

❶ `rename-keys` doesn't work on structs.

11.4.3 clojure.set/map-invert

`map-invert` swaps keys and values in a map:

```
(require '[clojure.set :refer [map-invert]]) ; ❶

(map-invert {:a 1 :b 2}) ; ❷
;; {1 :a, 2 :b}
```

❶ `map-invert` lives in the `clojure.set` namespace.
❷ A simple input map is inverted with values becoming keys and keys becoming values.

If there are identical values in the input map, then the last instance of the value wins. However, ordering of keys in a hash-map is undetermined, so is the key that ultimately appears in the results:

```
(def m (hash-map :y 0 :y 0 :d 0 :k 0 :w 0 :i 0
                 :f 0 :a 0 :n 0 :v 0 :s 0 :w 0))

(map-invert m) ; ❷
;; {0 :a}
```

❶The `hash-map` "m" contains only zero values.
❷The key ":a" is not the last during creation of the map, however it becomes the last key to enter the map as value of the key "0".

Let's have a look at some corner cases on empty collections:

```
(map-invert {})
(map-invert [])
(map-invert ())
(map-invert "")
;; {} ; ❶
```

❶All these examples return the same empty map, despite the input type is not necessarily a map type.

In general, all map types can be inverted. In the following examples, all `map-invert` return a map with the same keys and values, however the order might be different:

```
(map-invert (hash-map :a 1 :b 2 :c 3)) ; ❶
;; {3 :c, 2 :b, 1 :a}
(map-invert (array-map :a 1 :b 2 :c 3))
;; {1 :a, 2 :b, 3 :c}
(map-invert (sorted-map :a 1 :b 2 :c 3))
;; {1 :a, 2 :b, 3 :c}
```

```
(map-invert (struct (create-struct :a :b :c) 1 2
;; {1 :a, 2 :b, 3 :c}

(defrecord A [a b c])
(map-invert (A. 1 2 3))
;; {1 :a, 2 :b, 3 :c}

(import 'java.util.HashMap)
(map-invert (HashMap. {:a 1 :b 2 :c 3})) ; ❷
;; {2 :b, 3 :c, 1 :a}
```

❶The input `hash-map` returns items in an undetermined order, which is visible in the result of `map-invert`.

❷Another example of undetermined ordering of keys.

`map-invert` is useful to enable bidirectional map lookup, for example to implement a simple scrambling algorithm to obfuscate text:

```
(def scramble-key ; ❶
  {\a \t \b \m \c \o \d \l
   \e \z \f \i \g \b \h \u
   \i \h \j \n \k \s \l \r
   \m \a \n \q \o \d \p \e
   \q \k \r \y \s \f \t \c
   \u \p \v \w \w \x \x \j
```

```
    \y \g \z \v \space \space})

(defn scramble [text scramble-key] ; ❷
  (apply str (map scramble-key text)))

(defn unscramble [text scramble-key] ; ❸
  (apply str (map (map-invert scramble-key) text

(scramble "try to read this if you can" scramble
;; "cyg cd yztl cuhf hi gdp otq"

(unscramble "cyg cd yztl cuhf hi gdp otq" scramb
;; "try to read this if you can"
```

❶ `scramble-key` is a map from char to char pairing up every letter in the alphabet with another random one.

❷ `scramble` uses the `scramble-key` to obfuscate the content of a sentence.

❸ `unscramble` can revert the obfuscation effect using the randomized letter (what appears as the value of `scramble-key`) as a key. We can quickly obtain this effect by using `map-invert`.

❹ The obfuscated text is transformed back into the correct clear text.

This concludes the chapter dedicated to maps and the functions processing them.

11.5 SUMMARY

Maps are an important asset in every developer's tool belt. Clojure provides an arsenal of functions to work with maps, including specific support for reducing (reduce-kv) or folding (reducers/fold). This chapter contains several examples on how to use them effectively. The next chapter is dedicate to another invaluable work-horse: vectors.

12 VECTORS

12.1 THANKS TO RACHEL BOWYER FOR CONTRIBUTING THIS CHAPTER.

Clojure's vector is one of the standout features of the language: performant, immutable and with a convenient literal syntax. Back in 2009 when Clojure 1.0 was launched, there was nothing else quite like it; and it set Clojure apart from earlier LISPs. Since then, other functional languages such as Scala [183] and Haskell [184] have added their own immutable vector.

Clojure's vector stores elements sequentially, using zero-based indexing. It provides efficient read, write and append making direct access to the index of the element. It also supports efficient delete from the tail of the vector (with pop), but not from other locations (for which the best workaround is to use subvec).

The literal syntax for a vector consists of merely enclosing a space separated list of the elements within a pair of square brackets:

```
[:a :b :c] ; ❶
;; => [:a :b :c]
```

❶Simple creation of a new vector by enclosing elements in square brackets.

As well as being a data structure, a vector is also a function that looks up a value. It takes one argument, the zero based index, and if it is out of range then an `IndexOutOfBoundsException` is thrown:

```
(ifn? []) ; ❶
;; true

([:a :b :c] 2) ; ❷
;; :c

([:a :b :c] 3) ; ❸
;; IndexOutOfBoundsException
```

❶Asking if `[]` can be invoked as a function returns `true`.
❷Vectors are indexed starting from zero. Here we are making access to the last item in the vector.
❸The element at index 3 was not present in the vector.

```
(get [:a :b :c] 2)
;; :c

(nth [:a :b :c] 2)
;; :c

(assoc [:a :b :c] 2 :d)
;; [:a :b :d]

(conj [:a :b :c] 3.1 :e)
;; [:a :b :c 3.1 :e]

(pop [:a :b :c])
;; [:a :b]

(peek [:a :b :c])
;; :c
```

To determine if a sequence is a vector, we can use vector?. The contains? function does not work the way that might be expected compared to the use with other data structures. When used on vectors, it works positionally without comparing values. `contains?` determines whether a vector has an element allocated at a specific index. If the value is what you're after, one solution is to use the underlying java `.contains` function instead.

```
(contains? [1 2 :a :b] 3) ; ❶
;; true

(contains? [1 2 :a :b] :a)
;; false

(.contains [1 2 :a :b] 3) ; ❷
;; false

(.contains [1 2 :a :b] :a)
;; true
```

❶The contains? function from the standard library is used to verify if there is an element at index 3, not if the value "3" is present in the vector.

❷Note that a dot "." is prefixed to this "contains" invocation, indicating the call to the instance method "contain" of the Java type used to implement a Clojure vector.

Vectors are "seq-able" (that is, `(instance? clojure.lang.Seqable [])` is `true`) so they work with any function expecting a sequence, although they tend to be less efficient than functions that operate directly on vectors:

```
(first [:a :b :c]) ; ❶
```

```
;; :a
```

❶Vectors support the sequential interface, enabling seamless usage of sequential functions.



Warning

Although vectors can behave like a sequence and thus used as an argument to seq, they are not a sequence type themselves: seq? returns `false`. The implications for vectors is that all sequence operations work on vector, but they are implicitly transformed into a sequence.

There are specialized versions of map and filter called mapv and filterv. `mapv` and `filterv` returns a vector rather than a sequence. Vectors also offer a way to be efficiently reversed with rseq. Likewise, subvec could be considered a specialized version of rest.

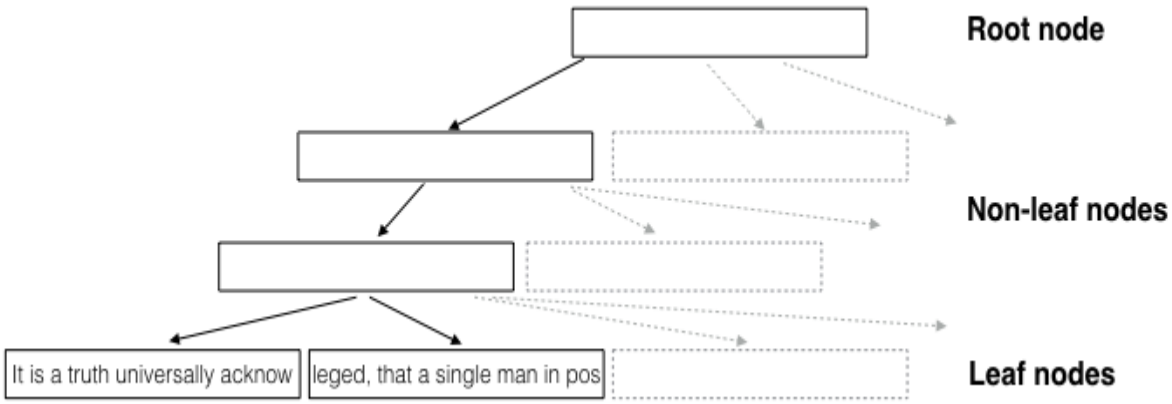
Persistent Vectors and RBB Trees

Clojure's vectors are based on a mutable implementation which is not accessible from the outside world. However, from the point of view of the client of the data structure, the vector

appears to be immutable. This is achieved by keeping old versions of the vector in memory while they are still referenced by the running application. Such data structures are known as "persistent".

Although it is straightforward to create a persistent vector by making a copy of it each time it is modified, this is very inefficient in time and space. The solution that Clojure adopts, inspired by Phil Bagwell's paper "Ideal Hash Trees" [185], is to use a balanced tree with the leaf nodes containing the elements in the vector. The next figure shows part of a persistent vector containing the text to "Pride and Prejudice".

Figure 12.1. Part of a persistent vector containing the text to "Pride and Prejudice"



When an element is modified, the leaf node, the root node and all the nodes on the path from the leaf node to root have to be copied, but crucially none of the other nodes. Further, the tree

uses 32 way branching at each non-leaf node (and leaf nodes that contain up to 32 values), leading to a very shallow tree. The entire text of "Pride and Prejudice" fits into a tree that is only 4 levels deep! Therefore, although technically modifying an element in a vector with n elements is O(log n) in time, in practice the behavior is almost constant time. This is because the trees are so shallow. Even a vector containing a billion elements would only be 6 levels deep as $\log_{32}(1 \text{ billion}) < 6$ [186].

In September 2011, Bagwell, along with Rompf, extended the Clojure vector implementation to create Relaxed Radix Balanced Tree (RRB Trees). These offer concatenation and insert-at in O(log n) rather than O(n) time. RRB Trees remain an active on going research topic [187].

- `clojure.lang.PersistentVector` is the most common. It's created by the vector literal syntax `[]`, the function vector or vec.
- `clojure.lang.APersistentVector$SubVector` is the type of "subvector" and is returned by subvec.
- `clojure.core.Vec` is the type of what is colloquially known as a "gvec", the primitive-type based vector returned by vector-of.

- `clojure.lang.MapEntry` is the tuple (a vector of two elements) which is formed iterating a map as a sequence.

Unfortunately these names are not particularly descriptive. In particular a "gvec" is also a persistent vector. However, these names are used by the Clojure community when discussing about them instead of the lengthy class type.

The following table is a quick summary of the behavior of the different types of vector. To learn more about their characteristics, please see the entries for the related functions.

Created with:	Supports transient?	Stores nil?	Stores mixed data types?	Space efficient?	Efficient construction?
vector	Yes	Yes	Yes	No	Yes
vector-of	No	No	No	Yes	No
subvec	No	Depends on underlying vector	Depends on underlying vector	Yes	Yes
`(first {:a 1})`	No	Yes	Yes	Yes	Yes

[183] See: news.ycombinator.com/item?id=6446922

[184] see hackage.haskell.org/package/persistent-vector-0.1.1/docs/Data-Vector-Persistent.html

[185] P. Bagwell. Ideal Hash Trees. Technical report, EPFL, 2001. lampwww.epfl.ch/papers/idealhashtrees.pdf

[186] For a more detailed but still accessible explanation of how vectors work in Clojure please see J. N. L'orange' blog hypirion.com/musings/understanding-persistent-vector-pt-1

[187] The following are a few links related to RBB-trees that might be of interest for further explorations: RRB-Trees paper by Bagwell infoscience.epfl.ch/record/169879/files/RMTrees.pdf, a Clojure implementation github.com/clojure/core.rrb-vector and Improving RRB-Tree Performance hypirion.com/thesis.pdf

12.2 FUNCTIONS THAT OPERATE ON VECTORS

There are overridden versions of get, nth, assoc, conj and pop that work with vectors. Elements can be retrieved from a vector using at any position with get or nth. Elements can be written using assoc and appended with conj. The element at the end of the vector can be removed using pop and retrieved with peek.

Here is some basic usage example of the most commonly used functions:

12.3 THE FOUR TYPES OF VECTORS

Clojure has not one, but four types of vectors. The end-user of the language doesn't necessarily need to know about the specific types, but sometimes their differences come into play:

12.4 VECTOR

`function` `since 1.0`

Listing 12.1. → *Creation of data type*

```
(vector
  ([])
  ([a])
  ([a b])
  ([a b c])
  ([a b c d])
  ([a b c d e])
  ([a b c d e f])
  ([a b c d e f & args]))
```

The function `vector` creates a vector (one of the main Clojure data structures) whose elements consist of its arguments. The order of elements in the vector is the same as the order of arguments given to the function:

```
(vector :a :b :c) ; ❶
;; [:a :b :c]
```

❶The `vector` function creates a new vector with the given items as content.

`vector` produces the same output as the reader literal `[]` (a pair of square brackets enclosing other forms or constants):

```
[:a :b :c] ; ❶
;; [:a :b :c]
```

❶The vector reader literal `[]` works similarly to the `vector` function.

If the elements are known in type and number at the time of writing the code, then the literal `[]` is normally used instead of the function as it is shorter and idiomatic [188]. The function is still used when it's not possible to write each argument

explicitly, for example to collect all the arguments of a function declaration:

```
(defn var-args [a b & all]
  (apply vector a b all)) ; ❶

(var-args :a :b :c)
;; [:a :b :c]
```

❶apply is used to collect the sequence of arguments and pass them to `vector`. Note how it wouldn't be possible to use the vector literal syntax in this case. This is because all arguments to the function "var-args" have been gathered up into "all" as a single element of type list.

Contract

input

- `vector` accepts zero arguments, returning an empty vector.
- "a", "b", "c", "d", "e", "f" (and any additional arguments) can be of any type including `nil` or other vectors.

Note

Although all provided arities are going to create the same kind of `vector`, the first 7 (from 0 to 6 arguments) are slightly faster (a common pattern in the Clojure standard library). See the call-out section further below for details.

---

output

- An object of type `clojure.lang.PersistentVector` containing the given arguments in order. If there are zero arguments, then an empty persistent vector is returned.

Examples

In the introduction, we've seen the difference between using `vector` as a function compared to the syntax literal `[]`. Another case in which `vector` needs to be used instead of the square brackets is when it is part of a function literal invocation. The syntax `#()` expands into a function call that would be unsuitable to contain a vector literal as first form. The following `longest-palindrome` example illustrate the use of `vector` from inside a function literal:

```
(def palindromes ["hannah" "kayak" "civic" "deif

(defn longest-palindrome [words]
  (->> words
    (filter #(= (seq %) (reverse %))) ; ❶
    (map #(vector (count %) %))       ; ❷
    (sort-by first >)
    first))

(macroexpand '#([(count %) %])) ; ❸
;; (fn* [p1] ([(count p1) p1]))

(longest-palindrome palindromes)
;; [7 "deified"]
```

❶We reverse the word to compare it with itself. This quick solution to the problem of finding a palindrome is simple enough for this example, but there are more efficient alternatives (see rseq).

❷The similar but incorrect syntax using a vector literal `(map ([(count %) %]))` would throw `ArityException` at runtime. This is because the function literal `()` expands its content into a function call (see below).

❸This macroexpansion shows why the vector literal inside the function literal doesn't work. The expansion shows that the generated fn uses the vector as a function, invoking it without the mandatory argument.

The next example shows how the data belonging to two real estate systems can be joined together before being compared. This could happen before releasing a new version of the data, so the new version can be regression tested against the old:

```
(require '[clojure.data :refer [diff]])

(def old-data
  [{:summary "Bijou love nest" :status "SSTC"}
   {:summary "Country pile" :status "available"}

(def new-data
  [{:summary "Bijou love nest" :status "SSTC"}
```

```
    {:summary "Country pile" :status "SSTC"}])

(doseq [[old-instruction new-instruction] ; ❶
        (map vector old-data new-data)]   ; ❷
  (let [[only-first only-second common] (diff ol
    (when (or only-first only-second)
      (println "Differences:" old-instruction ne

;; Differences: {:summary Country pile, :status
;;              {:summary Country pile, :status
;; nil
```

❶Vector destructuring is used to retrieve the two real estate instructions.

❷To allow doseq to process two elements at a time, firstly `map vector` is used to create a sequence of pairs.

Using map with `vector` is typical for creating the transpose of a matrix [189]:

```
(def m [[1 2 3] ; ❶
        [4 5 6]
        [7 8 9]])

(apply map vector m) ; ❷
;; ([1 4 7]
```

```
  ;;  [2 5 8]
  ;;  [3 6 9])
```

❶A matrix is two-dimensional structure with rows and columns. We can encode a matrix as a nested `vector`. Each inner `vector` represents a row of the matrix.

❷map iterates on each row in "m" taking items from them at the same index. `vector` in this case receives three arguments to recreate the new transposed row.

Clojure faster "arities"

`vector` comes with many, apparently overlapping, arities. So when you use doc at the REPL you are presented with:

```
  (doc vector) ; ❶

  ;; ([] [a] [a b] [a b c] [a b c d] [a b c d e] [
  ;; Creates a new vector containing the args.
```

❶The function `doc` is readily available at the Clojure REPL. If that wasn't the case in your environment, you can require it with `(require '[clojure.repl :as repl])` and then `(repl/doc vector)`.

Many other functions in the standard library follow a similar pattern. You might wonder why the signature isn't simply `(defn vector [& args])` and the reason is performance. There are two aspects connected to arities and performances:

- In general, the presence of a "& args" arity implies that the function implementation contains somewhere an iteration over the variable number of arguments. This iteration is more expensive than directly accessing arguments.
- Depending on the function implementation, the different arities could simply share the same underlining code or have completely different ways to achieve the final result.

`vector` in particular, takes advantage of knowing the number of arguments at compile time (0 to 6) by creating the tail of the persistent vector directly. The variable arity case instead, makes use of a more flexible "create and expand" loop. Here's a simple benchmark that shows the speed gain:

```
(require '[criterium.core :refer [quick-bench]])

(quick-bench (vector 1 2 3 4 5 6))
;; Execution time mean : 8.281921 ns
```

```
(quick-bench (vector 1 2 3 4 5 6 7)) ; ❶
;; Execution time mean : 214.268271 ns
```

❶The additional element added to the vector slows down creation about 20 times.

The behavior up to the first 6 arguments covers a wide range of cases where small vectors are created as intermediate results. It also covers the possible need to model data in an application as small groups: tuples, triplets and so on.

In summary: if some business problem needs to process large amounts of data as small vectors of various sizes, processing is faster if vectors are up to 6 items long. If the application requires bigger chunks, you could use the following macro to generate additional `vector` handlers:

```
(defmacro defnvector [n] ; ❶
  (let [args (map #(symbol (str "x" %)) (range n
    `(defn ~(symbol (str "vector" n)) [~@args] [

(macroexpand '(defnvector 7)) ; ❷
;; (def vector7 (fn ([x0 x1 x2 x3 x4 x5 x6] [x0 

(defnvector 7) ; ❸
```

```
  (defnvector 8)
  (defnvector 9)
  (defnvector 10)

  (quick-bench (vector7 1 2 3 4 5 6 7)) ; ❹
  ;; Execution time mean : 8.577230 ns
```

❶The `defnvector` macro generates a function definition in the current namespace with defn. It takes a number of arguments `n` that is used to generate parameters and building the vector.

❷We can see how the macro works by expanding it with macroexpand.

❸New vector builders need to be declared explicitly.

❹The benchmark shows that the speed of the generated functions is comparable with the original vector implementation up to 6 arguments.

See also

- vec creates a persistent vector from a seqable collection. Prefer `vec` if the input content for the newly created vector is coming from an already existing sequence.

- vector-of creates a vector specifying a primitive Java type. Use `vector-of` if space efficiency is the biggest concern.
- make-array creates a Java array. Use a Java array if the performance benefits of a mutable data structure are required, or when interfacing Java code.
- assoc can be used to swap an element at the specified index in the vector. Although mainly used with maps, assoc on vectors can be quite invaluable to quickly change the content of the vector.
- replace allows multiple changes at once on vectors using a dictionary-like structure. All items at the corresponding index will be swapped with the related change.

Performance considerations and implementation details

=> `O(n log(n))` **time, with n number of arguments**

=> `O(n)` **space**

`vector` and vec have similar performance characteristics so this section applies to both, except where noted, with "n" referring to the number of arguments or "coll" when `(vec coll)` is called.

As described in the introduction to this chapter, persistent vectors are implemented as 32 way trees. The last `(inc (mod (dec n) 32))` elements in the vector are stored separately. These elements are known as the "tail" of the vector. Storing the tail separately is what allows constant time access to the last element in the vector with peek for example.

In the implementation of `vector` for 32 items and up, an empty mutable transient object is created Java side. Elements are then appended one at a time. Below 32 items the tail has space and adding a new item completes in constant time. But in general, for every 32^{nd} element the tail overflows and has to be added to the tree. This takes $O(log_{32}n)$ operations leading to creation of a persistent vector being $O(n\ log(n))$. For all the practical purposes though, the performance can be considered linear.

vec implementation is also based on the type of the input collection. `vec` shares the same implementation of `vector` except in two cases:

- If the input collection is a "reducible" (implementing the `clojure.lang.IReduce` interface) then `vec` ultimately delegate to the input collection reduce implementation.

- If the input collection is a Java non-primitive array of 32 or fewer elements, then an empty vector is **not** created but the tail is instead, creating an alias to the Java array.

The following benchmark examines how the speed of `vector` and vec changes as n increases:

```
(require '[criterium.core :refer [benchmark]])

(defmacro b [expr] ; ❶
  `(first (:mean (benchmark ~expr {}))))

(def results ; ❷
  (doall
    (for [i (range 10)]
      (let [num-elements (* (inc i) 100000)
            data (range num-elements)]
        [(b (vec data))
         (b (apply vector data))]))))
```

❶The `b` macro extracts just the mean execution time from the benchmark, removing all other statistics that are not relevant for this test.

❷The loop produces benchmarks for `vector` and `vec` while increasing the number of items from 100,000 to 1,000,000 by 100,000 increments.

"results" contains a collection of pairs: the first item is the elapsed for `vec` in nanoseconds and the second is the same for `vector`. To make it easy to visualize, we can produce a chart using the following:

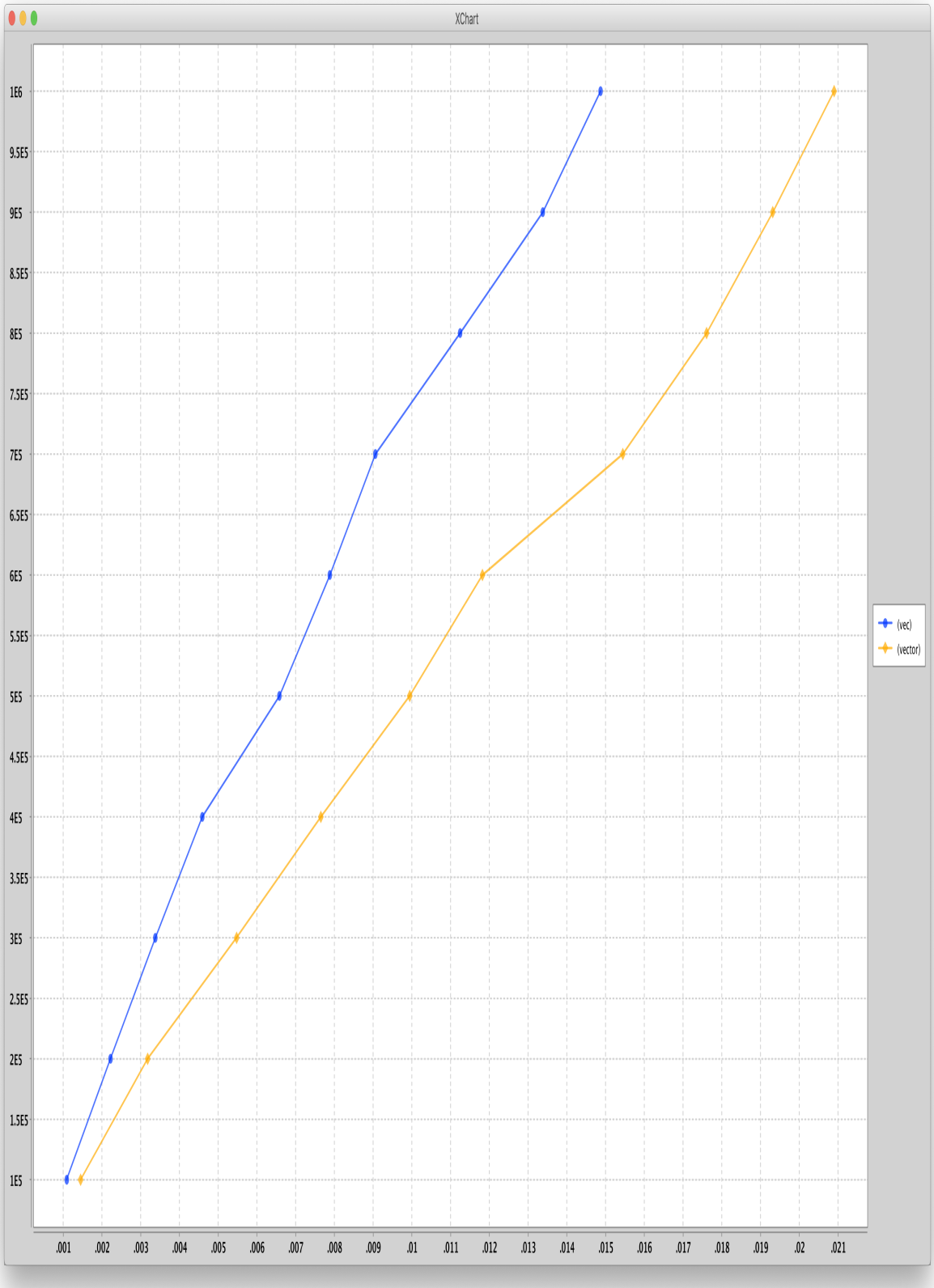
```
(require '[com.hypirion.clj-xchart :as chart]) ;

(let [[vec-results vector-results] (apply map ve
      labels (range 100000 1100000 100000)] ; ❷
  (chart/view
    (chart/xy-chart
      {"(vec)"    [vec-results labels]
       "(vector)" [vector-results labels]})))
```

❶We used `clj-xchart` in other parts of the book to produce visually appealing charts.

❷We need to isolate each result series to draw each line of the chart.

Figure 12.2. Performance of `vec` vs `vector`


XChart
1E6
9.5E5
9E5
8.5E5
8E5
7.5E5
7E5
6.5E5
6E5
5.5E5
5E5
4.5E5
4E5
3.5E5
3E5
2.5E5
2E5
1.5E5
1E5
.001
.002
.003
.004
.005
.006
.007
.008
.009
.01
.011
.012
.013
.014
.015
.016
.017
.018
.019
.02
.021
(vec)
(vector)

The linear behavior of the functions can be clearly seen. `vec` runs slightly faster than `vector` because `vector` iterates the input using a `first/next` semantic compared to the faster `reduce`. We can verify the assumption by using a lazy-sequence which does not implement `IReduce`:

```
(require '[criterium.core :refer [quick-bench]])

(let [reducible (doall (range 1000))
      lazy (doall (map inc (range 1000)))]
  (quick-bench (vec lazy))
  (quick-bench (apply vector lazy))
  (quick-bench (vec reducible))
  (quick-bench (apply vector reducible)))

;; Execution time mean : 22.523189 µs ; ❶
;; Execution time mean : 22.624104 µs
;; Execution time mean : 16.058917 µs ; ❷
;; Execution time mean : 19.471246 µs
```

❶ Execution time is equivalent for `vector` and `vec` when the input collection is seqable but not reducible.

❷ `vec` is faster if the `reduce` implementation provided by the input is more performant.

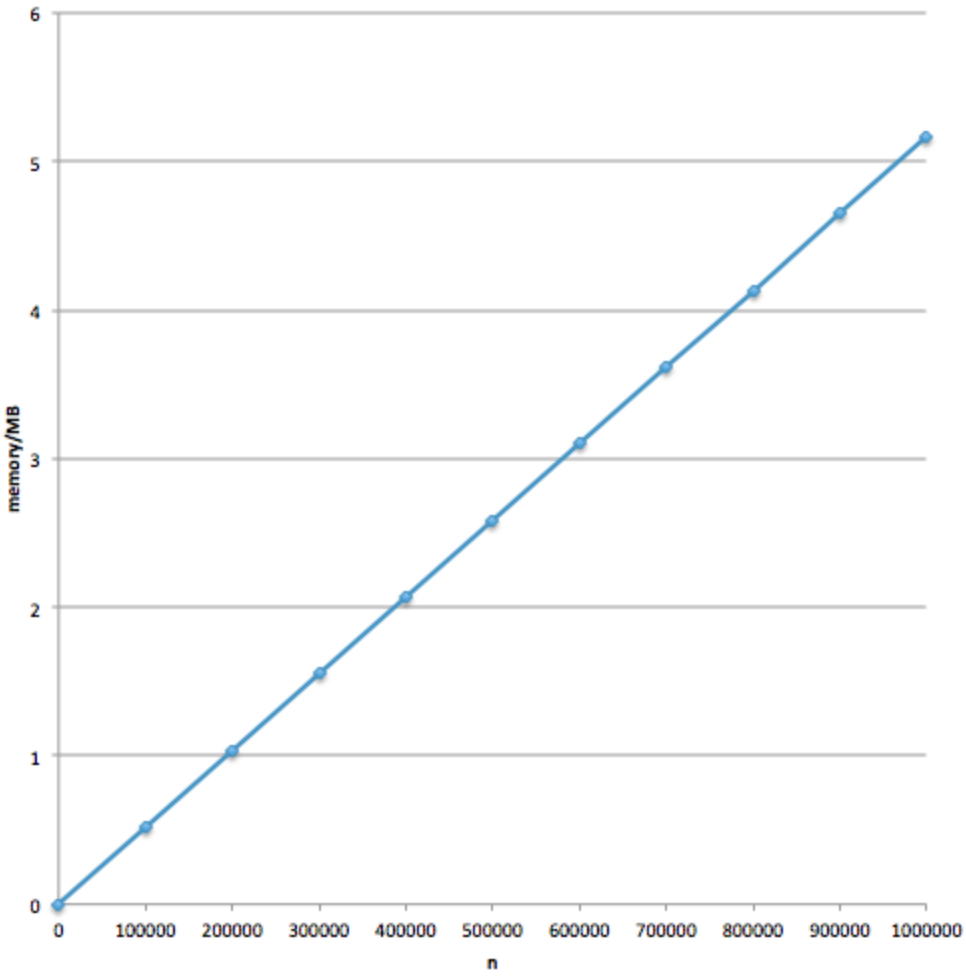
Let's now have a look at the memory allocation using the Java Jamm library [190]. The snippet below is used to illustrate the memory used by a persistent vector in the plot that follows (to run the example, Jamm requires to set a specific Java agent with `-javaagent`):

```
(import 'org.github.jamm.MemoryMeter)

(defn test-memory-vector-of-jamm []
  (let [meter (MemoryMeter.)
        results (for [elements (range 100000 110
                  [elements
                   (.measureDeep meter (make-arr
                   (.measureDeep meter (vec (rep
    (doseq [i (range 4)]
      (doseq [result results]
        (printf "%11d " (get result i)))
      (println))))
```

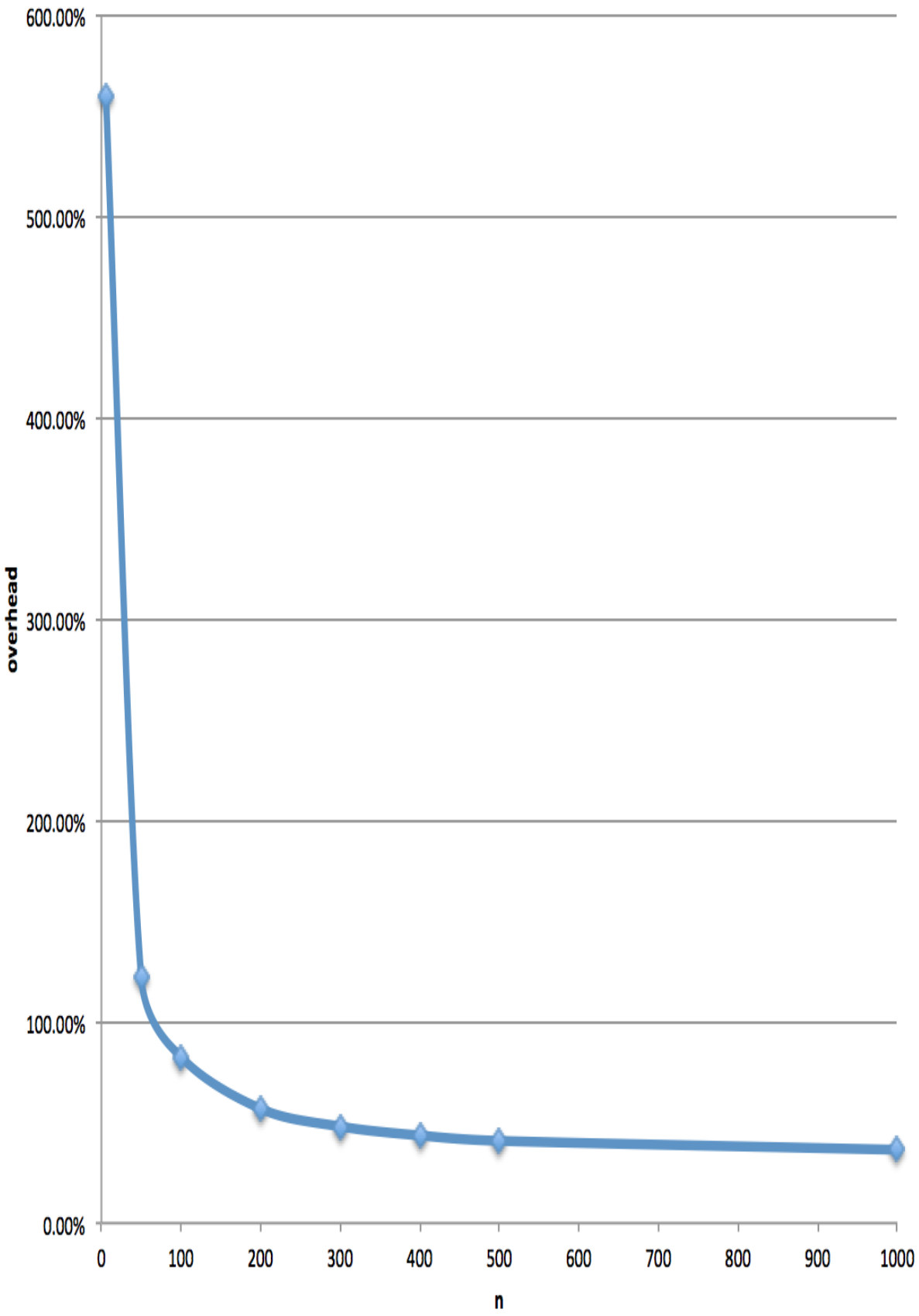
The next plot (created using a spreadsheet charting feature) shows the memory usage (in megabytes) for a Clojure vector and confirms the linear behavior as "n" increases.

Figure 12.3. Memory consumed by a persistent vector



Unfortunately immutability and the ability to grow dynamically comes at a price. The following plot shows the overhead percentage of using a Clojure vector compared to a Java array. The percentage illustrates how much additional space the vector consumes compared to a `java.util.ArrayList` with the same content:

Figure 12.4. Memory overhead of using a persistent vector compared to a Java array


600.00%
500.00%
400.00%
300.00%
200.00%
100.00%
0.00%
overhead
0
100
200
300
400
500
600
700
800
900
1000
n

The overhead is very high for small vectors, but after 1000 elements it settles down at around 35%. This is because each time the array resizes, it increases in length by 50%. However, if the size of the `ArrayList` is known in advance, then the array will be sized correctly for the number of elements and produce even smaller overhead.

[188] The Clojure Style Guide github.com/bbatsov/clojure-style-guide#literal-col-syntax
[189] The transpose of a matrix is a new matrix having the same element as the input but with rows and columns indexes exchanged. Also see: en.wikipedia.org/wiki/Transpose
[190] github.com/jbellis/jamm

12.5 VEC

`function` `since 1.0`

Listing 12.2. → *Creation of data type*

```
  (vec coll)
```

`vec` creates a new vector given another collection as input. `vec` works on almost all collection types:

- Common Clojure collections like lists, sets, maps, etc.

- Java iterables like `clojure.lang.PersistentQueue` or `java.util.ArrayList`
- Native Java arrays (like the ones created with make-array).

The order of the resulting vector matches the order of the elements in the input except for unsorted collections (like sets or maps), for which there isn't a specific order. `vec` can be used like:

```
(vec '(:a 1 nil {})) ; ❶
;; [:a 1 nil {}]
```

❶Note that other nested collections are not transformed by `vec` recursively.

Warning

If the input collection is a Java array of reference types containing 32 elements or fewer, the output vector produced by `vec` will be just an alias to the native array. Therefore, the Java array should not be modified after the call to `vec` or else the immutable Clojure vector may change value! See the examples for more information.

Contract

input

- "coll" can be any seqable collection, an "iterable" collection (such that `(instance? java.lang.Iterable coll)` is `true`) or a Java array (such that `(.isArray (class coll))` is `true`).
- "coll" can also be `nil`.
- The only collection-like Clojure data structures that do not work with `vec` are transients and structs.

notable exceptions

- In event that "coll" cannot transform into a sequence, then a `RunTimeException` is thrown.

output

- `vec` returns a persistent vector instance containing the elements in "coll". Order in the produced vector is respected for ordered collections.

The following table shows several examples of `vec` using different collection types. A brief note is given in the table to explain the results. Note: `clojure.lang` is abbreviated "c.l":

Table 12.1. vec applied to several collection types.

Input Type	Example
`c.l.PersistentList`	`(vec (list 1 2 3))`
`c.l.PersistentVector$TransientVector`	`(vec (transient [1 2 3]`
`clojure.core.Vec`	`(vec (vector-of :int 1 3))`
`c.l.APersistentVector$SubVector`	`(vec (subvec [1 2 3] 1)`
`c.l.PersistentHashSet`	`(vec #{1 2 3 4 5 6})`
`c.l.PersistentHashMap`	`(vec (apply hash-map (r 10)))`
`c.l.PersistentTreeSet`	`(vec (sorted-set :a :b`

`c.l.PersistentTreeMap`	`(vec (sorted-map :a 1 : 2))`
`c.l.PersistentArrayMap`	`(vec (array-map :c "c" "d"))`
`c.l.LongRange`	`(vec (range 10))`
`c.l.LazySeq`	`(vec (map inc (range 10`
`[[Ljava.lang.Long`	`(vec (make-array Long 0`
`[I`	`(vec (int-array [1 2 3]`
`c.l.PersistentQueue`	`(vec (c.l.PersistentQueue/EMF`
record	`(defrecord Name [first last])` `(vec (Name. "Johr`

	`"Doe"))`
`java.lang.Iterable`	`(import java.util.ArrayList) (v (ArrayList.))`

Examples

The following example shows the potential side effects of using `vec` on Java arrays:

```
(def a (make-array Long 3)) ; ❶
(def v (vec a))

v
;; [nil nil nil]

(aset a 1 99)
;; 99

v
;; [nil 99 nil] ; ❷
```

❶Note that we create a Java array using a reference type `Long` instead of a primitive type `long`.

❷aset operation on the array is side-effecting on the vector created by `vec`.

The same doesn't happen for array of primitives:

```
(def a (long-array 3)) ; ❶
(def v (vec a))

v
;; [0 0 0]

(aset a 1 99)
;; 99

v
;; [0 0 0] ; ❷
```

❶This time, the array is of a primitive type `long`.

❷We can see that there is no side effect on the new vector created with `vec`.

Now a more involved example. Madison is looking to buy a blue dress from the "Rachel's Rags" website. Firstly, she searches on the site for a blue dress. Behind the scenes the website queries a

database, converts the results to a vector, allocates a search id and caches the results. Then the website returns the first page of the results as JSON to Madison's browser along with the search id. Madison's browser then renders the JSON as HTML.

Madison, for some reason, then decides to look at page 3 of the results. In response her browser makes an AJAX request to the website passing the search id [191]. The website then retrieves page 3 of the results and returns it to her browser as JSON.

Caching the search results as a vector works particularly well as Madison can jump from page to page of the search results at random. However, as most Clojure database libraries will return their results as a sequence, `vec` is needed to convert the results to a vector. A simplified version of the server code, with the database mocked, may look like this:

```
(import '(java.util UUID))

(defn search-merchandise [& search-options]
  '({:description  "Pencil Dress"             :ty
     :color        :blue                      :p
    {:description  "Asymmetric Lace Dress"    :ty
     :color        :blue                      :p
    {:description  "Short Sleeve Wrap Dress"  :ty
     :color        :blue                      :p
```

```
(def cache (atom {}))

(defn cache-user-search-results! [search-id sear
  (swap! cache assoc search-id (vec search-result

(defn retrieve-user-search-results [search-id pa
  (get (get @cache search-id) page))

(defn render-to-json [{:keys [description price]
  (format "[{'description':'%s', 'price':'%s'}]"

(def search-id (str (UUID/randomUUID)))

(cache-user-search-results!
  search-id
  (search-merchandise {:type :dress :color :blue

(println
  (-> (retrieve-user-search-results search-id 0)
      render-to-json))

;; [{'description':'Pencil Dress', 'price':'60'}

(println
  (-> (retrieve-user-search-results search-id 2)
      render-to-json))
```

```
;; [{'description':'Short Sleeve Wrap Dress', 'p
```

❶ `search-merchandise` searches the database based on the "search-options" passed in and returns a list. For simplicity the results have been mocked.

❷ The search results are cached in a map held by an atom. This is thread safe and for small systems completely appropriate depending on memory requirements. Larger more complicated systems may benefit from using libraries like core.cache [192] or a distributed cache such as Redis [193].

❸ `cache-user-search-results!` takes the search results, converts them to a vector using `vec`, and stores them in the cache.

❹ Given a search id, `retrieve-user-search-results` returns the results to show on a given page. "page" is the page of results required and is zero-based indexed, so for example to retrieve the second page of results, "page" would be 1. `retrieve-user-search-results` is efficient as the results are stored as a vector. To keep the example simple, one item per page is returned.

❺ `render-to-json` is simple json render. A real world system would use an external library such as Cheshire [194].

❻ This code simulates the scenario when Madison initiates her search. A unique search id is allocated at random using Java UUID generation. Then a search of the database is performed

for "blue dresses" and the result is cached. Finally the first page of her results is retrieved and converted to JSON.

❼ This code simulates the scenario when Madison views the third result page. Using the existing search id, the third page of her results is retrieved from the cache and converted to JSON.

See also

- vector-of creates a vector of a primitive type. Use `vector-of` if space efficiency is the biggest concern.
- make-array creates a Java array. Use a Java array if the performance benefits of a mutable data structure are required or to interface Java code.
- long-array creates a Java array of primitive type `long`.
- into can be used to "transfer" the content of a collection into another, including vectors. There are a few small differences compared to `vec`: `into` does not alias arrays, it supports transducers and can also create other collection types. If you focus on vectors only, `vec` better conveys the meaning of the transformation and spares a few keystrokes.

Performance considerations and implementation details

=> `O(n log(n))` **time, with n number of arguments**

=> `O(n)` **space**

vec logic to create the new vector is based on the type of the input collection. `vec` shares the same implementation of `vector` for seqable collections but a different path is taken when:

- The input collection is a "reducible" (implementing the `clojure.lang.IReduce` interface) then `vec` ultimately delegates to the input collection's `reduce` implementation.
- The input collection is a Java (non-primitive) array of 32 or fewer elements, then an empty vector is **not** created but the tail is instead, creating an alias to the Java array.
- The input collection is specifically a `java.util.ArrayList` then the input is iterated with an index.
- The input collection is generically a `java.lang.Iterable` (for example `java.util.LinkedList`) then the input is iterated with a first/rest like semantic.

One important aspect that differentiates `vec` from vector, is the option offered by `vec` to ask the input to build itself. This is possible if the input collection supports the `clojure.lang.IReduce` interface. In that case, reduce is used to build the vector. `vec` performance profile is thus dependent on the type of the input collection, as demonstrated by the following benchmark:

```
(require '[criterium.core :refer [quick-benchmar
(import '[java.util ArrayList LinkedList])

(defmacro b [expr]  ; ❶
  `(first (:mean (quick-benchmark ~expr {}))))

(let [c1 (range 1000) ; ❷
      c2 (map inc c1)
      c3 (ArrayList. c1)
      c4 (LinkedList. c1)]
      (for [t [c1 c2 c3 c4]]
        [(type t) (b (vec t))]))

;; ([clojure.lang.LongRange 1.0874153277485413E-
;;  [clojure.lang.LazySeq   2.0303272494887527E-
;;  [java.util.ArrayList    1.0039155519384434E-
;;  [java.util.LinkedList   1.4243774103139014E-
```

❶The `b` macro extracts the mean execution time in seconds from the statistics returned by Criterium.
❷ `vec` is used to create a vector of 1000 elements. We test four types of input collections. The version using a `java.util.ArrayList` is the fastest of the set, while lazy-seqs are the slowest.

Since we've compared into and `vec` briefly before, here's a benchmark between them:

```
(require '[criterium.core :refer [bench]])

(let [l (range 1000)] (bench (vec l)))
;; Execution time mean : 16.765533 µs

(let [l (range 1000)] (bench (into [] l)))  ; ❶
;; Execution time mean : 17.946582 µs
```

❶into performs slightly worse than `vec`.

The difference between into and `vec` is very small and is not sufficient alone to determine a clear winner. As explained before, `vec` should be preferred in the context of vector processing to better convey the meaning of the computation. In terms of implementation, while into is mainly implemented in

Clojure, `vec` delegates almost immediately to `clojure.lang.LazilyPersistentVector` that proceeds to invoke the correct sequential transformation on the input sequence to create the final vector.

[191] AJAX, short for "Asynchronous JavaScript and XML", is a group of techniques to allow web page content update without the need of a full page refresh. Please see en.wikipedia.org/wiki/Ajax_(programming) for more information.
[192] github.com/clojure/core.cache
[193] redis.io/
[194] github.com/dakrone/cheshire

12.6 PEEK AND POP

`function` `since 1.0`

Listing 12.3. → *Item access, Item removal, Queue operations*

```
(peek [coll])
(pop [coll])
```

`peek` and `pop` access or remove (in the immutable data structure sense) the "head element" from either a vector, list or

queue. The head position depends on the collection type and is not necessarily the first element:

```
(import '[clojure.lang PersistentQueue])
(def q (into (PersistentQueue/EMPTY) [1 2 3]))
(def v [1 2 3])
(def l '(1 2 3))

(peek q) ; 1   ; ❶
(peek v) ; 3   ; ❷
(peek l) ; 1   ; ❸
```

❶ `peek` called on a `clojure.lang.PersistentQueue` type returns the first element added to the queue.
❷ `peek` on a vector, returns the last element added to the vector (when printed, this appears as the right-most element).
❸ `peek` is used on a list, which returns the first element added (when printed it appears as the left-most element).

It's easy to get confused, as there are multiple meanings to identify the position of an item in a collection. It could be:

- The element that was added first or last.
- The implementation detail about how items are store internally.

- The way the items appear when printed.

`pop` returns what's remaining after removing the head of the collection:

```
(-> (PersistentQueue/EMPTY) (conj "a" "b" "c") p
;; ["b" "c"]

(pop ["a" "b" "c"]) ; ❷
;; ["a "b"]

(pop '("a" "b" "c")) ; ❸
;; ("b" "c")
```

❶When we invoke `pop` on a queue the non-printable queue object is returned. We can see the content of the queue using vec.

❷ `pop` on a vector removes the tail element, which is the right-most item when printed and the last item that was added.

❸ `pop` used on a list removes the head element, which is the left-most and was the first added.

The following table summarizes what we have seen so far:

Table 12.2. Difference between insertion and printing order for `peek` and `pop`

Type	`peek`	`pop`	Pop empty?	queue
Retrieve first added	Remove first added	Ok	vector	Retrieve last added
Remove last added	Throws	list	Retrieve first added	Remove first added

The table also shows that `pop` on an empty collection only works for queues.

Contract

input

- "coll" is the only mandatory argument. The collection needs to implement the `IPersistentStack` interface, which can be verified with `(instance? clojure.lang.IPersistentStack coll)`. The commonly used collection types supporting this interface are: Queues, lists and vectors.

notable exceptions

- `ClassCastException` when the collection does not support the `IPersistentStack` interface. Common examples producing this error are: `(peek (range 10))` or `(peek #{1 2 3 4})`.
- `IllegalStateException` trying to `pop` from an empty vector or an empty list. It's okay to `pop` an empty queue.

output

- `peek`: returns the element that was added first (for Queues and lists) or the element that was added last (vectors). `peek` returns `nil` if "coll" is empty or `nil`.

- `pop`: returns what remains after removing the element returned by `peek`. `pop` throws exception for an empty vector or list. Returns `nil` when "coll" is `nil`.

Examples

`peek` and `pop` are typical queue operations. Their presence in the standard library helps to convey that meaning. A queue is an abstract data type characterized by its insertion/extraction order [195].

Thanks to `peek` and `pop`, a LIFO queue (last-in first-out queues are also called stacks) can be implemented efficiently on top of vectors (FIFO queues are implemented directly by queue). In the following example we can use a vector-based LIFO queue to verify if a Clojure form contains balanced parenthesis:

```
(require '[clojure.set :refer [map-invert]])

(defn queue [] []) ; ❶
(def push conj) ; ❷
(def brackets {\[ \] \( \) \{ \}}) ; ❸

(defn check [form] ; ❹
```

```
    (reduce
      (fn [q x]
        (cond
          (brackets x)
          (push q x)  ; ❺
          ((map-invert brackets) x) ; ❻
          (if (= (brackets (peek q)) x)
            (pop q)
            (throw
              (ex-info
                (str "Unmatched delimiter " x) {}))
          :else q))
      (queue) form)) ; ❼

  (check "(let [a (inc 1]) (+ a 2))")
  ;; ExceptionInfo Unmatched delimiter ]

  (check "(let [a (inc 1)] (+ a 2))")
  ;; []
```

❶To better conform to the idea of using a queue, we add this simple constructor that wraps an empty vector. We could swap this with an empty list or another queue implementation provided they support the `peek` `pop` and `conj` semantic.

❷Similarly `push` is just an alias for conj to help us thinking in terms of queues.

❸We need a list of all the allowed brackets and their matching pairs. We can organize this as a dictionary to quickly lookup the closing bracket given the open bracket as a key.

❹The `check` function performs the scan of the input. It is organized as a reduce operation around a (initially empty) queue and the list of input tokens.

❺Each character goes through a cond expression: if we have an opening bracket, we push it in the queue waiting to see what happens the next iteration.

❻If we have a closing bracket, we check to see if the related opening bracket is the last thing we saw. This is a `peek` operation from the queue. If we have a match, we `pop` the matching bracket and we wait the next iteration. If there is no match, an exception is thrown. Note the use of map-invert from the `clojure.set` namespace to invert keys-values in a map.

❼If the token is not a bracket, we do nothing and return the queue.

`peek` and `pop` can also be used efficiently in vector-based loops, where the current item is extracted at each iteration and the remainder is sent to the next via recur:

```
(defn reverse-mapv [f v]           ; ❶
  (loop [v v res (transient [])]   ; ❷
    (if (peek v)                   ; ❸
      (recur                       ; ❹
        (pop v)
        (conj! res (f (peek v))))
      (persistent! res))))         ; ❺

(reverse-mapv str (vec (range 10)))
;; ["9" "8" "7" "6" "5" "4" "3" "2" "1" "0"]
```

❶ `reverse-mapv` returns the reverse of a vector and, at the same time, can apply a transformation to each element.
❷ The loop starts by assigning the vector to the local binding "v" and creating an empty vector for the results. For additional efficiency, we can use a transient vector because the accumulation of results is local to the loop.
❸ `peek` returns `nil` when there are no more elements, so we can use this as the signal for stopping the recursion.
❹ The two arguments for recur are the `pop` of the rest of the vector with the new results vector in which we conj the transformation of the first element.
❺ The transient vector needs to be made immutable again before leaving the local context with persistent!.

See also

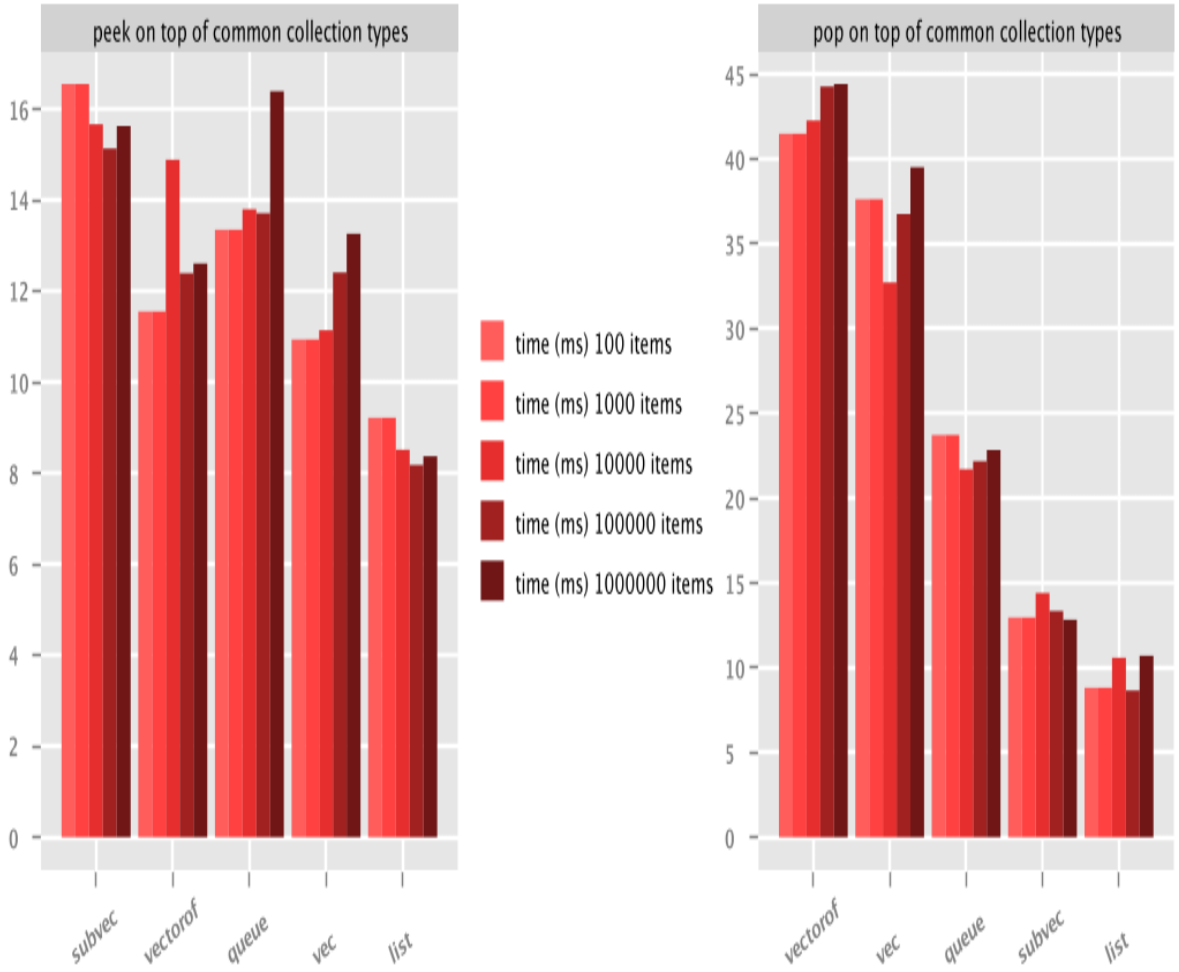
- conj is what is used to push element in lists, vectors and queues. There is no "push" function in the standard library but `conj` can be used the same way.
- first and rest, while working natively for lists, require a transformation into a sequence when used on a vector. `peek` and `pop` represent the efficient way to perform similar operations on a vector.

Performance considerations and implementation details

=> `O(1)` **Constant time**

`peek` and `pop` are fast constant time operations. The following chart shows them executing on supported collections of increasing sizes:

Figure 12.5. `peek` and `pop` benchmarks.



lists tend to perform better in both benchmarks, although in absolute terms these are all very fast operations. The histograms are not perfectly flat (by collection type) because of small fluctuations in the benchmarks. Some lack of precision is possible at this resolution of just a few milliseconds.

[195] Abstract data types are a specification of the semantic of a data structure without specification details. The most common abstract data type are summarized on this Wikipedia page: en.wikipedia.org/wiki/Abstract_data_type

12.7 VECTOR-OF

`function` `since 1.2`

Listing 12.4. → *Persistent Data Type, Primitives, Memory Efficient Storage*

```
(vector-of
  ([t])
  ([t x1])
  ([t x1 x2])
  ([t x1 x2 x3])
  ([t x1 x2 x3 x4])
  ([t x1 x2 x3 x4 & xn]))
```

`vector-of` creates a persistent vector that stores elements internally as a primitive type. It is used when a lower memory footprint is required than the version created by vector, provided primitive types can be used for the specific problem. `vector-of` behaves very similar to a normal vector: it can be accessed randomly or treated as sequential, it can be used as a function to access its elements by index and it's comparable:

```
(vector-of :int) ; ❶
;; []

(vector-of :int 16/5 2.8 1M Double/NaN) ; ❷
;; [3 2 1 0]

((vector-of :int 1 2 3) 2) ; ❸
;; 3

(sort [(vector-of :int 7 8 9)  ; ❹
       (vector-of :int 0 1 2)])
;; ([0 1 2] [7 8 9])
```

❶An empty vector of integer primitives.

❷Numeric types that are not originally primitives can be added to the vector provided they can be transformed into the target primitive type. See for example ratios and bigdec in this example. Note that `Double/NaN` equals 0 when coerced to int.

❸Using a newly created vector as a function. The single numeric argument is used to access the vector at that index (zero-based).

❹Vectors created with `vector-of` support sorting using via compare semantic.

Contract

input

- "t" is one of the following 8 keywords representing their respective types: `:int`, `:long`, `:float`, `:double`, `:byte`, `:short`, `:char` or `:boolean`.
- "x1", "x2", "x3, "x4 and "xn" are optional arguments. The elements can be of different types than "t", but they must be able to be coerced to "t".

notable exceptions

```
(vector-of Integer 1 2 3) ; ❶
;; NullPointerException

(vector-of :double \a \b \c) ; ❷
;; ClassCastException

(vector-of :int 1 2 nil 3 4) ; ❸
;; NullPointerException

(vector-of :short (inc Short/MAX_VALUE)) ; ❹
;; IllegalArgumentException out of range: 32768
```

❶ `NullPointerException` when "t" is not one of the 8 accepted types. This was fixed in Clojure 1.9 [196].

❷ If an element cannot be coerced to "type", then a `CastClassException` is thrown.

❸ If an element is `nil`, then a `NullPointerException` is thrown.

❹ If there is an underflow or overflow of an element then an `IllegalArgumentException` is thrown.

output

- A persistent vector containing the given elements in order, if any. Otherwise an empty vector.



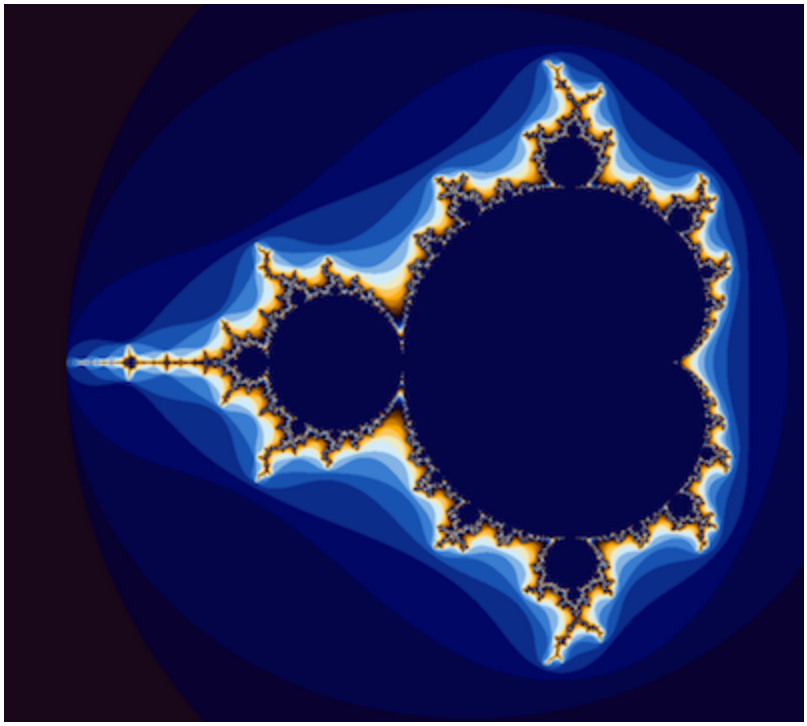
Warning

vectors created with `vector-of` cannot contain `nil`, whether passed at construction time or added via conj. This is because `vector-of` only allow primitive types.

Examples

One area where `vector-of` is useful is in numerical computing where many numbers have to be stored in memory, for example in creating fractal images. Fractal images, such as the Mandelbrot set [197], have entered the popular culture and illustrate the beauty and complexity of mathematics.

Figure 12.6. The Mandelbrot set.



To produce the image of the Mandelbrot set, an iterative process is applied to the numbers on the complex plane. For each complex number [198], the number of iterations before the process heads off to infinity is counted. The 3-tuple consisting of the real part, the imaginary part and the number of iterations, can be efficiently stored in a vector of primitives. The triplets are then plotted: the real part on the x-axis, the imaginary part on the y-axis, and the number of iterations is mapped on to a color gradient.

Here is a simplified version of the code [199] that produced the image of the Mandelbrot set:

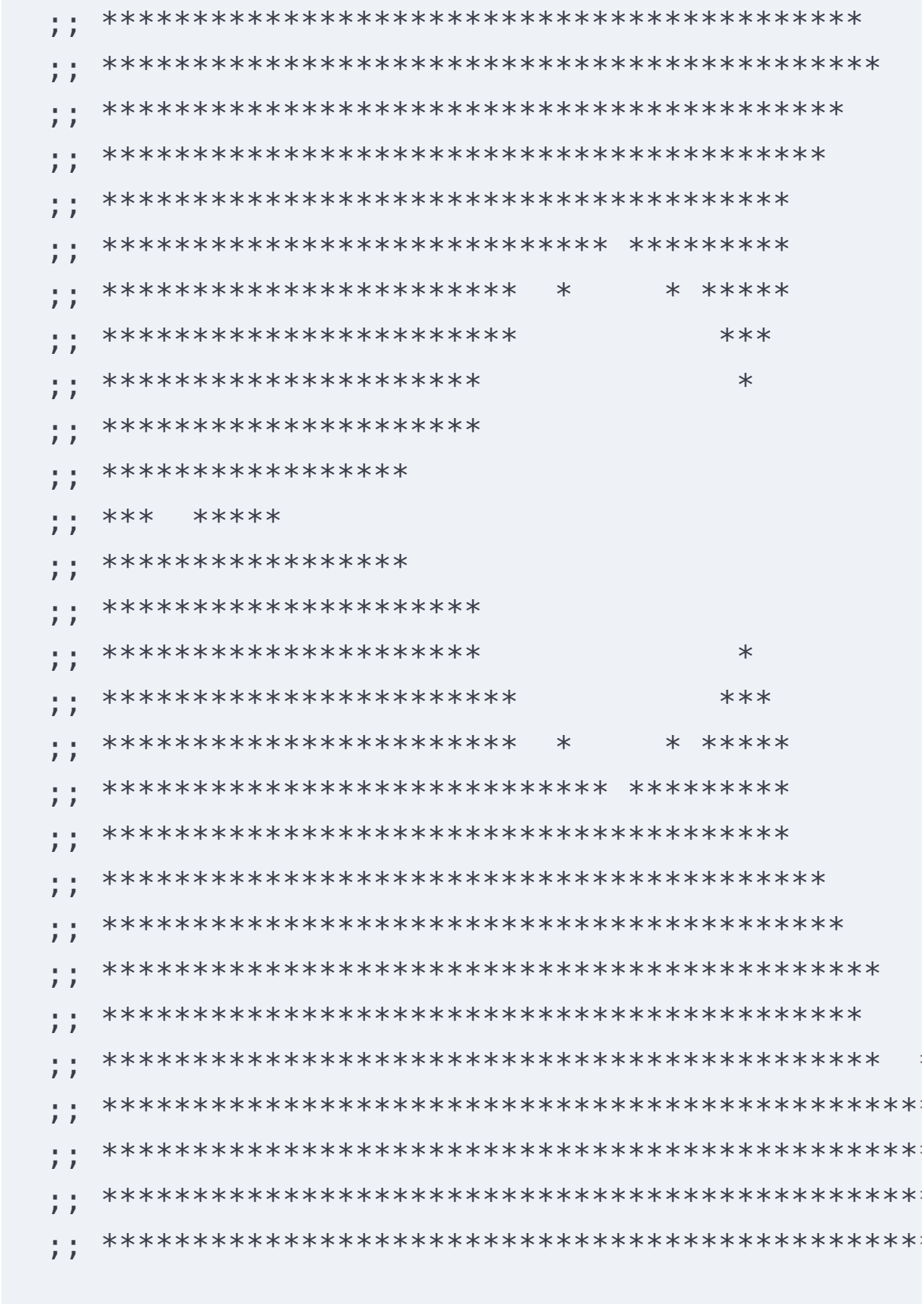
```
(def max-iterations 99) ; ❶

(defn calc-mandelbrot [c-re c-im] ; ❷
  (let [sq    (fn [x] (* x x))
        iter  (reduce (fn [[z-re z-im] i]
                        (if (or (= i 99) (> (+ (
                          (reduced i)
                          [(+ c-re (sq z-re) (-
                           (+ c-im (* 2 z-re z-i
                      [0 0] (range (inc max-iter
        (vector-of :double c-re c-im iter))) ; ❸

(def mandelbrot-set ; ❹
  (for [im (range 1 -1 -0.05) re (range -2 0.5 0
    (calc-mandelbrot re im)))

(doseq [row (partition 80 mandelbrot-set)] ; ❺
  (doseq [point row]
    (print (if (> max-iterations (get point 2))
  (println))

;; ************************************************
;; ************************************************
;; ************************************************
;; ************************************************
;; ************************************************
;; ************************************************
;; ********************************************  *
```



```
;; ********************************************************
;; ********************************************************
```

❶Indexes are zero based.

❷ `calc-mandelbrot` takes a complex number c (represented as `c-re` and `c-im` corresponding to the real and imaginary parts respectively). It then applies the iterative calculation $z_i = \overline{z_{i-1}}^2 + c$, $z_0 = 0$ and counts the number of iterations before z_i starts to head to infinity.

❸It returns its result as a triplet modeled as a 3 element `vector-of` doubles.

❹In this example only a thousand or so elements of the Mandlebrot set are calculated. However, for high resolution images, millions of numbers need to be calculated. The `mandelbrot-set` var definition stores the set for later use.

❺This is a simple low resolution ASCII based renderer.

`vector-of`: an ideal hash tree implementation in Clojure

`vector-of`, for all common cases and scenarios, behaves like a vector producing a persistent data structure which is constrained to primitive types. But there are deeper differences if we look at how these two are implemented. The vector implementation shipped with the original Clojure 1.0 is one of the main selling point of Clojure: a persistent data structure that is thread-safe and efficient. It is written in Java and is at the basis of all the persistent data types (see the call-out section in vector for an overview of how it is implemented). `vector-of` is instead written in

Clojure itself in a file called "gvec.clj" [200] that is imported by "core" when bootstrapping Clojure.

Reading the sources of `vector-of` implementation is an interesting learning experience for many reasons:

- See how an ideal hash-tree could be implemented in Clojure.
- It makes good uses of definterface and deftype to mix-in the many interfaces to support sequential behavior, equality semantic, chunking, printing and anything else required to be a "good Clojure citizen".
- If you ever need to implement your own data structure in Clojure, refer to `gvec.clj` to see what interfaces need to be supported (there are many).
- It uses low level Java arrays from Clojure to efficiently construct the 32-way shallow trees needed for efficient persistence.

See also

- vec creates a vector from different types of collections. You probably want to use normal vectors most of the time, as they are the most flexible. However, if the application has memory constraints related to storing many small vectors, consider using `vector-of` primitives.
- make-array to create a Java array. Use a Java array if, along with space optimizations, the performance benefits from a mutable data structure. Also use Java arrays if interfacing with Java code that requires them.

Performance considerations and implementation details

=> `O(n log(n))` **time, with n number of arguments**

=> `O(n)` **space**

All arities except the last, are faster options to create a new vector with `vector-of`:

```
(require '[criterium.core :refer [quick-bench be

(quick-bench (vector-of :int 1 2 3 4)) ; ❶
;; Execution time mean : 15.340593 ns

(quick-bench (vector-of :int 1 2 3 4 5)) ; ❷
;; Execution time mean : 124.127511 ns
```

❶A vector is created with `vector-of` and 4 initial elements.
❷A fifth element is added to creation of the vector. The additional element forces `vector-of` to iterate the final list of variable arguments with a visible slow-down.

For larger vectors there is no `vec-of` function, but the overhead can be slightly alleviated using conj:

```
(def data (doall (range 100000)))

(bench (apply vector-of :int data))
;; Execution time mean : 6.975521 ms

(bench (reduce conj (vector-of :int) data)) ; ❶
;; Execution time mean: 5.926824 ms
```

❶The performance gain by using conj is around 15%.

`vector-of` has a similar invocation semantic as vector. The two can be compared with the following benchmark, which shows vector being more than twice faster than vector-of:

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (range 100)]
  (quick-bench (apply vector xs)))  ; ❶
;;  Execution time mean : 2.051646 µs

(let [xs (range 100)]
  (quick-bench (apply vector-of :long xs))) ; ❷
;; Execution time mean : 5.004903 µs
```

❶apply is used to create a vector of many arguments.
❷Similarly a `vector-of` `:long` is created.

vector makes use of an optimization where internally a transient is used to create a mutable data structure before the reference to the final immutable data structure is returned. However `vector-of` does not implement a similar optimization [201].

As reminded throughout the chapter, `vector-of` sweet spot is space efficiency. The Mandelbrot set example used triplets created using `vector-of` rather than `vector`. The space saving of a triplet created with `vector-of` can be measured with the JAMM library [202]. To execute the following example, the REPL process needs to be instrumented with the Java Agent provided by JAMM:

```
(import '[org.github.jamm MemoryMeter])

(defn memory-vector-of []
  (let [items (range 1.0 1e3) ; ❶
        meter (MemoryMeter.)
        bytes-vector (.measureDeep meter (apply v
        bytes-vector-of (.measureDeep meter (app
        saving (* (double (/ (- bytes-vector byt
    (println "Bytes used by vector" bytes-vector
```

```
      (println "Bytes used by vector of" bytes-vec
      (println (str "Saving " (format "%3.2f" savi

  (memory-vector-of) ; ❷
  ;; Bytes used by vector 29456
  ;; Bytes used by vector of 9472
  ;; Saving 67.84%
```

❶range generates 1000 doubles.

❷The space gain by using `vector-of` is substantial.

Another area `vector-of` is not as fast as `vector`, is accessing elements:

```
  (let [v1 (vec (range 10000))]
    (bench (nth v1 1000))) ; ❶
  ;; Execution time mean : 12.264993 ns

  (let [v1 (apply vector-of :int (range 10000))]
    (bench (nth v1 1000))) ; ❷
  ;; Execution time mean : 19.324863 ns
```

❶Making access to the element at index "1000" using nth on a normal vector created with vec.
❷Access to the same element at index "1000" on a vector created with `vector-of`.

As the example shows, the slow down can be as high as 40% (but in absolute scale, we are still talking about very fast access times in the order of nanoseconds). The cause of this is that the accessors functions get and nth or even using a vector as a function, all return a reference type. Therefore, the element has to be boxed before it can be returned. The user of `vector-of` should therefore pay attention if the vector is frequently accessed and weight the gain in memory space against access speed to decide which implementation to use.

[196] See the Clojure Jira ticket system at clojure.atlassian.net/browse/CLJ-1705

[197] See en.wikipedia.org/wiki/Mandelbrot_set

[198] See en.wikipedia.org/wiki/Complex_number

[199] See github.com/rachbowyer/csl-10-vector-public

[200] github.com/clojure/clojure/blob/master/src/clj/clojure/gvec.clj

[201] Open ticket to add transients to `vector-of` clojure.atlassian.net/browse/CLJ-1416

[202] github.com/jbellis/jamm

12.8 MAPV

`function` `since 1.4`

Listing 12.5. → *Sequence processing, Higher-order functions*

```
(mapv
  ([f coll])
  ([f c1 c2])
  ([f c1 c2 c3])
  ([f c1 c2 c3 & colls]))
```

`mapv` is a specialized version of map returning a persistent vector rather than a sequence. It applies the function passed as

argument to each element in a collection and returns the results in a vector:

```
(mapv inc [0 1 2 3]) ; ❶
;; [1 2 3 4]
```

❶ `mapv` has a similar interface to map, but returns a vector.

`mapv` also works with multiple collections. In this case "f" is applied simultaneously to the first element in each collection, then the second and so on until reaching the end of the shortest collection:

```
(mapv hash-map [:a :b :c] (range))
;; [{:a 0} {:b 1} {:c 2}]
```

There are two key differences between `mapv` and `map`:

- `mapv` returns a vector rather than a lazy sequence. In doing so it also consumes any lazy input completely.
- There is no single arity version of `mapv` to produce a transducer.

Contract

inputs

- "f" is a function returning any type whose arity must match the number of collections passed to `mapv`. It is recommended that "f" is a pure function.
- "c1, c2, c3 and coll" are seqable collections (such as `(instance? clojure.lang.Seqable coll)` is `true`. Any `nil` is treated as an empty collection resulting in an empty vector as output.
- "colls" are optional additional seqable collections.

notable exceptions

- If "f" has an incorrect arity, then an `ArityException` is thrown.
- If "f" is not a function, then a `ClassCastException` is thrown.
- If any collection is "colls" is not seqable, then an `IllegalArgumentException` is thrown.

outputs

- A persistent vector whose first element consists of applying "f" to the first element in "coll" and each first element in additional "colls", whose second element consists of applying "f" to the second element in "coll" and each second element in "colls" and so on.

Where is my "removev" function?

As Clojure has `mapv` and filterv it is natural to ask where are the other vector versions of the sequence functions. The answer is that the only sequence functions with vector versions are map and filter. However, the addition of transducers in Clojure 1.7 allows efficient vector versions of other sequence functions to be created easily. So instead of writing:

```
(require '[criterium.core :refer [quick-bench]])

(vec (remove odd? (range 10))) ; ❶
;; => [0 2 4 6 8]

(let [r (range 100)]
  (quick-bench (vec (remove odd? r)))) ; ❷
;; Execution time mean : 3.619460 µs
```

❶vec produces a vector out of a sequence based computation.
❷The benchmark reports around 3.61 microseconds execution time mean.

It is now possible to write:

```
(into [] (remove odd?) (range 10)) ; ❶
;; => [0 2 4 6 8]

(let [r (range 100)]
  (quick-bench (into [] (remove odd?) r))) ; ❷
;; Execution time mean : 1.605351 µs
```

❶remove without the last sequence argument produces a transducer.
❷The benchmark reports around 1.60 execution time mean.

In this example the transducer runs more than 50% faster than the sequence version removing the need for a "removev".

Examples

Persistent vectors of type double and length "n" can be used to represent mathematical vectors in \mathbb{R}^n [203]. It is then straightforward to use `mapv` to implement addition, subtraction, scalar multiplication as follows:

```
(defn create-vector-fn [f]           ; ❶
  (fn [a & b] (apply mapv f a b)))

(def add (create-vector-fn +))       ; ❷

(def subtract (create-vector-fn -))

(defn scalar-multiply [c a]
  (mapv (partial * c) a))

(defn dot-product [a b]              ; ❸
  (reduce + (map * a b)))
```

```
(add [1 2] [3 4])
;; [4 6]

(subtract [2 7 3] [5 4 1])
;; [-3 3 2]

(scalar-multiply 3 [1 2 3])
;; [3 6 9]

(dot-product [1 1 0] [0 0 1])        ; ❹
;; 0
```

❶There is a lot going on in this function. It is a high order function that takes as function "f" as input, and returns another function. The function returned takes one or more vectors as arguments and uses `mapv` to apply "f" to their elements. The argument "b" optionally contains a list of vectors and so apply is needed to execute `mapv` correctly.

❷Here we make use of the high order function `create-vector-fn`. We pass in the operator + and receive back a function that adds one or more vectors together. The function is then bound to "add" using def.

❸This function implements the algebraic dot product [204]. The operation multiplies elements in the vectors before summing them. As a scalar is returned, not a vector, `mapv` does not help us. Instead map and reduce are used.

❹As `[1 1 0]` and `[0 0 1]` are perpendicular, then their dot product should be 0 as expected.

See also

- map the standard map operation, which produces either a transducer or a lazy linked list. Use standard `map` when you are not interested in the result as a vector.

- mapcat is useful when the result of applying `f` to an item is again a sequence, with the overall results of producing a sequence of sequences.mapcat applies a final concat operation to the resulting list, flattening the result one level.
- amap operates with the same semantics of `map` on Java arrays.
- pmap executes the `map` operation on a separate thread thus creating a parallel `map` execution pool. Replacing `map` with pmap makes sense when the overall cost of handling the function `f` to separate threads is less than the execution of `f` itself. Long or otherwise processor-consuming operations usually benefit from using pmap.
- clojure.core.reducers/map is the version of `map` used in the context of reducers. It has the same semantic of `map` and should be used similarly in the context of a chain of reducers.

Performance considerations and implementation details

=> `O(n)` **time and space, n number of elements in the shortest input collection**

Without considering laziness, `mapv` typically outperforms the single input collection version of map as `mapv` writes its results directly to a transient vector whereas map creates a lazy sequence:

```
(require '[criterium.core :refer [quick-bench]])

(let [r (range 10000)] (quick-bench (into [] (ma
;; Execution time mean : 428.559902 µs

(let [r (range 10000)] (quick-bench (mapv inc r)
;; Execution time mean : 257.362476 µs
```

❶We give away laziness by forcing the sequence into a vector as this is the goal of the current comparison.
❷The similar operation with `mapv` is around 40% faster.

Using map with transducers skips the intermediate sequence creation and performs considerably better than the basic map version but still worse than the `mapv` version:

```
(let [r (range 10000)] (quick-bench (into [] (ma
;; Execution time mean : 293.384399 µs
```

❶The solution that uses map with into via transducers perform roughly in between the two other versions seen before.

In some other cases, the benefits of being lazy leads to code using the two-arity version of map outperforming `mapv`. For example, if an application is predominantly using just a few items from a bigger collection, it makes sense to use lazy sequences instead of vectors (if possible):

```
(let [r (range 10000)] (quick-bench (subvec (mapv
;; Execution time mean : 263.561818 µs

(let [r (range 10000)] (quick-bench (vec (take 10
;; Execution time mean : 1.548534 µs
```

❶After incrementing the element of the vector, we use subvec to extract the first 10 of them.
❷Similarly, the first 10 elements are extracted after map is applied, with a vector transformation happening at the end. This operation, since it only realizes just a few element of the sequence, is 200 times faster!

`mapv` arities taking more than one collection are not making direct use of transients. Despite the fact that the input/output are still vectors, `mapv` performs an intermediate

transformation into a sequence and back, with appreciable performance impact:

```
(let [r (range 10000)] (quick-bench (into [] (ma
;; Execution time mean : 1.139211 ms

(let [r (range 10000)] (quick-bench (mapv + r r)
;; Execution time mean : 1.171993 ms
```

❶Multiple collections are now used as input.

❷ `mapv` is now performing roughly the same as plain map (the implementation of `mapv` for multiple collections is indeed the same shown in the previous line).

If we wanted to implement a version of `mapv` that performs better using two collections as input, the following could be a possible option:

```
(defn mapv+ [f c1 c2] ; ❶
  (let [cnt (dec (min (count c1) (count c2)))]
    (loop [idx 0
           res (transient [])]
      (if (< cnt idx)
        (persistent! res)
        (recur (+ 1 idx) (conj! res (f (nth c1 i
```

```
  (let [r (vec (range 10000))] (quick-bench (mapv+

  ;; Execution time mean : 427.152060 µs
```

❶ `mapv+` strategy is to use a loop-recur over the shortest of the two input vectors. A transient vector is then gradually built using conj!. A persistent! vector is then returned when the max number of elements has been reached.

❷ The benchmark confirms around a 50% improvement compared to plain `mapv`.

[203] en.wikipedia.org/wiki/Real_coordinate_space

[204] en.wikipedia.org/wiki/Dot_product#Algebraic_definition

12.9 FILTERV

`function` `since 1.4`

Listing 12.6. → *Sequence processing, Higher-order functions, Selection*

```
  (filterv pred coll)
```

`filterv` is a specialized version of filter returning a persistent vector rather than sequence. `filterv` takes a predicate and

uses it to decide what items should be retained (and implicitly what should be removed) from the final resulting vector:

```
(filterv odd? (range 8)) ; ❶
;; [1 3 5 7]
```

❶Basic application of `filterv`.

Apart from returning a vector, `filterv` differs from filter in the following:

- `filterv` is not lazy and will eagerly load the resulting vector into memory.
- `filterv` is missing an arity for transducers.

Contract

input

- "pred" is a predicate function. The returned value is interpreted as logical boolean.
- "coll" is any seqable collection. If "coll" is `nil`, then it is treated as an empty collection.

exceptions

- If "pred" has the incorrect arity, then an `ArityException` is thrown.
- If "pred" is not a function, then a `ClassCastException` is thrown.
- If "coll" is not seqable, then a `IllegalArgumentException` is thrown.

output

A persistent vector consisting of all items in "coll" for which `(pred item)` is truthy. The order of the items in the vector matches the order of the items in "coll".

Examples

In the following example, two tasks need asynchronous processing. An invaluable tool to orchestrate asynchronous tasks is "core.async" [205] a library commonly found in many concurrent Clojure applications [206].

After the first task completes, `filterv` could be used to remove the channel that has completed from the list of all channels [207]. For simplicity, the code below calculates the "e" and "π" mathematical constants instead of calling external resources. The important concept to consider is that the tasks

will be busy for some considerable time depending on the precision requested:

```
(require '[clojure.core.async :refer [>!! <!! >!

(defn calculate-pi [precision]
  (->> (iterate #(* ((if (pos? %) + -) % 2) -1)
       (map #(/ 4 %))
       (take-while #(> (Math/abs %) precision))
       (reduce +)))

(defn calculate-e [precision]
  (letfn [(factorial [n] (reduce * (range 1 (inc
    (->> (range)
         (map #(/ (+ (* 2.0 %) 2) (factorial (in
         (take-while #(> (Math/abs %) precision)
         (reduce +))))

(defn get-results [channels]
  (let [[result channel] (alts!! channels)
        new-channels     (filterv #(not= channel
    (if (empty? new-channels)
      [result]
      (conj (get-results new-channels) result)))

(let [[pi-in pi-out e-in e-out] (repeatedly 4 ch
  (go (>! pi-out {:type :pi :num (calculate-pi (
  (go (>! e-out  {:type :e  :num (calculate-e  (
```

```
    (>!! pi-in 1e-4)
    (>!! e-in 1e-5)
    (get-results [e-out pi-out]))

  ;; [{:num 3.1415426535898248, :type :pi} {:num 2
```

❶ π is calculated using the Leibniz formula [208].
Mathematicians view the calculation of π as the summation of an infinite series and the power of a functional language like Clojure can easily reflect this. "precision" is used in an informal way to specify the accuracy.

❷ e is calculated using one of the Brothers' formulae [209]. Again "precision" is used to specify the accuracy in an informal way.

❸ `get-results` is a recursive function that waits for each channel in turn to complete and return its results.

❹ `filterv` is used to remove the channel that completed from the list of all channels. The code would work perfectly well if it used `filter` instead of `filterv`, but this would introduce a subtle problem. When `get-results` is called, "channels" is a vector. But `filter` returns a sequence, so the data type of "channels" is changed as the code runs in a non-obvious way. This type of behavior should be avoided as it can lead to bugs.

❺ Two `go` blocks are setup, the first calculates π and the second e.

❻ Requests to calculate π and e are placed on the appropriate channel and the code then waits for both results

See also

- filter is the less specific and more frequently used `filterv` sister function. Usage of `filterv` should be restricted to cases where the input/output are expected to be (and remain) vectors. Prefer filter to `filterv` when the type of the output is not relevant or laziness is more important.
- mapv is the another vector oriented operation to process each element of a vector.

Performance considerations and implementation details

=> `O(n)` **time and space, where n is the number of items in "coll"**

The comments in the "Performance Considerations and Implementation Details" section for mapv regarding transducers and laziness also apply to `filterv` and `filter`. In summary, `filterv` typically outperforms the two-arity version of `filter` as `filterv` writes its results directly to a transient vector whereas `filter` first creates a lazy sequence. For instance:

```
(require '[criterium.core :refer [quick-bench]])

(let [r (range 10000)] (quick-bench (into [] (fi
;; Execution time mean : 309.609565 µs
```

```
(let [r (range 10000)] (quick-bench (filterv odd
;; Execution time mean : 117.814547 µs

(let [r (range 10000)] (quick-bench (into [] (fi
;; Execution time mean : 196.445960 µs
```

❶We remove laziness from filter by forcing the final result into a vector.

❷ `filterv` is about 60% faster.

❸The transducers version of filter shows good results.

[205] github.com/clojure/core.async

[206] There are many possible candidates for asynchronous processing. `core.async` has the unique advantage of being able to park waiting jobs so the allocated thread can be reused for active action, resulting in better resource allocation. You can always achieve the same result with explicit callbacks, but this approach results in side-effecting code that is difficult to follow. See the `core.async` rationale at clojure.org/news/2013/06/28/clojure-clore-async-channels

[207] This example is inspired by the following stack overflow post stackoverflow.com/questions/31858846/waiting-for-n-channels-with-core-async

[208] en.wikipedia.org/wiki/Leibniz_formula_for_π

[209]

en.wikipedia.org/wiki/List_of_representations_of_e#As_an_infinite_series. Amazingly this formula was only discovered in 2004!

12.10 SUBVEC

`function` `since 1.0`

Listing 12.7. → *Vector creation, Vector splitting, Vector access*

```
(subvec
  ([v start])
  ([v start end]))
```

`subvec` creates a "sub-vector" from contiguous items contained in another vector without the need of walking all elements from the beginning. We can for example extract a specific "window" into the input vector specifying an end and start:

```
(subvec [1 2 3 4] 1 3) ; ❶
;; [2 3]
```

❶Creating a sub-vector that goes from index "1" to index "3".

sub-vectors can be created on top of the other kind of vectors. In this case the resulting sub-vector inherits the characteristics of the underlining vector:

```
(def subv (subvec (vector-of :int 1 2 3) 1)) ; ❶
(conj subv \a)
;; [2 3 97] ; ❷

(conj subv nil)
;; java.lang.NullPointerException ; ❸
```

❶ `subvec` is used on top of a vector-of primitive integer types.

❷ The char `\a` enters the vector as the resulting cast from char to int

❸ `nil` is not allowed in primitive vectors. The same affects the resulting sub-vector.

Warning

The vector returned by `subvec` is an independent view of a range of elements contained in the input vector. Once the `subvec` is generated, the originating vector can be altered without any impact on related sub-vectors instances. Although the two instance are substantially independent, there is one subtle side effect: as Fogus M. and Houser C. warn [210], vectors are not queues and `subvec` should not be used to implement a pop from the front of the vector. Although `subvec` completes in constant time, `subvec` keeps a reference to the underlying vector so none of the items popped off are ever garbage collected. See the "Performance and Implementation Details" section below for more more details.

Contract

inputs

- "v" parameter is a vector. It derives that `(vector? v)` must be `true`. "v" can be empty but it cannot be `nil`.

- "start" and "end" are numbers in the integer range (from `Integer/MIN_VALUE` to `Integer/MAX_VALUE`) representing the index at which to start the sub-vector (inclusive) and the index to end the sub-vector (exclusive).

notable exceptions

- If "v" is not a vector, then a `ClassCastException` is thrown.
- If "v" is `nil`, then a `NullPointerException` is thrown.
- If "start" or "end" cannot be coerced to a `Number` than a `ClassCastException` is thrown.
- If "start" or "end" are beyond the `Integer` range, then `IllegalArgumentException` is thrown.
- If "start" < 0 or "start" >= `(count v)` then an `IndexOutOfBoundsException` is thrown.
- If "end" is provided and "end" < "start" or "end" > `(count v)` then an `IndexOutOfBoundsException` is thrown.

outputs

- `subvec` creates the sub-vector starts as the zero based position "start" inclusive and runs to the zero based position "end" exclusive. if "end" is provided, otherwise it runs until the end of "v".
- If the "end" is omitted, the sub-vector contains the range starting from the element at index "start" (inclusive) up to `(count v)` (exclusive):

```
(subvec [1 2 3 4] 1) ; ❶
;; [2 3 4]
```

❶The "end" argument is optional. When "end" is not given, `(count v)` is used as a default.

Examples

`subvec` is an efficient solution to "remove" an element from a vector. Vectors are immutable data structures, so "removing" an element implies creating a new vector from two sub-vectors where the element to remove has been left out:

```
(defn remove-at [v idx] ; ❶
  (into (subvec v 0 idx)
        (subvec v (inc idx) (count v))))
```

```
  (remove-at [0 1 2 3 4 5] 3) ; ❷
  ;; [0 1 2 4 5]
```

❶into can be used to the first sub-vector to include the second one, starting from the given index `idx` plus 1.

❷ `remove-at` returns a new sub-vector that holds a reference to "v" indefinitely. If "v" is large, consider transforming the result back into a `vector`. See the performance section for more information.

`subvec` can be used for recursion, in a similar way the first/rest is used to advance a sequence. The following `norm` function calculates the norm of a vector [211]:

```
  (defn norm [v]
    (loop [v v
           res 0.]
      (if (= 0 (count v))
        (Math/sqrt res)
        (recur (subvec v 1) ; ❶
               (+ res (Math/pow (nth v 0) 2))))))

  (norm [-2 1])
  ;; 2.23606797749979
```

❶ `subvec` with the input vector and 1 as arguments is similar to the effect of rest on sequences.
❷The "head" of the vector is accessed via nth.

`subvec` is a natural choice for divide and conquer algorithms on vectors. As an example, `subvec` can be used to implement a recursive purely functional merge sort [212]:

```
(defn- merge-vectors [v1-initial v2-initial cmp]
  (loop [result []
         v1 v1-initial
         v2 v2-initial]
    (cond
      (empty? v1) (into result v2)
      (empty? v2) (into result v1)
      :else
      (let [[v1-head & v1-tail] v1
            [v2-head & v2-tail] v2]
        (if (cmp v1-head v2-head)
          (recur (conj result v1-head) v1-tail v
          (recur (conj result v2-head) v1 v2-tai

(defn merge-sort
  ([v]
   (merge-sort v <=))
  ([v cmp]
   (if (< (count v) 2)
```

```
      v
      (let [split (quot (count v) 2)
            v1 (subvec v 0 split)          ; ❷
            v2 (subvec v split (count v))]
        (merge-vectors (merge-sort v1 cmp) (merge

(merge-sort [2 1 5 0 3])
;; [0 1 2 3 5]

(merge-sort [[2 :b] [2 :a] [1 :c]] #(<= (first %
;; [[1 :c] [2 :b] [2 :a]]
```

❶ `merge-vectors` takes two sorted vectors ("v1-initial" and "v2-initial") and "cmp" a comparison function and merges the vectors into one sorted vector. It compares the first element of each vector and the element that is less according to the comparison function is added to the "result" vector. This is repeated recursively until all the elements have been merged.

❷ `subvec` is used to split the vector "v" in half. Each half is sorted before being merged into the final vector.

❸One of the key properties of the merge sort algorithm is that it is a stable sort, which means that if two elements have equal keys their relative order is unchanged. Here the key of both [2 :b] and [2 :a] is 2, so their order is left unchanged.

See also

- vec and vector also produce a persistent vector from a seqable collection. Use `vec` when the creation of the new vector does not involve a subset from another vector. Use vector to specify the element that should belong to the vector.
- vector-of to create a vector of a primitive types. Use `vector-of` if space efficiency is the biggest concern.
- into was used throughout the chapter to join sub-vectors back together.

Performance considerations and implementation details

=> `O(1)` **time and space**

`subvec` is a very efficient way to slice a vector into parts that is independent from the length of the vector. Although `subvec` is constant time, it still comes with an associated cost. Let's revisit the `norm` example to compare the `subvec` based solution to an index based solution:

```
(require '[criterium.core :refer [quick-bench]])

(defn norm [v] ; ❶
  (loop [v v
         res 0.
         idx (dec (count v))]
```

```
    (if (< idx 0)
      (Math/sqrt res)
      (recur (subvec v 0 idx)
             (+ res (Math/pow (peek v) 2))
             (dec idx)))))

(let [v (vec (range 1000))]
  (quick-bench (norm v)))
;; Execution time mean : 91.908294 µs ; ❷

(defn norm-idx [v] ; ❸
  (loop [idx (dec (count v))
         res 0.]
    (if (< idx 0)
      (Math/sqrt res)
      (recur (dec idx)
             (+ res (Math/pow (nth v idx) 2))))))

(let [v (vec (range 1000))]
  (quick-bench (norm-idx v)))
;; Execution time mean : 15.174786 µs ; ❹
```

❶ `norm` was slightly modified compared to the version in the examples. The introduction of the index `idx` in the parameters of the loop-recur avoids the use of count at each iteration to count the remaining elements in the sub-vector.

❷ The timing for a medium size vector of 1000 elements is about 91 micro-seconds (10^{-6} seconds)..

❸ `norm-idx` is very similar to `norm`. The changes are removing the sub-vector and using nth to fetch the element at `idx`.

❹ `norm-idx` is roughly 6 times faster. Considering that the loop is identical but missing sub-vector (the difference between `nth` and peek is negligible) the slow-down is determined by the use of `subvec` alone.

Thanks to immutability, the view created by `subvec` is isolated from additional changes happening to the original vector. The isolation is not perfect though and can result in subtle side effects dealing with large vectors. The newly created sub-vector is essentially a wrapper around the original vector screening out the unwanted elements, so no copying of elements occurs and hence `subvec` completes in constant time and space. The generated sub-vector maintains a reference to the original vector though, which can prevent elements being garbaged collected. The following example illustrates the problem: 2 sub-vectors are created with the intention to join them later:

```
(defn bigv [n]
  (vec (range n)))

(let [v1 (subvec (bigv 1e7) 0 5) ; ❶
      v2 (subvec (bigv 1e7) 5 10)]
      (into v1 v2))
;; OutOfMemoryError GC overhead limit exceeded ;

(let [v1 (into [] (subvec (bigv 1e7) 0 5)) ; ❸
      v2 (into [] (subvec (bigv 1e7) 5 10))]
      (into v1 v2))
;; [0 1 2 3 4 5 6 7 8 9] ; ❹
```

❶ `subvec` is used to cut a tiny slice of a much larger vector. The large vector is not referenced anywhere else, but is kept alive by a reference living inside the `subvec` implementation.

❷Depending on the JVM settings, you might need to tweak the size of the larger vector to see the out of memory problem shown here. The JVM was started with 512Mb of heap size in this case.

❸Each subvector is transferred into a new vector instance, so their inner vector reference can be garbage collected.

❹No out of memory problem is happening with this change.

[210] "The Joy of Clojure by Michael Fogus and Chris Houser. Chapter 5.2.7"
[211] en.wikipedia.org/wiki/Norm_(mathematics)
[212] en.wikipedia.org/wiki/Merge_sort

12.11 SUMMARY

A vector is an all-round data structure that is commonly found in many languages. Clojure's vector implementation is thread-safe, immutable and perform well in most situations. In this chapter we've seen the function specifically dedicated to vectors, but collections and sequences functions also work on vectors (with differences in performance and intent).

The choice between using vectors or sequences depends on many factors: speed, memory consumption and laziness implications to cite a few. One important aspect to consider (and often a problem for Clojure beginners) is to avoid mixing between different abstractions by forcing a sequence into a vector (or the other way around). After making a choice it's important to maintain consistency. If that cannot be achieved, prefer delegating the choice to the user by offering multiple version of the same function dedicated to each data structure (see for example mapv or filterv).

13 SETS

A set is a data structure that can store unique items without a particular ordering. Sets are widely used in computer science and have a strong connection with mathematical set theory. Clojure offers two types of sets with their corresponding initializer functions:

- hash-set is the most used, offering fast lookup, a special syntax literal `#{}` and a transient version. `hash-set` is implemented on top `clojure.lang.PersistentHashSet` which in turn is a thin layer on top of `clojure.lang.PersistentHashMap`, the same of hash-map. Both `hash-set` and `hash-map` are instances of Hash Array Mapped Trie [213]. `hash-set` offers near constant time lookup (at `O(log32N)`), addition and removal of items.
- sorted-set also guarantees uniqueness of items but additionally maintains ordering based on a comparator. Their implementation is based on Red Black trees (see subseq call-out section) and offers a well balanced logarithmic access (at `O(log2N)`).

Both set types can be used as functions (especially as predicates) to verify the presence of items with a concise syntax:

```
((sorted-set 5 3 1) 1) ; ❶
;; 1

(some #{1 2 3} [0 4 6 8 1]) ; ❷
;; 1
```

❶An example of using a sorted-set as a function. The `sorted-set` looks up the argument in the set and returns it, if present. It returns `nil` or a default value otherwise.

❷Another idiomatic use of sets with some to determine if at least one of the items from the input vector is in the set. In this case, the number "1" is present and returned.

The `clojure.core` namespace contains some basic functions to create a set (hash-set, Set, sorted-set) or remove elements from a set (disj). conj and into also works on sets (along with many other collection types). Functions like union, intersection or difference are available from the `clojure.set` namespace.

The chapter closes with an overview at functions like `project`, `select` or `index` (again from `clojure.set`) which are

dedicated to "relations". A relation is representation of data using a set of maps inspired by relational databases.

13.1 HASH-SET

`function` `since 1.0`

Listing 13.1. → *Data Structure, Sets, Constructor*

```
(hash-set [& keys])
```

`hash-set` is the main initializer function for Clojure unsorted sets. Note that any duplicate items is removed from the resulting set:

```
(hash-set :yellow :red :green :green) ; ❶
;; #{:yellow :green :red}
```

❶ `hash-set` takes any number of values. Note that `:green` is present twice in the input but only once in the resulting set.

Contract

input

- "keys" can be any number of items of any type.

output

- returns: a `clojure.lang.PersistentHashSet` instance containing the given "keys". It returns an empty set when invoked without arguments. If multiple instances of the same item (as per equality semantic) are present, only one instance is added to the set.

Examples

`hash-set` (like hash-map) has a syntax literal `#{}` to enable quick creation of sets.

```
(= #{3 2 1} (hash-set 1 2 3)) ; ❶
;; true
```

❶The reader literal `#{}` expands to create the related `hash-set` data structure.

Note however that the syntax literal `#{}` does not take care of duplicates automatically:

```
#{:yellow :red :green :green} ; ❶
;; IllegalArgumentException Duplicate key: :gree
```

❶Differently from `hash-set`, the set syntax literal does not allow duplicate keys and doesn't create the set, throwing exception.

The syntax literal enables elegant use of sets as predicates, like in the following example using some:

```
(some #{:x :c} [:a :b :c :d :e]) ; ❶
;; :c

(some #{:x :y} [:a :b :c :d :e]) ; ❷
;; nil
```

❶A set created with `hash-set` or the syntax literal `#{}` is also a function. As a function the set verify if the given argument is present in the set or not. `some` applies the set as a predicate on all the elements in the input collection stopping at the first one returning something different from `nil`.
❷If there are no elements from the collection in the set, `some` returns `nil`.

Differently from the syntax literal, `hash-set` is useful when the input items are not compile-time constants (such as numbers, strings, keywords etc.) but need evaluation:

```
#{(rand) (rand) (rand)} ; ❶
;; IllegalArgumentException Duplicate key: (rand

(hash-set (rand) (rand) (rand)) ; ❷
;; #{0.53148213003 0.7171734431 0.5055531620}
```

❶We are trying to create a set with 3 random numbers in it. The syntax literal `#{}` treats the input items as compile-time constants and only evaluates the expression `(rand)` after it is added to the set. This results in a duplicate key exception.

❷Using `hash-set` we make sure input elements are evaluated before entering the set, avoiding unexpected compile-time errors.

Let's clarify the meaning of uniqueness of the items in a set in the context of metadata. If multiple instances of the same value are added to the set and they have different metadata, `hash-set` retains the metadata from the first item:

```
(def set-with-meta
  (hash-set ; ❶
    (with-meta 'a {:pos 1})
    (with-meta 'a {:pos 2})
    (with-meta 'a {:pos 3})))
```

```
set-with-meta ; ❷
;; #{a}

(meta (first set-with-meta)) ; ❸
;; {:pos 1}
```

❶We use `hash-set` as usual. The items are all the same symbol `a` but with different metadata.

❷When we check the content of the set, we see a single element as expected.

❸The metadata on the element are the one of the element that was first inserted. This behavior is similar to hash-map treatment for multiple instances of the same key.

The powerset of a set

Set theory is an important branch of mathematics studying collection of objects and their relationship. It contains a rich set of functions and Clojure provides a few in the standard library: `union`, `difference` and `intersection` are available in the `clojure.set` namespace and are designed for set manipulation.

An interesting problem related to sets is generating the set of all subsets of a set, also known as the powerset. We are going to explore the problem of generating the powerset, as it forces us to use several functions and concepts related to `hash-set`. The problem is the following: given an initial group of items "s", the "powerset-of-s" contains all the unsorted combinations of those items:

```
(def s #{:a :b :c}) ; ❶
```

```
(def powerset-of-s ; ❷
  #{#{} #{:a} #{:b} #{:c} #{:a :b} #{:a :c} #{:b :c} #{:a :b :c}}
```

❶Our input is a set of 3 elements. We want to find all the combinations of the 3 elements irrespective of ordering, including the empty set `#{}`. If we were to take order into account, we should more specifically talk about permutations instead of combinations.

❷We can see here the expected output. Note that we are using set literals `#{}` for the outer and inner sets to enforce uniqueness of items.

The powerset of a set contains `(Math/pow 2 (count s))` items (the example above contains 2^3 elements including the empty set), where `(count s)` is the size of the input. There are several ways to calculate the powerset. With reference to the powerset of the 3 items `:a`, `:b` and `:c`, we can start by observing the following equivalence (pseudo-code):

```
[[]   [:a]    [:b]    [:a :b]]     U  ; ❶
[[:c] [:a :c] [:b :c] [:a :b :c]]  =  ; ❷

————————————————————————————————————

[[] [:a] [:b] [:c] [:a :b] [:a :c] [:b :c] [:a
```

❶The `U` operator means union. The first term of the union is the `powerset(:a, :b, :c)` after removing all the subsets containing the item `:c`. The item `:c` has been selected at random.

❷The second term of the union are all the other subsets that instead contain the previously removed item `:c`. Note that this second subset can be derived from the first one by adding `:c` to each item.

❸The union `U` results in the complete `powerset(:a, :b, :c)`.

A more formal definition of what we observe in the example above is: the `powerset(:a :b :c)` can be assembled together from the union of `powerset(:a :b)` with the set obtained from `powerset(:a :b)` once we've added `:c` from each element. We can use this observation to recursively build a powerset:

```
(require '[clojure.set :refer [union]])

(defn powerset [s]
  (when-first [x s] ; ❶
    (let [p (or (powerset (disj s x)) (hash-set
      (union p (set (map conj p (repeat x))))))))
```

```
  (powerset #{1 2 3}) ; ❹
  ;; #{#{} #{3} #{2} #{1} #{1 3 2} #{1 3} #{1 2} #
```

❶when-first takes the first element from the input set and, at the same time, checks if it's `nil`. When the input set "s" is empty, we're done with the recursion.

❷disj is used next to remove the first element from the set. The recursive `powerset` call potentially returns `nil` after reaching the end of the set (the result of `when-first`), so we provide the initial value for the computation using or. This is the empty `hash-set` containing another empty set. We could have written `{{}}` instead, but the double nested constant literal is less readable.

❸We have now the result of calling `powerset` on "s" (minus one item) available as the local binding "p". We can now apply the rest of the observation: the `clojure.set/union` function is applied to "p" and the result of merging back the removed element into each item of "p".

❹The test confirms that the `powerset` function produces the expected 8 output items.

This `powerset` formulation faithfully implements the initial observation that we made on what constitutes a "powerset", but it's not tail-recursive. It takes a top-down approach by defining

the powerset from it's final content into the empty set by consuming the stack and building the actual result on the way back. If we want to express it as tail-recursive, we need a way to accumulate results from the bottom-up, starting from the empty set and gradually adding elements to each subset. The observation is the same but reads backward: the next `powerset` is equal to the union of the previous one plus the set where each item has been added a new element. The following formulation takes this approach resulting a better performing and more concise code:

```
(defn powerset [items]
  (reduce ; ❶
    (fn [s x]
      (union s (map #(conj % x) s))) ; ❷
    (hash-set #{})
    items))

(powerset #{1 2 3}) ; ❸
;; #{#{} #{3} #{2} #{1} #{1 3 2} #{1 3} #{1 2} #
```

❶Instead of custom recursion or loop-recur, we can now delegate recursion to reduce and build the incremental results starting from the empty set.
❷Each `reduce` recursion we are presented with the partial view of the powerset so far and the next element "x". We can proceed to apply the observation using `union` of the current powerset "s" with all element in "s" after adding the new element "x".
❸We confirm hat the new formulation is returning the same results.

See also

- Set is optimized to create a set starting from an already existing collection. Prefer Set to `hash-set` with apply to create a set from a collection.

- sorted-set maintains the order of the items in a set through a comparator (default or custom). When the set is iterated, the items are returned following the ordering generated with the comparator. Use `sorted-set` instead of `hash-set` if you require ordering of the elements in a set. Note that `sorted-set` is not based on insertion order (like array-map for instance). If you need insertion order, the closest option is to use a vector and remove duplicates with distinct before iteration.

Performance considerations and implementation details

=> `O(n)` **linear, with n number of items**

=> `O(n)` **space**

`hash-set` has a very similar performance profile to hash-map. Similarities go beyond performance, as `hash-set` implementation is based on `PersistentHashMap`, the same as `hash-map`. The way a set collapse into a map is by having each item in the set correspond to a key-value pair inside the underlying map, with the key associated to itself.

The following benchmark compares `hash-set` to the other ways of creating a large set. We can see very small difference

between them, mainly because they all use a transient version of the set to remove immutability overhead during construction:

```
(require '[criterium.core :refer [bench]])

(let [items (into [] (range 100000))] ; ❶
  (bench (apply hash-set items)))
;; Execution time mean : 20.470692 ms

(let [items (into [] (range 100000))] ; ❷
  (bench (into #{} items)))
;; Execution time mean : 18.549452 ms

(let [items (into [] (range 100000))] ; ❸
  (bench (set items)))
;; Execution time mean : 19.287860 ms
```

❶The first benchmark uses `hash-set` with apply to create a large set. `apply` transforms the arguments in a sequence and each element is added to the set. Internally, items are added to a transient version of the set.

❷into supports transients. The multiple conj operations can now happen on a mutable instance.

❸Set is a specialized function to transform other collections into a set. It also uses transients.

Please also check Set performance section for additional benchmarks related to different collection types and percentage of distinct elements in the input.

[213] HAMT, or Hash Array Mapped Trie, is a tree-like data structure suitable for implementing persistent collections. We introduced its general properties at the beginning of the vector chapter.

13.2 SET

`function` `since 1.0`

Listing 13.2. → *Sets, Set creation*

```
(set [coll])
```

`set` creates a new `clojure.lang.PersistentHashSet` (the Clojure set type) from the items in the given input collection:

```
(set [1 2 3 4 1 4]) ; ❶
;; #{1 4 3 2}
```

❶ `set` "converts" the input vector into a new set instance. Note that the output set doesn't have a specific ordering and doesn't contain duplicates.

Along with hash-set and into, `set` is the main way to create new set instances.

Contract

input

- "coll" is the only mandatory argument. "coll" can be any collection including the most common Java iterable types but excluding transients.

notable exceptions

- `IllegalArgumentException` when "coll" is not a collection.

output

- returns: a new set instance containing the items from the input collection. If "coll" is empty or `nil`, it returns an empty set. All duplicated items from "coll" (if any) are removed in the output set. If "coll" is already a set the input is passed through without transformations, but the metadata, if any, are removed.

Examples

A typical use of `set` is to transform an existing collection, removing any duplicates in the process and preventing new ones in future operations. Note that `set` implies the creation of a new independent set, so if any metadata is present in the input, it is removed on purpose:

```
(def input-set (with-meta #{} {:original true}))

(meta input-set) ; ❷
;; {:original true}

(meta (set input-set)) ; ❸
;; nil
```

❶The original input set has some metadata attached using with-meta.

❷We can see the metadata anytime using meta.

❸The input set is used as an input for the `set` function. We can see that the metadata are stripped away.

Also note that if the input is a sorted-set it is not transformed into an unordered `hash-set`:

```
(type (set (sorted-set 8 7 4 2 1 3))) ; ❶
;; clojure.lang.PersistentTreeSet
```

```
(type (into #{} (sorted-set 8 7 4 2 1 3))) ; ❷
;; clojure.lang.PersistentHashSet
```

❶Calling `set` on a sorted-set does not change the type to `clojure.lang.PeristentHashSet`.
❷into performs the transformation from ordered to unordered.

Transformations with `set` are also useful in conjunction with contains?. In the following example, we setup a simple "honeypot" mechanism to prevent fraudulent use of a web form [214]. The honeypot consists of an input HTML tag that is not visible to human users but appears legitimate when bots parse the page.

Once the web request comes in as a hash-map we need to verify if it contains a specific value encoded for the honeypot input. Depending on the page, there could be one or more honeyput fields with legitimate names like "option1" or "option2":

```
(def honeypot-code "HP1234")

(def valid-request
  {:name "John"
   :phone "555-1411-112"
```

```
   :option1 ""
   :option2 ""}) ; ❶

(def fake-request
  {:name "Sarah"
   :phone "555-2413-111"
   :option1 "HP1234" ; ❷
   :option2 ""})

(defn honeypot? [req]
  (contains?
    (set (vals req))  ; ❸
    honeypot-code))

(honeypot? valid-request)
;; false

(honeypot? fake-request)
;; true
```

❶ `valid-request` contains a few honeypot fields that are correctly empty. A legitimate user can't see the corresponding input and can't make a choice.

❷ If we access the raw source of the same web page instead (like an automatic program would do), there is no way to distinguish honeypot inputs (at least in this simple example). The automatic program would then proceed to analyze the page and fill the input with the honeypot code (because it's mandatory choice for a radio-button for instance).

❸ A straightforward way to verify the presence of one or more honeypot codes, is to use `set` on the sequence of values from the request map. contains? supports sets (it would not work with a sequence or vector) to verify the presence of the honeypot code.

See also

- hash-set also creates new set data structures, allowing any number of items as arguments. In general, `(set coll)` should be preferred instead of `(apply hash-set coll)`.
- sorted-set is the ordered version of a `set`. The ordering is based on a comparator. The `comparator` is used to establish the relative ordering of the items in the set.

- contains? is often used with sets to verify the presence of an element.

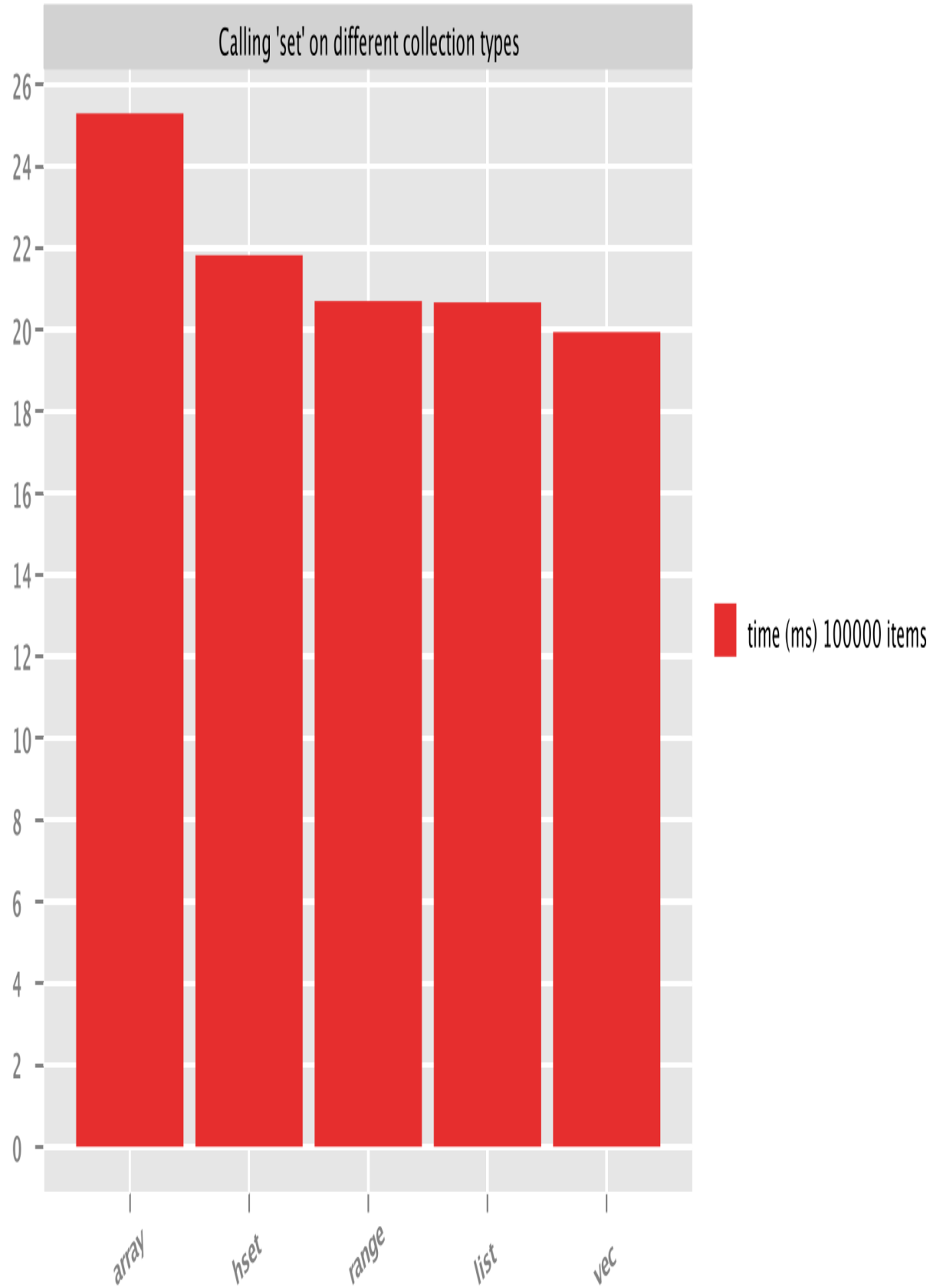
Performance considerations and implementation details

=> `O(n)` linear in the number of items

`set` performance profile is similar to hash-set and is linear in the number of items. `set` implements a loop to conj items into the new set using reduce, while `hash-set` executes a similar loop in Java.

The following chart shows `set` applied to different collections types of the same size (the chart also includes a native array). We can't see a big difference, as reduce is optimized for most of the collection types:

Figure 13.1. Chart showing benchmarks against different collection types.


Calling 'set' on different collection types
26
24
22
20
18
16
14
12
10
8
6
4
2
0
array
hset
range
list
vec
time (ms) 100000 items

In terms of comparison with `hash-set`, both functions use transients to populate a new mutable set and convert it into persistent before returning it. The differences in terms of pure speed are small, with some advantage for `set`:

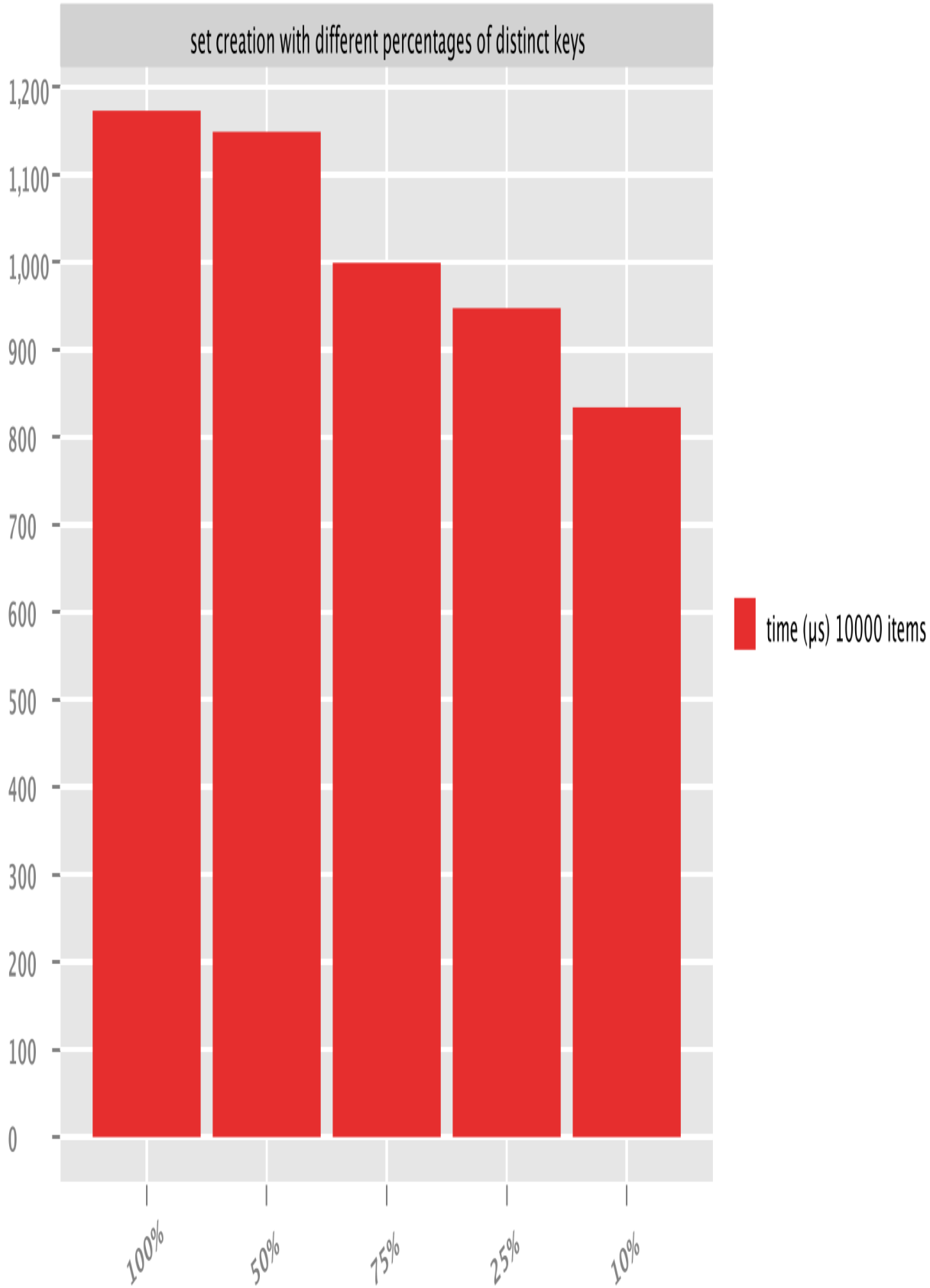
```
(require '[criterium.core :refer [quick-bench]])

(let [coll (range 10000)]
  (quick-bench (apply hash-set coll)))
;; Execution time mean : 1.282033 ms

(let [coll (range 10000)]
  (quick-bench (set coll)))
;; Execution time mean : 1.132248 ms
```

Let's now explore how `set` behaves related to the percentage of distinct elements. The chart below confirms that the more items are repeating in the input (clashing with already existing ones in the output) the faster the creation is:

Figure 13.2. Chart showing how creation speed changes related to the percentage of distinct items in the input (100% means no repeating items, 0% means the input contains the same repeating element).


set creation with different percentages of distinct keys
1,200
1,100
1,000
900
800
700
600
500
400
300
200
100
0
100%
50%
75%
25%
10%
time (µs) 10000 items

[214] A "Honeypot" in computing is a legitimate mechanism to interact with a service that allows to distinguish fraudulent use, see en.wikipedia.org/wiki/Honeypot_(computing) for more information

13.3 SORTED-SET AND SORTED-SET-BY

`function` `since 1.0 (sorted-set), 1.1 (sorted-set-by)`

Listing 13.3. → *Sorting, Data Structures, Unique Elements*

```
(sorted-set [& keys])
(sorted-set-by [comparator & keys])
```

`sorted-set` and `sorted-set-by` are initializers for ordered sets, a collection type similar to hash-set that also maintains ordering based on a comparator. We can build a new `sorted-set` passing the required elements as arguments:

```
(sorted-set "t" "d" "j" "w" "y") ; ❶
;; #{"d" "j" "t" "w" "y"}
```

❶Strings are comparable objects. `sorted-set` uses a default comparator to invoke `Comparable::compareTo()` method with two strings to know which one comes first.

We can force a different comparator than the default one using `sorted-set-by`:

```
(sorted-set-by #(compare %2 %1) "t" "d" "j" "w"
;; #{"y" "w" "t" "j" "d"}
```

❶When default ordering is not desirable, we can pass a new comparator to `sorted-set-by`. The compare function in the standard library is compatible with many Clojure and Java types and can be used with strings as well. If we want to invert the order, we just need to swap the arguments.

Contract

input

- "keys" can be any number of elements to be added to the newly created sorted set.
- "comparator" can be any object implementing the `java.util.Comparator` interface. Clojure functions (with the exception of data structures used as functions) implement the `Comparator` interface, a nice trick to allow `<`, `>`, `>=`, `⇐` as comparators.

notable exceptions

- `ClassCastException` is thrown when any pair of "keys" is not comparable. In general keys are not comparable when they are of a different type (although different numeric types are comparable). For a more detailed list of corner cases, please see compare.

output

- returns: a new instance of `clojure.lang.PersistentTreeSet`, the class implementing sorted sets in Clojure. The new instance contains all distinct "keys" passed as input. When iterated, the sorted set returns "keys" in the order determined by "comparator" (or the default comparator if none given).



Note

Please note that, unlike normal hash-set, there is no transient version of `sorted-set` or sorted-map.

Examples

`sorted-set` and `sorted-set-by` guarantee their content does not contain duplicates. Like hash-map and hash-set, `sorted-map` maintains the metadata from the first inserted item in case of duplicates with different metadata:

```
(defn timed [s] ; ❶
  (let [t (System/nanoTime)]
    (println "key" s "created at" t)
    (if (instance? clojure.lang.IMeta s)
      (with-meta s {:created-at t})
      s)))

(def s (sorted-set (timed 'a) (timed 'a))) ; ❷
;; key a created at 206892376620199
;; key a created at 206892376884656

(meta (first s)) ; ❸
;; {:created-at 206892376620199}
```

❶We use the `timed` function on the items to store the creation date as part of the object metadata.
❷"s" is a `sorted-set` created with the same symbol "a" twice. Symbols support `metadata` along with most of the collection types.
❸We can see the retained metadata is the one attached to the first symbol.

`sorted-set` is useful to maintain order as the set gets updated incrementally, as each new item is inserted in order following the comparator. In the next example we let users update a core dictionary of words with their own custom words and spellings. If we store the dictionary in a `sorted-set` there is no need to sort after each update:

```
(require '[clojure.string :refer [split-lines]])

(def dict ; ❶
  (atom
    (->> "/usr/share/dict/words"
      slurp
      split-lines
      (into (sorted-set)))))

(defn new-word [w] ; ❷
  (println "Could not find the word:" w)
```

```
  (println "Add word to dictionary? [y/n]")
  (when (= "y" (read-line))
    (swap! dict conj w)
    (take 5 (subseq @dict >= w)))) ; ❸

(defn spell-check [w]
  (if (contains? @dict w) ; ❹
    (println "Word spelled correctly")
    (new-word w)))

(defn ask-word []
  (println "Please type word:")
  (when-let [w (read-line)]
    (spell-check w)))

(ask-word) ; ❺
;; Please type word:
;; google
;; Could not find the word: google
;; Add word to dictionary? [y/n]
;; y
;; word added (google googly googol googolplex g

(ask-word)
;; Please type word:
;; google
;; Word spelled correctly
```

❶ `dict` is a top level definition in the current namespace that holds the initial load of the words from a local file. `/usr/share/dict/words` is a file present on most Unix file system. After splitting the file into lines, we store the corresponding words into a `sorted-set`. The set is wrapped in an atom to allow controlled mutation.

❷ `new-word` is used when we discover a word that is not in the dictionary already. The function asks the user if they want the word to be added and proceed to update the `atom`.

❸subseq is a perfect choice to extract a portion from the ordered set, as it avoids a linear scan to reach the target word. We can provide a quick feedback about the position of the word in the dictionary.

❹Another idiomatic operation on `sorted-set` is contains? that we use here to verify if a word is in the set.

❺We can see an user interacting with the example to add a new word to the dictionary.

Custom comparators

Thanks to Clojure extended equality, we can also use collections in a `sorted-set`:

```
(sorted-set [1 "b" :x] [1 "a" :y]) ; ❶
;; #{[1 "a" :y] [1 "b" :x]}
```

```
(sorted-set-by compare [1 "b" :x] [1 "a" :y]) ; ❷
;; #{[1 "a" :y] [1 "b" :x]}
```

❶equality of vectors is the same as equality of each single item positionally. After checking that "1" is the same, the comparison continues with the next item. Since "a" comes before "b" the entire vector containing "a" is moved before the entire vector containing "b".

❷This second formulation using `sorted-set-by` with compare is equivalent to the first. You can use `sorted-set-by` with a different comparator in case the default behavior is not what you want.

If the default compare is not sufficient, we can pass a custom one, for example to sort by first or last element only. At this point, it's useful to understand how the comparator works. There are two distinct phases involved in adding a new element to a `sorted-set`:

- Skip the item if it's already in the set. To do this, the comparator is called on each existing item against the new one. If any comparison returns "0" (which means they are the same for a possibly custom definition of equality), then the element is not added to the set.
- If not already in the set, modify the set structurally to accommodate the new item in the right place. This phase uses the comparator again to decide where the new item should be added.

One important aspect to consider is that equality and relative ordering are not necessarily the same. In the following example we want to create a `sorted-set` of vectors so they are ordered by count, larger size first:

```
(sorted-set-by ; ❶
  (fn [a b] (compare (count b) (count a)))
  [1 :a] [:b] [3 :c] [:v])
;; #{[1 :a] [:b]}
```

❶A naive attempt at creating a `sorted-set` of vectors ordered by relative size.

The problem with this approach is that some items in the input are not appearing in the output, which is correct from the point of view of the comparator, since it returns "0" when the items have the same size.

To prevent the problem, the custom comparator should distinguish between the two aspects: equality and relative ordering. The following example introduces a condition in case two sizes are the same. In that case we want to verify if the elements are also equal:

```
(sorted-set-by
  (fn [a b]
    (let [cmp (compare (count b) (count a))]
      (if (zero? cmp) ; ❶
        (compare a b)
        cmp)))
  [1 :a] [:b] [3 :c] [:v])
;; #{[1 :a] [3 :c] [:b] [:v]}
```

❶ It's not enough to compare sizes, as the two items "a" and "b" could be equal in size but not content-equal.

The comparator above produces the expected behavior. We can now write a more concise version of the same behavior using wrapping vectors. This is possible because vector equality compares items by index and here we create a wrapping vector with the count first, then the actual content:

```
(sorted-set-by
  (fn [a b] ; ❶
    (compare [(count b) a] [(count a) b]))
  [1 :a] [:b] [3 :c] [:v])
;; #{[1 :a] [3 :c] [:b] [:v]}
```

❶ A more concise version of the comparator that uses counts first and the actual content next to compare vectors.

See also

- hash-set and `sorted-set` serve different purposes and have different performance profiles. Use `hash-set` if you are not interested in ordering but still need uniqueness guarantee.
- sorted-map are very closed cousins to `sorted-set` and `sorted-set-by`. They are actually based on the same implementation. Use a `sorted-map` when it makes sense to have key value pairs.
- compare was mentioned many times in this chapter. It is probably the most flexible tool for comparison as it expands equality semantic to most of the Clojure types.

Performance considerations and implementation details

=> `O(N)` linear in the number of items to insert

`sorted-set` creates a new `clojure.lang.PeristentTreeSet` object, which is a thin wrapper around `clojure.lang.PersistentTreeMap`. The implementation of a set on top of a map follows a simple principle: each item in the set becomes a key-value pair inside the map (where the key is the same as the value).

It follows that `sorted-set` and `sorted-set-by` have very similar performance profile to `sorted-map` and `sorted-map-by` : insertion of new elements is linear in the number of elements to add and other operations are `O(log32N)` . The reader is invited to review the sorted-map chapter for additional performance information.

13.4 DISJ

`function` `since 1.0`

Listing 13.4. → *Set handling, Removing items*

```
(disj
  ([set])
  ([set key])
  ([set key & ks]))
```

`disj` removes one or more elements from a set (sorted or not sorted):

```
(disj #{1 4 6 8} 4 8) ; ❶
;; #{1 6}
```

```
  (disj (sorted-set-by > 1 4 6) 4) ; ❷
  ;; #{6 1}
```

❶ `disj` is used to remove the number "4" and the number "8" from the input set.
❷ `disj` can be used similarly on sorted set.

Contract

input

- "set" is the argument name for the input set. When this is the only argument, it can be of any type including `nil`. When additional arguments are present, then "set" is expected to implement the `clojure.lang.IPersistentSet` interface. There are two Clojure built-in types implementing `IPersistentSet`, those created using Set and sorted-set.
- "key" and "ks" are any number of items to remove. They can be of any type and they are optional arguments.

notable exceptions

- `ClassCastException` when "set" does not implement the `clojure.lang.IPersistentSet` interface. To understand if the "set" argument implements the right interface, you can use the `set?` function or alternatively, `(instance? clojure.lang.IPersistentSet set)` should return `true`.

output

- returns: the same input "set" with element "key" or "ks" removed. If "set" is the only argument it returns the argument itself. If "set" is `nil`, then `disj` returns `nil` even if "key" or "ks" are present.

Examples

In the first example, `disj` is used to detect the presence of invalid values. We receive a vector containing configuration values and a list of allowed values. Here's how `disj` could detect the presence of unwanted values:

```
(defn valid? [allowed values]
  (empty?
    (apply disj (set values) allowed))) ; ❶
```

```
(def allowed [:a :b :c])

(valid? allowed [:c :c 1 :a]) ; ❷
;; false

(valid? allowed [:c :c :a]) ; ❸
;; true
```

❶After transforming the input values into a Set (which also removes any duplicate), we repeatedly remove all allowed values with `disj`. If anything is left, it is not part of the list of valid values.
❷When `valid?` is used against a collection that includes an invalid number, it returns `false`.
❸after removing the invalid number, `valid?` returns `true`.

In the next example we are going to use a Set to maintain a list of open connections. Each connection corresponds to a local port served by an echo sever (a server that listen for incoming connections and replies by repeating the input). We maintain a list of used ports in a globally accessible `hash-set` of ports and start a new listener only if the port is free:

```
(require '[clojure.java.io :as io])
(import '[java.net ServerSocket])
```

```
(def ports (atom #{})) ; ❶

(defn serve [port]
  (if (= @ports (swap! ports conj port)) ; ❷
    "Port already serving requests."
    (future ; ❸
      (with-open [server (ServerSocket. port) ; 
                  socket (.accept server)
                  writer (io/writer socket)]
        (.write writer (.readLine (io/reader soc
        (.flush writer))
      (swap! ports disj port)))) ; ❺
```

❶ `ports` is an atom wrapping the initially empty set of ports.

❷ Before we create a new server listener, we need to verify that the port is already in the set, which means it's already in use. We rely on the uniqueness property of sets to verify if, after calling conj, `ports` content has changed. `swap!` always returns the `atom` after the changes.

❸ If `ports` does change, it means the requested port is not in use. In this case we want to create a Java `ServerSocket` object in a separate thread with future, so the current call doesn't hang on the main thread.

❹ What follows is the Java interoperability necessary to create a new `ServerSocket` object with the given port, read from the socket and write back into the socket before closing it. We need to remember to "flush" anything going to the output stream, because any buffered writer (such as the one returned by `io/writer`) have an internal cache of bytes.

❺ Before closing the thread, we make sure to remove the port that was served from `ports`. This allows following requests to reuse the same port.

Note

Please note that to simplify the previous example, the condition that checks the presence of a free port and the following start of a new `ServerSocket` is not in the same transaction. Many concurrent requests on the same port might actually result in exceptions because the port is already in use.

To see the echo-sever in action, we need a mix of Clojure from the REPL and a command like Telnet [215]. After calling `serve` at the REPL, we need a corresponding `telnet` from the command line to unblock the port:

```
(serve 10001) ; ❶
;; #object[future_call 0x41da {:status :pending,

(serve 10001) ; ❷
;; "Port already serving requests."

;; telnet localhost 10001 ; ❸
;; Connected to localhost.
;; Escape character is '^]'.
;; hello
```

```
  ;; hello
  ;; Connection closed by foreign host.
```

❶ `serve` on an unused port creates a new `future` instance which is returned in `:pending` state, as the `readLine` call from standard input is blocking. The thread is ready to receive a request from port `10001`.

❷ A second call to `serve` on the same port, results in a message that the port is already in use.

❸ We use `telnet` from a command line. After establishing a connection, `telnet` waits for the input. The input ends after hitting "return" on the keyboard. If we type "hello" and hit "return" we can see another "hello" repeated just below.

See also

- dissoc is the equivalent operation for hash-map.
- `disj!` is the equivalent operation for transient sets.
- conj is the opposite of `disj`, but `disj` is not polymorphic like conj and only works on sets.
- difference is another option to remove multiple items from a set. `difference` takes the items to remove grouped in a set, while `disj` accepts them as distinct arguments.

Performance considerations and implementation details

=> `O(log32N)` **single item, hash-set**

=> `O(log2N)` **single item, sorted-set**

=> `O(n)` **linear in the number of arguments**

`disj` (similarly to dissoc) has a linear dependency on the number of keys to remove. In case of single key, the performance profile is different based on the type of set: `disj` for `hash-set` is close to constant time, more precisely `log32N` where "N" is the number of items in the set. `disj` for sorted sets is still logarithmic but with a different constant factor (the base is 2 instead of 32). In general, removing a single element from a set is a quick operation that shouldn't generate any major concern.

In case `disj` needs to be used to remove multiple arguments, there are alternatives to consider. In the following benchmark we measure `disj` and `clojure.set/difference`:

```
(require '[clojure.set :refer [difference]])
(require '[criterium.core :refer [quick-bench]])

(let [s (set (range 1000)) ; ❶
```

```
        xs (range 400 600)]
    (quick-bench (apply disj s xs)))
  ;; Execution time mean : 40.580589 µs

  (let [s (set (range 1000)) ; ❷
        xs (range 400 600)]
    (quick-bench (difference s (set xs))))
  ;; Execution time mean : 59.548475 µs

  (let [s (set (range 1000)) ; ❸
        xs (set (range 400 600))]
    (quick-bench (difference s xs)))
  ;; Execution time mean : 37.345393 µs
```

❶The first benchmark measures `disj` on a medium size set to remove 200 arguments. Note that we need to use apply to spread the collection of arguments.

❷ `clojure.set/difference` requires the items to remove to be in a set. The second benchmark assumes you don't have a set already, so we create a set with the arguments along with the call to `difference`.

❸In the final benchmark, we assume the optimal case in which the arguments are already a set.

The difference between `disj` and `clojure.set/difference` for this specific case is small, but you should use

`clojure.set/difference` if the items to remove are already in a set (the creation of a set slows the benchmark considerably).

If your application has a critical section that requires removing items from a set, you should look into transients:

```
(defn disj* [s & ks] ; ❶
  (persistent!
    (reduce disj! (transient s) ks)))

(let [s (set (range 1000)) ; ❷
      xs (range 400 600)]
  (quick-bench (apply disj* s xs)))
;; Execution time mean : 21.942524 µs
```

❶This version of `disj` called `disj*` transforms the set into a `transient` before removing items.
❷The benchmark confirms that removing items while the set is in `transient` state has positive effects on speed.

`disj*` is roughly 50% faster than normal `disj`, but remembers that this is true only if the number of items to remove is sufficiently large.

[215] Telnet is installed by default in many Linux distributions, Mac OS and Windows (although on Windows it might require configuration to enable it).

13.5 UNION, DIFFERENCE AND INTERSECTION

`function` `since 1.0`

Listing 13.5. → *namespace: clojure.set*

```
(union
  ([])
  ([s1])
  ([s1 s2])
  ([s1 s2 & sets]))

(intersection
  ([s1])
  ([s1 s2])
  ([s1 s2 & sets]))

(difference
  ([s1])
  ([s1 s2])
  ([s1 s2 & sets]))
```

`clojure.set/union`, `clojure.set/difference` and `clojure.set/intersection` are common operations on sets deriving from set theory [216]. They are not included in the core namespace, so they need a `require` instruction to become available:

```
(require '[clojure.set :as s])

(s/union #{1 2 3} #{4 2 6}) ; ❶
;; #{1 4 6 3 2}

(s/difference #{1 2 3} #{4 2 6}) ; ❷
;; #{1 3}

(s/intersection #{1 2 3} #{4 2 6}) ; ❸
;; #{2}
```

❶The `union` of two sets groups together all items from two sets removing duplicates.

❷The `difference` of two sets removes all items in the first set that are also present in the second set.

❸The `intersection` of two sets groups together all the items that are common between the sets.

Contract

input

- `union` is the only function accepting no arguments. `intersection` and `difference` require at least one set argument or `nil`.
- "s1" and "s2" can be `nil`, hash-set or sorted-set. If they are not `nil`, it follows that `(set? s1)` and `(set? s2)` return `true` and they should both implement the `clojure.lang.IPersistentSet` interface.
- "sets" is any additional sets following the same specification as "s1" and "s2".

notable exceptions

Most exceptions happen when arguments are not implementing `clojure.lang.IPersistentSet` interface:

- `UnsupportedOperationException` when an argument is not of type `set` and does not support count.
- `IllegalArgumentException` when an argument is not sequential.

output

`union`, `intersection` and `difference` return a new set instance or `nil`. The new set instance has the type of the first argument (`hash-set` or `sorted-set` in case of native set types, or possibly `nil`). The content of the result set depends on the operation:

- The `union` of "s1", "s2" and "sets" is the set containing the sum of all the unique elements from "s1", "s2" and any additional "sets".
- The `intersection` of "s1", "s2" and "sets" is the set containing all the common elements in "s1", "s2" or any additional "sets".
- The `difference` of "s1", "s2" and "sets" is the set containing all the items in "s1", minus the common items from "s2", minus other common elements from any additional "sets".



Warning

Use of `union`, `difference` and `intersection` on other types of collections could silently fail with unpredictable results. Please try to ensure the arguments are all sets before using them.

Examples

Let's illustrates a few interesting cases first. `intersection` results in empty set or `nil` if any of the sets is empty or `nil` respectively. It returns empty set even in the presence of `nil` if the last argument is an empty set:

```
(require '[clojure.set :as s])

(s/intersection #{1 2 3} #{} #{4 2 6}) ; ❶
;; #{}

(s/intersection #{1 2 3} nil #{4 2 6}) ; ❷
;; nil

(s/intersection #{1 2 3} nil #{}) ; ❸
;; #{}
```

❶ `intersection` returns an empty set if any of the arguments is empty.
❷Similarly, `intersection` returns `nil` if any of the arguments is `nil`.
❸But `intersection` returns empty set in presence of `nil` if the last argument is the empty set.

The effect of the presence of `nil` (or empty set) depends on where it appears in the argument list. In any case, if `nil` is one of the arguments, you should not rely on empty set equivalence

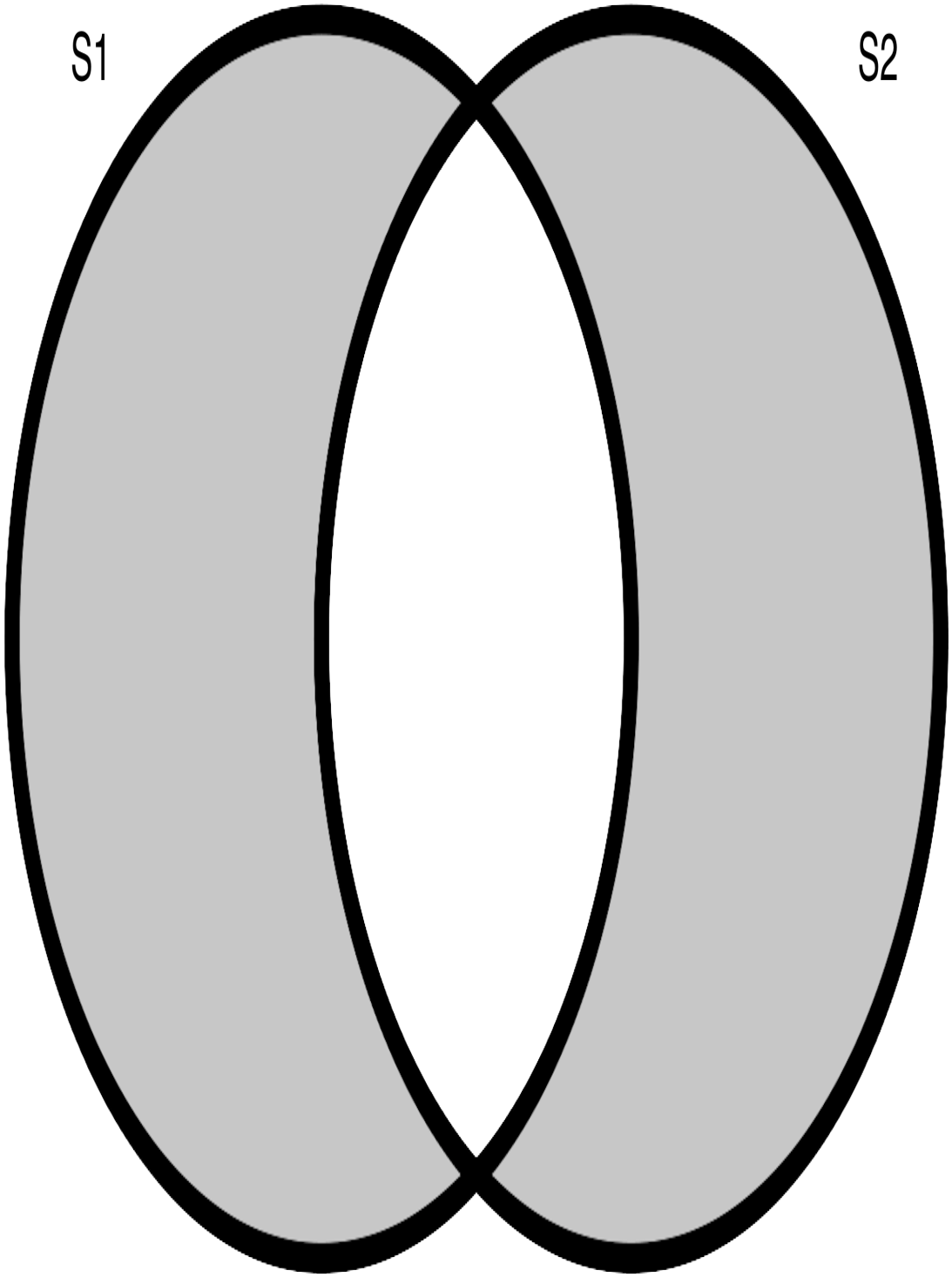
to implement a conditional statement. The safest option is to remove `nil`:

```
(apply s/intersection
  (remove nil? [#{1 2 3} nil #{4 2 6}])) ; ❶
;; #{2}
```

❶We use remove to get rid of potentially `nil` arguments before using `intersection`.

Building on the mathematical foundations of sets, we can create other interesting operations. The "symmetric difference" of two sets, for example, is the group of items in either sets that is not in their intersection. We can visualize the symmetric difference between the sets "S1" and "S2" using the following diagram:

Figure 13.3. Diagram that shows the symmetric difference between S1 and S2 (darker color).


S1
S2

The symmetric difference of "s1" and "s2" is equivalent to the `union` of the sets after removing their `intersection`:

```
(require '[clojure.set :refer [union difference

(defn symmetric-difference [s1 s2] ; ❶
  (difference (union s1 s2) (intersection s1 s2)

(symmetric-difference (sorted-set 1 2 4) #{1 6 8
;; #{2 4 6 8}
```

❶The `symmetric-difference` function implements the definition of symmetric difference almost literally, illustrating the good use of `union`, `difference` and `intersection`.

❷Note the use of `sorted-set` as first argument which produces an ordered set as output.

See also

- disj, when used with multiple arguments, works similarly to `difference`. Prefer `difference` when all arguments are part of a set. If the items to subtract are part of another collection type, the apply with disj is a possible option. There are minor performance implications to consider discussed in the disj section.

Performance considerations and implementation details

=> `O(n)` **linear**

`union`, `difference` and `intersection` are linear operations with "n" dependent on the function:

- `union` requires the iteration over all items of all arguments, so "n" is equals to the sum of the counts of all arguments.
- `difference` is linear in the count of the items in the first argument.
- `intersection` is also linear in the count of the items in the first argument.

[216] For additional information on set theory and other possible operations, please see the Wikipedia page: en.wikipedia.org/wiki/Set_theory

13.6 SUBSET? AND SUPERSET?

13.6.1 → *namespace: clojure.set*

This short section describes functions to verify containment relationship between sets. Given two sets "s1" and "s2", `subset?` returns `true` if all items in "s1" are also present in

"s2", while `superset?` returns `true` when "s1" contains all the items that are also in "s2":

```
(require '[clojure.set :as s]) ; ❶

(s/subset? #{1 2} #{1 2 3}) ; ❷
;; true

(s/superset? #{:a :b :c} #{:a :c}) ; ❸
;; true
```

❶ `subset?` and `superset?` lives in the `clojure.set` namespace.

❷The numbers "1" and "2" (the full content of the first set) are also in the second set.

❸The first set contains both ":a" and ":c" keywords from the second set.

Like other functions in this chapter, `subset?` and `superset?` should not be used with collection types other than sets or `nil`:

```
(s/superset? nil #{}) ; ❶
;; true
```

```
  (s/subset? #{0 3} [:a :b :c :d]) ; ❷
  ;; true
```

❶You should avoid `nil` arguments because they can give inconsistent results.

❷Similarly you should avoid collection types that are now set. In this example we are testing the presence of items at index "0" and "3" in the second vector argument, not the actual content.

13.7 SELECT, INDEX, RENAME, JOIN AND PROJECT

13.7.1 → *namespace: clojure.set*

This short section describes a group of specialized functions in the `clojure.set` namespace. These functions are inspired by relational algebra [217]: `select`, `index`, `project`, `rename` and `join`.

Relations are implemented in Clojure as sets of maps like the following:

```
  (def users ; ❶
    #{{:user-id 1 :name "john"   :age 22 :type "pe
```

```
     {:user-id 2 :name "jake"   :age 28 :type "co
     {:user-id 3 :name "amanda" :age 63 :type "pe

(def accounts  ; ❷
  #{{:acc-id 1 :user-id 1 :amount 300.45 :type "
    {:acc-id 2 :user-id 2 :amount 1200.0 :type "
    {:acc-id 3 :user-id 1 :amount 850.1 :type "d
```

❶A relation like `users` has a strong resemblance with the table in a relational database.

❷The `accounts` relation contains an "user-id" key. This key can be used to lookup the related record in another relation and is commonly referred as a "foreign key".

The resemblance of "users" and "accounts" to the tables in a modern database is not a coincidence. We can now use relation-oriented functions to perform interesting queries on relations:

```
(require '[clojure.set :as s])

(s/select #(> (:age %) 30) users) ; ❶

;; #{{:id 3, :name "amanda", :age 63, :type "per

(s/project ; ❷
  (s/join users accounts {:user-id :user-id}) ;
```

```
  [:user-id :acc-id :name])

;; #{{:user-id 2, :acc-id 2, :name "jake"}
;;   {:user-id 1, :acc-id 1, :name "john"}
;;   {:user-id 1, :acc-id 3, :name "john"}}

(s/project
  (s/join users accounts {:user-id :user-id})
  [:user-id :acc-id :type]) ; ❹

;; #{{:user-id 2, :acc-id 2, :type "saving"}
;;   {:user-id 1, :acc-id 1, :type "saving"}
;;   {:user-id 1, :acc-id 3, :type "debit"}}

(s/project
  (s/join users (s/rename accounts {:type :atype)
  [:user-id :acc-id :type :atype])

;; #{{:user-id 1, :acc-id 1, :type "personal", :a
;;   {:user-id 2, :acc-id 2, :type "company", :at
;;   {:user-id 1, :acc-id 3, :type "personal", :a

(s/index users [:type]) ; ❻

;; {{:type "company"}
;;  #{{:user-id 2, :name "jake", :age 28, :type
;;  {:type "personal"}
```

```
;;  #{{:user-id 3, :name "amanda", :age 63, :typ
;;    {:user-id 1, :name "john", :age 22, :type
```

❶ `s/select` is similar to the "select" construct in SQL. SQL is the structured query language used in relational databases. `s/select` filters relations based on a predicate (which is similar to the "where" clause in SQL).

❷ `s/project` only keeps the given keys from each map in a relation, removing all the others. It is similar to select-keys applied to all relations in the set.

❸ We can also `s/join` two relations, which creates the union of the relations based on a common value of one or more keys. In this specific example, an user map merges into the account map that has the same ":used-id" value. `s/join` would automatically join on `user-id`, even if we didn't pass the mapping explicitly as we did. Note that we don't see "Amanda" in the names, because no account belongs to her and there is no matching `user-id` in `accounts`.

❹ We have a problem of clashing keys between relations: if we join two maps with the same key, the last key to be merged override any previous values. What if we wanted to see both the user type and the account type?

❺ We can solve clashing keys in a join using `s/rename` to give a specific key a new name before `s/join`. We can see that now we have access to both types. This is equivalent to the "AS" renaming contract in SQL. Note that this time we didn't pass

an explicit key to join on, as `s/join` automatically uses `user-id` which exists in both relations.

❻ The final example shows how can we group a relation by a specific key using `s/index` (something similar to "GROUP BY" is SQL).

Relations and the functions operating on them enable a small but fully working in-memory database for Clojure. An in-memory database can be used on relatively small datasets like configuration, rules or other structured data that are limited to the to the lifetime of the application.

[217] The relational algebra defines the semantic of operations to handle data stored in a relational database. It was invented around 1970 at IBM. Please see en.wikipedia.org/wiki/Relational_algebra for more information.

13.8 SUMMARY

This chapter illustrates functions about sets, another important data structure in computer science. Basic set functions are part of the `clojure.core` namespace, but Clojure also offers additional functions dedicated to relational algebra [218] in the `clojure.set` namespace.

[218] Please see en.wikipedia.org/wiki/Relational_algebra.

PART 4. SOLVING SPECIFIC PROBLEMS

The last part of the book groups functions and macros dedicated to the solution of specific programming problems. Examples of such broad category are concurrency, polymorphic types, definition of variables, formatting and printing and many others.

This part of the book uses a less formal structure to describe functions and abandons the fixed template adopted in the previous parts. In doing so, the book wants to focus even more on practical aspects of everyday programming. For this reason, functions in the following sections are more likely to appear grouped by area of application.

The last chapter titled "the toolbox", takes this approach to the extreme giving just a brief summary of all the remaining namespaces in the standard library. This choice also reflects the fact that some functions have a very narrow field of application.

14 CONCURRENCY

Unlike other mainstream languages, Clojure main approach to concurrency does not require explicit locking [219]. Locking is still available as a low level option, but the default use of persistent data structures provides Clojure with alternative strategies to solve the hardest concurrency problems.

This chapter describes what functions in the standard library are dedicated to concurrency and how to use them. There are three main groups of them:

- future, promise and delay control threads in different ways. These functions are not connected to a specific way to handle state, nor they wrap any state themselves.
- ref, atom and agent are concurrency model designed to handle state, each one providing different guarantees in case of concurrent access. var and volatile! also belong to this group, but they are illustrated in other chapters for their role in scenarios other than pure concurrency.
- deref, validators and watchers are common features implemented in all concurrency models.

- Finally, locking is the last resort in terms of concurrency handling around critical sections of the code. The need for explicit locking is normally exceptional and relegated to some case of low level Java interoperability.

14.1 FUTURE

Note

This section also mentions other related functions such as: `future-call`, `future-done?`, `future-cancel`, `future-cancelled?` and `future?`

`future` takes one or more expressions as input and evaluates them asynchronously in another thread:

```
(defn timer [seconds] ; ❶
  (future
    (Thread/sleep (* 1000 seconds))
    (println "done" seconds "seconds.")))

(def t1 (timer 10)) ; ❷
```

```
;; #'user/t1
;; done 10 seconds.
```

❶The `timer` function takes a number of seconds as input and creates a `future` that contains a `Thread/sleep` call for the requested time and prints a message on screen. The result of the computation is `nil`.

❷Invoking `timer` doesn't block. The message prints after the requested amount of seconds.

future-done? and future?

`future` and `future-call` returns an object implementing the `java.util.concurrent.Future` interface. Instead of using native Java functions, Clojure offers a few convenient functions to manage the computation inside a `future`. We could for example perform a check to verify if the object is a `future` or if the computation is done:

```
(def t2 (timer 10)) ; ❶

(future? t2) ; ❷
;; true

(future-done? t2) ; ❸
```

```
  ;; false

  ;; wait 10 seconds.

  (future-done? t2) ; ❹
  ;; true
```

❶ `timer` is the same function defined at the beginning of the section. We create a second timer "t2" set to 10 seconds. The timer starts immediately.
❷ `future?` confirms that "t2" is effectively a `future` object.
❸ `future-done?` expects an object with type `java.util.concurrent.Future` which is part of the interface returned by `future`. `future-done?` returns `false` while the timer is still running.
❹When the "done 10 seconds." message prints, the timer is done evaluating the form.

We don't necessarily need to wait for the `future` to finish. We can use deref (or the corresponding reader macro `@`) to access the result of the computation. Beware that if the `future` is not done yet, the call to `deref` is going to block:

```
  (def sum (future (Thread/sleep 10000) (+ 1 1)))

  (realized? sum) ; ❷
```

```
  ;; false

  (deref sum) ; ❸
  ;; 2

  (realized? sum) ; ❹
  ;; true

  (deref sum) ; ❺
```

❶The expression `(+ 1 1)` is evaluated in a future preceded by a 10 seconds pause.

❷If we call `realized?` on the `future` immediately after, it returns `false` as it's not done yet.

❸Calling `deref` on `sum` blocks until the thread wakes up and the expression `(+ 1 1)` is evaluated. At that point it returns the result "2".

❹Calling `realized?` after the `future` has evaluated returns `true`.

❺Note that a second call to `deref` doesn't incur in another 10 seconds pause, as the result is now cached and the `future` never evaluates again.

future-cancel and future-cancelled?

`future-cancel` attempts to cancel a potentially running `future`. The action fails if the future was already canceled or for other reasons, such as the `future` does not respond (for whatever reason):

```
(def an-hour (timer (* 60 60))) ; ❶

(future-cancelled? an-hour) ; ❷
;; false

(future-cancel an-hour) ; ❸
;; true

(future-cancelled? an-hour) ; ❹
;;true

(future-cancel an-hour) ; ❺
;; false
```

❶ `an-hour` is an hour long timer.

❷ Right after creation we ask if the `future` was cancelled but it was not (as expected).

❸ `future-cancel` attempts to stop the running timer successfully.

❹ We can see that the `future` was cancelled.

❺ Any other attempts to `future-cancel` the already cancelled `future` returns `false`.

future-call

`future-call` is the lower level function used by the `future` macro to create a `future`. There are very few reasons to use `future-call` directly. One reason is if you need to use it as higher order function:

```
(mapv future [:f1 :f2]) ; ❶
;; CompilerException java.lang.RuntimeException

(pprint (mapv future-call [(^:once fn* [] :f1) (
;; [#<Future@3ef6c6cd: :f1> #<Future@1d90b8d2: :
```

❶You can't pass macros to functions.
❷mapv uses `future-call` to create a future for each item in the vector. Note that `future-call` requires a function, not an expression.

You might have noticed in the example above a rare use of `fn*` with "once" metadata, which creates a functional closure with local clearing support. Here's why this is a good idea with `future`.

Future's locals clearing

Compared to `future-call`, the `future` macro applies a potentially important optimization to the function that wraps the user expressions:

```
(defmacro future [& body] ; ❶
  `(future-call (^{:once true} fn* [] ~@body)))
```

❶The `future` macro is a wrapper around `future-call`.
❷fn* is given the `:once` metadata that influences how the related class is generated by the compiler.

When a function is created using fn* with `^{:once true}` metadata, the Clojure compiler generates a method that

enforces clearing of all the class attributes. Clearing is implemented by setting arguments to "null" as soon as possible. This prevents the same function object to be callable again:

```
(let [s "yes"
      f1 (^:once fn* [] (str "local-var: " (or s
      f2 (^{:once false} fn* [] (str "local-var:
  [(f1) (f1)
   (f2) (f2)])
;; ["local-var: yes"
;;  "local-var: no"    ; ❶
;;  "local-var: yes"
;;  "local-var: yes"]  ; ❷
```

❶Note that the second time "f1" is invoked, the local value "s" has been cleared. Note that `^{:once true}` and `^:true` are equivalent notations.

❷When the once-only semantic is disabled, the local var is never cleared.

The reason why `future` benefits from once-only semantic is because after running the body expression the `future` object can stay around for arbitrarily long time (even when terminated). The once-only semantic gives the garbage collector an opportunity to remove potentially large locals even when a

`future` is still referenced by a thread pool (a common scenario). If you decide to use `future-call` directly, it is advisable you use the once-only semantic to avoid potential out of memory issues.

One use of `future` is to decouple the current thread from one or more expensive computations. In the next example, several `futures` wrap potentially long http requests and each call requires roughly the same time. By using `future` all requests start in parallel (and results are available when the slowest request completes):

```
(require '[clojure.xml :as xml])

(defn fetch-async [url]          ; ❶
  (future (doall (xml-seq (xml/parse url)))))

(let [guardian (fetch-async "https://git.io/guar
      nytimes (fetch-async "https://git.io/nyt-w
      reuters (fetch-async "https://git.io/reute
      washpost (fetch-async "https://git.io/wash
      wsj (fetch-async "https://git.io/wsj-rss-x
  (count (concat                 ; ❸
           (take 10 @guardian)
           (take 5 (drop 15 @nytimes))
           (take 5 (drop 20 @wsj))
           (take 2 (drop 5 @washpost))
```

```
              (take 10 @reuters))))
  ;; 32
```

❶ `fetch-async` takes an URL and makes an HTTP requests as part of `xml/parse`. The resulting XML is transformed into Clojure data structures which are completely realized.

❷ Each binding in the let block uses `fetch-async` to download a different feed. Instead of blocking for each http request, `future` decouples the call and returns immediately.

❸ At the point of concatenating the articles, some request might be done but others still downloading. The last call to `deref` (`@`) to return results is the longest call. But at that point, all feeds have been downloaded.

While `future` models independent threads effectively, it becomes even more powerful when used with other concurrency functions like promise or delay. We are going to see related examples in the following sections.

[219] For an introduction to locking in concurrency problems, please see en.wikipedia.org/wiki/Lock_(computer_science)

14.2 PROMISE AND DELIVER

A `promise` creates a "gate" around an initially empty memory location. The gate protects the content from threads trying to

access it. The function `deliver` atomically writes a value to the location and opens the gate. Once the value is delivered, all blocking threads are immediately able to see and use the value:

```
(def p (promise)) ; ❶

(future (println "Thread 1 got access to" @p)) ;
(future (println "Thread 2 got access to" @p))
(future (println "Thread 3 got access to" @p))

(deliver p :location) ; ❸
;; #object[clojure.core$promise$reify__7005 0x16
;; Thread 3 got access to Thread 2 got access to
;; got access to  :location:location:location
```

❶ `promise` creates a protected location "p". The location is initially empty and any attempt to deref "p" results in a blocking call (unless `deref` with timeout is used).

❷ We create three separate threads with future. The expression contains a request to access the protected location "p" and blocks immediately. The thread evaluating these forms at the REPL never blocks.

❸ `deliver` performs the following actions atomically: it stores the value `:location` inside the promise "p" and opens the gate to allow access. What we see is the concurrent printing of the threads created earlier.

As you can see from the example above, as soon as a value is delivered to the promised location, all blocking threads gain access to the location at the same time. `promise` creates a callable object (which is used by `deliver`):

```
(def p (promise))
(future (println "Delivered" @p))
(p :value) ; ❶
;; Delivered :value

(realized? p) ; ❷
;; true
```

```
  (p :value) ; ❸
  ;; nil
```

❶A `promise` object is also a function of one argument. We can invoke the function with a value to obtain the same effect as `deliver`.

❷ `realized?` returns the current state of the `promise` which can be either realized or not realized.

❸Any further delivery to the promise produces no action.

`promise` and `future` can be useful for threads coordination. The next example is inspired by the cigarette smokers problem described by Suhas Patil in 1971 [220]. The problem models the following:

- A cigarette requires tobacco, paper and matches to be prepared and smoked.
- 3 smokers sit at a table having either an infinite supply of tobacco, or paper or matches, but missing the other 2 ingredients.
- A person not sitting at the table picks 2 ingredients at random and put them on the table.
- Each round, there should be only one person able to light up a cigarette.

The problem poses interesting challenges such as thread contention and synchronization. We can model the problem using a `promise` for each ingredient and a future for each player:

```
(def msgs (atom [])) ; ❶

(defn smoke [smoker ingr1 ingr2] ; ❷
  (swap! msgs conj (str smoker " attempts"))
  (deref ingr1)
  (deref ingr2)
  (swap! msgs conj (str smoker " successful!")))

(defn pick-two [tobacco paper matches] ; ❸
  (rest
    (shuffle
      [#(deliver tobacco :tobacco)
       #(deliver paper :paper)
       #(deliver matches :matches)])))

(defn run [] ; ❹
  (dotimes [i 5]
    (swap! msgs conj (str "Round " i))
    (let [tobacco (promise) paper (promise) matc
      (future (smoke "tobacco holder" paper matc
      (future (smoke "paper holder" tobacco matc
      (future (smoke "matches holder" tobacco pa
```

```
        (doseq [add (pick-two tobacco paper matche
        (Thread/sleep 10)))
    @msgs)
```

❶ Printing to standard output would produce unreadable messages because of the multiple threads trying to write to the same stream. A possible option is to serialize messages into a vector inside the `msgs` atom.

❷ `smoke` takes the name of a smoker and the two missing ingredients they are waiting for. It then deref both ingredients in the attempt to complete and light a cigarette.

❸ `pick-two` selects two ingredients at random. It uses shuffle to randomize access to the ingredients. Note that we need to wrap the `deliver` request in a function to avoid immediate execution.

❹ `run` orchestrates the simulation. Each `promise` represents the location for an ingredient and future wraps the attempt at smoking a cigarette for each smoker. doseq evaluates delivery of only two ingredients while three smokers compete for them. `run` returns the messages collected so far.

There is also the possibility for a deadlock if, for example, the smoker with tobacco removes the paper while another removes the matches from the table. The simulation avoids this kind of

deadlock because the `promise` caches its value internally. Unfortunately, there is another problem we need to fix. Let's check the output from the simulation:

```
(pprint (partition 5 (run))) ; ❶

;; ("Round 0" ; ❷
;;  "tobacco holder attempts"
;;  "paper holder attempts"
;;  "matches holder attempts"
;;  "matches holder successful!")
;; ("Round 1"
;;  "tobacco holder attempts"
;;  "paper holder attempts"
;;  "tobacco holder successful!"
;;  "matches holder attempts")
;; ("Round 2"
;;  "tobacco holder attempts"
;;  "paper holder attempts"
;;  "matches holder attempts"
;;  "tobacco holder successful!")
;; ("Round 3"
;;  "tobacco holder attempts"
;;  "paper holder attempts"
;;  "matches holder attempts"
;;  "matches holder successful!")
;; ("Round 4"
;;  "tobacco holder attempts"
```

```
  ;;  "paper holder attempts"
  ;;  "paper holder successful!"
  ;;  "matches holder attempts"))
```

❶ `run` returns the collection of messages retrieved during the simulation. We use partition to group messages by iteration.
❷There is only a successful smoker each round, as expected.

Each round of distribution at the table results in a smoker successfully lighting up a cigarette. But what is the destiny of the other two smokers? Only one smoker receives the right ingredients while the other two are waiting for an ingredient that is never delivered. The `future` thread hangs forever waiting to `deref` the `promise`. As designed, the simulation leaks threads each cycle, potentially crashing the Java virtual machine after a few thousands threads. We can stop the leakage by using a timeout argument on the deref call, an important feature offered by Clojure to manage concurrency:

```
(defn smoke [smoker ingr1 ingr2] ; ❶
  (let [i1 (deref ingr1 100 "fail!")
        i2 (deref ingr2 100 "fail!")]
    (swap! msgs conj (str smoker " " i1 " " i2))

(run)
```

```
(print @msgs)

;; ["Round 0" ; ❷
;;  "tobacco holder :paper :matches"
;;  "Round 1"
;;  "matches holder :tobacco :paper"
;;  "Round 2"
;;  "matches holder :tobacco :paper"
;;  "Round 3"
;;  "tobacco holder :paper :matches"
;;  "Round 4"
;;  "tobacco holder :paper :matches"
;;  "matches holder fail! :paper"
;;  "paper holder fail! :matches"
;;  "tobacco holder :paper fail!"
;;  "paper holder :tobacco fail!"
;;  "paper holder :tobacco fail!"
;;  "tobacco holder :paper fail!"
;;  "matches holder fail! :paper"
;;  "paper holder fail! :matches"
;;  "matches holder fail! :paper"
;;  "paper holder fail! :matches"]
```

❶The new version of the `smoke` function adds a timeout of 100 milliseconds to the `deref` call. It also adds a default message to return when reaching the timeout.

❷The messages are now showing failure messages after the timeout expires.

The modified version of `deref` using timouts prevents threads from hanging indefinitely when an ingredient is not delivered to the `promise`. We can't partition the results as before because now there could be any number of attempts by other smokers to light up a cigarette.

[220] The Cigarette and Smokers problem is described in this Wikipedia page en.wikipedia.org/wiki/Cigarette_smokers_problem

14.3 DELAY



Note

This section also mentions other related functions: `delay?` and `force`.

`delay` is a macro that takes a Clojure form as argument. `delay` guarantees that the form is going to be evaluated only once, the first time deref is called on it:

```
(def d (delay (println "evaluated"))) ; ❶

(deref d) ; ❷
;; evaluated
;; nil

@d ; ❸
;; nil
```

❶ `delay` returns an object of type `clojure.lang.Delay` that we save in a var called "d". Nothing prints on screen as `delay` stores the expression without evaluating it.

❷ As soon as we call deref on "d" the expression is evaluated.

❸ We can also use the reader macro "@" instead of `deref`. The result of evaluating the expression is cached internally and returned. We can see there is no printout the second time we call `deref`.

An evaluation wrapped with `delay` has the following properties:

- It decouples a definition from the actual evaluation of that definition. Most of the definitions in Clojure happens at compile time hence `delay` can be used to defer evaluation at run time.

- In the presence of multiple threads, `delay` guarantees that the form is going to be evaluated only once.
- The evaluation results are cached and returned on following calls to `deref`.

> Note
>
> Delayed evaluation, especially of stateful resources like connections, thread pools or local files, becomes necessary if the application is AOT compiled (please see compile for an overview of AOT compilation). Without `delay`, an application could try to connect to a database while compiling it.

It's worth noting that using atom would not replace the need for `delay`. In the following example using an atom to store state causes multiple initializations. To see the output from the example, you'll need to have a service listening on port "61817". You can achieve this using the "netcat" command line utility with `nc -l 61817`:

```
(import '[java.net InetAddress Socket])
(def connection (atom nil)) ; ❶
```

```
(defn connect [] ; ❷
  (swap! connection
    (fn [conn]
      (or conn
        (let [socket (Socket. (InetAddress/getBy
          (print "Socket connected to 61817\n")
          socket)))))

(defn handle-request [s] ; ❸
  (let [conn (connect)]
    (print (format "Doing something with %s\n" s

(dotimes [i 3] ; ❹
  (future (handle-request i)))
(flush)
;; Socket connected to 61817
;; Socket connected to 61817
;; Doing something with 1
;; Doing something with 0
;; Socket connected to 61817
;; Doing something with 2
```

❶ The example simulates a connection to a service listening on port "61817" using a `java.net.Socket` connection. To avoid the cost of creating a connection each request, the application uses an atom to store an open connection.

❷ `connect` creates a new socket connection. It swap! the atom with the new socket object only if there is not one already available.

❸ `handle-request` executes the main computation. It takes a parameter and requests the connection to communicate with the external service.

❹ When the application starts it starts receiving many simultaneous requests. The effects of the incoming requests is simulated using future. The output shows 3 lines in which the socket appears to have been connected.

When the application starts receiving requests, multiple threads are able to execute `swap!` requests concurrently. atom correctly handles concurrency by allowing a few retries until one thread is able to store the socket connection. In doing so, other connections have been created but immediately abandoned, wasting resources.

By using `delay` instead of atom, we can achieve the desired effect of initializing the connection once:

```
(def connection
  (delay ; ❶
    (let [socket (Socket. (InetAddress/getByName
      (print "Socket connected to 61817\n")
      socket)))

(defn handle-request [s] ; ❷
  (let [conn @connection]
    (print (format "Doing something with %s\n" s

(dotimes [i 3] ; ❸
  (future (handle-request i)))
(flush)
;; Socket connected to 61817
;; Doing something with 2
;; Doing something with 1
;; Doing something with 0
```

❶Instead of defining an atom, the connection is now declared as a `delay`. The logic required to compare the current content of the `atom` was removed.

❷ `handle-request` now uses @ to dereference the `delay`.

❸We start the same number of threads as before and now we can see a single printout confirming that the socket object has been created just once. Following that, we can see the 3 outputs generated by `handle-request`.

delay? and force

`delay?` and `force` are utility functions to help manage `delay` objects. `delay?` asks if the given argument is a `delay`:

```
(def d (delay (println :evaluated)))

(if (delay? d) ; ❶
  :delay
  :normal)
;; :delay
```

❶ `delay?` is a predicated function returning `true` when the given argument is a `delay` object. Note that `delay?` does not force any evaluation.

`force` is useful to deal with objects that could potentially be a `delay` but we are not sure. If the argument is a `delay`, it `deref` its content, else it returns the object itself:

```
(def coll [(delay (println :evaluated) :item0) :
(map force coll) ; ❷
;; :evaluated
;; (:item0 :item1 :item2)
```

❶The collection "coll" contains a mix of delayed and normal values.

❷Instead of checking each argument with `delay?` we can use `force` to handle delayed values as well as normal values.

Note that if a delayed computation produces an exception, the same exception object is re-thrown at each `deref`:

```
(def d (delay (throw (ex-info "error" {:cause (r

(try @d (catch Exception e (ex-data e))) ; ❷
;; {:cause 0.14105452022720477}

(try @d (catch Exception e (ex-data e)))
;; {:cause 0.14105452022720477}
```

❶ This `delay` definition produces an error on purpose. The error created with ex-info contains a random number to verify if the body that creates the exception is evaluated multiple times or not.

❷ As we can see by calling `deref/@` twice on the same `delay` object, the exception is the same.

14.4 REF

Note

This section also mentions other related functions such as: `sync`, `dosync`, `alter`, `commute`, `ensure`, `ref-set`, `ref-history-count`, `ref-min-history`, `ref-max-history` and `io!`.

`ref`, `dosync` and the other functions in this section are the main entry point into Clojure implementation of Software Transactional Memory (or STM for short). The STM is a concurrency control mechanism analogous to database transactions designed to protect memory areas (instead of disks) [221]. Specifically, `ref` is one of the concurrency primitives along with atoms, agents and vars. What differentiates `ref` from the other concurrency primitives is that multiple `ref` can coordinate inside the same transaction.

The canonical example of reference coordination in a transaction is modeling the transfer of a sum from a bank account to another:

```
(def account-1 (ref 1000)) ; ❶
(def account-2 (ref 500))

(defn transfer [amount]
  (dosync ; ❷
    (when (pos? (- @account-1 amount))
      (alter account-1 - amount) ; ❸
      (alter account-2 + amount))
    {:account-1 @account-1 :account-2 @account-2}

(transfer 300) ; ❹
;; {:account-1 700, :account-2 800}
```

❶ `ref` creates an object of type `clojure.lang.Ref`. The `ref` accepts an initial value of any type (including `nil`).
❷ `dosync` initializes a transaction context that monitors access to the reference objects inside the body of the expression.
❸ Operations like `alter` notifies the STM about the intention to perform a change to the reference. Such changes can happen immediately, at the end of the `dosync` block, repeat multiple times or not happening at all.
❹ In this simple example, we can verify that a sum of "300" was withdrawn from "account-1" and moved to "account-2".

An apparently innocuous account transfer operation (such as the one above), leads to many challenges in concurrent applications: after checking that there is enough money, how do we make sure that another thread does not empty the first account before we are able to transfer money to the second? A traditional solution to this problem is to use explicit locking and delegate programmers the responsibility to deal with concurrency. Clojure however takes a lock-free approach to thread coordination with the STM [222].

Note

`dosync` is a wrapper around `sync` that does not pass any option. `sync` was designed to accept options but as of now, there is no option available.

ref-history-count, ref-min-history, ref-max-history

`ref` instances contain a queue-like storage space that can be used during transactions. By default, `ref` doesn't store committed values as `ref-min-history` is 0, which means no space to store values:

```
(def r (ref "start value"))

(ref-min-history r) ; ❶
;; 0

(ref-max-history r) ; ❷
;; 10
```

❶A `ref` created using just the initial value has a default `ref-min-history` of 0.

❷The default for `ref-max-history` is 10.

The effect of this default can be seen in the following example. A slow reader "T1" produces a read fault when a faster writer "T2" is able to commit the `ref` before "T1" can access it:

```
(let [r (ref 0)   ; ❶
      T1 (future  ; ❷
           (dosync
             (println "T1 starts")
             (Thread/sleep 1000)
             (println "T1 history-count"
               (ref-history-count r))
             @r))
      T2 (future
           (dosync
             (println "T2 starts")
             (alter r inc)))] ; ❸
  [@T1 @T2 @r])

;; T1 starts
;; T2 starts
;; T1 history-count 0 ; ❹
;; T1 starts          ; ❺
;; T1 history-count 0
;; [1 1 1]
```

❶A `ref` "r" is initialized to 0.

❷The first transaction "T1" sleeps for 1 second after printing a start message and before printing the history-count. When "T1" starts the in-transaction value of "r" is 0.

❸The second transaction "T2" immediately commits a change to "r", setting the value to 1. "T1" is still sleeping.

❹Note that when we ask the current `ref-history-count`, the `ref` returns 0.

❺We can see that "T1" prints the starting message again, which means the transaction was restarted.

Even if the example appears trivial on the surface, the STM has to solve quite a few challenges to produce the correct result. After sleeping for 1 second, "T1" requests read access to the `ref`, but "r" has already been committed successfully by "T2". When "T1" accesses "r" it retrieves a stale value "0" and the STM restarts the transaction. We can see the printout "T1 starts" twice in the logs. At the end, "T1" correctly prints "1" as the result of reading "r", because by the time the thread completed, that was the value available.

The STM default is the most correct and conservative assumption: a thread cannot access a stale value and the transaction restarts to pick up the changes. However, if the new value of the `ref` does not depend on the old, there is no need

to restart the transaction and a stale read is tolerated. In this case, each transaction could see a different value for the `ref`. We can enable this behavior by setting `min-history` to a positive number which represents how many in-transaction values the `ref` tolerates before producing a read fault. We can do this using the `ref-min-history` function, or when creating the `ref`:

```
(let [r (ref 0 :min-history 1)  ; ❶
      T1 (future
           (dosync
             (println "T1 starts")
             (Thread/sleep 1000)
             (println "T1 history-count"
               (ref-history-count r))
             @r))
      T2 (future
           (dosync
             (println "T2 starts")
             (alter r inc)))] ; ❷
  [@T1 @T2 @r])

;; T1 starts
;; T2 starts
;; T1 history count 1
;; [0 1 1] ; ❸
```

❶ The new definition of the reference "r" allows for one in-transaction value of the `ref`. This means that at least one thread is allowed to read a stale value of the `ref` in case of concurrent access.

❷ The rest of the computation remains the same as the previous example, except that the transaction "T1" does not restart.

❸ Note that "T1" now reads 0 even if the `ref` was changed to 1 by another thread.

If `:min-history` allows more stale reads without a restart, `:max-history` determines the maximum number of values the reference can store before a read fault occurs. This is because the STM can automatically detect situations where a transaction restarts too many times and bumps the history count accordingly, but up to the number in `:max-history` (which defaults to 10). In general, the slower the initial transaction (or the higher the number of concurrent threads), the higher `:max-history` needs to be to prevent a restart on a faulty read [223].

alter and ref-set

`alter` and `ref-set` mutate a reference inside a transaction. They work in a similar way, but `alter` takes a function of the old value to the new:

```
(def r (ref 0))

(dosync (alter r inc)) ; ❶
;; 1

(dosync (ref-set r 2)) ; ❷
;; 2
```

❶ `alter` takes the reference and a function of one argument. The function receives the old value of the `ref` that can be used to compute a new value.
❷ `ref-set` ignores any old value in the `ref` and just replaces it with a new one.

Concurrent modifications of the same `ref` produce a restart of the transaction to guarantee consistency. The following example shows a function `perform` that first increments and then add two numbers in a short loop. Operands and results are mutable and shared, so they are implemented as `ref` objects. The `dosync` transaction boundary marks an area of atomic computation that either completes as a whole or not:

```
(def op1 (ref 0)) ; ❶
(def op2 (ref 1))
(def result (ref []))
```

```
(defn perform [] ; ❷
  (dosync
    (dotimes [i 3] ; ❸
      (println (format "###-%s-###\n" ; ❹
        (hash (Thread/currentThread))))
      (alter op1 inc)
      (alter op2 inc)
      (alter result conj (+ @op1 @op2))
      (print
        (format "%s + %s = %s (i=%s)\n"
          @op1 @op2 (+ @op1 @op2) i))
      (Thread/sleep 300))
    @result))

(perform) ; ❺
;; ###-2023564354-###
;; 1 + 2 = 3 (i=0)
;; ###-2023564354-###
;; 2 + 3 = 5 (i=1)
;; ###-2023564354-###
;; 3 + 4 = 7 (i=2)
;; [3 5 7]

(perform)
;; ###-2023564354-###
;; 4 + 5 = 9 (i=0)
;; ###-2023564354-###
```

```
;; 5 + 6 = 11 (i=1)
;; ###-2023564354-###
;; 6 + 7 = 13 (i=2)
;; [3 5 7 9 11 13]
```

❶Three `ref` objects are given an initial value of 0, 1 and empty vector respectively.

❷ `perform` executes a few calculations inside a `dosync` block. The results are stored in the `ref` and returned.

❸Inside each dotimes loop, `perform` increments the operands and prints their sum on screen. The loop executes inside a transaction, enforcing the constraint that once `perform` is called, all changes either happens as a whole or they don't.

❹At the beginning of each loop, we also print a message that contains the thread identification (as a hash of the thread object). This is useful to understand how threads are competing to control the execution of the code.

❺We call `perform` without concurrency at first, to show what results to expect. We can see `[3 5 7]` as the first result and `[3 5 7 9 11 13]` if we call `perform` again without resetting `op1` and `op2`.

In the previous example, `perform` is invoked twice without concurrency just to show how the loop behaves. If we run multiple `perform` in separate threads, the STM guarantees the

same result seen in the sequential case at the price of potentially having to restart transactions:

```
(dosync ; ❶
  (ref-set op1 0)
  (ref-set op2 1)
  (ref-set result []))

(let [p1 (future (perform)) ; ❷
      p2 (future (perform))]
  [@p1 @p2]
  @result)

;; ###-1235449187-###  ; ❸
;; 1 + 2 = 3 (i=0)
;; ###-326623499-###   ; ❹
;; ###-326623499-###
;; ###-326623499-###
;; ###-1235449187-###
;; 2 + 3 = 5 (i=1)
;; ###-326623499-###
;; ###-326623499-###
;; ###-326623499-###
;; ###-1235449187-###
;; 3 + 4 = 7 (i=2)
;; ###-326623499-###
;; ###-326623499-###
;; ###-326623499-###
```

```
;; ###-326623499-###   ; ❺
;; 4 + 5 = 9 (i=0)
;; ###-326623499-###
;; 5 + 6 = 11 (i=1)
;; ###-326623499-###
;; 6 + 7 = 13 (i=2)
;; [3 5 7 9 11 13]
```

❶Before starting each experiment, it's good practice to reset the `ref` objects to some initial known state.

❷ `perform` is now invoked from two separate threads and then we wait for results to come back by `deref` them in a vector. The results are available by calling `@ref` at the end of the let block.

❸One of the threads, either p1 or p2, enters the transaction first.

❹We can see repeated attempts of the second thread id "326623499" trying to access. What we see is the result of the STM restarting the body of the `dosync` instruction. Considering our sleep period of 300 milliseconds, we can infer that the STM applies a 100 milliseconds wait period between transactions retries (which is indeed the case and is not user configurable).

❺Eventually, the second thread is able to enter the loop. This happens when the first one completes the transaction.

To understand how the output from the example above interleaves, we need to consider that one thread always enters the transaction first. As soon as that happens, the late thread is forced a few restarts. A single restart is not enough time for the first thread to complete the transaction, so we can see a few restarts happening. There is a hard limit of 10,000 retries before the STM gives up and throws exception (in our example, we are pretty far from hitting that limit).

commute

`commute` is a relaxed form of `alter` that signals the STM that writes operation using this function can execute in any order (assuming the write operations are commutative). When `commute` is used instead of `alter`, transactions don't need to restart waiting on each other results because the computation does not dependent on read order.

`commute` is the wrong choice for non commutative operations, such as the mix of increment and addition seen in the previous example. It is however a good candidate in other scenarios where the order of updates is not important. In the following simulation, a polling system receives votes for candidates and prints the name of the first candidate who reaches 100 votes. We are not interested in maintaining an ordered list of the

preferences as they are received, so `commute` seems a natural choice:

```
(def votes (ref {})) ; ❶

(defn counter [poll votes] ; ❷
  (future
    (dosync
      (doseq [pref poll]
        (commute votes update pref (fnil inc 0))

(defn generate-poll [& preference] ; ❸
  (eduction
    (map-indexed #(repeat %2 (str "candidate-" %
    cat
    preference))

(let [c1 (counter (generate-poll 40 64 19 82 11)
      c2 (counter (generate-poll 10 89 23 75 22)
  [@c1 @c2]
  @votes)

;; {"candidate-0" 50 ; ❺
;;  "candidate-1" 153
;;  "candidate-2" 42
;;  "candidate-3" 157
;;  "candidate-4" 33}
```

❶ `votes` is a Clojure map wrapped by a `ref`. Votes are collected by the `ref` and shared across the system.

❷ `counter` is a function returning a future. The body of the `future` takes the incoming batch of votes and increments a counter inside the map corresponding to the votes just received. If we used `alter`, it would force a restart to maintain read consistency, but we are not interested to know what number gets incremented first, just their totals. The STM can optimize for this scenario with `commute`.

❸ `generate-poll` simulates users casting votes for candidates. It takes any number of votes, assuming the position of each number in the arguments is related to a specific candidate starting at index 0, then index 1 and so on.

❹ The incoming batches of votes are assigned to different `counter` threads, so the counting can operate in parallel. The creation of the future object also starts the computation. The vector `[@c1 @c2]` makes sure all futures have finished before reading the results.

❺ We can see the expected count of results.

Note

even when using `ref` with isolated transactions, consistency is possible only by using appropriate data structures. By using a collection that is not concurrent nor immutable, the STM alone wouldn't be able to enforce consistency.

ensure

Along with faulty reads, snapshot isolation can also generate "writes skew". A write skew could happen for example when there are constrains shared by multiple `refs`. To illustrate the problem, let's add a constraint to the voting system that stops the competition as soon as there are more than five "honeypots" votes. A "honeypot" on a web page consists of adding a hidden input field on a form. Humans don't see the input but bots do (and fill it). The new design of the voting system allows to stop the competition as soon as some amount of suspicious submissions are detected. Each thread must now ensure (not coincidentally using `ensure`) that the shared honeypot count constraint does not change outside the current transaction:

```
(def votes ; ❶
  {"honeypot" (ref 0)
```

```
    "candidate-0" (ref 0)
    "candidate-1" (ref 0)
    "candidate-2" (ref 0)
    "candidate-3" (ref 0)
    "candidate-4" (ref 0)})

(defn batch [prefs]
  (future
    (dosync
      (ensure (votes "honeypot")) ; ❷
      (doseq [color prefs
              :while (< @(votes "honeypot") 5)]
        (update votes color commute inc)))))

(defn generate-poll [honeypot & preference] ; ❸
  (concat
    (repeat honeypot "honeypot")
    (eduction
      (map-indexed #(repeat %2 (str "candidate-"
      cat
      preference)))

(let [c1 (batch (generate-poll 3 10 30 20 30 20)
      c2 (batch (generate-poll 5 20 10 10 30 20)
  [@c1 @c2]
  {:total-votes (reduce + (map deref (vals votes
   :winner (ffirst (sort-by (comp deref second) >
   :fraud? (= @(votes "honeypot") 5)})
```

```
;; {:total-votes 115, :winner "candidate-1", :fr
```

❶ All candidates are now modeled as reference objects inside the `votes` map. This enables to `ensure` the honeypot key as well as `commute` votes independently.

❷ The `dosync` body now contains a call to `ensure` on the reference containing the honeypot count. `doseq` has also been updated to check the honeypot count before proceeding any further.

❸ `generate-poll` accepts a number of honeypot entries to generate along proper candidates for the simulation.

❹ A test run of the new voting system confirms a possible fraud. A winner is still calculated but additional batch of votes would not alter the current result.

io!

An important aspect to consider when using the STM is to make sure that expressions in a transaction are side-effects free (more properly, the expression doesn't rely on side effects to succeed). As the transaction could call into any other function, there needs to be a way for arbitrary code to signal unsuitability for transactions. We can signal this fact using `io!`. For example, function `f1` opens a transaction context

that involves calling function `f2` (possibly many more layers below). `f2` is side effecting but designed for the general case. We can use `io!` to signal the fact that should `f2` ever be part of a transaction, the transaction should throw an exception:

```
(def counter (ref 0))

(defn f2 [value] ; ❶
  (io! (println "Sorry, side effect on" value))
  (inc value))

(defn f1 [] ; ❷
  (dosync (f2 (commute counter f2))))

(f1) ; ❸
;; IllegalStateException I/O in transaction  use
```

❶ `f2` is a function that processes a value and makes use of side effects. While `f2` is designed for the general case, using it in a transaction could create problems. Knowing about this possibility, the author of the function wrapped the side effect in `io!`.

❷ `f1` performs operations inside a transaction and explicitly use `f2` (but the use of `f2` might now be so easy to see).

❸ An attempt to run `f1` reveals the presence of `io!` down the calling chain.

To finish the section, it's worth mentioning that you can pass metadata to a `ref` object during construction using the `:meta` option:

```
(def r (ref 0 :meta {:create-at :now})) ; ❶

(meta r) ; ❷
;; {:create-at :now}
```

❶ Functions like with-meta doesn't work with reference types. But `ref` offers the `:meta` option during construction to specify metadata.

❷ We can see the metadata are correctly set.

[221] To understand more about the STM, please consult the following link: en.wikipedia.org/wiki/Software_transactional_memory

[222] The STM does use some degree of locking internally. The claim that the STM is "lock-free" is from the user perspective, not the internal implementation

[223] The STM has a fairly sophisticated behavior. One of the best document that explains the STM in detail, is java.ociweb.com/mark/stm/article.html

14.5 ATOM



Note

This section also mentions other related functions such as: `swap!`, `reset!` and `compare-and-set!`

A Clojure `atom` is a concurrency-aware construct for controlling synchronous and uncoordinated state changes. Let's expand this definition:

- Concurrency-aware: multiple threads can share the `atom` state without explicit locking. Changes to the `atom` are guaranteed to be a function of the old value as read at the beginning of an update. If another thread is able to change the state of the atom before the current can apply its changes, the operation repeats.
- Synchronous: the update operation happens on the same thread that starts the update. This is different from agents for example, but similar to refs.
- Uncoordinated: there is no mechanism to coordinate different `atom` instances (no dosync construct to enclose multiple `atom` operations).

`atom` was introduced to implement the frequent use case of protecting a single reference from concurrent access. The concept of "transaction" still exists for `atom`, but it's implicit and delimited by the update function itself:

```
(def a (atom 0)) ; ❶
(swap! a inc) ; ❷
;; 1
@a ; ❸
;; 1
```

❶ An `atom` is created through the `atom` function passing an initial value.

❷ `swap!` accepts an `atom` instance and a function of the current value. It returns the new updated value (not the `atom` instance). The function is inc in this case, but it could be any function of a single argument.

❸ We can read the value from the `atom` at any time using the same pattern used by other Clojure reference types, either using deref or the equivalent reader macro `@`.

swap!

`swap!` is the most flexible choice to work with `atom`. It takes any additional number of arguments to pass to the update function, making it an ideal choice for updating collections:

```
(def m (atom {:a 1 :b {:c 2}})) ; ❶

(swap! m
  (fn [m]
    (update-in m [:b :c]
      (fn [x] (inc x))))) ; ❷
;; {:a 1 :b {:c 3}}
```

```
  (swap! m update-in [:b :c] inc) ; ❸
  ;; {:a 1 :b {:c 4}}
```

❶The atom "m" contains a deeply nested Clojure map.

❷This form uses explicit functions to show the nesting of update-in and inc to change the number 2 to 3 inside the map.

❸Functions like `swap!`, update-in and inc all operate on the same semantic: they take a function of the old value to the new and allow for any number of additional arguments. The previous form can be written removing the anonymous functions in a more concise way.



Note

If the update function passed to `swap!` contains side effects, please be aware that `swap!` might execute the function any number of times, especially in highly concurrent scenarios.

---

reset!

In case there was no interest in changing the `atom` depending on the previous value, we could use `reset!` instead of `swap!`. Differently from `swap!`, `reset!` always evaluates without

retries. If many threads are attempting a `reset!` operation, the last to get access to the `atom` wins:

```
(def configuration (atom {})) ; ❶

(defn initialize [] ; ❷
  (reset! configuration (System/getenv)))

(initialize) ; ❸
(take 3 (keys @configuration))
;; ("JAVA_MAIN_CLASS_65503" "IRBRC" "PATH")
```

❶ `configuration` is designed to contain the view of the environment variables available after initialization. Other parts of the program can alter the configuration at any time but an additional initialization should reset the `atom` to the system environment.

❷ `initialize` can remove any previous state of the configuration `atom`. `reset!` is more appropriate than `swap!` in this case, as `reset!` does not attempt and compare logic.

❸ After calling `initialize` we can see the configuration contains the current view of the system environment.

Note

An `atom` object does not need to be necessarily a top level var. A typical use of `atom` is for example to "close-over" state in memoization. Please see memoize performance section to see `atom` in action to cache repeating computations.

---

compare-and-set!

`atom` returns an instance of `clojure.lang.Atom`, a wrapper object around `java.util.concurrent.atomic.AtomicReference`. The main goal of Java's `AtomicReference` is to give access to the low level CAS (Compare And Swap) implementations offered by modern hardware architectures. A CAS operation fails if another thread changes the current value before the current has a chance to do the same. So a typical usage pattern involving CAS, is to put the change logic in an infinite loop until the operation succeeds. `swap!` performs such loop for us, but in case we wanted additional control of the loop (for example to stop after 3 attempts), we could implement something like the following `swap-or-bail!`:

```
(defn swap-or-bail! [a f & [attempts]] ; ❶
  (loop [i (or attempts 3)]
    (if (zero? i)
      (println "Could not update. Bailing out.")
      (let [old (deref a)
            success? (compare-and-set! a old (f
        (when-not success?
          (println "Update failed. Retry" i)
          (recur (dec i)))))))

(defn slow-inc [x] ; ❸
  (Thread/sleep 5000)
  (inc x))

(def a (atom 0))
(def f (future (swap-or-bail! a slow-inc))) ; ❹
(reset! a 1) ; ❺
;; "Update failed. Retry 3"
(reset! a 2)
;; "Update failed. Retry 2"
(reset! a 3)
;; "Update failed. Retry 1"
;; Could not update. Bailing out.
```

❶ The interface of `swap-or-bail!` is similar to `swap!` except for taking an additional "attempts" argument with a default of 3 (if none is given).

❷ `(f old)` triggers a slow update function on the old value. If, by the time `(f old)` returns, there is a change to the value in the `atom` (situation that we are going to create in this example), then `compare-and-set!` fails and the operation retries.

❸ `slow-inc` is an intentionally slow version of inc. We have 5 seconds to evaluate a `reset!` call on the atom to force a CAS retry.

❹ `swap-or-bail!` runs in its own thread, so we are free to change the content of the atom while its running. To run in a separate thread we use a future.

❺ When each `reset!` operation happens before 5 seconds from the last `compare-and-set!` attempt, we force another attempt. But after reaching the max number of attempts, `swap-or-bail!` prints a failure message and exits.

You should avoid calling `compare-and-set!` with a value that is not coming from the same `atom` instance you're trying to update. The risk is to incur in surprises determined by Java equality semantic used by `compare-and-set!`. For example:

```
(def a (atom 127))

(compare-and-set! a 127 128) ; ❶
;; true

(compare-and-set! a 127 128) ; ❷
;; false

(compare-and-set! a 128 129) ; ❸
;; false
```

❶ `compare-and-set!` requires 3 arguments: the `atom` instance, a comparison value and the desired new value. The desired value becomes the new value of the `atom` only if the comparison value is the same as the current value of the `atom`.

❷ Since the `atom` was mutated to contain 128, the second `compare-and-set!` operation fails, as the comparison value does not match the current value (127 != 128).

❸ Strangely enough, even passing the right value (128), `compare-and-set!` refuses to update.

The reason why `compare-and-set!` refuses to update the atom even when the old value is apparently the same as provided, is because of Java reference equality (Java operator

`==`) and Clojure autoboxing. Long values from -127 to 127 are cached so `new Long(127) == new Long(127)` is `true` because the two numbers are effectively the same instance. But `new Long(128) == new Long(128)` is `false` in Java because the two objects are effectively different instances (as there is no implicit caching). Clojure wraps numerical arguments into a new `java.lang.Long` instance, resulting in the observed `compare-and-set!` behavior.

14.6 AGENT

Note

This section also mentions other related functions: `send`, `send-off`, `send-via`, `set-agent-send-executor!`, `set-agent-send-off-executor!`, `restart-agent`, `shutdown-agents`, `release-pending-sends`, `await`, `await-for`, `agent-error`, `set-error-handler!`, `error-handler`, `error-mode`, `set-error-mode!`

`agent` is a concurrency-aware construct that accepts asynchronous, uncoordinated and sequential state changes. Let's expand this dense definition to understand more:

- Asynchronous: `agent` performs actions in a separate thread than the caller. In this respect, an `agent` is analogous to a future (they also share the same thread pool). Differently from `future`, `agent` has an internal state.
- Uncoordinated: like atom, `agent` can't coordinate with another `agent` (preventing state changes in another `agent` based on conditions on the current), but they can participate in a STM transaction by holding actions until commit.
- Sequential: actions sent to an `agent` executes in the order they are received. This makes an `agent` a good candidate to handle side effecting operations. The `agent` works on queued actions unless it encounters an error condition. Pending work can resume or be removed after an error. The `agent` can also create tasks for itself or other agents to execute immediately or after a state change.

Note: `agent` brings some resemblance with Erlang's actors [224]. There are however a few fundamental differences: `agent` is not distributed, they accept any function (not just a predefined set of messages) and you can access an `agent` anytime without sending a message to it.

send

`send` delivers the `agent` a function from the current value to the new:

```
(def a (agent 0)) ; ❶

(send a inc) ; ❷
(deref a) ; ❸
;; 1

@(send a inc) ; ❹
;; 1

(deref a) ; ❺
;; 2
```

❶ `agent` creates a new `clojure.lang.Agent` instance with an initial value.
❷ `send` delivers the `agent` the inc function.
❸ We can check the current value of the `agent` with deref or the reader macro `@`.
❹ `send` returns the `agent` instance itself, which we might be tempted to `deref` straight away. However, the action runs asynchronously and the result might not be updated yet.
❺ inc is a simple operation, so we can immediately see the results at the REPL.

await

As seen in the previous example, `agent` are asynchronous and we might need to wait to see the results of sending an action to them. If we want to wait for the action to perform, we can use `await`:

```
(def a (agent 10000)) ; ❶
(def b (agent 10000))

(defn slow-update [x] ; ❷
  (Thread/sleep x)
  (inc x))

(send a slow-update) ; ❸
```

```
(send b slow-update)

(time (await a b)) ; ❹
;; "Elapsed time: 7664.066924 msecs"
;; nil
```

❶Two `agent` contain a number of milliseconds to wait.
❷ `slow-update` waits the number of milliseconds given as input before incrementing the number.
❸The `agent` are sent the slowly updating actions. As expected, the main thread does not need to block.
❹However, if we want to explicitly wait for all the actions sent to the agents to finish, we can use `await`. Please note: the elapsed time shown here depends on how fast the commands are types at the REPL. The faster the evaluation, the closer the elapsed is to 10 seconds.

await-for

Both `await` and `await-for` accept any number of `agent` instances. `await-for` also accepts a timeout in case the `agent` takes too long to return:

```
(send a slow-update) ; ❶
(time (await-for 2000 a)) ; ❷
```

```
  ;; "Elapsed time: 2003.144351 msecs"
  ;; false
```

❶We send another slow updating action to one of the `agent`.
❷This time, instead of returning after 7/8 seconds, the call to `wait-for` returns in approximately the number of milliseconds given as the timeout. Note that `wait-for` also returns `false` in case it returned because of the timeout instead of the `agent` finishing all relevant actions.

send-off

In the previous example, a `slow-update` function was sent to the `agent` keeping a thread busy for potentially long time. To accommodate for different workloads , `agent` has two default options available: `send` executes on a fixed size thread pool (containing the number of cores + 2 threads) while `send-off` uses an unbounded thread pool. Using one or the other is a function of the particular problem at hand. Longer input/output operations would normally benefit from the unbounded pool with `send-off`, while shorter CPU intensive operations are better suited with `send`. However, unbounded thread pools could result in lagging applications or out of memory problems if left unattended.

Controlling thread pools

send-via

If none of the preconfigured thread pools is suitable for a particular problem, `send-via` allows to use a custom thread pool. A good option for shorter, CPU-bound tasks, is the ForkJoin pool introduced with Java 8 [225]:

```
(import '[java.util.concurrent Executors])

(def fj-pool (Executors/newWorkStealingPool)) ; (

(defn send-fj [^clojure.lang.Agent a f & args]
  (apply send-via fj-pool a f args)) ; ❷

(def a (agent 1)) ; ❸
(send-fj a inc)
(await a)
@a
```

❶ The class `java.util.concurrent.Executors` creates a new `WorkStealingPool` that we can save for later use.
❷ `send-fj` offers a similar interface to `send` or `send-off` to send a task to an `agent`, using the newly created pool through `send-via`.
❸ Usage of `send-fj` is the same as `send` or `send-off`.

Clojure also allows changing the thread pool strategy for `send` or `send-off`, for example to control `agent` usage in an application without the need of changing all call sites:

```
(import '[java.util.concurrent Executors])

(def fj-pool (Executors/newWorkStealingPool 100)

(set-agent-send-executor! fj-pool) ; ❷
(set-agent-send-off-executor! fj-pool)
```

❶ `fj-pool` defines a `WorkStealingPool` that will attempt to maintain the number of concurrent worker below or equal to 100 (this does not reflect necessarily on the number of created threads).
❷ From this point onward, `send` and `send-off` are going to use the newly created thread pool.

The `*agent*` dynamic variable

One interesting pattern used with `agent` is a form of internal recursion that consists of the `agent` sending the next action to itself. This can be achieved thanks to the `*agent*` dynamic variable that is set to the current `agent` instance inside the body of the updating function:

```
(def a (agent 10))

(send a #(do (println (identical? *agent* a)) (i
;; true ; ❶
```

❶We can see that `*agent*` and `a` are the same object from within the updating function sent to the `agent`.

Using `*agent*` we could create the following "ping" agent, that executes a request to some URL just to verify if it responds correctly:

```
(def a (agent {})) ; ❶

(defn ping [{:keys [enable url kill] :as m}] ; ❷
  (when (and enable url)
    (try
```

```
        (slurp url)
        (println "alive!")
        (catch Exception e
          (println "dead!" (.getMessage e)))))
    (Thread/sleep 1000)
    (when-not kill (send-off *agent* ping)) ; ❸
    m)

(send-off a ping) ; ❹
(send-off a assoc :url "https://google.com")
(send-off a assoc :enable true) ; ❺
;; alive!
;; alive!
;; alive!

(send-off a assoc :url "http://nowhere.nope") ;
;; dead! nowhere.nope
;; dead! nowhere.nope
;; dead! nowhere.nope
(send-off a assoc :kill true) ; ❼
```

❶An `agent` is initialized with a state map which is initially empty.

❷The `ping` function is sent to the `atom` as the update function. It takes the current state map as input and prints a message regarding the availability of the target when the `:url` is available and the `:enable` condition is `true`. Apart from that, `ping` never change its internal state and always return the current state of the `agent` unaltered as the last line.

❸Note that the `ping` function `send-off` another request to execute itself after waiting 1 second, but only when the `:kill` key is set to `nil` or `false`. As soon as the `:kill` key is set to `true` the recursion stops.

❹The first `send-off` call starts the internal `agent` recursion, but given the conditions are not met, the `agent` doesn't actually ping the requested web page.

❺After all conditions are met, we can finally see messages coming from the inner loop.

❻If we change the target page to a non-existent one, the catch block executes printing a different message.

❼We need to remember to shutdown the recursive agent by setting the `:kill` key to `true`.

release-pending-sends

An `agent` action could result in any of the following (or combination thereof):

- Nothing happens and the state of the `agent` remains unchanged.
- The state of the `agent` changes.
- The `agent` dispatches another action to itself (like the `ping` function seen before).
- The `agent` dispatches one or more actions to other `agents`.

In case of additional dispatches from within the update function, the default behavior of the `agent` is to wait until the state has changed before proceeding sequentially with all created actions. But in case this wait is undesirable (usually because the order in which actions are performed doesn't matter), we can force all pending actions to start immediately with `release-pending-sends`. In the following example, we use the coordination between different agents to find words and letters frequencies in a large text. While the first `agent` is busy calculating the word frequencies, we can release messages for other agents, one for each letter of the alphabet:

```
(require '[clojure.string :refer [split]])
(def alpha (mapv agent (repeat 26 0))) ; ❶
```

```
(def others (agent 0)) ; ❷
(def words (agent {})) ; ❸

(def war-and-peace "https://tinyurl.com/wandpeac
(def book (slurp war-and-peace))

(send-off words ; ❹
  (fn [state]
    (doseq [letter book
            :let [l (Character/toLowerCase lette
                  idx (- (int l) (int \a))]]
      (send (get alpha idx others) inc))
      (release-pending-sends) ; ❺
    (merge-with + state (frequencies (split book

(apply await alpha)
(map deref alpha) ; ❻
;; (202719 34657 61621 118297 313572
;;  54901 51327 167415 172257 2575 20432
;;  96530 61648 184185 190083 45533 2331
;;  148431 162897 226414 64400 27087
;;  59209 4384 46236 2388)
```

❶ `alpha` is a vector containing 26 agents, one for each letter in the alphabet.

❷ We also prepare `others` for any other letter which is not part of the plain alphabet.

❸ The "words" `agent` collects the frequencies of all the words found in the text.

❹ Processing start from the `send-off` instruction. The update function first process the book letter by letter, sending each corresponding `agent` an increase by one request. The second part updates the `agent` state with the word frequencies.

❺ Creating frequencies is a potentially expensive operation so we take advantage of `release-pending-sends` to start processing letter frequencies even if the state of the current `agent` has not been updated yet.

Handling Errors

When a problem happens in a different thread, the source of the problem is often lost unless specific care is taken to raise the problem to the attention of another controlling thread. For example, the following `agent` is given the impossible task of dividing a number by zero. We can check if there was a problem with `agent-error`:

```
(def a (agent 2))
(send-off a #(/ % 0)) ; ❶

(agent-error a) ; ❷
;; #error {
;;  :cause "Divide by zero"
;;  :via
;;  [{:type java.lang.ArithmeticException
;;    :message "Divide by zero"
;;    :at [clojure.lang.Numbers divide "Numbers.
;; ...
;; }
```

❶After generating an `ArithmeticException` on purpose, the `send-off` call does not report about the problem as it is happening on a different thread.

❷The root cause of the problem was saved inside the agent for later inspection and is visible when calling `agent-error`.

restart-agent

After the `agent` enters an error condition, it stops processing more work and any task in the working queue is suspended. To reset the state of the `agent` and resuming any pending work, we can use `restart-agent` (or we can discard any pending work by passing `:clear-actions true`):

```
(restart-agent a 2) ; ❶
(send-off a #(/ % 2)) ; ❷
@a
(restart-agent a 2) ; ❸
;; Execution error at user/eval339 (REPL:1).
;; Agent does not need a restart
```

❶ `restart-agent` removes any error condition and replace the current state with the new one passed as parameter.
❷The `agent` is now ready to accept additional work.
❸Note that `restart-agent` throws exception when called on a healthy `agent` with no error conditions.

Alternatively, we can call `set-error-handler!` to specify a function of two arguments: the `agent` instance and the error. In case of an error, the `agent` will call the error handler instead of entering error state:

```
(def a (agent 2))

(defn handle-error [a e]
  (println "Error was" (.getMessage e))
  (println "The agent has value" @a)
  (restart-agent a 2))

(set-error-handler! a handle-error) ; ❶
```

```
(send-off a #(/ % 0))
;; Error was Divide by zero ; ❷
;; The agent has value 2

@a ; ❸
;; 2
```

❶ `set-error-handler!` allows to specify a custom error handler in case of `agent` errors.

❷By generating an error on purpose, we can see the custom message appearing.

❸The custom handler also resets the `agent` state directly after the error, so processing can resume right away. In a real world scenario, we could implement such a behavior for recoverable exceptions, leaving the `agent` in error state for more serious errors.

If all error conditions are considered recoverable and we can always accept to resume working after an error, we can set the `:continue` error mode on the `agent`, completely ignoring the problem (and not requiring a `restart-agent` call):

```
(def a (agent 2))
(set-error-mode! a :continue) ; ❶
(send-off a #(/ % 0)) ; ❷
```

```
@a ; ❸
;; 2
```

❶ `set-error-mode!` changes the way the `agent` handles errors. If we use `:continue` (as opposed to the default `:fail` mode) the `agent` would simply ignore any error.
❷ This `send-off` operation generates an error. The `agent` does not enter failure state and doesn't throw error.
❸ The state does not change in case of an update generating error.

Finally, if running agents get out of control or if the application doesn't exit properly after executing, chances are that some `agent` thread are still allocated in the thread pool (wasting resources). In that case, `shutdown-agents` performs a graceful shutdown of all `agent` pools and prevents the execution of any other action from that point onward.

[224] Erlang is a popular functional language with a solid industrial history. It popularized the "actor" approach to concurrency from which many other languages took inspiration. Clojure agent brings some similarities to actors but they are also fundamentally different in several ways.
[225] The ForkJoin pool is also used by reducers. Please refer to the related chapter to know more about the fork-join paradigm

and work-stealing.

14.7 DEREF AND REALIZED?

`deref` and `realized?` works with many of the reference types described in this chapter. `deref` (and the related reader macro `@`) has been used extensively throughout the book and it also works for types that are not considered references such as reduced. `deref` also has a variant supporting a timeout (with a default value) for those reference types that could result in a blocking call.

The following table shows all types that support `deref`:

Table 14.1. All reference types supported by `deref`

type	example	reader
`clojure.lang.Agent`	`(deref (agent 0))`	`@(age`
`clojure.lang.Atom`	`(deref (atom 0))`	`@(ato`
`clojure.lang.Delay`	`(deref (delay 0))`	`@(del`
`clojure.lang.Reduced`	`(deref (reduced 0))`	`@(red 0)`
`clojure.lang.Ref`	`(deref (ref 0))`	`@(ref`
`clojure.lang.Var`	`(deref (def a 0))`	`@(def`

`clojure.lang.Volatile`	`(deref (volatile! 0))`	`@(vol 0)`
`java.util.concurrent.Future`	`(deref (future 0))`	`@(fut 0)`
`java.util.concurrent.Future`	`(deref (promise))` (*)	`@(pro`

`deref` works similarly for the types in the table, but there are a few differences:

- When applied to a agent, var, volatile! or atom it returns its current state.
- When applied to a delay it also forces its body to evaluate, unless it was already evaluated: in that case it returns the cached value immediately.

- When applied to ref in a transaction, it returns the in-transaction value for the `ref`. When outside a transaction, it returns the most recently committed value.
- When applied to a future the call might be blocking waiting for the `future` to finish the evaluation of its body.
- When applied to a promise it blocks until a value is delivered.



Warning

An expression like `@(promise)` blocks the current thread. You should always define the `promise` as an independent reference (local binding or var), so you can deliver a value to it, for example: `(let [p (promise)] (deliver p 0) @p)`. Alternatively, use the `deref` variant that supports a timeout.

---

realized?

We can also use `realized?` to check if a value is available from a `promise` or `future`. Differently from `deref`,

`realized?` just returns `true` or `false` without blocking if the value is not available:

```
(def p (promise)) ; ❶

(def f ; ❷
  (future
    (loop []
      (let [v (deref p 100 ::na)]
        (if (= ::na v) (recur) v)))))

(realized? p) ; ❸
;; false
(realized? f)
;; false

(deref f 100 ::not-delivered) ; ❹
;; :user/not-delivered

(deliver p ::finally)
(deref f 100 ::not-delivered) ; ❺
;; :user/finally

(realized? p) ; ❻
;; true
(realized? f)
;; true
```

❶ We create a `promise` referenced by the `var` "p".

❷ We also define a `future` that contains a (potentially infinite) loop. The `loop` verifies if a value has been delivered to the `promise` and if one was delivered, it returns the value. We don't want to block completely, so we use the `deref` variant with a timeout of 100 milliseconds. We use the keyword `::na` ("not available") as a sentinel value for the condition that follows.

❸ At this point, neither "p" nor "f" has been realized.

❹ The `future` is still executing the `loop`. If we try to call `deref` on it without a timout, the main thread will block. We can see that the `promise` has not been delivered a value yet.

❺ After delivering a value to the `promise` we repeat the call to `deref` on the `future`. This time the call returns the delivered value.

❻ `realized?` confirms that both references are now realized.

`realized?` also works with Clojure lazy sequences to verify if the first item in the sequence has been evaluated (and cached for later use):

```
(def s1 (map inc (range 100))) ; ❶

(realized? s1) ; ❷
;; false
```

```
(first s1) ; ❸
;; 1

(realized? s1) ; ❹
;; true
```

❶map (as well as many other common sequential processing functions) produces a lazy sequence.

❷The sequence is initially unrealized, as proved by calling `realized?` on it just after construction.

❸By calling first, we force realization of the first (and potentially others) element in the sequence.

❹ `realized?` now returns `true`.

14.8 SET-VALIDATOR! AND GET-VALIDATOR

"Validation" is a feature designed for vars, atoms, agents and refs to prevent unwanted state changes. A "validator" is a function of the new value that is about to persist as the new reference state. If the new state is not acceptable, the validator function is expected to return `false` or throw an exception:

```
(def a (atom 1)) ; ❶
(set-validator! a pos?) ; ❷
(swap! a dec) ; ❸
;; IllegalStateException Invalid reference state
```

❶"a" is an atom initialized to the value 1.

❷`set-validator!` installs a new validator that verifies the new state is a positive number.

❸A change of state is requested to decrease the current value of the `atom`. Zero is not accepted by the validator function because `(pos? 0)` returns `false`.

If we want to be more specific than the generic `IllegalStateException`, we can throw a more descriptive error:

```
(def a (atom 1))

(defn- should-be-positive [x] ; ❶
  (if (pos? x)
    x
    (throw
      (ex-info (format "%s should be positive" x
        {:valid? (pos? x)
         :value x
```

```
            :error "Should be a positive number"
            :action "State hasn't changed"})))))

  (set-validator! a should-be-positive)

  (swap! a dec) ; ❷
  ;; ExceptionInfo 0 should be positive

  (try (swap! a dec)
    (catch Exception e (ex-data e))) ; ❸
  ;; {:valid? false,
  ;;  :value 0,
  ;;  :error "Should be a positive number",
  ;;  :action "State hasn't changed"}
```

❶ `should-be-positive` is a more descriptive validator function than pos? alone. Instead of returning just `false` it throws a descriptive exception with ex-info.

❷ An invalid operation now explains what exactly went wrong.

❸ If we want even more details, we can catch the exception and extract the descriptive error map with ex-data.

`get-validator` retrieves the validator function, if any was set:

```
(def a 1)
(get-validator #'a) ; ❶
(set-validator! #'a pos?)

(-> (get-validator #'a) ; ❷
    class
    .getSimpleName
    clojure.repl/demunge
    symbol)
;; core/pos?

(def a 0) ; ❸
;; IllegalStateException Invalid reference state
```

❶var objects also accept validation functions. We need to remember to access the var object using the var function or the equivalent reader literal "#'".

❷After setting the pos? function as the validator, we can get back to its name from the function object returned by `get-validator`. There is no method to retrieve the name of a function from its compiled class other than using `clojure.repl/demunge` to format the class name back into a readable form.

❸The `var` "a" is now a global object and attempting a redefinition is now controlled by the existing validator. To remove a validator, use `set-validator!` with `nil`.

Note

`set-validator!` will fail if, at the moment of installing the new validator function, the current state of the reference already violates the validator constrains. For example: given `(def a (atom 0))`, then `(set-validator! a pos?)` throws exception and the validator is not installed.

Validator functions are also accepted at reference creation time (except for vars). We could use validators to prevent account overdraft in the following bank transfer simulation:

```
(def account-1 (ref 1000 :validator pos?)) ; ❶
(def account-2 (ref 500 :validator pos?))

(defn transfer [amount a1 a2] ; ❷
  (dosync
    (alter a1 - amount)
    (alter a2 + amount))
  {:account-1 @a1
   :account-2 @a2})

(transfer 1300 account-1 account-2) ; ❸
;; IllegalStateException Invalid reference state
```

❶Each account is represented by a `ref` object. Each `ref` is installed a validator that prevents the account from going to 0 or below.

❷The `transfer` function moves money from one account to another inside a dosync block.

❸An attempt to transfer more money than what is actually available results in a `IllegalStateException`.

14.9 ADD-WATCH AND REMOVE-WATCH

Watchers, similarly to validators, are called on state changes to signal the fact that a reference state is about to change.

Differently from validators, watchers can't prevent the state transition to happen but they can take other kind of actions. One or more watchers can be set with `add-watch`:

```
(def account-1 (ref 1000 :validator pos?)); ❶
(def account-2 (ref 500 :validator pos?))

(defn- to-monthly-statement [k r old new] ; ❷
  (let [direction (if (< old new) "[OUT]" "[IN]"
    (spit (str "/tmp/statement." k)
      (format "%s: %s$\n" direction (Math/abs (-
      :append true)))

(add-watch account-1 "acc1" to-monthly-statement
(add-watch account-2 "acc2" to-monthly-statement

(transfer 300 account-1 account-2) ; ❹
(transfer 500 account-2 account-1)

(println (slurp "/tmp/statement.acc1")) ; ❺
;; [IN]: 300$
;; [OUT]: 500$

(println (slurp "/tmp/statement.acc2"))
;; [OUT]: 300$
;; [IN]: 500$
```

❶ `account-1` and `account-2` are ref created with a validator function preventing them to go in overdraft.

❷ Additionally, we want the account to sends transfers to a monthly statement file every time there is a money transfer in or out from the account.

❸ The monthly statement tracker is installed as a watcher on each of the reference objects.

❹ `transfer` is the same function that was used in the previous example about validators. The function subtract the sum from the origin account and moves it into the destination account inside a transaction.

❺ We can print the content of the statement files to verify withdraws and deposits from the account.

Watchers execute synchronously and only after the new reference state has been set, isolating correct state handling from any problem happening on the watcher call. Multiple watchers are called in an unspecified order:

```
(def multi-watch (atom 0))

(dotimes [i 10] ; ❶
  (add-watch multi-watch i (fn [k r o n] (print
```

```
  (swap! multi-watch inc) ; ❷
  ;; 0 ,7 ,1 ,4 ,6 ,3 ,2 ,9 ,5 ,8 ,1
```

❶There are 10 watches with keys ranging from 0 to 9 added to an `atom` instance.

❷After calling swap!, each watch prints its own key. The order in which watchers are invoked is not specified (in the same way the keys of a hash-map are returned in no specific order). Note: the last "1" in the list is the return value from the `swap!` call.

If a watcher stops being useful for a specific scenario, it can be removed with `remove-watch`:

```
  (dotimes [i 10] ; ❶
    (remove-watch multi-watch i))

  (swap! multi-watch inc) ; ❷
  ;; 2
```

❶With reference to the previous example, we call `remove-watch` for all 10 keys.

❷Then next call to `swap!` to change the state only prints the new value of the `atom`.

14.10 LOCKING, MONITOR-ENTER AND MONITOR-EXIT

`locking` is the Clojure's equivalent of Java's `synchronize` keyword. `locking` creates a relationship between a critical section of code and a "lock" object (any object will do, as far as it is used to protect the critical section only). At runtime, a thread would have to "acquire" the lock exclusively before executing the critical section. If the lock is already acquired by another thread, the current thread needs to wait until it becomes available again. The following is a version of bank transfer example based on mutable volatiles!. Since we don't use a concurrent-aware reference, we need explicit locking to prevent unpredictable results:

```
(def lock (Object.)) ; ❶
(def acc1 (volatile! 1000)) ; ❷
(def acc2 (volatile! 300))

(defn transfer [sum orig dest]
```

```
  (locking lock ; ❸
    (let [balance (- @orig sum)]
      (when (pos? balance)
        (vreset! orig balance)
        (vreset! dest (+ @dest sum)))))
  [@orig @dest])

(dotimes [_ 1500] ; ❹
  (future (transfer 1 acc1 acc2)))

[@acc1 @acc2] ; ❺
;; [1 1299]
```

❶The "lock" `Object` is only used to synchronize the critical section in the `transfer` function. We could use any new object reference, but creating a new `java.lang.Object` is the most natural choice.

❷A volatile! is a mutable object that does not implement any protection against concurrent access, forcing us to use some form of explicit locking.

❸With `locking`, we prevent another thread from changing the content of the accounts while the current is free to check if there are enough money to move.

❹The example is deliberately creating many more concurrent requests to transfer money than the actual amount available.

❺Our assumption is that, at the end of all the transfer attempts, there should always be "1" in the first account and "1299" in the second.

If you try to remove the `locking` protection from the example above you would see that the second account has been credited much more money than expected, as if they appeared from nowhere. This is the reason why protecting the critical section of the code with `locking` is so important.

Clojure main approach to concurrency is through reference types (such as atoms or agents) but `locking` is given as last resort when an explicit lock is required. Explicit locking can be

and difficult to implement and debug, so `locking` should not be abused and only appear in some rare interoperability scenarios with pre-existing Java objects.

Note

"Threads contention" usually refers to the situation in which many threads are trying to access the same lock-protected section of code, although contention can be experienced for other contended resources, not just locks.

`monitor-enter` and `monitor-exit` are even lower level primitives and they have even less reasons be used explicitly. They are special forms translating directly to the corresponding Java bytecode that marks the critical section for lock protection:

```
(def v (volatile! 0))
(def lock (Object.))

(try
  (monitor-enter lock) ; ❶
  (vswap! v inc)
  (finally
```

```
      (monitor-exit lock))) ; ❷
  ;; 1
```

❶ `monitor-enter` marks the beginning of the critical section that should be protected from concurrent access.

❷ If we failed to release the lock (for example because of an exception thrown while updating the `volatile`, no other thread would be allowed to execute the code.

Releasing the lock is critical and that's one of the main reason to prefer `locking` which ensures the critical section is always followed by a finally block which exits the monitor. As they are implemented at the moment, `monitor-enter` and `monitor-exit` don't have strong reasons to be used explicitly instead of the `locking` macro. One scenario would be to allow execution of enter/exit operations from different functions, but that's not currently possible [226]. The reader should check the documentation for `ReentrantLock` in the Java documentation for an alternative option.

[226] The need for locking across function calls is rare and complicated to implement correctly. This articles illustrates some of conditions requiring such implementation: www.ibm.com/developerworks/java/library/j-jtp10264/

14.11 SUMMARY

The way Clojure handle concurrency was one of the main selling points of Clojure since the first release. Clojure references are inspired by the reference model in Standard ML, where a reference implies a certain semantic to access a value [227]. Clojure takes ML references to the next level by adding concurrency semantic. There are four main concurrency primitives: var, ref, atom and agent. The chapter also illustrates functions to deal with Java threads and futures.

[227] See "The Definition of Standard ML" book by Milner and others.

15 TYPES, CLASSES, HIERARCHIES AND POLYMORPHISM

The next section explores the functions dedicated to manage or generate new types.

A first group of functions is mainly dedicated to the interoperability with the Java type system. These functions allow close integration with Java types and interfaces, for example to provide framework extensions, class inheritance, access to super-classes and so on.

A second group of functions implements Clojure "object orientation" point of view. Clojure's stance on object oriented programming departs from classic Java in at least two important aspects: it does not allow concrete inheritance and it separates inheritance from interface declaration. The latter is often referred as "a-la carte polymorphism" [228].

The chapter opens with a group of functions dedicated to checking, converting or creating Clojure core types. It then moves to more structured and custom ones, starting from

functions closer to Java semantic and moving up to functions expressing Clojure approach to polymorphism:

- gen-class generate a new Java class with methods delegating to function implementations hosted by the surrounding namespace. It supports a rich set of options to specify how the class should be generated, including implemented interfaces and extended classes.
- gen-interface generates a new interface using a mechanism similar to `gen-class`.
- deftype generates a new class which additionally supports fields (mutable or immutable).
- definterface generate an interface using the same semantic as `deftype` but implemented on top of `gen-interface`.
- proxy creates an anonymous class that is meant to be used through its only instance for the duration of the program. It can extend other classes and override protected methods. It supports closures (it captures surrounding local bindings).
- reify is similar to `proxy`, but it's limited to implementing interfaces. It's also faster than `proxy` in most situations.

- defrecord is a macro that creates a new object based on `deftype` and enables access to fields through a map-like interface.
- defprotocol builds on top of `gen-interface` to create a new interface, but works in conjunction with defrecord to establish dynamic dispatch to functions independently from Java.

Accessing Types

Every Clojure object has a related Java type. Type names are usually composed by a list of nested namespaces (based on Java packages) separated by dots and the name of the type. We can access the type of an object using the function type or class:

```
(class []) ; ❶
;; clojure.lang.PersistentVector

(class "") ; ❷
;; java.lang.String

(class #()) ; ❸
;; user$eval25$fn__26

(class nil) ; ❹
;; nil

(type {:a 1}) ; ❺
;; clojure.lang.PersistentArrayMap

(type (with-meta {:a 1} {:type :custom})) ; ❻
;; :custom
```

❶ `[]` is a type defined by Clojure. The name has the typical format of any other class in Java.

❷ `""` The string type is borrowed directly from Java and has exactly the same type as the Java type.

❸ `#()` anonymous classes generate a new Java class on each evaluation. The printed class name shows the following information: the class name is "user$eval25$fn__26", it was assigned the evaluation id "25" and generated as "fn" expression with id "26". Incremental numbers and name rules are not used in further explicit uses of the type, so the reader should consider them as implementation details.

❹ `nil` doesn't have a class associated with it. If that was the case, then there would be instances of the type `nil`. But this would lead to a contradiction as `nil` is by definition the absence of an object.

❺ We can retrieve the type of a class also using the type function.

❻ In this example we can see the difference between type and class: `type` will use the `:type` key in the metadata of the object if available.

15.1 SYMBOL AND KEYWORD

Note

This section also mentions other related functions such as: `name` and `find-keyword`.

Clojure borrows many of its basic types from Java. Strings, numbers, chars and booleans for example, share the same implementation between the two languages.

`clojure.lang.Symbol` and `clojure.lang.Keyword` are

specific to Clojure and functions like `symbol` and `keyword` create the new corresponding instances:

```
(symbol "s") ; ❶
;; s

(keyword "k") ; ❷
;; :k
```

❶Creating a new symbol "s" using the `symbol` function.
❷Creating a simple keyword ":k" using the `keyword` function.

Symbols, when printed, are similar to strings without the double quotes. A symbol is also different from a string in the following ways:

- It can be further qualified using a namespace. The namespace part of the symbol comes before its name and is followed by "/". For example: `a/b` is the symbol "b" in the namespace "a".
- It supports metadata.
- It can be used as a function to lookup itself into a map.
- Although symbols allow spaces or punctuation, they are not normally used as they don't represent general text.

Warning

Using the `symbol` or the `keyword` functions allows to bypass some validation checking that happens when using their respective constant literal. Please refers to the Clojure reference guide for the rules concerning valid symbols and keywords at clojure.org/reference/reader.

Symbols are used extensively in Clojure to alias var objects, local bindings or function parameters. If a symbol appears in an expression, Clojure attempts to lookup the symbol in the current namespace or surrounding scope:

```
first ; ❶
;; #object[clojure.core$first__4339]

(let [a 1] (inc a)) ; ❷
;; 2
```

❶The symbol "first" resolves to a value in the "user" namespace. This value is a reference to the `clojure.core/first` function.

❷The first appearance of the symbol "a" is used by let to define a local binding. The second appearance is looked up by Clojure in the local context and, if not found, inside the current namespace. "a" in this case was found in the local context.

Symbols also appear after reading Clojure code, which is one of the main reasons they exist. Without this distinction, we wouldn't be able to tell the difference between text that belongs to the program and text that represents data. macro evaluation also produces symbols for similar reasons: a macro is a function that executes after reading "text" but before actual evaluation:

```
(def form (read-string "(a b)")) ; ❶

(map type form) ; ❷
;; (clojure.lang.Symbol clojure.lang.Symbol)

(defmacro reading-symbols [& symbols] ; ❸
  `(map type '~symbols))
```

```
  (reading-symbols a b) ; ❹
  ;; (clojure.lang.Symbol clojure.lang.Symbol)
```

❶read-string takes a string of Clojure text and trasnforms it into code.

❷What appears as text in the input string, becomes a symbol once the text is interpreted as code.

❸A macro is a special function that evaluates at read-time, just after the reader finishes transforming the string into Clojure data structures.

❹The non-evaluated form appears to the macro as a list of symbols.

The presence of a forward slash "/" in the name of a symbol or keyword assigns them to a specific namespace. To access the different parts of a qualified [229] `symbol` we use the function `name` and namespace:

```
  (def ax (symbol "a/x")) ; ❶
  (def bx (symbol "b/x"))

  [(name ax) (name bx)] ; ❷
  ;; ["x" "x"]

  (= ax bx) ; ❸
```

```
  ;; false

  [(namespace ax) (namespace bx)] ; ❹
  ;; ["a" "b"]
```

❶The presence of a "/" in the name of a symbol creates a link between the symbol and a namespace object.

❷Namespace qualification plays a role in equality: here the `symbol` named "x" is defined in 2 different namespaces. To see the name of the `symbol` we need to use the `name` function.

❸Even if the 2 symbols have the same name, they are not equal.

❹The reason they are not equal is because their namespace component is not the same.

The equivalent way to assign a symbol or keyword to a namespace is to use their two arguments constructors:

```
  (def ax (symbol "a" "x")) ; ❶
  (def bx (keyword "b" "x"))

  [(namespace ax) (namespace bx)] ; ❷
  ;; ["a" "b"]
```

❶This example produces an equivalent result as the previous. The namespace portion of the name of a symbol or keyword can also be assigned using the first argument of the two argument constructor.
❷We can see that the symbol and the keyword belongs to the expected namespaces.

On the surface, symbols and keywords are similar. They both help to name elements of the language instead of representing some kind of text. But `keyword` additionally implements a form of caching called "interning". Once a keyword is created, the object instance is cached and reused:

```
(identical? (symbol "a") (symbol "a")) ; ❶
;; false

(identical? (keyword "a") (keyword "a")) ; ❷
;; true
```

❶Two calls to `symbol` using the same letter "a" produce different objects.
❷The effect of interning are visible on `keyword`.

Internal caching of keywords is particularly useful in Clojure because of their common use as keys in maps. Thanks to

interning, if an application process mainly maps, using keywords as keys produces a smaller memory footprints. If keyword interning is better for memory, it is slightly slower in terms of speed:

```
(require '[criterium.core :refer [quick-bench]])

(quick-bench (symbol "a"))  ; ❶
;; Execution time mean : 3.663330 ns
(quick-bench (keyword "a")) ; ❷
;; Execution time mean : 11.791798 ns
```

❶ We compare the cost of creating symbols and keywords. `symbol` is faster in producing a new object, but different objects are allocated for the same `symbol` "a".

❷ `keyword` is slower with the advantage of creating a single instance of the `keyword` ":a" thanks to interning.

The speed difference is related to the additional cost of the caching mechanism. If you need to check if a `keyword` is already cached, you can use `find-keyword`:

```
(find-keyword "never-created") ; ❶
;; nil
```

```
  (find-keyword "doc") ; ❷
  ;; :doc
```

❶When `find-keyword` returns `nil` the keyword was never created.

❷The keyword `:doc` is used in the standard library to add documentation to functions.

[228] Rich Hickey articulated the reasons why there are so many ways to generate classes in Clojure in this post to the Clojure mailing list: groups.google.com/forum/#!msg/clojure/pZFl8gj1lMs/qVfIjQ4jDDMJ

[229] Qualification of a symbol or keyword in Clojure means that they have been assigned a namespace reference. The presence of a namespace relationship is optional.

15.2 TYPE CHECKING AND COERCION

Clojure contains a long list of functions dedicated to check or transform types. We can group these functions as follows:

1. Get or verify the class of an object: `type`, `instance?`, `class`

2. Type coercion: `cast`, `unchecked-byte`, `byte`, `unchecked-char`, `char`, `unchecked-double`, `double`, `unchecked-float`, `float`, `unchecked-int`, `int`, `unchecked-long`, `long`, `unchecked-short`, `short`, `boolean`, `bigint`, `biginteger`, `bigdec`, `rationalize`, `name`, `keyword` and `num`
3. Type checking: `byte?`, `char?`, `double?`, `float?`, `long?`, `short?`, `some?`, `nil?`, `string?`, `decimal?`, `integer?`, `number?`, `ratio?`, `rational?`, `class?`, `fn?`, `ifn?`, `map-entry?`, `map?`, `list?`, `set?`, `vector?`, `record?`, `seq?`, `coll?`, `sequential?`, `associative?`, `sorted?`, `counted?`, `var?`, `keyword?`

type, instance? and class

We can use `type` or `class` to obtain the type of an object. Differently from `class`, `type` first checks to see if a `:type` tag appears in the metadata of the object:

```
(let [add-meta (with-meta [1 2 3] {:type "MyVect
      no-meta [1 2 3]]
  [(type add-meta) (class add-meta) (type no-met
;; ["MyVector" clojure.lang.PersistentVector clo
```

❶ `add-meta` is a vector with an associated `:type` key in metadata. `no-meta` is the same vector without metadata.
❷ We can see that `type` first verifies the presence of the `:type` key and falls back to the class name when no key is present.

We could build a simple form of polymorphism using maps with associated `:type` metadata. For example, a "contact" could be a person or a business and we want to print them differently:

```
(defn make-type [obj t] ; ❶
  (vary-meta obj assoc :type t))

(def person (make-type {:name "John" :title "Mr"
(def manning (make-type {:name "Manning" :owner

(defn print-contact [contact] ; ❸
  (condp = (type contact)
    :person (println (:title contact) (:name con
    :business (println (:name contact) (str "("
    String (println "Contact:" contact)
    (println "Unknown format.")))

(print-contact person) ; ❹
;; Mr John
```

```
  (print-contact manning)
  ;; Manning (Marjan)

  (print-contact "Mr. Renzo") ; ❺
  ;; Contact: Mr. Renzo

  (print-contact nil)
  ;; Unknown format.
```

❶ `make-type` alters an incoming object adding the `:type` key to the metadata map. `vary-meta` is the perfect choice if we want to maintain any existent metadata intact.

❷ We can see a person and a business definition. The two maps differ in number and type of keys. Their metadata contains a `:type` key with value `:person` and `:business` respectively.

❸ `print-contact` takes a contact and print it differently based on type. condp ("cond" with "p"-predicate) creates a condition on top of the `type`. As you can see, we don't need explicit access to metadata.

❹ Each contact prints differently based on its type.

❺ `type` also works with class types when there is no `:type` metadata.

The previous example works okay for a small and fixed number of types. If the types are added frequently and in large numbers, Clojure offers a much better solution based on multimethods. With multimethods, types can be added incrementally to the application without touching existing code. We are going to look into the details of multimethods in the related section and the following is just a basic example:

```
(defmulti print-contact type) ; ❶

(defmethod print-contact :person ; ❷
  [contact]
  (println (:title contact) (:name contact)))

(defmethod print-contact :business
  [contact]
  (println (:name contact) (str "(" (:owner cont

(defmethod print-contact String
  [contact]
  (println "Contact:" contact))

(defmethod print-contact :default
  [contact]
  (println "Unknown format."))

(print-contact person) ; ❸
```

```
  ;; Mr John

  (print-contact manning)
  ;; Manning (Marjan)

  (print-contact "Mr. Renzo")
  ;; Contact: Mr. Renzo

  (print-contact nil)
  ;; Unknown format.
```

❶ `type` is the dispatch function for the multimethod `print-contact`.

❷ We now take each condition that was previously inside condp and extract it into its own `defmethod` declaration. We can add more `defmethod` later on in the life of the application without touching existing code: the dispatch mechanism would take care of routing calls to the right `defmethod`.

❸ We can see that printing the contacts defined in the previous example achieves the same output.

Type naming rules

The type name has to conform to the Java Language Specification [230]. One of the most visible effects of the Java influence is that Clojure idiomatic dashes "-" need to be replaced by underscores "_" (and other restrictions that can be found in the specification related to package names). For example:

```
(in-ns 'my-package) ; ❶
(clojure.core/refer-clojure) ; ❷
(type (fn q? [])) ; ❸
;; my_package$eval1777$q_QMARK___1778
```

❶We define and set "my-package" as the current namespace name.

❷In moving to the new namespace, Clojure doesn't automatically include all of the functions in the standard library. To do that, we can `refer-clojure`.

❸We create a new named function "q?". Both "?" and "-" are not allowed in package names (leaving only a few other separators possible, like "$"). Clojure transforms "?" to "_QMARK_".

`instance?` returns `true` is an object is "instance of" a specific class or interface following Java inheritance rules:

```
(instance? java.lang.Number (bigint 1)) ; ❶
;; true

(instance? java.lang.Comparable 1) ; ❷
;; true
```

❶ `bigint` is one of the available Clojure types representing an infinite precision integer. `java.lang.Number` is the abstract class at the base of all numbers (in Java and Clojure).

❷ `java.lang.Comparable` is an interface. Reference type numbers like `lava.lang.Long` (and many other objects) are comparable.

`instance?`, in production use, should appear sparingly. When a few `instance?` calls accumulate in a cascading "if-then-else" or a cond expression it's probably time to think about something more flexible such as protocols. multimethods are also an option, but protocols are specifically designed to handle type dispatch efficiently.

[230] Versions of the Java Language Specification are available from docs.oracle.com/javase/specs/

15.3 GEN-CLASS AND GEN-INTERFACE

`gen-class` and `gen-interface` are macros dedicated to the generation of Java classes or interfaces. `gen-class` in particular, is the fundamental tool for AOT (Ahead Of Time) compilation along with compile [231]. The generated class can be used in several ways:

- To provide the physical presence of a class on the file system (so Java can load and use it).
- To offer an interface or class to extend from Java.
- To provide a "main" method to run a Clojure application from the command line using Java tooling.
- To extend, access or use existing Java classes.

`gen-interface` supports a few key-value parameters (`:name` to specify the interface name, `:extends` to specify other interfaces to implement and `:methods` which specify the method signatures) and returns the newly generated class:

```
(gen-interface
  :name "user.BookInterface" ; ❶
  :extends [java.io.Serializable]) ; ❷

(ancestors user.BookInterface) ; ❸
;; #{java.io.Serializable}

(reify user.BookInterface ; ❹
  Object (toString [_] "A marker interface for b
;; #object[user$eval20 0x2e "A marker interface
```

❶We request the generation of an interface named `BookInterface` in the package "user".

❷The interface can extend another interface (and optionally more).

❸We can inspect the newly created class with `ancestors`.

❹The interface is now ready to be used, for example to create an object instance implementing it with reify.

Let's now look at `gen-class`. The class generated by `gen-class` is not designed to be called directly, but to work in cooperation with a Clojure namespace that contains the related method implementations. `gen-class` does not work like an ordinary function or macro. For instance, if you call `gen-class` directly, there is no output or side effect:

```
(gen-class 'testgenclass) ; ❶
;; nil
```

❶A direct call to `gen-class` does not produce any noticeable effect, either memory, file system or error.

If we look inside `gen-class` implementation, we can see that `gen-class` checks if the dynamic variable `*compile-files*` is set to `true` before doing any work. But if we try again, even binding the variable does not produce any visible effect:

```
(binding [*compile-files* true]
  (gen-class 'testgenclass)) ; ❶
```

❶Another attempt made at class generation setting `*compile-files*` fails without producing any noticeable effect.

The reason why we are unable to generate a class is that `gen-class` is a macro evaluated when the Clojure runtime is bootstrapping. If we start Clojure from a REPL for example, `*compile-files*` is set to `false`. `gen-class` evaluation captures the `false` value of `*compile-files*` and from that point onward it will ignore any call. We can see `gen-class` in action only if we perform the evaluation from a different context than the current REPL. We can achieve this effect by calling compile on a file that contains the `gen-class` declaration:

```
(spit (str *compile-path* "/bookgenclass.clj") ;
  "(ns bookgenclass)
   (gen-class :name book.GenClass
              :main true)")       ; ❷

(compile 'bookgenclass) ; ❸
;; bookgenclass
```

❶ We create a Clojure file using spit. The simple file contains a namespace declaration and a `gen-class` directive. The file is created in the current root of the classpath, as indicated by `*compile-path*`. Note that the name of the file needs to correspond to the name of the namespace (with added extension ".clj").

❷ By using `:name` we make sure the generated class has a specific package and name. `:main true` enables the generation of a `public static void main[]` Java method.

❸ `compile` searches for files inside the classpath of the running JVM. If should see the name of the newly created class if everything goes as expected.

If we inspect the folder at `*compile-path*` (this could be for example "target/classes" or a different folder depending on your system), we can see a few classes starting with "bookgenclass" generated by Clojure, including a new folder "**compile-path**/book/GenClass.class". We can now call the generated `main` method on `GenClass`:

```
(import 'book.GenClass) ; ❶
;; book.GenClass
```

```
  (GenClass/main (make-array String 0)) ; ❷
  ;; UnsupportedOperationException bookgenclass/-ma
```

❶We `import` the class generated by `compile` during the previous step.

❷Note that the exception refers to a missing `bookgenclass/-main` function.

As soon as we try to call methods defined on the newly generated `book.GenClass` class, we can see that the class assumes the presence of the function `-main` in the `bookgenclass` namespace (the one that was previously written on disk). Note the added prefix "-", which is a default and can be changed. We can provide the missing function to prove the connection between the generated class and the namespace:

```
(spit (str *compile-path* "/bookgenclass.clj")
  "(ns bookgenclass)

   (gen-class :name book.GenClass
              :main true)

   (defn -main [& args] ; ❶
     (println \"Hello from Java!\"))")
```

```
(compile 'bookgenclass) ; ❷

(GenClass/main (make-array String 0)) ; ❸
;; Hello from Java!
```

❶Compared to the previous example, we added a new function called "-main" taking a variable number of arguments.

❷The related class on disk needs to be generated again. The previous version is automatically written over.

❸If we try again to call the static `GenClass/main` method we can see that the generated class correctly looks up the `-main` function in the namespace.

If the namespace exists with the only goal to contain a `gen-class` or `gen-interface` declaration, there is also the option to embed the directive inside ns directly. Please note that the following namespace needs to be written on disk and loaded as part of the Clojure classpath to force the generation of a `book.GenClass2` class on the file system (or you can use `spit` as demonstrated before):

```
(ns bookgenclass2
  (:gen-class :name book.GenClass2)) ; ❶
```

```
  (defn -main [& args] ; ❷
    (println "More greetings from Java!"))
```

❶The embedded `:gen-class` key produces the same effect as the previous isolated `gen-class` call, with the difference that the main function is now generated by default (no `:main true` is necessary).

❷The generated class will call into this specially designed `-main` function.

`gen-class` accepts a long list of parameters to influence the generation of the class, covering the most complicated interoperability scenarios [232]. The rich set of features requires some time to use proficiently. Moreover, there are easier alternatives like `proxy` or `reify` covering the most common scenarios. For this reason, `gen-class` is mostly used as a low level tool when other options fail. One exception is the generation of the main entry point for Clojure applications for which `gen-class` is used pervasively.

[231] AOT compilation is used in Clojure to produce physical class files on disk. The generated classes contain the bytecode necessary to run the Clojure application. When classes are not saved to disk (the default), Clojure just generates them in memory. AOT compilation can be used to avoid distributing Clojure sources or to speedup Clojure start time

[232] `gen-class` and `gen-interface` are well documented, as you can see if you type `(clojure.repl/doc gen-class)` at the REPL

15.4 DEFTYPE AND DEFINTERFACE

`deftype` and `definterface` are low level constructs to generate Java classes or interfaces. Compared to gen-class, `deftype` allows the declaration of attributes for the generated Java class. Another difference is that `deftype` supports (but does not require) AOT compilation. Here's for example a `Point` type in the two-dimensional space:

```
(deftype Point [x y]) ; ❶

(def p (new Point 1 2)) ; ❷
(def p (Point. 1 2))
(def p (->Point 1 2))

(.x p) ; ❸
```

```
;; 1
(.-x p)
;; 1
(. p y)
;; 2
```

❶ `deftype` requires a vector of attributes. In this case the generated class contains two attributes "x" and "y". Note that you can easily redefine `Point`, as the new definition replaces the old one.

❷There are several ways to create an instance of the newly generated class. We can see here a few equivalent options. `new` is a special form that result in the invocation of the related constructor. Appending a dot after the class name is a shorter form to invoke the `new` operator. The last form using an arrow "→" is generated by `deftype` and immediately conveys the fact that `Point` was generated by a `deftype` call.

❸Once an instance is created, we can access the attributes using Java interoperability with the "." (dot) macro. Again, there are multiple options: in case of the presence of both an "x" attribute and an "x" method, the second form `.-x` uniquely identifies the attribute call. The last form (where the "." is detached from the rest) is the most basic form that gives emphasis to the receiver of the call instead of the sender.

After the attribute declaration, `deftype` accepts one or more interface declarations followed by the implementation of the related functions. We could for example compare points based on their distance from the origin of the 2D plane:

```
(defn- distance [x1 y1 x2 y2] ; ❶
  (Math/sqrt
    (+ (Math/pow (- x1 x2) 2)
       (Math/pow (- y1 y2) 2))))

(deftype Point [x y]
  Comparable
  (compareTo [p1 p2] ; ❷
    (compare (distance (.x p1) (.y p1) 0 0)
             (distance (.x p2) (.y p2) 0 0))))

(sort [(->Point 5 2) (->Point 2 4) (->Point 3 1)
;; (#object[user.Point 0x30fd77bf "user.Point@30
;;  #object[user.Point 0x1d81121b "user.Point@1d
;;  #object[user.Point 0x59dd184c "user.Point@59
```

❶The euclidean distance between two points is the length of the straight line connecting them.

❷We define a `Point` including a declaration for the `java.lang.Comparable` interface. This interface requires a method `compareTo` taking the current `Point` instance and another point. The function calculates the distance between the current `Point` and the origin, then the other `Point` and the origin.

❸When we call sort on a collection of points they are returned in increasing order of distance from the origin. However, the type is still missing the instructions on how to print itself.

The reader can probably see what we should do next in order to print a point in a way that it displays its coordinates. We could for example override `toString` from the Object class:

```
(deftype Point [x y]
  Object
  (toString [this] ; ❶
    (format "[%s,%s]" x y)))

(Point. 1 2) ; ❷
;; #object[user.Point 0x65f02188 "[1,2]"]
```

❶ This `deftype` declaration shows how to override `toString` from the Object class. You can see that to access declared fields inside an implemented method, there is no need of interoperability ("x" and "y" are used without reference to "this").

❷ We can see the coordinates of the `Point` printed as part of the object signature.



Note

`java.lang.Object` is the only class accepted by `deftype` that is not an interface.

If we complete our comparable `Point` definition with a `toString` method, we can prove that `sort` is correctly ordering the point on the plane:

```
(deftype Point [x y] ; ❶
  Object
  (toString [this]
    (format "[%s,%s]" x y))
  Comparable
  (compareTo [p1 p2]
    (compare (distance (.x p1) (.y p1) 0 0)
             (distance (.x p2) (.y p2) 0 0))))
```

```
  (sort [(->Point 5 2) (->Point 2 4) (->Point 3 1)

  ;; (#object[user.Point 0x15de9a05 "[3,1]"]
  ;;  #object[user.Point 0x5ed68d62 "[2,4]"]
  ;;  #object[user.Point 0x344c5f2a "[5,2]"])
```

❶A complete `Point` declaration which is both comparable and printable showing that sort is ordering points starting from the closer to the origin (0,0).

`deftype` is one of the fewest options to create truly mutable objects in Clojure. `deftype` attributes are normally declared `public` and `final` (a Java keyword that prevents the attribute from being written once assigned). We can force `deftype` to remove the `final` keyword in 2 ways:

1. By using the `^:unsynchronized-mutable` metadata, attributes are declared not `public` and not `final`. Since attributes stop being public, they need to be exposed with getter/setter functions.
2. By using the `^:volatile-mutable` metadata, we make an attribute not `public`, not `final` and `volatile` [233].



Warning

by making `deftype` attributes mutable, the programmer has to deal with explicit synchronization in case of concurrent access to the type, exactly like it would happen in Java.

In the following example, we create a bean-like `deftype` definition, a Java class that obeys the JavaBean convention for accessing and writing attributes [234]:

```
(definterface IPerson ; ❶
  (getName [])
  (setName [s])
  (getAge [])
  (setAge [n]))

(deftype Person [^:unsynchronized-mutable name ;
                 ^:unsynchronized-mutable age]
  IPerson
  (getName [this] name)
  (setName [this s] (set! name s)) ; ❸
  (getAge [this] age)
  (setAge [this n] (set! age n)))

(def p (->Person "Natasha" "823")) ; ❹
```

```
  (.getAge p)
  ;; "823"

  (.setAge p 23)
  ;; 23
```

❶ `deftype` requires function declarations to appear in an interface. We can create an interface with `definterface`. Note that interface declared functions don't need an explicit "this" parameter. Also note that in this special case, we follow Java convention for method names.

❷ The `Person` type has two mutable attributes `name` and `age`.

❸ To write the attribute we need to use the `set!` function. The exclamation mark is conventionally used to denote mutation.

❹ We make a mistake by inputting a large number of years as string. We fix it by writing the number 23 into the age field.

`deftype` generated classes are especially useful during AOT (Ahead Of Time) compilation, so they can be exported on the file system for Java application to use. In the following example we write the `deftype` declaration to a Clojure file, simulating the normal conditions under which compilation occurs:

```
  (spit (str *compile-path* "/bookdeftype.clj") ; (
    "(ns bookdeftype)
```

```
    (defn bar [] \"bar\")
    (defprotocol P (foo [p]))
    (deftype Foo [] P (foo [this] (bar)))")

(compile 'bookdeftype) ; ❷
;; bookdeftype
```

❶The spit instruction creates a new file called `bookdeftype.clj` in the folder indicated by `*compile-path*`. The file contains a namespace declaration followed by a `bar` function declaration, a defprotocol declaration and a `deftype` directive. We are also showcasing the fact that `deftype` supports `defprotocol` as extension mechanism. The function `foo` declared in the type "Foo" invokes the `bar` function.

❷Once the namespace is on disk, we can ask Clojure to compile the file to produce the actual Java class. `compile` returns the class object just created.

If we now inspect the file system at the location indicated by `*compile-path*`, we can see several files generated by the Clojure compiler. If we now exit the current REPL session and restart a new one, we should be able to use the previously generated class that was stored on disk, but we encounter an error:

```
;; After restarting the REPL
(import 'bookdeftype.Foo) ; ❶
;; bookdeftype.Foo

(def p (Foo.))
(.foo p) ; ❷
IllegalStateException Attempting to call unbound
```

❶After opening another REPL session we import the newly created class. `bookdeftype.Foo` should be part of the classpath, as generated by the previous step.

❷Something goes wrong when we try to use the `Foo` class. The problem is that functions inside the same namespace but outside the `deftype` definition might not load automatically.

Functions that are inside the namespace used by the `foo` function inside the `deftype` definition, are not necessarily loaded just by importing the generated Java class. `deftype` contains an option to load the content of the hosting namespace after importing the generated `deftype`:

```
(spit (str *compile-path* "/bookdeftype.clj")
  "(ns bookdeftype)
   (defn bar [] \"bar\")
   (defprotocol P (foo [p]))
```

```
    (deftype Foo [] :load-ns true P ; ❶
      (foo [this] (bar)))")

(compile 'bookdeftype)
;; bookdeftype
```

❶This is the same namespace definition as before, but now we used the `:load-ns` option in the `deftype` declaration.

If we try again to restart a REPL session and invoke the function `foo` we should not see any error this time:

```
(import 'bookdeftype.Foo) ;; After restarting th
;; bookdeftype.Foo

(def p (Foo.))
(.foo p) ; ❶
"bar"
```

❶After loading the class `Foo` as we did before, we invoke the `.foo` method on a newly created instance. This time it prints "bar" as expected, as the corresponding definition of the function "bar" inside the namespace has been evaluated by Clojure.

[233] The `volatile` keyword in Java has deep implications on the attribute visibility by concurrent threads. A very simplistic view is the following: without `volatile` there is no immediate guarantee that changes made by a thread are seen by other threads. For more information, please see chapter 3.1.4 of <<jcip.net,Java Concurrency in Practice>> book)

[234] The JavaBeans specification is accessible from www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

15.5 PROXY

`proxy` generates a Java class that can either extend or implement other classes or interfaces. The main purpose of `proxy` (going back to when it was first introduced), is to allow complex interoperability scenarios. There are for example Java frameworks that explicitly require clients to extend from a specific class. In that situation you have the option of using `gen-class` (but it requires a specific compilation step) or use `proxy` which returns an instance of the newly created class right away:

```
(def ^Runnable r ; ❶
  (proxy [Runnable] [] ; ❷
    (run [] (println (rand))))) ; ❸
```

```
  (.run r) ; ❹
  ;; 0.16782038795307 64
```

❶Adding type hints to proxies is normally a good idea considering they are consumed as any other Java object. Without type hints, Clojure doesn't know the type of an object until runtime when it's too late for any optimization.

❷We are creating a new class that implements the `java.lang.Runnable` interface.

❸The `Runnable` interface requires to implement a single method called `run`. If we don't provide an implementation, Clojure generates a default one that throws exception when called.

❹We can call `run` directly on the newly created `proxy`.

`proxy` captures any binding surrounding the `proxy` definition form and we can use them as part of our implementations. We could for example implement new functionalities on `java.io.File` without touching the existing interface:

```
  (import '[java.io File])

  (definterface Concatenable ; ❶
    (^java.io.File concat [^java.io.File f])) ; ❷
```

```
(defn cfile [fname]
  (proxy [File Concatenable] [^String fname] ; ❸
    (concat [^File f] ; ❹
      (spit (.getPath f) (slurp this) :append tr
      f)))

(def ^Concatenable etchosts (cfile "/etc/hosts"))
(def ^Concatenable f2 (cfile "temp2.txt"))

(spit f2 "# need to create this file\n") ; ❻

(.concat etchosts f2) ; ❼
```

❶We want `proxy` to extend the existing class `java.io.File`. The generated `proxy` can be used anywhere a file would be used. In this case we want to add a new function (not altering the behavior of the existing ones). The only way to add new behavior is to create an interface containing the new methods.

❷Note that the interface to concatenate two files takes a single argument. This is because the call happens on the first implicit argument `this`. Type hints on the interface are correctly propagated by `proxy` to the corresponding function overrides.

❸ `cfile` creates a new "concatenable-file" that extends `java.io.File` and implements `Concatenable`. The second vector in `proxy` lists the arguments to use to invoke the constructor from the super class. In this case we invoke `java.io.File` with a string representing the file path.

❹It follows a list of function implementations. `concat` copies the content of `this` into the second file argument.

❺We store the files in local vars using def. Note that only the first call to `cfile` generates a new class. The second call to create `f2` creates a new instance but not another class definition. Generated classes conforming the same interface are cached and reused.

❻We need to use `spit` on `f2` to create the file before copying content into it.

❼The `concat` call appends `etchosts` into `f2`. If we open "temp2.txt" we can see an initial comment line "need to create this file" followed by the content of "/etc/hosts".

`proxy` classes with the same signature are cached and reused to enable fast instantiating of similar proxies. Although classes are cached, actual `proxy` instances are not. `proxy` uses instance attributes to store overridden methods along with their closed-over vars. Using functions like `proxy-mappings` or `update-proxy` we can inspect or alter implemented functions after the proxy creation. With reference to the previous example, we can now add an option to automatically create a file before using it to the already existent `etchosts` instance:

```
(update-proxy ; ❶
  etchosts
  {"concat"
    #(let [^File f1 %1 ^File f2 %2]
       (.createNewFile ^File f2) ; ❷
       (spit (.getPath f2) (slurp f1) :append tr
       f2)})

(-> etchosts ; ❸
```

```
    (.concat (cfile "temp3.txt"))
    (.concat (cfile "hosts-copy.txt")))
```

❶ `update-proxy` takes a proxy instance and a map of method names into functions. We call `update-proxy` on the previously created `etchosts` proxy. Inside the map each key represents a method to be overridden/implemented.

❷ Compared to the previous implementation of the same function, we added a call to `.createNewFile` that creates the file when the file is not existent. Note also that the function takes 2 explicit arguments: the first is `this` the second is the target file. The let block was added with the only goal of type hinting the arguments.

❸ We don't need to create "temp3.txt" or "hosts-copy.txt" anymore before adding them into chain of `concat` calls.

The function `proxy` groups together class generation, object creation and functional overrides into a single call (which is in general very convenient). However, you can separate the life cycle phases using functions like `get-proxy-class`, `construct-proxy` and `init-proxy`:

- `get-proxy-class` generates the proxy given one (optional) concrete class to extend and any number of interfaces. With `get-proxy-class` we could for example pre-generate all proxy classes at start-up and avoid the cost of generating them later on.
- `construct-proxy` selects one constructor to invoke and instantiate an actual object. When using `proxy` we get a single choice of constructing arguments, but with `construct-proxy` we can pick different ones based on circumstances.
- `init-proxy` accepts a method overrides map similar to the `update-proxy` map seen before.

In the following example we create a small group of exception classes. Generation of classes happens at definition time, when Clojure parses the namespace containing the calls to `get-proxy-class`. We then provide a `bail` function which creates instances based on the number of arguments. You can use the generated exception in a throw/catch clause. Additionally you can `deref` the exception to access the message describing the error:

```
(import '[clojure.lang IDeref])
(def DocumentException (get-proxy-class Exceptio
(def SyntaxException (get-proxy-class Exception
```

```
(def FormattingException (get-proxy-class Except

(defn bail ; ❷
  ([ex s]
    (throw
      (-> ex
        (construct-proxy s)
        (init-proxy
          {"deref" (fn [this] (str "Cause: " s))
  ([ex s ^Exception e]
    (throw
      (-> ex
        (construct-proxy s e)
        (init-proxy
          {"deref" (fn [this] (str "Root: " (.ge

(defn verify-age [^String s] ; ❸
  (try
    (Integer/valueOf s)
  (catch Exception e
    (bail SyntaxException "Age is not a number"

(try ; ❹
  (let [age "AA"]
    (verify-age age))
(catch Exception e @e))
;; "Root: For input string: \"AA\""
```

❶We create three new exceptions extending from `java.lang.Exception` and implementing the `clojure.lang.IDeref` interface. The generation of the new class happens here at definition time.

❷The function `bail` shows how we can pick different constructors if we have only a message available or the message plus the root exception. `construct-proxy` accepts any number of arguments after the first class object.

❸`verify-age` is an example of possible input verification. If we can't transform the input into a number we catch the original exception and we call `bail` adding a message and the original exception.

❹In this `try-catch` block we call `verify-age` with a purposely wrong age "AA". As expected, we can catch the proxy generated exception instance and call `deref` on it (using the convenient reader literal "@").

`proxy` has several limitations:

- If a class or interface that is extended/implemented by the proxy changes, the proxy continues to use the cached class, even if the class or interface is redefined. This is very unlikely scenario in production code, but is much more common while developing at the REPL.

- `proxy` is not suitable to create complex hierarchies. For example, while you can extend and implement classes or interfaces, you can't extend another `proxy`.
- Performance are good in general, but every method on a proxy performs a lookup to see if there is an override available or not.
- It's also quite important to keep reflective calls under control (using `(set! warn-on-reflection true)` for example) and add the proper type hinting.
- It's "closed polymorphism": either the methods are in the interface at definition time, or you're not going to be able to call them later on even if you `update-proxy` with the correct override.

The lesson learned from `proxy` is that unless you are forced to extend a class from a Java framework in order to use it, you should probably look into reify instead of `proxy` for the creation of quick throw-away instances. If instead your goal is

polymorphism in Clojure, there are better options with protocols and multimethods.

15.6 REIFY

`reify` is a lightweight proxy. It focuses on the essentials: generate a one-off object instance implementing a set of interfaces. `reify` can be useful when a framework (or computation model) requires the creation of an object with a specific interface (like "events", "observables", "listeners" etc.). These objects are short-lived and there is not much value in creating and maintaining an explicit class for them.

In the following example, a Java framework provides classes with properties that are "fired" when something interesting happens (it could be a button click for example). The framework uses the `PropertyChangeSupport` facility in the `java.beans` package to implement this feature:

```
import java.beans.PropertyChangeSupport;
import java.beans.PropertyChangeListener;

public class ClassWithProperty { ❶
   private final PropertyChangeSupport pcs = new
   private String value;
```

```
    public String getValue() { return this.value;

    public void addPropertyChangeListener(Property
        this.pcs.addPropertyChangeListener(listen
    }

    public void setValue(String newValue) { ❷
        String oldValue = this.value;
        this.value = newValue;
        this.pcs.firePropertyChange("value", oldV
    }
}
```

❶ `ClassWithProperty` is an example class with observable properties the way it would be implemented inside the Java framework.

❷ When we alter the content of the `value` field, the class fires a property change call to notify all potential listeners.

Listeners interested to this sort of events should register themselves through the `PropertyChangeListener` interface. We are going to do this using `reify`:

```
(import 'java.beans.PropertyChangeListener)
(import 'ClassWithProperty)
```

```
  (let [observed (ClassWithProperty.)
        listener (reify PropertyChangeListener ; ❶
                   (propertyChange [this evt]
                     (let [{:keys [oldValue newVa
                       (println "Button Clicked!"
    (.addPropertyChangeListener observed listener)
    (.setValue observed "I click")) ; ❸

  ;; Button Clicked! nil I click
```

❶We implement the `PropertyChangeListener` interface using `reify`. This interface has a single method `propertyChange`. Note that we need to pass the implicit `this` parameter even if this is not declared in the interface. We use the `bean` method to transform the event argument so we can access its property like keys from a Clojure map.

❷The reified instance is ready for use and we can now register it to receive events.

❸As soon as we change the value on the Java class, `propertyChange` is invoked and we can see what was the old value along with the new one.

When Clojure needs to provide points of extension, it normally uses defprotocol. A protocol also includes a Java interface that `reify` can implement. We've seen reified `protocols` already

in the book, for example when talking about reducer to implement the `clojure.core.reducers.CollFold` protocol. We could use a similar mechanism to extend reduce-kv to `java.util.HashMap`:

```
(import 'java.util.HashMap)
(import 'clojure.core.protocols.IKVReduce)

(def m (doto (HashMap.) ; ❶
         (.put :a "a")
         (.put :b "b")
         (.put :c "c")))

(defn stringify-key [m k v] ; ❷
  (assoc m (str k) v))

(reduce-kv stringify-key {} m) ; ❸
;; IllegalArgumentException No implementation of

(reduce-kv stringify-key {}
  (reify IKVReduce ; ❹
    (kv-reduce [this f init]
      (reduce-kv f init (into {} m)))))

;; {":b" "b", ":c" "c", ":a" "a"}
```

❶ We create a `java.util.HashMap` instance using `doto` and adding a few key-value pairs invoking the method `.put` directly.

❷ `stringify-keys` is a simple function to assoc a key and a value to a map "m" passed as arguments.

❸ If we try to use `reduce-kv` directly on a `java.util.HashMap` the call generates exception. The reason is that Clojure doesn't have an implementation of `reduce-kv` for `java.util.HashMap`.

❹ We have several options to provide `reduce-kv` to `java.util.HashMap`. The easiest by far is to create a Clojure map out of the Java map and delegate to that version of `reduce-kv`. A better performing version would be to mutate the Java map in place, assuming our application is single-threaded and there are no concurrency issues.

`reify` is generally faster than `proxy`. `proxy` has some advantage for the implementation of interfaces with many methods, where class caching removes continuous generation

of large classes. `reify` is preferable in all cases where you don't need any of the `proxy` features.

15.7 DEFRECORD

`defrecord` generates a deftype-based class that additionally implements Clojure map semantic on declared attributes [235]. Differently from `deftype`, objects produced by `defrecord` are immutable:

```
(defrecord Point [x y]) ; ❶

(def p (Point. 1 2)) ; ❷
(.x p)
;; 1

(def p (map->Point {:x 1 :y 2})) ; ❸
(:x p)
;; 1
```

❶ We declare a `Point` record which includes the attributes "x" and "y".

❷ We can use the `deftype` nature of the record through Java interoperability.

❸ `defrecord` also understands map semantic, so we can access attributes as key in a map, or create a new `Point` from a map containing those attributes.

In addition to simple map operations like access by keyword or assoc-dissoc, `defrecord` is fully integrated with the rest of the Clojure ecosystem: you can compare records with `=`, use them as keys in hash-maps or use them with metadata.

> Note
>
> `defrecord` replaces defstruct in generating custom types with map-like semantic. There are still very few legitimate cases to prefer `defstruct`: please have a look at defstruct to understand more.

Like deftype, `defrecord` can implement any number of interfaces. A record also extends `java.lang.Object` by default. We could write a `Comparable` Point record as follows (you can see a similar example in the deftype section):

```
(defn- euclidean-distance [x1 y1 x2 y2]
  (Math/sqrt
    (+ (Math/pow (- x1 x2) 2)
       (Math/pow (- y1 y2) 2))))

(defrecord Point [x y] ; ❶
  Comparable
  (compareTo [p1 p2]
    (compare (euclidean-distance (.x p1) (.y p1)
             (euclidean-distance (.x p2) (.y p2)

(sort [(->Point 5 2) (->Point 2 4) (->Point 3 1)
;; (#user.Point{:x 3, :y 1} #user.Point{:x 2, :y
```

❶Compared to the deftype version of the same `Point` class, the only difference is the use of `defrecord` instead of `deftype`. After the `defrecord` keyword we have the option to implement one or more interfaces. The methods conventionally follow after the interface name, but they could appear in any order.

❷We can prove points are `Comparable` by sorting them. In this case they are ordered by increasing Euclidean Distance from the origin (0,0) of the coordinate system.

`defrecord` prints more useful information than the corresponding `deftype`. This is because there is a print-method override for records that works with println. If we call str on a record though, we get back a raw string without attributes. We can alter the way `defrecord` transforms into a string by overriding `toString()`:

```
(defrecord Point [x y])
(str (->Point 1 2)) ; ❶
;; "user.Point@78de238e"

(defrecord Point [x y] ; ❷
  Object
  (toString [this]
    (format "[%s,%s]" x y)))

(str (->Point 1 2)) ; ❸
;; "[1,2]"
```

❶If we don’t provide a specific implementation, the default `toString()` formats an object with the name of its class followed by the "@" sign and the result of invoking `hashCode()` on the object (which normally results in a hexadecimal string that roughly maps an address in memory).
❷Let’s extend `Point` to include an override of `toString()`.
❸The new string rendering now contains relevant information.

Apart from the option of using them as map-like classes, `defrecord` plays a fundamental role in the Clojure polymorphic offering, working in conjunction with protocols. We are going to explore this aspect in the following sections.

[235] When we say that a data structure offers "Clojure map semantic", we intend the ability to access attributes by name, similarly to access a value by key in a map

15.8 DEFPROTOCOL

The `defprotocol` macro initializes a polymorphic dispatch mechanism for functions based on types. The macro orchestrates the generation of:

- A native Java interface (making good use of gen-interface). The interface is given the same name as the protocol.
- A var holding data about the methods and their signatures.
- The dispatch functions connecting the protocol methods to their implementations.

We can see the generated artifacts in the current namespace after calling `defprotocol` with a name and a list of methods:

```
(defprotocol MyProtocol ; ❶
  (method1 [this])
  (method2 [this]))

(pprint (vec (.getDeclaredMethods user.MyProtoco
;; [#object[Method "public abstract java.lang.Ob
;;  #object[Method "public abstract java.lang.Ob

(pprint MyProtocol) ; ❸

;; {:on user.MyProtocol,
;;  :on-interface user.MyProtocol,
;;  :sigs
;;  {:method1 {:name method1, :arglists ([this])
;;   :method2 {:name method2, :arglists ([this])
;;  :var #'user/MyProtocol,
```

```
;;  :method-map {:method1 :method1, :method2 :me
;;  :method-builders
;;  {#'user/method2 #object["user$eval1675$fn__1
;;   #'user/method1 #object["user$eval1675$fn__1


(fn? method1) ; ❹
;; true

(fn? method2)
;; true
```

❶ The `defprotocol` definition takes a name (this will be the name of the generated class and local var) and a list of method signatures. The methods are part of the generated interface.

❷ The class `user.MyProtocol` was generated by `defprotocol` and it contains the two expected methods.

❸ The var `MyProtocol` is also generated by `defprotocol`. It contains a map that defines the content of the protocol, including method signatures and method builders. Each method builder generates an instance of the dispatching mechanism for that specific function.

❹ `defprotocol` also creates a function for each method. The body of each function is generated from the method builder with the same name of the function.

`defprotocol` generates functions in the current namespace, one for each method found in the interface declaration. The generated functions contain the dispatch mechanism to find and invoke the right function based on the type of the first argument:

```
(method1 "arg") ; ❶
;; IllegalArgumentException:
;; No single method: foo of interface: user.Foo
```

```
(extend java.lang.String ; ❷
  MyProtocol
  {:method1 #(.toUpperCase %)})

(method1 "arg") ; ❸
;; ARG
```

❶If we try to invoke `method1` just after defining the protocol, Clojure tells us that there is no implementation of that method that can be called.

❷There are several ways to add a valid implementation. One way is to use extend and pass a map from the name of the method on the protocol to the function to invoke.

❸After adding the method for the type `String`, `method1` returns the result of invoking the provided function.

When we "extend" the protocol to a type, we provide a new entry in the dispatch table for that type. However, we cannot extend the protocol for a type that is already part of the protocol declaration:

```
(deftype MyProtocolImpl [] ; ❶
  MyProtocol
  (method1 [this] "MyProtocolImpl::method1"))
```

```
  (method1 (MyProtocolImpl.)) ; ❷
  ;; "MyProtocolImpl::foo"

  (extend MyProtocolImpl ; ❸
    MyProtocol
    {:method1 (constantly "extend::method1")})
  ;; IllegalArgumentException class FooImpl alread
```

❶ `MyProtocolImpl` is a class defined with deftype implementing the interface `MyProtocol` generated by the protocol previously.

❷The protocol method `method1` has a dispatch available for the class `MyProtocolImpl` and the call succeeds.

❸An attempt to extend the same class `MyProtocolImpl` to the protocol `MyProtocol` fails because the class already implements the same interface directly.

Instance based dispatch

Clojure 1.10 introduces a new dispatch method based on metadata:

```
(defprotocol Foo
  :extend-via-metadata true ; ❶
  (foo [x]))
;; Foo
```

```
(foo (with-meta [42] {`foo (fn [x] :boo)})) ; ❷
;; :boo
```

❶The new dispatch mechanism is not enabled by default. We need to pass the `:extend-via-metadata` set to `true` during the protocol creation.

❷We can now define an instance based dispatch on the vector `[42]`. Note the use of syntax-quote (back-tick) for the symbol "foo" to fully qualify the symbol to the current namespace.

The dispatch mechanism was 1.10 is updated as follows: first the presence of the method is checked on the actual class, then in the metadata (when enabled) and finally in the extension table.

From the examples we can see that there are two options to provide the implementation of a protocol method:

- Implementing the protocol interface directly (this can be achieved with any of defrecord, deftype, proxy or reify).
- Extending the protocol to an existing class (using extend and related variants `extend-type` or `extend-record`).

Implementing the interface directly is as fast as extending the protocol at some later point. Let's have a look at the following benchmarks:

```
(require '[criterium.core :refer [bench]])

(defprotocol Bench (m [this])) ; ❶

(deftype DirectBench [] Bench (m [this])) ; ❷

(deftype LaterBench [])
(extend-type LaterBench Bench (m [this])) ; ❸

(let [db (DirectBench.)] (bench (m db))) ; ❹
;; Execution time mean : 3.052222 ns
(let [lb (LaterBench.)]  (bench (m lb)))
;; Execution time mean : 4.168180 ns
```

❶ The protocol `Bench` has a single method `m`.

❷ `DirectBench` implements the protocol at definition time. The class contains a definition of the method `m`.

❸ `LaterBench` does not implement the protocol interface and thus doesn't contain a definition of the method `m`. It is later extended to the protocol.

❹ The difference of 1 nanosecond is completely negligible, the two lookup options perform roughly the same.

Considering what we've seen so far, there is little or no advantage in using `definterface` over `defprotocol`. `defprotocol` offers a more idiomatic way to call interface

methods using normal Clojure functions, including avoiding type hints most of the time. Additionally, protocols can be later extended into existing classes allowing for flexible polymorphism (and also lightweight inheritance). We are going to see a few examples of this in the next section dedicated to extend.

15.9 EXTEND, EXTEND-TYPE AND EXTEND-PROTOCOL



Note

This section also mentions other related functions like `satisfies?`, `extends?` and `extenders`.

`extend` (along with the related helper macros `extend-type` and `extend-protocol`) adds a new dispatch option to a protocol:

```
(require '[clojure.string :as s])

(defprotocol Reflect ; ❶
  (declared-methods [this]))

(extend java.lang.Object ; ❷
```

```
    Reflect
    {:declared-methods
     (fn [this]
       (map
         (comp #(s/replace % #"clojure\.lang\." "cl
               #(s/replace % #"java\.lang\." "jl."
         (.getDeclaredMethods (class this))))})

  (pprint (declared-methods (atom nil))) ; ❸

  ;; ("public jl.Object cl.Atom.reset(jl.Object)"
  ;;  "public jl.Object cl.Atom.swap(cl.IFn,jl.Obj
  ;;  "public jl.Object cl.Atom.swap(cl.IFn,jl.Obj
  ;;  "public jl.Object cl.Atom.swap(cl.IFn,jl.Obj
  ;;  "public jl.Object cl.Atom.swap(cl.IFn)"
  ;;  "public boolean cl.Atom.compareAndSet(jl.Obj
  ;;  "public jl.Object cl.Atom.deref()")
```

❶ We create a `Reflect` protocol that contains a method `declared-methods` that inspect a class to retrieve its publicly declared methods.

❷ With `extend` we can attach this new capability to any class. Since all classes in Java (and Clojure) ultimately extend from `Obejct`, `declared-methods` is now available to any type of argument.

❸ As a test we call `declared-methods` on the object returned by an atom. The resulting list of methods is shortened for readability.

`extend` takes a map from function names (as keywords) to function bodies, enabling simple reuse of functions across different objects. We could for example implement a lightweight version of the corresponding features of Java abstract classes [236]:

```
(defprotocol IFace ; ❶
  (m1 [this])
  (m2 [this])
  (m3 [this]))

(def AFace ; ❷
  {:m1 (fn [this] (str "AFace::m1"))
   :m2 (fn [this] (str "AFace::m2"))})
```

```
(defrecord MyFace []) ; ❸

(extend MyFace
  IFace
  (assoc AFace :m1 (fn [this] (str "MyFace::m1")

(m1 (->MyFace)) ; ❺
;; "MyFace::m1"
(m2 (->MyFace)) ; ❻
;; "AFace::m2"
(m3 (->MyFace)) ; ❼
;;  No implementation of method: :m3 of protocol
```

❶ `IFace` is a protocol with 3 methods `m1`, `m2` and `m3`.

❷The map `AFace` contains default implementations of `m1` and `m2` but no `m3` which doesn't have a common implementation suitable for all types.

❸We declare a new defrecord MyFace. As you can see we don't implement any protocol at declaration time.

❹We now `extend` the protocol IFace to the record MyFace using a "mix-in" of methods coming from the default AFace map plus any override. MyFace is going to "inherit" `m2` default and override `m1` with a different behavior. We still haven't provide a `m3` implementation.

❺The function `m1` has been replaced with a custom implementation and prints a different message from the default `m1` in the `AFace` map.

❻The function `m2` has been "inherited" instead and prints the default message.

❼Invoking `m3` on MyFace produces an error telling us that there's no implementation provided.

defrecord (or deftype) has two options for implementing a protocol:

- Implementing the protocol at declaration time adds the methods to the interface of the generated class, making them visible from Java but also impossible to `extend`. Once the implementation is attached at declaration time it can't change unless the entire record is redefined.
- Implementing the protocol by extension allows instances of the record to be extended to the protocol at some later time, including the option to change the implementation without redefining the record.

In the next example we see that a previously created record changes behavior once we `extend` the protocol defined method `m1`. The same happens to all instances of the same record type running in the application:

```
(def my-face (->MyFace))

(m1 my-face) ; ❶
;; "MyFace::m1"

(extend MyFace
  IFace
  (assoc AFace :m1 (fn [this] (str "new")) ; ❷
               :m3 (fn [this] (str "m3"))))
```

```
(m1 my-face) ; ❸
;; "new"

(m3 my-face) ; ❹
"m3"
```

❶With reference to the previous example, we assign an instance of the MyFace record to the var `my-face`, showing that it prints the custom `m1` implementation as expected.

❷We now repeat the `extend` call, this time changing the implementation of the method `m1`.

❸Previously created instances are now extending the new version of the function `m1`.

❹We also take the opportunity to provide the last missing implementation of `m3` that now prints correctly.

We can extend protocols to interfaces or other protocols. In the next example we model the relationship between nodes in a tree. The fact of being a "Node" and having a "value" is inherited by both branches and leaves in the tree:

```
(defprotocol INode (value [_])) ; ❶
(defprotocol IBranch (left [_]) (right [_]))
(defprotocol ILeaf (compute [_]))
```

```
(extend user.INode IBranch) ; ❷
(extend user.INode ILeaf)

(defrecord Branch [id left right] ; ❸
  INode (value [_] (str "Branch::" id))
  IBranch (left [_] left) (right [_] right))

(defrecord Leaf [id]
  INode (value [_] (str "Leaf::" id))
  ILeaf (compute [_] (str "computed:" id)))

(def tree ; ❹
  (->Branch 1
    (->Branch :A (->Leaf 4) (->Leaf 5))
    (->Branch :B (->Leaf 6) (->Leaf 7))))

(defn traverse
  ([tree]
    (traverse [] tree))
  ([acc tree]
    (let [acc (conj acc (value tree))] ; ❺
      (if (satisfies? IBranch tree) ; ❻
        (into
          (traverse acc (left tree))
          (traverse acc (right tree)))
        (conj acc (compute tree))))))

(traverse tree) ; ❼
```

```
;; ["Branch::1" "Branch:::A" "Leaf::4" "computed
;;  "Branch::1" "Branch:::A" "Leaf::5" "computed
;;  "Branch::1" "Branch:::B" "Leaf::6" "computed
;;  "Branch::1" "Branch:::B" "Leaf::7" "computed
```

❶ `INode`, `IBranch` and `ILeaf` are protocol definitions. `INode` represents everything that is common to the node in a tree and is designed to be "mixed-in" along with other node specializations.

❷ We express the fact that a branch and a leaf are also nodes by extending the `INode` protocol into the `IBranch` and `ILeaf` protocols. This enables the `value` method in `INode` to be invoked on branches and leaves.

❸ The `Branch` defrecord takes an `id` and a `left` and `right` branches. It implements both the `INode` and `IBranch` protocols.

❹ We create a sample `tree` by instantiating and linking together a root, branches and leaves.

❺ The function `traverse` shows that the call `(value tree)` is always valid independently from the type of the node. `traverse` calls recursively on a branch and always accumulate the value of the node. `traverse` also collects the result of calling `compute` on leaves in the final results.

❻ Note the use of `satisfies?` to verify if the `IBranch` protocol contains an implementation for `(type tree)` (the class of the `tree` object).

❼ We can see the result of the traversal going depth first into "Leaf::4" and follow the tree left to right to the last available leaf.

`extend-type` is a shorter form of `extend` that uses fully formed function body, very similarly to what is available at defrecord declaration time:

```
(extend-type MyFace
  IFace
  (m2 [this] (str "MyFace::m2"))) ; ❶

(m2 my-face)
;; "MyFace::m2"
```

❶The `extend-type` syntax is the same found at declaration time for the `MyFace` record. In this case we override the initial default implementation of `m2`.

If the same protocol needs to be extended to many types, it can be repetitive to write an `extend-type` for each of the types the protocol should extend to. `extend-protocol` aggregates all types into a single call:

```
(defprotocol Money
  (as-currency [n]))

(extend-protocol Money ; ❶
  Integer
    (as-currency [n] (format "%s$" n))
```

```
    clojure.lang.Ratio
      (as-currency [n]
        (format "%s$ and %sc" (numerator n) (denom

  (extenders Money) ; ❷
  ;; (java.lang.Integer clojure.lang.Ratio)

  (extends? Money Integer) ; ❸
  ;; true
```

❶The `Money` protocol is extended to only 2 types in this example, but the list could be longer. We could express the same with one `extend-type` for each numeric type, but it would be more verbose.

❷ `extenders` shows that the `Money` protocol has now two extensions.

❸Similarly we can ask if a protocol extends to a specific type with `extends?`.

In this section we've seen how to use `defprotocol` to create interesting relationships between types and their interfaces. `extend` is the idiomatic mechanism to dispatch function calls, but we could include other aspects of an object, not just the type. This is what we are going to see in the next section dedicated to derivation and multimethods.

[236] Java abstract classes contain a mix of implemented methods and abstract methods. You can't instantiate an abstract class until you sub-class it and provide the missing implementation part.

15.10 DERIVE AND MAKE-HIERARCHY

Note

This section also includes examples for `underive`, `isa?`, `parents`, `ancestors` and `descendants`.

`derive` (along with `make-hierarchy`, `underive`, `isa?`, `parents`, `ancestors` and `descendants`) is a group of functions dedicated to create and manage "hierarchies" in Clojure. `isa?`, for instance, is at the core of multimethods dispatching where it provides flexible polymorphism.

The functions described in this section provides the same power to users wishing to create their own dispatching mechanism. While Java conflates inheritance with types, Clojure allows hierarchies to exist as an isolated feature. With Clojure hierarchies we can:

- Use the provided "global hierarchy" or use any number of custom hierarchies independently.
- Create a parent-child relationship with `derive` or remove an existing one with `underive`.
- Query hierarchies for existing relationships (with `isa?`).
- Obtain all derivable entities (with `descendants`).
- Obtain all entities from which we can derive an object (with `ancestors`).
- Fetch all the direct parents of an object (with `parents`).

Hierarchies are supported for symbols, keywords and classes. Clojure allows parent-child relationships both ways, but only for symbols and keywords. Java classes are allowed only as children:

```
(defn custom-hierarchy [& derivations] ; ❶
  (reduce (fn [h [child parent]] (derive h child
    (make-hierarchy)
    derivations))

(def h (custom-hierarchy ; ❷
  [:clerk :person]
  ['owner 'person]
  [String :person]))
```

```
(isa? h 'owner 'person) ; ❸
;; true
(isa? h :clerk :person)
;;true
(isa? h String :person)
;; true
```

❶We are going to use `custom-hierarchy` now on to repeatedly apply `derive` on pairs of parent-child derivations. This allows us to create many of them conveniently.

❷To create a new hierarchy, we invoke `custom-hierarchy` with any number of vector pairs. In this example we use a keyword, symbol and one Java class (but we could use any other object provided it implements the `clojure.lang.Named` interface).

❸We can check the relationships we just created with `isa?`. The three example all return `true`.

Derivation is transitive:

```
(def h (custom-hierarchy
  [:unix :os]
  [:bsd :unix]
```

```
    [:mac :bsd]))

(isa? h :mac :unix) ; ❶
;; true
```

❶The fact that a `:mac` is a `:unix` is not declared explicitly, but it can be inferred traversing the hierarchy on multiple levels.

We can further query the hierarchy to focus on interesting relationships:

```
(def h (custom-hierarchy
  [:unix :os] [:windows :os] [:os2 :os]
  [:redhat :linux] [:debian :linux]
  [:linux :os] [:linux :unix] [:bsd :unix]
  [:mac :bsd]))

(descendants h :unix) ; ❶
;; #{:redhat :linux :debian :bsd :mac}

(ancestors h :mac) ; ❷
;; #{:unix :os :bsd}
```

❶We can use `descendants` to discover "what is a Unix?"
❷Alternatively, we can use `ancestors` to verify the "lineage" of a specific item.

If we want to change the hierarchy, we can remove relationships anytime using `underive`:

```
(def h (custom-hierarchy ; ❶
  [:unix :os]
  [:windows :unix]
  [:mac :unix]))

(isa? h :windows :unix)
;; true

(def h (underive h :windows :unix)) ; ❷

(isa? h :windows :unix)
;; false
```

❶ We want to modify this hierarchy to remove the relationship that says that `:windows` is a `:unix`.

❷ `underive` takes the hierarchy (optional) and the parent-child pair to remove.

`isa?` works on vectors by testing their respective items. We could for example check different inheritance chains in the same hierarchy:

```
(def h (custom-hierarchy ; ❶
  [:clerk :person]
  [:owner :person]
  [:unix :os]
  [:bsd :unix]
  [:mac :bsd]))

(isa? h [:mac :owner] [:unix :person]) ; ❷
;; true
```

❶This hierarchy contains specializations for "clerks" and "owners" as well as flavors of Unix systems.

❷`isa?` tests the pair `:mac` and `:unix` followed by `:owner` and `:person`. It then returns `true` only if all the relationships are `true`.

`ancestors` is useful to retrieve the set of classes or interfaces a class extends or implements. When directed toward Java classes, the result will always contain at least `java.lang.Object`:

```
(ancestors String) ; ❶
;; #{java.lang.CharSequence
;;   java.io.Serializable
```

```
  ;;   java.lang.Comparable
  ;;   java.lang.Object}
```

❶We can use `ancestors` directly with Java classes and use this information to cast the object to one of his superclasses or interfaces.

You can use derivation functions with multimethods to express polymorphic behavior which is not based on types. `isa?`, for instance, is used by `multimethods` instead of plain equality to enable derivation on keywords or symbols. We are going to explore their use in the next section.

15.11 DEFMULTI AND DEFMETHOD

Note

This section also includes examples for `remove-all-methods`, `remove-method`, `prefer-method`, `methods`, `get-method` and `prefers`.

The group of functions and macros presented in this section controls "multimethods" in Clojure. A "multimethod" is a special Clojure function that has multiple implementations. The choice for a specific implementation is done through a dispatch

function that the user has to provide. `defmulti` is the most flexible flavor of polymorphism in Clojure. Compared to protocols, multimethods can dispatch on anything, not just types. Additionally, multimethods don't need the explicit definition of a Java interface and can use custom derivation rules to design complex dispatch hierarchies.

Here's for example how we would design a multimethod to evaluate mathematical operations represented as data. We can create a `calculate` multimethod that dispatches known operations and use a `:default` case to evaluate all other expressions:

```
(def total-payments ; ❶
  {:op 'times
   :expr
   [[:loan 150000]
    {:op 'pow
     :expr
     [{:op 'plus
       :expr
       [[:incr 1]
        {:op 'divide
         :expr [[:rate 3.16]
                [:decimals 100]
                [:months 12]]}]}
```

```
      {:op 'times
       :expr [[:months 12] [:years 10]]}]}]})

(def ops ; ❷
  {'plus +
   'times *
   'divide /
   'pow #(Math/pow %1 %2)})

(defmulti calculate ; ❸
  (fn [form] (:op form)))

(defmethod calculate 'plus ; ❹
  [{:keys [op expr]}]
  (apply (ops op) (map calculate expr)))

(defmethod calculate 'times
  [{:keys [op expr]}]
  (apply (ops op) (map calculate expr)))

(defmethod calculate 'divide
  [{:keys [op expr]}]
  (apply (ops op) (map calculate expr)))

(defmethod calculate 'pow
  [{[x y] :expr}]
  (Math/pow (calculate x) (calculate y)))
```

```
(defmethod calculate nil ; ❺
  [[descr number]]
  number)

(defmethod calculate :default [form] ; ❻
  (throw (RuntimeException. (str "Don't know how

(calculate total-payments) ; ❼
;; 205659.10262863498
```

❶ `total-payments` represents the data structure for the formula: `(* 150000 (Math/pow (+ 1 (/ 3.16 100 12)) (* 12 10)))`. Operations are encoded as Clojure maps with an `:op` key and an `:expr` vector. An expression can be a pair `[<:decription> <number>]` or another operation recursively.

❷ `ops` is a dictionary that translates from the operation as symbol to the actual function to call.

❸The definition of the multimethod starts with a `defmulti` declaration called `calculate`. It takes a single argument "form" and uses the `:op` key to select a specific implementation.

❹Each `defmethod` implements a specific operation. We can destructure the form in the parameters and just concentrate on "expr", a vector of other expressions. For most of the supported operations, we use apply to accept any number of arguments. Note that we call `calculate` on the content of the expression recursively.

❺Calling a keyword on a vector such as `(:op [:int 20])`, invariably gets a `nil` in return (keywords are only able to lookup themselves in maps). When we reach a "leaf" in the data structure, we stop recursion and return the number from the vector pair.

❻ The special dispatch value ":default" receives calls that are not matching `defmethod` declarations. We can use it to prompt the user about a potential missing dispatch value by stopping the evaluation and throwing exception.

❼ `calculate` is now ready to evaluate the `calculate-payments` data. What we see here is how much we are going to repay a 150,000$ loan over 10 years at the annual interest rate of 3.16%.

We could use a custom hierarchy to take care of the potential duplication generated by similar `defmethods`. In the previous example, `defmethod` implementations are changing in relation to the number of arguments supported by each operation. In the following improved implementation, we provide `calculate` with a custom hierarchy to group operations by number of arguments. The custom hierarchy builds on top of a new data structure that replaces the previous dictionary called `ops`:

```
(def ops
 {'plus [+ :varargs] ; ❶
  'times [* :varargs]
  'divide [/ :varargs]
  'pow [#(Math/pow %1 %2) :twoargs]})
```

❶ `ops` is a mapping from the operation symbol (for example `'plus`), to the actual function to call (for example `+`) and the operation type (for example, `:onearg`, `:twoarg` or `:varargs`).

Here is how we could model `calculate` on top of a custom hierarchy expressed by the new `ops` mapping:

```
(defn- add-ops [hierarchy ops] ; ❶
  (reduce
    (fn [h [op [f kind]]] (derive h op kind))
    hierarchy
    ops))

(def hierarchy ; ❷
  (add-ops (make-hierarchy) ops))

(defn resolve-op [ops op] ; ❸
  (first (ops op)))

(do ; ❹
  (def calculate nil)
  (defmulti calculate
    (fn [form] (:op form))
    :hierarchy #'hierarchy))

(defmethod calculate :varargs ; ❺
```

```
    [{:keys [op expr]}]
    (apply (resolve-op ops op) (map calculate expr

  (defmethod calculate :onearg
    [{op :op [x] :expr}]
    ((resolve-op ops op) (calculate x)))

  (defmethod calculate :twoargs
    [{op :op [x y] :expr}]
    ((resolve-op ops op) (calculate x) (calculate

  (defmethod calculate nil
    [[_ number]] ; ❻
    number)

  (defmethod calculate :default
    [form]
    (throw
      (RuntimeException.
        (str "Don't know how to calculate " form))

  (calculate total-payments) ; ❼
  ;; 205659.10262863498
```

❶ `add-ops` creates or modify an existing hierarchy by adding child-parent relationship found in the `ops` argument.

❷ `hierarchy` stores the custom derivation hierarchy as a var object. We use the data mapping `ops` defined previously to create the new hierarchy.

❸ `resolve-op` knows how to inspect the operators symbols mapping and find which function to call.

❹The previous `defmulti` needs redefinition to accept the newly created custom hierarchy. However, Clojure does not replace an already existing `defmulti` definition, so we need to remember to explicitly destroy the previous by setting `calculate` to `nil`.

❺Previously defined `defmethod` also need to be replaced using a new dispatching value: `:varargs`, `:onearg`, `:twoargs`, `nil` or `:default`. `total-payment`, the data structure defined previously, does not need to change. The operator `:plus` for example, dispatches to `:varargs` thanks to the custom hierarchy.

❻Dispatch for `nil` and `:default` needs redefinition because even if they haven't changed, they have never been registered with the new `defmulti` definition.

❼After re-evaluating the multimethod components, `calculate` works as expected on the previous `total-payment` data structure.



Warning

`defmulti` definitions are not overridable. Attempts to redefine a `defmulti` with the same name and namespace of an existing one would silently fail.

Another interesting feature of multimethods is that we can extend them after they've been declared. Differently from protocols (which are also extensible at run-time), we don't need to register a new type to add behavior. The next example contains a different formula with new operators. We are going to use a special form (which is not part of the actual data) to instruct the defmethod about the new operators:

```
(defn sound-speed-by-temp [temp] ; ❶
  {:op 'with-mapping
   :expr
   [{'inc [inc :onearg]
     'sqrt [(fn [x] (Math/sqrt x)) :onearg]}
    {:op 'times
     :expr
     [[:mph 738.189]
      {:op 'sqrt
       :expr
       [{:op 'inc
```

```
            :expr
            [{:op 'divide
              :expr [[:celsius temp]
                     [:zero 273.15]]}]}]}]}])

  (calculate (sound-speed-by-temp -60)) ; ❷
  ;; RuntimeException Don't know how to calculate
```

❶The function `sound-speed-by-temp` takes a temperature and produces the data equivalent of the formula `(* 738.189 (Math/sqrt (inc (/ temp 273.15))))`. The formula calculates the speed of sound at a given temperature, returning the result in miles per hour. The formula is wrapped inside `'with-mapping`, a special operator used by the `:default` dispatch multimethod.

❷If we try to invoke the new formula we get an error, as it contains unknown symbols: `'with-mapping`, `'sqrt` and `'inc`. The `:default` multimethod informs us about about the problem.

The formula to calculate the speed of sound in relation to temperature introduces two new operations: `'sqrt` and `inc`. The next example designs a new `:default` dispatch method to intercept the presence of `'with-mapping`. In the presence of

the special operator, we alter the hierarchy and the operator mappings to introduce the new operations:

```
(defmethod calculate :default
  [{op :op [ops forms] :expr :as form}]
  (if (= 'with-mapping op)
    (do
      (alter-var-root #'hierarchy add-ops ops) ;
      (alter-var-root #'ops into ops) ; ❷
      (calculate forms)) ; ❸
    (throw (RuntimeException. (str "Don't know h

(- (calculate (sound-speed-by-temp -60))   ; ❹
   (calculate (sound-speed-by-temp  20)))
;; -112.64352508635466
```

❶ We use alter-var-root to change the content of the `hierarchy` and add the new operators. Note that this operation is idempotent: the hierarchy doesn't change if the same relationship is already present.

❷ The mapping definitions `ops` also needs updating. Differently from the previous update on the hierarchy, this second `alter-var-root` "upsert" (update or insert) the given operator in the table.

❸ `calculate` can now move forward to the rest of the data structure.

❹ We can see that the speed of sound at -60 celsius (a typical temperature at about 11000 feet of altitude) is around 112 MPH slower than room temperature.

Let's now review other utility functions for multimethods. While `defmethod` adds a new dispatch option to an already existing multimethod, `remove-method` performs the opposite operation (`remove-all-methods` removes all dispatch options instead):

```
(remove-method calculate :twoargs) ; ❶
(calculate {:op 'pow :expr [[:int 2] [:int 2]]})
;; RuntimeException Don't know how to calculate
```

❶ With reference to the previous example involving the `calculate` multimethod, we now proceed to remove the dispatch method for the `:towargs` key.
❷ `calculate` is now unable to calculate the power of a number.

`methods` offers a way to verify dispatching values currently registered with a multimethod (`get-method` retrieves a single dispatch function given a dispatch value):

```
(pprint (methods calculate)) ; ❶
;; {nil
;;  #object[user$eval2033$fn__2035 0x35addc75 "u
;;  :onearg
;;  #object[user$eval2045$fn__2047 0x5df37f84 "u
;;  :default
;;  #object[user$eval2082$fn__2084 0x414b3262 "u
;;  :varargs
;;  #object[user$eval2013$fn__2015 0xcfa4eb3 "us
```

❶ `methods` prints all expected multimethods, confirming the fact that `:twoargs` was removed from the dispatch table.

Finally let's have a look at `prefers` and `prefer-method`. The reason why we are able to express a preference is that

sometimes there is ambiguity to dispatch the correct `defmethod`. This is typical situation extending multimethods to interface types, as Java allows inheritance from multiple interfaces. Here's an example using Clojure data structures:

```
(defmulti edges ; ❶
  "Retrieves first and last from a collection" ty

(defmethod edges java.lang.Iterable [x] ; ❷
  ((juxt first last) (seq x)))

(defmethod edges clojure.lang.IPersistentList [x
  ((juxt first last) (seq x)))

(edges (list 1 2 3)) ; ❹
;; IllegalArgumentException Multiple methods in m

(prefer-method edges clojure.lang.IPersistentLis
(edges (list 1 2 3)) ; ❺
;; [1 3]
```

❶ `edges` is a multimethod to retrieve the items at the edge of a collection. It dispatches using the type function.
❷ Many collections are `Iterable` in Clojure so we defined a version of `edges` to deal with them.
❸ We are also interested in a specific version for Clojure lists.
❹ If we try to call `edges` on a list we discover that lists are both `IPersistentList` and `Iterable`.
❺ `prefer-method` establishes a preference order to use in case of ambiguous dispatch.

We can check the presence of entries from the table of preferences using `prefers`. Here's for example the rich set of preferences defined by print-dup:

```
(pprint (prefers print-dup)) ; ❶
;; {clojure.lang.ISeq
;;  #{clojure.lang.IPersistentCollection java.ut
;;  clojure.lang.IRecord
;;  #{java.util.Map clojure.lang.IPersistentColl
;;    clojure.lang.IPersistentMap}
;;  java.util.Map
;;  #{clojure.lang.Fn}
;;  clojure.lang.IPersistentCollection
;;  #{clojure.lang.Fn java.util.Map java.util.Co
;;  java.util.Collection #{clojure.lang.Fn}}
```

❶ `prefers` shows quite a few choices for `print-dup`. Almost everything is printable in Clojure. Collections are a typical example. `print-dup` defines a format suitable for serialization of an object to file and has to deal with the many similar interfaces required for interoperability.

Recursive Multimethods [237]

Multimethods are function definitions wrapped by a dispatch mechanism. When we declare a `defmulti` definition, a var contains the dispatch mechanism in the current namespace. When we declare a `defmethod`, the function declaration is not part of the namespace but part of an instance of `clojure.lang.MultiFn` class.

`defmethod` signature is the following: `[multifn dispatch-val & fn-tail]`. `fn-tail` is called a "vararg", a parameter that represents one or more values at the end of the `defmethod` declaration. We usually pass a vector of arguments (it must be the same declared by the dispatch function) and the body of the function. `fn-tail` is used to create an anonymous function by splicing the arguments with `(fn ~@fn-tail)`, so we are also allowed an optional name for the anonymous function. The name can be used for recursion inside the body of the `defmethod` without passing through the dispatch:

```
(defmulti recursive identity) ; ❶

(defmethod recursive 1 recursive-impl [cnt] ; ❷
  (if (< cnt 5)
    (do (println cnt)
        (recursive-impl (inc cnt))) ; ❸
    cnt))

(recursive 1) ; ❹
;; 1
```

```
;; 2
;; 3
;; 4
;; 5
```

❶For the purpose of this example, the dispatch function is identity.

❷This `defmethod` definition contains a locally bound name `recursive-impl` between the dispatch value and the arguments.

❸ `recursive-impl` can be used inside `defmethod` to call the same definition without passing through the dispatch mechanism.

❹When we call `recursive` with a number, we dispatch to the corresponding (and only) multimethod available. The number is then used as the initial value for the recursion. On reaching "5" the recursion ends as expected.

The local name is also useful for debugging:

```
(defmulti throwing identity)
(defmethod throwing :default throwing-impl [x] ;
  (throw (RuntimeException. (str "Problems with"

(throwing (symbol " this fn")) ; ❷
;; RuntimeException Problems with this fn  user/
```

❶This multimethod throws an exception on purpose to illustrate how the local name `throwing-impl` is propagated to the stack trace.

❷When the exception prints on the screem, we an see that the name of the anonymous function has been used in the generated class name.

The name of the anonymous function is attached to a specific `defmethod` definition so we can track it down in stack traces while debugging.

[237] Thanks to Rupert Ede for suggesting the inclusion of this section in the book

15.12 SUMMARY

This chapter illustrates the facilities available in Clojure to generate types and manage polymorphic dispatch. Clojure offers different ways to create types, starting from the more powerful but lower level gen-class to the more specialized proxy or reify. Clojure also offers different options for polymorphic dispatch. multimethods allow a great level of flexibility to select the receiver for a method call. At the same time, type based dispatch with protocols is a faster alternative when speed is important.

16 VARS AND NAMESPACES

Before introducing the functions and macros in this section, it's useful to refresh a few definitions about Clojure namespaces, vars and local bindings.

Namespaces

- A "namespace" is an instance of `clojure.lang.Namespace` class and is essentially a container for other objects. It contains a mapping table and an alias table.
- The "mapping table" is a dictionary associated with each namespace. It contains a mapping between Clojure symbols and objects such as vars or classes. Functions like intern or def adds items into the table, while ns-map shows its content.
- The "alias table" is another type of dictionary associated with a namespace. It contains relationships with other namespaces, using a symbol as key and a namespace name as value. Items can be added to the alias table with `:as` during namespace creation or with the alias function. ns-aliases shows its content.

- Once a namespace is created, a reference is added to the "global namespace repository". This is a static map inside the `clojure.lang.Namespace` class itself. It derives that namespaces are "global", that is, the running process doesn't need to hold an explicit namespace reference to keep them alive (from the garbage collector perspective).

Libraries

- A "library" is a source file named after the namespace declaration it contains. The concept of library overlaps substantially with namespace, but libraries dictates a few conventions to store and reuse Clojure code from files.
- You can see what libraries are currently loaded by using loaded-libs.
- The existence of a library implies the existence of the related namespace, while the opposite is not always true (for example, create-ns creates just a namespace, not a library). require loads a library and any additional transitive dependency.

Vars and Bindings

- A "var" object is an instance of the `clojure.lang.Var` class (to avoid confusion with other languages, we are going to use the shorter form "var" instead of "variable"). A var can be considered a pointer to a single value, with the option to hold multiple values when concurrency is involved.
- The "root binding" is the optional value that the var can be associated with. It is visible from all threads. Assigning the root binding is the most typical use of a var.
- A "dynamic binding" is a value that the var associates with a specific thread. Each thread can access a different value for the same var. The feature is not enabled by default, but it can be enabled by passing the `^:dynamic` metadata during var definition.
- A var that allows dynamic bindings is called "dynamic" and its name is conventionally surrounded by "*" (colloquially called "earmuffs").
- A var can be "bound" (when it has a root value) or "thread-bound" when it has at least one thread bound value.

- Compared to root bindings, thread bindings can "stack up". This is achieved by nesting binding forms. Previous dynamic values are preserved while moving back from each nested `binding` scope.
- A var is always associated with at least one namespace (through the mapping table). It follows that vars (like namespaces) are also global.

Vars behavior is heavily influenced by the metadata attached to them. The most important are:

- `:dynamic` : indicates that the var is thread-local enabled and can stack-up values on a per-thread basis. A var marked as dynamic can be used with set! or var-get.
- `:inline` and `:inline-arities` enable var "inlining", an alternative implementation that takes precedence over the root binding or thread binding when present. See definline for more information.

Vars are also functions themselves (`ifn?` on a var returns `true`) and this is integral part of Clojure indirection mechanism (what makes Clojure a dynamic language). For example, the evaluation of the expression `(+ 1 1)` in the `user` namespace triggers the following steps:

- The symbol `+` is resolved through the namespace mappings. Assuming an entry with key `+` is present (this is always the case for the `user` namespace) the value of the entry is returned. The value for the key `+` in the namespace mapping is the var object with name `clojure.core/+`.
- If the value from the mappings is a var object, it gets invoked with the given arguments. This is equivalent to call `(#'clojure.core/+ 1 1)`. Note that the var object is used as a function in this case.
- When a var object is invoked as a function, it delegates the call to the first available option between the inlined version, the thread local value or the root binding of the var (in this order). Specifically, `+` has an inlined version for two arguments, so the inlined version takes precedence. If more arguments are present, `+` is not inlined, so the root binding is invoked instead.

At the beginning of this detailed lookup chain sits the namespace mapping for the symbol `+`. At the end of the chain, we have the bytecode that was generated during compilation. To understand what's happening when we call `(+ 1 1)` at the REPL, we can reproduce each step individually:

```
(def user-ns-mapping (ns-map 'user))  ; ❶
(def lookup-var ('+ user-ns-mapping)) ; ❷
(lookup-var 1 1) ; ❸
;; 2
```

❶First, we save a reference to the content of the mapping table for the `user` namespace.
❷Second, we access the value for the symbol `+`. This is an object of type var.
❸Finally, we invoke the var object on two arguments. This in turns

We are going to start the chapter with the functions dedicated to create vars. Apart from the creating the var object itself, these functions also produce side effects on namespace data structures that allow Clojure to work dynamically.

16.1 DEF, DECLARE, INTERN AND DEFONCE

def

`def` is a special form. When invoked with a "name" and a "body" (the most common invocation) produces the following effects:

- Creates and returns a `clojure.lang.Var` object (if not already existent) with name "name".
- Assigns the var object with the evaluated "body".
- Creates (or replaces) the corresponding entry in the current namespace mapping (also referred as "interning" of the var into the namespace).
- Adds information like the source file, row and column numbers to the var object.

We can verify all of the above with the following steps:

```
(ns myns) ; ❶

(type (def mydef "thedef")) ; ❷
;; clojure.lang.Var

mydef ; ❸
"thedef"

(identical? (var mydef) ((ns-map 'myns) 'mydef))
;; true

(meta (var mydef)) ; ❺
;; {:line 1,
;;  :column 7,
;;  :file "/private/var/form-init3920299731829241
```

```
  ;;  :name mydef,
  ;;  :ns #object[clojure.lang.Namespace 0x68ff111
```

❶ns creates a new namespace (unless already existent) and assign the value of the `*ns*` dynamic variable to the related namespace object. The reason we create a new namespace, is that we want to be sure to control all the aspects of the example without external influences. The namespace just created is in "pristine" conditions: it has no changes to the mapping table nor the alias table.

❷As expected `def` returns an object of type `clojure.lang.Var`.

❸Typing the symbol "mydef" at the REPL produces the lookup of the var from the namespace mapping by default. Also automatically, the REPL returns the value of the var, not the var object itself. In other words, the REPL automatically call deref on variable references.

❹We can fetch the var object from the namespace mappings with `ns-map`. The 2 expressions shown here retrieve the same object. The last one is the most explicit: `ns-map` returns the mappings corresponding to `'myns`. We then access the map using the symbol `'mydef` as key.

❺Finally, we can see what additional information are attached to the var metadata. The REPL instance where the form was evaluated is using a temporary file that we can see here as ":file".

`def` also supports custom metadata and a documentation string (normally referred as "docstring"):

```
(ns myns)

(def ^{:created-at "date"} ; ❶
  def-meta-doc
  "A def with metadata and docstring." 1) ; ❷

(clojure.repl/doc def-meta-doc) ; ❸
;; -------------------------
;; myns/def-meta-doc
;;   A def with metadata and docstring.

(:created-at (meta (var def-meta-doc))) ; ❹
;; "date"
```

❶A metadata literal map was added between `def` and the name of the definition. It contains a creation date for the definition.

❷The documentation string should go just after the `def` name and before the body of the definition.

❸We can see the documentation extract by `clojure.repl/doc`.

❹The metadata are attached to the var object as expected.

declare

We can also invoke `def` without a body. The generated var is said to be "unbound":

```
(ns myns)

(def unbound-var) ; ❶
;; #'myns/unbound-var

(type unbound-var) ; ❷
;; clojure.lang.Var$Unbound
```

❶We can call `def` with just a name and no body.
❷The evaluation of `unbound-var` produces a placeholder object of type `Var$Unbound` which represents the missing body.

Unbound vars can be useful in case of mutually recursive definitions. Clojure even provides a `declare` macro which creates a named unbound var. `declare` helps to convey the semantic meaning of creating a var without a body. In the following example we create a state machine that verifies the presence of alternating "0" and "1" in a string:

```
(declare state-one) ; ❶

(def state-zero ; ❷
  #(if (= \0 (first %))
      (state-one (next %))
      (if (nil? %) true false)))

(def state-one  ; ❸
  #(if (= \1 (first %))
      (state-zero (next %))
      (if (nil? %) true false)))

(state-zero "0100100001") ; ❹
;; false
(state-zero "0101010101") ; ❺
;; true
```

❶ `(declare state-one)` is equivalent to `(def state-one)` but it immediately clarifies the reason for the missing body: `declare` it's a warning about the presence of a recursive cycle in the following definitions.

❷ `state-zero` defines a function that calls `state-one` which is not defined yet. `declare` allows `state-one` definition regardless of the presence of a body.

❸ `state-one` is defined again, this time with a proper body. The var object linked to `state-one` is not created again, but it is assigned a body for evaluation.

❹The state machine only matches patterns starting with "0" and then alternating "1" and "0". This pattern has a series of repeating zeroes and therefore is not valid.

❺A valid pattern triggers the correct answer.

intern

`intern` works similarly to `def`, but offers the possibility to create definitions in other namespaces:

```
(ns myns)

(create-ns 'ext) ; ❶

*ns* ; ❷
;; #object[clojure.lang.Namespace 0x68ff111c "my
```

```
(intern 'ext 'ext-var 1) ; ❸
;; #'ext/ext-var

((ns-map 'ext) 'ext-var) ; ❹
;; #'ext/ext-var

(intern 'yet-to-exist 'a 1) ; ❺
;; Exception No namespace: yet-to-exist found
```

❶ `create-ns` create a new namespace `'ext`. Compared to other namespace related macros, `create-ns` does not change the current namespace.

❷ We can check the content of the dynamic variable `*ns*` to verify that we are still in the same namespace.

❸ `intern` performs the equivalent of `def` targeting the `'ext` namespace.

❹ The mappings for the `'ext` namespace contains the newly generated entry.

❺ Note however that `intern` does not create a non existent namespace automatically and throws exception if the namespace does not exist.

`intern` is useful for all programmatic definitions of vars. We could for example create a new namespace and a list of vars

without the need of creating a Clojure source file:

```
(def definitions ; ❶
  {'ns1 [['a1 1] ['b1 2]]
   'ns2 [['a2 2] ['b2 2]]})

(defn defns [definitions] ; ❷
  (for [[ns defs] definitions
        [name body] defs]
    (do
      (create-ns ns)
      (intern ns name body))))

(defns definitions) ; ❸
;; (#'ns1/a1 #'ns1/b1 #'ns2/a2 #'ns2/b2)
```

❶The `definitions` map contains namespaces to be created as keys and definitions as vectors.

❷ `defns` takes the map of definitions and iterate through namespaces and required definitions using for. We need to remember to `create-ns` to make sure the namespace exists before we call `intern`.

❸ `defns` returns the list of vars created and mapped to the respective namespaces.

defonce

`defonce` is another `def` variant. `def` allows redefinition by "upserting" the current namespace mappings: if an entry exists for the definition, then the existing var is updated removing the old value (what the var evaluated to). `defonce` first checks for the presence of an already defined var for the given name and only creates the new definition if not already existing:

```
(def redefine "1") ; ❶
(defonce dont-redefine "1")
(def redefine "2")
(defonce dont-redefine "2")

redefine
;; "2"

dont-redefine
;; "1"
```

❶ `def` is compared to `defonce` by repeating the same definitions and attempting to change their value from "1" to "2". `defonce` does not change its value.

One good reason to use `defonce` is to protect important data from accidental redefinition. One common usage of `defonce` is with component systems like Stuart Sierra's Components [238]. The "system" exists as `def` in some namespace and we want to

prevent a namespace reload to throw away our system without being able to properly shutting it down. For this specific use case is advisable to use `defonce`.

Note

`defonce` should not be confused with `(def ^const)`. The presence of the metadata `:const` on a definition produces an effect similar to "inlining": all references to the definition are replaced verbatim with the value of the definition. After evaluation, the var definition using `(def ^const)` effectively stops existing, without a namespace mapping or a var being created.

---

[238] github.com/stuartsierra/component

16.2 VAR, FIND-VAR AND RESOLVE

Note

This section also mentions other related functions such as: `clojure.repl/dir-fn`, `clojure.repl/dir`, `bound?`, `thread-bound?`, `namespace`

---

var

`var` is a special form that retrieves a var definition (the `clojure.lang.Var` object associated with a symbol) from the current or another namespace. It throws exception if the var is not found and does not create namespaces or symbols automatically:

```
(var a) ; ❶
;; CompilerException java.lang.RuntimeException:

(def a 1)
(var a) ; ❷
;; #'user/a

(var test-var/a) ; ❸
;; CompilerException java.lang.RuntimeException:

(create-ns 'test-var)
(intern 'test-var 'a 1)
(var test-var/a) ; ❹
;; #'test-var/a

(= (var a) (var test-var/a)) ; ❺
;; false
```

❶ The var "a" does not exist in the current namespace and `var` throws exception. Note that "a" is evaluated as a symbol even if we don't explicitly qualify it as such. `var` behaves as a macro in this respect.

❷ After defining "a", `var` correctly retrieves the var object.

❸ `var` accepts namespace-qualified symbols such as `test-var/a`. But the namespace does not exist yet, let alone the var "a" in that namespace.

❹ After creating the namespace and a definition for "a", `var` returns the var object associated with `test-var/a`.

❺ The var objects are different despite the fact that the definition is about the symbol "a" in both cases.

`var` and the syntax literal `#'`

`var` also has an equivalent reader macro `#'`:

```
(var clojure.core/+) ; ❶
;; #'clojure.core/+

#'clojure.core/+ ; ❷
;; #'clojure.core/+

(identical? (var clojure.core/+) #'clojure.core/+) ; ❸
;; true
```

❶ `var` retrieves the var object given the qualified symbol `clojure.core/+`. The printer method for vars is instructed to print the var object as the reader syntax literal.

❷ The Clojure reader interprets "#" as lookup into the reader syntax table for `'` single quote that follows. This is translated internally as `(var clojure.core/+)` and then compiled as usual, resulting in exactly the same form being evaluated.

find-var

`find-var` works similarly to `var` but requires fully qualified symbols as input (a symbol name preceded by a namespace and separated by a forward slash, such as: `a/b`). It does not throw exception in case the var isn't found:

```
(find-var 'user/test-find-var) ; ❶
;; nil

(find-var 'test-find-var) ; ❷
;; IllegalArgumentException Symbol must be names
```

❶ `find-var` returns `nil` when searching for the yet to be created `test-find-var`. Note that the name of the var is qualified with `user`, a namespace that is guaranteed to exist at the REPL.

❷ Not fully qualified symbols are not accepted by `find-var`.

Prefer `find-var` to `var` if you don't want to use try-catch to deal with non-existing vars.

resolve and **ns-resolve**

`resolve` and `ns-resolve` adds a couple of additional options on top of `var` and `find-var` while searching for vars. `resolve` always uses the current namespace for searching (the content of the `*ns*` dynamic var) while `ns-resolve` can be given a specific namespace to search for.

The first added feature is searching for Java classes as well as var objects:

```
(resolve 'Exception) ; ❶
;; java.lang.Exception

(resolve (symbol "[I")) ; ❷
;; [I
```

❶ `resolve` returns the class associated with the symbol `'Exception`. This is how, at the REPL, we can just type "Exception" without having to import the class first. Clojure imports most of the `java.lang.*` classes in namespaces automatically.

❷ An array of integers has a class type in Java named using the open square bracket followed by the letter "I". We can retrieve this kind of class creating the symbol from a string.

The second feature offered by `resolve` is to prevent resolution of some symbols. We could use the feature to selectively allow override of values for some vars, but not for "protected" ones:

```
(defn replace-var [name value] ; ❶
  (let [protected #{'system}]
    (when (resolve protected name)
      (intern *ns* name value))))

(def mydef 1) ; ❷
(def system :dont-change-me)

(replace-var 'x 2) ; ❸
;; nil

(replace-var 'mydef 2) ; ❹
mydef
```

```
;; 2

(replace-var 'system 2) ; ❺
system
;; :dont-change-me
```

❶ `replace-var` is a function that swaps existing var values with new ones. It contains a set of "protected" vars that cannot be overridden. The set is passed to `resolve` to avoid resolution of an existing var.

❷ We define `mydef` and `system`. The latter is protected by `replace-var`.

❸ `resolve-var` accepts non-existing vars and does nothing.

❹ Existing vars that are not protected are replaced with a new value/expression.

❺ However, we are unable to replace `system` which is protected and not visible by `resolve`.

clojure.repl/dir-fn and **clojure.repl/dir**

`dir-fn` and `dir` are two functions in the `clojure.repl` namespace using var related functions to retrieve or print public definitions in a namespace. For example:

```
(require '[clojure.repl :refer [dir-fn dir]])
(require 'clojure.set)

(dir-fn 'clojure.set) ; ❶
;; (difference index intersection
;;  join map-invert project rename
;;  rename-keys select subset?
;;  superset? union)

(dir clojure.set) ; ❷
;; difference
;; index
;; intersection
;; [..]
;; union
;; nil
```

❶ `dir-fn` returns the list of public definitions in a namespace. In this case we can see public definitions for `clojure.set`. The list is sorted alphabetically.

❷ `dir` retrieves the same list and prints it on the screen returning `nil`.

bound? and **thread-bound?**

A `clojure.lang.Var` object can be "bound" or "thread-bound". A var is said to be "bound" if an expression was

assigned as the value of the var. In this case all threads share the same view of the var. But a var can also be assigned a value in the scope of a specific thread. In this last case, each thread would see its own isolated value:

```
(def ^:dynamic *dvar*) ; ❶

((ns-map 'user) '*dvar*)
;; #'user/*dvar1*

(bound? #'*dvar*)
;; false

(thread-bound? #'*dvar*)
;; false

(binding [*dvar* 1] ; ❷
  [(bound? #'*dvar*)
   (thread-bound? #'*dvar*)])
;; [true true]

(def avar) ; ❸

(bound? #'avar)
;; false

(thread-bound? #'avar)
```

```
  ;; false

  (intern *ns* 'avar 1) ; ❹
  (bound? #'avar)
  ;; true

  (thread-bound? #'avar)
  ;; false
```

❶ `*dvar*` is a var marked as `^:dynamic` using metadata. The var was not bound at definition. We can see that the var object has been created and entered to the namespace mappings. Both `bound?` and `thread-bound?` returns `false`.

❷With binding we can open a thread-aware context to set the dynamic var. Both functions return `true` inside the binding.

❸We define now a normal var called `avar`. The var is still unbound at definition and both functions agree on this fact.

❹If we `intern` the var with a value (which looks up the var in the mapping and update its root) we can see that the var is now bound. But from the perspective of the current thread, the var still does not have a value and `thread-bound?` returns `false`.

Both functions accept more than one parameter returning `true` only if all vars are bound:

```
(def a 1) (def b 2) (def c 3)
(bound? #'a #'b #'c) ; ❶
;; true
```

❶Both `bound?` and `thread-bound?` accept any number of vars. In this case it verifies if all the vars are bound.

16.3 ALTER-VAR-ROOT AND WITH-REDEFS

alter-var-root

Like `def` or `intern`, `alter-var-root` changes the root value of a var object. Differently from them, `alter-var-root` does not touch the namespace mappings. `alter-var-root` accepts a function from the old value to the new as a parameter:

```
(def a-var {:a 1})

(alter-var-root
  #'a-var
  update-in [:a] inc) ; ❶
;; {:a 2}
```

❶ `alter-var-root` changes the root binding of the var `a-var`. It accepts a function from the old value into the new (in this case `update-in`), plus any additional parameters. The new value is returned.

`alter-var-root` performs the change atomically: while changing the var, the corresponding var object is locked for reading or writing ("synchronized" in Java terminology):

```
(def a-var 1)

(future ; ❶
  (alter-var-root
    #'a-var
    (fn [old]
      (Thread/sleep 10000)
      (inc old))))

;; blocking call for 10 seconds
a-var ; ❷
;; 2
```

❶ `alter-var-root` executes from another thread with a 10 seconds delay.

❷ If we immediately try to access `a-var` to read its value, we are blocked until `alter-var-root` is done.

`alter-var-root` and `definline`

Even if `alter-var-root` is effectively swapping the root binding of a var, the effects might not be seen if the var is inlined. "Inlined" means that the var contains an alternative implementation in metadata that is invoked instead of the root binding:

```
(definline timespi [x] `(* ~x 3.14)) ; ❶
(alter-var-root #'timespi (fn [_] (constantly 1))) ; ❷
(timespi 100) ; ❸
;; 314.0

(alter-meta! #'timespi dissoc :inline-arities :inline) ; ❹
(timespi 100) ; ❺
;; 1
```

❶ `timespi` is a simple function that multiplies a number with the Pi constant. The var that contains `timespi` is inlined: the metadata on the var contains a pre-compiled evaluation of the function.

❷ We call `alter-var-root` on the var object, to change the root binding to a new function that always returns 1.

❸ Calling `timespi` invoked the inlined version of the function that hasn't changed.

❹ We need to change or remove the related metadata on the var object to force evaluation through the root binding.

❺ `timespi` now invokes the root binding of the var object.

with-redefs and **with-redefs-fn**

`with-redefs` and `with-redefs-fn` create a context in which a list of vars temporary assumes a different root binding.

The previous bindings are established after returning back from the enclosing form:

```
(defn fetch-title [url] ; ❶
  (let [input (slurp url)]
    (last
      (re-find #"Title: (.*)\." input))))

(def sample-article "Some Title: Salary increase

(with-redefs [slurp (constantly sample-article)]
  (= "Salary increases announced" (fetch-title "
;; true
```

❶ `fetch-title` is designed to request the content of a given web address and search the text with a regular expression. The function as is, would need to connect to the network to execute the request.

❷ Instead of testing the function assuming a network connection is available, we use `with-redefs` to temporarily change the function attached to the `slurp` var, forcing it to return a sample string.

`with-redefs` is a macro wrapping `with-redefs-fn` so we don't need to use explicit var objects and wrapping functions.

The following example is equivalent to the previous but slightly less readable:

```
(with-redefs-fn {#'slurp (constantly sample-arti
  #(= "Salary increases announced" (fetch-title
;; true
```

❶ `with-redefs-fn` requires some more syntactic sugar compared to `with-redefs` but the two functions are equivalent.

The example above is related to testing and this is not a coincidence. It is dangerous for a production Clojure application to contain calls to `with-redefs` or `with-redefs-fn`, as they are not thread safe:

```
(defn x [] 5) ; ❶
(defn y [] 9)

(dotimes [i 10] ; ❷
  (future (with-redefs [x #(rand)] (* (x) (y))))
  (future (with-redefs [y #(rand)] (* (x) (y)))))

[(x) (y)] ; ❸
;; [0.6022778872500808 9]
```

❶ `x` and `y` are example functions returning a 5 and 9.

❷ We create a few threads, each one of them changing `x` or `y` inside a `with-redefs` form. We expect the respective vars to be intact after going through the threads.

❸ `x` or `y` (or both) could have been permanently changed by `with-redefs`. Note: you could get different results than the one presented in this example.

The problem with the example above is that `with-redefs` might access an altered value of a variable while it's in the process of being changed by the other threads. It then replaces back a root binding that was not the original. To solve the problem, we should use dynamically thread-bound vars:

```
(defn ^:dynamic x [] 5) ; ❶
(defn ^:dynamic y [] 9)

(dotimes [i 10] ; ❷
  (future (binding [x #(rand)] (* (x) (y))))
  (future (binding [y #(rand)] (* (x) (y)))))

[(x) (y)] ; ❸
;; [5 9]
```

❶Now `x`, and `y` are both dynamic vars.

❷`with-redefs` does not bind values to the thread local state of the var. We need binding to access the thread local state.

❸Threads stop interfering with each other, as each binding is isolated and does not touch the root binding.

16.4 BINDING

Note

This section also mentions other related functions such as: `with-binding`, `with-binding*`, `push-thread-bindings`, `pop-thread-bindings`, `bound-fn`, `bound-fn*`.

`binding` creates a context in which vars can be assigned a thread-local value, leaving the root binding untouched. Dynamic vars can be used to share simple state between calls in the same thread without necessarily passing the same parameter to all functions. The following example assumes a concurrent system allocating one thread per-request (common case for web applications). If the system is invoked with `trace=enabled` we collect a lot more information about that specific request:

```
(in-ns 'user)

(def ^:dynamic *trace*) ; ❶

(defmacro trace! [msg & body] ; ❷
  `(do
     (when (thread-bound? #'*trace*)
       (set! *trace* (conj *trace* ~msg)))
     ~@body))

(defn params [query] ; ❸
  (let [pairs (clojure.string/split query #"&")]
    (trace! (format "Handling params %s" pairs)
      (->> pairs
         (map #(clojure.string/split % #"="))
         (map #(apply hash-map %))
         (apply merge)))))

(defn handle-request [{:strs [op arg1 arg2]}] ;
  (let [op (resolve (symbol op))
        x (Integer. arg1)
        y (Integer. arg2)]
  (trace!
    (format "Handling request %s %s %s" op x y)
    (op x y))))

(binding [*trace* []] ; ❺
  (let [query "op=+&arg1=1&arg2=2"
```

```
          res (handle-request (params query))]
      (pprint *trace*)
      res))

;; ["Handling params [\"op=+\" \"arg1=1\" \"arg2=
;;  "Handling request #'clojure.core/+ 1 2"]
;; 3
```

❶ The `*trace*` var is `^:dynamic` and unbound.

❷ `trace!` is macro that adds a trace message before performing the evaluation of a form. Tracing is subject to the presence of a thread-local value in `*trace*` which happens only if `*trace*` appears in a `binding` context. The trace is enabled if a `binding` is present anywhere upstream.

❸ `params` takes a string of URL encoded parameters and transforms it into a map. Note the use of the `trace!` macro to produce a message when the trace is enabled.

❹ `handle-request` performs the actual interpretation of the parameters. It follows the same pattern to trace data.

❺ Request processing starts inside a `binding` block that initializes `*trace*` to an empty vector. The presence of the binding signals the need for tracing the request. This aspect could be attached to parameters in the request itself.

Note how, in the example above, the vector containing the messages is never passed as parameter to the other functions. Messages are shared through the `*trace*` dynamic var without requiring any special synchronization apart from the enclosing `binding` form.

bound-fn

The other functions in this section, `with-binding`, `with-binding*`, `push-thread-bindings`, `pop-thread-bindings`, `bound-fn` and `bound-fn*` are lower level or more specific (and rarely used). Here's a summary:

- `with-bindings` and `with-bindings*` (macro and function version) is similar to `binding` but requires a map of binding and explicit var objects. Use them to customize the way to fetch var objects or if you want to store the binding pairs (var objects and values) separately.

- `push-thread-bindings` and `pop-thread-bindings` are lower level primitives that are used to set thread local binding. They should be called with the following sequence: push bindings, evaluate body, pop bindings in a finally block. It's unlikely you'll ever need to process bindings in a different way and that's what `binding` does for us already.
- `bound-fn` and `bound-fn*` (macro and function version) are helpers to wrap existing functions so thread-locals are propagated when the function creates a new thread from within a `binding` form.

`bound-fn` deserves some additional explanation. Dynamic vars values are thread local and don't share their state with other threads. There is however the legit case in which a new thread is created inside an already existing `binding` context:

```
(def ^:dynamic *debug*)

(defn debug [msg] ; ❶
  (when (and (thread-bound? #'*debug*) *debug*)
     (println "Debugging..." msg)))

(binding [*debug* true] ; ❷
  (.start (Thread. #(debug "from a thread."))))
;; nil
```

```
(binding [*debug* true] ; ❸
  (.start (Thread. (bound-fn* #(debug "from a th
;; Debugging... from a thread.
;; nil
```

❶ `debug` is a function that checks the presence of a thread binding for the `*debug*` dynamic var.

❷ If we wrap function calls inside a `binding` form that set `*debug*` to `true`, we expect debugging messages to print on screen. But messages are not appearing. The reason is that the inner form is creating a separate thread and the thread local bindings are by definition invisible to the new thread.

❸ `bound-fn*` wraps the function passed as argument into another function that, before calling the inner function, copies the binding from the current thread into the next. We can see the debugging message correctly appearing.

`bound-fn` helps propagating bindings correctly. Clojure itself uses `bound-fn` internally with future and other concurrency primitives.

16.5 WITH-LOCAL-VARS, VAR-GET AND VAR-SET

`with-local-vars` creates a thread local scope where vars can be created and mutated. Along with `var-get` and `var-set`, `with-local-vars` allows an imperative, yet thread-safe, style of programming. Here's for example a `count-even` function that increments a counter each time a number is divisible by 2:

```
(defn ++ [v] ; ❶
  (var-set v (inc (var-get v))))

(defn count-even [xs] ; ❷
  (with-local-vars [a 0]
    (doseq [x xs]
      (when (zero? (rem x 2))
      (++ a)))
    @a))

(count-even (range 10)) ; ❸
;; 5
```

❶ `++` implements mutation with increment on a var object. The function would like to imitate the way Java mutates local variables in loops, with the difference that Clojure provides concurrency handling without any explicit locking.

❷ `with-local-vars` creates a new var object called "a" and then assigns its thread-local value to 0. The var "a" is now available inside the form for reading or writing.

❸ As expected, there are 5 even numbers in the range from 0 to 9.

Note that multiple threads accessing `count-even` would perform isolated changes to the counter resulting in thread-safe mutations. Also note how the var "a" was never defined explicitly (for example with def) outside the function. Even if that was the case, (e.g. `(def a 10)` appears somewhere at the top level), the definition of "a" inside `with-local-vars` would shadow any external reference.

The only reason to use `with-local-vars` is to force imperative style mutable locals instead of idiomatic recursion, a concession given to mimic other Lisps that have this option

available. However, `with-local-vars` is rarely seen in real-life Clojure code.

16.6 NS, IN-NS, CREATE-NS AND REMOVE-NS

`ns` is one of the most fundamental macros in Clojure, as a namespace needs to exist ahead of declaring anything else. The call to `(ns myns)` to create the namespace "myns", expands into a call to `in-ns` which performs the following preliminary actions:

- Search the global namespace repository for an existing namespace "myns". If none is found, it creates a new one which is added to the global namespace repository.
- When creating a new namespace, `in-ns` also injects the "default imports" into the namespace mapping, for example the list of `java.lang.*` classes that are directly available to other forms declared in the namespace.
- It sets the dynamic var `*ns*` to "myns" (the REPL uses this information to change the cursor name).

The `ns` macro adds the following actions on top `in-ns`:

- Adds an entry to "myns" mapping table for each public var in `clojure.core` (allowing access to the standard library functions without explicit require).
- Adds "myns" (as a symbol) to the list of loaded libraries which stored in the private dynamic variable `*loaded-libs*` (you can access the content of `*loaded-libs*` using the function `loaded-libs`).

Definitions of vars implicitly use `*ns*` as their namespace, so every definition is automatically associated to the current namespace declaration (or `user` if none):

```
(def a 1) ; ❶
(ns-name (.ns #'a))
;; user

(ns ns1)

(def b 1) ; ❷
(ns-name (.ns #'b))
;; ns1
```

❶ The Clojure REPL bootstraps itself in the `user` namespace. We can verify this creating a definition for the var "a" and asking for the name of its namespace. There is no standard library function to access the namespace of a var object directly, so we use Java interop to call the `.ns` method. `ns-name` extracts the name of the namespace.

❷ After creating a new namespace "ns1", we repeat the operation and create another definition. This time the var namespace is "ns1".

`ns` supports a large set of options to alter mappings and aliases at the time the namespace is created. Most of the options are also available as top-level functions and can be used independently (by making an implicit reference to the current `*ns*` namespace). Since the function works like the related option in `ns`, please refer to the following for details: refer-clojure, refer, require, use, import and gen-class. Here's an example that showcases all of them

```
(create-ns 'a)

(ns my.ns
  (:refer a) ; ❶
  (:refer-clojure :exclude [+ - * /]) ; ❷
  (:import java.util.Date)  ; ❸
```

```
    (:require (clojure.set))  ; ❹
    (:use (clojure.xml))      ; ❺
    (:gen-class))             ; ❻
```

❶ `:refer` copies public vars from the mapping table of namespace "a" into the mapping table of namespace "my.ns". Note that we created "a" before.

❷ `:refer-clojure` is the same as `:refer` but for `clojure.core` only. We see here one of the supported options `:exclude` that prevents a few arithmetic functions to be available in "my.ns".

❸ `:import` creates a new entry in "my.ns" mapping table for the one or more Java classes. The Java class is added using just the name as key, not the entire package.

❹ `:require` assumes the existence of a file on the current classpath. The conventionis that all dot-separated words except the last form the folder path from the root of the classpath (e.g. "clojure/") and the last word is the name of the file (e.g. "set.clj"). If the file is present, the corresponding namespace is created.

❺ `:use` loads the corresponding file like `:require` but additionally `:refer` to all public symbols.

❻ `:gen-class` enable the same feature of gen-class during the namespace creation. In this case, if the namespace is compiled, it will also generate a class with the same name.

Compared to `ns` and `in-ns`, `create-ns` is purely about creating the namespace. There are only two side effects of using `create-ns`: one is the entry created in the global namespace

repository, the other is the import of classes from the `java.lang.*` package which becomes immediately available in the mapping table. Let's verify `create-ns` behavior:

```
new-ns/var-def ; ❶
;; CompilerException java.lang.RuntimeException:

(contains? (set (map ns-name (all-ns))) 'new-ns)
;; false

(create-ns 'new-ns) ; ❸
(intern 'new-ns 'var-def "now it's working") ; ❹
(contains? (ns-map 'new-ns) 'var-def) ; ❺
;; true

new-ns/var-def ; ❻
;; "now it's working"

(contains? (set (map ns-name (all-ns))) 'new-ns)
;; true

('Integer (ns-map 'new-ns)) ; ❽
;; java.lang.Integer
```

❶The namespace "new-ns" does not exist. We get an error if try to access anything in that namespace.
❷The global namespace repository confirms there is no such namespace.
❸ `create-ns` creates the namespace (and doesn't change the current one).
❹ `intern` creates a new var "var-def" in `new-ns`.
❺We also verify that `intern` added the related entry in "new-ns"'s mapping table.
❻The simple expression now resolves correctly, looking up the namespace in the global repository, then the symbol in the mapping table.
❼One final check for the presence of the newly created namespace.
❽Java classes belonging to the `java.lang.*` namespace (apart a few exceptions) are immediately available in the namespace mapping table.

A namespace can also be deleted with `remove-ns`. Let's remove the namespace "new-ns" that was created in the previous example:

```
(remove-ns 'new-ns) ; ❶
```

```
new-ns/var-def ; ❷
;; CompilerException java.lang.RuntimeException:
```

❶ `remove-ns` removes the given namespace from the global repository.

❷As expected var resolution stops working.

`remove-ns` should be used sparingly, because it could be still referenced in other namespaces, generating all sort of problems. If a namespace is referenced by another namespace, the garbage collector would be unable to claim the corresponding object reference. Even more dangerously, if the same namespace is created again, there could be a mix of namespaces pointing at the old reference and the new:

```
(create-ns 'disappear) ; ❶
(intern 'disappear 'my-var 0)
(refer 'disappear :only ['my-var])
;; nil

my-var
;; 0

(remove-ns 'disappear)
;; #object[clojure.lang.Namespace 0x1f780201 "di
```

```
(.ns #'my-var) ; ❷
;; #object[clojure.lang.Namespace 0x1f780201 "di

(create-ns 'disappear) ; ❸
(intern 'disappear 'my-var 1)

my-var ; ❹
;; 0

@#'disappear/my-var ; ❺
;; 1
```

❶ The sequence of calls create a new namespace "disappear", including a var "my-var". The var is imported into the mappings of the current namespace, where it evaluates to 0 as expected.

❷ After removing the namespace "disappear" we can see that "my-var" is keeping the namespace alive. The garbage collector cannot remove the namespace object until there are no more references to it.

❸ A namespace with the same name and var is created again, this time with value 1.

❹ However, the local "my-var" entry is still running the old copy of the var.

❺ We can see that the new var should evaluate differently, but we have a stale copy in our namespace.

To avoid the danger related to removing namespaces, we should track all the references to the namespace from other namespaces and cascading-delete them accordingly. This is tedious and error prone, but luckily there are robust libraries like "clojure/tools.namespace" to help us out [239].

[239] Please see github.com/clojure/tools.namespace

16.7 ALIAS, NS-ALIASES AND NS-UNALIAS

The group of functions `alias`, `ns-aliases` and `ns-unalias` is the small programming interface that controls alias mappings for a namespace. After creating a namespace, the alias table is usually empty:

```
(create-ns 'com.web.tired-of-typing-this.myns)
(ns-aliases 'com.web.tired-of-typing-this.myns)
;; {}
```

❶ `ns-aliases` shows the content of the alias table for namespace "com.web.tired-of-typing-this.myns". The name is annoyingly long on purpose.

The namespace just created has a somewhat long name, but this is a pretty common scenario. If the created namespace contains a var, we can reference the var without the need to require the namespace:

```
(intern 'com.web.tired-of-typing-this.myns 'myva
com.web.tired-of-typing-this.myns/myvar ; ❷
;; 0
```

❶ `intern` creates a var "myvar" in the namespace.
❷ Evaluating `com.web.tired-of-typing-this.myns/myvar` produces the expected results of accessing and dereferncing "myvar".

Even if vars are accessible by fully qualifying them, typing the package name every time is repetitive and not very readable. We can assign the long namespace an `alias` and use it to refer to the original namespace from that point onward:

```
(alias 'myns 'com.web.tired-of-typing-this.myns)

(ns-aliases *ns*) ; ❷
;; {myns #object[clojure.lang.Namespace 0x58d455
;;  "com.web.tired-of-typing-this.myns"]}

myns/myvar ; ❸
0
```

❶ `alias` adds an entry to the alias table for the current namespace.
❷ If we inspect the alias table, we can see a new key "myns" and the namespace object as value.
❸ Using the `alias` instead of the original namespace has the same effect of making access to the variable.

ns offers a similar feature with `:require` when using the `:as` option, but only if the namespace can load from a file (which is the most common case):

```
(ns anotherns (:require [clojure.set :as s])) ; 

(ns-aliases 'anotherns) ; ❷
;; {s #object[clojure.lang.Namespace 0x5d1fa08b
```

❶ns (and require) offers the option of creating the namespace and the alias at the same time using the `:as` option. This works only if the namespace is defined on a file in the classpath. This is the case for `clojure.set` which is part of the standard library.

❷We check the alias table to see that the symbol "s" is now pointing at the `clojure.set` namespace.

A final mention for `ns-unalias`, which not surprisingly, removes an alias entry from the alias table. With reference to the preceding example, we can decide to remove the alias:

```
(ns-aliases 'anotherns) ; ❶
;; {s #object[clojure.lang.Namespace 0x5d1fa08b
(ns-unalias 'anotherns 's) ; ❷
```

```
(ns-aliases 'anotherns) ; ❸
;; {}
```

❶The alias to `clojure.set` created previously is still visible in the alias table.

❷ `ns-unalias` takes a namespace symbol and the entry key to remove.

❸The alias was removed as expected.

16.8 NS-MAP AND NS-UNMAP

Each namespace holds a reference to a mapping table. The mapping table is a map containing pairs of symbol keys into var object references. The mapping table is the main device to resolve names when symbols are evaluated. The mapping table is never empty if the namespace is created using the function of the standard library (and not, for example, Java interop):

```
(ns myns (:require [clojure.pprint :refer [pprint

(pprint (ns-map 'myns)) ; ❶
;; {primitives-classnames #'clojure.core/primiti
;;  +' #'clojure.core/+',
;;  Enum java.lang.Enum,
;;  decimal? #'clojure.core/decimal?,
```

```
;;  restart-agent #'clojure.core/restart-agent,
;;  sort-by #'clojure.core/sort-by,
;; [...]

(distinct (map #(map type %) (ns-map 'myns))) ; ❶
;; ((clojure.lang.Symbol clojure.lang.Var)
;;  (clojure.lang.Symbol java.lang.Class))
```

❶The list of mappings in a newly created namespace is quite large. What we show here is just the initial part.

❷The only 2 variants of pairs present are symbol to var or symbol to class mappings.

We can remove an entry from the mapping (for example to remove the content or a deleted namespace) using `ns-unmap`. Here we remove `+` from the functions available in "myns":

```
(ns-unmap 'myns '+) ; ❶
;; nil

(+ 1 1) ; ❷
;; Unable to resolve symbol: +
```

❶The entry for "+" was removed from the namespace.

❷As a consequence we are unable to sum numbers.

The `+` function has not disappeared from the system, it's just unavailable in "myns". We can put it back with refer:

```
(refer 'clojure.core :only ['+]) ; ❶
(+ 1 1)
;; 2
```

❶refer has access to the mapping table of the namespace and adds back the removed entry.

16.9 NS-PUBLICS, NS-INTERNS, NS-IMPORTS

`ns-publics`, `ns-interns` and `ns-imports` are different filters on the results returned by `ns-map`. `ns-map` retrieves the content of the namespace mapping table as is, which includes both public vars, private vars and classes. Before moving forward with the examples, let's setup a conveniently empty namespace:

```
(ns user)

(defn clean-ns [ns] ; ❶
  (let [ks (keys (ns-map ns))]
    (doseq [k ks]
```

```
      (ns-unmap ns k))))

(ns myns)
(#'user/clean-ns 'myns)
(clojure.core/alias 'c 'clojure.core) ; ❷

(c/ns-map 'myns) ; ❸
;; {}
```

❶ `clean-ns` completely removes any mapping from the mapping table of a given namespace.

❷After cleaning the namespace, we need a quick way to access the functions in the standard library without creating any mapping. We can use `alias` to create an alias "c" to `clojure.core`.

❸We can see that the namespace is definitely empty.

Now that we cleared the namespace we can add a few definition to test the filters:

```
(def normal-var :public) ; ❶
(def ^:private private-var :private)
(c/import 'java.lang.Number)

(c/ns-map 'myns) ; ❷
;; {private-var #'myns/private-var,
```

```
;;  Number java.lang.Number,
;;  normal-var #'myns/normal-var}

(c/ns-publics 'myns) ; ❸
;; {normal-var #'myns/normal-var}

(c/ns-interns 'myns) ; ❹
;; {private-var #'myns/private-var
;;  normal-var #'myns/normal-var}

(c/ns-imports 'myns) ; ❺
;; {Number java.lang.Number}
```

❶ Note that `def` is a special form and it doesn't have an entry in the mapping table of `clojure.core` or any other namespace, so we don't need to prefix the call with `c/def`.

❷ The setup for the mapping table is complete and we can see its content.

❸ `ns-publics` retrieves only entries where the value is a public var.

❹ `ns-interns` retrieves only entries where the value is a public or private var.

❺ `ns-imports` retrieves only entries where the value is a class (more properly, anything that is not a var).

16.10 REFER, REFER-CLOJURE, REQUIRE, LOADED-LIBS, USE, IMPORT

We already used most of the functions mentioned in this section talking about vars and namespaces. The ns macro for instance, delegates most of its options to one or more of `require`, `refer`, `use`, `import`, `refer-clojure`. We are going to see additional details here.

refer

`refer` adds entries to the mapping table of a target namespace using another namespace as source. We are going to to start

from a completely clean namespace to show how it works:

```
(ns user)

(defn clean-ns [ns]
  (ns 'user)
  (create-ns ns)
  (let [ks (keys (ns-map ns))]
    (doseq [k ks]
      (ns-unmap ns k)))
  (ns-map ns))

(clean-ns 'myns) ; ❶
;; {}

(binding [*ns* (the-ns 'myns)] ; ❷
  (refer 'clojure.core
    :only ['+ '-]
    :rename {'+ 'plus '- 'minus}))

(ns-map 'myns) ; ❸
;; {minus #'clojure.core/-
;;  plus #'clojure.core/+}
```

❶The mapping table for the "myns" namespace is completely empty.

❷ `refer` does not accept a target namespace, but it uses the current value of `*ns*` as the source namespace. We can use a binding form to change `*ns*` temporarily to use "myns" as source. `refer` supports a few options, such as `:only` to filter the symbol to import and a `:rename` map to change their name.

❸The list of the mappins in "myns" confirms we imported two symbols.

`refer-clojure` is pretty much the same as `refer` but restricted to use `clojure.core` as the source for importing. So the previous could be written:

```
(binding [*ns* (the-ns 'myns)] ; ❶
  (refer-clojure
    :only ['+ '-]
    :rename {'+ 'plus '- 'minus}))
```

❶Rewriting the previous `refer` in terms of `refer-clojure`.

require

`require` unit of work is the "library", a file available on the classpath that follows a specific naming convention. As a side effect of loading a library, it also creates a new namespace:

```
(contains? (set (map ns-name (all-ns))) 'clojure
;; false

(require 'clojure.set)

(contains? (set (map ns-name (all-ns))) 'clojure
;; true
```

❶ Assuming a freshly started REPL session from the Clojure uberjar, the `clojure.set` namespace does not exist, despite the fact that the classpath contains a file called `clojure/set.clj` that contains the recipe for creating that namespace. But the file was never required.

❷ After calling `require` on the namespace, it will be created once the file is loaded. The namespace eventually appears in the global repository.

We can use `loaded-libs` to verify what libraries have been loaded so far. This can be the result of calling `require` or use explicitly, or the result of traversing other `require` statements transitively because another library requires them:

```
(def libs (loaded-libs))

(pprint libs) ; ❶
;; #{clojure.core.protocols clojure.core.server
;;   clojure.instant clojure.java.browse clojure
;;   clojure.java.javadoc clojure.java.shell clo
;;   clojure.repl clojure.string clojure.uuid cl

(require '[clojure.data :refer [diff]])
(def nss (set (map ns-name (all-ns))))
(pprint (diff libs nss)) ; ❷
;; [nil   ; ❸
;;  #{user clojure.core clojure.set clojure.data
;;  #{clojure.core.protocols clojure.core.server
;;    clojure.instant clojure.java.browse clojur
;;    clojure.java.javadoc clojure.java.shell cl
;;    clojure.repl clojure.string clojure.uuid c
```

❶This is the typical results of running `loaded-libs` on a freshly opened REPL. We can see a few of the usual files from the Clojure standard library.

❷An interesting aspect to verify is that there is definitely a namespace corresponding to each library, but the opposite is not necessarily true. Namespaces created with create-ns or in-ns for instance, are not registered as libraries.

❸We interpret the results of clojure.data/diff as following: there are no libs that are not also namespaces (as expected).

❹Interestingly, there are a few namespaces that don't have a library, although namespaces like `clojure.set` or `clojure.data` definitely have a corresponding file in the Clojure distribution. One reason for this behavior, is that macros like ns starts their existence as special forms and they are later redefined in the bootstrap process. The special form of `ns` lacks any feature related to updating the global namespace repository. Some namespaces are created using the special form `ns` while others are using the macro redefinition.

❺Finally, we can see the list of namespaces with a corresponding library file, the large majority.

Note that `require` is not going to work on a namespace that is not backed by a file, even if the namespace already exists:

```
(create-ns 'test-require) ; ❶
(require 'test-require) ; ❷
;; Could not locate test_require__init.class or
```

❶ `create-ns` creates a new namespace.

❷ `require` does not work on namespaces that are not created as a side effect of loading the corresponding lib.

`require` is often used with `:as` or `:refer` options. This is true especially in conjunction with the ns macro, but they can be used directly:

```
(ns myns)

(require
  '[clojure.set ; ❶
    :as se ; ❷
    :refer [union]] ; ❸
  '[clojure.string ; ❹
    :as st
    :refer :all]) ; ❺
;; WARNING: reverse already refers to: #'clojure
;; WARNING: replace already refers to: #'clojure
```

❶Note the use of the square brackets. They are required to include options, as there could be different set of options for different libraries.

❷ `:as` creates an alias in the current namespace that is resolved to the library.

❸ `:refer` imports the specified symbols in the namespace mappings.

❹You can list many libraries in a single `require` call.

❺When `:refer :all` is specified, all public vars from the library are imported into the namespace mappings.

Note

When an entry in the namespace mapping table is replaced by a different entry with the same key, the Clojure compiler emits a warning. In the example above we can see that both reverse and replace were already present in the mapping table and they have been replaced by new definitions coming from the `clojure.string` namespace. This is one of the reasons why `:refer :all` should be used sparingly, to avoid accidental redefinition of core functions.

The general wisdom about importing mappings from libraries, is to restrict them to the minimum necessary for readability and prefer aliases when possible. Once all symbols from a library are imported into the current namespace mappings, it becomes difficult to understand where they are defined, with the risk of polluting the namespace of unwanted mappings over time.

use

`use` mixes the semantic of `require` with the options of `refer`:

```
(ns myns)

(use '[clojure.java.io ; ❶
       :only [reader file] ; ❷
       :rename {reader r}] ; ❸
   :verbose
   :reload-all) ; ❹

;; (load "/clojure/java/io")
;; (in-ns 'myns)
;; (refer 'clojure.java.io :only '[reader file]
```

❶ `use` requires quoted symbols when used outside the ns macro. We can quote the vector to quote all symbols within.
❷ `:only` restricts the number of symbol to import to the local namespace mapping. Also `:exclude` is supported with opposite intent.
❸ `:rename` works by offering the option of interning symbols with a different name than the one of the original namespace.
❹ `use` and `require` also supports `:reload`, `:reload-all` and `:verbose`. Reloading forces the reload of the file to re-sync with possible changes on the file system. `:reload-all` also reloads any transitive dependency. `:verbose` prints information about the namespace loading, in particular regarding the dependency tree.

`use` does not support aliasing with `:as`, which could lead to very long lists of `:only` symbols. Possibly for this reason, it has been abused in the past to import all symbols all the time, attracting a good amount of negative feedback. Nowadays, `require` with aliasing offers a better option to establish namespace dependencies than `use` and is generally preferred. But `use` can still be used for renaming options that are not supported by `require`.

import

`import` is the macro equivalent of `refer` for classes instead of vars:

```
(ns user)
(clean-ns 'myns) ; ❶
;; {}

(binding [*ns* (the-ns 'myns)] ; ❷
  (import '[java.util ArrayList HashMap])) ; ❸

(ns-imports 'myns) ; ❹
;; {HashMap java.util.HashMap
;;  ArrayList java.util.ArrayList}
```

❶Please see refer for the definition of the `clean-ns` function. It deletes all mappings for a given namespace.

❷ `import` does not accept an origin namespace as a parameter, so we temporarily swaps the current namespace with `binding`.

❸ `import` does not support options, but it can import many classes at once. The vector shows how to group several class names from the same package.

❹We can see that the requested classes have been added to the namespace mapping table.

16.11 FIND-NS AND ALL-NS

`find-ns` and `all-ns` are relatively simple functions to access the global namespace repository:

```
(pprint (all-ns)) ; ❶
;; (#object[clojure.lang.Namespace 0x20312893 "c
;;  #object[clojure.lang.Namespace 0x70eecdc2 "c

(ns-name (first (all-ns))) ; ❷
;; clojure.edn
```

❶ `all-ns` returns a lazy sequence of namespace objects.

❷We can use ns-name to extract the name as a symbol.

`find-ns` retrieves a namespace object given its name as a symbol, assuming the namespace exists:

```
(find-ns 'clojure.edn) ; ❶
;; #object[clojure.lang.Namespace 0x20312893 "cl

(find-ns 'no-ns) ; ❷
;; nil
```

❶ `find-ns` retrieves the namespace object corresponding to the given symbol, assuming the namespace was created at some point.

❷ If we try with a namespace not created yet, we get unsurprisingly a `nil`.

16.12 THE-NS, NS-NAME AND NAMESPACE

`the-ns`, `ns-name` and `namespace` are utility functions to help working with vars, symbols and namespaces. `the-ns` uses find-ns to retrieve a namespace or throw exception:

```
(the-ns 'notavail) ; ❶
;; Exception No namespace: notavail found

(the-ns 'clojure.edn) ; ❷
```

```
  ;; #object[clojure.lang.Namespace 0x20312893 "cl

  (the-ns *ns*) ; ❸
  ;; #object[clojure.lang.Namespace 0xcc62a3b "use
```

❶ `the-ns` throws exception if the namespace is not existent.

❷ If the namespace exists, `the-ns` returns the namespace object.

❸ `the-ns` also works with namespace objects directly, in that case they are just returned.

`ns-name` works on top of `the-ns` to access the name of the namespace as a symbol. This is useful if you have the namespace object and want to transform it into a symbol that can be used as a key:

```
  (ns com.package.myns)
  (ns-name *ns*) ; ❶
  ;; com.package.myns
```

❶ `ns-name` transforms a namespace object (the current namespace in this example) into the corresponding name as a symbol.

`namespace` retrieves the namespace part of namespace-aware objects such as symbols or keywords:

```
(ns user)

(namespace :notcreateyet/a) ; ❶
;; "notcreateyet"

(namespace ::a) ; ❷
;; user

(namespace 'alsosymbols/s) ; ❸
;; "alsosymbols"
```

❶We use `namespace` on a fully qualified keyword. We can see the namespace part of the keyword returned as a string.
❷Implicit full qualification to the current namespace is possible by using the double colon notation "::".
❸Finally, we can see `namespace` in action on a symbol.

16.13 META, WITH-META, VARY-META, ALTER-META! AND RESET-META!

`meta` and the other meta-related functions, are the functions dedicated to metadata management. Metadata (literally, "data about data") is an unobtrusive communication mechanism

(compared to functional arguments for example). Non-functional requirements are often a good candidate for metadata: debugging information, types, documentation etc. [240]. Metadata, by definition, do not impact equality semantic of the object they are attached to, which means that two equal objects still compare equal when their metadata are different.

Clojure uses metadata internally for many purposes. The metadata attached to a var object can drastically alter the way code is evaluated or executed. Metadata can also be used to store documentation, mark functions for testing, give type hints to the compiler and so on. We also used metadata in interesting ways throughout the book. Please check the following examples:

- We used metadata in defn to mark functions for benchmarking.
- Metadata was used to store database mapping in array-map
- sorted-set has an example of storing timestamps in metadata.

We can classify an object supporting metadata using the following categories:

1. **Read**: metadata are present on the object and we can read them with `meta`.
2. **Clone**: the object supports the creation of a new object of the same type and value of the old, but with different metadata using `with-meta` or `vary-meta`.
3. **Write**: the object supports thread-safe mutation of metadata (without cloning them into a new object) with `alter-meta!` or `reset-meta!`.

Many Clojure objects in the standard library are metadata-aware:

- Persistent data structures usually support reading and cloning: lists, vectors, sets or maps.
- Also lazy sequences support reading and cloning: ranges, cons, repeat, iterate etc.
- Reference types support mutable (and thread-safe) metadata: vars, atoms, refs, agents. namespaces also support mutable metadata, but they are not a type of reference. The metadata function to mutate metadata are `alter-meta!` and `reset-meta!`.
- Other object supports a mix of different kind of metadata support: symbols, functions, sub-vectors.

`meta` shows the metadata attached to an object, or `nil` otherwise:

```
(pprint (meta #'+)) ; ❶

;; {:added "1.2",
;;  :ns #object[clojure.lang.Namespace 0x1edb61b
;;  :name +,
;;  :file "clojure/core.clj",
;;  :inline-arities
;;  #object[clojure.core$_GT_1_QMARK_ 0x7b22ec89
;;  :column 1,
;;  :line 965,
;;  :arglists ([] [x] [x y] [x y & more]),
;;  :doc
;;  "Returns the sum of nums."
;;  :inline
;;  #object[clojure.core$nary_inline 0x790132f7

(meta 1) ; ❷
;; nil
```

❶ `meta` shows quite a rich set of metadata for the function "" from the standard library. Metadata are attached to the var object, not the symbol "".

❷ We can point `meta` to unsupported objects without having to catch an exception.

`with-meta` is pretty common with persistent data structures:

```
(def v (with-meta [1 2 3] {:initial-count 3})) ;

(meta (conj v 3 4 5)) ; ❷
{:initial-count 3}

(meta (with-meta (with-meta [1 2 3] {:a 1}) {:a
;; {:a 2}

(meta (into [] v)) ; ❹
;; nil
```

❶ `with-meta` stores the information about the initial count at the time the vector was created.

❷ The fact that new elements are added to the vector doesn't change its metadata. So we can always see how many items were in the vector originally.

❸ In this example we can see that `with-meta` completely replaces an existing set of metadata.

❹ The policy for metadata migration between data structures is different depending on the function that is applied. into creates a new data structure using a copy of the content of another one, but it doesn't migrate the original metadata.

`with-meta` replaces any existing set of metadata. If you want to preserve the existent metadata, or update them selectively, you can use `vary-meta` instead:

```
(def v (with-meta [1 2 3]
  {:initial-count 3 :last-modified #inst "1985-0

(meta v)
;; {:initial-count 3
;;  :last-modified #inst "1985-04-12T00:00:00.00

(def v (vary-meta (conj v 4) assoc :last-modifie

(meta v)
```

```
  ;; {:initial-count 3
  ;;  :last-modified #inst "1985-04-13T00:00:00.00
```

❶ We create a vector with 3 numbers and an last-modified time stamp as part of the metadata.

❷ `vary-meta` takes a function of the old metadata and any additional arguments. We can selectively update the last-modified timestamp and leave any other metadata intact.

`alter-meta!` behaves similarly to `vary-meta` but the mutation happens in place:

```
  (def counter ; ❶
    (atom 0
      :meta {:last-modified #inst "1985-04-12"}))

  (meta counter)
  ;; {:last-modified #inst "1985-04-12T00:00:00.00

  (alter-meta! ; ❷
    (do (swap! counter inc) counter)
    assoc :last-modified #inst "1985-04-13")

  (meta counter) ; ❸
  ;; {:last-modified #inst "1985-04-13T00:00:00.00
```

❶An atom is a type of reference supporting mutable metadata. The atom constructor takes a `:metadata` key at construction time to initialize metadata.
❷`alter-meta!` takes a function of the old metadata plus any additional arguments. We use `assoc` to selectively change the `:last-modified` key.
❸Calling `meta` on the atom correctly reports the change of time.

Finally, `reset-meta!` offers the possibility to completely replace the metadata map in case we are not interested in keeping the old:

```
(reset-meta! *ns* {:doc "The default user namesp
(meta *ns*)
;; {:doc "The default user namespace"}
```

❶Namespaces are not reference types, but they support mutable metadata. We use `reset-meta!` to specify some documentation about the namespace.

[240] Non-functional requirements of an application are all aspects that are not directly driven by business requirements: logging, tracing, performance, stability, robustness, etc. all have an impact on code, but they are not the main goal of the application.

16.14 SUMMARY

We reached the end of the chapter. The set of functions dedicated to var handling and namespace organization is quite rich and reflects the range of features that Clojure offers to produce modular code. One important aspect to remember is that thanks to `var` indirection, Clojure is able to offer a very dynamic programming environment.

17 EVALUATION

The functions illustrated in this section are at the core of the read, evaluation and print loop (the REPL for short). When you open the Clojure REPL, you're welcomed by a prompt waiting for a form to evaluate. The prompt is effectively a blocking call to the read function using the standard input as argument. After hitting the return key, eval analyzes the forms and emits the related bytecode that immediately executes. The result is printed on screen and the loop continues blocking on the next `read` call. Clojure uses the same `read` and `eval` functions extensively also outside the REPL.

The following is a summary of the functions illustrated in this chapter and their role during the evaluation cycle:

- read and read-string transform a valid textual form into a Clojure data structure. The data structure could be pure data or executable code. As other Lisps, Clojure treats data as code and code as data and `read` is the entry point for this transformation.

- eval produces an in-memory evaluation of a Clojure data structure. If the data structure represents a function call, `eval` returns the result of the evaluation.
- compile, load, load-file, load-reader and load-string operate on libraries instead of data structures. A library is a group of forms stored in a file [241]. Libraries normally evaluate into definitions. compile also produces one or more class files on disk.
- test and assert perform the evaluation of an expression to define the general validity of another expression or function.
- <edn,clojure.edn/read>> and clojure.edn/read-string are the equivalent read and read-string operations for EDN, the Extensible Data Notation. EDN is a subset of the Clojure syntax designed specifically for transporting data and load them safely.

17.1 READ AND READ-STRING

read

`read` takes an input stream and transforms its content into a Clojure data structure, following the rules of the Clojure syntax. The input stream should be an instance of

`java.io.PushbackReader` or subclasses. The easiest way to test the function is without arguments, forcing it to use the dynamic var `*in*` as input:

```
(instance? java.io.PushbackReader *in*) ; ❶
;; true

(def output (read)) ; ❷
;; (+ 1 2) ; ❸
;; #'user/output

output ; ❹
;; (+ 1 2)

(type output) ; ❺
;; clojure.lang.PersistentList
```

❶The dynamic var `*in*` is initialized to `clojure.lang.LineNumberingPushbackReader`, an instance of `java.io.PushbackReader`.

❷When called without arguments, `read` uses `*in*` as input.

❸The terminal outputs what we type before hitting return, while the prompt is blocking.

❹The result of `read` does not evaluate.

❺The type of the output is a list.

`read` supports a few options to control the reading process. For example, reader conditionals are turned off by default [242]. `:read-cond` can be set to either `:allow` or `:preserve` to allow reader conditionals and to preserve all branches respectively:

```
(def example ; ❶
  "#?(:clj (System/currentTimeMillis)
      :cljs (js/Console :log)
      :cljr (|Dictionary<Int32,String>|.)
      :default <anything you want>)")

(defn reader-from [s] ; ❷
  (-> (java.io.StringReader. s)
      (clojure.lang.LineNumberingPushbackReader.

(read (reader-from example)) ; ❸
;; RuntimeException Conditional read not allowed

(read {:read-cond :allow} (reader-from example))
;; (System/currentTimeMillis)

(read {:read-cond :preserve} (reader-from exampl
;; #?(:clj (System/currentTimeMillis)
;;     :cljs (js/Console :log)
```

```
  ;;    :cljr (|Dictionary<Int32 String>|.)
  ;;    :default <anything you want>)
```

❶The `#?` prefix is the read conditional macro and specifies a version of a form based on one of the available platforms.

❷ `reader-from` takes a string and create a suitable reader to use with `read`.

❸By default, `read` does not allow conditional macros and throws exception.

❹We ask `read` to enable reader conditionals with `:read-cond :allow`. The selected form is the one corresponding to the hosting platform of the Clojure runtime (in this case "clj" means Clojure on the JVM).

❺If we use `:preserve`, we prevent `read` to make any choice and return the form as is. This can be useful for any platform related post-processing.

We can add a new platform and using the `:features` option. The new platform becomes the default option instead of the one indicated by the `:default` key:

```
(def example ; ❶
  "#?(:cljs :cljs :my :my :default <missing>)")

(read {:read-cond :allow} (reader-from example))
```

```
  ;; <missing>

  (read {:read-cond :allow :features #{:my}} (read
  ;; :my
```

❶The example contains a keyword for a previously unregistered host ":my".

❷Assuming the current platform key is missing (which is `:clj` in this example), `read` would pick the `:default`.

❸Adding `:features #{:my}` to the options, allow `read` to return `:my` even when the current platform is different.

Using the `:eof` option, we can control the behavior of `read` in case it reaches the end of file (abbreviated "eof") before reading a form:

```
  (read (reader-from ";; a comment")) ; ❶
  ;; RuntimeException EOF while reading

  (read {:eof nil} (reader-from ";; a comment")) ;
  ;; nil

  (read (reader-from ";; a comment") false nil) ;
  ;; nil
```

❶A comment skips to the end of the stream without reading a form. `read` throws an exception in this case.
❷If we prefer `nil`, we can indicate so using the `:eof` option.
❸The same option is also available as argument.

Beside the options we've seen so far, the reader also complies with the `*read-eval*` dynamic var. `*read-eval*` controls the behavior of the reader when parsing the read-eval macro `#=`. When present in front of a form, the reader first parses the form as usual and then invokes eval on it:

```
(read (reader-from "#=(+ 1 2)")) ; ❶
;; 3
```

❶Instead of the expected list of a symbol and two numbers `('+ 1 2)`, the reader also evaluates the form.

The read-eval macro is enabled by default, as it's fundamental feature contributing to the dynamicity of the language. But there are security implications, for example if some code tries to execute malicious instructions:

```
(read (reader-from "(java.lang.System/exit 0)"))
;; (java.lang.System/exit 0)
```

```
;; WARNING: the JVM will exit.
(read (reader-from "#=(java.lang.System/exit 0)"
```

❶ The `System/exit` call exits the running JVM. But `read` just load the form into a list.

❷ The read-eval macro in front of the form forces evaluation of the form, which exits the JVM.

To prevent evaluation, wrap `read` in a binding context that sets `*read-eval*` to `false`:

```
(binding [*read-eval* false] ; ❶
  (read (reader-from "#=(java.lang.System/exit 0
;; RuntimeException EvalReader not allowed when
```

❶ `read` throws exception if a read-eval macro is present in the form and `*read-eval*` is set to `false`.

In case you wanted to prevent reading altogether (for example to prevent large data structures to load into memory via `read`), you could use `:unknown` instead of `false`:

```
(binding [*read-eval* :unknown] ; ❶
  (read (reader-from "(+ 1 2)")))
```

```
  ;; RuntimeException Reading disallowed - *read-e
```

❶Setting `*read-eval*` to `:unknown` prevents reading completely.

Note

There is also another feature influencing how data are read: tagged literals. This is a way to extend the set of available reader syntax macros beyond the ones that are installed by default. Please check the section on tagged literals to know more.

---

read-string

The examples so far have created a `clojure.lang.LineNumberingPushbackReader` on top of a string to simulate the content of a file (or other input stream). But if you are dealing directly with a string, `read-string` works the same as `read`:

```
(read-string "(+ 1 2)") ; ❶
;; (+ 1 2)
```

```
  (read-string {:eof "nothing to read"} "") ; ❷
  ;; "nothing to read"
```

❶ `read-string` works exactly like `read` using a string to create a reader object similarly to what we did for the example above.

❷ We can use the end of file option to prevent exception reading an empty string.

`read-string` supports the same options as `read`, so the description of the function is also the same (except for taking a string as input).

[241] Please review the introduction to "Var and Namespaces" for an explanation about the concept of library

[242] Reader Conditionals is a relatively new feature to support Clojure implementations on other platforms, in particular ClojureScript and ClojureCLR. See clojure.org/guides/reader_conditionals for more information

17.2 EVAL

`eval` takes a Clojure data structure and returns its evaluated form. If the object is a native sequence (it's not enough for the object to be just sequential), `eval` treats the first element as a function and invokes the function on the rest of the sequence:

```
(eval '(+ 1 2)) ; ❶
;; 3

(eval [+ 1 2]) ; ❷
;; [#object[clojure.core$_PLUS "clojure.core$_PL
```

❶A list is a native sequence. `eval` assumes the first element is a function and the rest are the arguments. In this case `eval` returns the evaluation of the function on the arguments.

❷A vector is sequential (it produces a sequence when calling seq on it) but is not a sequence itself. The `vector` evaluates to itself but the symbol "+" is looked up in the mapping table of the current namespace and replaced with the actual function object.

To evaluate multiple forms, we need to enclose them in a do block:

```
(eval '(do (println "eval-ed") (+ 1 2))) ; ❶
;; eval-ed
;; 3
```

❶In case of multiple forms wrapped in a do block, `eval` proceeds to evaluate all of them.

The separation between `read` and `eval` is essential tool in metaprogramming. Metaprogramming is the aspect of a language that enables a program manipulate another program. During normal execution, activities like parsing and evaluation are orchestrated by Clojure, but when using `read` and `eval` explicitly, we have an opportunity to interpret a program differently. This is particularly useful to design Domain Specific Languages (or DSL). For example, the following is a simple DSL to describe a traffic light system. The rules in this system are not valid Clojure syntax, but we can use `read-string` to implement an intermediate translation layer:

```
(require '[clojure.string :refer [split-lines]])

(def rules ; ❶
  "If the light is red, you should stop
   If the light is green, you can cross
   If the light is orange, it depends")

(defmacro If [light & args] ; ❷
  (let [[_ _ op v & action] args]
    `(when (= '~light '~v) '~action)))

(defn parenthesize [s] ; ❸
  (->> s
    split-lines
```

```
    (remove empty?)
    (map #(str "(" % ")"))))

(defn traffic-light [color rules] ; ❹
  (->> rules
    parenthesize
    (map read-string)
    (map #(list* (first %) color (rest %))) ; ❺
    (some eval)))

(traffic-light 'red rules) ; ❻
;; (you should stop)

(traffic-light 'green rules)
;; (you can cross)

(traffic-light 'orange rules)
;; (it depends)
```

❶ `rules` contains simple facts about the meaning of the different colors in a traffic light. Our goal is to evaluate these facts to produce the program behavior.

❷ We transform uppercase `If` into a macro and the rest of the sentence into arguments we can manipulate. `If` has to be a macro so the content of the sentence does not evaluate. We then destructure the input and use when to verify if the "light" argument is the same color as that described by the sentence.

❸ `parenthesize` is a small transformation of the input string so that each sentence is wrapped in a set of parenthesis. The sentence should appear to `eval` as a list, so it triggers a call to the `If` macro.

❹ `traffic-light` orchestrates the process: rules are first wrapped in parenthesis, then transformed into lists using read-string.

❺ Now that rules are encoded as lists, we inject the missing "light" parameter. Note that at this point we are manipulating lists, not processing strings. The list is ready for `eval` which invokes the `If` macro and returns the answer.

❻ These are a few example with different colors to test the answers.

17.3 LOAD, LOAD-FILE, LOAD-READER

AND LOAD-STRING

`load`, `load-file`, `load-reader` and `load-string` are dedicated to the evaluation of Clojure code from libraries. Libraries are files containing ordinary Clojure code, but they have to declare a namespace and obey specific conventions:

- The file should be available in the classpath of the current process.
- The name of the namespace should reflect the relative path of the file it is declared in (replacing "/" with dots ".").

load

`load` takes a string file path as argument. If the path begins with "/" (forward slash), the path starts at the root of the classpath. In all other cases, the path starts from the location of the current namespace:

```
(ns user)
(clojure.zip/vector-zip []) ; ❶
;; ClassNotFoundException clojure.zip

(load "zip") ; ❷
;; FileNotFoundException Could not locate zip__i
```

```
(ns clojure.set) ; ❸
(load "zip")
;; nil

(clojure.zip/vector-zip []) ; ❹
;; [[] nil]
```

❶"clojure/zip.clj" is a library present in every Clojure distribution, but we cannot use it without loading it first.

❷If we try to load the library "zip" without specifying its absolute location on the classpath, Clojure tries to load it from the same location of the "user" namespace. In this location there is no "zip" library and `load` fails with an error.

❸After setting the current namespace to `clojure.set`, we are able to load the "zip" library.

❹We can now use `vector-zip`.

`load` supports an useful verbose mode that prints every loaded file while traversing the dependency tree. Use the `*loading-verbosely*` dynamic var to activate this feature:

```
(ns user)

(binding [clojure.core/*loading-verbosely* true]
```

```
    (load "clojure.reflect"))

  ;; (clojure.core/load "/clojure.reflect")
  ;; (clojure.core/in-ns 'clojure.reflect)
  ;; (clojure.core/alias 'set 'clojure.set)
  ;; (clojure.core/load "/clojure/reflect/java")
  ;; (clojure.core/in-ns 'clojure.reflect)
  ;; (clojure.core/alias 'set 'clojure.set)
  ;; (clojure.core/in-ns 'clojure.reflect)
  ;; (clojure.core/alias 'str 'clojure.string)
```

❶ `load` supports verbose output to print all dependencies loaded during traversal, including additional information on created alias and refer.

In the previous trace we can see what additional library clojure.reflect depends on. We can also see alias definitions created for each library.

load-file

Compared to `load`, `load-file` evaluates files not necessarily available from the Java classpath. `load-file` (like `load`) uses the presence of the forward slash "/" to set the initial location of the path. We could use `load-file` to run a simple Clojure script (a program that runs and terminates from a single file):

```
(spit "source.clj" ; ❶
  "(ns ns1)
   (def a 1)
   (def b 2)
   (println \"a + b =\" (+ a b))")

(load-file "source.clj") ; ❷
;; a + b = 3
```

❶ `source.clj` contains a simple Clojure program that defines two vars, "a" and "b" and sums them up.

❷ `load-file` without a forward slash uses the relative location starting from where the current process was started (the content of the "user.dir" Java property).

load-string

`load-string` is substantially equivalent to read-string followed by eval:

```
(= (eval (read-string "(+ 1 1)")) ; ❶
   (load-string "(+ 1 1)"))
;; true
```

❶ `load-string` performs both parsing and evaluation of a string. We can see it produces the same result of `read-string` followed by `eval`.

However, there are a few differences which make `load-string` more suitable to load the content of a file (as string):

- `load-string` does not support options, while `read-string` can be instructed about several aspects of the reading process (see read-string for a list of the supported options).
- `load-string` does not require wrapping multiple forms in a do block. `read-string` on the other hand, reads a single form only.
- `load-string` keeps track of line numbering in vars metadata while `read-string` does not.

Let's investigate the last aspect with an example:

```
(ns user) ; ❶
(def code "(do (def a 1)\n(def b 2)\n(def c 3))")

(ns code1) ; ❷
(load-string user/code)
(map (comp :line meta) [#'a #'b #'c])
;; (1 2 3)
```

```
(ns code2) ; ❸
(eval (read-string user/code))
(map (comp :line meta) [#'a #'b #'c])
;; (1 1 1)
```

❶ `code` is a string containing var definitions separated by newline, forming the equivalent of a file with 3 lines.

❷A new namespace "code1" loads and evaluate the content of `user/code`. Metadata on the var definitions contain the expected line numbers.

❸Another namespace "code2" loads and evaluate `user/code` using `eval` and `read-string` combination. Every var definition now loads from the same line 1.

The example shows that `read-string` ignores new-line terminations and considers all expressions as part of a single first line. `load-string`, on the other hand, correctly identifies line numbers on each expression.

load-reader

`load-reader` behaves exactly like `load-string` or `load-file`, but it requires a `java.io.Reader` input type. `load-reader` is useful to control the specific type of reader to use.

17.4 COMPILE

`compile` accepts a library path as a symbol and performs parsing and evaluation similarly to load (we talked about the definition of "library" in load and the introduction to the vars chapter). `compile` additionally dumps the generated bytecode to disk:

```
(spit (str *compile-path* "/source.clj") ; ❶
  "(ns source)
   (defn plus [x y] (+ x y))")

(compile 'source) ; ❷
;; source
```

❶spit creates a simple Clojure library called "source" on disk. The file is saved to the `*compile-path*` which depending on your environment could be pointing at "target/classes" or similar folder. The important aspect of `*compile-path*` is that is part of the Java classpath.

❷We call `compile` on the symbol representing the namespace we want to compile. `compile` returns the symbol of the class that was just created, in this case "source".

`compile` produces a few class files on disk for "source.clj": you can look at them by opening the folder corresponding to `*compile-path*` (by default "classes"). Some of them initialize the namespace and associated vars, including static loading to register the namespace in the global repository and related vars in the namespace mapping table. Another class file implements the "plus" function (this should be named "source$plus.class"). `compile` is going to produce additional class files for each function (including anonymous) from the source library. The availability of the generated classes on the classpath is sufficient to make sure that namespaces and vars declared therein are available after bootstrap.

Compilation with `compile` (or alternatively by invoking the main method of `clojure.lang.Compile` from the command line) can be used to perform Ahead Of Time compilation (or

simply AOT) which caches the result of Clojure compilation to disk. AOT compilation has several advantages:

- Eliminates the need for distributing Clojure sources. Once class files are generated with `compile` they take precedence over the sources, if they are both present. Class files are also amenable for "obfuscation", the process by which class files are made uneasy to read or decompile.
- Reduces the application startup time, which is useful especially for large applications.
- Makes the application available to other languages on the JVM (including gen-class class declarations).

However, AOT cannot be enabled by default on all environments. The reason is that AOT compilation also removes some flexibility:

- Generated classes become linked to some specific Clojure-JVM version. This might have negative consequences especially for Clojure libraries which are designed to run as part of other applications.

- AOT adds complexity to the build process. Good AOT tooling would need to understand when classes need compilation because of changes in the related sources, especially for larger projects.
- AOT also adds complexity to the testing process. Unit tests might be passing but the application could still suffer from unexpected bugs. This is could be the result of unit tests not necessarily forcing evaluation of some parts of the code. AOT is commonly configured to compile all files, even those that are not impacted by unit tests. A good testing approach should also exercise the application after AOT compilation.

The choice of compiling a Clojure application ahead of time is therefore a trade-off between the additional complexity introduced by AOT and the advantages it gives in distributing applications. Normally, Clojure standalone applications are AOT compiled along with all their transitive dependencies (this is called an "Uberjar") so the application becomes very easy to install and performs well. Clojure libraries, on the other hand,

are normally not AOT compiled to let the hosting application the freedom to decide the best compilation strategy.

17.5 TEST AND ASSERT

`test` and `assert` perform evaluation of an expression finalized to verify its correctness. `test` is the most basic form of testing available in Clojure and is quite limited, especially considering that Clojure ships with the clojure.test namespace which contains a full fledged testing framework.

There are a few simple use cases for `test`, for example when we want to express expectations along with the code we are writing. The following function calculates the square root of a number and it has been added a few tests to verify its behavior:

```
(defn sqrt
  {:test ; ❶
   #(when-not (== (sqrt 4) 2.)
      (throw (RuntimeException. "sqrt(4) should
  [x]
  (loop [guess 1.]
    (if (> (Math/abs (- (* guess guess) x)) 1e-8
      (recur (/ (+ (/ x guess) guess) 2.))
      guess)))
```

```
  (test #'sqrt) ; ❷
  ;; RuntimeException sqrt(4) should be 2
```

❶defn accepts metadata in several locations. We need to provide a `:test` key with a function of no argument as value. The most readable location in this case is before the arguments (also depending on the length of the testing function). A second option would be at the end of the function body.

❷We can prove the algorithm works as expected by calling `test` on the var object that contains the function. In this case we encounter a surprise, as our function guesses an approximate square root of 4 as 2.0 using the Newton method. The result is very close to 2.0 but not exactly 2.

As you can see from the example, `test` requires to throw an exception to signal the presence of a problem. To reduce the amount of code necessary to create and throw the exception, we can use `assert`. `assert` is a macro that evaluates the given expression and throws exception in case the expression is `false` or `nil`:

```
  (assert (= 1 (+ 3 3)) "It should be 6") ; ❶
  ;; AssertionError Assert failed: It should be 6
```

```
  ;; (= 1 (+ 3 3))
```

❶ A simple example to showcase how `assert` works.

Here's how we can use `assert` in the previous `sqrt` function:

```
(defn sqrt
  {:test #(assert (== (sqrt 4) 2.) "sqrt(4) shou
  [x]
  (loop [guess 1.]
    (if (> (Math/abs (- (* guess guess) x)) 1e-8
      (recur (/ (+ (/ x guess) guess) 2.))
      guess)))

(test #'sqrt)
;; AssertionError Assert failed: sqrt(4) should
;; (== (sqrt 4) 2.0)
```

❶ `assert` removes the need to create and throw an exception in case of an error condition.

We could use test metadata on a function to document the presence of important properties. If a function is expected to expose a specific behavior and that information should be immediately apparent to the reader of the function, we could

leave a "comment" in the form of `test` metadata. Occasional run of the tests after any change should be enough to guarantee the preconditions have been met. However, if `test` metadata becomes too complicated, you should consider splitting tests away from the function. For a more scalable approach to automated testing see clojure.test.

17.6 CLOJURE.EDN/READ AND CLOJURE.EDN/READ-STRING

EDN, the Extensible Data Notation, is a subset of the Clojure syntax (with additional changes) designed for data exchange with other languages [243]. One reason to restrict EDN to a subset of Clojure, is that Clojure contains specialized reader macros that would not make immediate sense for use in other languages. A few example are: `()` (anonymous function reader), `@` (var dereference reader), `=` (read-eval macro), syntax quote (back-tick), `?#` (reader conditionals), etc.

`clojure.edn/read` and `clojure.edn/read-string` work similarly to their `clojure.core` equivalents: `clojure.edn/read` takes an instance of `java.io.PushbackReader` as input while `clojure.edn/read-string` takes a simple string. We are going to use `clojure.edn/read-string` for the examples in

this section as `clojure.edn/read` is functionally equivalent. `clojure.edn/read-string` restricts the use of some reader macros by throwing exception:

```
(require '[clojure.edn :as edn]) ; ❶
(alias 'core 'clojure.core)

(core/read-string "@#'+") ; ❷
;; (clojure.core/deref (var +))

(edn/read-string "@#'+") ; ❸
;; RuntimeException Invalid leading character: @
```

❶The `clojure.edn` namespace is not available by default and requires explicit require. Just for this example and following ones, we are going to add an alias to `clojure.core` for clarity.

❷The `"@"` sign in front of an expression is equivalent to call the var function and is interpreted normally by the core reader.

❸EDN does not have such macro (along with several others) and throws exception.

Another difference between the EDN reader and `clojure.core` are the supported options. EDN ignores

`:read-cond` or `:features` (see read) because reader conditionals are not supported. But EDN reading adds the following new options: `:readers` and `:default`.

The `:readers` option declares additional tagged literals as a map of tag names to tag implementations. If a key already exists for a given tag, then the new tag overrides the default:

```
(edn/read-string ; ❶
  "#point [1 2]")
;; RuntimeException No reader function for tag p

(edn/read-string ; ❷
  {:readers {'point identity}}
  "#point [1 2]")
;; [1 2]

(edn/read-string ; ❸
  {:readers {'inst (constantly "override")}}
  "#inst \"2017-08-23T10:22:22.000-00:00\"")
;; "override"
```

❶ `#point` is a tagged literal unknown to the system.

❷ We use the `:readers` option to register a `point` key and identity to handle it (it could of course implement more complex behavior).

❸ "#inst" is a default tagged literal. This instruction, instead of parsing "#inst" as a `java.uti.Date`, uses an implementation that always return the string "override".

The `:default` option creates a default implementation if a tagged literal is not found in default-data-readers or `*data-readers*`:

```
(edn/read-string ; ❶
  {:default #(format "[Tag '%s', Value %s]" %1 %
  "[\"There is no tag for \" #point [1 2] \"or\"

;; ["There is no tag for "
;;  "[Tag 'point', Value [1 2]]"
;;  "or"
;;  "[Tag 'line', Value [[1 2] [3 4]]]"]
```

❶ Trying to read a "#point" or "#line" tags would result in exception. We can handle all missing tags with the `:default` options. `:default` takes a function of two arguments, the tag name and its value.

The following section describes tagged literals and their features in more details.

[243] The EDN rationale and general design are available at github.com/edn-format/edn

17.7 TAGGED-LITERAL AND TAGGED-LITERAL?

`tagged-literal` is an helper function that creates a new `clojure.lang.TaggedLiteral` object given a "tag" and a "form" parameters:

```
(tagged-literal 'point [1 2]) ; ❶
;; #point [1 2]

(:tag (tagged-literal 'point [1 2])) ; ❷
;; point
(:form (tagged-literal 'point [1 2]))
;; [1 2]
```

❶ `tagged-literal` creates a new `TaggedLiteral` object. Clojure knows how to print them nicely.

❷ We can also access the `:tag` and `:form` keys.

Tagged literal objects can be used whenever the reader requires one to parse a custom tag instead of a custom function, sparing some typing. There are currently the following options:

```
(require '[clojure.edn :as edn])

(edn/read-string
  {:default tagged-literal} ; ❶
  "[\"There is no tag for \" #point [1 2] \"or\"

;; ["There is no tag for " #point [1 2] "or" #li

(binding  [*default-data-reader-fn* tagged-liter
  (read-string "[\"There is no tag for \" #point

;; ["There is no tag for " #point [1 2] "or" #li
```

❶We've seen this example in edn/read-string, where we've used an anonymous function of 2 arguments. `tagged-literal` receives the unregistered tag `point` and related form as arguments.

❷read and read-string don't support a `:default` option. However, the dynamic var `*default-data-reader-fn*` assumes the same meaning.

`tagged-literal?` verifies if an object is a tagged literal instance:

```
(tagged-literal? (tagged-literal 'tag :form)) ; ❶
;; true
```

❶ `tagged-literal?` returns `true` when the given object is an instance of `clojure.lang.TaggedLiteral`.

17.8 DEFAULT-DATA-READERS

`default-data-readers` retrieves the default data readers installed with Clojure. Currently, Clojure ships with the following data readers:

```
default-data-readers ; ❶
;; {inst #'clojure.instant/read-instant-date,
;;  uuid #'clojure.uuid/default-uuid-reader}
```

❶ `default-data-readers` contains a mapping between the name of the tag as a symbol and the function of one argument that is going to receive the form as read by the reader.

Data readers have been introduced along with EDN to allow easy "round-tripping" of data to and from strings:

```
(def date (edn/read-string "#inst \"2017-08-23T1(

(= date (edn/read-string (pr-str date))) ; ❷
;; true
```

❶"#inst" is a default tagged literal instructed to parse the string that follows as a `java.util.Date` object.

❷Tagged literals are designed so they write and read strings, enabling round-trip data exchange over a network (or file). We can verify that transforming a date to a string and reading that string back produces the same object.

The dynamic var `*data-readers*` uses the same format as `default-data-reader` and when bound, it allows to add or modify the default readers. Here for example we teach Clojure how to serialize and de-serialize `java.net.URL` objects:

```
(import '[java.net URL] [java.io File])

(defmethod print-method URL [url writer] ; ❶
  (doto writer
    (.write "#url ")
    (.write "\"")
    (.write (.toString url))
    (.write "\"")))
```

```
(-> "/etc/hosts" File. .toURL pr-str) ; ❷
;; "#url \"file:/etc/hosts\""

(binding [*data-readers* {'url #(URL. %)}] ; ❸
  (-> "/etc/hosts" File. .toURL pr-str read-stri
```

❶We extend print-method to enable `pr-str` to print URL objects with a tag that the reader can understand.

❷This shows how the URL object will look like once it is transformed into a string.

❸Going full circle: printing the URL object into a string and read it back into an object after binding `*data-readers*` to a proper constructor call.

17.9 READER-CONDITIONAL AND READER-CONDITIONAL?

`reader-conditional` is a simple helper function (similar to tagged-literal) that creates a new `clojure.lang.ReaderConditional` object:

```
(reader-conditional '(:clj :code) false) ; ❶
;; #?(:clj :code)
```

```
  (reader-conditional '(:clj [1 2 3]) true) ; ❷
  ;; #?@(:clj [1 2 3])
```

❶ `reader-conditional` creates a new `ReaderConditional` object. Clojure has specific printing behavior in place to handle reader conditions.

❷ The second boolean parameter determines if this is a "spliced" condition. Note the conditional now prints with an added "@" at sign.

`reader-conditional` is useful to assemble reader conditional options without having to format the corresponding string manually. When the form in a reader condition is a collection, we have the option to interpret it as a whole (not spliced) or to take each items in turn, removing the set of wrapping parenthesis (splicing). Here's an example that highlights the different interpretation:

```
  (read-string {:read-cond :allow} "(list #?(:clj
  ;; (list [1 2 3])

  (read-string {:read-cond :allow} "(list #?@(:clj
  ;; (list 1 2 3)
```

❶When reading a reader conditional without splicing, the reader interpret the corresponding form literally, in this case the vector `[1 2 3]`.
❷Splicing unwrap the form (assuming it's inside a collection) and retrieves just the elements.

`reader-conditional?` performs a check to verify if the given argument is a reader conditional. This can be useful in `:preserve` mode while analyzing the output of the reader:

```
(def parse (read-string {:read-cond :preserve} "
(reader-conditional? parse) ; ❶
;; true
```

❶ `reader-conditional?` returns `true` when the given argument is an instance of `clojure.lang.ReaderConditional` object.

After evaluating a reader conditional instance, we can access its form and splicing conditions with:

```
(def parse (read-string {:read-cond :preserve} "

(:form parse) ; ❶
;; (:clj [1 2 3])
```

```
  (:splicing? parse) ; ❷
  ;; false
```

❶A reader conditional object offers a `:form` key to access the matching form, including the platform.

❷Another `:splicing?` key accesses the splicing status, in this case `false` (there is no "@" sign).

17.10 SUMMARY

This concludes the chapter about Clojure reading and evaluation. `read-string` and `eval` are part of the Clojure API into the reader and compiler, respetively. By reading text with `read-string`, we transform the textual representation of a Clojure program into a data structure. By evaluating the data structure with `eval`, we compile and potentially run a Clojure program. The other functions in this chapter are related to either reading or compiling Clojure, offering different ways to control this process.

18 FORMATTING AND PRINTING

18.1 FORMAT, PRINTF AND CL-FORMAT

`format`, `printf` and `cl-format` are functions dedicated to string formatting. `format` is a wrapper around Java's `String::format` method, which is inspired by the venerable `printf` function from the C language. In Clojure `printf` is a small function wrapping println with `format`.

`cl-format` is instead a port of Common Lisp's `format` function, formerly an external package called XP [244].

We've seen both `format` and `cl-format` in action many times throughout the book. Here's a few examples for review:

- In memoize, we used `format` to print the cache hit (or miss) information.
- In rand we used `format` to print a text-based progress indicator.
- In vec we used `format` to render a simple JSON snippet.

- In Chapter 1, when describing how to improve printing of decimal values, we used `cl-format` in the XML example.

`format` has a rich set of formatting directives. The reader is invited to check the `java.util.Formatter` API reference for the full details [245], but here's a group of useful examples:

```
(format "%3d" 1) ;; "  1" ; ❶
(format "%03d" 1) ;; "001" ; ❷
(format "%.2f" 10.3456) ;; "10.35" ; ❸
(format "%10s", "Clojure") ;; "   Clojure" ; ❹
(format "%-10s", "Clojure") ;; "Clojure   " ; ❺
(format "%-11.11s" "truncatefixedsize") ;; "trun
(format "%tT" (java.util.Calendar/getInstance))
```

❶Left padding a number with spaces for a total length of 3.

❷Left padding a number with zeroes for a total length of 3.

❸Rounding a decimal number to 2 digits.

❹Left padding a string with spaces.

❺Right padding a string with spaces.

❻Truncating a string if too long, right padding if too short.

❼Formatting the current local time.

`cl-format` is more sophisticated and supports additional features at the price of an increased complexity. The following is a group of basic examples [246]:

```
(require '[clojure.pprint :refer [cl-format]]) ;

(cl-format nil "~:d" 1000000) ;; "1,000,000" ; ❷
(cl-format nil "~b" 10) ;; "1010" ; ❸
(cl-format nil "Anno Domini ~@r" 25) ;; "Anno Do

(cl-format nil "~r" 158) ;; "one hundred fifty-e
(cl-format nil "~:r and ~:r" 1 2) ;; "first and
(cl-format nil "~r banana~:p" 1) ;; "one banana"
(cl-format nil "~r banana~:p" 2) ;; "two bananas
```

❶ `cl-format` is declared in the `clojure.pprint` namespace.

❷Large number formatting with comma separators.

❸Printing in binary format. `cl-format` also supports octal and hexadecimal.

❹Roman numerals conversion.

❺Number conversion into words.

❻Number conversion into ordinals.

❼An example of "double matching" of the input which is used first to translated into an ordinal and then to trigger the pluralized form.

An interesting `cl-format` feature is conditional formatting where the final output depends on the size of the input. Conditional formatting and multiple matches provide `cl-format` with a way to produce grammatically correct english:

```
(require '[clojure.pprint :refer [cl-format]])

(def num-sentence "~#[nope~;~a~;~a and ~a~:;~a, ~

(apply cl-format nil num-sentence [1 2]) ;; "1 an
(apply cl-format nil num-sentence [1 2 3]) ;; "1,
(apply cl-format nil num-sentence [1 2 3 4]) ;;

(def pluralize "I see ~[no~:;~:*~r~] fish~:*~[es~

(cl-format nil pluralize 0) ;; "I see no fishes."
(cl-format nil pluralize 1) ;; "I see one fish."
(cl-format nil pluralize 100) ;; "I see one hund
```

❶We store the format directive as a local variable "num-sentence" that we can reuse in multiple examples.

❷"pluralize" contains a simple formatting directive to transform words to plural, including an exception to the rule of appending "s" at the end of the word which doesn't work with "fish" (for instance).

In the following example we see `cl-format` in action to wrap text to a specific line size [247]:

```
(def paragraph
  ["This" "sentence" "is" "too" "long" "for" "a"
  "and" "should" "appear" "in" "multiple" "lines
  "than" "20" "characters" "each" "."])

(println (cl-format nil "~{~<~%~1,20:;~A~> ~}" pa
;; This sentence is too
;; long for a small
;; screen and should
;; appear in multiple
;; lines no longer than
;; 20 characters each.
```

❶We use `cl-format` to produce the arranging of the words in a sentence so they never go beyond 20 characters in length. Such functionality is non-trivial, as is the format directive that requires some learning curve.

As you have seen `cl-format` is quite powerful, but it takes some time to use proficiently: there are many directives and their syntax can be difficult to read. Luckily, most of the literature about the `format` function in Lisp can be easily adapted to work with `cl-format`: one example is chapter 18

of "Practical Common Lisp" by Peter Siebel [248]. However, if a directive becomes too complicated, the user should consider longer but explicit alternatives, like using sequential processing.

[244] The XP pretty printing library description is available in detail at dspace.mit.edu/bitstream/handle/1721.1/6503/AIM-1102.pdf. The paper also contains historical notes linking our Clojure `cl-format` all the way back to MacLisp original print system in 1977.

[245] docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Formatter.html

[246] The `clojure.pprint.cl-format` namespace source is well documented and worth a read: github.com/clojure/clojure/blob/master/src/clj/clojure/pprint/cl_format.clj

[247] This example is adapted from the following paper: cybertiggyr.com/fmt/fmt.pdf

[248] www.gigamonkeys.com/book/a-few-format-recipes.html

18.2 PR, PRN, PR-STR, PRN-STR, PRINT, PRINTLN, PRINT-STR, PRINTLN-STR

Clojure has a rich set of printing functions and this section of

the book presents those declared in the `clojure.core` namespace. Functions starting with the full prefix "print" (`print`, `println`, `print-str` and `println-str`) produce output optimized for human consumption. Those starting with "pr" only (`pr`, `prn`, `pr-str`, `prn-str`) produce a slightly different output which is designed to be readable back by read-string, enabling basic Clojure serialization [249].

The following example shows the typical difference between `pr` and `print` (and related functions of the same group):

```
(pr "a" 'a \a) ; ❶
;; "a" a \a
;; nil

(print "a" 'a \a) ; ❷
;; a a a
;; nil
```

❶ Both functions end with a `nil` which is an artifact of the REPL printing the result of the last evaluated expression. Both `pr` and `print` are side effecting functions returning `nil` and `nil` is printed last. `pr` distinguishes between strings, symbols and characters by printing them with additional double quotes, removing the single quote and printing back-slash `\` respectively.

❷ `println` prints the same objects the same way, removing any quoted decoration. Given the result from `print` we wouldn't be able to distinguish their original type.

What makes `pr` output suitable to read back into Clojure is the presence of specific quoting that helps the reader interpreting the character stream. The difference is more visible when printing generic objects, such as Java maps:

```
(import 'java.util.HashMap)

(def java-map (HashMap. {:a "1" :b nil}))

(prn java-map) ; ❶
;; {:a "1", :b nil}

(println java-map) ; ❷
;; #object[java.util.HashMap 0x1ffddcad {:a=1, :
```

❶ `prn` appends a new line to standard output after printing its arguments. The evaluation of the form is `nil` and the REPL prints it after the new line: in this example the `nil` was omitted for clarity.

❷ The human readable output for `java-map` contains the Java object hash in hexadecimal, the name of the class and the content of the map. As you can see `nil` is printed as `null`.

The four functions ending with `*-str` return their content as a string instead of side effecting to the current value of `*out*` (a dynamic variable pointing at standard output by default):

```
(def data {:a [1 2 3]
           :b '(:a :b :c)
           :c {"a" 1 "b" 2}})

(pr-str data) ; ❶
;; "{:a [1 2 3], :b (:a :b :c), :c {\"a\" 1, \"b

(prn-str data) ; ❷
;; "{:a [1 2 3], :b (:a :b :c), :c {\"a\" 1, \"b

(print-str data) ; ❸
;; "{:a [1 2 3], :b (:a :b :c), :c {a 1, b 2}}"
```

```
  (println-str data) ; ❹
  ;; "{:a [1 2 3], :b (:a :b :c), :c {a 1, b 2}}\n
```

❶ `pr-str` is like `pr` but the result is the printout of the arguments as a string instead of printing to standard output (the default value of `*out*`).

❷ `prn-str` just adds a new line to the output of `pr-str`.

❸ `print-str` is the same as `pr-str` but some objects print differently, like strings for example (see the double quotes surrounding them in the two examples).

❹ `println-str` appends an additional new line at the end of the string that becomes visible as `\n`.

In the case of `pr`, `prn`, `print` and `println`, the value of `*out*` can be bound with binding to output to an alternate Java `OutputStream` or `Writer` (Java interfaces dedicated to byte-oriented and character-oriented output, respectively). In the next example we redirect `print` output to a file:

```
  (require '[clojure.java.io :as io]) ; ❶

  (with-open [w (io/writer "/tmp/range.txt")] ; ❷
    (binding [*out* w] ; ❸
      (print (range 100000)))) ; ❹
```

❶ `clojure.java.io` is a namespace part of the standard library. It contains functions wrapping the Java IO framework.

❷ `io/writer` returns a `BufferedWriter` object "w". `with-open` makes sure that the buffer is closed after evaluating the body. Buffering accumulates characters in memory before writing to disk, limiting the number of transmissions to physical disk (an expensive operation).

❸ `binding` temporarily swaps the current value of `*out*` with the newly created writer.

❹ `print` is instructed to output a long range of numbers. Note that range creates the 100,000 numbers lazily.

Printing to a file using a `java.io.BufferedWriter` (which is the default object type returned by `clojure.java.io/writer`) enables more efficient printing of very large objects, for example those generated from a lazy sequence. The output of the lazy sequence never exists in memory at once, because as soon as new elements are generated and printed they are immediately garbage collected.

[249] Serialization is a mechanism for persisting arbitrarily nested data structures or objects (in case of Java) to disk or network. The name derives from the fact that what appears normally as a graph of data in memory is flattened to a single row of bytes to be sent on the wire. The graph is then restored once the bytes are de-serialized back into their original shape.

18.3 PPRINT, PP, WRITE AND PRINT-TABLE

Note

please also refers to clojure.pprint dynamic variables, a section dedicated to describe the many configuration options available in `clojure.pprint`.

The functions in this section are dedicate to "pretty printing", a feature that enhances readability beyond the basic printing primitives (see pr and print). Pretty printing is readily available at the REPL using the function `pprint` or `pp` (no require needed, as the REPL imports them from `clojure.pprint` implicitly):

```
(def data {:a ["red" "blue" "green"]
           :b '(:north :south :east :west)
           :c {"x-axis" 1 "y-axis" 2}})
```

```
data ; ❶
;; {:a ["red" "blue" "green"], :b (:north :south

(pp) ; ❷
;; {:a ["red" "blue" "green"],
;;  :b (:north :south :east :west),
;;  :c {"x-axis" 1, "y-axis" 2}}

(pprint data) ; ❸
;; {:a ["red" "blue" "green"],
;;  :b (:north :south :east :west),
;;  :c {"x-axis" 1, "y-axis" 2}}
```

❶By simply typing "data" at the REPL we trigger a basic printout of the content of the corresponding var object. If "data" is large, we could potentially wait a few seconds for the screen to scroll a dense wall of text.

❷ `pp` invokes `pprint` on the last evaluated expression. We can see that `pprint` is aware of what kind of object we want to print and nicely aligns keys and values for us in a readable way. Also in this case, if the object is large, we would need to wait a few seconds, but the result would still be readable.

❸We can call `pprint` directly on any printable object. We can see that `pprint` produces exactly the same output on "data" as `pp` before.

Pretty printing contribution to Clojure

`pprint` and cl-format are one of the many contributions to Clojure developed independently and then incorporated into the (now deprecated) "Clojure Contrib" project.

Tom Faulhaber (github.com/tomfaulhaber/cl-format) deserves a mention for the work he did to port `cl-format` from Lisp and pretty printing in general. He announced his library on the Clojure mailing list at the end of 2009 [250].

`pprint` and `pp` are the most prominent entry points in `clojure.pprint`, but there are other functions available in the same namespace. `clojure.pprint/write` for instance, is used by `pprint` and offers a rich set of parameters:

```
(require '[clojure.pprint :as pretty])

(doc pretty/write) ; ❶
;; -------------------------
;; clojure.pprint/write
;; ([object & kw-args])
;; Write an object subject to the current bindin
;; Use the kw-args argument to override individu
;; recursive calls). Returns the string result i
;;
;; The following keyword arguments can be passed
;;   Keyword              Meaning
;;   :stream              Writer for output or n
;;   :base                Base to use for writin
```

```
  ;;   :length                Maximum elements to sh
  ;;   :level                 Maximum depth
  ;;   :miser-width           Width to enter miser m
  ;;   :dispatch              The pretty print dispa
  ;;   :pretty                If true, do pretty pri
  ;;   :radix                 If true, prepend a rad
  ;;   :right-margin          The column for the rig
  ;;   :suppress-namespaces If true, no namespaces
```

❶The documentation on the `write` function itself is quite rich and worth showing.

❷Most of the settings are self-describing, but some of them require some explanation. "Miser-mode" is an extra-compact style of printing that triggers when the available printing width reaches the threshold indicated by `*print-miser-width*`. The reason why this is needed is that deeply nested data structure consume line width by indenting. Please check *print-miser-width* for additional information.

The following example shows how to use some of the parameters available in `clojure.pprint/write`:

```
  (require '[clojure.pprint :as pretty]) ; ❶
  (require '[clojure.java.io :as io])

  (with-open [w (io/writer "/tmp/prettyrange.txt")
```

```
   (pretty/write
     (map range (range 12 0 -1))
     :stream w   ; ❸
     :base 2     ; ❹
     :length 6)) ; ❺

 ;; cat /tmp/prettyrange.txt ; ❻
 ((0 1 10 11 100 101 ...)
  (0 1 10 11 100 101 ...)
  (0 1 10 11 100 101 ...)
  (0 1 10 11 100 101 ...)
  (0 1 10 11 100 101 ...)
  (0 1 10 11 100 101 ...)
  ...)
```

❶Other functions such as `clojure.pprint/write` are available by requiring the `clojure.pprint` namespace. We also need `clojure.java.io` in this example.

❷We open a `BufferedWriter` backed by a file in the `/tmp` folder where we can write into. The buffered writier is closed for us right after evaluating the form.

❸The ":stream" key sets the output stream to the file. This would normally be the value of `*out*`.

❹The ":base" key change the representation number to a different base. Normally numbers are printed in base 10, but here we are requesting binary representation.

❺The ":length" key establishes the max number of items to print in a collection. `write` will print only the first 6 elements, truncating the rest with "...".

❻The formatted output is visible by opening the file. On Unix based systems, you can use `cat` from the command line.

`clojure.pprint/write` supports a rich range of parameters controlling the formatting of the output. Available parameters are conveniently grouped into "dispatch tables" that can be used to print different Clojure objects. By default, `clojure.pprint/simple-dispatch` is used, but others are available such as `clojure.pprint/code-dispatch` to handle code fragments instead of data structures.

In the next example we receive some Clojure code as text. Despite being correctly formatted in the original string, we can't print correctly with default `pprint` settings:

```
(def op-fn ; ❶
  "(defn op [sel]
     (condp = sel
        \"plus\" +
        \"minus\" -
        \"mult\" *
        \"div\" /
        \"rem\" rem
        \"quot\" quot))")

(pprint (read-string op-fn)) ; ❷

;; (defn ; ❸
;; op
;;  [sel]
;;  (condp
;;   =
;;   sel
;;   "plus"
;;   +
;;   "minus"
;;   -
;;   "mult"
;;   *
```

```
  ;;   "div"
  ;;   /
  ;;   "rem"
  ;;   rem
  ;;   "quot"
  ;;   quot))
```

❶We are given a Clojure function as string. This can be the result of opening a Clojure file or perhaps it was stored in a database.

❷read-string invokes the Clojure Reader to read the content of the string. `read-string` returns the list containing the list starting with the symbol "defn". The `list` remains in its unevaluated form.

❸ `pprint` does not distinguish between a list containing code (a list that is supposed to be evaluated at some point in time) and a simple data structure.

A list that contains Clojure code should print differently from a list that contains generic data and `pprint` solves the problem by allowing different "dispatch" settings. We can change default dispatch for `pprint` using `with-pprint-dispatch` and passing a different dispatch function:

```
(pretty/with-pprint-dispatch
  pretty/code-dispatch ; ❶
  (pprint (read-string op-fn)))

;; (defn op [sel]
;;   (condp = sel
;;     "plus" +
;;     "minus" -
;;     "mult" *
;;     "div" /
;;     "rem" rem
;;     "quot" quot))
```

❶ `code-dispatch` is a multimethod defining a set of printing directives to handle code fragments. When `code-dispatch` encounters a "defn" symbol, it triggers the correct behavior to print such type of Clojure form (and likewise for "condp" or other supported forms).

A final mention for `print-table`, a function that renders Clojure maps as two-dimensional tables:

```
(require '[clojure.pprint :as pretty])

(pretty/print-table (repeat 4 (zipmap (range 10)
```

```
;; |   0 |   7 |   1 |   4 |   6 |   3 |   2 |
;; |-----+-----+-----+-----+-----+-----+-----+--
;; | 100 | 107 | 101 | 104 | 106 | 103 | 102 | 1(
;; | 100 | 107 | 101 | 104 | 106 | 103 | 102 | 1(
;; | 100 | 107 | 101 | 104 | 106 | 103 | 102 | 1(
;; | 100 | 107 | 101 | 104 | 106 | 103 | 102 | 1(
```

❶ `print-table` takes a collection of maps and prints a two-dimensional table with keys as columns and values as rows.

`print-table` uses the keys of the first map to define the headers for the table. We can override the choice of column names using the first argument:

```
(def headers [7 0 9 2])
(pretty/print-table
  headers ; ❶
  (repeat 4 (zipmap (range 10) (range 100 110))))

;; |   7 |   0 |   9 |   2 |
;; |-----+-----+-----+-----|
;; | 107 | 100 | 109 | 102 |
;; | 107 | 100 | 109 | 102 |
;; | 107 | 100 | 109 | 102 |
;; | 107 | 100 | 109 | 102 |
```

❶When passing specific keys, `print-table` only selects those keys from the collection of maps.

[250] Tom Faulhaber announced the `cl-format` library in this post groups.google.com/d/msg/clojure/hkDA8zotzUc/x3b-QBbBfvYJ to the Clojure mailing list.

18.4 PRINT-METHOD, PRINT-DUP AND PRINT-CTOR

`print-method`, `print-dup` and `print-ctor` are the entry points for the Clojure printing extension mechanism [251].

When talking about pr and print (and generally throughout the book), we've seen that Clojure produces a consistent textual representation for most of the Clojure data structures. The printing mechanism is based on a multimethod and the standard library comes with a default implementation for most of Clojure types. When that's not the case, printing defaults to the name of the class and a some additional information:

```
(deftype Point [x y]) ; ❶
;; user.Point
```

```
(pr (Point. 1 2)) ; ❷
;; #object[user.Point 0x2e6b5958 "user.Point@2e6
```

❶deftype creates the corresponding Java class in the current classpath. Clojure doesn't have any directive regarding how this new object should be printed.

❷When we `print` a new custom type, Clojure uses the default formatting: it includes the initial "#object" declaration, followed by the class name and the object hash as hexadecimal ("0x2e6b5958").

If we want to print custom types differently, we can tell Clojure using `print-method`:

```
(defmethod print-method user.Point [object write
  (let [class-name (.getName (class object))
        args (str (.x object) " " (.y object))]
    (.append writer (format "(%s. %s)" class-nam

(def point (Point. 1 2))
(def point-as-str (pr-str point))
(def point-as-list (read-string point-as-str))
(def back-to-point (eval point-as-list))

[point-as-str :type (type point-as-str)] ; ❸
;; ["(user.Point. 1 2)" :type java.lang.String]
```

```
  [point-as-list :type (type point-as-list)] ; ❹
  ;; [(user.Point. 1 2) :type clojure.lang.Persist

  [back-to-point :type (type back-to-point)] ; ❺
  ;; [(user.Point. 1 2) :type user.Point]
```

❶ `print-method` is a multimethod. We extend the multimethod with defmethod and the type of the object. `print-method` is defined with two arguments, the object that received the call to `print` (or other printing functions) and the currently open "writer" instance.

❷ We can now `.append` anything we want to the writer: in this example we pick the same format used to create a new `Point` instance `(user.Point. x y)` replacing "x" and "y" with the current content of the "object".

❸ Let's print the string representation of a `Point` first. We also print the type for clarity.

❹ We can now take the string representation and ask the Clojure Reader to parse the content of the string. We do this using read-string. The result is a `PersistentList` instance ready for evaluation.

❺ Calling eval on a list forces interpretation of the first element as a function (or Java interoperability call, like our case) and the rest of the list as arguments. `eval` invokes the `Point` constructor, generating a duplicate instance of the initial `Point` with coordinates [1 2].

The example above achieves two main goals:

1. Printing the custom type `Point` in a meaningful way.

2. Printing the custom type in such a way that can be read back into Clojure from a string (serialization).

The two aspects don't need to be handled together and Clojure provides an additional mechanism for basic serialization through `print-dup`, `print-ctor` (and the `#=` reader macro). `print-dup` is a multimethod similar to `print-method` and Clojure provides an implementation of `print-dup` for most of the Clojure types:

```
(binding [*print-dup* true] ; ❶
  (pr-str {:a 1 :b 2}))

;; "#=(clojure.lang.PersistentArrayMap/create {:
```

❶ `print-dup` is not a function to invoke directly. The dynamic variable `*print-dup*` controls `print-dup` serialization format. We can see that the map `{:a 1 :b 2}` is serialized as `PersistentArrayMap` and needs to be created explicitly with `create`.

`print-dup` can be extended to other types. In the following example we redefine `print-method` for the `Point` class to have a compact and readable representation and `print-dup` with a format suitable for serialization:

```
(defmethod print-method user.Point [object write
  (.append writer (format "[x=%s, y=%s]" (.x obj

(pr-str (Point. 1 2)) ; ❶
;; "[x=1, y=2]"

(defmethod print-dup user.Point [object writer]
  (print-ctor
    object
    (fn print-args [object writer]
      (.append writer (str (.x object) " " (.y o
    writer))

(binding [*print-dup* true] (pr-str (Point. 1 2)
;; "#=(user.Point. 1 2)"
```

❶ With a new definition of `print-method`, the `Point` instance produces a visually appealing representation. However, note that the output is not valid Clojure.

❷ We can use `print-dup` to create a Clojure-aware string representation. We define a new multimethod instance to deal with the `Point` class. `print-ctor` takes care of generating the correct constructor call.

❸ To trigger the alternate `print-dup` representation, we bind the dynamic variable `*print-dup*` to `true` before using any of the printing functions (in this case pr-str). Note that `print-ctor` outputs the constructor call inside the "reader eval macro" `#=()`.

The reader eval macro `#=` has the same effect of calling eval on the form that follows it. As a consequence, read-string can be used to read the string back into a list and evaluate it at the same time:

```
(binding [*print-dup* true]
  (read-string (pr-str (Point. 1 2)))) ; ❶
;; [x=1, y=2]
```

❶ `read-string` produces the combined effect of reading and evaluating Clojure code when used with `print-dup`-aware objects.

Clojure serialization with `print-dup` is effective but vulnerable to code injection. This might explain why `print-dup` is generally undocumented and `read-string` is discouraged (unless you are in total control of serialized data, a better option is to use clojure EDN). However, there might be cases where it makes sense to use `print-dup`, for example to temporarily park data on disk.

[251] For a historical perspective see groups.google.com/d/msg/clojure/5wRBTPNu8qo/1dJbtHX0G-IJ. The suffix "dup" refers to "duplication", as the printed object is effectively duplicated once it is evaluated back from a string.

18.5 SLURP AND SPIT

`slurp` and `spit` are very common functions and they are used extensively throughout the book. They are possibly the simplest way to read and write a file, respectively:

```
(spit "/tmp/test.txt" "Look, I can write a file!
(slurp "/tmp/test.txt") ; ❷
;; "Look, I can write a file!"
```

❶ `spit` takes two arguments, a destination path and the content.
❷ `slurp` takes one argument (and additional options not visible here).

Additionally, if the first string argument can be interpreted as an URL [252], `slurp` can read from it and return a string (`spit`, on the other hand, works only with URLs starting with `file://`):

```
(def book (slurp "https://tinyurl.com/wandpeace"
(reduce str (take 24 book))
;; "\r\nThe Project Gutenberg"
```

❶ `slurp` comes handy to quickly load the content of a web site, such as this book from the Gutenberg project. The argument is a string, but the format is compatible with a `java.net.URL` object.

Both functions also accept additional key-value pairs arguments:

```
(slurp "/etc/hosts" :encoding "UTF-16") ; ❶
;; "潳瑰舣瑯扡獭ㄲㄲ潢慲捰"
```

```
  (spit "/tmp/txt" "Something." :append true) ; ❷
  (spit "/tmp/txt" "Something." :append true)
  (slurp "/tmp/txt")
  ;; "Something.Something."
```

❶The `:encoding` key forces a particular encoding on the string being read (like in this case) or written (using `spit`). Here we see the result of interpreting an UTF-8 file as UTF-16: the content stops making sense. The default encoding is the value of the Java property "file.encoding" which is normally set to "UTF-8".

❷We can see the effect of `:append true` on `spit`: subsequent calls to write on the same file are appended to the current content instead of overwriting it (the default).

Both `slurp` and `spit` do their best to deal with the type of their arguments automatically. The following table shows an example for each of the supported types. To run the snippets in the table, you need the following imports:

```
  (import '[java.io FileReader FileWriter])
  (import '[java.io ByteArrayInputStream ByteArray
  (import '[java.io File])
  (import '[java.net URL Socket])
```

Table 18.1. Types of read/write channels accepted by `slurp` and `spit`

I/O Type	`slurp`
`java.io` Reader/Writer subclass	`(slurp (FileRea "/etc/hosts"))`
`java.io` InputStream/OutputStream\nsubclasses	`(slurp **out** `\n array [1 2 3]))))`
`java.io` File	`(slurp (File. "/etc/hosts"))`
`java.net` URL (or URI)	`(slurp (URL. "http://google.`
`java.net` Socket	`(slurp (Socket. "localhost" 300`

[252] An Uniform Resource Locator is a conventional format to describe a resource on the Internet defined by the following RFC: tools.ietf.org/html/rfc1738

18.6 SUMMARY

Following up with Lisp tradition, Clojure has an elegant and complete printing system that allows deep customization and extensibility. The chapter gives a few examples, but there are more parameters to explore especially for formatting and pretty printing. The formatting system uses a small embedded language to define formatting directives. However, similarly to regular expressions, one should not abuse `cl-format` to avoid code that is too cryptic to follow. The chapter also explores how to perform serialization of custom objects, a technique with many applications in distributed systems.

19 STRINGS AND REGULAR EXPRESSIONS

One of the most common programming tasks is handling strings. Java provides an immutable `java.lang.String` type with a rich API which has been battle tested over many years of enterprise application development. Apart from a few functions in `clojure.core` and a dedicated `clojure.string` namespace, Clojure builds on top of this robust foundations:

```
(class "I'm a Clojure string") ; ❶
;; java.lang.String
```

❶A Clojure string is implemented on top of a Java string.

Another common feature found in programming languages is the presence of facilities for regular expressions. A "regex" is a string which defines a search pattern for other strings. Clojure uses the same Java regular expression abstraction while offering specialized functions including a pattern match syntax literal:

```
(class #"I'm a pattern match string") ; ❶
;; java.util.regex.Pattern
```

❶ Clojure provides a quick way to create pattern match objects (the "query string" that defines the way to search into another string).

String support and regular expression functions are the topic of this chapter.

19.1 STR

`str` is one of the most used Clojure functions. It takes one or more objects and returns the concatenation of their string representation:

```
(str "This " 'is \space 1 " sentence") ; ❶
;; "This is 1 sentence"
```

❶ `str` concatenates all arguments into a single string after converting them to their string representation.

It is common to see `str` used on all sort of values, because every type in Clojure (inheriting this behavior from Java) contains at least a default conversion to string. The default

string conversion only contains the class of the object and a hexadecimal id. Sometimes this behavior is not desirable, as in the case of this lazy sequence we would like to print on screen:

```
(str :a 'b 1e8 (Object.) [1 2] {:a 1}) ; ❶
;; ":ab1.0E8java.lang.Object@dd2856e[1 2]{:a 1}"

(str (map inc (range 10))) ; ❷
;; "clojure.lang.LazySeq@c5d38b66"

(pr-str (map inc (range 10))) ; ❸
;; "(1 2 3 4 5 6 7 8 9 10)"
```

❶A small sample of values supported by `str`. All types have a string representation, because `Object` (the parent of all classes in Java) has a default `.toString` method to use when there is no other choice.

❷Types like `clojure.lang.LazySeq` are containers for collections that might not be evaluated yet. They don't have a specialized `toString` method by design, to prevent unsolicited evaluations of potentially infinite sequences.

❸There is a way to ask Clojure to print the lazy sequence using pr-str. `pr-str` finds the presence of a more specific print-method override for the serialization of the sequence and prints it as a list.

If a data container does not provide a string representation, we can always apply or reduce each items with `str`:

```
(apply str (range 10)) ; ❶
;; "0123456789"

(reduce str (interpose "," (range 10))) ; ❷
;; "0,1,2,3,4,5,6,7,8,9"
```

❶ We can use `apply` or `reduce` (example below) to concatenate the string representation of all items in an input collection.

❷ Separation with commas is typical for human-readable data rendering, for example in CSV (comma separated values) files.

In terms of general performance, `str` is based on `java.lang.StringBuilder`, which accumulates all string fragments in a mutable buffer (in case of multiple arguments). Although reduce is generally a faster option, this is one of the cases in which apply performs better: `reduce` would simply call `str` each iteration throwing away any intermediate `StringBuilder`. Here's a benchmark that shows the difference:

```
(require '[criterium.core :refer [quick-bench]])

(let [v (vec (range 1000))]
  (quick-bench (apply str v))) ; ❶
;; Execution time mean : 36.978891 µs

(let [v (vec (range 1000))]
  (quick-bench (reduce str v))) ; ❷
;; Execution time mean : 609.917674 µs
```

❶apply calls the variadic arity of `str` which pushes each converted string into a mutable Java `StringBuilder` gradually building the string. This is the most efficient way to concatenate strings.

❷reduce produces the same result but it's much slower: each iteration creates a new `StringBuilder` of the accumulated string plus the appended new item, which is immediately concatenated into a new string and thrown away.

19.2 JOIN

The `join` function is part of the `clojure.string` namespace. The basic form accepts a sequential collection of objects and concatenates them into a string similarly to str:

```
(require '[clojure.string :refer [join]]) ; ❶

(join (list "Should " "this " "be " 1 \space 'se
;; "Should this be 1 sentence?" ; ❷
```

❶We need to require the function from `clojure.string` namespace.

❷We can see objects of different types being converted to their string equivalent.

`join` also accepts an optional separator with semantic similar to interpose, but returning the result as a string concatenation:

```
(join "," (range 10)) ; ❶
;; "0,1,2,3,4,5,6,7,8,9"

(apply str (interpose "," (range 10))) ; ❷
;; "0,1,2,3,4,5,6,7,8,9"
```

❶ `join` also allows a separator, producing a shorter version of the similar `interpose` call.

❷A quick comparison with the equivalent `interpose` function call that achieve the same effect.

`join` works similarly to `str` using a `StringBuilder` instance to accumulate string fragments before producing the final string. Compared to the `interpose` version above, `join` does not produce an intermediate lazy sequence, resulting in faster processing (especially for large collections):

```
(require '[criterium.core :refer [quick-bench]])

(let [xs (interpose "," (range 10000))]
  (quick-bench (apply str xs))) ; ❶
;; Execution time mean : 595.661421 µs

(let [xs (range 10000)]
  (quick-bench (join "," xs))) ; ❷
;; Execution time mean : 418.126884 µs
```

❶"xs" is the result of the lazy application of interpose to a range. Considering we want a single string out of the collection, we pay the price of producing a lazy sequence to always consume its content eagerly.

❷Internally, `join` adds the separator to the `StringBuilder` instance while iterating the input collection, without the need to cons the separator into a lazy sequence. This results in a speed improvement.

19.3 REPLACE, REPLACE-FIRST, RE-QUOTE-REPLACEMENT

`clojure.string/replace` (or `s/replace` for short) replaces segments of a string with another string. The segments can be identified by a single character, string or regular expression. In the following example we want to replace all hyphen "-" with spaces:

```
(require '[clojure.string :as s]) ; ❶

(def s "Chat-room messages are up-to-date")
(s/replace s \- \space) ; ❷
;; "Chat room messages are up to date"
```

❶We need to require the `clojure.string` namespace to access `replace`.

❷Hyphens are replaced by spaces using `s/replace`.



Warning

There is another replace function in the `core` namespace. `clojure.core/replace` performs substitutions of items in a collection instead of fragments in a string.

We could use a string instead of a single character to replace entire words, but we can't mix characters substitutions with string targets (or vice versa):

```
(s/replace "Closure is a Lisp" "Closure" "Clojur
;; "Clojure is a Lisp"

(s/replace "I'm unjure" "j" \s) ; ❷
;; ClassCastException
```

❶We can use `replace` to replace entire words using strings as arguments.

❷However, we can't mix the target and the substitution types.

`s/replace` also accepts regular expressions: when using a regular expression as the first argument, we also need to pass a function of a single argument, where the argument is the matching expression:

```
(def s "Why was 12 afraid of 14? Because 14 ate

(s/replace s #"\d+" #(str (/ (Integer/valueOf %)
;; "Why was 6 afraid of 7? Because 7 ate 9."
```

❶This joke contains a fundamental mistake: it looks like the number in the sentence were doubled.

❷The last argument of `s/replace` is a function of one argument returning a string. We match all numbers in the sentence and divide them by 2.

Additionally, in case of a regular expression target, we can use group capturing patterns "$1", "$2", etc. [253]:

```
(def s "Easter in 2038: 04/25/2038, Easter in 22

(s/replace s #"(\d{2})/(\d{2})/(\d{4})" "$2/$1/$
;; "Easter in 2038: 25/04/2038, Easter in 2285:
```

❶A string contains multiple dates expressed in month, day and year format. We can create a matching pattern that uses round parenthesis around the subparts of the date string to capture them individually.

❷The special symbol "$" followed by a progressive number represents the different matching groups. To swap month and day, we need to invert the position of "$1" and "$2".

In the last example we can see that the dollar sign "$" has a special meaning when used inside a regular expression. If we wanted to treat it literally (for example to express a currency), we could use `re-quote-replacement`:

```
(def s "May 2018, June 2019")

(s/replace s #"May|June" "10$ in") ; ❶
;; IllegalArgumentException

(s/replace s #"May|June" (s/re-quote-replacement
;; "10$ in 2018, 10$ in 2019"
```

❶ We want to replace "10$" for each month appearing in the string. Our first attempt fails because the "$" has a special meaning for regular expression.
❷ `re-quote-replacement` prevents the wrong (in this case) interpretation of the dollar sign.

`replace-first` has the same calling contract as `replace`, but it only executes the first substitution, if any:

```
(def s "I could have a drink here and wine home.

(s/replace s #"a drink|beer|wine" "water") ; ❶
;; "I could have water here and water home."

(s/replace-first s #"a drink|beer|wine" "water")
;; "I could have water here and wine home."
```

❶ `s/replace` performs a substitution for each matching expression in the input string.
❷ `s/replace-first` stops at the first substitution.

[253] Entire books and websites are dedicated to regular expressions. One useful online resource is this website.

19.4 SUBS, SPLIT AND SPLIT-LINES

The functions `subs` (in `core`), `clojure.string/split` and `clojure.string/split-lines` are useful to retrieve portions of a string:

```
(require '[clojure.string :as s]) ; ❶

(def s "The quick brown fox jumps over the lazy

(subs s 20 30) ; ❷
;; "jumps over"

(s/split s #"\s") ; ❸
;; ["The" "quick" "brown" "fox" "jumps" "over" "

(s/split-lines s) ; ❹
;; ["The quick brown fox" "jumps over the lazy d
```

❶ `s/split` and `s/split-lines` are defined inside the `clojure.string` namespace.

❷ `subs` returns a sub-string delimited by "start" (exclusive) and "end" (inclusive). If the second argument is omitted, the end index becomes the length of the string.

❸ `s/split` creates a split of the input string for each match of the given regular expression. "\s" means "any white space character" which also includes "\n" new line separator. The resulting vector contains all the splits.

❹ `s/split-lines` executes an implicit split for each "\n" new line separator. The resulting vector contains all splits by new lines.

`subs` is useful when the input string has a fixed structure and the portion to extract always appears in the same position:

```
(def errors ; ❶
  ["String index out of range: 34"
   "String index out of range: 48"
   "String index out of range: 3"])

(map #(subs % 27) errors) ; ❷
;; ("34" "48" "3")
```

❶ `errors` contains error messages coming from a log file. The message is repeating with the same structure and the variable portion always appear at index "27" (the index is 0 based).
❷ We don't need to specify the end position if we know that the number always appear at the end of the string.

If the portion of the string we need to extract is not fixed, we can use a regular expression with `s/split`:

```
(def errors ; ❶
  ["String is out of bound: 34"
   "48 is not a valid index."
   "Position 3 is out of bound."])

(map #(peek (s/split % #"\D+")) errors) ; ❷
;; ("34" "48" "3")
```

❶ The new error messages contain the information about the wrong index at different position in the string.
❷ By splitting at "\D+" ("one or more non-digit characters") we remove everything from the input message that is not a digit. In doing so, we produce a vector as output that contains the number as the last element. peek access the last element of a vector efficiently.

With `s/split` we have access to more sophistication than `subs`. In the following example we access the list of printer configurations available from "/etc/cups/ppd" (the default location on a Mac operative system). The var definition below contains the output of the command "ls" executing on the folder. The output returns a single string with all the files in the folder. We can use `s/split-lines` first and then `s/split` to isolate the details we are interested to see:

```
(require '[clojure.java.shell :refer [sh]]) ; ❶
(def ls (:out (sh "ls" "-al" "/etc/cups/ppd")))
(def printers (s/split-lines ls)) ; ❸

(last printers) ; ❹
;; "-rw-r--r--   1 root  _lp  1111829 10 May 13:

(sequence
        (comp (map #(s/split % #"\s+")) ; ❺
                                  (map last)
                                  (filter #(re-fin
        printers)

;; ("Brother_DCP_7055.ppd" ; ❼
;;  "Training_room.ppd"
;;  "_192_168_176_12.ppd")
```

❶ sh executes a command in the default shell provided by the operating system. It is available in the clojure.java.shell namespace.

❷ `sh` takes a list of string representing the command (with options) and return a map that contains the output string as a value of the `:out` key.

❸ As a first step, we split the string into a list of lines, more or less as they would appear on the screen after executing the command from a terminal.

❹ We can see a sample of the content of `printers`. Each line contains further information about the files in the folder, including permissions, ownership and so on. The last part of the line contains the name of the printer configuration with extension ".pdd".

❺ Processing of each line starts with a `s/split` instruction that splits the line at each group of spaces or tabs, isolating the interesting parts into single strings.

❻ The printer name will appear as the last element on each line. A few lines don't contain a printer name though, such as "." or ".." (special directory files). We remove them using filter.

❼ We can see an example of the intended output (you might need to tweak the shell command and the regular expression to execute this example on your system).

A regular expression engine, such as the one provided with Clojure (through Java), is a sophisticated tool that comes with some performance penalty. Most of the time, regular expressions are invaluable feature, but if you have a fixed structure string to analyze, it makes sense to avoid regular expressions to speed up computation. Here's a benchmark that compares `subs` and `s/split` to give you an idea of the performance implications:

```
(require '[criterium.core :refer [quick-bench]])

(let [s "String index out of range: 34"]
  (quick-bench (subs s 27))) ; ❶
;; Execution time mean : 22.716615 ns

(let [s "String index out of range: 34"
      re #"\D+"]
  (quick-bench (s/split s re))) ; ❷
;; Execution time mean : 561.346569 ns
```

❶ `subs` retrieves the matching number as a string, not a vector.

❷ `s/split` always return a vector. We should use peek to obtain the same result as `subs`, but we can leave that out of the benchmark for the moment.

As you can see regular expressions are much slower than simply extract the portion of the string from a fixed position. This is expected, given the kind of capabilities they offer. However, when doing performance analysis on a slow application, regular expressions are almost always involved if they are present. If you can't do without, there are ways to optimize them, for example avoiding patterns that are too broad or unnecessary backtracking [254].

[254] This article contains a few tricks to speed up regular expressions: www.loggly.com/blog/five-invaluable-techniques-to-improve-regex-performance/

19.5 TRIM, TRIML, TRIMR, TRIM-NEWLINE

The goal of the functions in this section is to remove one or more "spaces" around strings. The definition of "space" comes from Java and includes quite a few variants, such as tabs, returns, other separators and related variants (see the following call-out for more details).

Java white space definition

Java "white space" definition includes the common "space", "tabs", "line feed" characters plus several more from the

extended Unicode character set [255]. Some white space characters are printable, others are not:

```
(map ; ❶
  #(hash-map :int % :char (char %) :hex (format
  (filter (comp #(Character/isWhitespace %) char

;;({:int 9     :hex "9"    :char \tab}
;; {:int 10    :hex "a"    :char \newline}
;; {:int 11    :hex "b"    :char \ } ; ❷
;; {:int 12    :hex "c"    :char \formfeed}
;; {:int 13    :hex "d"    :char \return}
;; {:int 28    :hex "1c"   :char \}
;; [...]

(require '[clojure.string :as s]) ; ❸

(s/blank? "\t \n \u000b \f \r \u001c \u001d \u00
;; true
(s/blank? "\u0020 \u1680 \u2000 \u2001 \u2002 \u
;; true
(s/blank? "\u2004 \u2005 \u2006 \u2008 \u2009")
;;true
(s/blank? "\u200a \u2028 \u2029 \u205f \u3000")
;;true
```

❶If we range over integers (up to 2^{16}, or 65536) we can list all entries in the UTF-16 Unicode character set. `Character/isWhitespace` only works up to 65536 in any case.

❷Some of the entries in this list are not visible, because the character is not printable.

❸clojure.string/blank? uses the same Java method to determine if a string contains only white spaces. We can confirm that `blank?` returns `true` when using any of characters in the list above.

Note how the Ogham space mark `\u1680` is a white space with a printable representation (similar to a "-") but others printable white space like the Ethiopic Wordspace `\u1361` are not considered a white space. The related Wikipedia page at en.wikipedia.org/wiki/Unicode contains much more information about the Unicode specification.

All four functions belong to the `clojure.string` namespace, so we need an explicit require to use them. `s/trim` removes one or more Java white spaces only if they appear at the edges of a string:

```
(require '[clojure.string :as s])
```

```
(s/trim "   *Look, no more spaces.*  ") ; ❶
;; "*Look, no more spaces.*"

(s/trim "\t1\t2n\n") ; ❷
```

❶ `s/trim` removes white spaces from both ends of a string.
❷ As discussed, the definition of white space also includes other non-printable characters such as tabulations.

`s/trimr` and `s/triml` are similar to `s/trim` but they only remove Java white spaces from the right or the left side of a string, respectively:

```
(s/trimr "   *Spaces on the left are not removed
;; "   *Spaces on the left are not removed with

(s/triml "   *Spaces on the right are not remove
;; "*Spaces on the right are not removed with tr
```

❶ `s/trimr` only removes Java spaces that appear on the right edge of a string.
❷ `s/triml` only removes Java spaces from the left side of the string.

Finally, `s/trim-newline` only removes newlines or return characters (they are encoded `\n` and `\r` respectively) from the right side of a string:

```
(s/trim-newline "\n  Only return and\n newline a
;; "\n  Only spaces and\n newline at the end."
```

❶ `s/trim-newline` only trims newlines `\n` and returns `\r` at the end of the string.

[255] Unicode defines some 25 characters under the name of white spaces: en.wikipedia.org/wiki/Whitespace_character#Unicode

19.6 ESCAPE, CHAR-NAME-STRING, CHAR-ESCAPE-STRING

`escape` is a function from the `clojure.string` namespace dedicated to the selective replacement of characters in a string. The function takes 2 arguments: a string "s" and a function "f" from char to a printable object "cmap". `escape` iterates each character "c" in "s" and if `(f c)` is not `nil`, then `(f c)` replaces `c` in the output string. Clojure maps are a very convenient option to use as argument:

```
(require '[clojure.string :as s])

(def link ; ❶
 "Patterson, John: 'Once Upon a Time in the West

(def link-escape ; ❷
  {\, "_comma_"
   \space "_space_"
   \. "_dot_"
   \' "_quote_"
   \: "_colon_"
   \newline "_newline_"})

(s/escape link link-escape) ; ❸
;; "Patterson_comma__space_John_colon__space__qu
;; space_a_space_Time_space_in_space_the_space_W
```

❶ `link` is a string that the user wants to use as a title of a section for web page. The section can then be used as a link to share on the Internet. As the reader can see, the section title contains many characters that are not allowed in the address bar of a browser.

❷The option to use a map is very convenient. `link-escape` can be used as a function for `escape`.

❸ `escape` replaces all matching keys with their corresponding values in the output string. Please note that the output contains an artificial newline to limit the length of the line.

`char-name-string` and `char-escape-string` are two public maps in the `core` namespace. The reasons for their existence is mainly for internal use by Clojure itself, but it might be useful to know about their existence. Similarly to `link-escape` that we used in he example, `char-name-string` and `char-escape-string` can be used with `escape` or stand-alone:

```
char-name-string ; ❶

;; {\newline "newline"
;;  \tab "tab"
;;  \space "space"
;;  \backspace "backspace"
```

```
  ;;  \formfeed "formfeed"
  ;;  \return "return"}

  (map #(char-name-string % %) "Hello all!\n") ; ❷
  ;; (\H \e \l \l \o "space" \a \l \l \! "newline"
```

❶By typing `char-name-string` without invoking it in parenthesis, we can see what substitution are available.

❷ `char-name-string` is available without using require. We treat the input string as a sequence and perform each transformation using `char-name-string`. Note that we can pass two arguments (the percent `%` appears twice): the first looks up the character, the second is used as the default.

`char-escape-string` performs a similar replacement, but instead of replacing the input with the name of the corresponding character, it uses its quoted escape sequence:

```
  char-escape-string ; ❶

  ;; {\newline "\\n"
  ;;  \tab  "\\t"
  ;;  \return "\\r"
  ;;  \" "\\\""
  ;;  \\  "\\\\"
  ;;  \formfeed "\\f"
```

```
  ;;  \backspace "\\b"}

  (def s "Type backslash-t '\t' followed by backsl
  (println s)
  ;; Type backslash-t '      ' followed by backsla
  ;; '

  (println (s/escape s char-escape-string)) ; ❸
  ;; Type backslash-t '\t' followed by backslash-n
```

❶As we did before, we can print the content of the substitution map to inspect its content.

❷We'd like to print these instructions as they appear in the string, but println correctly convert special sequences like `\n` into a new line on screen.

❸ `s/escape` with `char-escape-string` converts the special character in their quoted string so they print as intended originally.

19.7 LOWER-CASE, UPPER-CASE, CAPITALIZE

`lower-case`, `upper-case` and `capitalize` are part of the `clojure.string` namespace. `lower-case` and `upper-case` turn all characters in a string to lowercase or uppercase,

respectively. We could use `upper-case` to render Clojure similarly to the original Lisp:

```
(require '[clojure.repl :refer [source]])
(require '[clojure.string :as s])

(-> some?          ; ❶
    source         ; ❷
    with-out-str   ; ❸
    s/upper-case   ; ❹
    println)

;; (DEFN SOME? ; ❺
;;   "RETURNS TRUE IF X IS NOT NIL, FALSE OTHERW
;;   {:TAG BOOLEAN
;;    :ADDED "1.6"
;;    :STATIC TRUE}
;;   [X] (NOT (NIL? X)))
```

❶some? is a simple function in the standard library that we want to lookup.
❷Given a var, `clojur.repl/source` reads the file the var was defined in and loads its source definition.
❸ `with-out-str` creates a thread binding for standard output (where the source code would usually prints) to a string, so we can use the string later on.
❹ `s/upper-case` produces the uppercase version of the input string. This string contains non-printable characters like "newlines" that are interpreted correctly by println.
❺The "all caps" Clojure now contains unrecognized symbols and wouldn't compile if we tried to use it.

Here's an example using `lower-case` to implement case-insensitive equivalence between strings:

```
(def primary-colors #{"red" "green" "blue"}); ❶

(def book (slurp "https://tinyurl.com/wandpeace"

(->> (s/split book #"\s+") ; ❸
     (filter primary-colors)
     frequencies)
;; {"red" 87, "blue" 64, "green" 38}

(->> (s/split book #"\s+") ; ❹
```

```
      (map s/lower-case)
      (filter primary-colors)
      frequencies)
;; {"red" 89, "blue" 64, "green" 38}
```

❶We have a Set of primary colors. We could equivalently spell all colors in uppercase, assuming we are happy with that specific portion of the code standing out for no special reasons.

❷ `book` contains a string version of a large text, for example "War and Peace".

❸We use `s/split` to split the book into single words. We do this with a simple regular expression that covers most of the cases. We can filter the `primary-colors` only by using filter and the set itself as a predicate. Finally, we call frequencies to see the numbers. In doing so, we compare our list of all lowercase colors to the content of the book, which might appear with a different case.

❹On a second attempt, words are converted to lower case before filtering. We can see that there are two additional "red" occurrences, probably because they appeared at the beginning of the sentence.

`capitalize` transforms the first letter of a string to uppercase and all the other into lowercase. This is useful to unify the

format of a list, or to spell proper nouns correctly. We could use `capitalize` on a vector of customer names collected from different sources to be sure they are spelled correctly:

```
(def names
  ["john abercrombie"
   "Brad mehldau"
   "Cassandra Wilson"
   "andrew cormack"])

(map
  (fn [name]
    (->> (s/split name #"\b") ; ❶
         (map s/capitalize)   ; ❷
         s/join))             ; ❸
  names)

;; ("John Abercrombie"
;;  "Brad Mehldau"
;;  "Cassandra Wilson"
;;  "Andrew Cormack")
```

❶ The first step is to `s/split` the string containing the full name. We need to join the string back at the end, so we want to preserve spaces during the split. The regex "\b" means "any word boundary" (which includes words made of spaces or other characters). mapcat collapses the inner vectors into a single collection of words.

❷ `s/capitalize` turns each first letter to uppercase.

❸ Now that the names are in the correct format, we need to prepare to join the strings back together with `s/join`.

Note that `upper-case`, `lower-case` and `capitalize` can be used on any printable object (virtually all Clojure and Java types):

```
(map s/upper-case ['symbols :keywords 1e10 (Obje
;; ("SYMBOLS" ":KEYWORDS" "1.0E10" "JAVA.LANG.OB
```

19.8 INDEX-OF, LAST-INDEX-OF

`index-of` and `last-index-of` retrieve the index of the given substring (or single character) inside another string:

```
(require '[clojure.string :as s]) ; ❶

(s/index-of "Bonjure Clojure" \j) ; ❷
```

```
;; 3

(s/last-index-of "Bonjure Clojure!" "ju") ; ❸
;; 11
```

❶Both `index-of` and `last-index-of` are declared in the `clojure.string` namespace.
❷The first `\j` appears at index 3 (uppercase "B" is at index 0).
❸The last occurrence of the string "ju" starts at index 11.

Both functions take an optional integer index to start the search from. `index-of` skips the first "n" characters if "n" is given, while `last-index-of` jumps at "n" before starting the search backward:

```
(s/index-of "Bonjure Clojure" \j 4) ; ❶
;; 11

(s/last-index-of "Bonjure Clojure!" "ju" 10) ; ❷
;; 3
```

❶ `index-of` skips the first "4" chars from the input string before starting the search. Since "4" is beyond the position of the first "j", the next "j" is found at index 11.

❷ `last-index-of` jumps at index "10" before starting the search backward. The previous "ju" in the string is found at index "3".

If the target char or string is not found or the start index is beyond the string boundaries, both functions return `nil`:

```
(s/index-of "Bonjure Clojure" "z")          ;; n
(s/index-of "Bonjure Clojure" "j" 20)       ;; n
(s/last-index-of "Bonjure Clojure" "z")     ;; n
(s/last-index-of "Bonjure Clojure" "j" -1)  ;; n
```

❶A group of example showing what happens when we search a non existent substring or we pass values of the start index "n" beyond the string boundaries.

Apart from strings and single character, other types of `java.lang.CharSequence` are accepted, for example `java.lang.StringBuffer`:

```
(import 'java.lang.StringBuffer)
```

```
(s/index-of ; ❶
  (doto (StringBuffer.)
        (.append "Bonjure")
        (.append \space)
        (.append "Clojure"))
  \j)
;; 3
```

❶ The `java.lang.CharSequence` interface has a few implementations available in the Java standard library, such as `StringBuffer`, `StringBuilder` or the very common `String` class. `index-of` and `last-index-of` work with any `CharSequence`.

19.9 BLANK?, ENDS-WITH?, STARTS-WITH?, INCLUDES?

These functions belonging to the `clojure.string` namespace, are predicates to identify the presence of a specific char or substring in another string (or more generally, a `java.lang.CharSequence` such as `StringBuffer` or `StringBuilder`).

`blank?` takes a `java.lang.CharSequence` and returns `true` if it contains only "white spaces". Java definition of "white space" includes a group of Unicode characters, including

proper spaces, tabs and other kind of non-printable chars (please see this summary list in the trim section):

```
(require '[clojure.string :as s]) ; ❶

(s/blank? " \t \n \f \r ") ; ❷
;; true

(s/blank? "\u000B \u001C \u001D \u001E \u001F")
;; true
```

❶Remember to require the `clojure.string` namespace.
❷Here's a first sample of some white space types: the space itself, tabulation, newline, carriage-feed, return.
❸In this example we can see a few more, less common, Unicode characters that are also considered white spaces.

`ends-with?` and `starts-with?` return `true` when the given substring appears at the end or the beginning of another string, respectively:

```
(s/starts-with? "Bonjure Clojure" "Bon") ;; true
(s/starts-with? "Bonjure Clojure" "Clo") ;; fals
(s/starts-with? "" "") ;; true
(s/starts-with? "Anything starts with nothing."
```

```
(s/ends-with? "Bonjure Clojure" "ure") ;; true
(s/ends-with? "Bonjure Clojure" "Bon") ;; false
(s/ends-with? "" "") ;; true
(s/ends-with? "Anything ends with nothing." "")
```

❶ Note how, for both `starts-with?` and `ends-with?`, the empty string always starts or ends a given string, returning `true`.

`includes` looks for a substring match in any position for a given string:

```
(s/includes? "Bonjure Clojure" "e C") ; ❶
;; true
```

❶ `includes?` verifies the presence of a substring inside another string.

19.10 RE-PATTERN, RE-MATCHER, RE-GROUPS, RE-SEQ, RE-MATCHES, RE-FIND

A regular expression is a pattern definition language that enables search, validation or replacement of strings. Clojure, like other programming languages, contains a set of functions

and facilities dedicated to regular expressions. Here's a brief summary before more detailed examples:

- `re-pattern` creates a new `java.util.regex.Pattern` object instance. This object performs the parsing and analysis of the regular expression, preparing it for the subsequent application to a string. There are good reason to separate the creation of the pattern from its application: the pattern is usually reused across many strings and we don't want to incur in the parsing costs multiple times. Clojure also contains a regular expression syntax literal `#""` which produces the same results as `re-pattern`.
- `re-matcher` is the next logical step and creates a `javax.util.regex.Matcher` object which is ready to work on a specific string. The resulting object can retrieve matching patterns, restrict the search to a specific region in the string or perform replacements. The object returned by `re-matcher` is stateful and remembers the position at which the last match was requested. Although the rest of the Clojure functions understand the `Matcher` object workflow, this isn't usually the most used regular expression entry point.

- `re-groups` retrieves all the matching groups of a regular expression against a string. Groups are a regular expression specific concept connected to the presence of round parenthesis in the regular expression. Each pair of parenthesis isolates a portion of the string that is returned. This is useful to match a larger expression but to only retrieve specific parts of it.
- `re-seq` creates a lazy sequence out of matching groups of a regular expression on a string. This is useful on long strings with thousands of potential matches to only consume part of the matches.
- `re-find` returns the first match (or the next match, if any) of the given regular expression on a string.

From the summary we can see that there is a stateless part of the API and a stateful part based on repeated interaction with a `Matcher` object. The stateless part is certainly the most used:

```
(filter
  #(re-find #"-seq" (str (key %))) ; ❶
  (ns-publics 'clojure.core))      ; ❷

;; ([tree-seq #'clojure.core/tree-seq] ; ❸
;;  [line-seq #'clojure.core/line-seq]
```

```
  ;;  [iterator-seq #'clojure.core/iterator-seq]
  ;;  [enumeration-seq #'clojure.core/enumeration-
  ;;  [resultset-seq #'clojure.core/resultset-seq]
  ;;  [re-seq #'clojure.core/re-seq]
  ;;  [lazy-seq #'clojure.core/lazy-seq]
  ;;  [file-seq #'clojure.core/file-seq]
  ;;  [chunked-seq? #'clojure.core/chunked-seq?]
  ;;  [xml-seq #'clojure.core/xml-seq])
```

❶You can use `re-find` as a predicate to verify the presence of a substring inside another string.

❷The list of strings in this case comes from all the public function names in the `core` namespace.

❸The results answer the question: "What functions in the standard library contains -seq in their name?"

`re-find` is a good choice as a predicate because even if there are more matches, it stops at the first one. But in case we want to extract all matching parts, we need to use `re-seq`. Here for example we want all emails found in a web page:

```
  (def so-contacts (slurp "https://stackoverflow.c

  (set (map last (re-seq #">(\S+@\S+\.com)<" so-co
  ;; #{"legal@stackoverflow.com"}
```

❶The regular expression presented here is a quick solution relying on the presence of HTML tags around the email address. It works sufficiently well in most cases of web scraping. It also assumes emails belong to the ".com" domain that is certainly not true for all email addresses. `re-seq` performs multiple "find" operations on the same `Matcher` to accumulate results.

We can now briefly compare a few options to verify if a string contains another string. We've seen index-of, re-find and includes?:

```
(require '[criterium.core :refer [quick-bench]])
(require '[clojure.string :as s])

(def contacts "Contact us: support@manning.com o

(let [s contacts]
  (quick-bench (s/index-of s "support@manning.co
;; Execution time mean : 16.570516 ns

(let [s contacts
      re #"support@manning.com"]
  (quick-bench (re-find re s)))
;; Execution time mean : 345.104914 ns ; ❸

(let [s contacts]
```

```
    (quick-bench (s/includes? s "support@manning.c
  ;; Execution time mean : 18.364512 ns
```

❶As for the rest of the book, the Criterium library is used to benchmark functions.

❷The benchmark consists of searching for the Manning support email address in a short string.

❸Searching with `re-find` is about 20 times slower.

`re-find` is penalized because it has to analyze the regular expression before applying it to the input string. At the same time `re-find` allows for much powerful features than checking the presence of a substring.

19.11 SUMMARY

In this chapter, we've seen the rich set of functions available in Clojure to process strings. In the unlikely case that the content of `clojure.string` was not enough, Clojure can always access the even larger set of methods available on the `java.lang.String` class. Similarly, Clojure relies on Java for regular expressions. In both cases, Clojure offers a much better and compact interface compared to Java.

20 MUTATION AND SIDE EFFECTS

Clojure implements and encourages immutability starting from its design principles. However, Clojure is a pragmatic language that allows mutability and side-effects when necessary. Clojure approach to mutability is still elegant and mostly functional. In this chapter we are going to see functions and data structures that mainly exist for their mutable or side-effecting nature. Other parts of Clojure are amenable to mutability and have been described in their respective chapters (for example, concurrency or Java interoperability). Let's get started with transients.

20.1 TRANSIENT, PERSISTENT!, CONJ!, POP!, ASSOC!, DISSOC! AND DISJ!

A transient is a mutable state available to some Clojure data structures and also a function with the same name. Currently supported collections are: vectors, maps and sets.

Transients exist to solve an optimization problem. When large persistent data structures are created or transformed, individual processing steps are subject to immutability overhead, even when clients of the collection are just interested

in the final output. Transients offer a way to turn immutability off while the collection is still processing. During transient state the collection:

- Stops supporting typical functions like assoc or conj.
- Supports similar but mutating functions like `assoc!` or `conj!`.
- Can turn back to persistent by calling `persistent!`.

A transient collection is designed to prevent accidental use by the rest of the standard library outside the transient state:

```
(class (transient [])) ; ❶
;; clojure.lang.PersistentVector$TransientVector

(conj (transient []) 1) ; ❷
;; ClassCastException
;; clojure.lang.PersistentVector$TransientVector
;; cannot be cast to clojure.lang.IPersistentCol
```

❶ `transient` called on a vector, returns an instance of the `clojure.lang.PersistentVector$TransientVector` inner class.
❷ The attempt to add 1 to the collection fails, because the transient does not support the `IPersistentCollection` interface.

While a subset of read-only functions like get, nth or count still works, an entire new set of functions is available to mutate the transient. Their name is like other standard functions with the conventional "!" added at the end:

```
(def v (transient []))
(def s (transient #{}))
(def m (transient {}))

((conj! v 0) 0) ; ❶
;; 0

((conj! s 0) 0) ; ❷
;; 0

((assoc! m :a 0) :a) ; ❸
;; 0
```

❶A transient vector "v" is mutated with `conj!` and used as a function to access the item at index "0". The item was just added to the transient and is the number "0".
❷Similarly, we can add a new element to a transient set using `conj!`. We can use the transient set as a function to verify if the set contains the item "0". When the element is not present, `nil` is returned.
❸The transient map "m" is mutated with `assoc!`.

The standard library implements many fundamental functions on top of transients: into, mapv, group-by, Set, frequencies (and more), are functions transforming a collection into another using `transient` to speed up internal processing. For example, here's the impact of `transient` on `frequencies`:

```
(require '[criterium.core :refer [quick-bench]])

(defn frequencies* [coll] ; ❶
  (reduce
    (fn [counts x]
      (assoc counts x (inc (get counts x 0)))) {}

(let [coll (range 10000)]
  (quick-bench (frequencies* coll)))
;; Execution time mean : 2.335759 ms
```

```
  (let [coll (range 10000)] ; ❷
    (quick-bench (frequencies coll)))
  ;; Execution time mean : 1.995676 ms
```

❶The `frequency*` function is the same as the one in the standard library, but it doesn't use `transient`.

❷We can see that the original frequencies is roughly 20% faster with `transient`.

There are other functions in the standard library that would benefit from `transient`. The book illustrated the use of `transient` to improve the performance of: zipmap, merge, tree-seq, disj and select-keys. We also used `transient` in:

- peek to create a `reverse-mapv` function.
- dotimes contain a fast "FizzBuzz" implementation using `transient`.
- nth shows how to implement a hash-table data structure on top of arrays. The hash-table grows and shrinks the internal array using faster `transient` operations.

The reader is invited to visit the above examples to see `transient` in action. For the remaining of the section, we are going to focus on a few important details. We need to be careful

about reusing the same reference to the `transient`, as it might become inconsistent. This is in contrast with other mutable data structures such as `java.util.HashMap`. The following example illustrate this behavior:

```
(import 'java.util.HashMap)

(def transient-map (transient {}))
(def java-map (HashMap.))

(dotimes [i 20] ; ❶
  (assoc! transient-map i i)
  (.put java-map i i))

(persistent! transient-map) ; ❷
;; {0 0, 1 1, 2 2, 3 3, 4 4, 5 5, 6 6, 7 7}

(into {} java-map) ; ❸
;; {0 0, 7 7, 1 1, 4 4, 15 15, 13 13, 6 6,
;;  3 3, 12 12, 2 2, 19 19, 11 11, 9 9, 5 5,
;;  14 14, 16 16, 10 10, 18 18, 8 8, 17 17}
```

❶dotimes iterates the body of the expression 20 times. Each iteration we add the key-value pair `[i i]` to both mutable maps.
❷The `persistent!` version of the `transient` map appears to be missing many keys.
❸The Java `HashMap` correctly shows 20 key-value pairs as expected.

Clojure `transient` doesn't make any promise that it will mutate the input in place. Each transient operation should instead consider the input "obsolete" and use the new version returned by the mutating function. The correct approach with `transient` should always be to use the output of the previous mutating operation:

```
(def transient-map (transient {}))

(def m ; ❶
  (reduce
    (fn [m k] (assoc! m k k))
    transient-map
    (range 20)))

(persistent! m) ; ❷
;; {0 0, 7 7, 1 1, 4 4, 15 15, 13 13, 6 6,
```

```
  ;;  3 3, 12 12, 2 2, 19 19, 11 11, 9 9, 5 5,
  ;;  14 14, 16 16, 10 10, 18 18, 8 8, 17 17}
```

❶The correct approach to apply multiple mutations to a `transient` is to always use the last mutated instance.

❷ `transient-map` is pointing to a different state of the same mutable `transient` map than the result of the reduce call. We can see that the `persistent!` version now contains the expected 20 keys.

The reader should also remember that transients are unsynchronized. Use of same `transient` instance by multiple threads can lead to unpredictable results. See locking for a way to ensure `transient` mutations happen in a synchronized context.

20.2 DOSEQ, DORUN, RUN!, DOALL, DO

Note

see also dotimes and while, two iterative functions designed for side effects that received individual treatment in a separate chapter.

All functions in this section assumes a computation that is dependent on some side effects to evaluate properly. The vast majority of functions in the standard library instead, because of laziness or general design principles, don't take side effects into proper consideration.

`doseq`, `dorun` and `run!` for instance, are designed exclusively for side effects. They walk a lazy sequence throwing away the results and returning `nil`:

```
(defn unchunked [n] ; ❶
  (map #(do (print ".") %)
    (subvec (vec (range n)) 0 n)))

(doseq [x (unchunked 10) :while (< x 5)] x) ; ❷
;; .....nil

(dorun 5 (unchunked 10)) ; ❸
;; ......nil

(run! #(do (print "!") %) (unchunked 10)) ; ❹
;; .!.!.!.!.!.!.!.!.!.!nil
```

❶ `unchunked` creates a non-chunked lazy sequence of size "n". By removing chunked evaluation, we make sure to evaluate exactly what is requested. With chunked sequences, we would always see the evaluation of the first 32 items independently from how many are requested. subvec is one of the few collections supported by seq that is not chunked and we can use it to create a new unchunked collection.

❷ `doseq` is fully equipped with a rich semantic to iterate the input, the same available with for. Shown here is the `:while` keyword that stops the iteration when "x" becomes equal to 5. The reader should review for to see all other options available. We can see a dot printed each input element passing through the map function, but the expression itself returns `nil`.

❸ `dorun` is simpler and offers less configuration. It accepts an optional number as first argument that represents how many elements to iterate. When there is no number, `dorun` iterates the entire input. The result is the same as `doseq`.

❹ `run!` additionally takes a function and applies the function to each element in the input. `run!` always runs through the entire input.

As we can see from examples, `doseq`, `dorun` and `run!` always return `nil`, independently from the input, so if anything interesting happens while iterating, it must

necessarily be a side effect. As a consequence, `doseq`, `dorun` and `run!` are all `O(1)` memory allocation as they don't retain the head (or any other item) of the sequence.

`doall` is similar in behavior to `dorun` but it returns the output forcing any side effects in the process. `doall` is often used to fully realize a lazy sequence before leaving a state that is necessary for the sequence to work properly. A typical example is with-open:

```
(require '[clojure.java.io :refer [reader]])

(defn get-lines [url] ; ❶
  (with-open [r (reader url)]
    (line-seq r)))

(def lines (get-lines "https://tinyurl.com/pi-di

(count lines) ; ❸
;; IOException Stream closed
```

❶ `get-lines` accepts an url and uses `clojure.java.io/reader` to produce the line-seq of the content of the url. with-open is a macro that ensure the `reader` is closed properly after exiting the block.
❷ No problem to read some text from the internet, apparently.
❸ But as soon as we try to move forward in the sequence, for example to check how many lines there are, we get an `IOException`.

What happens in the example above is that the `java.io.Reader` instance from where the lazy sequence is supposed to read is closed by the time the related code evaluates. with-open, as expected, took care of that aspects inside the function leaving the lazy sequence free to escape the context where the connection was still open. One solution is to fully realize the sequence:

```
(require '[clojure.java.io :refer [reader]])

(defn get-lines [url] ; ❶
  (with-open [r (reader url)]
    (doall (line-seq r))))

(def lines (get-lines "https://tinyurl.com/pi-di
```

```
  (count lines) ; ❷
  ;; 29301
```

❶The only change to `get-lines` was to add `doall` to the lazy sequence produced by `lazy-seq` before leaving the `with-open` context.

❷Counting the lines does not produce any problem once the sequence is fully realized.

Full evaluation with `doall` comes at the price of loading potentially large data into memory. When such approach is impractical, the computation would need to move inside the context where the side effects (an open connection for example) is still possible. If we want to count the lines of a large remote file for example, we would need to move count inside with-open:

```
(require '[clojure.java.io :refer [reader]])

(defn count-lines [url] ; ❶
  (with-open [r (reader url)]
    (count (line-seq r))))
```

```
(count-lines "https://tinyurl.com/pi-digits")
;; 29301
```

❶ Instead of just fetching the lines from the remote url, we now count them at the same time with `count-lines`. This function is less general, but doesn't need to load the potentially large file into memory for later use.

The last special form of this section is `do`. `do` evaluates all expression arguments and returns the result of the last (or `nil` in case of no expressions). The expressions preceding the last are presumably side effects, as their result is completely ignored:

```
(do
  (println "hello") ; ❶
  (+ 1 1))
;; hello
;; 2

(if (some even? [1 2 3])
  (do
    (println "found some even number") ; ❷
    (apply + [1 2 3]))
  (println "there was no even number."))
```

```
  ;; found some even number
  ;; 6
```

❶This form of `do` expression is often introduced temporarily to print a debug message in a critical section of the code. Other tracing techniques, such as logging, are more likely to be left in the code inside the `do` expression.

❷Another typical use of `do` is with if statements, as their "then-else" blocks only accept a single expression.

Functions designed for side effects such as the one we visited in this section, should not appear frequently in every day code and be constrained to specific parts of the application. Their presence should signal the need to produce side effects that are necessary for the computation to execute correctly. Clojure pragmatic approach to functional programming is realized in practice by embedding such awareness in the design of the standard library. Other languages prefer to wrap effectul programming in more constrained abstractions (for example Monads or Applicative Functors [256]) which offer additional control at the price of a sharp rise in complexity. Each approach has its advantages, with Clojure leaning toward simplicity and conventions.

[256] See the following paper by Conor McBride: www.staff.city.ac.uk/%7Eross/papers/Applicative.html

20.3 VOLATILE!, VRESET!, VSWAP! AND VOLATILE?

`volatile!` is a relatively recent addition to Clojure starting from version 1.7. Like atom, agent, var or ref, `volatile!` is a concurrency-related construct that stores some form of state. Unlike the standard reference types, `volatile!` is not thread-safe. `volatile!` is useful in a few specific scenarios where state changes should be immediately visible to other threads. In this sense, `volatile!` is more about side effects than concurrency. Using `volatile!` is pretty simple:

```
(def v (volatile! 0)) ; ❶

(volatile? v)         ; ❷
;; true

(vswap! v inc)        ; ❸
;; 1

(vreset! v 0)
;; 0
```

❶ `volatile!` is designed to hold data like other concurrency constructs. The initial value must be part of the call to `volatile!`.

❷ `volatile!` returns a `clojure.lang.Volatile` that is saved in the var "v". We can then ask if "v" is a `volatile?`.

❸ There are two ways to mutate the `volatile!`: we can use `vswap!` passing a function of the old value to the new, or use `vreset!` to just replace the value.

Java Volatile

`volatile!` does not protect its state against concurrent access like other concurrency primitives. On the opposite, the goal of `volatile!` is to make sure concurrent threads promptly see any state changes. To understand why this might be necessary, we need to talk about the default thread visibility of instance attributes in Java (or variables for short).

The Java Virtual Machine has many ways to speed up code execution. One of them is variable access reordering, which changes the order in which variables are read or written to improve the performance of a critical section of the code. In addition to reordering, the JVM might decide to cache variable values in CPU registries. When the code executes sequentially, reordering and local caches aren't normally a problem. But when multiple threads are involved, there is no guarantee that a read operation will execute after the expected write operation.

The problem could be solved by locking, but locking is costly [257]. However, many cases would be better solved by disallowing reordering and registers caching. In Java we can achieve such a goal declaring the variable "volatile". When an instance attribute is volatile, the compiler notifies the runtime that access to the variable

should not be subject to reordering and its value should not be stored in cache registries.

Clojure `volatile!` defines a reference type that implements such behavior. Before the introduction of `volatile!`, there was no way to produce a stand-alone "volatile" attributes with the exception of creating a whole deftype with the `:volatile-mutable` directive.

The main reason for the introduction of `volatile!` in Clojure are stateful transducers and their use in concurrent environments like core.async [258].

Some of the `core.async` constructs like "pipelines" achieve parallelism through coordinated multi-threaded access. In a pipeline, multiple threads take care of each stage of a transducer chain, but each thread makes access to only one stage at a time. It follows that a thread executing the next step of a stage needs to see any previously written state.

Under the conditions of a core.async pipeline, the state inside a stateful transducer doesn't need to be necessarily thread-safe. What's more important is that state should be immediately visible, not reordered and not cached in CPU registries. The reader is invited to take a look at transduce to see an example of custom stateful transducer that uses `volatile!`.

Other than stateful transducers, `volatile!` could be used to solve some thread coordination problems. In the following

example, a "producer" thread wants to communicate a consumer thread that work is finished and results are available:

```
(def ready (volatile! false))
(def result (volatile! nil))

(defn start-consumer []
  (future
    (while (not @ready) ; ❶
      (Thread/yield))
    (println "Consumer getting result:" @result)))

(defn start-producer [] ; ❷
  (future
    (vreset! result :done)
    (vreset! ready :done)))

(start-consumer) ; ❸
(start-producer) ; ❹
;; Consumer getting result: :done
```

❶ `start-consumer` implements a "spinning" loop that continuously check to see if results are ready. `Thread/yield` communicates the willingness of the current thread to give back control to the CPU for other threads to execute.

❷ `start-producer` starts a new future thread that delivers the result and change the `volatile!` "ready" from `false` to `true`.

❸ The spinning thread starts checking to see if there are results.

❹ As soon as the producer flips the "ready" flag, the consumer exits the loop and prints a message.

In plain Java, by forgetting to declare "ready" or "result" as "volatile", we could introduce inconsistent results such as printing "Consumer getting result: null" whereas in Clojure, there is no way to produce this kind of invalid results.

Before the introduction of `volatile!`, we could have used plain vars or atoms with equivalent valid results. `volatile!` however, achieves lightweight thread synchronization without internal locking (var) or compare and swap semantic (atom). As such `volatile!` should be considered primarily a performance optimization.

[257] See en.wikipedia.org/wiki/Lock_(computer_science) for an introduction to locking.
[258] core.async is a Clojure library implementing a form of concurrency called Concurrent Sequential Processes (CSP)

20.4 SET!

`set!` is a mutating macro that assigns a value to a mutable target. It works differently based on the type of argument. For example, `set!` can write into static or instance attributes of Java classes (assuming fields are "public" and not "final" [259]):

```
(import 'java.awt.Point) ; ❶

(def p (Point.)) ; ❷
[(. p x) (. p y)]
;; [0 0]

(set! (. p -x) 1) ; ❸
(set! (. p -y) 2)
[(. p x) (. p y)]
;; [1 2]
```

❶ `java.awt.Point` is a simple class from the Abstract Window Toolkit (AWT) in the Java standard library. It contains the public instance fields "x" and "y".

❷ A Point initializer with no arguments initializes the point at x=0, y=0 coordinates. We can see the coordinates using the getter method "getX" available on the point instance. Clojure automatically translates `(. p x)` into `p.getX()`.

❸ The dash sign "-" tells Clojure that "-x" is the instance attribute "x" and not the getter method "p.getX()". `set!` uses the attribute instance to set the new coordinates.

`set!` can also mutate thread-bound var objects (never the root value):

```
(def non-dynamic 1) ; ❶
(def ^:dynamic *dynamic* 1)

(set! non-dynamic 2) ; ❷
;; IllegalStateException

(set! *dynamic* 2)
;; IllegalStateException

(binding [*dynamic* 1] ; ❸
  (set! *dynamic* 2))
;; 2
```

❶ `non-dynamic` and `*dynamic*` are 2 vars. `*dynamic*` is declared dynamic and can accept locally bound values.

❷ `set!` cannot change the root binding of the `var` in both cases and throws exception.

❸ It can however change the locally bound value of the dynamic `var`.

`set!` is commonly seen to set locally bound variables such as `*warn-on-reflection*`, a special dynamic `var` that is implicitly thread-bound by the Clojure compiler. Since it is already thread-bound, `*warn-on-reflection*` doesn't require to be surrounded by binding, effectively looking like a global `var`:

```
(fn [x] (.toString x)) ; ❶
;; #object["user$eval1935$fn__1936@146c987a"]

(set! *warn-on-reflection* true) ; ❷

(fn [x] (.toString x)) ; ❸
;; Reflection warning - reference to field toStr
```

❶This anonymous function is using Java interop to call the `toString` method on the object "x". The call produces a Java object that is printed on screen.
❷Access to `*warn-on-reflection*` is granted to `set!` because `*warn-on-reflection*` is implicitly bound to the currently running thread.
❸After setting the `var`, any reflective call is promptly reported.

Another possible use of `set!` is for mutations inside deftype definitions:

```
(deftype Counter [^:unsynchronized-mutable cnt]
  clojure.lang.IFn
  (invoke [this] (set! cnt (inc cnt)))) ; ❷

(def counter (->Counter 0)) ; ❸

(counter)
;; 1

(counter)
;; 2
```

❶ `deftype` is one of the few options in Clojure to create concurrently "unsafe" objects. `Counter` defines a class with a private field "cnt".

❷ Functions defined as part of the interface of the class can `set!` the private field. In this case we make the counter "callable" by implementing the `clojure.lang.IFn` interface.

❸ After creating a new counter instance, we can increment its content by invoking the object without arguments.

`set!` is quite rare function to see in production applications. The presence of `set!` could signal the presence of unsafe mutable objects for which Clojure has plently of effective alternatives.

[259] The following tutorial contains a good summary of Java field visibility options docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html.

20.5 SUMMARY

Clojure does not have a long of functions dedicated to side effects or mutation. But when side effects are necessary, the standard library methodically regulates their use. Users are still free to create a mutable mess if they want, but Clojure steadily

pushes applications toward functional purity without being prescriptive.

21 JAVA INTEROPERABILITY

The functions and macros in this chapter are predominantly related to Java interoperability (or simply "interop"), the part of the standard library that sits closer to the Java runtime. Despite dealing with a state-oriented world, Clojure offers a functional syntax that is concise and readable. The following is a summary of what is described in the chapter:

- "." (also referred to as "dot" but used as the single char "." in code) is a versatile special form to access Java methods and attributes. Beside the basic form, `dot` has several variants to help maintaining a readable syntax. ".." (double-dot) is useful to assemble chained calls while "doto" concatenates side-effecting calls.
- new creates a new instance of an object.
- try and catch (and related special forms finally and throw) are the fundamental mechanism for exception handling in Clojure.
- ex-info and ex-data builds on top of Java exception mechanism by hiding some of the low level syntax and offering a mechanism to transport data with exceptions.

- bean is a macro that wraps Java objects in map-like interfaces.
- The functions in the clojure.reflect namespace perform introspection on the structure and content of Java classes.
- Clojure also has a rich set of functions to deal with Java arrays. make-array is just one of the many ways to create an array from Clojure. Macros like aset, amap or areduce are useful to transform the content of the array with a familiar functional interface. The section also contains a brief explanation of array macro variants to leverage primitive types (`int` , `boolean` , `byte` and so on).

21.1 ".", ".." AND DOTO

“. (dot)” is a multipurpose special form to access methods and attributes of Java objects. Several variants exist, with the target object appearing as the first argument ("forward form"), or as the second ("inverted form"), or abbreviated with a slash "/" ("slashed" form). The meaning of the expression also changes based on the type of the arguments. Here’s is an exhaustive list of all the available variants:

```
; ❶
(. Thread sleep 1000)   ;; Access static method (
(. Math random)         ;; Access static field f
(. Math (random))       ;; Access static method (
(. Math -PI)            ;; Access static field a
(. Thread$State NEW)    ;; Access inner class sta

; ❷
(Thread/sleep 1000)     ;; Access static method (
(Math/random)           ;; Access static field f
(Math/-PI)              ;; Access static field (

; ❸
(def point (java.awt.Point. 1 2))
(. point x)      ;; Access instance field first
(.x point)       ;; Same as above.
(. point (getX)) ;; Access instance method (unam
(.-x point)      ;; Access instance field (unamb
```

❶The first group of dotted forms shows how to use them to access static members of Java classes. A Java class could declare a static field and a static method of no arguments with the same name. While in Java this generates no ambiguity, Clojure needs a way to distinguish between the two options. If not specified, it tries to access the static field first and the static method of no arguments next. To force static field access only, we can prefix the field name with a dash "-". To force exclusive static method access, we can use a pair of parenthesis.

❷The "slashed" form expands into the corresponding "dotted" form. It is shorter and easier to read, so it is the preferred choice when possible.

❸The last group shows how to use "." to access instance members. Similarly to static members, a dash "-" prefix force exclusive access to the instance field, even when an instance method of no arguments with the same name exists. Parenthesis requests unambiguous access to the instance method.

“. (dot)” behaves like a macro in terms of arguments evaluation. If we need to call a method but the method name is not known at compilation time, we need to assemble the dotted form at <<defmacro,macro> expansion time:

```
(defmacro getter [object field] ; ❶
  (let [getName# (symbol (str "get" field))] ; ❷
    `(. ~object ~getName#)))

(defmacro setter [object field & values] ; ❸
  (let [setName# (symbol (str "set" field))]
    `(. ~object ~setName# ~@values)))

(getter (java.awt.Point. 2 2) "X") ; ❹
;; 2.0

(doto  ; ❺
  (java.awt.Point. 2 2)
  (setter "Location" 4 5))
;; #object[java.awt.Point 0x64d15330 "java.awt.P
```

❶In this example, we want to assemble the name `getX()` passing the field name "x" as argument.
❷The `getter` method name is assembled in the let block as a symbol.
❸Here's an equivalent `setter` method to set attributes on an object instance.
❹We can see how to use `getter` to access the "x" coordinate.
❺The `Point` object has a `setLocation(x, y)` method taking 2 arguments, the "x" and "y" coordinates. We use `doto` (see below) to return the `Point` instance after setting the new location (side effect).

Double dot ".." uses the result of invoking the first two arguments as the input for the next pair, and so on, until there are no more arguments. ".." is useful to connect a chain of state-mutating calls so they always operate on the initial object. A common case, is building instances using the Java builder pattern:

```
(import '[java.util Calendar Calendar$Builder])

(.. (Calendar$Builder.) ; ❶
    (setCalendarType "iso8601")
    (setWeekDate 2019 4 (Calendar/SATURDAY))
    (build))
```

```
;; #inst "2019-01-26T00:00:00.000+00:00"

(macroexpand ; ❷
  '(.. (Calendar$Builder.)
   (setCalendarType "iso8601")
   (setWeekDate 2019 4 (Calendar/SATURDAY))
   (build)))

;; (. (. (. (Calendar$Builder.) (setCalendarType
;;       (setWeekDate 2019 4 (Calendar/SATURDAY)
```

❶The Calendar API in the Java standard library supports creation of arbitrarily complex calendar instances through the builder pattern. The builder is created as the first instruction and the result of each invocation is passed down to the next instruction up to the final `build` call that returns the assembled calendar. The returned calendar instance shows the date of the fourth Saturday of 2019.

❷The macroexpansion of the same form shows the much less readable dotted form necessary to execute the same instructions.

The last macro of this section is `doto`. `doto` resembles the double dot ".." macro on the surface, but instead of passing the results of the previous invocation, the target object remains the

same through all the steps of the chain. This is useful for repeated changes to mutable objects, such as Java collections:

```
(import 'java.util.ArrayList)
(def l (ArrayList.))

(doto l ; ❶
  (.add "fee")
  (.add "fi")
  (.add "fo")
  (.add "fum"))
;; ["fee" "fi" "fo" "fum"]
```

❶The `ArrayList` is repeatedly added elements without the need to interleave "l" through all the forms.

21.2 NEW

`new` is a special form (for all practical purposes, a macro) that takes a variable number of arguments: the first argument must be a symbol that resolves to a Java class and the rest of the arguments are optional. The additional arguments, if any, are passed directly to the constructor of the class:

```
(def sb (new StringBuffer "init")) ; ❶
```

```
  (.append sb " item")

  (str sb) ; ❷
  ;; "init item"
```

❶A `java.lang.StringBuffer` is a mutable string with an option to append new fragments at the end of an empty or given string.

❷Once the string is assembled correctly, it can be easily transformed back into an immutable string.

Constructors definitions for a class can come from the class itself or any superclass (constructors in the child class are allowed to extend the corresponding constructors from the superclass with additional behavior). In order to find the right constructor, Clojure goes through a process of pattern matching against the number of arguments and their types. Because of potential type inheritance on arguments, there could be multiple matching constructors for the same invocation. One subtle example is the following involving `java.lang.BigDecimal`:

```
  (let [l (Long. 1)] (new BigDecimal l)) ❶
  ;; java.lang.IllegalArgumentException: More than
```

❶An example of ambiguous constructor call for the `BigDecimal` class.

`BigDecimal` does not have a constructor with a `java.lang.Long` parameter but it does contain one for primitive `int` or `long`. The problem from Clojure's perspective is that both are possible as the boxed type `Long` could downcast to primitive `int` or `long` without loss of precision. Passing a primitive `long` solves the problem:

```
(let [l 1] (new BigDecimal l)) ❶
;; 1M
```

❶In this example, `new` receives a request to create e new `BigDecimal` with a primitive `long` "l". `BigDecimal` contains an explicit constructor for `long` and there is no ambiguity on which one to use.

`new` has a shorter form that is normally recommended. It removes the need to use the `new` keyword which is replaced by a "." (dot) after the name of the class:

```
(StringBuffer. "init") ; ❶
;; #object[java.lang.StringBuffer 0x5fa1cc83 "in
```

```
  (macroexpand '(StringBuffer. "init")) ; ❷
  ;; (new StringBuffer "init")
```

❶ `new` has a shorter form that removes the need of using the keyword `new` itself.

❷We can verify that the dotted form expands into calling `new`.

21.3 TRY, CATCH, FINALLY AND THROW

`try`, `catch`, `finally` and `throw` are special forms found in any non-trivial Clojure application. They are used to handle special conditions called "exceptions" where the program alters its normal control flow.

Let's talk about `throw` first, as it comes handy to illustrate the other forms in this section. `throw` expects a single argument of type `java.lang.Throwable` or any subclass (most notably, `java.lang.Exception` and `java.lang.Error` are both subclasses of `java.lang.Throwable`):

```
  (throw (Throwable. "'there was a problem.'")) ;
  ;; Throwable 'there was a problem.' ; ❷

  (clojure.repl/pst) ; ❸
  ;; Throwable : there was a problem.
```

```
  ;;   user/eval1927 (form-init5670973898278733609
  ;;   user/eval1927 (form-init5670973898278733609
  ;;   clojure.lang.Compiler.eval (Compiler.java:6
  ;;   [....]
```

❶There is no need to import `java.lang.Throwable` because many classes in `java.lang.*` are available by default.

❷ `Throwable` (and by extension all other exception classes) supports a constructor with a string that is normally used to describe the reason for the exception.

❸Depending on the REPL and JVM settings, throwing an exception also shows the stack trace at the point the exception was generated. If it's not showing it by default, you can use `clojure.repl/pst` (Print Stack Trace) to show that on the screen.

`try` takes a body expression (automatically wrapped in a do block) and any number of `catch` clauses. The `catch` clause declares a type, a symbol and an expression. If there is an exception in the outer `try` and a compatible `catch` clause, the corresponding expression evaluates:

```
(try
  (println "Program running as expected")      ;
  (throw (RuntimeException. "Got a problem.")) ;
```

```
    (println "program never reaches this line")  ;
    (catch Exception e                            ;
      (println "Could not run properly" e)        ;
      "returning home"))                          ;

  ;; Program running as expected
  ;; Could not run properly. #error {
  ;;  :cause Got a problem.
  ;;  :via
  ;;  [{:type java.lang.RuntimeException
  ;;    :message Got a problem.
  ;;    :at [user$eval1933 invokeStatic form-init5
  ;;  :trace
  ;;  [...]}
  ;; "returning home"
```

❶ `try` supports any number of top level expressions, similarly to the do block.
❷ We `throw` an exception on purpose. This alters the normal program flow, for example skipping lines that would be normally evaluated.
❸ This line never prints because it follows a `throw` instruction.
❹ The `catch` directive declares `java.lang.Exception` as the type of exception class that is able to handle. The clause is unable to catch more general errors such as `java.lang.Throwable` or `java.lang.Error`.
❺ `catch` contains an implicit do block. The local binding "e" is bound to the instance of exception that was captured.
❻ After printing a message, the catch block returns the evaluation of the last expression.

The rule of evaluation is based on inheritance: a clause matches if the exception type is the same, or a subclass, of the declared `catch` type. This allows for "cascading" style catching from more specific types to the most general:

```
(import '[java.net
          Socket InetAddress
          ConnectException SocketException])

(try
```

```
    (Socket. (InetAddress/getByName "localhost") 6
    (catch ConnectException ce ; ❷
      (println "Could not connect. Retry." ce))
    (catch SocketException se ; ❸
      (println "Communication error" se))
    (catch Exception e ; ❹
      (println "Something weird happened." e)
      (throw e)))
  ;; nil
```

❶Our application needs to connect to some local port for communication. Several things can go wrong, for example the other party might not be ready and still preparing to listen the given port, or we might be able to connect and then unable to communicate.

❷If we are unable to connect, it means that for some reason the other application is not listening. It could be temporary and it could be worth implementing a "retry" mechanism such as the `with-backoff` macro presented in the macro chapter. Note that we put the exception object in the printout as part of the println instruction. If we weren't, we could be throwing away important information stored in the exception (this is also called "swallowing exceptions").

❸After a successful connection to the socket, there could be unexpected problem related to the packets sent through the channel. We can handle this condition separately, perhaps retrying the connection again.

❹Finally, the general `Exception` clause catches any other condition. Note that instead of just printing a message, we also re-throw the same exception. Re-throwing signals that this block of code is unable to handle the exception but allows other blocks upstream to handle it.

`finally` is a special `catch` clause that is always evaluated, even when there is no exceptional condition. The `try-`

`finally` pattern is pretty popular to handle resources that must always be released, even in case of exception. with-open is an excellent example of this behavior and it uses `finally` to make sure any resource open as part of the bindings is correctly closed. Similarly locking uses `finally` to always release the lock after executing a critical region. In the following example, we build a simplified `with-open` to deal with `java.io.Reader` objects only:

```
(require '[clojure.java.io :as io])

(defmacro with-reader [r file & body] ; ❶
  `(let [~r (io/reader ~file)]
     (try
       ~@body ; ❷
     (finally ; ❸
       (.close ~r)))))

(with-reader r "/etc/hosts"
  (last (line-seq r)))
;; ::1
```

❶ The macro `with-reader` binds the first parameter to an open `java.io.Reader` instance using a file as input source.
❷ When "body" evaluates, the parameter "r" is bound to an open reader.
❸ `finally` guarantees that the reader instance is closed correctly, even in case of exception while evaluating the body.

`finally` can be used alone or as the final statement after a group of `catch` clauses. The entire expression returns the evaluation of the matching `catch` but the `finally` block always executes for side effects:

```
(try
  (/ 1 0)
  (catch Exception e "Returning from catch") ; ❶
  (finally (println "Also executing finally")))
;; Also executing finally
;; "Returning from catch"
```

❶The result of the entire expression is the string evaluated in the `catch` block which handles the exception generated by dividing 1 by 0.

❷The `finally` block evaluates regardless of a catch block handling the exception, producing the message in output.

21.4 EX-INFO AND EX-DATA

`ex-info` and `ex-data` (this section also mentions `Throwable→map`) are relatively new additions to the standard library making their first appearance in Clojure 1.4 (and with `Throwable→map` even later in 1.7).

`ex-info` and `ex-data` main purpose is to decorate Java exceptions with metadata, enriching the amount of information they can carry over exceptional situations. They do so by introducing `clojure.lang.ExceptionInfo`, a new type of exception that also accepts a map during construction:

```
(def ex
  (ex-info "Temperature drop!" ; ❶
    {:time "10:29pm"
     :reason "Front door open."
     :mitigation #(println "close the door")}))

(type ex) ; ❷
```

```
  ;; clojure.lang.ExceptionInfo

  ex ; ❸

  ;; #error {
  ;;  :cause "Temperature drop!"
  ;;  :data {:time "10:29pm", :reason "Front door
  ;;         :mitigation #object[user$fn__1569 0x1
  ;;  :via
  ;;  [{:type clojure.lang.ExceptionInfo
  ;;    :message "Temperature drop!"
  ;;    :data {:time "10:29pm", :reason "Front doo
  ;;           :mitigation #object[user$fn__1569 0
  ;;    :at [clojure.core$ex_info invokeStatic "co
  ;;  :trace
  ;;  [[clojure.core$ex_info invokeStatic "core.cl
  ;;   [...]
  ;;   [java.lang.Thread run "Thread.java" 748]]}
```

❶ `ex-info` works as a constructor for the exception type `clojure.lang.ExceptionInfo`. Along with a message it accepts a map that can contain any sort of information.

❷The type of `ex` is `ExceptionInfo`.

❸The exception object renders itself as a record-like object if we try to print it at the REPL.

`ex-info` hides any Java interoperability detail necessary to create the exception object. The only information required is a message and the metadata map. It optionally accepts a third argument to capture or re-throw a root-cause exception:

```
(try
  (/ 1 0)
  (catch Exception e ; ❶
    (throw
      (ex-info "Don't do this."
        {:type "Math"
         :recoverable? false} e)))) ; ❷

;; ArithmeticException Divide by zero
```

❶ `catch` captures an exception after evaluating some code.
❷ `ex-info` wraps around "e" and re-throw a new exception decorated with additional metadata.

By default metadata are hidden and don't appear when printing the `ExceptionInfo` instance created with `ex-info`. To see and use the metadata, we can use `ex-data`:

```
(defn randomly-recoverable-operation [] ; ❶
  (throw
    (ex-info "Weak connection."
```

```
      {:type :connection
       :recoverable? (< 0.3 (rand))})))

(defn main-program-loop []
  (try
    (println "Attempting operation...")
    (randomly-recoverable-operation)
    (catch Exception e
      (let [{:keys [type recoverable?]} (ex-data
        (if (and (= :connection type) recoverable
          (main-program-loop)
          (ex-info "Not recoverable problem."
            {:type :connection} e))))))

(main-program-loop) ; ❸

;; Attempting operation...
;; Attempting operation...
;; #error {
;;  :cause "Weak connection."
;;  :data {:type :connection, :recoverable? false
;;  :via
;;  [{:type clojure.lang.ExceptionInfo
;;    :message "Not recoverable problem."
;;    :data {:type :connection}
;;    :at [clojure.core$ex_info invokeStatic "co
;;   {:type clojure.lang.ExceptionInfo
;;    :message "Problem."
```

```
  ;;    :data {:type :connection, :recoverable? fa
  ;;    :at [clojure.core$ex_info invokeStatic "co
```

❶ `randomly-recoverable-operation` simulates a function that frequently fails but sometimes the operation can be repeated. The function applies a "threshold" (simulated using rand) to decide if the problem is recoverable.

❷ `main-program-loop` is the client using the randomly recoverable function. After catching a potential exception, we extract metadata information using `ex-data`. In case the exception is recoverable, the `main-program-loop` evaluates again.

❸ After a couple of attempts, the `main-program-loop` exits as the error is not recoverable.

From the example above we can see that nested `ex-info` exceptions add up nicely when examining the stack trace: we can see a "Not recoverable problem." caused by a "Weak connection." and so on for all the nested exceptions.

The last function in this section is `Throwable→map`, an useful function to transform the fragmented information inside a hierarchy of exceptions into a nice Clojure data structure:

```
(def error-data ; ❶
  (try (throw (ex-info "inner" {:recoverable? fa
    (catch Throwable t
      (try (throw (ex-info "outer" {:recoverable
        (catch Throwable t
          (Throwable->map t))))))

(keys error-data) ; ❷
;; (:cause :via :trace :data)

(:cause error-data) ; ❸
;; "inner"

(:via error-data) ; ❹
;; [{:type clojure.lang.ExceptionInfo,
;;   :message "outer",
;;   :at [clojure.core$ex_info invokeStatic "cor
;;   :data {:recoverable? false}}
;;  {:type clojure.lang.ExceptionInfo,
;;   :message "inner",
;;   :at [clojure.core$ex_info invokeStatic "cor
;;   :data {:recoverable? false}}]

(nth (:trace error-data) 3) ; ❺
;; [user$fn__2151 invoke "form-init5670973898278
```

```
  (:data error-data) ; ❻
  ;; {:recoverable? false}
```

❶ `error-data` simulates two nested exceptions for illustrative purposes. `Throwable→map` converts the last `java.lang.Throwable` instance to a Clojure map.

❷ `error-data` has 4 keys containing different aspects describing the error.

❸ The `:cause` key shows the root cause error description, in this case `inner`.

❹ `:via` is a vector of maps. Each map contains the basic data of each exception instance. Each `:data` key contains metadata if the exception was created with `ex-info`.

❺ The `:trace` key is another vector containing one entry for each entry in the stack trace of the exception.

❻ Finally, `:data` contains the initial map that was used to create the exception with `ex-info`.

21.5 BEAN

`bean` is a function that creates a map-like representation of the attributes available in an object:

```
  (import 'java.awt.Point)
```

```
(def point (bean (Point. 2 4))) ; ❶
(keys point) ; ❷
[(:x point) (:y point)] ; ❸
;; [2.0 4.0]
```

❶ `java.awt.Point` is a simple class in AWT, the original (and basic) Java Abstract Windowing Toolkit. It exposes x,y coordinates as public attributes, including the methods `getX()` and `getY()` to access them.

❷ `bean` creates a proxy instance implementing Clojure's map-like features, including keys for public attributes.

❸ We can now use map-like keys to access the attributes of the original class.

`bean` uses introspection to analyze the content of a Java class through the JavaBean programming interface (see the call-out below for additional information on JavaBeans). It follows that not all available attributes are visible, but just those exposed through the JavaBean standard. Compare for example the following:

```
(bean (Object.)) ; ❶
;; {:class java.lang.Object}

(import 'javax.swing.JButton)
```

```
  (pprint (bean (JButton.))) ; ❷
  ;; {:y 0,
  ;;  :selectedObjects nil,
  ;;  :componentPopupMenu nil,
  ;;  :focusable true,
  ;;  :managingFocus false,
  ;;  :validateRoot false,
  ;;  :requestFocusEnabled true,
  ;;  :containerListeners [],
  ;;  :rolloverSelectedIcon nil,
  ;;  :iconTextGap 4,
  ;;  :mnemonic 0,
  ;;  :debugGraphicsOptions 0,
  ;;  [...]
```

❶ `bean` can process any type of object, but only those using the JavaBean conventions typically provide more useful information. Here we use `bean` on a basic Object instance which only contains the default `:class` key.

❷ `javax.swing.JButton` contains many getter methods (those starting with "get" and then the name of a property) that `bean` can use to extract a large map of key-property values.

What is a JavaBean?

Introspection (also called reflection) is a Java feature to access a class structure at runtime, for example listing methods, constructors or attributes. With reflection, a Java program can query the structure of Java classes, or even invoking constructors to create new objects. Java took introspection to the next level after introducing JavaBeans in 1996, not too long after Java was born.

The JavaBean standard specifies a set of conventions that a class subscribing to the JavaBean specification should obey [260]:

- All attributes should be private but accessible through getters/setters methods.
- It should provide a public, no-argument constructor.
- It should implement the `java.io.Serializable` interface.

Provided with this set of conventions, tools can find, analyze, instantiate and call methods on JavaBeans. Tools can also store a JavaBean as bytes or send them through the network. JavaBeans are popular with graphical interfaces, where they offer automatic discovery capabilities.

`bean` is an useful tool to keep in mind for quick conversions of Java bean-like objects into Clojure maps. However, because of its heavy reflection use, you should avoid `bean` in critical parts of the code where speed is important. For example, here's the difference between using `bean` or using a custom function to create a map out of a point instance:

```
(require '[criterium.core :refer [quick-bench]])
(import 'java.awt.Point)
```

```
(let [p (Point. 2 4)]
  (quick-bench
    {:class java.awt.Point
     :location (.getLocation p)
     :x (.getX p)
     :y (.getY p)}))

;; Execution time mean : 16.101893 ns

(let [p (Point. 2 4)]
  (quick-bench (bean p)))

;; Execution time mean : 1.229609 µs
```

The difference between using `bean` or rolling out a custom map is roughly three orders of magnitude.

[260] The JavaBean specification is available from www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

21.6 REFLECT AND TYPE-REFLECT

`clojure.reflect/reflect` performs introspection on a Java type returning useful information such as base classes, constructors, methods, signatures and return types. While

`reflect` works both with objects or classes, `type-reflect` only works on classes but provides the same functionality as `reflect` (indeed `reflect` is a thin wrapper on top of `type-reflect`). For the remainder of this section, everything we say about `reflect` is also valid for `type-reflect` unless otherwise specified. Here's a simple example on how to use `reflect`:

```
(require '[clojure.reflect :as r]) ; ❶

(keys (r/reflect {})) ; ❷
(keys (r/reflect clojure.lang.APersistentMap)) ;

;; (:bases :flags :members) ; ❹
```

❶All functions in this sections belong to the `clojure.reflect` namespace.
❷ `reflect` accepts an object as argument. It will convert the object into the related class internally.
❸In this example, `reflect` uses a class directly.
❹Both calls to `reflect` return a Clojure map with 3 keys.

The map returned by `reflect` contains 3 keys:

- `:bases` shows all direct super-classes or implemented interfaces.
- `:flags` shows the modifiers of the class (such as `public`, `final` etc.).
- `:members` is a list of all public methods.

`reflect` also accepts the following options: an `:ancestors` key and a `:reflector` key. With `:ancestors`, `reflect` retrieves super-classes and super-interfaces at all levels, not just directly above the target class:

```
(pprint (:ancestors (r/reflect {} :ancestors tru

;; #{java.lang.Object clojure.lang.Associative
;;   java.util.concurrent.Callable java.util.Map
;;   java.lang.Runnable clojure.lang.IPersistent
;;   clojure.lang.IHashEq clojure.lang.IObj cloj
;;   clojure.lang.MapEquivalence clojure.lang.IK
;;   clojure.lang.Counted clojure.lang.IPersiste
;;   java.io.Serializable clojure.lang.AFn
;;   clojure.lang.IEditableCollection clojure.la
;;   clojure.lang.APersistentMap java.lang.Itera

(count (:members (r/reflect {}))) ; ❷
;; 37
```

```
(count (:members (r/reflect {} :ancestors true)))
;; 180
```

❶ When `:ancestors true` is present, `reflect` retrieves super-classes and super-interfaces beyond the first hierarchy level and collect the result in the additional `:ancestors` key.

❷ When `:ancestors true`, the `:members` key additionally includes methods declared in any of the ancestors. We can see the number going up from 37 to 180 collected members (methods or constructors).

The `:reflector` key allows to specify a different "Reflector". By default, `reflect` uses Java reflection (which requires to load the class into memory for inspection). Other libraries offering similar or better features than built-in Java reflection can be plugged into `reflect` using the `:reflector` key. Here's for example how to create a dummy reflector for illustration purposes:

```
(deftype StubReflector [] ; ❶
  r/Reflector
  (do-reflect [this typeref]
    {:bases #{} :flags #{} :members #{}}))
```

```
(r/reflect java.lang.Integer :reflector (StubRef
;; {:bases #{}, :flags #{}, :members #{}}
```

❶ `StubReflector` is a class implementing the `clojure.reflect/Reflector` protocol. The protocol requires to implement a `do-reflect` function taking a type reference. Our implementation just returns a map of empty values.

❷To use the newly created `StubReflector`, we pass a fresh instance to the `:reflector` option when calling `reflect`.

21.7 JAVA ARRAYS

Clojure provides a rich set of functions dedicated to Java arrays. Arrays are special type of objects initialized to contain a fixed amount of items. Arrays are not particularly sophisticated but they are very efficient. Clojure applications seeking for maximum performance often use arrays: fast math operations, image processing, byte streaming and so on.

Arrays are "indexed": each item is numbered starting at index 0. Items in the array must be of the same type. The only way to have items of different types is to declare the array of type `Object`. The following table summarizes the different functions available to create arrays and their main features:

Table 21.1. A summary of the main array-creating functions.

function	input values	size argument	multi-dimensional	custom types
`make-array`	no	yes	yes	yes
`object-array`	yes	yes	no	no
`to-array`	yes	no	no	no
`into-array`	yes	no	no	yes

Here's a brief explanation of the meaning of each column:

- **input values** shows if the function accepts an initial set of values for the array. When there are no initial values, Java initializes the array to a default value (for example 0 for int, `false` for boolean, "null" for reference types and so on). `make-array` is the only function in this set not allowing custom initialization with alternative values.
- **size argument** shows if the function accepts an integer to establish the size of the array. Functions like `to-array` or `into-array` offers that functionality implicitly by creating an array of the same size of the input collection.
- **multi-dimensional** is an array in which the type of an item is another array. There could be up to 255 dimensions. `make-array` is the only function that allows the creation of multi-dimensional arrays.
- **custom types** shows if the function can create arrays of a specific type (different than `Object` or other primitive types). For example: with `make-array` we could create an array of type `java.awt.Point`, while with `to-array` we could only create an array of type `Object` which contains `java.awt.Point`.

- **input collection** shows if the function can initialize the array with items found in another collection. `make-array` is the only function that does not allow collections as input.

In the following sections we are going to see how to create and process arrays from Clojure in more detail.

21.7.1 make-array

`make-array` creates a Java array of the requested type and length:

```
(def a (make-array Boolean/TYPE 3)) ; ❶

(vec a) ; ❷
;; [false false false]

(def b (make-array Boolean 3)) ; ❸

(vec b)
;; [nil nil nil]
```

❶ `Boolean/TYPE` is a handy shortcut to identify the primitive boolean type in Java.

❷ The array won't print its content without explicit iteration. A quick way to display its content is to transform it into a vector with vec.

❸ For all reference types (such as uppercase `Boolean`), the default initialization value is `nil`.

Any additional number after the last argument triggers the creation of a multi-dimensional array:

```
(def a (make-array Integer/TYPE 4 2)) ; ❶

(mapv vec a) ; ❷
;; [[0 0] [0 0] [0 0] [0 0]]
```

❶ The presence of 2 integers prompts `make-array` to create a two-dimensional array. The structure contains four arrays of two integers each.

❷ This time, to convert back into a vector, we need to call vec on each item of the initial array.

Warning

`make-array` has some (reasonable) limitations: the number of dimensions cannot exceed 255 and only zero or positive integers are accepted. In general, the requested array should fit into the available memory.

`make-array` is the most general function to create empty arrays (where empty means array initialized with the default value for the requested type). The need for arrays of primitive types is so common that an entire collection of array initializers has been dedicated to that task. We are going to see them in the following section.

21.7.2 object-array and other typed initializers

Note

This section also mentions other related functions such as: `int-array`, `boolean-array`, `byte-array`, `short-array`, `char-array`, `long-array`, `float-array` and `double-array`.

`object-array` (and other array initializers dedicated to primitive types such as `int`, `boolean`, `byte` and so on) creates an empty array of the specified type and size:

```
(vec (object-array 3)) ; ❶
;; [nil nil nil]

(vec (char-array 3)) ; ❷
;; [\ \ \]

(vec (double-array 3)) ; ❸
;; [0.0 0.0 0.0]
```

❶ `object-array` creates an array of the given size. Here the created array of type "Object" contains 3 `nil` elements.
❷ `char-array` initializes the array to the non-printable character corresponding to ASCII table index "0".
❸ `double-array` creates the array initialized to double type "0.0".

By passing a second parameter of compatible type, we can initialize the array with a specific value instead of the default provided by Java:

```
(vec (float-array 3 1.0)) ; ❶
```

```
;; [1.0 1.0 1.0]
```

❶When a second parameter is present and is compatible with the type of the created array, the value is used to initialize all the items in the array.

When the first parameter is a sequential collection instead of a number, the content of the input collection is copied into the newly created array:

```
(vec (int-array [1 2 3])) ; ❶
;; [1 2 3]
```

❶When `int-array` first argument is a sequential collection, an array of `int` is created which contains the items from the input.

The items in the input collection needs to be compatible with the type of the array. If they are not compatible (that is, there is no "cast" operation that transforms the input into the correct type) the compiler throws exception:

```
(vec (int-array [\a \b \c])) ; ❶
;; ClassCastException java.lang.Character cannot
```

❶ The items in the input collection needs to be compatible with the type declared by the array.

Some compatible cases are subtle, as Java tolerates precision loss for certain types:

```
(vec (int-array [4294967296])) ; ❶
;; [0]
```

❶ The input collection contains a large number ($2^{\overline{32}}$ in this case, which is beyond `int` capacity). Java truncates the most significant bit to fit the large number in the 32 available bits. The result of the truncation is the number "0".

Finally, we can pass both a size and an input collection. If the content of the collection is not sufficient to fill the resulting array, values fall back to the default:

```
(vec (int-array 5 (range 10))) ; ❶
;; [0 1 2 3 4]

(vec (boolean-array 5 [true true])) ; ❷
;; [true true false false false]
```

❶In this case, the resulting array can only fit 5 of the available 10 items from the input collection.
❷If there is not enough input, the rest of the items in the array is initialized with the default value for that primitive type.

In the next section we are going to see other ways to create arrays starting from an existing input collection.

21.7.3 to-array, into-array, to-array-2d

`to-array` is very similar to `object-array`, but uses a faster implementation to transform the input collection. The following benchmark compares `to-array` and `object-array` with an input vector. The output is the same, but `to-array` is twice as fast:

```
(require '[criterium.core :refer [bench]])

(let [v (vec (range 100))] (bench (to-array v)))
;; Execution time mean : 302.673403 ns

(let [v (vec (range 100))] (bench (object-array
;; Execution time mean : 609.085730 ns
```

❶ `to-array` performs twice as fast as `object-array` when the input collection is a vector.
❷ `object-array` is slower, but it's useful option to create an array of objects without the need to pass an input collection.

The speed-up is not guaranteed, but depends on the type of the input collection. When possible, `to-array` delegates the creation of the array to the input collection, while `object-array` always transforms the input into a sequence and then iterates its content. Prefer `to-array` to create an object array out of a collection, while `object-array` still offers the option to create an object array without any input.

`into-array` is similar to `to-array`, but it will try to guess or force a specific type for the output array:

```
(type (to-array [1 2 3])) ; ❶
;; [Ljava.lang.Object;

(type (into-array [1 2 3])) ; ❷
;; [Ljava.lang.Long;
```

❶ `to-array` always creates a new `Object` array independently from the type of the items in the input collection.
❷ `into-array` specializes the type of the output array choosing `java.lang.Long` instead of the more generic `java.lang.Object`.

`into-array` uses the type of the first element to guess an appropriate type for the output array. Once the type has been fixed all following elements needs to be of the same type:

```
(into-array [1 2 (short 3)]) ; ❶
;; IllegalArgumentException array element type m
```

❶ `into-array` does not accept mixed type arrays. This is true even for type-compatible casting such as `short` to `long`.

Alternatively, `into-array` accepts an explicit type parameter. In this case, `into-array` attempts coercion of the input only if this does not result in a loss of precision:

```
(def a (into-array Short/TYPE [1. 2 (short 3)]))

(type a) ; ❷
;; [S
```

```
(map type a) ; ❸
;; (java.lang.Short java.lang.Short java.lang.Sh

(into-array Short/TYPE [Integer/MAX_VALUE]) ; ❹
;; IllegalArgumentException Value out of range f
```

❶The input collection contains a mix of numeric types.

❷The first parameter requires an output array of type `short`. Here we can see what is the Java encoding for the type "array of shorts".

❸All input numbers have been coerced to short, as expected.

❹ `into-array` throws exception if we try to coerce a number too large to fit the 16 bits allocated for a `short` type.

`to-array-2d` is a handy shortcut to create a two-dimensional object array:

```
(def a (to-array-2d [[1 2] [3 4]])) ; ❶

(map type a) ; ❷
;; ([Ljava.lang.Object; [Ljava.lang.Object;)

(mapv vec a) ; ❸
;; [[1 2] [3 4]]
```

❶"a" is an array of type "array of objects" containing 2 items each.

❷We can verify that each element of the first array is itself another array by using map with type.

❸We can rebuild the initial input vector by using mapping with vec over the array.

21.7.4 aget, aset, alength and aclone

`aget` and `aset` are the basic operations to read and write an array item by index:

```
(def a (into-array [:a :b :c]))

(aget a 0) ; ❶
;; :a

(aset a 0 :changed) ; ❷
;; :changed

(vec a) ; ❸
;; [:changed :b :c]
```

❶We can use `aget` to access the element at index "0" from the array "a". `aget` returns the value of the array at that index.
❷With `aset` we can write a specific location in the array. `aset` returns the item that was just written.
❸Mutable arrays operate very differently from immutable Clojure collections: after writing the array location with `aset` the array has permanently changed.

If the array is multi-dimensional, `aget` and `aset` accepts additional indexes to access the nested arrays:

```
(def matrix (to-array-2d [[0 1 2] [3 4 5] [6 7 8
(aget matrix 1 1) ; ❷
;; 4

(aset matrix 1 1 99) ; ❸
(mapv vec matrix)
;; [[0 1 2] [3 99 5] [6 7 8]]
```

❶We use to-array-2d to create a 3x3 matrix of array of longs.
❷ `aget` can access the element at index "[1,1]" which is the middle of the matrix.
❸Similarly to `aget`, also `aset` allows for additional indexes to access nested arrays.

In the following example, we are going to see `aget`, `aset` and `alength` in action to produce the transpose of a square matrix. The transpose of a matrix is a common transformation that flips each item over its diagonal [261]. For speed purposes, we decide to implement the matrix as a mutable array of doubles:

```
(defn transpose! [matrix] ; ❶
  (dotimes [i (alength matrix)]
    (doseq [j (range (inc i) (alength matrix))]
      (let [copy (aget matrix i j)] ; ❷
        (aset matrix i j (aget matrix j i))
        (aset matrix j i copy)))))

(def matrix ; ❸
  (into-array
    (map double-array
      [[1.0  2.0  3.0]
       [4.0  5.0  6.0]
       [7.0  8.0  9.0]])))

(transpose! matrix)
(mapv vec matrix) ; ❹

;; [[1.0 4.0 7.0]
;;  [2.0 5.0 8.0]
;;  [3.0 6.0 9.0]]
```

❶ `transpose!` swaps items in place without the need to create a copy of the matrix. This version of the algorithm is particularly effective for large matrices, as they don't need duplication in memory at any given point. As a side effect, the input matrix is permanently changed so the name of the function ends with an exclamation mark to convey this fact.

❷ We need a total of 2 `aget` and 2 `aset` operations. Note that we can use dotimes for the outer index (cross the length of the side of the square matrix) and doseq for the inner loop, which needs to start from the outer index plus one.

❸ We can't use to-array-2d to create a two-dimensional array of type double. We need double-array for the inner array and into-array for the outer array (because `into-array` can infer the type at this point).

❹ After calling `transpose!` we can have a look at the result transforming the array into a vector of vectors.

In the version of `transpose!` above we decided to mutate the array in place. The solution has the advantage of not requiring a copy of the entire matrix at the price of side effects. We could re-work the example to transpose the array into a new copy and avoid any side effect with `aclone`:

```
(defn transpose [matrix]
  (let [size (alength matrix)
```

```
          output (into-array (map aclone matrix))]
      (dotimes [i size] ; ❷
        (dotimes [j size]
          (aset output j i (aget matrix i j))))
    output))

(def matrix
  (into-array
    (map double-array
      [[1.0  2.0  3.0]
       [4.0  5.0  6.0]
       [7.0  8.0  9.0]])))

(def transposed (transpose matrix)) ; ❸
(mapv vec transposed)

;; [[1.0 4.0 7.0]
;;  [2.0 5.0 8.0]
;;  [3.0 6.0 9.0]]
```

❶ `transpose` is a rework of the function `transpose!` seen previously. This version does not require the mutation of the input array. We use `aclone` to clone each of the inner arrays that compose the matrix. All changes can now happen on a brand new array.

❷ The new approach simplifies looping over the coordinates "i" and "j", which is just assigning the value found at "[i,j]" to the inverted coordinated "[j,i]" on the output.

❸ Note that the transposed matrix is now the output of the function. The "matrix" array is left untouched.

`aclone` is not the only approach for the problem of transposing a matrix. We could create an empty multi-dimensional array instead of cloning the input. Note however that `aclone` has the advantage of handling the type of the copied array without the need for `transpose` to mention `double-array` (or other typed function) explicitly.

Calculating the transpose matrix (as the example above) requires to know the length of the array. Clojure provides a specific function `alength` for that purpose. count would also work, but it would generate an expensive reflective call, as `count` is designed to receive a generic object without a compile-time notion of its type. When the type of the array is

known, `alength` can take advantage of that information. Have a look at the following benchmark:

```
(require '[criterium.core :refer [quick-bench]])

(let [a (int-array (range 1000))] (quick-bench (
;; Execution time mean : 109.572276 ns

(let [a (int-array (range 1000))] (quick-bench (
;; Execution time mean : 2.496323 ns
```

❶ `count` works with arrays, but it's not the most effective way to calculate the length, as the benchmark demonstrate.

❷ `alength` knows the type of the array and delegates the call accordingly without the need of reflection.

21.7.5 amap and areduce

`amap` and `areduce` simplify the work of processing Java arrays by removing the need for an explicit loop while enforcing immutability. `amap` (named after "a" for array and "map" for map) executes the given expression on all elements of the array:

```
(def a1 (int-array (range 10)))

(def a2 (amap a1 idx output 1)) ; ❶

(vec a2)
;; [1 1 1 1 1 1 1 1 1 1]

(vec a1) ; ❷
;; [0 1 2 3 4 5 6 7 8 9]
```

❶This simple transformation replaces each element with the number 1. Note that we are not using "idx" or "output". See below for an explanation of the meaning of these two parameters.

❷The source array is unaltered after the transformation.

`amap` takes the array, an index symbol (conventionally "idx") and a result symbol (conventionally "output") and an expression. The expression evaluates once for each element in the input, with the index symbol bound to the current index and the result symbol bound to the output which is currently under construction:

```
(def a1 (int-array (range 4)))

(defn debug [idx output]
```

```
    (println "idx:" idx "output:" output) 9)

  (def a2 (amap a1 idx output (debug idx (vec outp
  ;; idx: 0 output: [0 1 2 3]
  ;; idx: 1 output: [9 1 2 3]
  ;; idx: 2 output: [9 9 2 3]
  ;; idx: 3 output: [9 9 9 3]
```

❶With the help of `debug`, we can see how "idx" and "output" are changing each iteration. The value of "output" displayed is the one just before the new value is assigned to the corresponding index. The number 9 is applied to each mapping operation.

We can use the index symbol to access the input array and perform a transformation of the old item into the new:

```
  (defn ainc [a]
    (amap a idx _ (inc (aget a idx)))) ; ❶

  (vec (ainc (int-array (range 10))))
  ;; [1 2 3 4 5 6 7 8 9 10]
```

❶ `ainc` (or Array Increment) adds plus one to each element in a numeric array. "idx" is bound by `amap` to the current index in the array and we can use it to transform the input item at that index.

The output symbol represents the output array under construction at each stage through the iteration. We could use this information to prevent any further changes when the sum of the updated items goes beyond a certain limit:

```
(defn asum-upto [a i] ; ❶
  (loop [idx 0 sum 0]
    (if (= idx i)
      sum
      (recur (inc idx) (+ sum (aget a idx))))))

(defn amap-upto [a f limit]
  (amap a idx out
    (let [old (aget a idx) ; ❷
          new (f old)
          sum (asum-upto out idx)]
      (if (> (+ new sum) limit) old new))))

(def a (int-array (range 10)))
```

```
  (vec (amap-upto a #(* % %) 60)) ; ❸
  ;; [0 1 4 9 16 25 6 7 8 9]
```

❶ `asum-upto` is a function that sums the numbers in a numeric array starting at index 0 up to index "i".

❷ The `amap` expression needs to calculate the current item at the index, the value of the new item after the transformation and the sum of all the transformed items so far. Only if that sum does not exceed the limit the current item is updated.

❸ We can see that after squaring a few numbers, the process "stops" (it doesn't actually stop, it continues to update items with a copy of themselves).

Array and type hints

One of the main reason to use arrays is speed, especially for primitive types. Primitive type arrays are good for two reasons: primitive types occupy less space in memory and they are in general much faster to process.

Clojure offers array functions that often come with all primitive variants: `int-array`, `aset-int` and so on. But there is still a problem to solve every time an array crosses function boundaries, which involves wrapping the array in a generic objects:

```
  (set! *warn-on-reflection* true)
  (def a (int-array (range 10)))

  (alength a) ; ❶
```

```
;; Reflection warning, call to static method alength on clojure.l
;; 10
```

❶When using `alength` on an array of integers, we get a reflection warning related to the fact that Clojure can't determine the type of "a".

The array "a" is defined as `clojure.lang.Var` in the current namespace. It has to be retrieved using an automatic indirection mechanism: the symbol "a" is found in the current namespace mappings and that is the key to access the related var object. The Clojure runtime proceeds to call deref on the var object automatically. deref is a function returning a `java.lang.Object` because there is no notion of what the var refers to. Hence Clojure does not know which `alength` type to call and resolves the call for a generic object type. This call uses reflection. To avoid the reflection warning we need a type hint:

```
(def a (int-array (range 10)))

(alength ^"[I" a) ; ❶
;; 10

(alength ^ints a) ; ❷
;; 10
```

❶How to type hint an array of primitive integers using Java array class encoding.
❷Clojure supports a shortcut version for all array types, so we don't need to remember the Java encoding for array classes.

Another important factor to consider to speed up performances is avoiding autoboxing. Let's go back at the function `amap-upto` that was used in the examples and have a look at reflection and boxing warnings:

```
(set! *warn-on-reflection* true)
(set! *unchecked-math* :warn-on-boxed) ; ❶

(defn amap-upto [a f limit]
  (amap a idx out
    (let [old (aget a idx)
          new (f old)
          sum (asum-upto out idx)]
      (if (> (+ new sum) limit) old new))))

; ❷
;; Reflection warning, method aclone on RT can't
;; Reflection warning, alength on RT can't be re
;; Boxed math warning, boolean Numbers.lt(long,O
;; Reflection warning, aget on RT can't be resol
;; Boxed math warning, Number Numbers.unchecked_
```

```
  ;; Boxed math warning, boolean Numbers.gt(Object
  ;; Reflection warning, aset on RT can't be resol
```

❶Along with `*warn-on-reflection*`, another useful warning to see is `*unchecked-math*` with `:warn-on-boxed` option. `:warn-on-boxed` shows a message each time Clojure has a primitive type variant of a function that can't be selected.

❷We can see a long list of reflection and boxed math problems to solve.

Reflective problems like those in `amap-upto` requires some practice to solve. First of all, you need to learn about the syntax for type hinting [262] and secondly you need to follow the compiler messages to understand which part of the code requires hinting. Luckily, the compiler is quite precise in indicating where the information is missing and sometimes a single type hint solves many warnings at once. Here's a version of `amap-upto` that fixes all warnings:

```
(defn amap-upto [^ints a f limit] ; ❶
  (amap a idx out
    (let [old (aget a idx)
          ^int new (f old) ; ❷
```

```
            ^int sum (asum-upto out idx)]
        (if (> (+ new sum) ^int limit) old new))))
```

❶The first type hints is about the input array which would otherwise enter the function as a generic object.

❷Another complain from the compiler is about the "+" function. The compiler was unable to understand which "\+" override to use. We need to say explicitly that both "new" and "sum" are primitive integers.

❸Final complain from the compiler is about ">" (called "gt"). We had to type hint "limit" as well.

The lesson to learn from this "warnings elimination" exercise is that, especially when working with native arrays, you often need to add type hints at function boundaries. Additionally, when using items from the array, there is often the risk of unwanted boxing of primitives.

We are going to close this section with a look at `areduce`. As you probably imagine from the name, this is a special reduce version for arrays. It works similarly to `amap` with an additional "init" parameter:

```
  (def a (int-array (range 10)))
```

```
(areduce a idx acc 0 (+ acc (aget a idx))) ; ❶
;; 45
```

❶ `areduce` binds "idx" and "acc" during the internal iteration, so the expression can access them. This is similar to `amap`, but "acc" (the accumulator) contains the sum so far instead of the output array. The new parameter is the initial value for the computation.

The same recommendation about type hinting are valid when using `areduce`.

21.7.6 aset-int and other types setters

Note

This section also mentions other related functions such as: `aset-boolean`, `aset-byte`, `aset-short`, `aset-char`, `aset-long`, `aset-float`, `aset-double`

The group of functions in this section are related to aset and the need to avoid reflective calls. Observe the following example:

```
(set! *warn-on-reflection* true) ; ❶

(def a (int-array [1 2 3])) ; ❷
(type a) ; ❸
;; [I

(aset a 0 9) ; ❹
;; Reflection warning, call to static method aset
;; can't be resolved (argument types: unknown, i
;; 9
```

❶ `*warn-on-reflection*` turns on compiler warnings related to reflective calls. If the compiler is unable to determine the type of one or more operands, it will issue a reflection warning. The warning does not prevent correctness or the program, but simply warn the user that in order to find the right method to dispatch to, the compiler had to use reflection.

❷ We create a simple array or primitive integers.

❸ The type of the array confirms that this is indeed an array of primitives.

❹ When we use aset on the array, the compiler issues a warning that it cannot find the right `clojure.lang.RT/aset` call to dispatch to. The compiler also provides details such as the type of the parameters that was searched for. The "Unknown" is about "a", the primitive "int" is the index and the number "9" was passed from the reader as a primitive "long". Note that the array is modified anyway and the item that was just written is returned.

The reflective problem above is produced by implicit var indirection. Let's rewrite the example so the implicit run time steps become explicit:

```
(def a-lookup (get (ns-map *ns*) 'a)) ; ❶
```

```
(type a-lookup) ; ❷
;; clojure.lang.Var

(type (deref a-lookup))
;; [I

(aset (deref a-lookup) 0 9) ; ❸
;; Reflection warning, - call to static method a
;; can't be resolved (argument types: unknown, i
;; 9
```

❶def produces an additional side effect apart from creating an instance of `java.lang.Var`: a new entry is created in the local namespace mappings (`*ns*` is a shortcut for the current namespace object). The entry is a pair with the symbol "a" as key and the var object as value. We can use ns-map to retrieve the var from the mappings.

❷ `a-lookup` has the expected "Var" type, as well as the content of the var which is the expected array of integers.

❸This is what `aset` actually sees at compile time.

When Clojure compiles the code above, it can only sees that deref returns an object (because a var could really point at anything) and even if specialized `aset` exists for different primitive types, that information is now lost.

There are 2 ways to remove the reflective call: first we could type hint `a-lookup` as array of primitive integers. Second we could use `aset-int` (or any other typed `aset-*` call in case of other primitive types). The case of type hinting the array is so common that Clojure offers `aset-int` to make things easier:

```
(aset ^ints a 0 9) ; ❶
;; 9

(aset-int a 0 9) ; ❷
;; 9
```

❶In the first case, we type hint "a" so it carries the information regarding its type at compile time and Clojure can emit a call to the right `aset` specialization (in `clojure.lang.RT`).
❷Alternatively, we can use `aset-int` and achieve the same result without the need to see the type hint.

`aset-int` and the other typed `aset-*` grouped in this chapter have the same semantic of `aset`. In particular, they support multiple indexes for multi-dimensional arrays:

```
(def matrix ; ❶
  (into-array
    (map int-array [[1 2 3] [4 5 6]])))
```

```
(aset-int matrix 0 2 99) ; ❷

(mapv vec matrix)
;; [[1 2 99] [4 5 6]]
```

❶ `matrix` is a multi-dimensional array of primitive integers.
❷ `aset-int` also accepts additional indexes before the last argument (which is always considered the value to set).

As expected, the type of the array should be consistent with the specific `aset-*` version we are trying to use. `aset-*` functions follow Java conventions for allowed type of casts. Forcing a type requiring less precision into one requiring more precision is always possible, but the opposite is not true:

```
(def int-a (int-array 5))
(def double-a (double-array 5))

(aset-int double-a 0 99) ; ❶
;; 99

(aset-double int-a 0 99.0) ; ❷
;; IllegalArgumentException argument type mismat
```

❶ We can `aset-int` into an array of `double`, because the implicit `int` to `double` conversion fits nicely into the number of bytes allocated for the items in the array.

❷ The opposite however is not true. The type conversion requires a loss of precision of the `double` number "99.0" to fit into the bytes allocated for a primitive `int` type.

21.7.7 ints and other typed array casting

> Note
>
> This section also mentions other related functions such as: `booleans`, `bytes`, `shorts`, `chars`, `longs`, `floats` and `doubles`.

Similarly to int-array (and related `*-array` functions) and aset-int (with related `aset-*` functions), Clojure also offers a group of specialized array cast functions.

Clojure provides such group of specialized primitive type functions to help working with primitive arrays. By using the specialized version of a function we can provide the necessary type information without type hinting, a nice plus in terms of general readability. Here's for example a function `asum` to sum numeric arrays:

```
(defn asum [a1 a2]
  (let [a (aclone (if (> (alength a1) (alength a
    (amap a idx ret
      (aset a idx
        (+ (aget a1 idx) (aget a2 idx))))))

(vec (asum (int-array [1 2 3]) (int-array [4 5 6
;; [5 7 9]
```

The `asum` function works generically on any array type, but performs poorly because of many reflective calls and boxing of primitive types:

```
(set! *warn-on-reflection* true) ; ❶
(set! *unchecked-math* :warn-on-boxed)

(defn asum [a1 a2]
  (let [a (aclone (if (> (alength a1) (alength a
    (amap a idx ret
      (aset a idx
        (+ (aget a1 idx) (aget a2 idx))))))

Reflection warning, call to static method alengt
Reflection warning, call to static method alengt
Boxed math warning, call public static boolean Nu
Reflection warning, call to static method aclone
```

```
Reflection warning, call to static method aclone
Reflection warning, call to static method alengt
Boxed math warning, call public static boolean Nu
Reflection warning, call to static method aget o
Reflection warning, call to static method aget o
Boxed math warning, call public static Number Nu
Reflection warning, call to static method aset o
Reflection warning, call to static method aset o
```

❶If after turning on reflective warnings and boxing warnings we re-evaluate the same `asum` function we get an impressive list of warnings.

If we can make the assumption that `asum` is always going to sum integer arrays, we can use the relative specialized casts:

```
(defn asum-int [a1 a2]
  (let [a1 (ints a1) a2 (ints a2) ; ❶
        a (aclone (if (> (alength a1) (alength a
    (amap a idx ret
      (aset a idx
        (+ (aget a1 idx) (aget a2 idx))))))

;; #'user/asum-int ; ❷
```

❶let bindings preserve and propagate type information to downstream forms. By casting "a1" and "a2" with `ints` we give the compiler all the necessary information for the input arrays.
❷After using `ints` on the inputs, all warnings disappear.

The other type array cast functions, work with the same principle as `ints` that was demonstrated in the example above. Primitive array casting is useful when the relative type hint is not available or when it's difficult to position it.

[261] (see en.wikipedia.org/wiki/Transpose).
[262] one good place to start is clojure.org/reference/java_interop#typehints

22 THE TOOLBOX

The last chapter of the book collects a variety of functions in the standard library dedicated to solve specific problems. Some of the functions or macros in this chapter have been used throughout the book and here we are going into greater details. Functions in this chapter are grouped by their originating namespace. Here's an overview:

- clojure.xml, as the name implies, contains functions to read or write XML.
- clojure.inspector contains tools to visualize data structures.
- clojure.repl contains functions dedicated to improve the REPL experience.
- clojure.main is the entry point to bootstrap and execute Clojure applications. It also contains the actual REPL implementation and a set of functions to customize the REPL experience.
- clojure.java.browse contains a single public function browse-url to open a native browser given a specific URL.

- clojure.java.shell: the most important function in this namespace is sh, a function to "shell-out" commands to the native operative system.
- clojure.core.server contains the implementation of a socket server, a service that offers Clojure evaluation (similarly to the REPL) to remote clients.
- clojure.java.io contains Clojure wrappers to manage the Java IO (Input/Output) system including files, streams, the classpath and more.
- clojure.test is the testing framework that ships with Clojure. `clojure.test` is configurable and extensible.
- clojure.java.javadoc contains facilities to access Java documentation using the default system browser.

22.1 CLOJURE.XML

Note

> `clojure.xml` has been recently proposed for deprecation (see the related Jira issue) and it is documented here as part of the public Clojure API. It was never refactored or re-designed beyond the first implementation and despite being fully functional (albeit with some limitation) the reader should use Clojure Data XML whenever possible.

The main goal of `clojure.xml` is to produce an in-memory data representation of an XML input source. In doing so, the XML content is loaded into memory for further processing. `clojure.xml/parse` is the main entry point that produces a Clojure data structure from an XML document:

```
(require '[clojure.xml :as xml]) ; ❶

(def document
  (xml/parse
    "http://feeds.bbci.co.uk/news/rss.xml")) ; ❷

(keys document) ; ❸
;; (:tag :attrs :content)
```

❶We need to require `clojure.xml` before using it.
❷RSS feeds are normally used by web sites to produce a machine readable version of their content. RSS is built on top of XML, so we can read it with `xml/parse`.
❸ `xml/parse` returns a struct-map of 3 keys.

`xml/parse` builds a nested structure of struct-map and vectors with the following recursive shape:

```
{:tag ...      ; ❶
 :attrs ...
 :content
 [{:tag ... :attrs ... :content [...]}
  ...
  {:tag ...
   :attrs ...
   :content
   [{:tag ... :attrs ... :content [...]}
    ...
    {:tag ... :attrs ... :content [...]}]}]}
```

❶A partial representations of the `struct-map` produced by `xml/parse` that only shows the skeleton data structure.

Each struct represents an XML node. The `:tag` key is the name of the node, the `:attrs` key contains a map of attributes of the

node and finally `:content` a vector of children nodes.

Note

The use of structs in `clojure.xml` is one of the few in the standard library (other notable examples are resultset-seq or cl-format). defrecords effectively replace the need for structs in other parts of the library and for everyday Clojure programming.

`xml/parse` accepts different types as input: a `java.io.File`, a string representing an URL or a generic `java.io.InputStream`. It also accepts an optional parameter "startparse" that can be used to configure special features of the parser. We could for example use this parameter to disable schema validation or prevent schema loading [263]:

```
(require '[clojure.xml :as xml]
         '[clojure.java.io :as io])

(def conforming ; ❶
  "<?xml version='1.0' encoding='utf-8'?>
     <!DOCTYPE html SYSTEM 'http://www.w3.org/TR
     <html xmlns='http://www.w3.org/1999/xhtml'>
       <article>Hello</article>
```

```
      </html>")

(-> conforming .getBytes io/input-stream xml/par

;; hangs 60 seconds if missing or misconfigured

;; {:tag :html,
;;  :attrs {:xmlns "http://www.w3.org/1999/xhtml
;;  :content [{:tag :article, :attrs nil, :conte
```

❶This conforming snippet of XHTML (an XML-compliant dialect of HTML) requires an external DTD for valiation. The DTD file contains the specification for the expected syntax in the XHTML.

❷For this experiment, we load XHTML directly as a string. `xml/parse` interprets strings as URL, so we need a transformation of the string into an input-stream, a function available from clojure.java.io.

❸If we have networking problems or the JDK is not configured properly to allow outgoing connections, we soon discover that parsing could take up to 60 seconds or more (depending on the JDK timeout settings).

Parsing XML with validation is usually wise feature to have, especially in production environment. However, for testing or

development, we might want to switch it off. We can disable XML validation passing a `non-validating` parser function to `xml/parse`:

```
(import '[javax.xml.parsers SAXParserFactory])

(defn non-validating [s ch] ; ❶
  (..
    (doto
      (SAXParserFactory/newInstance) ; ❷
      (.setFeature "http://apache.org/xml/featur
    (newSAXParser)
    (parse s ch)))

(-> conforming .getBytes io/input-stream (xml/pa
;; {:tag :html,
;;  :attrs {:xmlns "http://www.w3.org/1999/xhtml
;;  :content [{:tag :article, :attrs nil, :conte
```

❶ `non-validating` is a function of the input source and a content handler "ch". The content handler is the default used by `clojure.xml` and we can reuse it if we don't need to alter the way XML is transformed.

❷ The `SAXParserFactory` object accepts configuration using the `setFeature` method.

❸ Parsing is now independent from network calls and but also non-validating.

Exactly like you can parse xml with `xml/parse` you can reverse the process with `xml/emit`. `xml/emit` takes the output of `xml/parse` and converts it back to a string sending it to standard output:

```
(def xml ; ❶
  {:tag :html,
    :attrs {:xmlns "http://www.w3.org/1999/xhtml
    :content [{:tag :article, :attrs nil, :conte

(xml/emit xml) ; ❷

;; <?xml version='1.0' encoding='UTF-8'?>
;; <html xmlns='http://www.w3.org/1999/xhtml'>
;; <article>
;; Hello
;; </article>
```

```
  ;; </html>

  nil
```

❶ `xml` is the output of the previous parsing example.

❷ `xml/emit` converts the data structure back to XML (except for the "DOCTYPE" declaration).

If you want to capture `xml/emit` output into a string or file, please have a look at with-out-str or binding to redirect the output to a different stream.

[263] XML documents can specify a schema they should be forced to. XML parsers usually follow the directive and validate the content against the schema. In some cases of corrupted documents or lack of network connectivity, we might need to switch off validation to be able to load the document. The list of all parser features is available here: xerces.apache.org/xerces2-j/features.html

22.2 CLOJURE.INSPECTOR

The `clojure.inspector` namespace contains a small visualization utility for structured data. The inspector wraps the data in a basic (Swing [264]) user interface that supports a few visualization models: tabular, sequential and tree-like. For

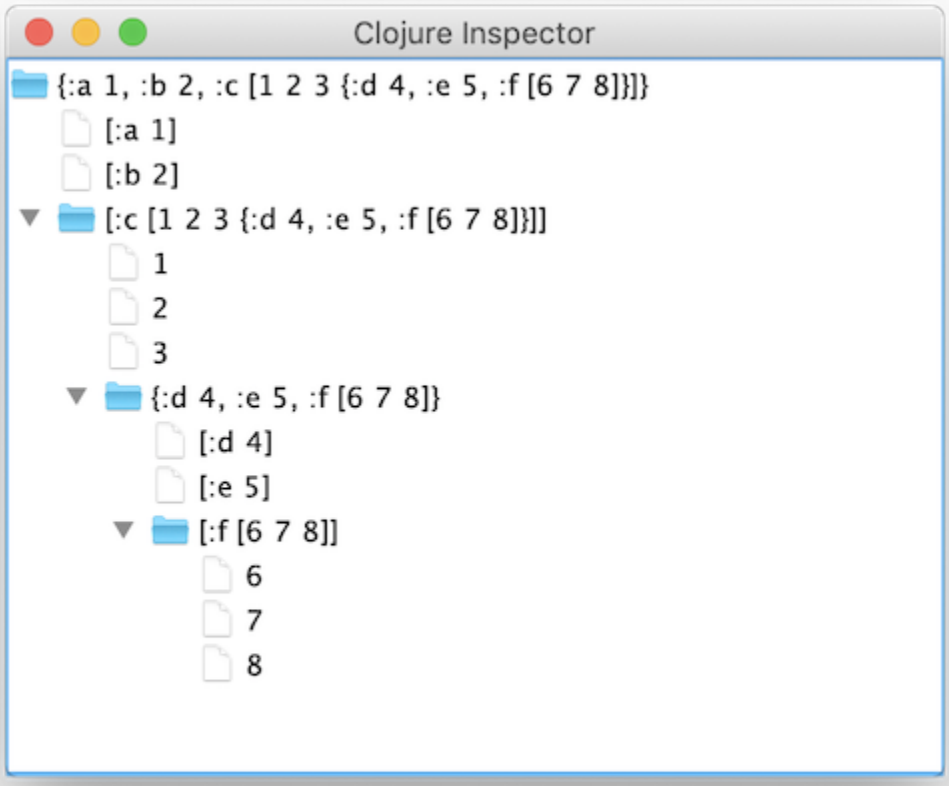
example, the following code produces the window displayed below:

```
(require '[clojure.inspector :refer [inspect-tre
(inspect-tree {:a 1 :b 2 :c [1 2 3 {:d 4 :e 5 :f
```

❶ `inspect-tree` is part of the `clojure.inspector` namespace.

❷ `inspect-tree` takes an arbitrarily nested data structure that we wish to visualize.

Figure 22.1. The inspector visualization tree for a nested data structure.



Alternatively, we can inspect a group of collections of the same type, such as a list of maps, using the tabular model shown below. The inspector uses the first record (or list of items) to establish the size of the table. That's why the data type should be the same (same keys or same size) to display correctly:

```
(require '[clojure.inspector :refer [inspect-tab

(def events [{:time "2017-05-04T13:08:57Z" :msg
```

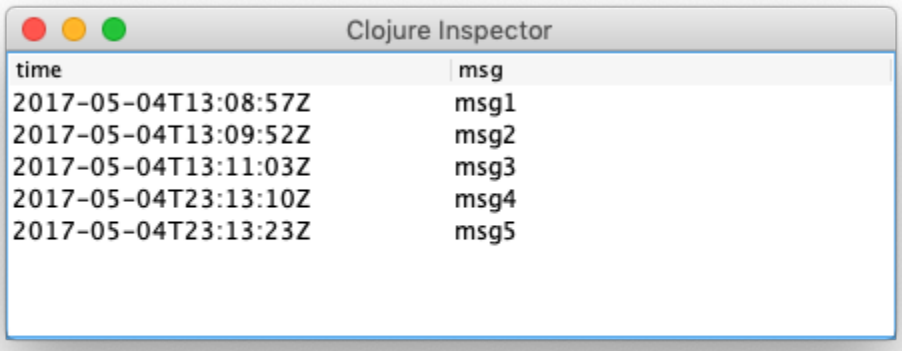
```
                {:time "2017-05-04T13:09:52Z" :msg
                {:time "2017-05-04T13:11:03Z" :msg
                {:time "2017-05-04T23:13:10Z" :msg
                {:time "2017-05-04T23:13:23Z" :msg

  (inspect-table events)
  ;; See image below
```

❶ `inspect-table` is part of the `clojure.inspector` namespace.

❷ This vector of events is the kind of data structure which renders well with a table model. The keys of the first map becomes the headers in the table.

Figure 22.2. The inspector visualization table for a group of uniform data.



Overall, `clojure.inspector` comes handy to quickly visualize a table or a tree. However, it doesn't offer any specific configuration or extensibility.

[264] The Java Swing framework is a windowing toolkit part of the standard library since Java 1.2. For more information please see the introductory Wikipedia page: en.wikipedia.org/wiki/Swing_(Java).

22.3 CLOJURE.REPL

`clojure.repl` contains functions and macros to help working within the REPL (the Read Eval Print Loop console). It provides two broad categories of utilities:

- Documentation functions such as doc and with-out-str.
- Source and environment related functions such as source, apropos and dir.
- Exception handling functions like: root-cause or pst.

Note

Some functions in `clojure.repl` are directly available at the REPL without requiring them: source, apropos, dir, pst, doc and with-out-str.

---

doc

`doc` is possibly the most used documentation macro. It takes a symbol and returns the value of the `:doc` key from the metadata of a var or namespace object. In the following example, we show `doc` before and after adding the documentation to a `var`:

```
(def life 42)

(doc life) ; ❶
;; -------------------------
;; user/life
;;   nil
;; nil

(alter-meta! #'life assoc :doc ; ❷
  "Answer to the Ultimate Question of Life the U

(doc life) ; ❸
;; -------------------------
;; user/life
;;   Answer to the Ultimate Question of Life the
;; nil
```

❶The definition of "life" has no documentation string.

❷alter-meta! permanently alters the metadata of a var, in this case by adding a new key to any of the already existing metadata map.

❸After calling `doc` one more time, we can see that it is correctly reporting the documentation string.

`doc` works similarly for functions and namespaces:

```
(doc doc) ; ❶

;; clojure.repl/doc
;; ([name])
;; Macro
;;   Prints documentation for a var or special f

(ns my-namespace
  "This namespace contains useful functions."
  (:require [clojure.repl :refer [doc]]))

(doc my-namespace) ; ❷

;; -------------------------
;; my-namespace
;;   This namespace contains useful functions.
;; nil
```

❶ `doc` can document itself or all other functions in the standard library.
❷ `doc` similarly extracts documentation strings from namespaces. Note that the new namespace `my-namespace` does not have the function `clojure.repl/doc` available by default, so require-refer of `doc` was added to the namespace declaration.

find-doc

`doc` works when you know exactly what you are searching for. If you just happen to know part of a name, or a specific use case you're interested in, you can invoke `find-doc` with a regular expression (as a string) to search for matches in all available documentation strings. The following query asks if there are functions or macros starting with "to-":

```
(in-ns 'user)
(find-doc "^to-") ; ❶

;; -------------------------- ; ❷
;; clojure.core/to-array
;; ([coll])
;;   Returns an array of Objects containing the
;;   can be any Collection.  Maps to java.util.C
;; -------------------------
```

```
  ;; clojure.core/to-array-2d
  ;; ([coll])
  ;;   Returns a (potentially-ragged) 2-dimensiona
  ;;   containing the contents of coll, which can
  ;;   Collection.
```

❶ `find-doc` takes a string that is transformed into a regular expression pattern. It searches for matches (including partial matches) in all available documentation strings, var or namespaces.

❷ The output of `find-doc` (as well as `doc`) is designed to print directly to standard output. We can see here all the functions starting with "to-".

apropos

`find-doc` can be very verbose when using patterns that are too generic. For example, if you try to search for "to-" instead of "^to-" as we did in the query above, the output is going to be several pages long. The reason is that "to-" can appear either in the function name or documentation string. If you want to see what functions contain "to-" in their name only, you can use `apropos`:

```
(apropos "to-") ; ❶

;; (clojure.core/into-array
;;  clojure.core/to-array
;;  clojure.core/to-array-2d)
```

❶Compared to `find-doc`, `apropos` search for matches in the name of definitions only, no namespaces or descriptions. Also note that the result is a list which is not printed to standard output.

dir

Another useful way to search what is available is by namespace. The `dir` macro prints an ordered list of all the public definitions declared within a namespace:

```
(require 'clojure.reflect)
(dir clojure.reflect) ; ❶

;; ->AsmReflector
;; ->Constructor
;; ->Field
;; ->JavaReflector
;; ->Method
;; ClassResolver
```

```
  ;; Reflector
  ;; TypeReference
  ;; do-reflect
  ;; flag-descriptors
  ;; map->Constructor
  ;; map->Field
  ;; map->Method
  ;; reflect
  ;; resolve-class
  ;; type-reflect
  ;; typename
```

❶ `dir` shows the content of the `clojure.reflect` namespace. The namespace has been required as usual to make it available for inspection.

dir-fn

`dir` prints results directly to standard output. This is the most useful way to access that information at the REPL, but if you need to manipulate the same results as a sequence, you can use `dir-fn`. `dir-fn` is a function, so the compared to `dir` you need to quote the namespace:

```
  (require 'clojure.java.browse)
  (require '[clojure.repl :refer [dir-fn]]) ; ❶
```

```
  (apply str (interpose "," (dir-fn 'clojure.java.
  ;; "*open-url-script*,browse-url"
```

❶ Differently from previous examples, `dir-fn` needs explicit refer.

❷ `dir-fn` returns a sequence. Here we separate each definition with a comma to produce a single string.

source

Clojure is quite readable language. Reading sources (especially from the standard library) is useful exercise and sometimes required to understand how to use a function. However, it might not be immediately clear where to look for sources. The `source` macro solves the problem by taking the name of a public definition and printing the sources to standard output:

```
  (source unchecked-inc-int) ; ❶

  ;; (defn unchecked-inc-int ; ❷
  ;;   "Returns a number one greater than x, an in
  ;;   Note - uses a primitive operator subject to
  ;;   {:inline (fn [x] `(. clojure.lang.Numbers (
  ;;    :added "1.0"}
  ;;   [x] (. clojure.lang.Numbers (unchecked_int_
```

❶ `source` is a macro and there is no need to quote the symbol passed as argument.
❷ The source definition of `unchecked-int-inc` prints on the screen with the same formatting (lines and indentation) used in the original definition.

Sources are available via `source` with a few restrictions:

- Sources become available only after loading the corresponding namespace (a require might be necessary to load the namespace).
- The namespace needs to be visible from the current Java classpath. This is the case for the standard library and normal application dependencies, but it rules out dynamically generate functions.
- Special form definitions are not available as they are implemented in Java. Similarly, we cannot get sources for Java methods or classes.
- The `*read-eval*` dynamic binding should not be set to `:unknown`.

The last point about `*read-eval*` requires an explanation. `source` uses the Clojure reader internally to load the sources from their location. The Clojure reader supports the `*read-eval*` macro to enable immediate evaluation of forms prefixed

with `#=` (please refers to read for more information on this aspect). The `:unknown` value is designed for testing compliance with the read-eval macro: any code using the reader to load sources should throw an exception if the value of the `*read-eval*` variable is `:unknown`.

source-fn

The `source` macro also exists in function version: `source-fn` takes a symbol and returns the original string containing the sources without printing it to standard output:

```
(require '[clojure.repl :refer [source-fn]])
(source-fn 'not-empty) ; ❶
;; "(defn not-empty\n  \"If coll is empty, retur
;; {:added \"1.0\"\n   :static true}\n  [coll] (

(println *1) ; ❸
;; (defn not-empty
;;   "If coll is empty, returns nil, else coll"
;;   {:added "1.0"
;;    :static true}
;;   [coll] (when (seq coll) coll))
```

❶ `source-fn` now requires quoting to pass the argument as a symbol.

❷ The source of the function not-empty appears with the original formatting (newlines and spaces).

❸ We can see the function with full pretty printing by calling `println` on it. Note: `*1` is a dynamic var which is automatically associated to the last evaluated expression at the REPL.

pst

`pst` stands for for "Print Stack Trace". `pst` is an useful function to retrieve just the right amount of error information. Java can produce huge stack traces that, in some extreme cases, require scrolling multiple pages to see the root cause which appears at the top. To avoid cluttering of the screen, the REPL by default only shows the most important information. For example, the REPL only prints the main message of the following division by zero error:

```
(/ 1 0) ; ❶

;; ArithmeticException Divide by zero  clojure.l
```

❶ The REPL does a good job showing just the essential information by default. In this case we can understand the problem quickly with just a short description.

For other types of error, we might need to have a look at the full stack trace. The stack trace shows how the exception propagated from the site where the problem happened up to the point the request was made. The REPL stores a copy of the full stack trace in the `*e` dynamic variable:

```
(/ 1 0)
;; ArithmeticException Divide by zero  clojure.la

*e ; ❶

;;#error {
;; :cause "Divide by zero"
;; :via
;; [{:type java.lang.ArithmeticException
;;   :message "Divide by zero"
;;   :at [clojure.lang.Numbers divide "Numbers.ja
;; :trace
;; [[clojure.lang.Numbers divide "Numbers.java"
;;  [clojure.lang.Numbers divide "Numbers.java" 1
;;  ....
```

❶ The full extent of the error message is stored in the `*e` dynamic variable. The error is truncated here to display properly in the book, but it's many lines longer.

❷ Starting from this item in the `:trace` we can see new information about how the exception propagated up the request site.

By default, `pst` takes the content of `*e` and presents the first 12 items from the stack trace:

```
(/ 1 0)
;; ArithmeticException Divide by zero  clojure.l

(pst) ; ❶

;; ArithmeticException Divide by zero
;;   clojure.lang.Numbers.divide (Numbers.java:1
;;   clojure.lang.Numbers.divide (Numbers.java:3
;;   user/eval2147 (form-init4179141376169992155
;;   user/eval2147 (form-init4179141376169992155
;;   clojure.lang.Compiler.eval (Compiler.java:6
;;   clojure.lang.Compiler.eval (Compiler.java:6
;;   clojure.core/eval (core.clj:3105)
;;   clojure.core/eval (core.clj:3101)
;;   clojure.main/repl/read-eval-print--7408/fn-
;;   clojure.main/repl/read-eval-print--7408 (ma
```

```
  ;;   clojure.main/repl/fn--7417 (main.clj:258)
  ;;   clojure.main/repl (main.clj:258)
```

❶ `pst` looks into the content of `*e` by default. You need to generate an exception before invoking `pst` to actually see a stack trace (otherwise it would print nothing and return `nil`).

`pst` optionally takes an exception argument if we are not interested in the one available in `*e`. Additionally, `pst` accepts how many items to retrieve from the top of the stack trace (12 by default):

```
(def ex (ex-info "Problem." {:status :surprise})

(pst ex) ; ❶

;; ExceptionInfo Problem. {:status :surprise}
;;   clojure.core/ex-info (core.clj:4617)
;;   clojure.core/ex-info (core.clj:4617)
;;   clojure.lang.Compiler$InvokeExpr.eval (Comp
;;   clojure.lang.Compiler$DefExpr.eval (Compile
;;   clojure.lang.Compiler.eval (Compiler.java:6
;;   clojure.lang.Compiler.eval (Compiler.java:6
;;   clojure.core/eval (core.clj:3105)
;;   clojure.core/eval (core.clj:3101)
```

```
  ;;   clojure.main/repl/read-eval-print--7408/fn-
  ;;   clojure.main/repl/read-eval-print--7408 (ma
  ;;   clojure.main/repl/fn--7417 (main.clj:258)
  ;;   clojure.main/repl (main.clj:258)

  (pst ex 4) ; ❷

  ;; ExceptionInfo Problem. {:status :surprise}
  ;;         clojure.core/ex-info (core.clj:4617)
  ;;         clojure.core/ex-info (core.clj:4617)
  ;;         clojure.lang.Compiler$InvokeExpr.eval
  ;;         clojure.lang.Compiler$DefExpr.eval (C
```

❶ `pst` accepts a `java.lang.Throwable` object as argument instead of looking into `*e` (the default).

❷ We can also decide how much items from the stack trace to see, for example 4.

Exception objects can be arbitrarily nested, so inner logic can catch the exception, do something about it and throw a new exception that wraps the original. The chain of exceptions formed this way gradually decorates the original with more information, but it also becomes harder to read. `pst` also truncates any nested exception to the required number of items:

```
(def ex ; ❶
  (ex-info "Problem." {:status :surprise}
    (try (/ 1 0)
    (catch Exception e
      (ex-info "What happened?" {:status :unkown

(pst ex 3) ; ❷

;; ExceptionInfo Problem. {:status :surprise}
;;         clojure.core/ex-info (core.clj:4617)
;;         clojure.core/ex-info (core.clj:4617)
;;         clojure.lang.Compiler$InvokeExpr.eval
;; Caused by:
;; ExceptionInfo What happened? {:status :unkown
;;         clojure.core/ex-info (core.clj:4617)
;;         clojure.core/ex-info (core.clj:4617)
;;         user/fn--2169 (form-init41791413761695
;; Caused by:
;; ArithmeticException Divide by zero
;;         clojure.lang.Numbers.divide (Numbers.
;;         clojure.lang.Numbers.divide (Numbers.
```

❶This snippet generates multiple nested exceptions.

❷ `pst` applies the same rule to all nested exceptions, creating a readable stack trace.

root-cause

When working with chained exception objects, it can be useful to access the root cause directly. `root-cause` takes the initial reference to a potentially long exception chain and retrieve just the root cause:

```
(require '[clojure.repl :refer [root-cause]]) ; ❶
(pst (root-cause ex) 3) ; ❷

;; ArithmeticException Divide by zero
;;   clojure.lang.Numbers.divide (Numbers.java:1
;;   clojure.lang.Numbers.divide (Numbers.java:3
;;   user/fn--2169 (form-init4179141376169992155
```

❶ `root-cause` is not available by default and we need to require it from the `clojure.repl` namespace.

❷ `ex` is the chained exception object generated previously. It contains a chain of 3 exceptions. `root-cause` just retains the inner-most.

munge, demunge and stack-element-str

Something that might not be immediately clear from looking at the stack traces so far, is that `pst` not only controls the overall length of the trace, but also operates a clean-up of Clojure "names" using `demunge`. We can use `demunge` directly or

through `stack-element-str`, which is specifically dedicated to improve the appearance of stack trace elements.

Note

Although `munge` and `demunge` are related, `munge` lives in the `clojure.core` namespace while `demunge` lives in `clojure.repl`. This is because `demunge` is mainly used to beautify trace items at the REPL.

`munge` / `demunge` reason to exist is related to the translation necessary between Clojure and Java naming conventions. Java packages map to file system folders, generating restrictions on what can be used in package names. Clojure is even more liberal in terms of what can be used in function names (for example allowing question marks and other symbols), so an implicit translation layers exist to transform between the two conventions. One visible consequence is that while dashes are allowed in namespace names, the relative file needs dashes replaced with underscores. To help with name transformations, `munge` translates valid Clojure function/namespace names into valid Java class/package names, while `demunge` operates the other way around:

```
(ns my-namespace) ; ❶

(require '[clojure.repl :refer [demunge stack-el

(defn my-funct! [] (throw (ex-info "error" {})))

(str my-funct!) ; ❸
;; "my_namespace$my_funct_BANG_@621ada4f"

(demunge (str my-funct!)) ; ❹
;; "my-namespace/my-funct!@4e212104"

(def stack-trace (try (my-funct!) (catch Excepti

(nth stack-trace 2) ; ❻
;; [my_namespace$my_funct_BANG_ invokeStatic "fo

(stack-element-str (nth stack-trace 2)) ; ❼
;; "my-namespace/my-funct! (form-init41791413761
```

❶ Note that `my-namespace` contains a dash "-" sign. Class names in Java does not allow dashes, so Clojure needs to do some transformation on namespaces before they can be part of the class name and package combination.

❷ Both `demunge` and `stack-element-str` need explicit `require` to be used.

❸ When a function is defined, its class is generated. Before generation, Clojure names needs translation to Java conventions. Even if a call to `munge` is not visible here, the name translation happens behind the scenes. The package and class name already follows Java name conventions.

❹ `demunge` accepts a string representing a Java name and translates into an idiomatic Clojure name. For example we can see words such as "BANG" replaced back with the original "!".

❺ `my-funct!` generates an exception when called. We can retrieve an array of stack trace elements calling `.getStackTrace` method on the exception instance.

❻ If we access the element at index 2 in the array, we can see it prints classes using the Java convention, even if they are generated from a Clojure function.

❼ `stack-element-str` prints the stack trace element using the better looking Clojure conventions.

The REPL offers a few additional configurations to produce even a better development experience. We are going to see some of them talking about the `clojure.main` namespace.

22.4 CLOJURE.MAIN



Warning

Most of the functions in `clojure.main` are public, but some of them should be considered implementation details of the REPL internals. This section touches briefly on some of them, focusing mainly on what can be repurposed outside the REPL.

`clojure.main` is the entry point for all Clojure applications. The namespace contains a function `main` responsible for parsing command line parameters and starting a new Clojure program. One of such programs is the REPL, which is the default when no other program is specified. The namespace contains other functions but just a few of them are general enough to be used for other purposes. Therefore we are just concentrating on the following:

- load-script compiles and evaluate the content of a Clojure file.
- repl starts the main REPL loop, but it can also be used to create a customized version of the REPL.

load-script

`load-script` takes a string argument that represents the path to a Clojure file. The file can be part of the Java classpath or the file system. `load-script` distinguishes between the two cases using a naming convention: if the file starts with "@" or "@/", then `load-script` loads and compiles the file from the classpath:

```
(require '[clojure.main :as main]) ; ❶

clojure.core.reducers/fold ; ❷
;; CompilerException java.lang.ClassNotFoundExce

(main/load-script "@clojure/core/reducers.clj")

clojure.core.reducers/fold ; ❹
#object["clojure.core.reducers$fold@41414539"]
```

❶Functions from the `clojure.main` namespace need explicit require.

❷If, after starting a new REPL session, we type the fully-qualified name of a var object that was never loaded before, we get an error.

❸We use `main/load-script` to load and compile the content of the `clojure.core.reducers` namespace. This produces the side effect of creating a new namespace object and load any definitions from the file.

❹After calling, `main/load-script` for the reducers namespace, we can see that the function fold now resolves correctly.

When we use the "@" sign to load a Clojure file, the effects are very similar to use the load function with just a different encoding of the path. If we remove the "@" sign we achieve the same effect of load-file, another function in the core namespace:

```
(require '[clojure.main :as main])

(spit "hello.exe" ; ❶
  "(ns hello)
   (println \"Hello World!\")")
```

```
(main/load-script "hello.exe") ; ❷
;; "Hello World!"
;; nil
```

❶We create a file on the file system called "hello.exe" which contains a namespace declaration and a line that prints "Hello World!". The file doesn't necessarily need to have the "clj" extension.

❷If the "@" sign is not present, then `load-script` tries to load "hello.exe" starting from the folder from where the REPL process was started. The file "hello.exe" is found and evaluated.

repl

The `clojure.main/repl` function starts a new Read Eval Print Loop (possibly on top of another, nesting the two). It takes a few configuration options which are helpful to customize the REPL experience, possibly creating a completely different REPL. In the following example, we start a new REPL assuming we are already inside the default one:

```
(require '[clojure.main :as main]) ; ❶

(main/repl :init #(println "Welcome to a new REP
```

```
;; Welcome to a new REPL! Press ctrl+D to exit.
;; my-namespace=>
```

❶The `repl` function is part of the `clojure.main` namespace.

❷The `:init` option takes a function of no arguments that executes once at the beginning of the session. In this example it prints a welcome message.

❸Note that you can use "ctrl+d" (pressing the "ctrl" key and the "d" key simultaneously) to exit the nested REPL. Careful: doing that a second time will also exit the default REPL!

The `:init` option can be useful to initialize the new REPL so it recognizes a few commands immediately. For example, let's assume we designed a small calculator that implements the 4 fundamental operations: "plus", "minus", "times" and "divide". When starting the new calculator REPL we want those functions to be readily available:

```
(ns calculator) ; ❶

(defn plus [x y] (+ x y))
(defn minus [x y] (- x y))
(defn times [x y] (* x y))
(defn divide [x y] (/ x y))
```

```
(require '[clojure.main :as main])
(main/repl :init #(require '[calculator :refer :

(plus 1 2) ; ❸
;; 3
```

❶The calculator functions are defined in the `calculator` namespace.

❷The functions of the `calculator` namespace are required immediately after initializing the new REPL.

❸We can now type any of the 4 arithmetic operation and use them accordingly.

There are other configuration options that can help improve the user experience designing a custom REPL environment. We could use `:prompt` to clearly state the purpose of the new REPL:

```
(require '[clojure.main :as main])

(def repl-options
  [:init   #(require '[calculator :refer :all])
   :prompt #(printf "enter expression :> ")])

(apply main/repl repl-options)
```

```
;; enter expression :> (+ 1 1) ; ❶
;; 2
```

❶By using the `:repl` option we can change the appearance of the prompt.

We can go further and completely depart from the way the Clojure REPL normally evaluates expressions. In the following example, a custom REPL calculates small infix mathematical expressions. To do this, we need to override both the `:read` function as well as the `:eval` function using the corresponding options keys:

```
(require '[clojure.main :as main])

(def repl-options
  [:prompt #(printf "enter expression :> ")
   :read   (fn [request-prompt request-exit] ; ❶
             (or ({:line-start request-prompt :s
                   (main/skip-whitespace *in*))
                   (re-find #"^(\d+)([\+\-\*\/])("
   :eval   (fn [[_ x op y]] ; ❹
             (({"+" + "-" - "*" * "/" /} op)
              (Integer. x) (Integer. y)))])

(apply main/repl repl-options)
```

```
;; enter expression :> 2*3 ; ❺
;; 6
;; click ctrl+d to exit calculator
```

❶ The `:read` option accepts a function of 2 arguments. We use them to indicate the REPL when a new prompt should be requested and a custom way to handle "ctrl+D" (which generates an end of stream signal).

❷ `main/skip-whitespace` walks the standard input skipping any white space characters (if any) and positioning the standard input (a stateful object) in one of the possible 3 positions: `:body`, `:line-start` or `:stream-end`. `:body` is the next readable token (the mathematical expression in our case) so the `or` expression jumps to the next that contains the read-line calls.

❸ `read-line` reads an entire line from standard input. In our case, it waits for the user to type an expression and hit enter. At that point, the line is read as a string and sent to a regular expression that splits it into the relative matching groups.

❹ The line returns from `:read` as a list of 4 items. The first is the entire expression itself (that we can ignore). The next 3 arguments are the first operand "x", the operator "op" and the second operand "y". `eval` proceeds to convert the operands into number and the operator into a function which is finally invoked.

❺ The different prompt warns the user of the different REPL semantic. In the new expression REPL, there is no need of

parenthesis and operators appears in infix position. Hit ctrl+D to exit the inner loop and go back to normal REPL.

The previous example shows that the `clojure.main/repl` was designed with extensibility in mind. The level of customization can get the REPL experience quite far, to the point of allowing interaction with small languages built on top of Clojure.

22.5 CLOJURE.JAVA.BROWSE

`clojure.java.browse` contains functions to visualize HTML content through the system browser. The main and only entry point is `browse-url` a function that takes an URL as a string and interact with the operative system to open the URL using one of the available methods:

- HTML Browser: this is the default method. The operative system where Clojure is running is assumed to have a default system browser to call to open the given URL.
- Swing browser: if a default HTML browser is not available, `browse-url` tries to use the Swing HTML rendering facilities. Swing is a graphical window toolkit available in the Java runtime.

- Custom script: the user can also customize which command line executable to use by setting the `clojure.java.broswe/*open-url-script*` dynamic variable.

Using `browse-url` is quite simple. For example, the following opens a browser showing the home page for this book:

```
(require '[clojure.java.browse :refer [browse-ur

(browse-url "https://www.manning.com/books/cloju
```

❶ `browse-url` opens the default system browser pointing at the given URL.

In the unlikely case a system browser is not available, `browse-url` attempts to open the page using the Swing window toolkit. To simulate this event, we are going to bypass `browse-url` to talk directly with private function responsible for rendering the Swing UI:

```
(require '[clojure.java.browse-ui :as bu]) ; ❶
```

```
(#'bu/open-url-in-swing "http://google.com") ; ❷
;; #object[javax.swing.JFrame 0x268234e0...]
```

❶ The content of `clojure.hava.browse-ui` is private, but we are going to bypass the restriction to show the Swing UI in action.

❷ You should see a new Java application starting up showing a very basic rendering of the Google homepage.

If the previous two methods are not available or if we need a specific way of rendering the page, we can use the dynamic variable `*open-url-script*` to provide a command line executable:

```
(require '[clojure.java.browse :refer [browse-ur

(binding [*open-url-script* (atom "wget")]  ; ❶
  (browse-url "https://tinyurl.com/wandpeace"))
;; true
```

❶ `*open-url-script*` has been bound to "wget", a popular command line browsing utility. Assuming "wget" is installed on the local system, the snippet downloads the book "War and Peace" from the Project Gutenberg website.

The basic functionalities offered by `clojure.java.browse` are not very practical to use from an application or the REPL. However, they can be used to open the documentation related to a function or a macro. This is achieved in the clojure.java.javadoc namespace that we are going to see at the end of this chapter.

22.6 CLOJURE.JAVA.SHELL

The `clojure.java.shell` namespace exposes a single entry point function `sh` which executes a command on the host operative system as a separate process:

```
(require '[clojure.java.shell :refer [sh]]) ; ❶

(sh "ls" "/usr/share/dict") ; ❷

;; {:exit 0, ; ❸
;;  :out "README\nconnectives\npropernames\nweb2
;;  :err ""}
```

❶ `sh` is the main and only entry point in the `clojure.java.shell` namespace. We can refer to the function directly and avoid using a namespace alias, as `sh` is a short and easy to recognize name.

❷ If the command line contains arguments, each argument has to be specified as a separate string. We can see here how to list the content of a folder in a Unix-based operating system.

❸ The result is a map with an `:exit`, `:out` and `:err` keys. If the `:exit` number is more than zero it indicates that the command reported an error condition. `:out` contains the output of the command directed to the standard output, while `:err` is the standard error stream.

While the command is executing in a sub-process (of the running Java Virtual Machine), `sh` blocks until the exit code is available. The command normally prints results to standard output or errors to standard error. Both outputs are reported as plain strings in the result map.

`sh` supports quite a few options. We can use the `:in` option to pass content to the standard input of the running process. `:in` supports several types: input stream, input reader, file, byte array or string. In the following example we call the Unix command "grep" passing a string to its standard input:

```
(def result (sh "grep" "5" :in (apply str (inter

(println (:out result)) ; ❷
;; 5
;; 15
;; 25
;; 35
;; 45
```

❶This `grep` command receives a string passed with the `:in` key.

❷We can see the result as it would appear outside the REPL by using println

If the input for the command is encoded in another character set (for example because it's not produced inside the running JVM) then we can use `:in-enc` to specify which encoding the input is. Similarly, `:out-enc` can be used to interpret the output from the command with a specific encoding. `:out-enc` also supports a special value `:bytes` that, when present, does not convert the output into a string, returning the raw bytes:

```
(def image-file "/usr/share/doc/cups/images/smil

(def cmd (sh "cat" image-file :out-enc :bytes))
```

```
(count (:out cmd)) ; ❷
;; 14120
```

❶This command assumes you have an image at the specified location. We could load the image with `sh` using "cat" to send the image to standard output, where it is collected as a byte array and returned. The presence of `:out-enc :bytes` prevents any conversion to a string, returning the raw bytes.

❷We can see that the image is about 14k in size.

Knowing how to pass inputs, we could build a helper function to pipeline commands together [265]:

```
(defn pipe [cmd1 & cmds] ; ❶
  (reduce
    (fn [{out :out} cmd] ; ❷
      (apply sh (conj cmd :in out)))
    (apply sh cmd1)
    cmds))

(println
  (:out
    (pipe ; ❸
      ["env"]
      ["grep" "-i" "java"])))
```

```
;; JAVA_HOME=/Library/Java/JavaVirtualMachines/j
;; JAVA_MAIN_CLASS_61966=clojure.main
;; _=/usr/bin/java
;; LEIN_JAVA_CMD=java
;; JAVA_MAIN_CLASS_62001=clojure.main
```

❶ The `pipe` function accepts at least one parameter and possibly more. It expects each parameter to be a vector of strings suitable for `sh` commands.

❷ `pipe` always starts by executing the first command, which becomes the initial value for reduce. The reducing function takes the last command output and the new command. It then executes the new command using the output of the previous as input for the next one.

❸ We can try to `pipe` the "env" command (which returns the list of all environment variables currently set) and the "grep" command which searches for sub-strings. Please note: the output you see here could be very different from the same command executed on your environment.

`sh` executes the command in a sub-process, which means that all the parent environment variables are inherited by the children. We can change this behavior and pass a completely different set of variables using the `:env` map:

```
(def env
  {"VAR1" "iTerm.app"
   "VAR2" "/bin/bash"
   "COMMAND_MODE" "Unix2003"})

(println
  (:out
    (sh "env" :env env))) ; ❶

;; VAR1=iTerm.app
;; VAR2=/bin/bash
;; COMMAND_MODE=Unix2003
```

❶We can see that the default environment variables have been completely replaced by the content of the map `env`.

Another overridable default for `sh` is the working folder, the initial path from which the command executes. In the following example we first print the current working folder and then we change it using the `:dir` key:

```
(println (:out (sh "pwd"))) ; ❶
;; /Users/reborg/prj/my/book
```

```
  (println (:out (sh "pwd" :dir "/tmp"))) ; ❷
  ;; /tmp
```

❶We execute the command "pwd" to retrieve the current working folder.

❷We execute the same command again, but this time we use the `:dir` key to pass a different working folder.

We might need to execute multiple `sh` invocations using the same settings for the environment and the working folder. To help avoid repeating the `:dir` and `:env` key in all `sh` commands, `clojure.java.sh` also contains two handy macros to set those inside a thread-local binding:

```
  (require '[clojure.java.shell :as shell :refer [

  (shell/with-sh-dir "/usr/share" ; ❶
    (shell/with-sh-env {:debug "true"}
      [(sh "env") (sh "pwd")]))

  ;; [{:exit 0, :out "debug=true\n", :err ""}
  ;;  {:exit 0, :out "/usr/share\n", :err ""}]
```

❶We use `with-sh-dir` and `with-sh-env` to set the working folder and the environment variables for all `sh` commands inside in the form.

[265] In Unix-like systems, a pipeline (or "pipe" for short) is an inter-process communication mechanism

22.7 CLOJURE.CORE.SERVER

`clojure.core.server` contains functions to expose the Clojure environment through a socket connection and across network boundaries. While a typical REPL accepts command from standard input and prints results to standard output, a socket-based REPL uses sockets to receive requests and send responses. On the other side of the socket, a process (or human) consumes the results of invoking Clojure functions as usual. By default, `clojure.core.server` uses a slightly modified version of the same REPL offered through the console. To start the server we use `start-server`:

```
(require '[clojure.core.server :as server]) ; ❶

(server/start-server ; ❷
  {:name "repl1" :port 8787 :accept clojure.core
;; #object["ServerSocket[addr=localhost/127.0.0.
```

❶ The functions required to start the socket server are available in the `clojure.core.server` namespace.
❷ `start-server` takes a map of mandatory options (others are optional).

The socket server is highly configurable. Here's a summary of the available options and their meaning:

- `:address` is the network interface the server should be using. It defaults to 127.0.0.1, the default host interface, so it's not required.
- `:port` is the port the server should be using. There is no default and it is mandatory argument.
- `:name` is an identifier for this server. There could be many socket servers running and each server requires a name. This is mandatory argument.
- `:accept` is a fully qualified function declared in a file available from the classpath. It's not possible to pass an anonymous function.
- `:args` is a list of optional arguments for the `accept` function.

- `:bind-err` tells the server if the standard error stream should be bound to the output socket (normally standard error is bound to `*err*` at the REPL). It defaults to `true` which means that anything printed to the standard error is also sent to the other end of the socket (along with standard output).
- `:server-daemon` determines if the running socket server is a daemon thread. By default, the socket server starts a daemon server, which means that the JVM can exit even if it's still serving requests.
- `:client-daemon` configures client threads as daemon. When the socket server receives a request, it handles the request as a separate client thread. By default, also the threads serving incoming requests are daemons.

Most of the available options are fairly self-explanatory. One that deserves some attention is `:accept` which determines the behavior of the server on handling an incoming request. By default `:accept` uses `clojure.core.server/repl` which starts a new repl loop. The following example runs on top of an existing REPL, using standard input/ouput instead of socket networking, but with similar effects:

```
(clojure.core.server/repl) ; ❶
;; nil

:repl/quit  ; ❷
;; nil
```

❶The effect of calling `clojure.core.server/repl` is not immediately visible. Under the hood, a new "while true" REPL loop has started to handle incoming requests. The loop would normally be attached to a socket, but here we can interact with it from the standard input/output.

❷One difference from the normal REPL is that a socket REPL needs a way to handle exit requests without necessarily typing "CTRL+D", as the remote side of the socket might not have a keyboard. The socket server adds a `:repl/quit` command that exits the REPL loop. In this case it returns to the outer REPL.

If we want to customize the socket REPL experience, we need to pass the `:accept` option a different function. The following example comes from the Replicant library, a small proof of concept by the same author of the socket server feature in Clojure [266]:

```
(ns data-server
  (:require [clojure.main :as main]
            [clojure.core.server :as server])
  (:import [java.net ServerSocket]
           [java.io BufferedWriter StringWriter]

(defn data-eval [form] ; ❷
  (let [out-writer (StringWriter.)
        err-writer (StringWriter.)
        capture-streams (fn []
                          (.flush *out*)
                          (.flush *err*)
                          {:out (.toString out-w
                           :err (.toString err-w
    (binding [*out* (BufferedWriter. out-writer)
              *err* (BufferedWriter. err-writer)
      (try
        (let [result (eval form)]
          (merge (capture-streams) {:result resu
        (catch Throwable t
          (merge (capture-streams) {:exception (

(defn data-repl [& kw-opts] ; ❸
  (println kw-opts)
  (apply main/repl
    (conj kw-opts
      :need-prompt (constantly false)
      :prompt (constantly nil)
```

```
        :eval data-eval)))

  (server/start-server
    {:name "repl2" :port 8788 :accept 'data-server/
```

❶ We need to remember that the accept function needs to be fully qualified. To make sure the example runs in the correct namespace, we create one before defining functions.

❷ `data-eval` is the evaluation function the REPL loop will use after reading from the socket. We don't use the default clojure.core/eval because standard output and standard error would not be visible on the other side of the socket. `data-eval` instead captures standard output and standard error on a `StringWriter` instance. The writer is then used to push the output through the socket by transforming it into a string.

❸ `data-repl` is a thin layer over clojure.main/repl so we can pass our custom evaluation function.

❹ We can now start a new server using the custom `:accept` function.

If you have Telnet installed [267] you can open a session to the running socket server like this:

```
> telnet 127.0.0.1 8788 ; ❶
Trying 127.0.0.1...
```

```
Connected to localhost.
Escape character is '^]'.
nil
clojure.core=> (+ 1 1) ; ❷
2
```

❶An example Telnet session that connects to the socket server to evaluate a simple expression.

❷Once connected we are offered the usual prompt and we can evaluate expressions as usual.

To stop one of the running socket server, we can use `stop-server` or `stop-servers` to close them all with a single call:

```
(server/stop-server "repl2") ; ❶
;; true

(server/stop-servers) ; ❷
;; nil
```

❶ `stop-server` requires a server name (or it will try to use the `server/**session**` dynamic variable if no server name is given). In this example we stop the server started previously. `stop-server` returns `true` when successful, or `nil` if no server was found with that name.

❷ Alternatively, `stop-servers` stops all running instances at once, without the need to pass their names.

It's worth remembering that one of the main goal of the socket server is to start a distributed REPL on top of an already existing application without the need to change the code, just configuration. We can open a socket server from any application by passing the necessary parameters to the command line:

```
; ❶
export M2_REPO=~/.m2/repository
export CLOJURE_18="$M2_REPO/org/clojure/clojure/

; ❷
java -cp .:$CLOJURE_18 \
-Dclojure.server.repl="{:port 8881 :accept cloju
clojure.main
```

```
  ;; Clojure 1.8.0
  ;; user=>
```

❶There are a few requirements for this example to work properly. You need a maven repository (this is usually there already if you use Leininghen) and change the environment variable `M2_REPO` to point at the root of that repository. By default, the Maven repository is installed in `.m2/repository` in the home folder. We are using here a Clojure 1.8 jar installation.

❷We start Clojure using the `clojure.main` class directly. We also set the `clojure.server.repl` Java property. The socket server checks for the presence of this property and starts one or more servers if as configured. As you can see, the property content is a Clojure map.

You should be able to open a Telnet connection to `127.0.0.1 8881` as before. To stop the server and the running Clojure instance, just type `CTRL+C` at the REPL above.

[266] The Socket REPL feature was implemented by Alex Miller in collaboration with the Clojure core team. You can find the Replicant library here: github.com/puredanger/replicant

[267] The venerable Telnet protocol is a way to utilize a terminal over the network. Telnet is also the name of the client utility that connects to remote sockets, not just the protocol.

22.8 CLOJURE.JAVA.IO

`clojure.java.io` contains a collection of functions to simplify the interaction with the Java Input/Output system (or simply IO). Over the years, Java evolved the original `InputStream` and `OutputStream` abstractions into `Reader` and `Writer`, eventually adding also asynchronous IO. During this transformation, Java put a lot of effort in maintaining backward compatibility, a principle also shared with Clojure. Unfortunately, there are now coexisting IO APIs that impact negatively on usability, forcing Java developers through bridges and adapters to move between different styles of IO.

The `clojure.java.io` does not implement a new IO system, but does a great job at shielding Clojure developers from most of the inconsistencies generated by the fragmented Java interface. This is achieved by a few polymorphic multimethods that can be further extended if needed. The namespace also

contains a few utility functions to work with the classpath, files and URLs.

Streams, Writers and Readers

Let's start by illustrating the following functions:

- `clojure.java.io/reader` and `clojure.java.io/writer` produce a `java.io.BufferedReader` and `java.io.BufferedWriter` objects respectively. They accept a variety of input types like readers, streams, files, URLs, sockets, arrays and strings. The fact that a `reader` accepts a `java.io.InputStream` for example, removes most of the boilerplate required in Java to move between the two abstractions.
- `clojure.java.io/input-stream` and `clojure.java.io/output-stream` produce a `java.io.InputStream` and `java.io.OutputStream` objects respectively. They accept the same input types as `reader` and `writer`, including accepting a `reader` as input (transforming between reader/writer and input/output streams).

In the following example we can see how to create a `reader` from a file. The reader should keep in mind that, in general, IO objects allocate resources on the host operating system and they need to be released or closed. We can use `with-open` to release resources after use:

```
(require '[clojure.java.io :as io]) ; ❶

(with-open [r (io/reader "/usr/share/dict/words"
  (count (line-seq r))) ; ❸
;; 235886
```

❶ `clojure.java.io` is usually aliased as `io`.
❷ `reader` interprets the first string argument as a path to a file or remote URL.
❸line-seq creates a lazy sequence by reading line items from the reader object.

Sometimes it's useful to create a `reader` from a string (especially for testing), but `reader` interprets strings as locations. We can achieve the desired effect by transforming the string into a character array first:

```
(require '[clojure.java.io :as io])
```

```
(def s "string->array->reader->bytes->string")

(with-open [r (io/reader (char-array s))] ; ❷
  (slurp r))                              ; ❸
;; "string->array->reader->bytes->string"
```

❶ `io/reader` is commonly used to load external resources. Sometimes, especially for testing, it's useful to create a `reader` directly from a string. We use this simple string for illustrative purposes.

❷ `char-array` transforms the string into a primitive array of chars, preventing `reader` interpretation of the string as location.

❸slurp has polymorphic behavior similar to `reader` and in this case transforms the `reader` back into a string by reading its content.

Note

The book contains other interesting examples of use of `io/reader`: in line-seq we show how to read from a `java.io.InputStream`. In disj instead, we can see an example about how to read from a `java.net.Socket` object.

Not surprisingly, `writer` creates a new writer object accepting the same first argument types as `reader`:

```
(with-open [w (io/writer "/tmp/output.txt")] ; ❶
  (spit w "Hello\nClojure!!")) ; ❷

(println (slurp "/tmp/output.txt")) ; ❸
;; Hello
;; Clojure!!
nil
```

❶Using a `writer` is very similar to using a `reader`. `writer` creates the object "w" that will automatically close at the end of the expression thanks to with-open.

❷spit sends the content of a string into a file. If the file already exists, the content is overwritten.

❸To test the content of the file, we can use slurp instead of passing through a `reader`.

When data processing consists of reading from a large file, operate some transformations and write the results back to disk, we can chain a `reader` and a `writer` together and process data using lazy functions like line-seq. By operating this way, we avoid to load the entire input into memory:

```
(require '[clojure.java.io :refer [reader writer
(require '[clojure.string :refer [upper-case]])

(with-open [r (reader "/usr/share/dict/words") ;
            w (writer "/tmp/words")]
  (doseq [line (line-seq r)] ; ❷
    (.append w (str (upper-case line) "\n")))) ;
;; nil
```

❶Both `reader` and `writer` need to be closed after use. In this example we use the dictionary file presents on most Unix-based systems. The file is large and we want to avoid to load its entire content to memory.

❷Using doseq, we make sure that side effects are evaluated lazily and without holding the head of sequence. The net effect is that just a small portion of the file is present in memory at any given time, while the garbage collector can claim any processed item that was already written to disk.

❸We wouldn't be able to use spit repeatedly, because the first call would close the `writer`.

`reader` accepts an `:encoding` key, while `writer` accepts both an `:encoding` and `:append` keys. The `:encoding` key forces a specific encoding for reading or writing data. The `:append` key forces `writer` to append new data at the end of

the output stream. The following example shows how to force "UTF-16" encoding (instead of the default "UTF-8") and removes the need to explicitly call the `.append` method (compared to the previous example):

```
(with-open [r (reader "/usr/share/dict/words" :e
            w (writer "/tmp/words" :append true
  (doseq [line (line-seq r)]
    (.write w (str (upper-case line) "\n")))) ;
```

❶We can use `:append` to prevent `writer` from removing any previous content from the file while writing new content.
❷Instead of using the `.append` method we can now using the more generic `.write` and control the behavior using configuration options.

Please note that forcing "UTF-16" encoding in the example above only makes sense if the input file is written with that encoding. It was used in the example for illustration purposes.

Resources and URLs

The examples throughout the book are showing `reader` or slurp loading resources using strings to indicate their location. `reader` interprets the given string similarly to an URL

(Uniform Resource Locator). An URL is a convention to format strings to encode the location of resources across the network. Confusingly enough, a `java.io.File` object has methods to transform it into an URI (Uniform Resource Identifier) but not an URL. An URI is a slightly more general object than an URL [268].

In Java programming, URL and URI need some convoluted transformations to use with readers and files. Clojure hides this complexity away allowing us to create a reader from a file or a string without thinking about such conversions. More specifically, the following functions from `clojure.java.io` are available to deal with resource and locations:

- `resource` retrieves an URL object given a string representing the location of a resource on the Java classpath. Resources on the classpath are different from resources on disk as their location is independent from the location of the running Java executable.
- `as-url`: creates an URL object given a string representing its location (it could be classpath, local file system, or other protocols such as "http").

`resource` is quite common in Clojure programming to retrieve resources from the Java classpath. The classpath normally contains compiled Java classes, Clojure sources (unless they are explicitly removed) or other artifacts. We could for example retrieve the source of the `clojure.java.io` namespace with the following:

```
(require '[clojure.java.io :refer [resource read

(def cjio (resource "clojure/java/io.clj")) ; ❶

(first (line-seq (reader cjio))) ; ❷
;; ";   Copyright (c) Rich Hickey. All rights re
```

❶Clojure sources are packaged as part of the Clojure executable. We can find them using the relative path of the file inside the Jar archive.

❷We can see the first line of the file after using a `reader` and line-seq.

`as-url` is a small utility function to create URL objects (without the need of importing `java.net.URL` to use its constructor directly). `as-url` adds some level of polymorphism to handle input types other than strings:

```
(require '[clojure.java.io :refer [as-url file]]
(import 'java.nio.file.FileSystems)

(def path ; ❶
  (.. FileSystems
      getDefault
      (getPath "/tmp" (into-array String ["words
      toUri))

(def u1 (as-url "file:///tmp/words")) ; ❷
(def u2 (as-url (file "/tmp/words"))) ; ❸
(def u3 (as-url path))                ; ❹

(= u1 u2 u3) ; ❺
;; true
```

❶ `path` shows how to convert a Java NIO (New IO Api) path into an URI.
❷ `as-url` accepts strings (with protocols) to identify a location on disc of a file.
❸ `as-url` also accepts the same location as a `java.io.File` object.
❹Finally, `as-url` also accepts an URI as the result of passing through a `java.nio.file.Path` object.
❺The 3 urls are different objects, but they represent the same location on disk of the file "/tmp/words".

Unfortunately, `clojure.java.io` doesn't handle coercions or transformations of `java.nio.file.Path` objects directly, as demonstrated by the previous example where we had to explicitly call `toUri()` before calling `as-url`. But `clojure.java.io` can be extended to handle `java.nio.file.path` (and similarly other protocols):

```
(require '[clojure.java.io :as io])
(import '[java.nio.file Path FileSystems])

(extend-protocol io/Coercions ; ❶
  Path
  (as-file [path] (io/file (.toUri path)))
  (as-url [path] (io/as-url (.toUri path))))
```

```
(def path ; ❷
  (.. FileSystems
      getDefault
      (getPath "/usr" (into-array String ["share

(io/as-url path) ; ❸
;; #object[java.net.URL 0x1255fa42 "file:"/usr/s

(io/file path) ; ❹
;; #object[java.io.File 0x1c80a235 "/usr/share/d
```

❶ `clojure.java.io` contains the `Coercions` protocol declaring two functions, `as-file` and `as-url`. While `as-file` has the `file` wrapper function available, `as-url` doesn't have a corresponding `url` function. The implementation consists of transforming the path into an URI and call the corresponding (and already existing) implementations.

❷ Java NIO Path objects are roughly equivalent to URLs. `java.nio.file.Path` only has a translation into URI available that we can use to create an URL. The `getPath()` method takes a first "root" argument of the initial part of the path, followed by any other segment as a variable argument type. Clojure needs to create an array of strings to be compatible with the type signature.

❸ After extending the protocol, we can use `as-url` to transform `java.nio.file.Path` directly.

❹ As a bonus, also `file` can now create a file object directly from a path.

Dealing with Files

Dealing with files is another important aspect in any programming language. Clojure relies on `java.io.File` for file operations and `clojure.java.io` contains a few utility function to deal with files.

We've already seen `io/file` in action multiple times in this section and throughout the book. The functon takes one or more arguments. When only one argument is present, it could be a string, another file, an URL or URI (or, less interestingly, `nil`):

```
(require '[clojure.java.io :as io])

(keys (:impls io/Coercions))        ; ❶
;; (nil java.lang.String java.io.File java.net.U

(io/file "/a/valid/file/path")
;; #object[java.io.File 0x7936d006 "/a/valid/fil

(io/file (io/file "/a/valid/file/path"))
;; #object[java.io.File 0x3f46ce65 "/a/valid/fil

(io/file (io/as-url "file://a/valid/url"))
;; #object[java.io.File 0x7af35ada "/valid/url"]

(io/file (.toURI (io/as-url "file://a/valid/uri"
;; #object[java.io.File 0x2de6a5c8 "/valid/uri"]

(io/file nil)
;; nil
```

❶ We can see what single argument types `io/file` accepts by checking the `:impl` key of the `Coercions` protocol. What follows is a list of all the possible calls to `io/file` with the respective argument types.

The default list of types that `io/file` can understand is visible inside the `Coercion` protocol map, as demonstrated in the example. We've already seen that, by extending this protocol, we can apply `io/file` to other argument types. `io/file` also accepts other arguments after the first, with the same type constraints. Additional arguments have to be relative paths (i.e., they cannot start with a forward slash '/'):

```
(io/file "/root" (io/file "not/root") "filename.
;; #object[java.io.File 0x6898f182 "/root/not/ro

(io/file "/root" (io/file "/not/relative") "file
;; IllegalArgumentException /not/relative is not
```

❶ All arguments to `io/file` after the first need to be relative paths.

❷ Here the second argument starts with '/' which denotes another root path after the first.

`io/file` does not actually create a physical resource, but just a "pointer" that other functions like writer can use to write content to. Another way to create content is to copy one file to another using the `io/copy` function:

```
(require '[clojure.java.io :as io])

(io/copy "/usr/share/dict/words" (io/file "/tmp/
;; nil

(.exists (io/file "/tmp/words2")) ; ❷
;; true
```

❶We can use `io/copy` to copy the existent `/usr/share/dict/words` file into a new file in the `/tmp` folder.

❷To check if the file was actually created, we can use the `exists()` on the `java.io.File` object.

`io/copy` supports many type combinations: from reader to writer, from string to file, from `InputStream` to `OutputStream` and so on. One of them, from file to file, is specifically optimized using `java.nio.channel.FileChannel` which guarantees optimal performance when the file is cached by the operative system.

`io/copy` however, does not support a string to string transfer (with a file to file copy implementation). We can extend `io/copy` using the related `do-copy` multimethod:

```
(require '[clojure.java.io :as io])

(defmethod @#'io/do-copy [String String] [in out
  (apply io/copy (io/file in) (io/file out) opts

(io/copy "/tmp/words2" "/tmp/words3") ; ❷
;; nil

(.exists (io/file "/tmp/words3")) ; ❸
;; true
```

❶The defmethod definition for `io/do-copy` is private in `clojure.java.io` but we can still make access to it by looking up the related var object (with the reader macro `#'`) and then dereferencing the var with `@` (another reader macro). The implementation simply call `io/file` on each argument.

❷ `io/copy` now accepts a pair of strings as arguments.

❸We can verify the file was effectively created.

The `io/do-copy` signature shows that `io/copy` accepts options:

- `:buffer-size` defaults to 1024 bytes and can be used when the origin argument is an `InputStream`.
- `:encoding` is similar in use to the same option we've seen for `reader` and `writer`, forcing a specific encoding. It defaults to "UTF-8".

When a file path requires sub-folders, but those are not existing yet, we can use `make-parents` to create all folders recursively. `make-parents` does not create the last path segment, considering it the name of a file:

```
(require '[clojure.java.io :as io])

(def segments ["/tmp" "a" "b" "file.txt"]) ; ❶

(apply io/make-parents segments) ; ❷
;; true

(io/copy (io/file "/tmp/words") (apply io/file s
;; nil

(count (line-seq (io/reader (io/file "/tmp/words
;; 235886
```

```
(count (line-seq (io/reader (apply io/file segme
;; 235886
```

❶Instead of a single string containing the path, we assembled the path out of fragments.

❷ `make-parens` creates any non-existent folder, but does not try to interpret "file.txt" as one, considering it a file name instead.

❸The same fragments of file name can be used with `io/file` to copy content over to the new folder.

❹We can check if the content was correctly copied comparing lines at origin with the destination.

We can use `delete-file` to remove files. The types supported are the same as `io/file`. We can additionally pass a second argument if we want to prevent `delete-file` to throw an exception in case of error:

```
(require '[clojure.java.io :as io])

(io/delete-file "/does/not/exist") ; ❶
;; IOException Couldn't delete /does/not/exist

(io/delete-file "/does/not/exist" :ignore) ; ❷
;; ignore
```

```
(io/delete-file "/tmp/a/b/file.txt" "This file s
;; true
```

❶When we try to delete a file that does not exist, `delete-file` throws exception.

❷We can prevent the exception in case of non existent files, by passing a second argument which is returned to signal that the operation was not successful.

❸This file was created previously and should exist on the file system. `delete-file` correctly returns `true`.

`as-relative-path` retrieves the path from resources objects (such as files, URIs, URLs). This is especially useful to convert file objects into path strings for further processing:

```
(require '[clojure.java.io :as io])

(def folders ["root/a/1" "root/a/2" "root/b/1" "

(map io/make-parents folders) ; ❷
;; (true false true true true)

(map io/as-relative-path (file-seq (io/file "roo
;; ("root" "root/a" "root/c" "root/c/1" "root/b"
```

❶We have a group of nested folders as a vector of strings.

❷We can use `make-parents` to create all the necessary folders. Note that folders don't start with "/" (on a Unix system this means they are not absolute paths).

❸After creataing a sequence of all the files within "root" with file-seq, we can extract their path strings with `as-relative-path`.

[268] The RFC describing URIs is available from tools.ietf.org/rfc/rfc3986.txt

22.9 CLOJURE.TEST

`clojure.test` is a testing framework included with the standard Clojure distribution. It works by attaching specific metadata to var objects to mark them as testing functions. This mechanism is the same adopted by clojure.core/test but `clojure.test` offers many additional features, such as:

- Nesting and composition of multiple testing contexts.
- Expressive and extensible assertion language with is and are.
- Extensible output format.
- "Fixtures" functions to setup and teardown testing contexts.

The design of a good test suite, including how to structure, implement and build tests, is beyond the scope of this book. Nevertheless this section is going to provide a good overview of available functions and extension points.

Creating tests

`clojure.test` offers several ways to create tests. To create somewhat realistic testing examples, we are going to use the following function to find the square root of a number using the Newton method:

```
(defn sqrt [x] ; ❶
  (when-not (neg? x)
    (loop [guess 1.]
      (if (> (Math/abs (- (* guess guess) x)) 1e
        (recur (/ (+ (/ x guess) guess) 2.))
        guess))))
```

❶The `sqrt` function calculates an approximation of the square root for the number "x" to the 8th decimal point. The rest of the section uses this function as an easy testing target.

Note

`clojure.test` is one of the few idiomatic use of `:refer :all` in a require declaration. `clojure.test` provides a "fluent interface" [269] and the use of qualified symbols would break readability of unit tests.

The most common way to define tests is `deftest`:

```
(require '[clojure.test :refer [deftest]]) ; ❶

(deftest sqrt-test (assert (= 2 (sqrt 4)) "Expec

(:test (meta #'sqrt-test)) ; ❸
;; #object[user$fn__1826 0xeb0db1f "user$fn__182

(test #'sqrt-test) ; ❹
;; AssertionError Assert failed: Expecting 2
;; (= 2 (sqrt 4))
```

❶ Although it's customary to `:refer :all` the entire `clojure.test` namespace, we limit ourself to what is necessary for a specific example to avoid any possible confusion.

❷ `deftest` creates a new function `sqrt-test` in the current namespace,

❸ It then adds a meta key `:test` to the var object `sqrt-test` using the body of the function as value.

❹ We can use clojure.core/test to verify that the tests are running as expected.

Note

`clojure.test` offers better assertions primitive than assert to set expectations. We are going to see them later on in this section.

`deftest` is a macro that generates a new function in the current namespace, as we can see from the related macroexpansion:

```
(macroexpand '(deftest sqrt-test)) ; ❶
;; (def sqrt-test
```

```
  ;;   (clojure.core/fn []
  ;;     (clojure.test/test-var (var sqrt-test))))
```

❶ `deftest` produces a var definition with def which contains a function.

If we don't want this function to be public, we can use `deftest-`:

```
  (require '[clojure.test :refer [deftest-]])

  (deftest- this-is-not-public) ; ❶

  (keys (ns-publics *ns*)) ; ❷
  ;; (clojuredocs help find-name sqrt sqrt-test
  ;;  cdoc cjio apropos-better)
```

❶We can use `deftest-` to create private test functions. These execute normally during a test run.

❷The public var definitions in the namespace does not contain the test function, confirming it is private. Please note: the results list in this example could be different when run in a different REPL session.

A slight variation on `deftest` is `with-test`. `with-test` creates the target function and the test definition at the same time and does not require the creation of an auxiliary function to hold the test implementation:

```
(require '[clojure.test :refer [with-test]])

(with-test ; ❶
  (defn sum [a b] (+ a b))
  (println "test called"))

(test #'sum) ; ❷
;; test called
;; :ok ; ❸
```

❶ `with-test` is the simplest macro to create a test other than setting the metadata manually.

❷ We call clojure.core/test on the target function itself instead of the generated test function like in the case of `deftest`.

❸ The ":ok" printed here is the return value from test assuming that the lack of exceptions means the test was successful.

Assertion Language

`clojure.test` offers better way to verify expectations other than the basic assert. For example, `is` verifies that the given

expression is truthy and produces a nice summary of the test results:

```
(require '[clojure.test :refer [is deftest test-

(deftest sqrt-test (is (= 2 (sqrt 4)) "Expecting

(test-var #'sqrt-test) ; ❷
;; FAIL in () (form-init796879.clj:1) ; ❸
;; Expecting 2
;; expected: (= 2 (sqrt 4))
;;   actual: (not (= 2 2.0000000000000002))
```

❶Compared to the previous example using `deftest` we replaced assert with `is`.

❷We started using `test-var` instead of `clojure.core/test`. There is no much difference, but `test-var` removes the confusing `:ok` that `clojure.core/test` generates.

❸`is` interacts with `clojure.test` 's report system and produces nicer looking results on the screen.

Thanks to `is` printed summary we can finally see why tests are failing. None of the `assert` variants seen so far was

printing the reason for the failure. `is` also takes an optional string to better describe what the test is about.

To make test results even more descriptive, we can enrich and group tests using `testing`:

```
(require '[clojure.test :refer [is deftest testi

(deftest sqrt-test
  (testing "The basics of squaring a number" ; ❶
    (is (= 3 (sqrt 9))))
  (testing "Known corner cases"
    (is (= 0 (sqrt 0)))
    (is (= Double/NaN (sqrt Double/NaN)))))

(test-var #'sqrt-test) ; ❷

;; FAIL in () (form-init796879.clj:3)
;; The basics of squaring a number
;; expected: (= 3 (sqrt 9))
;;   actual: (not (= 3 3.000000001396984))
;;
;; FAIL in () (form-init796879.clj:5)
;; Known corner cases
;; expected: (= 0 (sqrt 0))
;;   actual: (not (= 0 6.103515625E-5))
;;
;; FAIL in () (form-init796879.clj:6)
```

```
  ;; Known corner cases
  ;; expected: (= Double/NaN (sqrt Double/NaN))
  ;;   actual: (not (= NaN 1.0))
```

❶We use `testing` to group related group of tests together. This has the effect of visually grouping the tests improving readability and also appears as a description in the output of the tests.

❷It seems that we have quite a bit of work to do to make the `sqrt` function more robust.

In the previous example we started stacking up groups of similar tests, all repeating the same operation with different values. `are` builds up on `is` offering a way to batch together many similar assertions:

```
(require '[clojure.test :refer [are deftest test

(deftest sqrt-test
  (are [x y] (= (sqrt x) y) ; ❶
    9 3
    0 0
    Double/NaN Double/NaN))

(test-var #'sqrt-test) ; ❷
```

```
  ;; FAIL in () (form-init7968799.clj:2)
  ;; expected: (= (sqrt 9) 3)
  ;;   actual: (not (= 3.000000001396984 3))
  ;;
  ;; FAIL in () (form-init7968799.clj:2)
  ;; expected: (= (sqrt 0) 0)
  ;;   actual: (not (= 6.103515625E-5 0))
  ;;
  ;; FAIL in () (form-init7968799.clj:2)
  ;; expected: (= (sqrt Double/NaN) Double/NaN)
  ;;   actual: (not (= 1.0 NaN))
```

❶ `are` requires 3 declarations: the first is what variables will be used (in our case, "x" and "y"). The second part is a template expression that relates "x" and "y". In our case we want to see if the square of the first number is equal to the second. Finally, a list of "x","y" values to use in the template.

❷The end result is similar to multiple execution of `is`, one for each of the pairs.

Using equality as a predicate is common with `is` and `are`, but there are tests which are difficult to express with equality, for example if a function should throw exception given some input. `clojure.test` comes with an extended set of predicates, `thrown?`, `thrown-with-msg?` and `instance?` to use for cases different from equality:

```
(require '[clojure.test :refer [is deftest] :as

(deftest sqrt-test
  (is (thrown? IllegalArgumentException (sqrt -4
  (is (thrown-with-msg? IllegalArgumentException
  (is (instance? Double (sqrt nil)))) ; ❸

(binding [t/*stack-trace-depth* 3] ; ❹
  (t/test-var #'sqrt-test)) ; ❺

;; FAIL in () (form-init7968799.clj:2)
;; expected: (thrown? IllegalArgumentException (
;;   actual: nil
;;
;; FAIL in () (form-init7968799.clj:3)
;; expected: (thrown-with-msg? IllegalArgumentEx
;;   actual: nil
;;
;; ERROR in () (Numbers.java:1013)
;; expected: (instance? Double (sqrt nil))
;;   actual: java.lang.NullPointerException: nul
;;  at clojure.lang.Numbers.ops (Numbers.java:10
;;     clojure.lang.Numbers.isNeg (Numbers.java:
;;     user$sqrt.invokeStatic (form-init7968.clj
```

❶ `thrown?` verifies that the target function throws a specific exception.

❷ We can also verify that the error message matches a specific regex using `thrown-with-msg?`.

❸ `instance?` can verify if expression returns a specific type.

❹ `t/*stack-trace-depth*` is a dynamic variable available in `clojure.test` that can be used to configure how many items to display in case of exception during a test. Here we are requiring only the first 3 items.

❺ All tests are failing. The first 2 tests are failing because there is no exception thrown on passing negative numbers. The last tests forces `sqrt` to throw an exception when it shouldn't: we want `(sqrt 0)` to return 0.0.

Creating custom predicates

`clojure.test` offers ways to extend the default set of predicates through the `assert-expr` defmethod. We could for example add a `roughly` predicate that tests for equality with some tolerance (2 decimals by default). The new predicate also illustrates the use of the reporting functionality through `do-report`. `do-report` requires the presence of a group of specific keys to signal success, expected value and actual value:

```
(require '[clojure.test :refer [is deftest] :as t

(defmethod t/assert-expr 'roughly [msg form] ; ❶
  `(let [op1# ~(nth form 1) ; ❷
         op2# ~(nth form 2)
         tolerance# (if (= 4 ~(count form)) ~(las
         decimals# (/ 1. (Math/pow 10 tolerance#
         result# (< (Math/abs (- op1# op2#)) deci
     (t/do-report   ; ❸
       {:type (if result# :pass :fail)
        :message ~msg
        :expected (format "%s should be roughly %
                          op1# op2# decimals#)
        :actual result#})
     result#))

(deftest sqrt-test ; ❹
  (is (roughly 2 (sqrt 4) 14))
  (is (roughly 2 (sqrt 4) 15)))

(t/test-var #'sqrt-test)

;; FAIL in (sqrt-test) (form-init205.clj:3)
;; expected: "2 should be roughly 2.000000000000
;;   actual: false
```

❶ `roughly` follows the implementation of other built-in predicates in `clojure.test`. The defmethod returns a syntax quoted expression because processing of predicate happens at macro expansion time.

❷ The "form" is the first argument passed to `is` which contains the symbol `roughly` as the first item, 2 operands and an optional tolerance value.

❸ After calculating the result, we side-effect into the reporting system returning if the test passed or not and any additional information we would like to display.

❹ We can start using the new predicate right away. We can now assert that `sqrt` returns "roughly" 2 when the input is 4, which passes the test up to 14 decimal digits. If we ask for precision up to the 15th decimal, we get an error because the result is not precisely 2.

Running Tests

Now that we've seen how to create and increase expressiveness of our tests, it's time to look into options about how to run them. The most basic one that we've used so far is `test-var`. `test-var` takes a var object and execute the function found in the `:test` key in the var metadata, if any.

`test-vars` (plural) is very similar and takes multiple var objects to test. But the most common case is to declare all testing functions (and relative var objects) in a specific namespace. To evaluate all tests in a namespace we have several options, for example `test-all-vars`:

```
(ns my-tests) ; ❶
(require '[clojure.test :refer [is deftest] :as

(deftest a (is (= 1 (+ 2 2))))
(deftest b (is (= 2 (+ 2 2))))
(deftest c (is (= 4 (+ 2 2))))

(ns user) ; ❷
(require '[clojure.test :refer [test-all-vars]])

(test-all-vars 'my-tests)
;; FAIL in (a) (form-init205934.clj:1)
;; expected: (= 1 (+ 2 2))
;;   actual: (not (= 1 4))
;;
;; FAIL in (b) (form-init20593408.clj:1)
;; expected: (= 2 (+ 2 2))
;;   actual: (not (= 2 4))
```

❶The example switches the current namespace to `my-tests` before defining new tests the usual way.

❷When we are back to the `user` namespace we can run all tests in `my-tests` using `test-all-vars`.

`test-ns` is almost the same as calling `test-all-vars` except that it also obeys "test hooks" and additionally prints a summary. `deftest` calls can be nested at will, or composed later on by grouping them in a special function `test-ns-hook`. If `test-ns-hook` is found in the target namespace, `test-ns` executes the hook instead of all vars in the namespace:

```
(ns composable-tests)
(require '[clojure.test :refer [is deftest]])

(deftest fail-a (is (= 1 (+ 2 2)))) ; ❶
(deftest fail-b (is (= 1 (+ 2 2))))
(deftest fail-c (is (= 1 (+ 2 2))))

(defn test-ns-hook [] (fail-a) (fail-c)) ; ❷

(ns user)
(require '[clojure.test :refer [test-ns]])

(test-ns 'composable-tests) ; ❸
```

```
;; FAIL in (fail-a) (form-init2059340.clj:1)
;; expected: (= 1 (+ 2 2))
;;   actual: (not (= 1 4))
;;
;; FAIL in (fail-c) (form-init2059340.clj:1)
;; expected: (= 1 (+ 2 2))
;;   actual: (not (= 1 4))
;; {:test 2, :pass 0, :fail 2, :error 0}
```

❶The `composable-tests` namespace defines 3 failing tests.
❷We also adds a `test-ns-hook` function that calls a subset of the failing tests, skipping "fail-b".
❸Back in the `user` namespace, we use `test-ns` to see that `test-b` does not execute, demonstrating that `test-ns` is using `test-ns-hook` instead.

Continuing with test runners, `run-tests` adds a summary at the end of the run compared to `test-ns`. `run-tests` also run by default the current namespace if no arguments given:

```
(ns running-tests)
(require '[clojure.test :refer [is deftest run-t

(deftest a (is (= 4 (+ 2 2))))
(deftest b (is (= 4 (+ 2 2))))
(deftest c (is (= 4 (+ 2 2))))
```

```
(run-tests)
;; Testing running-tests
;;
;; Ran 3 tests containing 3 assertions.
;; 0 failures, 0 errors.
;; {:test 3, :pass 3, :fail 0, :error 0, :type :
```

Until now we've see how to run tests in a single namespace, but with `run-all-tests` we can run all of them in any loaded namespace. It also accepts a regular expression to filter a subset of the namespaces:

```
(ns a-new-test) ; ❶
(require '[clojure.test :refer [is deftest]])

(deftest a-1 (is (= 4 (+ 2 2))))
(deftest a-2 (is (= 4 (+ 2 2))))

(ns b-new-test)
(require '[clojure.test :refer [is deftest]])

(deftest b-1 (is (= 4 (+ 2 2))))
(deftest b-2 (is (= 4 (+ 2 2))))

(ns user)
(require '[clojure.test :refer [run-all-tests]])
```

```
(run-all-tests #".*new.*") ; ❷
;; Testing b-new-test
;;
;; Testing a-new-test
;;
;; Ran 4 tests containing 4 assertions.
;; 0 failures, 0 errors.
;; {:test 4, :pass 4, :fail 0, :error 0, :type :
```

❶Two namespaces are created containing the "new" word in their name. They contain some simple illustrative tests.

❷ `run-all-tests` run all the tests found in all loaded namespaces. If we pass the optional regular expression argument, `run-all-tests` only run matching namespaces.

Fixtures

`clojure.test` also supports fixtures. A good guiding principle to write effective unit tests, is that they should be isolated and repeatable. Unfortunately, some portion of the code cannot be completely side effect free. Fixtures can help recreating necessary preconditions for the test to run reliably. A common case is the presence of a specific file on disk, or a table in a database that the executing test needs to write results to. Once defined, the fixture can apply before the test or after.

`clojure.test` also offers the option to run fixtures at each test execution, or once only in a testing namespace.



Warning

If `test-ns-hook` is defined, fixtures never run. See below for usage of `test-ns-hook`.

To register fixtures for execution in a namespace, we can use `use-fixtures`:

```
(ns fixture-test-1
  (:require [clojure.test :refer :all]))

(defn setup [tests] ; ❶
  (println "### before")
  (tests)
  (println "### after"))

(use-fixtures :each setup) ; ❷

(deftest a-test (is (= 1 1)))
(deftest b-test (is (= 1 1)))

(run-tests) ; ❸
;; Testing fixture-test-1
;; ### before
```

```
;; ### after
;; ### before
;; ### after
;;
;; Ran 2 tests containing 2 assertions.
;; 0 failures, 0 errors.
;; {:test 2, :pass 2, :fail 0, :error 0, :type :
```

❶A fixture is a function of one argument. The argument is a single test or composition thereof. Calling the argument executes the test (or tests). Before the test executes we can setup a database, file or other resources. Similarly, we can reestablish any pre-existent condition after running the test.

❷`use-fixtures` registers a new fixture with either `:each` or `:once` semantic. In this case we expect the fixture to run for each declared test, so we use `:each`.

❸The printout confirms that the fixture function ran once for each test.

[269] A fluent interface is an APIs where functions are designed to be chained together. See also: en.wikipedia.org/wiki/Fluent_interface

22.10 CLOJURE.JAVA.JAVADOC

It's quite common for Clojure programmers to lookup a method or a class in the Java documentation. One of the main sources of documentation is the "Javadoc", a specific markup that creates documentation as part of Java sources [270]. The `javadoc` command produces HTML rendering of documented classes, structured in folders and subfolders. It also produces an "index.html" that can be open with a browser for offline or online viewing.

`clojure.java.javadoc` gives Clojure developers a way to organize offline and online locations for Javadocs. It also interfaces with the system browser to open Javadocs directly from the REPL. For example, to open the documentation for the String class, we can use `clojure.java.javadoc/javadoc`:

```
(require '[clojure.java.javadoc :as browse]) ; ❶

(browse/javadoc "this is a string object") ; ❷
;; true
```

```
  (browse/javadoc #("this fn class is not document
```

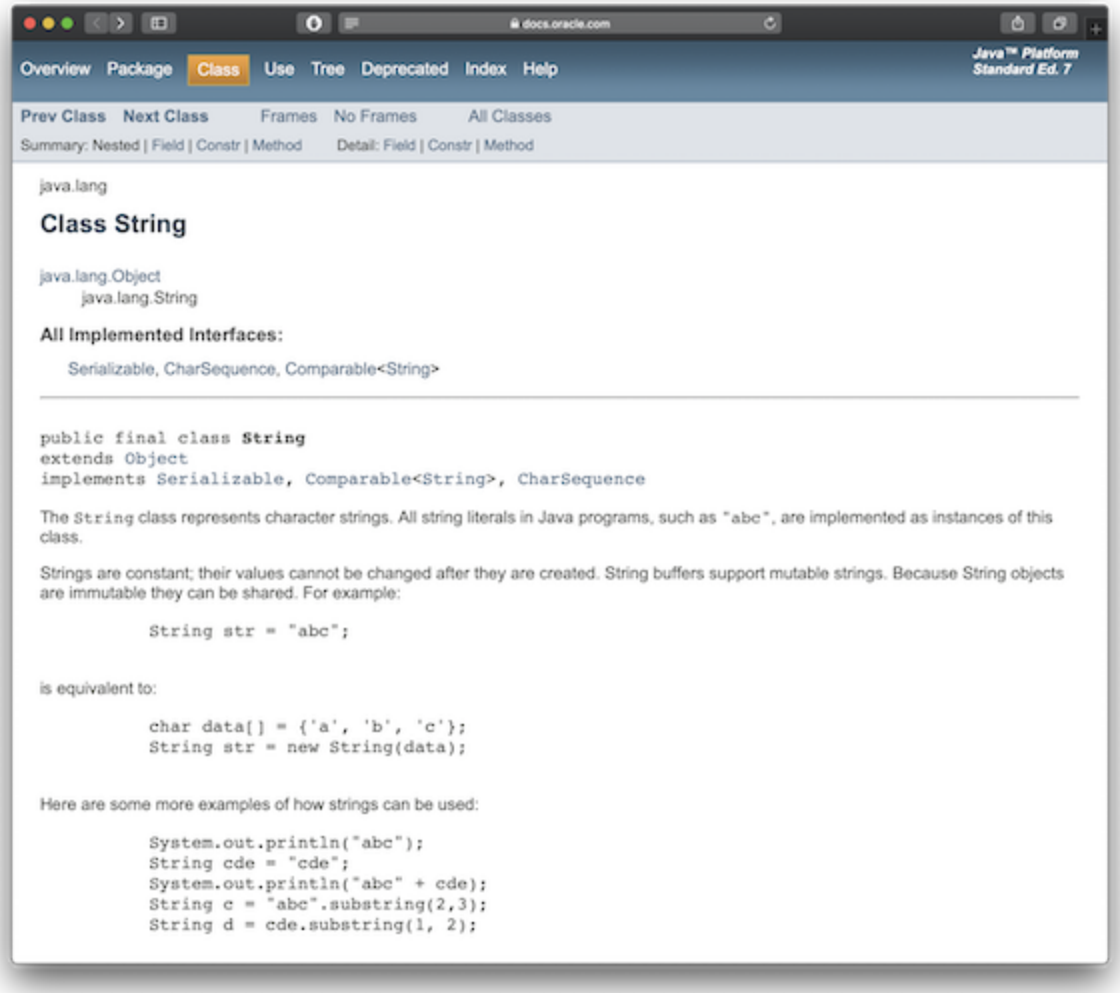
❶ `javadoc` is available after requiring the `clojure.java.javadoc` namespace.

❷ We can call `javadoc` on objects and classes. This call to `javadoc` for example is opening the system browser to the Javadoc for the String class in Java.

❸ When the given class does not have generated Javadoc (a Clojure lambda function dynamically generates a corresponding class that cannot have corresponding documentation) `javadoc` opens a generic Google search.

The image below shows how the Javadoc for the String class looks like:

Figure 22.3. Javadoc opening the documentation on the String class.



Unfortunately, `javadoc` opens by default what is now very old documentation. `javadoc` knows about Java 6 (if currently used) or 7 (for any other version). This means that even if the current REPL is running on JDK 14 (or later version) `javadoc` opens the JDK 7 documentation regardless. However, `javadoc` relies on a few dynamic vars to pass a different documentation version. There is no need to access them directly, we can use `add-remote-javadoc` to add them. The following script

checks for the current Java version and makes an educated guess at searching the corresponding Javadoc from the Oracle website:

```
(require '[clojure.java.javadoc :as browse])

(defn java-version [] ; ❶
  (let [jsv (System/getProperty "java.specificat
    (if-let [single-digit (last (re-find #"^\d\.
      single-digit jsv)))

(def jdocs-template ; ❷
  (format "https://docs.oracle.com/javase/%s/doc

(def known-prefix ; ❸
  ["java." "javax." "org.ietf.jgss." "org.omg."
   "org.w3c.dom." "org.xml.sax."])

(doseq [prefix known-prefix] ; ❹
  (browse/add-remote-javadoc prefix jdocs-templa

(clojure.pprint/pprint @browse/*remote-javadocs*
;; {"java." "https://docs.oracle.com/javase/8/do
;;  "javax." "https://docs.oracle.com/javase/8/d
;;  "org.apache.commons.codec."
;;  "http://commons.apache.org/codec/api-release/
;;  "org.apache.commons.io."
;;  "http://commons.apache.org/io/api-release/",
```

```
  ;;  "org.apache.commons.lang."
  ;;  "http://commons.apache.org/lang/api-release/
  ;;  "org.ietf.jgss." "https://docs.oracle.com/ja
  ;;  "org.omg." "https://docs.oracle.com/javase/8
  ;;  "org.w3c.dom." "https://docs.oracle.com/java
  ;;  "org.xml.sax." "https://docs.oracle.com/java
```

❶To deal with the change from double to single digit, `java-version` checks the reported Java version from the `java.specification.version` property and extract the version as single digit in case it starts with a number followed by a dot. If the reported Java version is "1.8" for instance, `java-version` returns "8" only.

❷Java Oracle published JDK documentation follows the same format for all versions, so we can just adjust the URL to the correct version.

❸ `javadoc` looks up the list of URLs for documentation using the package name of the target class. We are going to update a few of the default prefix with the new Javadoc URL which are listed in the `known-prefix` definition.

❹We repeatedly update prefixes and URLs using doseq and `add-remote-javadoc`.

❺The current list of known remote locations is visible after printing `browse/remote-javadoc`, the dynamic var responsible for storing them in the namespace.

After running the example above, `javadoc` should be able to open the documentation related to the current Java version. If the example doesn't happen to work for your Java version, it's possible that the Oracle website has changed location for Javadocs. To fix the problem, you'll need to update `jdocs-template` to point at the different URL.

NOTES: Some settings for `clojure.java.javadoc` are also available as dynamic variables. Please check the dynamic variables chapter for more information.

[270] Please have a look at en.wikipedia.org/wiki/Javadoc for an overview of Javadoc

23 DYNAMIC VARIABLES IN THE STANDARD LIBRARY

This chapter describes the dynamic variables available in the Clojure standard library. Some of them already appeared in the book, but others still deserve a more detailed explanation which is the subject of this chapter. One of the main use of dynamic variables is as communication channel across the call stack: by wrapping a function call in a binding form for example, all subsequent invocations will be able to read and use dynamic values. Dynamic variables are used extensively inside Clojure itself to expose configuration options for different features: the reader, the compiler, the printing system and much more. Some dynamic variables have been grouped into a single section when they are intimately correlated or part of the same sub-system.

23.1 *1, *2, *3 AND *E

Dynamic variable names are conventionally surrounded by a pair of asterisks (colloquially known as "earmuffs"), but `*1`, `*2`, `*3` and `*e` are an exception. One possible reason is that

they are meant to be read-only and for internal use of the Clojure REPL only:

- `*1` stores the result of the previous evaluation, if available.
- `*2` stores the result of the evaluation before `*1`.
- `*3` stores the result of the evaluation before `*2`.
- `*e` stores the exception object if the previous evaluation resulted in an error.

For example, let's have a look at the following REPL session:

```
user=> (+ 1 1)
2
user=> (+ 2 2)
4
user=> (+ 3 3)
6
user=> [*1 *2 *3] ; ❶
[6 4 2]
```

❶The results of the 3 previous evaluations is automatically stored in the corresponding dynamic variables.

`*1`, `*2` and `*3` effectively represent the index of a stack of size 3 where the REPL pushes all evaluations. If there is an

error during the evaluation of an arbitrarily complex form, we can always access the content of the exception using `*e`:

```
user=> (+ 1 (/ 1 0))
Execution error (ArithmeticException) at user/ev
Divide by zero
user=> *e ; ❶
#error {
 :cause "Divide by zero"
 :via
 [{:type java.lang.ArithmeticException
   :message "Divide by zero"
   :at [clojure.lang.Numbers divide "Numbers.jav
 :trace
 [...]
```

❶The dynamic variable `*e` is automatically bound to the last uncaught exception object for further inspection.

23.2 *IN*, *OUT* AND *ERR*

These dynamic variables are automatically bound by Clojure to objects representing the <<en.wikipedia.org/wiki/Standard_streams,standard streams>>. These objects are also bound to `System/in`, `System/out` and `System/err` respectively, which are the equivalent Java

references for standard streams. For example, after starting a REPL session, `*in*` is bound to an instance of `clojure.lang.LineNumberingPushbackReader` a Clojure class which is used by default when reading forms from the console:

```
user=> *in* ; ❶
#object["clojure.lang.LineNumberingPushbackReade
user=> (read *in*)
hello ; ❷
hello ; ❸
user=>
```

❶We can see that `*in*` is bound to a reader instance.
❷We type "hello" at the REPL and hit return.
❸Clojure uses the reader instance to read what was typed at the console up to EOF (end of line, normally `\n` "newline" character).

Similarly, we can write to the standard output `*out*` which at the REPL is bound to the console:

```
user=> *out* ; ❶
#object["java.io.OutputStreamWriter@36dfbdaf"]
user=> (.write *out* "hello\n") ; ❷
```

```
  hello ; ❸
  nil ; ❹
  user=>
```

❶If we print the content of `*out*` we can see that it is bound to an instance of `java.io.OutputStreamWriter` (or `java.io.PrintWriter` depending on the REPL implementation).

❷We can call `.write` on the output stream to send the string "hello" followed by a new line terminator.

❸The console prints our message.

❹ `nil` is the result of evaluating the side effect only form that writes into the standard output.

The error stream is traditionally an independent stream from the output of the running process. By separating them, we can keep error and diagnostic messages separated from the main output [271]. At the REPL we can see the error stream by redirecting it to the output stream:

```
  (.write *err* "error!\n") ; ❶
  nil
  user=>

  user=> (binding [*err* *out*] (.write *err* "err
```

```
  error!
  nil
  user=>
```

❶If we attempt to write to `*err*` we don't immediately see anything appearing on the screen. The message was indeed sent to the independent error stream which is not connected to the console standard output.

❷We can use binding to redirect `*err*` into `*out*` and see the error message.

We used binding to set the input/output stream in many example in the book. For additional information, the reader is invited to review the following:

- In the defmacro backoff! function, the standard output is redirected to the standard error, with the effect that any use of println in the form will print to standard error instead of standard output.
- In identical? the example shows how to create a thread of execution making sure that the standard output is redirected to the parent thread, so no messages are lost.

- In sort save-chunk! function, the standard output is bound to a file writer, so any use of pr inside the form is redirected to a file.

One interesting use of `out` is to capture the standard output into a string for later inspection (the same concept extends of course to `err`). `with-out-str` is a macro that establishes a binding of the standard output to a `java.io.StringWriter`, so it can be captured as a string:

```
(def captured-string
  (with-out-str ; ❶
    (println "This is not going to standard outp

captured-string
;; "This is not going to standard output\n"
```

❶ `with-out-str` redirects the output of functions and macros that would generally print to the standard output. The captured string can be used or printed again.

[271] On Unix based systems, we can redirect the standard error stream into the standard output using the following `2>&1` (in Bash shell) or `>&` (in C Shell)

23.3 *AGENT*

The `*agent*` dynamic variable is automatically bound to the current agent instance inside the function closure send to the agent itself:

```
(println "Is there an *agent*?" *agent*)
;; Is there an *agent*? nil ; ❶

(send
  (agent 10)
  (fn [value]
    (Thread/sleep 1000) ; ❷
    (println "Is there an *agent*?" (deref *agen
;; #object[clojure.lang.Agent 0x1ff5d0e {:status
;; Is there an *agent*? 10 ; ❸
```

❶The `*agent*` dynamic variable is always available, but not bound unless it is used as part of the agent closure.
❷This pauses the closure function sent to the agent for 1 second, so the output from evaluating the expression and the printed text don't mix up in the console.
❸When we access `*agent*` from within the agent closure, we can see that it is automatically bound to the agent instance itself.

The `*agent*` dynamic variable can be used to call the agent instance recursively, as it was demonstrated in the corresponding example in agent. The reader is invited to visit the section to know more.

23.4 *ASSERT*

`*assert*` is a dynamic variable that controls the behavior of the assert macro. During standard Clojure compilation, `*assert*` is set to `true` which configures the assert macro to throw `java.lang.AssertionError` when the given expression is false:

```
(assert (= 1 (+ 3 3))) ; ❶
;; AssertionError Assert failed: (= 1 (+ 3 3))
```

❶The standard behavior of assert is to stop the computation and raise an error when the evaluation of the expression produces a falsey result.

`*assert*` controls the compile-time behavior of the `assert` macro, so by the time Clojure is running, `assert` has already been macro-expanded and compiled with `*assert*` bound to `true`. The following example illustrates this fact:

```
(binding [*assert* false] (assert (= 1 2)))
;; AssertionError Assert failed: (= 1 2)  ; ❶

(binding [*assert* false] (eval '(assert (= 1 2))))
;; nil ; ❷
```

❶At the time binding set `*assert*` to `false`, the `assert` macro has been already expanded with `*assert*` set to `true`, so the it throws `AssertionError`.

❷If we call eval on the same form, we force evaluation of `assert` in a context where `binding` delivers the desired effect and no error is thrown.

Clojure does not offer a command line option to start Clojure and set `*assert*` to `false`, but you have some control during AOT compilation which can be used to turn off assertion

checking. In the following example we are going to create a small namespace on the file system which contains a function that uses `assert` and use compile to generate the compiled version of the namespace:

```
;; export CLOJURE_JAR=~/.m2/repository/org/cloju
;; java -cp ${CLOJURE_JAR}:$(pwd)/classes clojur
;; user=>

(require '[clojure.java.io :as io])
(def source-file (str *compile-path* "/assertion
(io/make-parents source-file) ; ❷

(spit source-file
  "(ns assertion)
   (defn select [n] ; ❸
     (case (rem n 3)
       0 :ok
       1 :possible
       2 :reject
       (assert false \"Should never happen\")))"

(compile 'assertion) ; ❹
;; assertion

(assertion/select 5)
;; :reject
```

```
(assertion/select -5) ; ❺
;; AssertionError Assert failed: Should never ha
```

❶To run this example, you need control of the classpath, so we are going to use the `java` command to start the REPL directly from the Clojure distribution jar. Please note that the location of the "clojure-x.y.z.jar" could be different on your local machine. If you don't know where it is located, you can also download a fresh Clojure installation from <<clojure.org/releases/downloads_older,Clojure website>>. Also note that other shell environments or operative systems might require a different syntax to run the command. If you run the example on Windows for instance, the classpath separator becomes ";" instead of ":".

❷We need to make sure that the "classes" folder (or any other folder that is bound to `*compile-path*`) is present on the file system, so we are going to create one if that doesn't exist with io/make-parents.

❸The "assertion" namespace contains a function `select` that allows/disallows specific numbers based on their divisibility by 3 on the assumptions that the parameter "n" will always be a positive number.

❹We can now compile the namespace for testing.

❺There are still cases that are not captured by the `select` function, for example if we pass a negative or decimal number we can still see the assertion error.

The case statement in the `select` function should cover all possible scenarios to match the reminder of an integer division by 3. However, we want to be extra-careful and added an `assert` clause to communicate our assumption. The assumption turns out not to be correct and after fixing all possible problems, we are now ready to ship our code with confidence. Here's one last test to verify it all works correctly:

```
(spit source-file
  "(ns assertion)
   (defn select [n]
     (if (integer? n)   ; ❶
       (case (mod n 3)  ; ❷
         0 :ok
         1 :possible
         2 :reject
         (assert false \"Should never happen\"))
       :reject))")

(compile 'assertion) ; ❸
;; assertion

(assertion/select -5) ; ❹
;; :possible
```

```
  (assertion/select 5.2)
  ;; :reject
```

❶ `select` now only works for integers, so it rejects any other kind of number.

❷However, it can accept negative numbers by using mod instead of rem, so the remainder of the integer division by a negative number is still positive.

❸The `assertion` namespace needs to be compiled again before use.

❹The result confirms that the bug has been fixed.

Our program is now ready to ship and we don't want the performance penalty of checking assertions. Instead of removing them, we can simply use `*assert*` to turn them off:

```
(spit source-file
  "(ns assertion) ; ❶
   (defn select [n]
     (assert false \"Never triggers\"))")

(binding [*assert* false] ; ❷
  (compile 'assertion))
;; assertion
```

```
  (assertion/select -5) ; ❸
  ;; nil
```

❶To demonstrate that the assertion never triggers, we replace the body of the `select` function with an `(assert false)` statements that will always fail unless assertions are disabled.
❷When compiling, we bind the `*assert*` dynamic variable to `false`, preventing assertions from triggering.
❸Calling `select` now returns `nil`, demonstrating that the `assert` is never evaluated.

Luckily, we don't need to perform ahead of time compilation manually as we demonstrated above. Clojure building tools like <github.com/technomancy/leiningen/blob/master/sample.project.clj#L289-L290,Leiningen> or Clojure <clojure.org/guides/deps_and_cli#aot_compilation,Deps> supports AOT compilation from the command line, including the option to set the `*assert*` variable. Java has a similar assertion facility which also includes the option of passing a JVM flag to turn assertions on or off (they are off by default) [272].

[272] The assertion facility in Java is well described in the <<docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html,Oracle official documentation>>

23.5 *CLOJURE-VERSION* AND *COMMAND-LINE-ARGS*

These two dynamic variables are meant to be read-only and used to communicate the version and the command line arguments Clojure was started with. The following example shows show how they can be read to print their results:

```
;; export CLOJURE_JAR=~/.m2/repository/org/cloju
;; java -jar $CLOJURE_JAR -r Hello Clojure ; ❶

*command-line-args* ; ❷
;; ("Hello" "Clojure")

*clojure-version* ; ❸
;; {:major 1, :minor 8, :incremental 0, :qualifi
```

❶To show an example of collecting information about the command line arguments, we start the Clojure REPL starting from the basic Java command and the location of a suitable Clojure distribution jar. In this case, the environment variable `CLOJURE_JAR` points at such location. You can see that Clojure is started with 2 command line arguments: "Hello" and "Clojure".

❷ `*command-line-args*` reports the list of command line arguments that were passed to the Clojure main class.

❸We can also access the Clojure version, returned for programmatic use already split into typical <<semver.org,semantic versioning>>: major, minor, incremental (also known as patch/fix) and qualifier (this could be an incremental build number but it's not part of the semantic versioning convention).

23.6 *COMPILE-FILES*

`*compile-files*` is a dynamic variable that indicates that Clojure compilation is in process. compile is the only function in the standard library that establishes the binding and set `*compile-files*` to `true`. The dynamic variable can be used to enable (or disable) evaluation during ahead of time (AOT) compilation, for example to prevent some part of the code to initialize with the wrong environment. A typical

example is the need to initialize some part of the system at startup:

```
(require '[clojure.java.io :as io])
(require '[clojure.string :as string])
(defonce hosts        ; ❶
  (with-open [r (io/reader "/etc/hosts")]
    (println "Loaded hosts")
    (->> r
      line-seq
      (map #(string/split % #"\t"))        ; ❷
      (filter (comp #{"127.0.0.1"} first)) ; ❸
      (remove (comp #{"localhost"} last))  ; ❹
      (map last)
      doall)))

(defn allow? [domain] ; ❺
  (not (some (partial re-find domain) hosts)))

(allow? #"reddit")    ; ❻
;; false
```

❶ `hosts` is a variable that is meant to initialize just once looking at the local "/etc/hosts" file. Note: on Linux and Mac, the file is called "/etc/hosts". On Windows, you should use "c:\Windows\System32\Drivers\etc\hosts".
❷ The lines we are interested in are in the form "<ip-address> <tab-char><domain-name>".
❸ Of all the lines, we want those routing to the IP address "127.0.0.1" in first position. This is a conventional way to prevent some domain to resolve correctly, by pointing them to a non existent web server installed on the local machine instead of allowing normal DNS resolution.
❹ However, the entry that maps "127.0.0.1" to "localhost" is legitimate, so we don't want it to appear in the final list.
❺ `allow?` uses `hosts` to verify if any entry matches the given domain.
❻ We can see that visiting "reddit" is not allowed on this computer.

The example above works as expected as far as we don't compile ahead of time. If we compile it, the list of hosts would initialize using the local computer "etc/hosts" file, but we want it to initialize differently based on the computer the application is deployed to. We have a couple of options to fix this problem. We've already seen a solution based on delay that the reader is invited to read in the section dedicated to that function. Using

`delay` is definitely a valid solution, but it requires to explicitly deref the variable holding the initialization data. Alternatively, we could use `*compile-files*` to conditionally initialize the `hosts` variable only when we are not compiling:

```
(defmacro defruntime [sym & body] ; ❶
  `(defonce ~sym
    (when-not *compile-files* ~@body)))

(defruntime hosts  ; ❷
  (with-open [r (io/reader "/etc/hosts")]
    (println "Loaded hosts")
    (->> r
      line-seq
      (map #(string/split % #"\t"))
      (filter (comp #{"127.0.0.1"} first))
      (remove (comp #{"localhost"} last))
      (map last)
      doall)))
```

❶The `defruntime` macro defines a new way to define a variable which proceeds to the evaluation of the `body` argument only when there's no compilation happening.

❷The only change we need to do is to use the `defruntime` macro instead of the previous `defonce`.

The example shows how to protect the evaluation of a specific expression based on the presence of compilation, something similar to the conditional compilation feature present in other languages like "C" or "C++". Common Lisp also contains similar (but extended capabilities) with the <www.lispworks.com/documentation/HyperSpec/Body/s_eval_w.htm#eval-when,eval-when> macro.

23.7 *COMPILE-PATH*

`*compile-path*` is used to communicate the compiler where to write compiled files. It defaults to "classes", a sub-folder of the current working folder. Clojure always compiles all forms to Java classes (that are loaded and evaluated on the fly), but it doesn't store them to disk by default. Clojure explicitly writes compiled classes to disk with compile or by invoking the main method of the `clojure.lang.Compile` class from the command line. In both cases we can request classes to be written to a specific folder (in the case of `clojure.lang.Compile` we can also use the "-Dclojure.compile.path" Java property).

Here's an example of compilation that involves a custom `*compile-path*`. In order for `compile` to work correctly with a custom path, we assume that "mypath" is a folder on disk

inside the current working folder and we can control the Java classpath when starting the REPL:

```
;; mkdir mypath
;; export CLOJURE_JAR=~/.m2/repository/org/cloju
;; java -cp $CLOJURE_JAR:mypath clojure.main ; ❶

(spit "mypath/example.clj"
  "(ns example)
   (println \"example loaded\")")

(binding [*compile-path* "mypath"] ; ❷
  (compile 'example))
;; example loaded
;; example
```

❶Note that we are starting the REPL using a specific classpath that includes the Clojure distribution and the folder "mypath".

❷If you open the "mypath" folder after compilation, you will see all the relevant Clojure compiled files for the "example" namespace.

compile works under the assumption that the compilation unit is a Clojure library, a specific convention to organize Clojure code following the Java packaging standard. However, we can bypass the standard `compile` call to write any form to disk,

not just libraries. We can achieve this using `*compile-files*` and eval in the following way:

```
(binding [*compile-files* true
          *compile-path* "."]
  (eval '(+ 1 1))) ; ❶
;; 2
```

❶This eval invocation produces compiled classes in the current working folder ".", but any other existing folder would do.

To understand why we need to `eval` a form to see its compiled output, consider that when the REPL starts, all functions and macros referencing `*compile-files*` have been already compiled with `*compile-files*` set to `false`. By calling `eval` we are going to force Clojure into re-evaluating related forms and macros with `*compile-files*` set to `true`: the form `(+ 1 1)` for example, is first compiled into Java bytecode, then its content saved to disk and finally it is evaluated to produce "2".

23.8 *COMPILER-OPTIONS*

The `*compiler-options*` dynamic variable can be used to change some aspects of the Clojure compilation. It currently

support 3 options:

- `:disable-locals-clearing` is a boolean that prevents the compiler from setting some local attributes to "null" directly after their use. This can be useful for debugging purposes, but it's normally set to `true` in production applications.
- `:elide-meta` is a vector of metadata keywords that should not be stored in emitted Java classes by the compiler. This is an optimization that reduces the general executable size and its startup time.
- `:direct-linking` is a boolean option that when `true` tells the compiler to enable "direct linking" a feature that removes the layer of indirection introduced by the need of dereferencing vars.

All the keys that can be part of the `*compiler-options*` map can also be passed as command line arguments with the `-D` option: `"-Dclojure.compiler.disable-locals-clearing=true"`, `"-Dclojure.compiler.elide-meta=[:doc :file :line :added]"` and `"-Dclojure.compiler.direct-linking=true"` respectively. We are going to explore these options in the examples below.

disable-locals-clearing

The book first mentioned "locals" when describing fn*, the only function definition form that offers control over locals clearing. To understand how `disable-locals-clearing` works, we are going to use a Java decompiler, for example one of the several available online at www.javadecompilers.com/ (modern IDE like IntelliJ or Eclipse also comes with a Java decompiler). In the following example we are going to compile a Clojure form into the corresponding Java classes so we can inspect them:

```
(binding [*compile-files* true
          *compile-path* "."]
  (eval '(let [s "potentially-huge"] (^:once fn*
```

❶eval compiles a let block and a simple lambda function that takes no arguments and returns the value of the local binding "s".

After running the example above, you should see 2 files appearing in the same folder the REPL was started in: one represents the evaluation of the let block, the other is the lambda function created by fn*. They have names similar to `user$eval29.class` and `user$eval29$fn__30.class`

(where the number could be different). We can now proceed to decompile both classes:

```
import clojure.lang.AFunction;

public final class user$eval29 extends AFunction

    public static Object invokeStatic() {
        String s = "potentially-huge";
        String var10002 = s;
        s = null;  ❶

        return new user$eval29$fn__30(var10002);
    }

    public Object invoke() {
        return invokeStatic();
    }
}

import clojure.lang.AFunction;

public final class user$eval29$fn__30 extends AF
    Object s;

    public user$eval29$fn__30(Object var1) {
        this.s = var1;
    }
```

```
    public Object invoke() {
        Object var10000 = this.s;
        this.s = null;    ❷
        return var10000;
    }
}
```

❶Two local string instances are created. The string "s" is part of the declaration of the `let` block, while "var10002" is added by the Clojure compiler to copy the content of "s" before setting "s" to "null".

❷The second class implements the `fn*` function and contains a declaration for an attribute "s" of type "Object" corresponding to the function closure. Like before, this object is set to "null" as soon as its value is copied locally.

The decompiled classes show two examples of locals clearing. The first case is explicitly requested by the user using the metadata `^{:once true}` on `fn*` and the second is related to the implementation of the let block. `disable-locals-clearing` applies to both:

```
(binding [*compile-files* true
          *compile-path* "."
```

```
            *compiler-options* {:disable-locals-cl
    (eval '(let [s "potentially-huge"] (^:once fn*
```

❶ The same expression from before is now evaluated setting the `*compiler-options*` map with the `:disable-locals-clearing` key set to `true`.

After generating new classes for the same expression, we can see that locals clearing has been disabled after setting `:disable-locals-clearing` to `true`:

```
import clojure.lang.AFunction;

public final class user$eval43 extends AFunction
    public user$eval43() {
    }

    public static Object invokeStatic() {
        Object s = "local"; ❶
        return new user$eval43$fn__44(s);
    }

    public Object invoke() {
        return invokeStatic();
    }
```

```
import clojure.lang.AFunction;

public final class user$eval43$fn__44 extends AF
    Object s;

    public user$eval43$fn__44(Object var1) {
        this.s = var1; ❷
    }

    public Object invoke() {
        return this.s;
    }
}
```

❶After turning off locals clearing, there is no local set to "null".

❷ `disable-locals-clearing` also impacts the `fn*` function with explicit requirement of locals clearing with the `^{:once true}` metadata.

The locals clearing optimization has important effects when class instances are re-used in a pool, allowing the garbage collector to free up memory in between requests (please see future for more information). However, since the Clojure nullify locals by default, this information is lost when trying to debug the class. This is one of the main reasons to use `disable-locals-clearing`.

elide-meta

`elide-meta` contains a vector of metadata keys that should be removed from compiled Clojure code. Standard metadata information like function descriptions or line/column numbers, are part of compiled Clojure code and useful for debugging or documentation. Other metadata could be part of the application to drive custom features. In general, metadata impact on the executable size or startup time is marginal. However, there could be cases where the presence of larger metadata in compiled code is not desirable. The `elide-meta` option tells the compiler to remove the given vector of metadata keys from the compilation output. To see the effect of `elide-meta`, consider the following standard compilation process:

```
;; export CLOJURE_JAR=~/.m2/repository/org/cloju
;; java -cp ${CLOJURE_JAR}:. clojure.main     ; 
(spit "./example.clj"
  "(ns example)
   (def ^{:t1 \"secret metadata\"
          :doc \"here's some useful info\"}
     foo 1)")

(binding [*compile-path* "."]     ; ❷
  (compile 'example))
```

❶We repeat here what we have done in other examples. We need to make sure that the Clojure REPL starts with the current folder "." added to the Java classpath, so everything we write in the current folder is immediately available for compilation.

❷After calling compile a few class files appear in the current folder, all starting with the prefix "example".

After calling compile in the example above, Clojure creates a few files in the current folder. One of them is called "example__init.class". This file initializes the namespace creating the "foo" variable with its metadata as indicated by the ":t1" and ":doc" keys. We can see the metadata if we decompile the file using a Java decompiler, for example one available online from www.javadecompilers.com:

```
public class example__init {

    public static void load() {
      [...]
    }

    public static void __init0() {
        [...]
        const__11 = (AFn) RT.map(new Object[] {
                RT.keyword((String) null, "t1"),
```

```
                    RT.keyword((String) null, "doc")
                    RT.keyword((String) null, "line"
                    RT.keyword((String) null, "colum
                    RT.keyword((String) null, "file"
        }
    [...]
    }
```

❶The initialization of the metadata for the `foo` variable appears here as the creation of a new map object with related keys.

The output of the decompiler has been heavily redacted to just show the relevant section. Metadata like "secret metadata" or "here's some useful info" are part of the static initialization of the related Java class. Clojure offers a way to control the information stored as metadata, thus reducing the size of the executable and potentially speeding up the generated application. We now require to elide the ":doc" and ":t1" keys from any metadata instance in the generated application:

```
  (binding [*compile-path* "."
            *compiler-options* {:elide-meta [:doc
    (compile 'example))
```

❶To prevent a specific set of metadata keys to be part of the final executable, we use `:elide-meta` from the `*compiler-options*`.

The newly decompiled "example__init.class" does not include now the `:doc` or `:tl` metadata values. Other metadata, such as the "line" or "column" indication remains available, but could be elided too if requested. It's unlikely that the Clojure program generated from the example above is going to be smaller or faster to execute. But the `elide-meta` option remains useful to remove larger or sensible metadata from the final executable.



Warning

be aware that removing the `:doc` value from the metadata, has the consequence of preventing doc from working properly. Likewise, you should pay attention not to remove metadata that are essential for the application to work properly.

---

direct-linking

Direct linking is relatively new feature introduced with Clojure 1.8 which applies to Clojure AOT compilation. When the `:direct-linking` key is set to `true` in `*compiler-`

`options*`, the compiler removes the indirection of `clojure.lang.Var` to access Clojure variables, including invoking functions. Such indirection is what allows Clojure to redefine the behavior of the system at runtime, changing function implementations while the system is running. When this option is active, instead of passing through the indirection of a var object, Clojure directly links each call sites to the corresponding invocation. The advantage is the removal of var object declarations in generated classes with a potential reduction in size and speed of the executable.

Similarly to other sections in this chapter, we'll need to decompile the generated Clojure code to see the effect of direct linking. Let's first AOT compile a small example using eval with specific bindings:

```
(binding [*compile-files* true
          *compile-path* "."
          *compiler-options* {:direct-linking fa
  (eval '(do
    (defn plus [a b] (+ a b))
    (plus 1 2))))
```

❶To begin with, we are making sure direct-linking is set to `false` to observe Clojure standard dynamic behavior.

The eval instruction above produces two files on disk, one called `user$plus.class` which contains the definition of the `plus` function. The other file will be named generically `user$eval25.class` with an increasing number at each recompilation. We proceed to decompile `user$eval25.class` using www.javadecompilers.com with the default Procyon option:

```
import clojure.lang.RT;
import clojure.lang.IFn;
import clojure.lang.Var;
import clojure.lang.AFunction;

public final class user$eval30 extends AFunction
{
    public static final Var const__0;
    public static final Object const__1;
    public static final Object const__2;

    static {
        const__0 = RT.var("user", "plus"); ❶
        const__1 = 1L;
        const__2 = 2L;
    }

    public static Object invokeStatic() {
        return ((IFn)user$eval30.const__0.getRawl
```

```
                .invoke(user$eval30.const__1,
                        user$eval30.const__2);
    }

    public Object invoke() {
        return invokeStatic();
    }

}
```

❶The class defines an attribute `const__0` of type `clojure.lang.Var` which is assigned the result of looking up the "plus" function from the pool of initialized vars. This implies the existence of a class "user$plus.class" that this var is holding a reference to.

❷To use the var object, we need to get a reference to the objects it is pointing at with `getRawRoot()`. This is the indirection that allows Clojure to swap the object referenced by the var, also at runtime, without interrupting the running program. After retrieving the referenced object, Clojure proceeds to `invoke` it with the numbers 1 and 2 of type "long".

As we can see from the decompiled class above, Clojure makes use of a `clojure.lang.Var` object to hold a reference to the code necessary to execute the "plus" function. To invoke the

function, we need to dereference the var first. This is one of the key ingredients that allows Clojure to swap the implementation of the "plus" function while the programming is running. However, such flexibility comes with a cost which is tolerable in most of the cases. If such cost becomes problematic, we can "direct link" the call site to the "plus" function without any var indirection:

```
(binding [*compile-files* true
          *compile-path* "."
          *compiler-options* {:direct-linking tr
  (eval '(do
    (defn plus [a b] (+ a b))
    (plus 1 2))))
```

❶We have now turned on direct linking.

We can now proceed to decompile the newly generated class, after we turned `:directing-linking` on:

```
import clojure.lang.AFunction;

//
// Decompiled by Procyon v0.5.36
//
```

```
public final class user$eval35 extends AFunction
{
    public static final Object const__1;
    public static final Object const__2;

    public static Object invokeStatic() {
        return user$plus    ❶
          .invokeStatic(user$eval35.const__1,
                        user$eval35.const__2);
    }

    public Object invoke() {
        return invokeStatic();
    }

    static {
        const__1 = 1L;
        const__2 = 2L;
    }
}
```

❶The result is now computed making access to the `user$plus` class directly, without any var object indirection.

The newly generated class does not contain a `clojure.lang.Var` object declaration, it doesn't lookup the

`Var` object from the pool of available vars and it doesn't access the root of the var object to get an instance of the "plus" function class. This example is too small to offer a worthwhile improvement in terms of class size or execution speed, but in normal applications the number of var objects and required indirection can be considerable. It might be worth considering turning on the `direct-linking` option, especially considering that swapping implementation dynamically is normally not necessary (or possibly dangerous) in running production applications.

Note

`direct-linking` will never operate on dynamic variables, as they might require thread-local redefinition. If you still wish to compile your application with direct linking but allow some var objects to redefine at runtime, you can use the `:redef` metadata. In this short definition `(def ^:redef doesnotdirectlink 1)` the var "doesnotdirectlink" won't be subject to direct linking even when the compiler is required to do so.

23.9 *FILE* AND *SOURCE-PATH*

`*file*` is set by the compiler to the full path of the file that is currently compiled. Similarly, `*source-path*` is set to just the file name. Let's assume the following file "examples/compile_file.clj" is available on the classpath:

```
(ns examples.compile-file)
(when *compile-files*
  (println "Compilation full path:" *file*) ; ❶
  (println "Compilation file name:" *source-path
```

❶A simple namespace which contains a println statement using the `*file*` dynamic variable.
❷A second statement prints the content of the `*source-path*` dynamic variable.

If we compile "examples/compile_file.clj" from a REPL that contains its location as part of the classpath (for example the current folder "."), we can see the following output:

```
(binding [*compile-path* "."]
  (compile 'examples.compile-file)) ; ❶
;; Compilation full path: examples/compile_file.
;; Compilation file name: compile_file.clj
;; examples.compile-file
```

❶Calling compile on the "examples.compile-file" symbol, produces a message that shows the content of both `*file*` and `*source-path*` dynamic variables.

We use *compile-files* to verify if the expression is evaluated as part of the Clojure compilation process. If it is, we print the content of `*file*` and `*source-path*`. As expected, they contain the full path and the name of the file itself. When `*file*` or `*source-path*` are not bound, they are assigned

the string "NO_SOURCE_PATH". This happens for any compilation which is not started through a compile call:

```
(println "Compiling" *file*) ; ❶
;; Compiling NO_SOURCE_PATH
```

❶There is no actual file compiled when this expression is evaluated.

The `*file*` and `*source-path*` dynamic variables are used extensively throughout the Clojure compiler in case of error, to

report which file unit resulted in a problem (along with other important data like the column/line numbers).

23.10 *USE-CONTEXT-CLASSLOADER*

> Warning
> As of this commit, the value of `*use-context-classloader*` is always bound to `true` regardless of the user preference. As such, `*use-context-classloader*` should be considered deprecated and not used. The initial design of this feature was discussed on the Clojure mailing list. More recent discussions are also visible on this Jira issue. The rest of this section gives a bit of background about how Clojure uses different classloaders.

Classloaders are a mechanism to achieve class isolation in Java. There are several built-in classloaders: the bootstrap, extension and application classloaders are all part of the standard Java development kit [273]. Other classloaders can be created by user applications. The Clojure runtime loads within the application classloader, the one created by the JVM to run the Clojure executable:

```
(defn get-classloader [f] ; ❶
  (.. f getClass getClassLoader))

(get-classloader #'+)
#object["jdk.internal.loader.ClassLoaders$AppCla

(get-classloader #'fn)
#object["jdk.internal.loader.ClassLoaders$AppCla
```

❶Each Java class is loaded in a classloader. To retrieve the classloader instance, we can call `getClassLoader` on a `java.lang.Class` instance.

❷We can see that the functions of the standard library have been loaded using the application classloader when the Clojure jar archive was loaded into memory (classes of the Clojure library comes in a pre-compiled state). Please note that the class type could be different depending on the version of the JDK (for example `sun.misc.Launcher$AppClassLoader` on Java 8).

To create a new classloader you always need to assign a parent classloader. The easiest and standard choice to assign the parent classloader is to use the application classloader. There are cases however in which external libraries or frameworks (one notable example is the OSGi framework [274]) would like to

impose a different choice in terms of parent classloader, for example to control multiple applications in the same JVM. One conventional way to communicate this preference is to use the thread local classloader, which is available from the `java.lang.Thread` class:

```
(.. Thread currentThread getContextClassLoader)
;; #object["clojure.lang.DynamicClassLoader@5314
```

❶ You can access and see the currently assigned thread local classloader by calling `getContextClassLoader()` on the current thread instance.

If the external library or framework communicates the classloader choice using the thread local, then Clojure is already looking into the thread local by default. When Clojure evaluates an expression, it creates and execute a new Java class that needs to be associated with a specific classloader instance. In order to enable dynamic redefinition of code, Clojure evaluates each new expression in an isolated classloader, so the definition of such expression can be easily discarded and replaced with another if needed:

```
(get-classloader (fn []))
#object["clojure.lang.DynamicClassLoader@1ac85b0

(get-classloader (fn []))
#object["clojure.lang.DynamicClassLoader@2ad3a1b
```

❶Note that the `clojure.lang.DynamicClassLoader` instance number is different every time we retrieve the classloader of a newly generated expression. This implies that the class generated to implement the expression is assigned a fresh classloader.

Each expression evaluation generates a new class which is loaded in a separate classloader instance of type `clojure.lang.DynamicClassLoader`, the Clojure class responsible to manage Clojure specific classloaders. Let's also have a look at the parent classloader:

```
(.getParent (get-classloader (fn [])))
#object["clojure.lang.DynamicClassLoader@5314245

(.getParent (get-classloader (fn [])))
#object["clojure.lang.DynamicClassLoader@5314245

(identical? ; ❷
```

```
    (.. Thread currentThread getContextClassLoader
    (.getParent (get-classloader (fn []))))
  ;; true
```

❶The parent classloader for all generated classes is the same "53142455" `DynamicClassLoader` instance.

❷The parent classloader is the same as the thread local classloader.

As we can see in the example above, newly generated expressions share the same parent classloader called "53142455". Note that is also the same as the thread local classloader. As said before, Clojure follows the rule of using the classloader available in the thread local to load new classes. The problem is if we want Clojure to ignore the thread local classloader (and use the application classloader for example). Here we show an example about how to change the thread local classloader, including showing that `*use-context-classloader*` has no effect:

```
(import '[clojure.lang DynamicClassLoader])

(def application-classloader ; ❶
  (.. #'+ getClass getClassLoader))
```

```
(def context-classloader ; ❷
  (.. Thread currentThread getContextClassLoader

(def custom-classloader (DynamicClassLoader.))

(.setContextClassLoader (Thread/currentThread) cu

(identical?
  (.. (get-classloader (fn [])) getParent)
  custom-classloader) ; ❹
;; true

(binding [*use-context-classloader* false] ; ❺
  (identical?
    (.. (class (fn [])) getClassLoader getParent
    custom-classloader))
;; true

(import '[clojure.lang DynamicClassLoader])
(def custom-classloader (DynamicClassLoader.))
(.setContextClassLoader (Thread/currentThread) cu

(identical?
  (.. (fn []) getClass getClassLoader getParent)
  custom-classloader)
;; true

(binding [*use-context-classloader* false]
```

```
    (identical?
      (.. (fn []) getClass getClassLoader getParen
      custom-classloader))
  ;; true
```

❶ `application-classloader` stores the application classloader instance, the same that was used to load all functions in the standard library.

❷ `context-classloader` saves the current thread local classloader instance, so we can change it back to the original if we need to.

❸ To change the thread local classloader, we can call `setContextClassLoader` on the current thread object. Now the thread local instance is our custom classloader.

❹ The parent classloader for freshly generated class is now our custom classloader.

❺ We can see that the attempt to prevent Clojure from using the thread local classloader fails.

Currently, there is no way to force Clojure to ignore the content of the thread local classloader, nor there is a way to tell Clojure a specific alternative to use. For this reason the `*use-context-classloader*` dynamic variable should be considered deprecated and not used.

[273] The topic of Java classloaders and their use is too broad for this book. For an introduction to classloaders, please see the following article: www.baeldung.com/java-classloaders

[274] OSGi is an initiative created to propose and maintain a standard for components life cycle isolation. The implementation makes heavy use of the classpath mechanism to load entire components (or several versions of the same component) in isolation from each other. For more information, please visit: en.wikipedia.org/wiki/OSGi.

23.11 *ALLOW-UNRESOLVED-VARS*

During Clojure compilation process, all symbols are analyzed and possibly resolved to a var instance. When the symbol cannot be resolved, Clojure throws an exception:

```
(+ 1 a) ; ❶
;; CompilerException java.lang.RuntimeException:
```

❶A familiar Clojure error: a symbol is used but not declared, producing an error.

This behavior is expected, as Clojure cannot produce valid bytecode unless all symbols are resolved. Clojure can however proceed to read or analyze the expression, without attempting

to load the generated bytecode. While access to the Clojure Reader is part of the public API with functions like read-string, access to the Analyzer is via Java only:

```
(import '[clojure.lang Compiler Compiler$C])

(Compiler/analyze Compiler$C/EVAL (read-string "
;; CompilerException java.lang.RuntimeException:
```

❶ Access to the Clojure Analyzer is only possible using Java interoperability. We can reproduce the same error we have seen previously.

The Clojure Reader does not attempt to resolve the symbol "a" (the reader can easily verify that `read-string` completes successfully). The Analyzer is responsible for making sure each symbol appearing in an expression can be resolved, either looking at the namespace registry or local bindings. In the example above, the analysis fails with the same error we've seen before. But we can request the Analyzer to accept unresolved variables using `*allow-unresolved-vars*`:

```
(binding [*allow-unresolved-vars* true] ; ❶
  (Compiler/analyze Compiler$C/EVAL (read-string
```

```
  #object["clojure.lang.Compiler$StaticMethodExpr@
```

❶By setting `*allow-unresolved-vars*` to `true` the Analyzer completes the analysis and returns an expression object we can use.

The Analyzer normally produces classes ending with "Expr" (denominating a specific type of expression). Expression classes are not meant for public consumption: they are package protected and not public, requiring some tricks to get access to them. The `clojure.lang.Compiler$StaticMethodExpr` object produced in the example represents the parsing tree resulting from analyzing the root expression (and all sub-expressions, if any). The expression object contains all the information necessary to generate bytecode and proceed to evaluate the expression. However, in this specific case, it contains a symbol "a" that cannot be resolved.

One reason to delay symbol resolution is to provide them in some other way, for example to execute Clojure code on some other back end which is not the JVM. In this scenario, missing variables could be provided as part of a different environment. We could for example reuse the standard Clojure Reader and Analyzer to emit JavaScript instead of Java bytecode. Indeed,

the very reason for the existence of `*allow-unresolved-vars*` comes from the original ClojureScript implementation [275]. The standard Analyzer has the advantage to always be up to date with the latest Clojure changes. On the other hand this approach is fragile, as the Analyzer was never completely designed to be shared across runtimes and many "Java-isms" remain in its implementation. Modern implementation of Clojure on other back ends have made use of more robust approaches like Clojure Tools Analyzer, a library that offers an interface specifically dedicated to this scenario.

[275] The curious reader can have a look at the following code that shows how the dynamic var was used originally.

23.12 *READ-EVAL* AND *SUPPRESS-READ*

The Clojure Reader can evaluate expressions directly if they use a special literal syntax `#=`, for example:

```
(read-string "(+ 1 (+ 1 1))")
;; (+ 1 (+ 1 1)) ; ❶

(read-string "(+ 1 #=(+ 1 1))")
;; (+ 1 2) ; ❷
```

❶The string represents valid Clojure code. When we read-string it results in the corresponding form as Clojure data structures, symbols and numbers.
❷After wrapping the inner expression `#=(+ 1 1)`, the reader immediately proceeds to evaluation of the enclosed expression.

We can use `*read-eval*` to prevent immediate evaluation, for example:

```
(binding [*read-eval* false] ; ❶
  (read-string "(+ 1 #=(+ 1 1))"))
;; RuntimeException EvalReader not allowed when
```

❶The use of `*read-eval*` produces exception if the input string contains the reader literal `#=`

We've encountered `*read-eval*` while introducing read. The reader is invited to check those examples for an extended explanation of `*read-eval*`.

`*suppress-read*` is effectively an implementation detail which has to do with how reader conditionals should behave in the presence of forms that are not defined for the current platform, allowing the reading process to skip the rest of the

forms immediately. Although the `*suppress-read*` is public, it is not meant for public use.

23.13 *DATA-READERS* AND *DEFAULT-DATA-READER-FN*

These dynamic variables have already been explained in detail throughout the book. The reader is invited to review the examples and explanation for `*data-readers*` in default-data-readers and `*default-data-reader-fn*` in tagged-literal.

23.14 *LOAD-TESTS* AND *STACK-TRACE-DEPTH*

These dynamic variables belong to the `clojure.test` namespace. The book already provided an example for `*stack-trace-depth*` while illustrating clojure.test, so this part is mainly going to concentrate on `*load-tests*`.

`*load-tests*` controls whether `clojure.test` should generate test functions while loading a namespace, an operation that could have performance implications. Such feature is almost never required in modern times, as build tools like Leiningen or clojure.tools.deps enforces separation of tests

from production code, preventing their presence in production code completely. To understand in which scenario `*load-tests*` could be necessary, we need to consider a style of writing unit tests that defines them along the function to be tested, for example:

```
(ns math ; ❶
  (:require [clojure.test :as t]))

(defn sum [a b] (+ a b))

(t/deftest sum-test ; ❷
  (t/is (= 2 (sum 1 1))))

(test #'sum-test) ; ❸
;; :ok
```

❶ `math` is a namespace containing mathematical functions.
❷deftest is a macro in `clojure.test` that defines a test for `sum`. The test lives in the same namespace as the target function itself.
❸We can use test to run tests for `sum`.

There are several problems with this style of writing unit tests. One of them is that the namespace becomes less readable, especially if there are many tests for each function. The other is

that test functions are generated even when the `math` namespace is deployed to production, when there is no intention to run tests. Assuming we developed tests for our application using this style, we could prevent generation of unit tests using `*load-tests*` when AOT compiling the namespace. The `math` namespace would be written to a file named "math.clj" which is available from the classpath. As we did previously discussing *compile-path*, we are going to store the file on disk and operate AOT compilation with compile. For compile to work properly, we need to run Clojure making sure it contains the current folder in the classpath:

```
;; export CLOJURE_JAR=~/.m2/repository/org/cloju
;; java -cp $CLOJURE_JAR:. clojure.main

(spit "math.clj" ; ❷
"(ns math (:require [clojure.test :as t]))
 (defn sum [a b] (+ a b))
 (t/deftest sum-test (t/is (= 2 (sum 1 1)))))")

(require '[clojure.test :as t])
(binding [*compile-path* "."
          t/*load-tests* false] ; ❸
  (compile 'math))
;; math
```

❶Build tools normally add folders like "src" or "target" to the classpath automatically. To make the example generic, we need to run the REPL making sure that the current folder "." is part of the classpath. There are many ways to achieve this, but one is to locate the Clojure library on your file system (for example as part of the Maven repository) and then run `clojure.main` with a custom classpath.

❷This is the same `math` namespace as before, saved to a file "math.clj" in the current folder.

❸We can see here how to use `*load-tests*` as we compile the math namespace.

After compiling the `math` namespace, you should see a few files created by the compilation process in the local folder, the same that Clojure would produce as part of a Jar archive that contains the application. We can now verify that there are no tests produced with the following:

```
(require '[math :as m])
(test #'m/sum-test) ; ❶
;; CompilerException java.lang.RuntimeException:
```

❶After compiling "math.clj" with `*load-tests*` bound to `false` we are unable to run tests.

The exception confirms that `deftest` did not generate tests functions.

23.15 *FEELING-LUCKY*, *LOCAL-JAVADOCS* AND *CORE-JAVA-API*

The dynamic variables in this section are all declared in the clojure.java.javadoc namespace (the reader is invited to refresh the general use and public API of this namespace in that part of the book). Even if they are not mentioned in the title, this section also shows example for `*feeling-lucky-url*` and `*remote-javadocs*`.

"Feeling lucky" is a feature that was first introduced by the Google search engine: a button on the main page allows the user to skip the results of the search and open the first hit directly. The label "feeling lucky" expresses the fact that the first result might (or might not) be what the user is searching for. In the context of the clojure.java.javadoc namespace, the `*feeling-lucky*` variable determines if, in the absence of any known Javadoc for the requested class, the `javadoc` function should try to access the "feeling lucky" functionality. But if we try this feature, we can see that is no longer working:

```
(require '[clojure.java.javadoc :as jdoc])

(jdoc/javadoc []) ; ❶

(binding [jdoc/*feeling-lucky* false] ; ❷
  (jdoc/javadoc []))
;; Could not find Javadoc for clojure.lang.Persi
```

❶The `clojure.lang.PersistentVector` class is not part of the JVM default distribution, so the call to `jdoc/javadoc` opens a rather generic Google search.

❷We can disable the feature completely by setting `*feeling-lucky*` to `false`. The effect is that now `jdoc/javadoc` returns an error.

As the reader can verify by running the example, the "feeling lucky" functionality does not work as advertised. This is because Google never designed the feature for programmatic access and the internals of how it works have changed over time. There currently no fully working solution to use the "feeling lucky" functionality. We can however set a different URL using `*feeling-lucky-url*`:

```
(binding [jdoc/*feeling-lucky-url* "https://duck
```

```
    (jdoc/javadoc []))
```

❶Showing how to use a custom "feelling lucky" URL.

The new URL points at DuckDuckGo, another popular search engine. This time the feature works as advertised and DuckDuckGo opens the GitHub page with the source for `clojure.lang.PersistentVector.java`. Other searches could return a different kind of page, depending on the settings of the search engine (it is still after all a "feeling lucky" request).

`*local-javadocs*` and `*remote-javadocs*` are two references containing a map from a package name to a local or remote package location, respectively. This is the "database" used by `clojure.java.javadoc/javadoc` to lookup for documentation pages (before attempting the "feeling lucky" option). Here's their default content:

```
(pprint @jdoc/*local-javadocs*) ; ❶
;; ()

(pprint (keys @jdoc/*remote-javadocs*)) ; ❷
;; ("java."
;;  "javax."
;;  "org.apache.commons.codec."
```

```
  ;;  "org.apache.commons.io."
  ;;  "org.apache.commons.lang."
  ;;  "org.ietf.jgss."
  ;;  "org.omg."
  ;;  "org.w3c.dom."
  ;;  "org.xml.sax.")
```

❶There is no default local URL for javadocs.
❷There are instead a few selection some common Java packages.

To alter the content of the database for the current JVM session, we can use either `add-local-javadocs` or `add-remote-javadoc` (we've seen how to add remote Javadocs in the clojure.java.javadoc section). For any local change, you can bind the variables directly:

```
  (def clojure-javadoc
    "https://javadoc.io/doc/org.clojure/clojure/la

  (binding [jdoc/*remote-javadocs*          ; ❶
            (ref {"clojure." clojure-javadoc})]
    (jdoc/javadoc []))                      ; ❷
```

❶The binding changes the content of `jdoc/*remote-javadocs` temporarily.

❷We can check the effect of adding the Clojure Javadoc calling a `jdoc/javadoc`. You should see the default browser pointing a javadoc.io/doc/org.clojure/clojure/latest/clojure/lang/Persistent

In the following example we'd like to add a new local source for the JDK 11 documentation [276]. After downloading and uncompressing the package to `~/javadocs/jdk11` in the user home (for example), we can bind `*local-javadocs*` with:

```
(def jdk11-javadoc ; ❶
  (str (System/getProperty "user.home")
       "/javadocs/jdk11/api/java.base"))

(binding [jdoc/*local-javadocs* (ref [jdk11-java
  (jdoc/javadoc (Object.)))
```

❶The `jdk11-javadoc` definition points to the user home folder where we extracted the content of the Javadoc archive.

❷Similarly to `*remote-javadocs*`, we need to wrap `*local-javadoc*` in a ref.

The last public dynamic variable, `*core-java-api*`, is for all practical purposes read-only:

```
jdoc/*core-java-api* ; ❶
;; "http://java.sun.com/javase/7/docs/api/"
```

❶The default value of `*core-java-api*`

`*core-java.api*` always returns the JDK 7 version of Javadoc (unless you run Clojure with the JDK 6). This is the version of the documentation that `clojure.java.javadoc` uses for the Java core API.

[276] You can download the Javadoc archive from www.oracle.com/java/technologies/javase-jdk11-doc-downloads.html

23.16 *FLUSH-ON-NEWLINE*

`*flush-on-newline*` is set to `true` by default. This forces a call to `flush` each time prn, println or prn-str is called. `flush` in turns calls `java.io.Writer::flush()` method, which guarantees that any characters that is still waiting in a buffer will be processed accordingly.

A visible effect of setting `*flush-on-newline*` to `false` can be seen with the following:

```
(binding [*flush-on-newline* false]
  (doseq [x (range 500)]
    (Thread/sleep 100)
    (println "Logging " x))) ; ❶
;; "Logging 0"
;; "Logging 1"
;; [...]
;; "Logging 84"
;; "Logging 85"
;; [...]
;; Logging 499
```

❶A simple loop to force side effects.

The doseq loop above prints "Logging <x>" each iteration. However, characters are buffered and only appear in burst of a hundred or so, depending on the buffer size. There are very few cases that could require turning off this functionality. To improve performance, we could turn off automatic flushing when using a non-buffered output.

23.17 *NS* AND *LOADING-VERBOSELY*

Clojure expressions always evaluate in the context of a namespace, a container abstraction that is linked to the concept of Java package and offers a few primitives to search, remove

or reload its definitions. Vars and Namespaces related functions use the `*ns*` dynamic variable to target a specific namespace, as we saw in refer for example. The reader is invited to review other examples in that chapter to see how to use `*ns*`.

`*loading-verbosely*` is not public but it is used indirectly as part of other namespace related functions, for example require or use:

```
(require '[clojure.reflect] :verbose) ; ❶

;; (clojure.core/load "/clojure/reflect") ; ❷
;; (clojure.core/load "/clojure/set")
;; (clojure.core/in-ns 'clojure.reflect)
;; (clojure.core/alias 'set 'clojure.set)
;; (clojure.core/load "/clojure/reflect/java")
;; (clojure.core/in-ns 'clojure.reflect)
;; (clojure.core/alias 'set 'clojure.set)
;; (clojure.core/in-ns 'clojure.reflect)
;; (clojure.core/alias 'str 'clojure.string)
```

❶ The option `:verbose` in require bounds `*loading-verbosely*` to `true` internally. This turns verbose logging of namespace related functions while traversing the dependency graph required to load all the required dependencies.

❷ We can see the dependency traversal necessary to load clojure.reflect.

If a standard library function does not support the `:verbose` option, we can always wrap it in a `*loading-verbosely*` binding to achieve a similar effect, even if `*loading-verbosely*` is private:

```
(binding [clojure.core/*loading-verbosely* true]
  (compile 'clojure.test))

;; (clojure.core/load "/clojure/test") ; ❷
;; (clojure.core/load "/clojure/template")
;; (clojure.core/in-ns 'clojure.template)
;; (clojure.core/alias 'walk 'clojure.walk)
;; (clojure.core/in-ns 'clojure.test)
;; (clojure.core/alias 'temp 'clojure.template)
;; (clojure.core/load "/clojure/stacktrace")
;; (clojure.core/in-ns 'clojure.test)
;; (clojure.core/alias 'stack 'clojure.stacktrac
;; (clojure.core/in-ns 'clojure.test)
```

```
  ;; (clojure.core/alias 'str 'clojure.string)
  ;; clojure.test
```

❶ `*loading-verbosely*` is private, but using it is pretty safe, so we make an exception and bypass the privacy control to enable the related binding by just prefixing it with `clojure.core`.

❷ We discover that during compilation the dependency tree of a namespace is traversed and we can see all the dependencies necessary to load clojure.test.

23.18 *MATH-CONTEXT*

`java.math.BigDecimal` is the Java implementation for a decimal number with arbitrary precision. Clojure provides a constructor for this class using the bigdec function and a configuration function with-precision that creates a `java.math.MathContext` object which is passed to all `BigDecimal` operations. with-precision offers everything that `MathContext` offers with a more idiomatic interface. If, for any reason, you're unable to use the with-precision macro, you can always use `*math-context*` directly like so:

```
  (import '[java.math MathContext RoundingMode])
```

```
  (binding [*math-context* (MathContext. 3 Roundin
    (/ 22M 7))

  ;; 3.15M
```

❶An example of using `*math-context*` to set the precision of a big-decimal division.

23.19 *WARN-ON-REFLECTION* AND *UNCHECKED-MATH*

`*warn-on-reflection*` and `*unchecked-math*` are not necessarily related dynamic variables and can be used independently. However, when the overall goal is performance optimization, they are often used together. The effect of setting `*warn-on-reflection*` to `true` is to print verbose information about the compiler need to use reflection, an expensive operation that Clojure prevents automatically whenever possible:

```
(set! *warn-on-reflection* true) ; ❶
;; true

(defn exists? [file] ; ❷
  (.exists file))
```

```
;; Reflection warning - reference to field exist

(import '[java.io File])

(defn exists? [^File file] ; ❸
  (.exists file))
```

❶ `*warn-on-reflection*` can be set! without the use of binding to activate reflection warnings globally.

❷A typical reflection warning using an operation on a `java.io.File` object. We've seen many examples of such reflection warnings throughout the book, showing how to avoid them using type hints.

❸We can use a type hint to guide the compiler to use a specific class to lookup the `.exists` method.

`*unchecked-math*` achieves a specific compiler optimization that applies to a set of bit-wise, arithmetic and array operations [277]. There are actually two optimizations performed by `*unchecked-math*`:

1. It compiles the expression directly into the corresponding JVM instruction (if available) without calling the related `clojure.lang.Numbers` operation.

2. It removes boundary checks. This is similar to replacing `+` with unchecked-add and it is a consequence of using the JVM instruction.

There is one caveat for `*unchecked-math*` to work properly, which is the arguments need to be primitive Java types. Let's look at the following `sum` function:

```
(defn sum [a b] ; ❶
  (+ a b))

(sum 1 2)
;; 3
```

❶A simple wrapper to add two numbers.

By the point "a" and "b" are passed to `+`, they are seen as `java.lang.Object` (their runtime type will be `java.lang.Long` after auto-boxing of "2" and "3"). `+` is implemented by invoking `clojure.lang.Numbers::add(Object a, Object b)` which is ultimately what receives the call. If we are not interested in checking for integer overflow, we can use unchecked-add instead:

```
(defn sum [a b] ; ❶
  (unchecked-add a b))

(sum 1 2)
;; 3
```

❶Using `unchecked-add` has the advantage of removing the boundary check for `java.lang.Long` in exchange for a risk of a silent overflow.

The same `sum` function is now written using unchecked-add producing a call to `clojure.lang.Numbers::unchecked_add(Object a, Object b)` instead. We push performance a little bit further by type hinting arguments and allowing Clojure to avoid argument auto-boxing and select a better function to perform the sum:

```
(defn sum ^long [^long a ^long b] ; ❶
  (unchecked-add a b))

(sum 1 2)
;; 3
```

❶Adding type hints remove the need for auto-boxing the two numbers "1" and "2". Note that we added a type hint for the return type as well, which is subject to the same auto-boxing fate.

The Clojure compiler now receives information about the parameter "a" and "b" that are required to be of type long. The compiler can use this information to dispatch the operation to `clojure.lang.Numbers::unchecked_add(long a, long b)` which is the most precise match available given two long numbers.

`*unchecked-math*` performs all of the above for us plus one additional optimization:

```
(set! *unchecked-math* true) ; ❶

(defn sum ^long [^long a ^long b] ; ❷
  (+ a b))

(sum 1 2)
;; 3
```

❶ We can set `*unchecked-math*` to `true` similarly to what we did with `*warn-on-reflection*`.

❷ Note that we don't need to be explicit about ignoring bound checks and we can go back using the simple `+`.

In the last example, the Clojure compiler stops using `clojure.lang.Numbers` to implement addition and use the JVM instruction `LADD` instead. This is the best solution possible as we skip an intermediate call and we use a JIT-friendly JVM instruction that HotSpot can cache aggressively. Note that we still need type hints in place for `*unchecked-math*` to work properly.

Furthermore, when `*unchecked-math*` is set to `:warn-on-boxed` all of the above are still applied, but it also prints a warning if primitive types are still boxed:

```
(set! *unchecked-math* :warn-on-boxed) ; ❶

(defn sum [^long a b] ; ❷
  (+ a b))

;; Boxed math warning, - call: ; ❸
;; public static Number clojure.lang.Numbers.uncl
```

❶ `*unchecked-math*` value is not strictly a boolean and it can also take a `:warn-on-boxed` keyword.

❷ Note that we left a single type hint but we forgot to type hint the second parameter.

❸ As expected, we receive a boxing warning because the type parameter "b" is passed as `java.lang.Object`.

[277] The full extent of what operations are supported is visible in this table of "Intrinsics": github.com/clojure/clojure/blob/master/src/jvm/clojure/lang/Intrinsics.java#L22

23.20 PRETTY PRINTING VARIABLES

We briefly introduced some capabilities of the sophisticated Clojure printing system when talking about cl-format, one of the main entry point of the `clojure.pprint` API. Perhaps not surprisingly, `clojure.pprint` is highly configurable and it makes large use of dynamic variables. Some of the variables are part of `clojure.core` while others are declared in `clojure.pprint`. The following table is a summary and a brief description:

Table 23.1. The dynamic variables related to pretty printing

Name	Namespace	Default	Descripti
`*print-readably*`	`clojure.core`	true	Print esca sequences literally.
`*print-dup*`	`clojure.core`	false	Print obje so they ca read back memory.
`*print-length*`	`clojure.core`	nil	Number o items to p in a collec
`*print-level*`	`clojure.core`	nil	Number o nesting collections level to pr
`*print-meta*`	`clojure.core`	false	Should metadata

			printed as part of the object.
`*print-pretty*`	`clojure.pprint`	true	Enable/dis pretty printing.
`*print-pprint-dispatch*`	`clojure.pprint`	fn	Set pretty printer dispatch function.
`*print-right-margin*`	`clojure.pprint`	72	Number o columns t use before breaking i a newline.
`*print-miser-width*`	`clojure.pprint`	40	The colum which to e miser styl

`*print-suppress-namespaces*`	`clojure.pprint`	nil	Remove namespac qualifier f symbols.
`*print-radix*`	`clojure.pprint`	nil	Print radi used for integers o ratio.
`*print-base*`	`clojure.pprint`	10	Change ba for printir integers o ratio.

print-readably

`*print-readably*` controls how non-printable characters such as "\n" (newline) or "\t" (tab) should appear when printed. It also controls characters used to decorate Clojure objects when printed, such as double quotes. `*print-readably*` defaults to `true` and is used internally by functions like print or println:

```
(binding [*print-readably* false] ; ❶
  (pr "a\nb\nc"))
;; a
;; b
;; cnil

(print "a\nb\nc") ; ❷
;; a
;; b
;; cnil
```

❶pr is one of the most basic printing functions. It obeys `*print-readably*` by default, printing the escaping version of control characters like newline `\n`. Here we avoid this behavior forcing `pr` not to escape them.

❷We can see the close relationship between `pr` and `print`: print is like `pr` but it binds `*print-readably*` by default.

There is no need to use `*print-readably*` directly, given the option to pick between `pr` (and related functions) or `print` (and related functions), as one set `*print-readably*` to `true` and the other to `false` respectively. It can however come handy if you don't control the code.

print-dup

`*print-dup*` is related to print-dup, a multimethod with the same name that we described in print-method. `print-dup` controls how objects should be serialized to string in such a way that they can be read back into Clojure with their value re-installed. We can use `*print-dup*` to enable this kind of printing (it defaults to `true`). Please refer to print-method and print-dup for more information and examples.

print-length and ***print-level***

`*print-length*` controls how many items of a collection to print. This can be useful when printing large and nested collections, to avoid too much output on the screen and only show the initial part of each collection, for example the first 3 items:

```
(require '[clojure.xml :as xml])

(def document
  (xml/parse
    "http://feeds.bbci.co.uk/news/rss.xml"))

(binding [*print-length* 3] ; ❶
  (pprint document))

;; {:tag :rss,
```

```
  ;;  :attrs
  ;;  {:xmlns:media "http://search.yahoo.com/mrss/
  ;;   :version "2.0",
  ;;   :xmlns:atom "http://www.w3.org/2005/Atom",
  ;;   ...},
  ;;  :content
  ;;  [{:tag :channel,
  ;;    :attrs nil,
  ;;    :content
  ;;    [{:tag :title, :attrs nil, :content ["BBC
  ;;     {:tag :description, :attrs nil, :content
  ;;     {:tag :link, :attrs nil, :content ["https
  ;;     ...]}]}
```

❶Printing only the first 3 items of a large deeply nested collection representing an XML news feed.

As you can see, after setting `*print-length*` to `true` collections are truncated and a visible `…` triple dot replaces the rest of the collection. Even if it's not possible to see the full data, printing just a few items helps understanding the general shape of the data.

We can also control how many levels of nesting to print with `*print-level*`:

```
  (binding [*print-length* 3
            *print-level* 4] ; ❶
    (pprint document))

  ;; {:tag :rss,
  ;;  :attrs
  ;;  {:xmlns:media "http://search.yahoo.com/mrss/
  ;;   :version "2.0",
  ;;   :xmlns:atom "http://www.w3.org/2005/Atom",
  ;;   ...},
  ;;  :content [{#, #, #}]}
```

❶The combined effect of `*print-length*` and `*print-level*`.

In the example above, we can see truncated items in a collection replaced by `…` and truncated nested branches replaced by `#`.

print-meta

`*print-meta*` enable metadata printing for serialization-oriented printing functions. Clojure has to major sets of printing functions: those starting with just pr (pr, prn, pr-str, prn-str) print objects so they can be read back with read-string.

`*print-meta*` also enables additional printing of metadata (and their deserialization back into Clojure):

```
(def s ; ❶
  (binding [*print-meta* true]
    (pr-str (with-meta [1 2 3] {:my :meta}))))

(println s) ; ❷
;; ^{:my :meta} [1 2 3]

(def v (read-string s))

(println v) ; ❸
;; [1 2 3]

(meta v) ; ❹
;; {:my :meta}
```

❶After binding `*print-meta*`, we produce a string version of the vector with included metadata.

❷As we can see, the string also contains the matadata map.

❸After reading the string back into Clojure data structures, the vector prints as expected.

❹Metadata have also been preserved as expected.

print-pretty

`*print-pretty*` is the main pretty printer "on/off switch". It is `true` by default, but we can turn it off locally (using binding) or globally using alter-var-root:

```
(require '[clojure.pprint :as pprint])

(pprint/write (range 100)) ; ❶
;; (0
;;  1
;;  2
;;  ...
;;  99)nil

(binding [pprint/*print-pretty* false] ; ❷
  (pprint/write (range 100)))
;; (0 1 2 ... 99)nil

(alter-var-root #'pprint/*print-pretty* (constan
(pprint/write (range 100))
;; (0 1 2 ... 99)nil

(alter-var-root #'pprint/*print-pretty* (constan
```

❶The default behavior of `pprint/write` is to arrange the sequence of numbers as a single column.
❷We can turn off any pretty printing behavior by setting `*print-pretty*` to `false`.
❸alter-var-root changes the default value of `*print-pretty*` switching off pretty printing from that point onward.
❹Remember to turn pretty printing back on.

print-pprint-dispatch

`clojure.pprint` works by defining different printing strategies based on the type of the object to print. By default, `clojure.pprint` uses a default multimethod called `simple-dispatch`:

```
(require '[clojure.pprint :as pprint])

(pprint/pprint (keys (methods pprint/simple-disp
;; (nil ;
;;  clojure.lang.PersistentQueue
;;  clojure.lang.ISeq
;;  :default
;;  clojure.lang.IPersistentVector
;;  clojure.lang.IPersistentMap
;;  clojure.lang.IDeref
```

```
  ;;  clojure.lang.IPersistentSet
  ;;  clojure.lang.Var)
```

❶methods returns the dispatch table of the `pprint/simple-dispatch` multimethod.

The dispatch table of the `pprint/simple-dispatch` multimethod shows specific pretty printing installed for the most common Clojure types. The main goal of the `pprint/simple-dispatch` implementation is to display Clojure data structures on screen in a human readable way. `clojure.pprint` also contains a different formatting strategy dedicated to Clojure source code. Clojure sources are still data structures, but with different requirements in terms of formatting. We can switch between different formatting strategies by selecting a different multimethod dispatch:

```
(require '[clojure.pprint :as pprint])

(def condp-statement
  '(condp = x
      1 "one" 2 "two" 3 "three"
      4 "four" 5 "five" 6 "six" :else :none))

(pprint/pprint condp-statement) ; ❶
```

```
;; (condp
;;  =
;;  x
;;  1
;;  "one"
;;  2
;;  "two"
;;  3
;;  "three"
;;  4
;;  "four"
;;  5
;;  "five"
;;  6
;;  "six"
;;  :else
;;  :none)

(binding [pprint/*print-pprint-dispatch* pprint/
  (pprint/pprint condp-statement))
;; (condp = x
;;   1 "one"
;;   2 "two"
;;   3 "three"
;;   4 "four"
;;   5 "five"
```

```
  ;;   6 "six"
  ;;   :else :none)
```

❶The first attempt at pretty printing `condp-statement` is not very readable, with `clojure.pprint` using the default strategy for sequential lists, without any knowledge of the specific semantic of a condp statement.

❷By assigning `*print-pprint-dispatch*` to `pprint/code-dispatch` (a pretty printer specifically dedicated to code sources) we can see a much better rendering.

`clojure.pprint` also provides with-pprint-dispatch, a macro that achieves the same effect as binding `*print-pprint-dispatch*` to a different dispatch multimethod.

print-right-margin

`*print-right-margin*` sets a desirable number of columns to use before truncating the current line and using a new one:

```
(require '[clojure.pprint :as pprint])

pprint/*print-right-margin* ; ❶
;; 72

(defn words [n] (take n (repeat 'clojure))) ; ❷
```

```
(defn print-with-length [s] ; ❸
  (pprint/pprint s)
  (count (with-out-str (pprint/pprint s))))

(print-with-length (words 8)) ; ❹
;; (clojure clojure clojure clojure clojure cloj
;; 66
```

❶The default value of `*print-right-margin*` is 72.

❷A function that creates a sequence of "n" words.

❸ `print-with-length` pretty prints the given input and also shows how many columns it requires when printed.

❹We can see that a sequence of 8 "Clojure" words takes 66 columns to print.

The example above counts the number of columns necessary to print a sequence of words. `clojure.pprint` proceeds to add spaces, parenthesis and any other formatting artifacts that improve visualization. Let's change the right margin to force pretty print to split into multiple lines:

```
(binding [pprint/*print-right-margin* 60] ; ❶
  (print-with-length (words 8)))
;; (clojure
;;  clojure
```

```
;;  clojure
;;  clojure
;;  clojure
;;  clojure
;;  clojure
;;  clojure)
```

❶Setting `*print-right-margin*` to 60.

After setting the right margin to 60, all words are printed on a new line.

print-miser-width

"Miser mode" is a style of indenting that helps in the case of formatting source code, which tends to accumulate toward the right margin when it is heavily nested. To see miser mode in action, we are going to force the situation by setting a small right margin:

```
(require '[clojure.pprint :as pprint])

pprint/*print-miser-width* ; ❶
;; 40

(def nested-statement ; ❷
```

```
  '(condp = x
   1 "one" 2 "two" 3 "three"
   4 "four" 5 "five" 6 "six"
   :else (condp = x
   1 "one" 2 "two" 3 "three"
   4 "four" 5 "five" 6 "six"
   :else :none)))

(binding [pprint/*print-pprint-dispatch* pprint/
          pprint/*print-right-margin* 30]
    (pprint/pprint nested-statement))

;; (condp ; ❹
;;   =
;;   x
;;   1 "one"
;;   2 "two"
;;   3 "three"
;;   4 "four"
;;   5 "five"
;;   6 "six"
;;   :else
;;   (condp
;;     =
;;     x
;;     1 "one"
;;     2 "two"
;;     3 "three"
```

```
  ;;     4 "four"
  ;;     5 "five"
  ;;     6 "six"
  ;;     :else :none))
```

❶The default miser width is 40.

❷This nested condp statement has been lightly formatted, but in reality it could appear on a single line.

❸We enable `pprint/code-dispatch` because the effect of miser mode are readily apparent. Note that we also set a shorter than the default right margin at 30 columns.

❹The result appears quite fragmented.

The `condp` statement appears to be correctly constrained by our forcibly narrow right margin creating a formatting that is still readable but a bit too heavy on new lines. If we now assume this formatting to appear on a more heavily nested code, there would be a lot of empty space to the left, because indentation has forced this code to accumulate to the right. `*print-miser-width*` is the number of columns to wait before pretty printing enters miser mode. In miser mode the pretty printer is allowed a more liberal use of new lines to compensate for this situation:

```
(binding [pprint/*print-pprint-dispatch* pprint/
          pprint/*print-right-margin* 30
          pprint/*print-miser-width* 20] ; ❶
     (pprint/pprint nested-statement))

;; (condp = x ; ❷
;;   1 "one"
;;   2 "two"
;;   3 "three"
;;   4 "four"
;;   5 "five"
;;   6 "six"
;;   :else (condp = x
;;           1 "one"
;;           2 "two"
;;           3 "three"
;;           4 "four"
;;           5 "five"
;;           6 "six"
;;           :else :none))
```

❶We set the `*print-miser-width*` to 20.

❷The code appears now less constrained by the narrow right margin, making better use of the available space.

print-suppress-namespaces

`*print-suppress-namespaces*` controls if symbols should print without namespace qualification (in case symbols are fully qualified) and it defaults to `false` (namespaces should appear):

```
(require '[clojure.pprint :as pprint])

(pprint '{num/val 1 num/name "one"}) ; ❶
;; {num/val 1, num/name "one"}

(binding [pprint/*print-suppress-namespaces* tru
  (pprint '{num/val 1 num/name "one"}))
;; {val 1, name "one"}

(binding [pprint/*print-suppress-namespaces* tru
  (pprint {:num/val 1 :num/name "one"}))
;; {:num/val 1, :num/name "one"}
```

❶When symbols are namespace qualified, `pprint` shows namespaces by default.

❷We can suppress namespaces by setting `*print-suppress-namespaces*` to `true`.

❸ `*print-suppress-namespaces*` has not effect on keywords, another Clojure type that supports namespace qualified names.

`*print-suppress-namespaces*` is especially useful to print macroexpanded forms. Symbols are automatically namespaced by syntax-quoting, which is an essential tool in hygienic macros. However, we don't necessarily want to see automatic qualification when printing:

```
(binding [pprint/*print-pprint-dispatch* pprint/
  (pprint (macroexpand-1 '(definline myfn [])))

;; (do
;;   (clojure.core/defn myfn [] nil)
;;   (clojure.core/alter-meta!
;;     #'myfn
;;     clojure.core/assoc
;;     :inline
;;     (clojure.core/fn myfn [] nil))
;;   #'myfn)

(binding [pprint/*print-pprint-dispatch* pprint/
          pprint/*print-suppress-namespaces* tru
  (pprint (macroexpand-1 '(definline myfn [])))

;; (do
;;   (defn myfn [] nil)
;;   (alter-meta! #'myfn assoc :inline (fn myfn
;;   #'myfn)
```

❶ `pprint/code-dispatch` was activated to print the macroexpansion of a simple form. `pprint` prints fully qualified namespaces, as they appear after the macro expansion.
❷ After setting `*print-suppress-namespaces*` to `true`, `clojure.core` prefixes are removed.

print-base and ***print-radix***

`*print-base*` allows printing of numbers in a different base (by default, numbers print in base 10):

```
(require '[clojure.pprint :as pprint])

(binding [pprint/*print-base* 2] ; ❶
  (pprint/pprint 500))
;; 111110100

(binding [pprint/*print-base* 16] ; ❷
  (pprint/pprint 3405691582))
;; cafebabe
```

❶ `*print-base*` tells the pretty printer to use base 2 for numbers.
❷ "CAFEBABE" is the magic number in the Java class file structure.

As demonstrated by converting 3405691582 in base 16, some numbers are difficult to tell apart from generic strings when printed in a different base. We can use `*print-radix*` to introduce a base specifiers:

```
(binding [pprint/*print-base* 16
          pprint/*print-radix* true] ; ❶
  (pprint/pprint 3405691582))
;; #xcafebabe
```

❶When `*print-radix*` is set to `true`, the pretty printer adds a base qualifier in front of numbers converted to a different base.

`*print-radix*` only prints the radix specifiers `#b`, `#o`, `#x` for base 2 (binary), 8 (octal) and 16 (hexadecimal) respectively.

23.21 OTHER DYNAMIC VARIABLES

This section collects other dynamic variables that are public and accessible, but are not meant for public consumption and should be considered implementation details of the respective namespaces.

- `*current*`, `*sb*`, `*state*` and `*stack*`. These dynamic variables are all part of the clojure.xml namespace.
- `*depth*`, `*var-context*`, `*report-counters*`, `*initial-report-counters*`, `*testing-vars*`, `*testing-contexts*` and `*test-out*` are part of clojure.test.
- `*fn-loader*` is an internal mechanism used by the Clojure compiler to communicate which parent classloader should be used to compile functions and expressions in a namespace. Please consult *use-context-classloader* for an overview of the Clojure loading mechanism. Also see this article about Clojure compilation that mentions how `*fn-loader*` is used in the compiler implementation.

23.22 SUMMARY

Dynamic variables are both a concurrency mechanism (they can stack up values by thread) as well as a communication mechanism that uses the thread-local memory. We can use dynamic variables to carry information from one part of the program to another. Clojure uses dynamic variables pervasively to configure how specific parts of the system behave and the goal of this chapter is to describe what is available to the end

user. Dynamic variables are also handy to configure programs we don’t own, by surrounding expressions with the related binding form. Dynamic variables belong to the public interface of the standard library similarly to functions or macros.

APPENDIX A: ALPHABETIC INDEX

"." , ".." and doto: 21.1

*1, *2, *3 and *e: 23.1

agent: 23.3

allow-unresolved-vars: 23.11

assert: 23.4

clojure-version and *command-line-args*: 23.5

compile-files: 23.6

compile-path: 23.7

compiler-options: 23.8

data-readers and *default-data-reader-fn*: 23.13

feeling-lucky, *local-javadocs* and *core-java-api*: 23.15

file and *source-path*: 23.9

flush-on-newline: 23.16

in, *out* and *err*: 23.2

load-tests and *stack-trace-depth*: 23.14

math-context: 23.18

ns and *loading-verbosely*: 23.17

read-eval and *suppress-read*: 23.12

use-context-classloader: 23.10

warn-on-reflection and *unchecked-math*: 23.19

+', -', *', inc' and dec': 5.8

+, -, * and /: 5.1

->: 2.3.1

->>: 2.3.2

< , > , <= , >=: 6.3

= (equal) and not= (not equal): 6.1

== (double equal): 6.2

add-watch and remove-watch: 14.9

agent: 14.6

aget, aset, alength and aclone: 21.7.4

alias, ns-aliases and ns-unalias: 16.7

alter-var-root and with-redefs: 16.3

amap and areduce: 21.7.5

and, or: 3.2.2

apply: 2.4.1

apropos: apropos

are: are

array-map: 11.1.2

as->: 2.3.5

as-relative-path: as-relative-path

as-url: as-url

aset-int and other types setters: 21.7.6

assoc, assoc-in and dissoc: 11.3.1

atom: 14.5

bean: 21.5

binding: 16.4

bit-and and bit-or: 3.2.3

blank?, ends-with?, starts-with?, includes?: 19.9

browse: browse

case: 3.3.4

cat: 7.2.4

chunk-cons, chunk-first, chunk-rest, chunk-next, chunk-buffer, chunk-append and chunk: 10.12

clojure.core.server: 22.7

clojure.data/diff: 6.7

clojure.edn/read and clojure.edn/read-string: 17.6

clojure.inspector: 22.2

clojure.java.browse: 22.5

clojure.java.io: 22.8

clojure.java.javadoc: 22.10

clojure.java.shell: 22.6

clojure.main: 22.4

clojure.repl: 22.3

clojure.set/map-invert: 11.4.3

clojure.set/rename-keys: 11.4.2

clojure.template/apply-template: 4.7

clojure.template/do-template: 4.8

clojure.test: 22.9

clojure.walk/keywordize-keys and clojure.walk/stringify-keys: 11.4.1

clojure.xml: 22.1

clojure.zip: 8.4.4

comp: 2.2.2

compare: 6.4

compile: 17.4

complement: 2.2.3

completing: 7.2.3

concat and lazy-cat: 9.3.9

cond: 3.3.2

cond-> and cond->>: 2.3.3

condp: 3.3.3

conj: 8.2.1

cons and list*: 9.4.2

constantly: 2.2.4

contains?: 8.2.3

copy: copy

count: 8.1.2

create-struct, defstruct, struct-map, struct and accessor: 11.1.4

def, declare, intern and defonce: 16.1

default-data-readers: 17.8

definline: 4.5

defmacro: 4.1

defmulti and defmethod: 15.11

defn and defn-: 2.1.1

defprotocol: 15.8

defrecord: 15.7

deftest: deftest

deftype and definterface: 15.4

delay: 14.3

delete-file: delete-file

demunge: demunge

deref and realized?: 14.7

derive and make-hierarchy: 15.10

destructure: 4.6

dir: dir

dir-fn: dir-fn

disj: 13.4

distinct, dedupe and distinct?: 10.8

doc: doc

doseq, dorun, run!, doall, do: 20.2

dotimes: 3.4.5

drop, drop-while, drop-last, take, take-while, take-last, nthrest, nthnext: 10.2

eduction: 7.2.2

empty: 8.1.4

empty? and not-empty: 8.1.6

escape, char-name-string, char-escape-string: 19.6

eval: 17.2

every-pred and some-fn: 2.2.9

every?, not-every?, some and not-any?: 8.1.5

ex-info and ex-data: 21.4

extend, extend-type and extend-protocol: 15.9

file: file

file-seq: 9.3.3

filter and remove: 3.5.3

filterv: 12.6

find, key and val: 11.2.2

find-doc: with-out-str

find-ns and all-ns: 16.11

first, second and last: 3.5.1

flatten: 10.7

fn: 2.1.2

fn*: 2.1.3

fnil: 2.2.1

fold: 7.1.1

foldcat, cat and append!: 7.1.4

for: 3.4.3

format, printf and cl-format: 18.1

frequencies: 8.3.4

future: 14.1

gen-class and gen-interface: 15.3

gensym: 4.4

get: 8.2.2

group-by: 8.3.6

hash: 6.6

hash-map: 11.1.1

hash-set: 13.1

identical?: 6.5

identity: 2.2.5

if, if-not, when and when-not: 3.3.1

if-let, when-let, if-some and when-some: 3.1.2

image-javadoc-string: image-javadoc-string

inc and dec: 5.2

index-of, last-index-of: 19.8

inspector: inspector

interpose and interleave: 10.5

into: 8.1.1

ints and other typed array casting: 21.7.7

is: is

iterate: 9.2.2

iterator-seq and enumeration-seq: 9.3.8

javadoc: javadoc

join: 19.2

juxt: 2.2.6

keep and keep-indexed: 10.3

keys and vals: 11.2.1

lazy-seq: 9.3.1

let and let*: 3.1.1

letfn and letfn*: 3.1.3

line-seq: 9.3.6

list: 9.4.1

load, load-file, load-reader and load-string: 17.3

load-script: load-script

locking, monitor-enter and monitor-exit: 14.10

loop, recur and loop*: 3.4.1

lower-case, upper-case, capitalize: 19.7

macroexpand, macroexpand-1 and macroexpand-all: 4.2

main: main

make-array: 21.7.1

make-parents: make-parents

map and map-indexed: 3.5.2

mapcat: 10.4

mapv: 12.5

max and min: 5.4

max-key and min-key: 5.5

memfn: 2.2.7

memoize: 2.4.2

merge and merge-with: 11.3.3

meta, with-meta, vary-meta, alter-meta! and reset-meta!: 16.13

monoid: 7.1.3

munge: munge

new: 21.2

not: 3.2.1

ns, in-ns, create-ns and remove-ns: 16.6

ns-map and ns-unmap: 16.8

ns-publics, ns-interns, ns-imports: 16.9

nth: 8.1.3

object-array and other typed initializers: 21.7.2

partial: 2.2.8

partition, partition-all and partition-by: 10.6

peek and pop: 12.3

pmap, pcalls and pvalues: 9.1.5

pprint, pp, write and print-table: 18.3

pr, prn, pr-str, prn-str, print, println, print-str, println-str: 18.2

prewalk and postwalk: 8.4.2

prewalk-replace and postwalk-replace: 8.4.3

print-method, print-dup and print-ctor: 18.4

promise and deliver: 14.2

proxy: 15.5

pst: pst

quot, rem and mod: 5.3

quote: 4.3

rand and rand-int: 5.6

rand-nth: 8.3.1

random-sample: 8.3.3

range: 3.4.2

re-pattern, re-matcher, re-groups, re-seq, re-matches, re-find: 19.10

re-seq: 9.3.5

read and read-string: 17.1

reader: reader

reader-conditional and reader-conditional?: 17.9

reduce and reductions: 3.5.4

reduce-kv: 11.3.4

reduced, reduced?, ensure-reduced, unreduced: 7.3

reducer and folder: 7.1.2

ref: 14.4

refer, refer-clojure, require, loaded-libs, use, import: 16.10

reflect and type-reflect: 21.6

reify: 15.6

repeat and cycle: 9.2.3

repeatedly: 9.2.1

repl: repl

repl: repl

replace: 8.3.7

replace, replace-first, re-quote-replacement: 19.3

resource: resource

rest, next, fnext, nnext, ffirst, nfirst and butlast: 10.1

resultset-seq: 9.3.7

reverse: 8.3.8

root-cause: root-cause

rseq: 9.1.2

run-all-tests: run-all-tests

run-tests: run-tests

select, index, rename, join and project: 13.7

select-keys and get-in: 11.2.3

seq and sequence: 9.1.1

seque: 9.1.4

set!: 20.4

set: 13.2

set-validator! and get-validator: 14.8

sh: sh

shuffle: 8.3.2

slurp and spit: 18.5

socket-server: socket-server

some-> and some->>: 2.3.4

sort and sort-by: 8.3.5

sorted-map and sorted-map-by: 11.1.3

sorted-set and sorted-set-by: 13.3

source: source

source-fn: source-fn

split-at and split-with: 10.10

stack-element-str: stack-element-str

stop-server: stop-server

str: 19.1

subs, split and split-lines: 19.4

subseq and rsubseq: 9.1.3

subset? and superset?: 13.6

subvec: 12.7

symbol and keyword: 15.1

table-inspector: table-inspector

tagged-literal and tagged-literal?: 17.7

take-nth: 10.9

test and assert: 17.5

test-all-vars: test-all-vars

test-ns: test-ns

test-var: test-var

testing: testing

the-ns, ns-name and namespace: 16.12

to-array, into-array, to-array-2d: 21.7.3

trampoline: 2.4.3

transduce: 7.2.1

transient, persistent!, conj!, pop!, assoc!, dissoc! and disj!: 20.1

tree-inspector: tree-inspector

tree-seq: 9.3.2

trim, triml, trimr, trim-newline: 19.5

try, catch, finally and throw: 21.3

unchecked-add and other unchecked operators: 5.9

unchecked-add-int and other unckecked-int operators: 5.10

union, difference and intersection: 13.5

update and update-in: 11.3.2

use-fixtures: use-fixtures

var, find-var and resolve: 16.2

vec: 12.2

vector: 12.1

vector-of: 12.4

volatile!, vreset!, vswap! and volatile?: 20.3

walk, prewalk-demo and postwalk-demo: 8.4.1

when-first: 10.11

while: 3.4.4

with-local-vars, var-get and var-set: 16.5

with-open: with-open

with-precision: 5.7

with-test: with-test

writer: writer

xml: xml

xml-seq: 9.3.4

zipmap: 11.1.5

APPENDIX B: GLOSSARY AND RECURRING TERMINOLOGY

This section contains a glossary of words and terminology that has been used throughout the book. Many of them are Clojure specific (they have been commonly accepted overtime as definition of certain concepts or part of the language). Other are more general and used outside the Clojure community, but worth repeating here for better comprehension.

B.1 FUNCTION GRAMMAR SPECIFICATION

The contract of some of the functions in the standard library is quite elaborated and in the spirit of maintaining this book practical, contract sections don't use a rigorous grammar. The book uses an informal grammar instead, which is hopefully more expressive than the corresponding function documentation, but does not aim to cover the full spectrum of all possible cases. The reader should use the grammar to get a general sense of what the function does, but then refer to the examples to cover the remaining cases. Here's a glossary of the syntax used in this book:

- "<term>": terms in angle brackets are terminal and they don’t result in any further expansion.
- "[]": terms in square brackets are actual vectors in the function definition.
- "()": terms in round brackets are actual lists in the function definition.
- "OR": expresses the option of choosing between one the possible choices.
- ":=>": shows a term that can be further expanded into other terms.
- "..": means "many other repetitions of" the expressions surrounding it.
- "?": after a term means that the term is optional.

B.2 TRUTHY, FALSEY AND LOGICAL BOOLEANS

Clojure extends the meaning of `true` and `false` to other types and constant literals, so a value is always interpretable as a boolean even if its type is not technically a boolean. A "logical boolean" is a pseudo-type so that:

- `false` and `nil` are considered `false`.
- anything else is `true`.

From this definition it follows that:

```
(if [] :true :false) ; ❶
;; :true

(if "false" :true :false) ; ❷
;; :true

(if 0 :true :false) ; ❸
;; :true
```

❶Empty collections are `true` values.
❷Anything resembling a boolean that is not actually a boolean (the string "true" or the keyword :false) are `true` values.
❸Numbers are always `true`.

B.3 PREDICATE

In mathematical logic, a predicate is a function of one argument returning either `true` or `false` [278]. In programming the definition is the same with the option to allow the function to return any type (not just boolean) provided the language has a convention to categorize all possible values as `true` or `false`.

Clojure contains a definition of logical boolean value that allows a predicate to return any type. Predicate functions are

frequently used throughout the standard library, for example with filter.

[278] The Wikipedia article about predicates in mathematical logic is en.wikipedia.org/wiki/Predicate_(mathematical_logic)

B.4 ARITY, ARITIES, VARARGS

An "arity" is the name given to one of the possible "bodies" of a function. Clojure functions allow for multiple independent implementation to be executed based on the number of arguments the function is called. A special parameter declaration called "vararg" (indicated by the presence of the ampersand `&`) allows the presence of a body to catch all other function cardinalities that are not otherwise declared.

A function supports "multiple-arities" when it contains more than one body. The function is "single-arity" when it contains only a single body. Here's for example the `doc` string for vector, a function with eight arities:

```
user=> (doc vector)
//-------------------------
clojure.core/vector ; ❶
([] [a] [a b] [a b c] [a b c d] [a b c d e] [a b
  Creates a new vector containing the args.
```

❶We can use doc to see the documentation string attached to the metadata of the var "vector".

There are two main reasons to design a function with multiple arities:

- Implement slightly different behaviors based on the number of parameters.
- Improve performance by producing statically compiled entry points to each of the different arities.

The case for vector above is a matter of performance: many calls to create vectors happen with just a few items. These calls have been optimized at compile time and there is no runtime dispatch to select the correct behavior.

B.5 SEQABLE

A "Seqable" is a Clojure value whose type supports the `clojure.lang.Seqable` interface so that `(instance? clojure.lang.Seqable value)` is `true`. Pretty much all

Clojure data containers are "Seqable" with just a few exceptions (transients, arrays and structs).

B.6 BINDINGS

A "binding" is the relationship between a symbol and its associated value. The binding can be:

- "local" when the binding encompasses a single form. let is the typical way to create local bindings. Also defn arguments are bindings between the argument name and the value after the function is called.
- "global" when the binding retains its value beyond the enclosing form. A global binding is usually defined with def and is visible across the current namespace by default. It's also available from other namespaces after requiring it. A global binding can be changed to point to another value at any time with alter-var-root.

- "dynamic" when the binding value can be changed (like the global binding) but only in the context of the calling thread. If a dynamic binding changes, it will only be visible from functions which are called in the context of the same thread which made the changes. Other threads executing the same function won't be able to see the changes. It requires the use of the binding macro.

B.7 BENCHMARKING

Benchmarking is the general activity of comparing metrics related to code execution. The most common benchmark is probably comparing speed between snippets of code producing roughly the same result to determine which solution is the most optimized. Other kind of benchmarks are about memory consumption, ephemeral objects generation (garbage collection pressure) or other stack related (how much stack space is used).

We used the Criterium library (github.com/hugoduncan/criterium) throughout the book to prepare the Java environment during a benchmark. The library, apart from performing warming up cycles, also repeats the code under benchmark many times collecting statistics that are then printed on screen.

In general, `criterium.core/quick-bench` is sufficient to quickly evaluate the relative speed of two or more forms. If the two forms produce roughly equivalent results, it's worth using `criterium.core/bench` to see more accurate results.

When necessary, is also useful to restart the JVM between benchmarks to make sure that there are no dependencies between tests.

B.8 JAVA INTEROPERABILITY

The term "interop" is a shortened form for interoperability. In Clojure this term is used to describe the part of the Clojure syntax that allows calling Java method and classes directly. The term "Java interop" describe the situation in which Clojure code calls into Java methods or constructors which includes the following macros, operators and syntax:

- "." The dot-macro enables most of the Java calls. Based on the position (before a space, immediately before or after a symbol, with an optional "-" dash) it determines the semantic of the call (method call, attribute access or constructor call).
- "/" used between the name of a class and a method name invokes the related static method on that class.

- "\^" type hints are also part of the Java interop capabilities.

B.9 BOXING AND AUTO-BOXING

"Boxing", in the context of Java, refers to the option of wrapping primitive types into their corresponding reference types: for example, `int` with `java.lang.Integer`, `bool` with `java.lang.Boolean` and so on. The Java compilers can do this automatically in certain cases (in this case is called auto-boxing).

Auto-boxing maybe necessary when a primitive type is assigned to a generic containers such as collection of objects or a function accepting generic object as parameters (the vast majority of bytecode generated Clojure functions). Boxing has a cost that should be considered in performance related scenarios. Clojure allows programmers to prevent boxing of primitive through careful use of type hints and other means.

APPENDIX C: BIBLIOGRAPHY

[Stoyan] Herbert Stoyan, Early LISP history (1956 - 1959), LFP '84 Proceedings of the 1984 ACM Symposium on LISP and functional programming, Pages 299-310, available from the ACM Digital Library at dl.acm.org/citation.cfm?id=802047

[clojure-style-guide] Bozhidar Batsov. The Clojure Style Guide. github.com/bbatsov/clojure-style-guide

[oleg-polyvariadic] Oleg Kiselyov. Polyvariadic functions and keyword arguments: pattern-matching on the type of the context. okmij.org/ftp/Haskell/polyvariadic.html

[compiler-macros] Arthur Lemmens. Compiler macros implementation from Lisp. www.pentaside.org/paper/compilermacro-lemmens/compiler-macros-for-publication.txt

[random-fail-plane] George Marsaglia. Random Numbers Fall Mainly In The Planes. www.ics.uci.edu/~fowlkes/class/cs177/marsaglia.pdf

[mccarthy-symbolic] John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part

I. www.brinckerhoff.org/clements/csc530-sp09/Readings/mccarthy-1960.pdf

[mccarthy-lispmanual] John McCarthy, Lisp I Programmer's Manual. Available at bitsavers.informatik.uni-stuttgart.de/pdf/mit/rle_lisp/LISP_I_Programmers_Manual_Mar60.pdf and many other links.