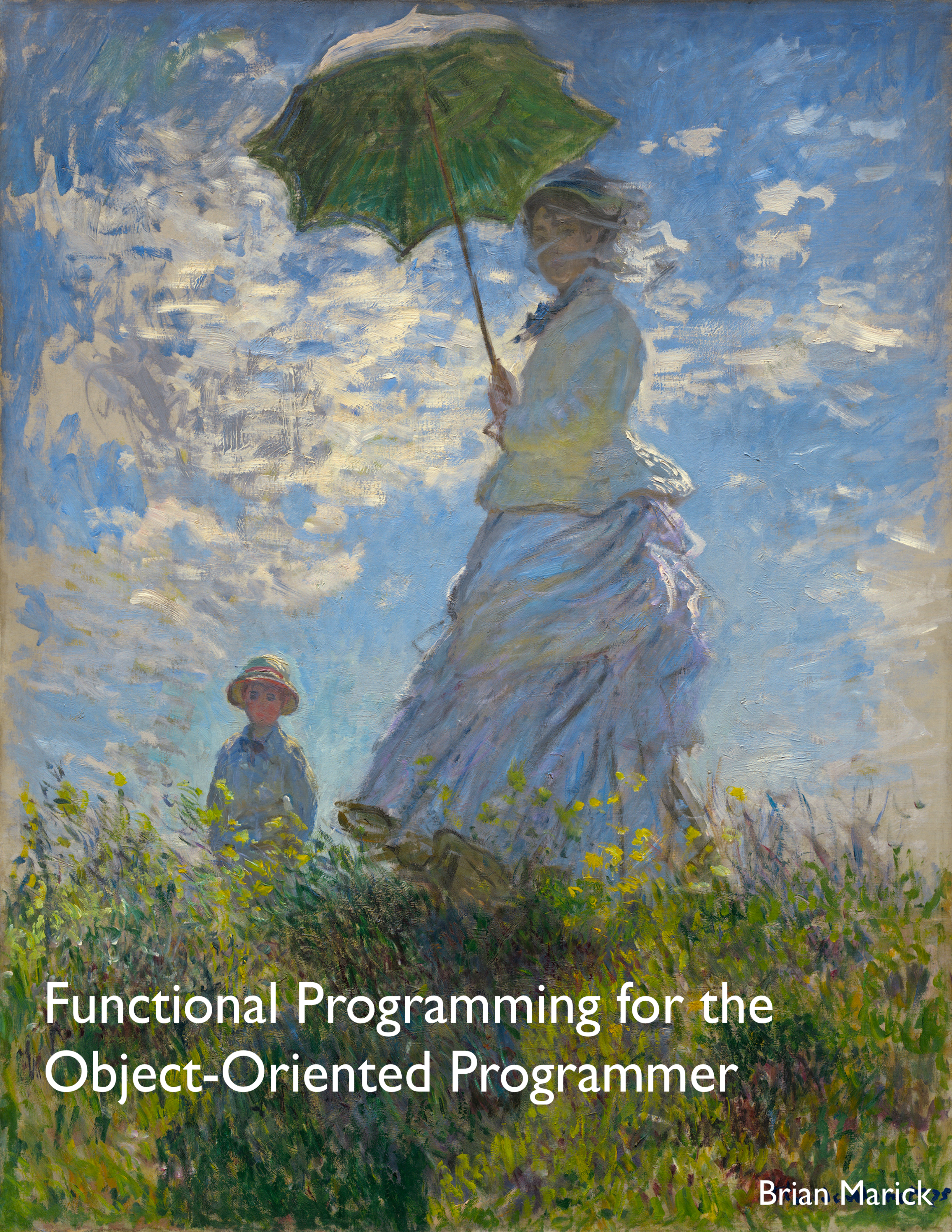


# Functional Programming for the Object-Oriented Programmer

Brian Marick

# Functional Programming for the Object-Oriented Programmer

Brian Marick

This book is for sale at http://leanpub.com/fp-oo

This version was published on 2015-04-30

## Tweet This Book!

Please help Brian Marick by spreading the word about this book on Twitter!

The suggested hashtag for this book is #fp_oo.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#fp_oo

*To my father, who taught me to care about the work.*

# Contents

# Introduction

Many, many of the legendary programmers know many programming languages. What they know from one language helps them write better code in another one. But it's not really the language that matters: adding knowledge of C# to your knowledge of Java doesn't make you much better. Those languages are too similar: they encourage you to look at problems in pretty much the same way. You need to know languages that conceptualize both problems and solutions in substantially different ways.

Once upon a time, object-oriented programming was a radical departure from what most programmers knew. So learning it was both hard and mind-expanding. Nowadays, the OO style (or some approximation to it) is the dominant one, so ambitious people need to seek out different styles.

The functional programming style is nicely different from the OO style, but there are many interesting points of comparison between them. This book aims to teach you key elements of the functional style, helping you take them back to your OO programming.

There's a bit more, though: although the functional style has been around for many years, it's recently become trendy, partly because language implementations keep improving, and partly because functional languages are better suited for the problem of running one program on multiple cores. Some trends with a lot of excitement behind them wither, but others (like object-oriented programming) succeed immensely. If the functional style becomes commonplace, this book will position you to be around the leading edge of that wave.

There are many functional languages. There are arguments for learning the purest of them (Haskell, probably). But it's also worthwhile to learn a slightly-less-pure language if there are more jobs available for it or more opportunities to fold it into your existing projects. According that standard, Clojure and Scala–both of which piggyback on the Java runtime–stand out. This book will use Clojure.

## Prerequisites

You need to know at least one object-oriented programming language. Anything will do: Objective-C, C++, C#, Java, Ruby, Python: you name it.

You need to be able to start a program from the command line.

## The flow of the book

I'll start by teaching you just enough Clojure to be dangerous. Then, in Part 1 of the book, we'll use that knowledge to embed an object-oriented language within Clojure in a more-or-less conventional way. That'll give you an understanding of how objects typically work "under the hood." It will will also introduce you to more Clojure features and give you practice with some fundamentals of functional style (functions as data, recursion, the use of general-purpose data types).

After about 50 pages of object system implementation, we'll have implemented an object system something like Java's, and we'll also have exhausted the example's usefulness for teaching functional style. Those interested in object models as a topic in themselves can temporarily branch off to the optional Part V, which fleshes out the Java-like object model into one inspired by Ruby's.

Part 2, Elements of Functional Style, is where I fulfill the promise to show you how functional languages help you "conceptualize both problems and solutions in substantially different ways."

- The first two chapters are about functional programmers' habit of using basic data types that "flow" through a series of functions. You'll have seen that in Part 1, but these chapters focus on it explicitly and also cover how this style can be hidden inside an object-oriented program.
- The next chapter, Functions That Make Functions, shows one of the main tools of abstraction for functional programs: functions that create other functions in a parameterized way.
- When data is flowing through functions, `if` statements and loops complicate the code. The next chapter is about how to eliminate them from the visible parts of your programs. Most of the emphasis will be an introduction to *monads*. Much as classes abstract away details like the concrete representation of data, monads abstract away details like how the results of a series of computations should be combined. This chapter introduces only some simple monads, but an optional part of the book describes more interesting ones and will give you a deeper understanding of how they work.
- The languages you're used to don't allow you to change the value of 1 to be, say, 2. Clojure, along with many other functional languages, extends the same *immutability* to all data. Once a list is created, you can't add to it or change it. In Part 1, you'll have seen that's not as crazy as it seems—at least for relatively flat data structures. However, things get more difficult when working with deeper structures. New approaches are needed for tasks like "change that 4 way down in that tree to a 5." In the next chapter, I'll show how the "zipper" datatype and its associated functions can be used for just such a task. I'll also work through the implementation of zippers, for two reasons:

  First, zippers illustrate how functional programmers often solve problems by writing code that builds a data structure that represents a computation, then pass that structure around to be used only when (and if) it's needed.

  Second, it provides a more complex example of using basic data types than does the first chapter. We'll see how it's useful to think about the *shape* of the data rather than its type or class.
- Throughout the book, I will have teased you with what seems to be deliberately and ridiculously inefficient code. In the next chapter, I'll show how that apparent inefficiency is an illusion. In reality, we're relying on the runtime to make two kinds of optimizations for us:

*Lazy evaluation*: In functional languages, it's common for some or all values to be *lazy*, meaning that they (and their component parts) are only calculated when demanded. I'll show how this collapses what appear to be many loops into just one. More importantly, I'll show how it allows you to let free of the idea that you must be able to calculate in advance *how much* data you'll need. Instead, you can just ask for *all of it* and let the runtime avoid generating too much.

*Sharing structure behind the scenes*: While *you* can't mutate functional data structures, the language runtime can. I'll show an example of how the runtime can optimize away the wasteful copying your program appears to be doing. I'll also discuss the implications of adopting immutability in object-oriented programs.

- When you start looking at data in terms of its shape, it begins to seem reasonable to have functions decide what code to run based not on explicit `if` tests but rather on matching patterns against shapes.
- Generic functions support a verb-centered way of thinking about the world: there are actions that can apply very broadly. The specifics of an action depend on some properties (determined at runtime) of the values it's applied to. Generic functions are the flip side of the noun-centered approach taken by object-oriented languages.

That finishes the book, except for the optional parts on object models and monads.

## About the exercises

I've taught some of the material in this book in a two-day tutorial. Most of the classroom time is spent doing exercises. They add a lot of value; you should do them. Failing that, you should at least read them and perhaps look at my solutions, as I'm not shy about having later sections build on functions defined by the exercises.

You can find exercise solutions in this book's Github repository[1]. If you use the Git version management tool, you can fetch the code like this:

```
git clone git://github.com/marick/fp-oo.git
```

You can also use your browser to download Zip or Tar files[2].

At last resort, you can browse the repository through your browser. The exercise descriptions include a clickable link to the solutions.

The repository also includes some code that you'll load before starting some of the exercises. The instructions for loading are given later.

[1]https://github.com/marick/fp-oo
[2]https://github.com/marick/fp-oo/downloads

## Testing

I’m a big fan of test-driven design and testing in general, so I’ve written tests for the presupplied code and my exercise solutions. If you’re interested in how to test Clojure code, you can find those tests also on Github[3].

These tests use my own testing library, Midje[4]. With it, you can run my tests like this:

```
709 $ lein midje
## Many lines of output, normally something I hate
## to see from tests. They appear in this case because
## my solution files print the results of examples when
## they're loaded.
All claimed facts (1117) have been confirmed.
710 $
```

To see how to install Midje, see its quickstart[5].

## About the cover

The painting is “Woman with a Parasol” (1875) by Claude Monet, a French Impressionist painter. Impressionist paintings are characterized by a fascination with color and shadow, which you can certainly see here.

I chose this painting because I’m struck by the way the woman (Monet’s wife) is looking down on us with something of a haughty expression. We’ve surprised her—interrupted her—and we need to give an account of ourselves.

This “prove you’re worthy of my attention” attitude has been, I’m sorry to say, all too often characteristic of functional programmers’ interactions with the rest of the programming world. Although that’s changing, it contributes to functional programming’s reputation as forbiddingly rigid, mathematical, and unforgiving of human limitations.

I want this book to be the opposite: informal yet serious, viewing programs as code to be grown rather than mathematical objects to contemplate. That’s the reason I went looking for an Impressionist painting for the cover. The loose brush strokes, the non-rigid edges, the fascination with variety over formal choices—even the embrace of awkwardness (I really don’t like the darker blue squiggles on the left side of the cloud): all these things fit my style of explanation.

[3]https://github.com/marick/fp-oo/test
[4]https://github.com/marick/midje
[5]https://github.com/marick/Midje-quickstart/wiki/Getting-started

## About links

In the PDF version, links within a book appear as colored text, and links to external sites appear in footnotes. Both are clickable.

In the Kindle version, all links appear inline. That makes tasks like installing Clojure or doing exercises awkward: you can't easily read the URLs on the Kindle and type them on your computer. (You have to actually follow the links on the Kindle to see the URLs.) I recommend getting both the Kindle and PDF versions. Use the latter when you need to work with links. (That should only be in exercises, which you can't do on the Kindle anyway.)

## There is a glossary

I never notice a book has a glossary until I turn the last page of the last chapter and discover it. If you're like me, you'll be glad to read now that there's a glossary. (However, if you're like me, you also skim introductions and so will miss this paragraph.)

## Getting help

There's a mailing list[6].

## Notes to reviewers

You can put your comments on the mailing list or by filing issues[7] on Github. It doesn't matter to me. Comments that would benefit from group discussion are better sent to the mailing list.

Please tag your comments with the version of the book to which they apply. The version you're reading now is **garrulous gastropod**.

## Changes to earlier versions

### garrulous gastropod

- Chapters 19 (Ruby-style multiple inheritance), 20 (dynamic binding and send-super), and 21 (objectifying messages in transit) are new.
- The Class as an Object chapter (formerly chapter 7) has been moved into the optional Part V.
- Deleted the last set of exercises in "Inheritance" (chapter 6), moving them into Part V.

[6] https://groups.google.com/group/fp-oo
[7] https://github.com/marick/fp-oo/issues

- Shouldn't have used "seq" as shorthand for "lazy sequence" since non-lazy lists are also technically seqs. Replaced with "lazyseq". This only matters in chapter 13 (and even then the ideas are unaffected).
- Changed the name of the method that makes new objects from `a` to `make`.

### fastidious flounder

- A new chapter (15) on generic functions.
- Bug fixes.
- A coda containing one of my favorite quotes (after chapter 15).

### ecstatic earthworm

- Three new chapters (12, 13, and 14): The zipper data structure as an example of working around the constraint of immutability, the implications of lazy evaluation, and pattern matching.
- Added material about using Light Table as an alternative to Leiningen.
- Miscellaneous fixes to the text.

### discursive diplodocus

- Another reorganization of the unwritten chapters. Part 2 of the book now focuses much more explicitly on characterizing "the elements of functional style". That more clearly fulfills the promise the title of the book makes. See the description of the flow of the book (above) to keep yourself oriented.
- Two new chapters begin Part 2. There's then an unfinished chapter. None of the later chapters will depend on it.
- Five new exercises in "Functions That Make Functions". That chapter has been somewhat rewritten. Those who've read it before need only read the new sections on lifting functions and higher-order functions from the object-oriented perspective.
- That chapter is followed by a chapter on avoiding `if` expressions. It also introduces monads.
- Two optional chapters on monads.

### crafty chameleon

- The flow of the book has changed, as described earlier.
- "Inheritance (and Recursion)" and "The Class as an Object" complete Part 1. "Functions That Make Functions" begins Part 2. "The Class That Makes Classes" begins Part 4.
- I've added tests for exercise sources and solutions[8].

[8]https://github.com/marick/fp-oo/test

**bashful barracuda**

- Use `(load-file "foo.clj")` instead of `(load "foo")`
- Added the glossary.
- Added the first four chapters of Part 1.
- Added an eighth exercise to the first chapter.

## Acknowledgments

I thank these people for commenting on the mailing list and filing bug reports:
Adrian Mowat,
Aidy Lewis,
Ben Moss,
Chris Pearson,
Greg Spurrier,
Jim Cooper,
Juan Manuel Gimeno Illa,
Julian Gamble,
Matt Mower,
Meza,
Mike Suarez,
Oliver Friedrich,
Ondrej Beluský,
Robert D. Pitts,
Robert “Uncle Bob” Martin,
Roberto Mannai,
Stephen Kitt,
Suvash Thapaliya,
Ulrik Sandberg,
and
Wouter Hibma.

And for other help, thanks to Dawn Marick, John MacIntyre, Paul Marick, and Sophie Marick.

## Advertisement

I would be happy to do Ruby or Clojure contract programming or consulting for you. I’m also competent to coach teams on working in the Agile style.

My contact address is marick@exampler.com[9].

[9] mailto:marick@exampler.com

# 1. Just Enough Clojure

Clojure is a variant of Lisp, one of the oldest computer languages. You can view it as a small core surrounded by a lot of library functions. In this chapter, you'll learn most of the core and functions you'll need in the rest of the book.

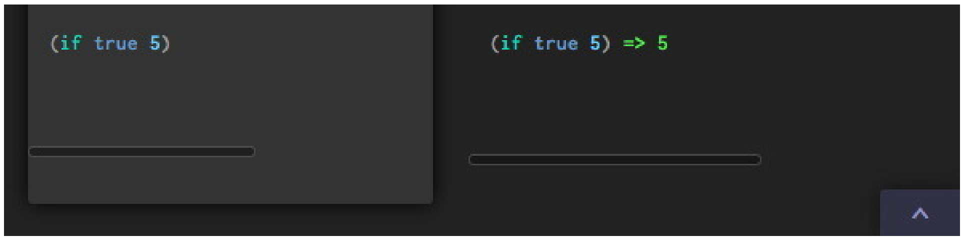
## 1.1 Installing Clojure

At this moment, there seem to be two attractive choices for running Clojure. In this section, I'll describe how to get started with either. If you have trouble, visit the installation troubleshooting page[1] or ask for help on the mailing list[2].

### Light Table

One simple way to install Clojure is to install the Light Table playground[3] instead. It is something like an IDE for Clojure, with the interesting property that it evaluates expressions as you type them. You can find out more at Chris Granger's site[4].

You'll work in something Light Table calls the "Instarepl". That's its version of the Clojure *read-eval-print loop* (typically called "the repl"). You type text in the left half of the window, and the result appears in the right. It looks like this:



```
(if true 5)
```

```
(if true 5) => 5
```

**Light Table**

The book doesn't show input and output in the same split screen style. Instead, it shows input preceded by a prompt, and output starting on the next line:

[1] https://github.com/marick/fp-oo/wiki/Installation-Troubleshooting
[2] https://groups.google.com/group/fp-oo
[3] http://app.kodowa.com/playground
[4] http://www.chris-granger.com/2012/04/12/light-table---a-new-ide-concept/

```
user=> (if true 5)
5
```

Important: as of this writing, Light Table does not automatically include all the normal repl functions. You have to manually include them with this magic incantation:

```
(use 'clojure.repl)
```

(Note the single-quote mark: it's required.)

## Leiningen

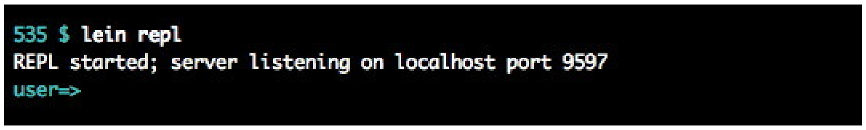
If you want to run Clojure from the command line, first install Leiningen[5]. Go to its page and follow the installation instructions.

When you're finished, you can type the following to your command line:

```
lein repl
```

That asks Leiningen to start the *read-eval-print loop* (typically called "the repl"). Don't be alarmed if nothing happens for a few seconds: because the Java Virtual Machine is slow to start, Clojure is too.

All is well when you see something like this:



```
535 $ lein repl
REPL started; server listening on localhost port 9597
user=>
```

**A command line repl, with a fashionable black background**

From now on, I won't use screen shots to show repl input and output. Instead, I'll show it like this:

```
user=> (if true 5)
5
```

The most important thing you need to know now is how to get *out* of the repl. That's done like this:

```
user=> (exit)
```

On Unix machines, you can also use Control-D to exit.

[5]https://github.com/technomancy/leiningen

## 1.2 Working with Clojure

All the exercises in this book can be done directly in the repl. However, many editors have a “clojure mode” that knows about indentation conventions, helps you avoid misparenthesizations, and helpfully colorizes Clojure code.

- Emacs: clojure-mode[6]
- Vim: one is VimClojure[7]

You can copy and paste Clojure text into the repl. It handles multiple lines just fine.

Many people like to follow along with the book by copying lines from the PDF version and pasting them into the repl. That’s usually safe. However, my experience is that such text is sometimes (but not always!) missing some newlines. That can cause perplexing errors when it results in source code ending up on the same line as a to-the-end-of-the-line comment. In later chapters (where there are more lines with comments), I’ll provide text files you can paste from.

If you want to use a Clojure command to load a file like (for example) `solutions/add-and-make.clj`, use this:

```
user> (load-file "solutions/add-and-make.clj")
```

*Warning*: I’m used to using `load` in other languages, so I often reflexively use it instead of `load-file`. That leads to this puzzling message:

```
user=> (load "sources/without-class-class.clj")
FileNotFoundException Could not locate sources/without-class-class.
clj__init.class or sources/without-class-class.clj.clj on classpath:
clojure.lang.RT.load (RT.java:432)
```

The clue to my mistake is the “.clj.clj” on the next-to-last line.

## 1.3 The Read-Eval-Print Loop

Here’s a use of the repl:

[6] https://github.com/technomancy/clojure-mode/blob/master/README.md
[7] https://github.com/vim-scripts/VimClojure

```
user=> 1
1
```

More is happening than just echoing the input to output, though. This profound calculation requires three steps.

First, the *reader* does the usual parser thing: it separates the input into discrete tokens. In this case, there's one token: the string `"1"`. The reader knows what numbers look like, so it produces the number `1`.

The reader passes its result to the *evaluator*. The evaluator knows that numbers evaluate to themselves, so it does nothing.

The evaluator passes its result to the *printer*. The printer knows how to print numbers, so it does.

Strings, truth values, and a number of other types play the same self-evaluation game:

```
user=> "hi mom!"
"hi mom!"

user=> true
true
```

Let's try something more exciting:

```
user=> *file*
"NO_SOURCE_PATH"
```

`*file*` is a *symbol*, which plays roughly the same role in Clojure that identifiers do in other languages. The asterisks have no special significance: Clojure allows a wider variety of names than most languages. Most importantly, Clojure allows dashes in symbols, so Clojure programmers prefer them to underscores. StudlyCaps or interCaps style is uncommon in Clojure.

Let's step through the read-eval-print loop for this example. The reader constructs the symbol from its input characters. It gives that symbol to the evaluator. The evaluator knows that symbols do *not* evaluate to themselves. Instead, they are associated with (or *bound to*) a value. `*file*` is bound to the name of the file being processed or to `"NO_SOURCE_PATH"` when we're working at the repl.

Here's a slightly more interesting case:

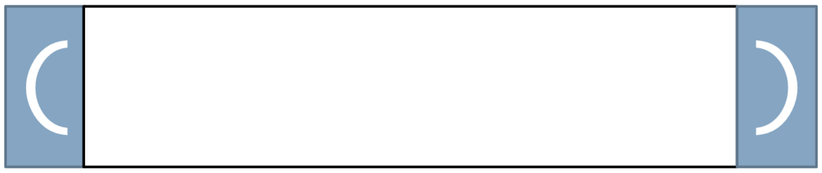
```
user=> +
#<core$_PLUS_ clojure.core$_PLUS_@38a92aaa>
```

The value of the symbol `+` is a *function*. (Unlike many languages, arithmetic operators are no different than any other function.) Since functions are executable code, there's not really a good representation for them. So, as do other languages, Clojure prints a mangled representation that hints at the name.

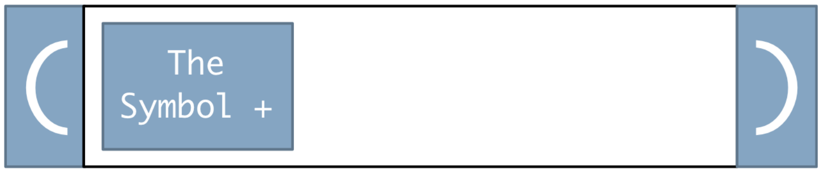
Now let's walk through what happens when you ask the repl to add one and two to get three:

```
user> (+ 1 2)
3
```

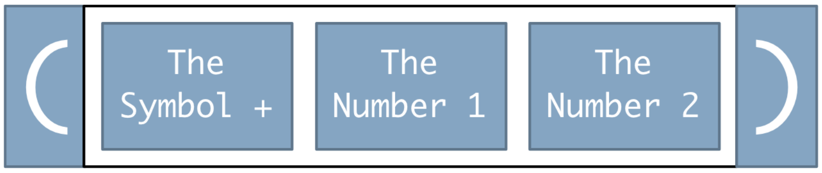
In this case, the first token is a parenthesis, which tells the reader to start a list, which I'll represent like this:



The second token represents the symbol +. It's put in the list:



The next two tokens represent numbers, and they are added to the list. The closing parenthesis signals that the list is complete:



The reader's job is now done, and it gives the list to the evaluator. Lists are a case we haven't described before. The evaluator handles them in two steps:

1. It recursively evaluates each of the list elements.
   - The symbol + evaluates to the function that adds.
   - As before, numbers evaluate to themselves.
2. The first value from a list *must* be a function (or something that behaves like a function). The evaluator *applies* the function to the remaining values (its arguments). The result is the number `3`.

The printer handles `3` the same way it handled `1`.

To emphasize how seriously the evaluator expects the first element of the list to be a function, here's what happens if it's not:

```
user=> (1 2 3)
java.lang.ClassCastException: java.lang.Integer cannot be cast to
clojure.lang.IFn (NO_SOURCE_FILE:0)
```

(I'm afraid that Clojure error messages are sometimes not as clear as they might be.)

## 1.4 A note on terminology: "value"

Since Clojure is implemented on top of the Java runtime, things like functions and numbers are Java objects. I'm going to call them *values*, though. In a book making distinctions between object-oriented and functional programming, using the word "object" in both contexts would be confusing. Moreover, Clojure values are (usually) used very differently than Java objects.

## 1.5 Functions are values

Clojure can do more than add. It can also ask questions about values. For example, here's how you ask if a number is odd:

```
user> (odd? 2)
false
```

There are other predicates that let you ask questions about what kind of value a value is:

```
user> (number? 1)
true
```

You can ask the `number?` question of a function:

```
user> (number? +)
false
```

If you want to know if a value is a function, you use `fn?`:

```
user> (fn? +)
true
user> (fn? 1)
false
```

It's important to understand that functions aren't special. Consider these two expressions:

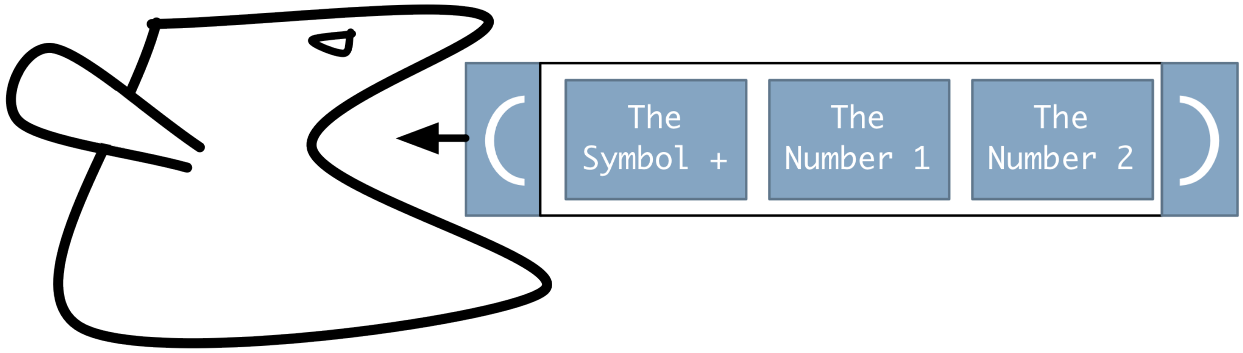
```
(+ 1 2)
(fn? +)
```

In one case, the + function gets called; in the other, it's examined. But the difference between the two cases is solely due to the *position* of the symbol +. In the first case, its position tells the evaluator that its function is to be executed; in the second, that the function is to be given as an argument to `fn?`.

## 1.6 Evaluation is substitution

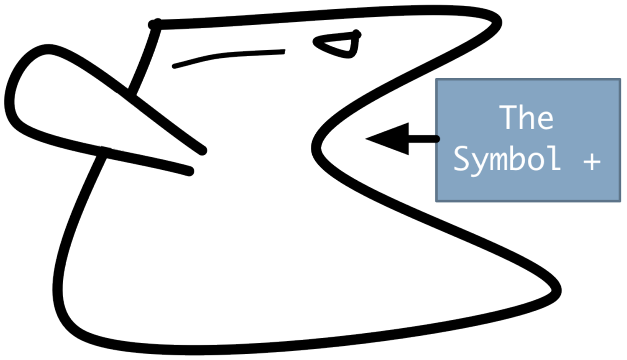
Let's look in more detail at the evaluator. This may seem like overkill, but one of the core strengths of functional languages is the underlying simplicity of their evaluation strategies.

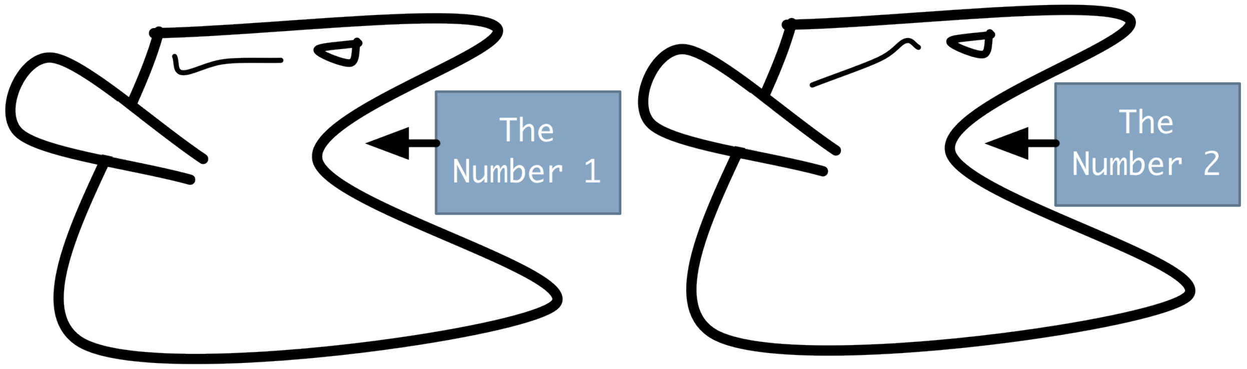
To add a little visual interest, let's personify the evaluator as a bird. That's appropriate because parent birds take in food, process it a little, then feed the result to their babies. The evaluator takes in data structures from the reader, processes them, and feeds the result to the printer. Here's a picture of the evaluator at taking in a list:



An evaluator is a lazy bird, though. Whenever it sees a compound data structure, it summons other evaluators to do part of the work.

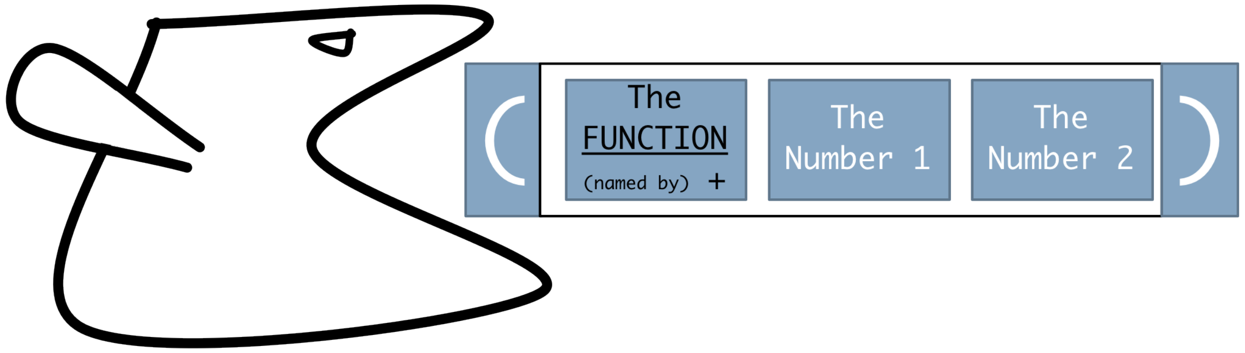
A list is a compound data structure, so (in this case) three evaluators are set to work:





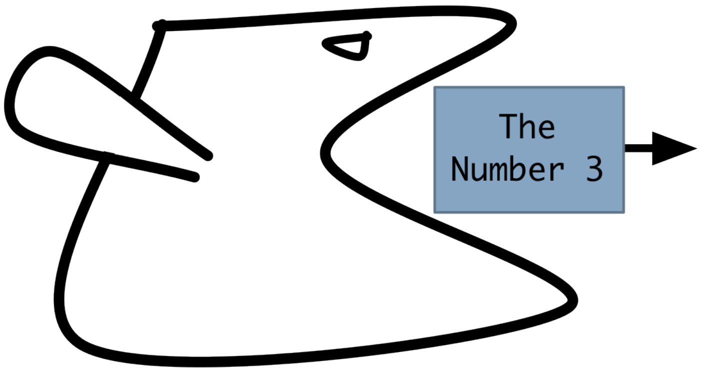
The bottom two have an easy job: numbers are already digested (evaluate to themselves), so nothing need be done. The top bird must convert the symbol into the function it names.

Each of these sub-evaluators feeds its result to the original evaluator, which substitutes those values for the originals, making a list that is almost–but not quite–the same:



(The symbol has been substituted with a function.)

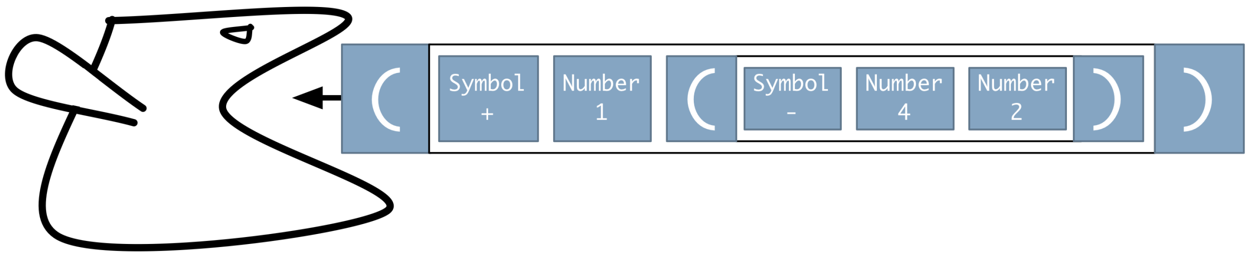
The original evaluator must process this list according to the rules for that data structure. That means calling the function in the first position and giving it the rest of the list as arguments. Since this is the top-level evaluator, that result is provided as nourishment to the printer:



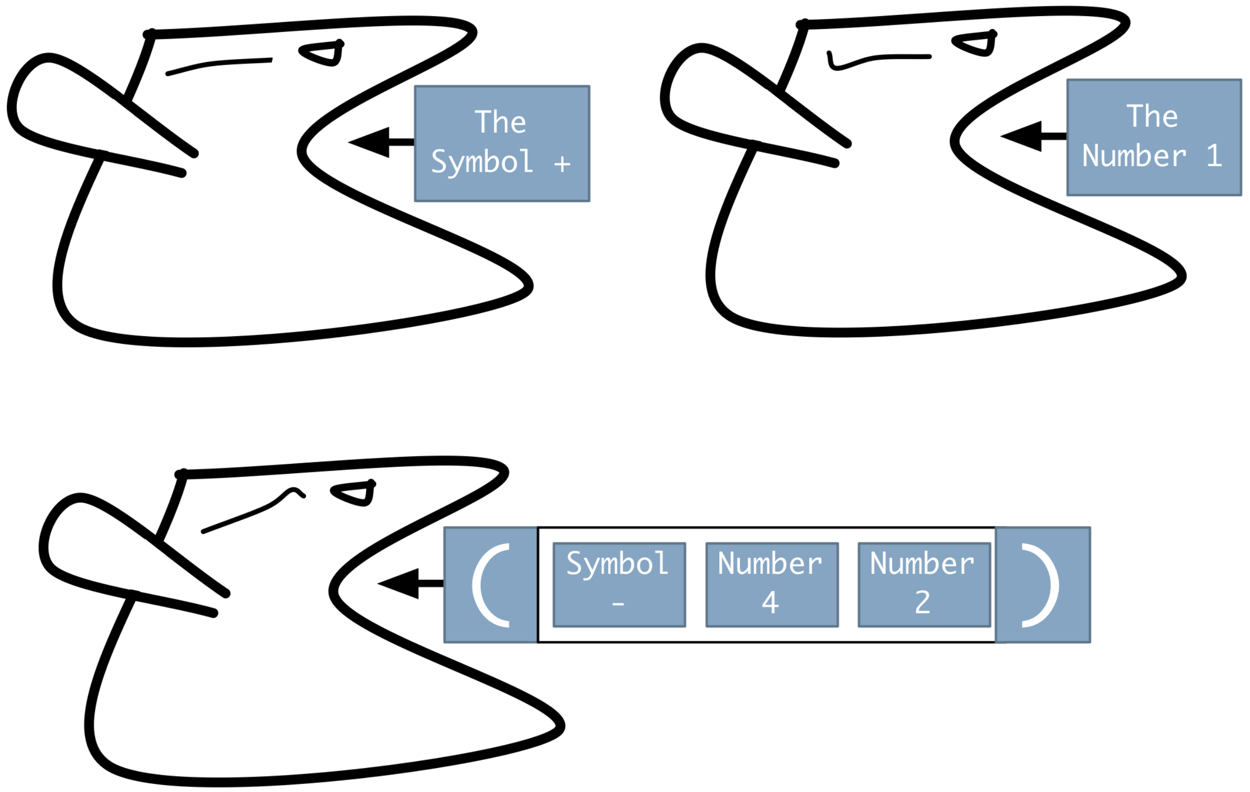
Now consider a longer code snippet, like this:

```
user=> (+ 1 (- 4 2))
3
```

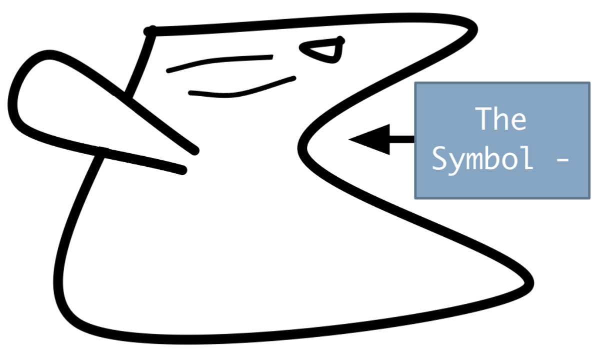
The first (top-level) evaluator is to consume this:

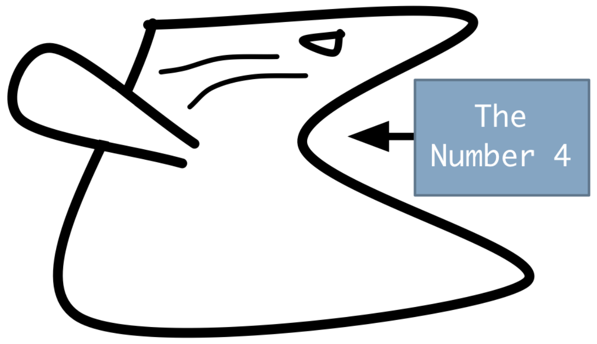


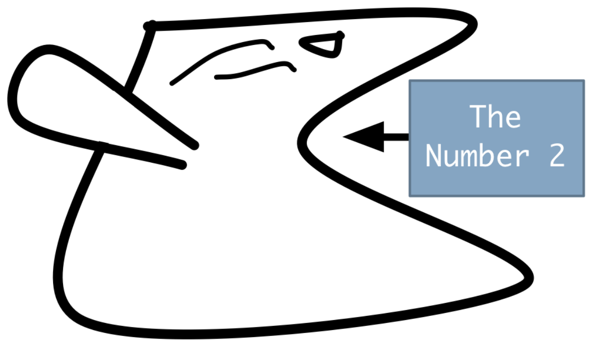
“Too complicated!” it thinks, and recruits three fellows for the three list elements:



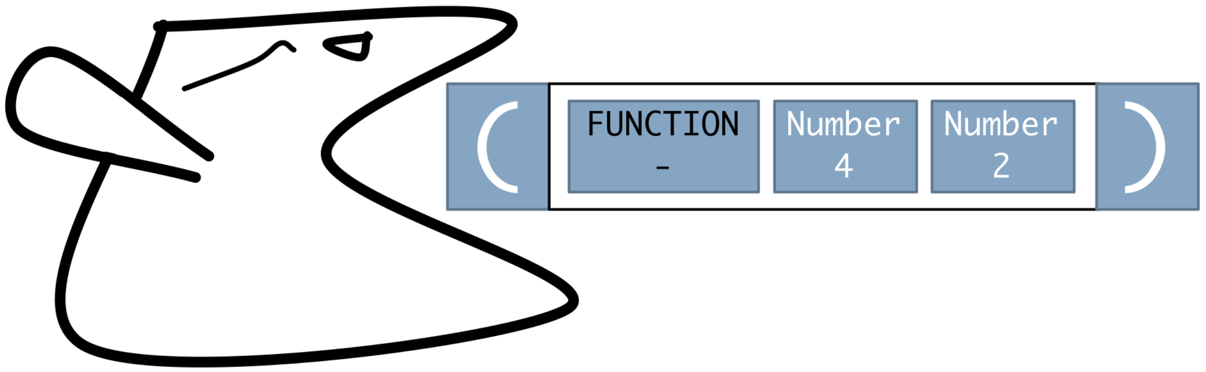
The first two are happy with what they've been fed, but the last one has gotten another list, so it recruits three *more* fellows:





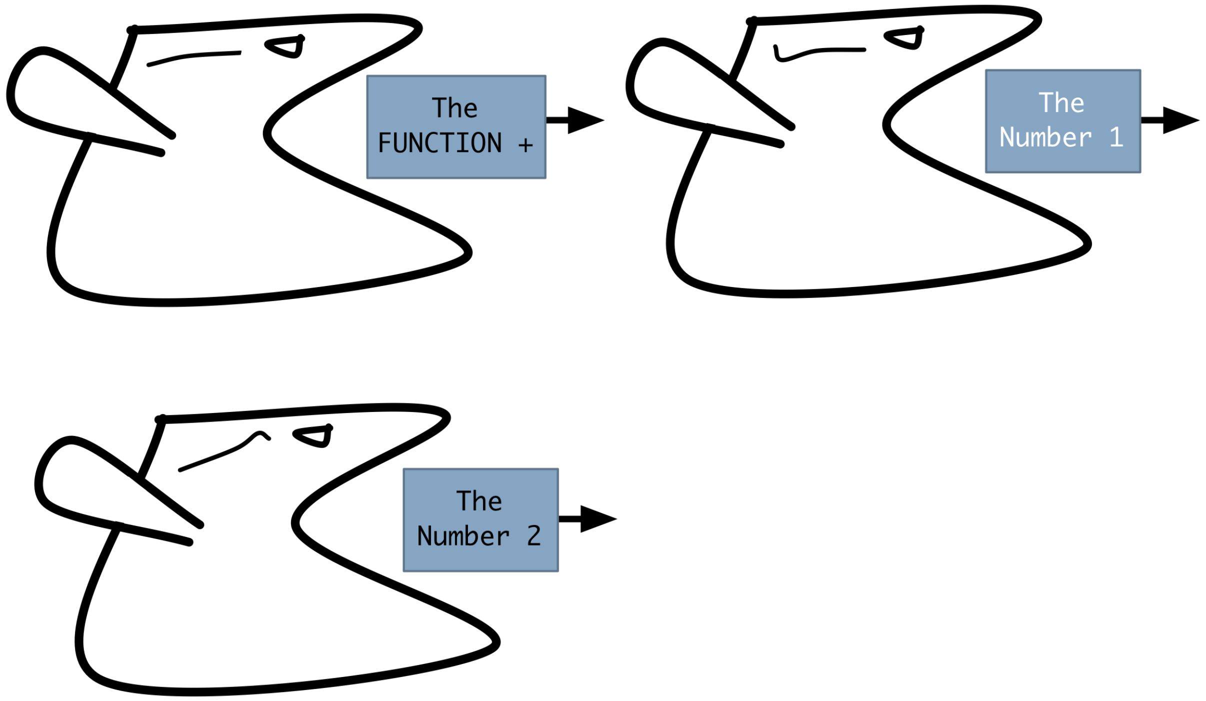


When the third-level birds finish, the lazy second-level bird substitutes the values they provide, so it now has this list:

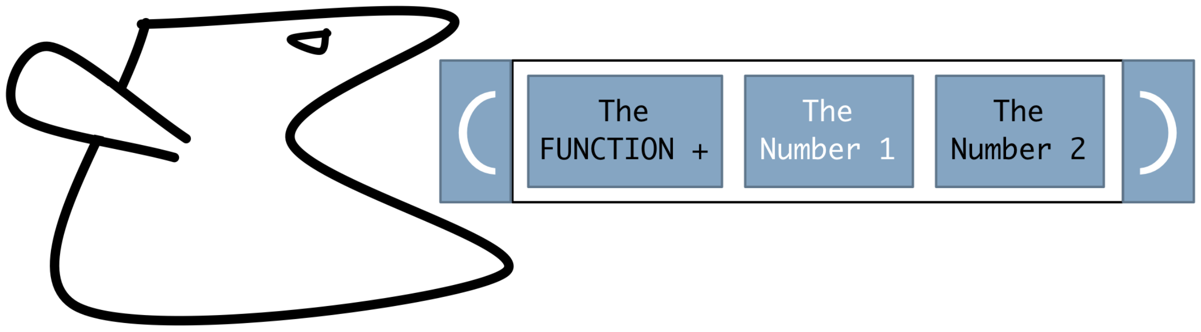


It applies the `-` function to `4` and `2`, producing the new value `2`.

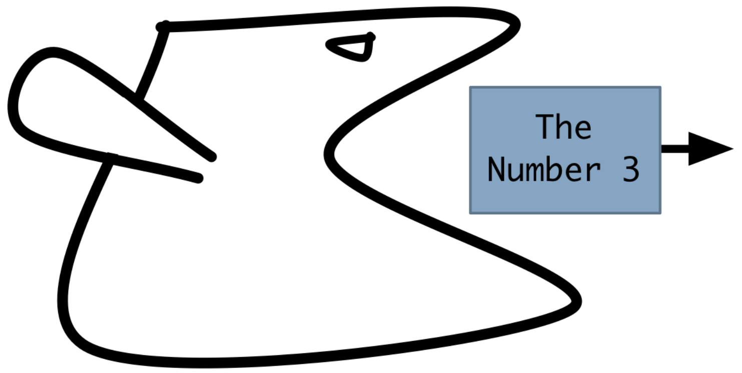
When all the second-level birds are done, they feed their values to the top-level evaluator:



The top-level evaluator substitutes them in:



It applies the function to the two arguments, yielding 3, which it feeds to the printer:

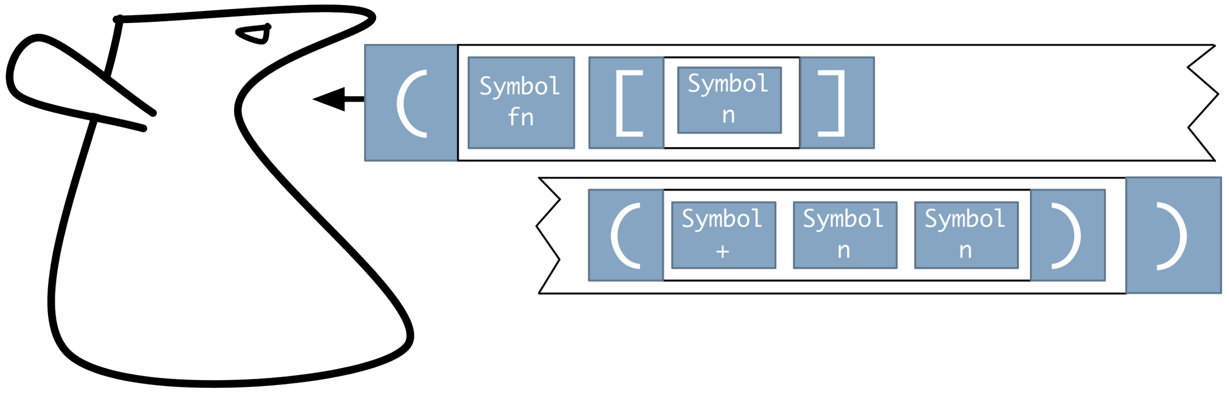


## 1.7 Making functions

Suppose you type this:

```
user> (fn [n] (+ n n))
```

That's another list handed to the evaluator, like this:

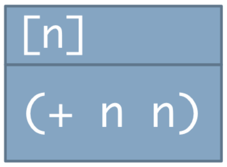


As another list headed by a symbol, this looks something like the function applications we've seen before. However, `fn` is a *special* symbol, handled specially by any evaluator. There are a smallish number of special symbols in Clojure. The expressions headed by a special symbol are called *special forms*.

In the case of this special form, the evaluator doesn't recruit a flock to handle the individual elements. Instead, it conjures up a new function. In this case, that function takes a single parameter[8], `n`, and has as its *body* the list `(+ n n)`. Note that the parameter list is surrounded by square brackets, not parentheses. (That makes it a bit easier to see the structure of a big block of code.)

[8]I will consistently use "argument" for real values given to functions and "parameter" for symbols used to name arguments in function definitions. So `n` is a parameter, while a real number given to our doubling function would be an argument.

I'll draw function values like this:



The functions you create are just as "real" as the functions that come pre-supplied with Clojure. For example, they print just as helpfully:

```
user=> (fn [n] (+ n n))
#<user$eval66$fn__67 user$eval66$fn__67@5ad75c47>
```

Once a function is made, it can be used. How? By putting it in the first position of a list:

```
user> ( (fn [n] (+ n n)) 4)
        ________________
8
```

(I've used the underlining to highlight the first position in the list.)

Although more cumbersome, the form above is conceptually no different than this:

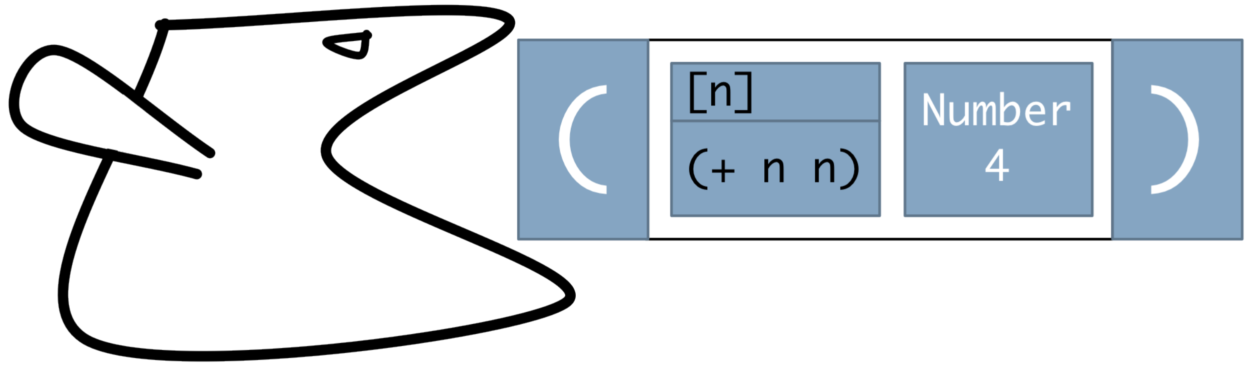
```
user> (+ 1 2)
3
```

In both cases, a function value will be applied to some arguments.

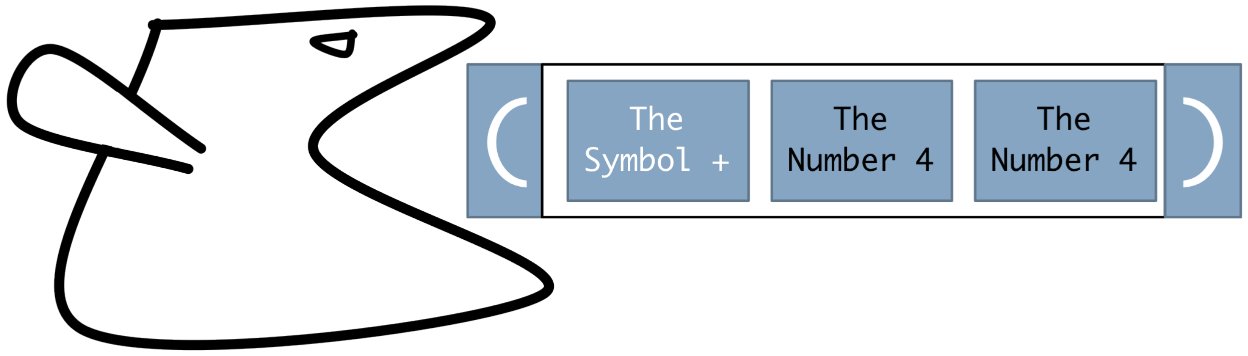
## 1.8 More substitution as evaluation

In `( (fn [n] (+ n n)) 4)`, our doubling function is applied to `4`. How, precisely, does that work?

The whole thing is a list, so the top-level evaluator recruits two lower-level evaluators. One evaluates a list headed by the symbol `fn`; the other evaluates a number. When they hand their values to the top-level evaluator, it substitutes, so it now has this:



It processes this function by *substituting* the actual argument, `4`, for its matching parameter, `n`, *anywhere in the body of the function*:



Hey! Look! A list! We know how to handle a *list*. The list elements are evaluated by sub-evaluators, and the resulting function (the `+` function value) is applied to the resulting arguments. Were the `+` function value a user-written function, it would also be evaluated by substitution. So would be most of the Clojure library functions. There are some *primitive* functions, though, that are evaluated by Java code. (It can't be turtles *all* the way down.)

Despite being tedious, this evaluation procedure has the virtue of being simple. (Of course, the real Clojure compiler does all sorts of optimizations.) But you may be thinking that it has the disadvantage that it **can't possibly work**. What if the code contained an assignment statement, something like the following?

```
(fn [n]
  (assign n (+ 1 n))
  (+ n n))
```

It doesn't make sense to substitute a number into the left-hand side of an assignment statement. (What are you going to do, assign `4` the new value `5`?) And even if you *did* change `n`'s value on the first line of the body, the two instances of `n` on the second line have already been substituted away, something like this:

```
  (assign n (+ 1 4))
  (+ 4 4))
```

Therefore, the assignment can't have an effect on any following code.

Clojure avoids this problem by not having an assignment statement. With one exception that you'll see shortly, there is no way to change the binding between a symbol and a value.

"Still," you might object, "what if the argument is some data structure that the code modifies? If you pre-substitute the whole data structure wherever the parameter appears, changes to the data structure in one part of the function won't be seen in other parts of the function!" Code suffering from this problem would look something like this:

```
(fn [tree]
  (replace-left-branch tree 5)
  (duplicate-left-branch tree))
```

Because of substitution, the tree on the last line that's having its left branch duplicated is not the tree that had its left branch replaced on the previous line.

Clojure avoids this problem by not allowing you to change trees, sets, vectors, lists, hashmaps, strings, or anything at all except a few special datatypes. In Clojure, you don't modify a tree, you create an entirely new tree containing the modifications. So the function above would look like this:

```
(fn [tree]
  (tree-with-duplicated-left-branch
    (tree-with-replaced-left-branch tree 5)))
```

We'll be discussing the details of all this in the chapter on immutability. For now, ignore your efficiency concerns–"Duplicating a million-element vector to change *one* element?!"–and delegate them to the language implementor. Also hold off on thinking that programming without an assignment statement has to be crazy hard–it's part of this book's job to show you it's not.

## 1.9 Naming things

You surely don't want to create the doubling function every time you use it. Instead, you want to create it once, then bind it to some sort of global symbol.

```
user> (def twice (fn [n] (+ n n)))
#'user/twice
```

Once named, a user-defined function can be used as conveniently as a built-in function:

```
user=> (twice 10)
20
```

Since functions are values not essentially different than other values, you might expect that you can give names to strings, numbers, and whatnot. Indeed you can:

```
user> (def two 2)
#'user/two
user> (twice two)
4
```

You can use `def` with a particular symbol more than once. That's the only exception to Clojure's "no changing a binding" rule[9]. It's useful for correcting mistakes:

```
user=> (def twice (fn [n] (- n n)))
user=> (twice 10)
0
user=> ;; Darn!  (This, by the way, is a Clojure comment.)
user=> (def twice (fn [n] (+ n n)))
user=> (twice 10)
20
```

## 1.10 Lists

We've seen that lists are surrounded by parentheses. The evaluator function interprets a list as an excuse to apply a function to arguments. But lists are also a useful data structure. How do you say that you want a list to be treated as data, not as code? Like this:

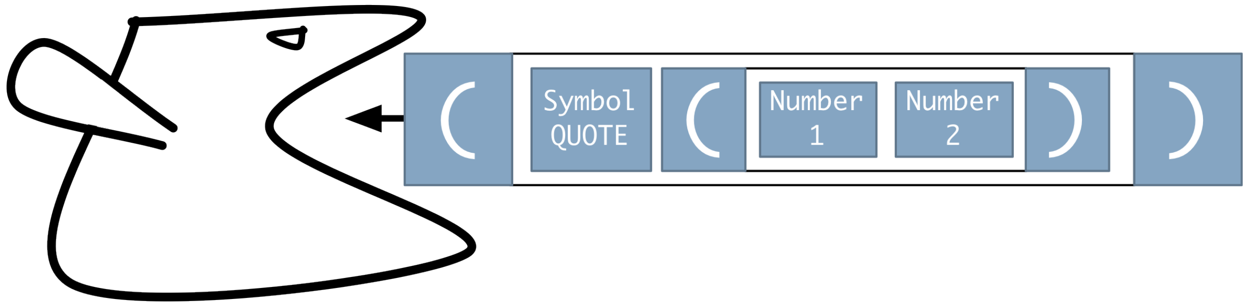
```
user> '(1 2)
(1 2)
```

The quote tells the evaluator not to interpret the list as a function call. That character is actually syntactic sugar for a more verbose notation:

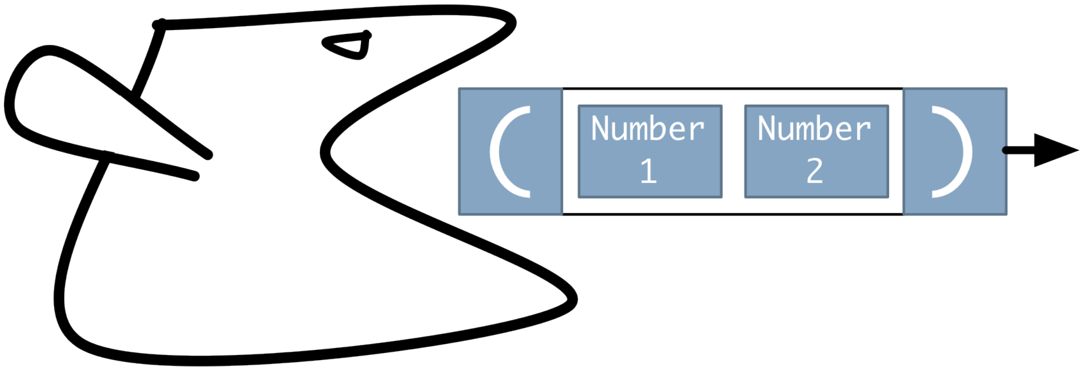
```
user> (quote (1 2))
(1 2)
```

[9]`def` isn't actually an exception. It looks like just another way of associating a symbol with a value, but it's actually doing something different. The difference, though, is irrelevant to this book, and would just complicate your understanding to no good end, so I'm going to ignore it. If you're curious, see the description of `Var` in the Clojure documentation or any other book on Clojure.

The reader gives the evaluator this list:



The evaluator notices that the first element is the special symbol `quote`. Instead of unleashing sub-evaluators, it digests the form into its single argument, which is what it feeds to the printer:



You can also create a list with a function:

```
user> (list 1 2 3 4)
(1 2 3 4)
```

You can take apart lists. Here's a way to get the first element:

```
user> (first '(1 2 3 4))
1
```

Here's a way to get everything *after* the first element:

```
user> (rest '(1 2 3 4))
(2 3 4)
```

You can pick out an element at a particular (zero-based) position in a list:

```
user> (nth '(1 2 3 4) 2)
3
```

**Exercise 1**: Given what you know now, can you define a function `second` that returns the second element of a list? That is, fill in the blank in this:

```
user> (def second (fn [list] ____))
```

Be sure to try your solution at the repl. (When you do, you'll notice that you've just overridden Clojure's built-in `second` function. Don't worry about that.)

You can find solutions to this chapter's exercises in `solutions/just-enough-clojure.clj`[10].

**Exercise 2**: Give two implementations of `third`, which returns the third element of a list.

## 1.11 Vectors

Lists are (roughly) the classic linked list that many of us encountered when we first learned programming. That means code has to traverse the whole list to get to the last element. Clojure's creator cares about efficiency, so Clojure also makes it easy to use vectors, where it takes no more time to access the last element than the first.

Vectors have a literal notation, in which the elements are surrounded by square brackets:

```
user> [1 2 3 4]
[1 2 3 4]
```

Note that I didn't have to quote the vector to prevent the evaluator from trying to use the value of `1` as a function. That only happens with lists.

There's also a function-call notation for creating vectors:

```
user> (vector 1 2 3 4)
[1 2 3 4]
```

The `first`, `rest`, and `nth` functions also work with vectors. Indeed, most functions that apply to lists also apply to vectors.

## 1.12 Vector? List? Who cares?

Both vectors and lists are *sequential* datatypes.

[10]https://github.com/marick/fp-oo/blob/master/solutions/just-enough-clojure.clj

```
user=> (sequential? [1 2 3])
true
user=> (sequential? '(1 2 3))
true
```

There's a third datatype called the *lazyseq* (for "lazy sequence") that's also sequential. That datatype won't be relevant until we discuss laziness. I mention it because some functions that you might think produce vectors actually produce lazyseqs. For example, consider this:

```
user=> (rest [1 2 3])
(2 3)
```

The first time I typed something like that, I expected the result to be the vector `[2 3]`, and the parentheses confused me. The result of `rest` is a lazyseq, which prints the same way as a list. Here's how you can tell the difference:

```
user=> (list? (rest [1 2 3]))
false
user=> (seq? (rest [1 2 3]))
true
user=> (vector? (rest [1 2 3]))
false
```

Such changes of type seem like they'd lead to bugs. In fact, the differences almost never matter. For example, equality doesn't depend on the type of a sequential data structure, only on the contents. Therefore:

```
user=> (= [2 3] '(2 3))
true
user=> (= [2 3] (rest [1 2 3]))
true
```

The single most obvious difference between a list and vector is that you have to quote lists.

It will never matter in this book whether you create a list or vector, so suit your fancy. I will often use "sequence" from now on when the difference is irrelevant.

**seqs**

The predicate `seq?` doesn't actually check specifically for a lazyseq. It responds `true` for both lists and lazyseqs, and the word *seq* is used as an umbrella term for both types. If you really need to know the complete set of sequential types and the names that refer to them, see the table below. However, the definition of a seq will never matter for this book.

| | **Lists** | **Vectors** | **Lazyseqs** |
|---|---|---|---|
| sequential? | YES | YES | YES |
| seq? | YES | no | YES |
| list? | YES | no | no |
| vector? | no | YES | no |
| coll? (for “collection”) | YES | YES | YES |

## 1.13 More on evaluation and quoting

When the evaluator is given a list from the reader, it first evaluates each element of the list, then calls the first element as a function. When it’s given a vector, it first evaluates each element and then packages the resulting values as a (different) vector.

That means that literal vectors can (and often do) contain code snippets:

```
user=> [ (+ 1 1) (- 1 1) ]
[2 0]
```

It also means quoting is sometimes required for vectors as well as lists. Can you guess the results of these two code snippets?

- `[inc dec]`
- `'[inc dec]`

The first is a vector of two *functions*:

```
user=> [inc dec]
[#<core$inc clojure.core$inc@13ab6c1c>
 #<core$dec clojure.core$dec@7cdd7786>]
```

The second is a vector of two *symbols* (which happen to name functions):

```
user=> '[inc dec]
[inc dec]
```

At first, you’re likely to be confused about when you need to quote. Basically, if you see an error like this:

```
java.lang.Exception: Unable to resolve symbol: foo in this context
(NO_SOURCE_FILE:67)
```

… it probably means you forgot a quote.

Quoting doesn't only apply to entire lists and vectors. You can quote symbols:

```
user=> 'a
a
```

You can also quote individual elements of vectors:

```
user=> [ 'my 'number (+ 1 3) ]
[my number 4]
```

## 1.14 Conditionals

Despite the anti-if campaign[11], the conditional statement is one of primordial operations of the Turing Machine (that is, computer). Conditionals in Clojure look like this:

```
user=> (if (odd? 3)
         (prn "Odd!")
         (prn "Even!"))
"Odd!"
nil
```

The `prn` function prints to the output. Unlike in some languages, `if`s are expressions that produce values and can be embedded within other expressions. The value of an `if` expression is the value of the "then" or "else" case (whichever is chosen). Since `prn` always returns the value `nil`, that's what the repl printed in the example above. (`nil` is called `null` in some other languages–it's the object that is no object, or the pointer that points to nothing, or what Tony Hoare called his billion-dollar mistake[12].)

## 1.15 Rest arguments

Clojure functions can take a variable number of arguments:

[11]http://www.antiifcampaign.com/
[12]http://lambda-the-ultimate.org/node/3186

```
user> (add-squares 1 2)
5
user> (add-squares 1 2 3)
14
```

As with other languages, there's a special token in a function's parameter list to say "Take any arguments after this point and wrap them up in a sequential collection (list, vector, whatever)". Clojure's looks like this:

```
user> (   (fn [& args] args) 1 2 3 4)
          __________________
(1 2 3 4)
```

That function gathers all the arguments into the list `args`, which it then returns.

Note the space after the `&`. It's required.

Now that we know how to define *rest arguments*, here's what `add-squares`' definition would look like:

```
(def add-squares
    (fn [& numbers]
      (...something... numbers)))
```

What could the "something" be? The next section gives us a clue.

## 1.16 Explicitly applying functions

Suppose we have a vector of numbers we want to add up:

```
[1 4 9 16]
```

That is, we want to somehow turn that vector into the same value as this + expression:

```
user=> (+ 1 4 9 16)
30
```

(Notice that + can take any number of arguments.)

The following is *almost* what we want, but not quite:

```
user=> (+ [1 4 9 16])
java.lang.ClassCastException (NO_SOURCE_FILE:0)
```

What we need is some function that hands all the elements of the vector to `+` as if they were arguments directly following it in a list. Here's that function:

```
user=> (apply + [1 4 9 16])
30
```

`apply` isn't magic; we can define it ourselves. I think of it as turning the second argument into a list, sticking the first argument at the front, and then evaluating the result in the normal way a list is evaluated. Or:

```
(def my-apply
    (fn [function sequence]
      (eval (cons function sequence))))
```

Let's look at this in steps.

1. `cons` (the name chosen more than 50 years ago[13] as an abbreviation for the verb "construct") produces a list[14] whose `first` element is `cons`'s first argument and whose `rest` is `cons`'s second argument:

   ```
   user=> (cons "the first element" [1 2 3])
   ("the first element" 1 2 3)
   ```

   (Notice that `cons`, like `rest` earlier, takes a vector in but doesn't produce one.)
2. `eval` is our old friend the bird-like evaluator. In `my-apply`, it's been given a list headed by a function, so it knows to apply the function to the arguments.

According to the substitution rule, `(my-apply + [1 2 3])` is first converted to this:

```
       (eval
         (cons + [1 2 3]))
```

After that, it is evaluated "from the inside out", each result being substituted into an enclosing expression, finally yielding 6.[15]

[13]http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf

[14]Strictly, `cons` produces a lazyseq, not a list, but the evaluator treats them the same.

[15]The substitution as printed isn't quite true. After it receives `(my-apply + ...)` from the reader, the evaluator processes the symbols `function` and `+` to find function values. Therefore, in the expansion of `my-apply`, the `function` parameter is substituted with an argument that's a function value. And so the list given to `eval` starts with a function value, not a symbol. That's different than what we've seen before. But it still works fine, because a function value self-evaluates the way a number does. I opted for the easier-to-read expansion.

## 1.17 Loops

How do you write loops in Clojure?

You don't (mostly).

Instead, like Ruby and other languages, Clojure encourages the use of functions that are applied to all elements of a sequence. For example, if you want to find all the odd numbers in a sequence, you'd write something like this:

```
user> (filter odd? [1 2 3 4])
(1 3)
```

The `filter` function applies its first argument, which should be a function, to each element of its second argument. Only those that "pass" are included in the output.

Question: How would you find the first odd element of a list?

Answer:

```
user> (first (filter odd? [1 2 3 4]))
1
```

Question: Isn't that grossly inefficient? After all, `filter` produces a whole list of odd numbers, but you only want the first one. Isn't the work of producing the rest a big fat waste of time?

Answer: No. But you'll have to read the later discussion of laziness to find out why.

The `map` function is perhaps the most common loop-like function. (If you know Ruby, it's the same as `collect`.) It applies its first argument (a function) to each element of a sequence and produces a sequence of the results. For example, Clojure has an `inc` function that returns one plus its argument. So if you want to increment a whole sequence of numbers, you'd do this:

```
user> (map inc [0 1 2 3])
(1 2 3 4)
```

The `map` function can take more than one sequence argument. Consider this:

```
user> (map * [0 1 2 3]
             [100 200 300 400])
(0 200 600 1200)
```

That is equivalent to this:

```
user=> (list (apply * [0 100])
             (apply * [1 200])
             (apply * [2 300])
             (apply * [3 400]))
```

## 1.18 More exercises

**Exercise 3**: Implement `add-squares`.

```
user=> (add-squares 1 2 5)
30
```

**Exercise 4**: The `range` function produces a sequence of numbers:

```
user=> (range 1 5)
(1 2 3 4)
```

Using it and `apply`, implement a bizarre version of factorial that uses neither iteration nor recursion.

*Hint*: The factorial of 5 is 1*2*3*4*5.

**Exercise 5**: Below, I give a list of functions that work on lists or vectors. For each one, think of a problem it could solve, and solve it. For example, we've already solved two problems:

```
user> ;; Return the odd elements of a list of numbers.
user> (filter odd? [1 2 3 4])
(1 3)
user> ;; (One or more semicolons starts a comment.
user>
user> ;; Increment each element of a list of numbers,
user> ;; producing a new list.
user=> (map inc [1 2 3 4])
(2 3 4 5)
```

You'll probably need other Clojure functions to solve the problems you put to yourself. Therefore, I also describe some of them below.

Clojure has a built-in documentation tool. If you want documentation on `filter`, for example, type this at the repl:

```
user=> (doc filter)
-------------------------
clojure.core/filter
([pred coll])
  Returns a lazy sequence of the items in coll for which
  (pred item) returns true. pred must be free of side-effects.
```

Many of the function descriptions will refer to "sequences", "seqs", "lazy seqs", "colls", or "collections". Don't worry about those distinctions. For now, consider all those synonyms for "either a vector or a list".

In addition to the built-in `doc`, clojuredocs.org[16] has examples of many Clojure functions.

**Functions to try**

- `take`
- `distinct`
- `concat`
- `repeat`
- `interleave`
- `drop` and `drop-last`
- `flatten`
- `partition` only the `[n coll]` case, like: `(partition 2 [1 2 3 4])`
- `every?`
- `remove` and create the function argument with `fn`

**Other functions**

- `(= a b)` – Equality
- `(count sequence)` – Length
- `and`, `or`, `not` – Boolean functions
- `(cons elt sequence)` – Make a new sequence with the `elt` on the front
- `inc`, `dec` – Add and subtract one
- `(empty? sequence)` – Is the sequence empty?
- `(nth sequence index)`– Uses zero-based indexing
- `(last sequence)` – The last element
- `(reverse sequence)` – Reverses a sequence
- `print`, `println`, `prn` – Print things. `print` and `println` print in a human-friendly format. For example, strings are printed without quotes. `prn` prints values in the literal format you'd type to create them. For example, strings are printed with double quotes. `println` and `prn` add a trailing newline; `print` does not. All of these can take more than one argument.

[16]http://clojuredocs.org/

- `pprint` – Short for "pretty print", it prints a nicely formatted representation of its single argument.

Note: if the problems you think of are anything like the ones I think of, you'll want to use the same sequence in more than one place, which would lead to annoying duplication. Since you don't know how to create local variables yet, the easiest way to avoid duplication is to create and call a function:

```
(def solver
  (fn [x]
    (... x ... ... ... ... x ... ...)))

(solver [1 2 3 4 5 6 7])
```

You can find my problems and solutions in solutions/just-enough-clojure.clj[17].

**Exercise 6**: Implement this function:

`(prefix-of? candidate sequence)`: Both arguments are sequences. Returns `true` if the elements in the `candidate` are the first elements in the `sequence`:

```
user> (prefix-of? [1 2] [1 2 3 4])
true
user> (prefix-of? '(2 3) [1 2 3 4])
false
user> (prefix-of? '(1 2) [1 2 3 4])
true
```

**Exercise 7**: Implement this function:

`(tails sequence)`: Returns a sequence of successively smaller subsequences of the argument.

```
user> (tails '(1 2 3 4))
((1 2 3 4) (2 3 4) (3 4) (4) ())
```

To implement `tails`, use `range`, which produces a sequence of integers. For example, `(range 4)` is `(0 1 2 3)`.

This one is tricky. My solution is very much in the functional style, in that it depends on sequences being easy to create and work with. So I'll provide some hints. Here and hereafter, I encourage you to try to finish without using the hints, but not to the point where you get frustrated. Programming is supposed to be fun.

*Hint*: What is the result of evaluating this?

[17] https://github.com/marick/fp-oo/blob/master/solutions/just-enough-clojure.clj

```
[(drop 0 [1 2 3])
 (drop 1 [1 2 3])
 (drop 2 [1 2 3])
 (drop 3 [1 2 3])]
```

*Hint*: `map` can take more than one sequence. If you give it two sequences, it passes the first of each to its function, then the second of each, and so on.

**Exercise 8**: In the first exercise in the chapter, I asked you to complete this function:

```
(def second (fn [list] ____))
```

Notice that `list` is a parameter to the function. We also know that `list` is (globally), a function in its own right. That raises an interesting question. What is the result of using the following function?

```
user=> (def puzzle (fn [list] (list list)))
user=> (puzzle '(1 2 3))
????
```

Why does that happen?

*Hint*: Use the substitution rule for functions.

# I Glossary

**apply**:
: To apply a function to some arguments is to substitute each argument for its formal parameter and then evaluate (execute) the function. I'll also write "invoke a function" or "call a function" when they read better. They all mean the same thing.

**argument**:
: In this book, I reserve "argument" for the actual values to which a function is applied. I use parameter for the symbols in a function definition's parameter list.

**atom**:
: In Clojure, an atom is a "container" for a value. The atom can be mutated to hold a different value (*not* to change the value within it). The change is made by fetching the current value, passing it to a function, and storing the function's return value. If two threads attempt to modify the atom at the same time, Clojure guarantees that one will complete before the other begins.

**binding**:
: A binding associates a symbol with a value.

**binding value**:
: In this book, used to contrast with monadic values. A monad accepts a monadic value, processes it, and then binds the resulting binding value to a symbol to make it available to later steps.

**class**:
: A class describes a collection of similar instances. It may describe the data those instances contain (by naming instance variables). It may also describe the methods that act on those instance variables.

**class method**:
: A class method is executed by sending a message to a class, rather than to an instance. In languages like Ruby and the embedded language of Part 1, classes *are* instances, so when you send a message to an instance that happens to be a class, you get an instance method of that class object, which we call a class method. That is, there's no *implementation* difference between `a-point.foo` and `Point.new`. See The Class as an Object chapter.

**classifier function**:
The classifier takes the arguments to a generic function and usually converts them into a small number of values that are used to select a specialized function.

**closure**:
A function that can be applied to arguments but that also has permanent access to all name/value bindings in its environment at *the moment of function creation*. As such, it can make use of named values defined “outside” itself, even after the names that refer to those values cease to do so.

**collecting parameter**:
In a recursive function, a collecting parameter is one that is passed a closer approximation to the final solution in each nested recursive call. See the explanation in the book.

**constructor**:
A constructor creates an instance based on the information in a class. The resulting instance is “of” that class.

**continuation**:
During a computation, the continuation is a description (in the form of a function) of the computation that remains to be done.

**continuation-passing style**:
Writing a computation as the calculation of one value that is then passed to a function that represents the continuation of the computation. See the description in the text.

**dataflow style**:
A programming style that emphasizes data flowing through a series of functions and being transformed at each stage.

**depth-first traversal**:
A tree traversal in which, if the traversal has a choice whether to go down first or right first, it chooses “down”.

**destructuring binding**:
When a sequence is passed as an argument, destructuring binding lets you bind parameter names to elements of the sequence without having to bind the whole sequence to a name and then pick it apart with code.

**dispatch function**:
When a name can refer to more than one function, the dispatch function decides which function to apply by examining the argument list.

**double dispatch**

A kludge required in conventional object-oriented programming, used when the correct behavior depends on both `this` and another object. See the discussion in the book.

**duck typing**:

A way of defining class relationships used in languages without static types. Inspired by the saying "If it walks like a duck and talks like a duck, it's a duck". When duck typing, you don't define one class as depending on another's type but rather on particular messages it responds to. It differs from (say) Java's interfaces in that the sets of messages aren't distinct named entities in the program, but rather implicit groups, one for each purpose.

**dynamic scoping**:

When a symbol is evaluated to find its bound value, the binding that's used is the one *most recently evaluated* during execution of the program. The position of the binding code in the program's text is irrelevant. Contrast to lexical scoping.

**eager evaluation**:

The opposite of lazy evaluation. Computation is performed immediately, rather than as values are demanded.

**encapsulation**:

Making the binding between a symbol and a value invisible to code outside a function or object boundary.

**environment**:

The environment collects all symbol/value bindings in effect at a particular moment.

**evaluator**:

An evaluator converts a data structure, usually obtained from the reader, into a value. See the explanation in the text. Clojure's evaluator is named `eval`.

**function**:

In general terms, a function is some executable code that is given arguments and produces a value. In Clojure, a function is specifically a closure.

**future**:

A future converts a computation into a value. A computation wrapped in a future executes on a different thread. If the value of the future is ever referenced, and the computation is not finished, the referencing thread is paused until the value is computed.

**generic function**:

In conventional object-oriented programming, the dispatch function looks only at the type of the object given as the implicit "this" argument. Generic functions provide a different strategy, in which the dispatch function is user-provided and can use any argument. In Clojure, generic functions are defined with `defmulti`.

Generic functions encourage a verb-centered way of thinking about the world: there are actions that can apply very broadly. The specifics of an action depends on some properties (determined at runtime) of the values it's applied to.

**global definition**:

In a global definition, a function is bound to a symbol using `def`. Such a function can be used by any other function in the namespace. Contrast with *local definition*.

**higher-order function**:

A function that either takes a function as an argument or produces a function as its return value.

**immutability**:

In Clojure, data structures cannot be modified once created. Within functions and `let` forms, the association of a symbol to a value cannot be changed once made (because there is no assignment statement in Clojure).

**instance**:

Synonymous with object, but emphasizes that the instance is one representative of a class (from which it is instantiated).

**instance method**:

The method applied in response to a message sent to an instance. Used when a distinction between instance methods and class methods is useful. More usually, an unqualified "method" is used.

**instance variable**:

The data an instance holds can be thought of as a collection of name/value pairs. "Instance variable" can refer to the name part or to both parts. For example, "initialize the instance variable to 5" associates a value with the name. In Clojure and other languages with immutable data, instance variables don't ever vary.

**instantiation**:

Creating an instance by allocating space, associating runtime-specific metadata with it, and then calling a class-specific function to initialize instance variables.

**keyword**:
A clojure datatype, written like `:my-keyword`. Keywords evaluate to themselves and are often used as the key in a map. Keywords are callables.

**lazy evaluation**:
In a fully lazy language, no computation is performed unless some other computation demands its results. In effect, evaluation is a "pull" process, where the need to print some output ripples "backward" to provoke only those computations that are needed. Clojure is not fully lazy, but it has the lazyseq data structure, which is.

**lazy initialization**:
In an object-oriented language, an instance variable is lazily initialized if its starting value is only calculated when some client code first asks for it.

**lazyseq**:
A Clojure sequence that uses lazy evaluation.

**lexical scoping**:
The most common sort of binding in modern programming languages. When there is more than one binding for a symbol, evaluating that symbol uses the closest enclosing binding in *the text of the program*. Nothing in the execution of the program can change which binding is used. Contrast to dynamic scoping.

**list**: A clojure sequence that has the property that it takes longer to access the last element than the first. Lists are used both to hold data and to represent Clojure programs.

**local definition**:
A function definition that is either used immediately (as in `( (partial + 1) 2)` and so has no name, or whose name is given in a `let` binding or a function's parameter list. Whereas a function with a global definition can be used by any function in the namespace, a local definition can be "seen" only within the body of its `let` or function definition.

**macro**:
A function that translates Clojure code into different clojure code. The transformed code is evaluated in the normal way. Macros are a way of inventing your own special forms.

**map**:
As a noun, an unordered collection of key/value pairs, like a Java HashMap or a Ruby Hash. Maps are callables.

As a verb, a function that applies a callable to each element of one or more collections. The return values are collected together and returned in a lazyseq.

**message**:

A message is the name of a method. When functions are used as methods, we use the metaphor that the program sends a message and arguments to an object.

**metaclass**:

A class that describes a class in the same way that a class describes an instance. Metaclasses store the methods invoked in response to a message sent to a class object.

**metadata**:

Data about data. An example in this book is the pointer from an instance to its class, which the dispatch function uses when deciding which method to apply.

**method**:

A method is a function with a (usually) implicit "this" or "self" argument that refers to an instance of a class. Metaphorically, the method is invoked when a message of the same name is sent to the instance.

**mock object**:

A mock object is used to test whether classes use their collaborating classes correctly. It stands in for one of the object-under-test's neighbor objects. The test programs the mock to expect the object-under-test to send it specific messages. If the mock object is not sent those messages, the test fails.

**module**:

In Ruby, a module is a class-like object that can be placed in the inheritance chain of a class

**monad**:

A set of functions that describes how to separate the steps of a computation from what happens between those steps.

**monad transformer**:

A function that takes one monad as its argument and produces another monad that has the properties of the argument monad plus different monad.

**monadic function**:

A function that takes a single binding value and converts it into a monadic value.

**monadic value**:

The type (or shape) of value that a monad operates on. A monad accepts monadic values, may or may not do something to them, and provides the results to a computational step as a binding value.

**multimethod**:
A synonym for generic function.

**multiple inheritance**:
In multiple inheritance, an object can have more than one direct superclass, so its ancestors could form a complicated graph (with classes appearing more than once) rather than a simple sequence.

**namespace**:
Namespaces are Clojure's equivalent of packages or modules in other languages: a way of restricting the visibility of names to other parts of a program. Roughly speaking, a namespace corresponds to a file.

For purposes of this book, a namespace is a map from symbols to values. (The reality is slightly more complicated.) There are functions that give one namespace access to values in another (by altering the client namespace's own map).

**object**:
Conventionally, encapsulated mutable state. Because of a class definition, certain methods can be applied to that state.

**ORM**:
Object-relational mapping. A library or framework that stores objects in a relational database and can reconstruct those objects later.

**override (a method)**:
In an object-oriented language, a method defined in a subclass overrides a method with the same name in a superclass. In that case, the dispatch function applied to an instance of the subclass will pick the subclass version.

**parameter**:
In a function definition, the parameter list is a vector of symbols. During function application, those symbols are replaced with the corresponding values in the argument list.

**partial application**:
Recasting a function of `n` arguments as one of `n-m` arguments, where the `m` arguments are replaced by constants. In Clojure, `(partial + 3)` produces a function that adds three to its argument. Often also called "currying", though that term is strictly incorrect.

**point-free definitions**:
Functions that are created without mentioning their parameters. Creation is done with higher-order-functions.

**polymorphic**:
When one name associated with potentially many functions (or methods). The dispatch function uses the argument list (perhaps including the receiver of a message) to decide which function to apply to the arguments.

**printer**:
The printer converts the internal representation of data to output strings. See the explanation in the text.

**reader**:
The reader converts text into Clojure's internal representation for data. See the explanation in the text.

**receiver**:
In the message/method metaphor, the receiver is the particular instance to which a method is applied.

**recursion**:
Traditionally, a book's definition of recursion reads "See recursion." Because I'm a humorless git, I'll point you to the appropriate section of this book.

**repl**: The read-eval-print loop. It reads a Clojure expression, evaluates it, and prints the result. Also used to refer to Clojure's interactive interpreter. See the explanation in the text.

**respond to a message**:
An object responds to a message if it has method with that name.

**rest arguments**:
When a functions function's parameter list contains an `&`, that signals that all remaining arguments should be collected into a sequence and associated with the parameter following the `&`. Those arguments are referred to as the "rest arguments".

**send a message**:
Sending a message is a stylized way of applying a function. The dispatch function uses the message, an instance, and an argument list to find the function. That function is applied to an argument list composed of the original argument list and the instance.

**seq**: Either a list or a lazyseq.

**sequence**:
An umbrella term referring to Clojure's list, vector, and seq data types. All sequences can be indexed by integers (starting with `0`).

**set**: A datatype in Clojure that acts much like a mathematical set. In particular it's easy to test membership in a set. A set is a callable. As such, it returns true iff its single argument is in the set.

**shadowing**:
When symbols can be defined to refer to values and a language allows such binding expressions to be nested, an enclosed definition shadows an enclosing one using the same name. In that case, evaluation of the symbol means the enclosed value.

In an object-oriented language, a method defined in a subclass shadows a method with the same name in a superclass. In that case, the dispatch function applied to an instance of the subclass will pick the subclass version.

**side effect**:
A "pure" function takes inputs, calculates a result, and does nothing else. A function with side effects can, during its calculation, change state in a way observable from outside the caller. For example, it may perform I/O. Or it may change the value of a global variable.

**signature**:
The name of a function (or method), together with its parameter list.

**software transactional memory**:
Controlling read and write access to memory in a way similar to the way databases control write access to tables.

**special symbol, special form**:
When a list is being used to represent code, certain symbols are treated specially by the evaluator. For example, `fn` heads a list used to create functions, and `quote` heads a list containing a value that should not be evaluated.

**specialized function**:
A specialized function is one that a generic function can dispatch to. In Clojure, a specialized function is defined by `defmethod`.

**state**:
Data that can be mutated, especially when the changes are made via side effects.

**structure sharing**:
Languages that have only immutable data structures seem to require much wasteful copying. In fact, both the new and old copies will share most of their structure. This is analogous to video compression, where frame N+1 is stored as only what changed to the frame N.

**substitution**, **substitution rule**:
In a pure functional language, evaluation of code can (modulo optimization) be accomplished by successive substitution of values. See the explanation in the text.

**symbol**:
A Clojure datatype that is typically used to refer to a value.

**syntactic sugar**:
Special syntax in a language to make common operations easier to write. Often disparaged by purists. "Syntactic sugar causes cancer of the semicolon."—Alan J. Perlis.

**unbound symbol**:
A symbol in an expression that is to be evaluated to yield a value—but no binding has been established for the symbol. (That is, it does not appear in an enclosing `let` or function parameter list.)

**value**:
In this book, I use "value" to refer to any piece of Clojure data, be it an integer, a list, a vector, or whatever.

**vector**:
A Clojure sequence that has the property that the last element is as fast to access as the first. Vectors are callables.

**zipper**:
A data structure that simulates random movement through, and editing of, immutable trees. They're explained in a chapter.