Clojure from the ground up

by Kyle Kingsbury

version 1.27.11

Table of content

Chapter 1: Welcome

This guide aims to introduce newcomers and experienced programmers alike to the beauty of functional programming, starting with the simplest building blocks of software. You'll need a computer, basic proficiency in the command line, a text editor, and an internet connection. By the end of this series, you'll have a thorough command of the Clojure programming language.

Who is this guide for?

Science, technology, engineering, and mathematics are deeply rewarding fields, yet few women enter STEM as a career path. Still more are discouraged by a culture which repeatedly asserts that women lack the analytic aptitude for writing software, that they are not driven enough to be successful scientists, that it's not cool to pursue a passion for structural engineering. Those few with the talent, encouragement, and persistence to break in to science and tech are discouraged by persistent sexism in practice: the old boy's club of tenure, being passed over for promotions, isolation from peers, and flat-out assault. This landscape sucks. I want to help change it.

Women Who Code, PyLadies, Black Girls Code, RailsBridge, Girls Who Code, Girl Develop It, and Lambda Ladies are just a few of the fantastic groups helping women enter and thrive in software. I wholeheartedly support these efforts.

In addition, I want to help in my little corner of the technical community–functional programming and distributed systems–by making high-quality educational resources available for free. The Jepsen series has been, in part, an effort to share my enthusiasm for distributed systems with beginners of all stripes–but especially for women, LGBT folks, and people of color.

As technical authors, we often assume that our readers are white, that our readers are straight, that our readers are traditionally male. This is the invisible default in US culture, and it's especially true in tech. People continue to assume on the basis of my software and writing that I'm straight, because well hey, it's a statistically reasonable assumption.

But I'm *not* straight. I get called faggot, cocksucker, and sinner. People say they'll pray for me. When I walk hand-in-hand with my boyfriend, people roll down their car windows and stare. They threaten to beat me up or kill me. Every day I'm aware that I'm the only gay person some people know, and that I can show that not all gay people are effeminate, or hypermasculine, or ditzy, or obsessed with image. That you can be a manicurist or a mathematician or both. Being different, being a stranger in your culture, comes with all kinds of challenges. I can't speak to everyone's experience, but I can take a pretty good guess.

At the same time, in the technical community I've found overwhelming warmth and support, from people of *all* stripes. My peers stand up for me every day, and I'm so thankful–especially you straight dudes–for understanding a bit of what it's like to be different. I want to extend that same understanding, that same empathy, to people unlike myself. Moreover, I want to reassure everyone that though they may feel different, they *do* have a place in this community.

So before we begin, I want to reinforce that you *can* program, that you *can* do math, that you *can* design car suspensions and fire suppression systems and spacecraft control software

and distributed databases, regardless of what your classmates and media and even fellow engineers think. You don't have to be white, you don't have to be straight, you don't have to be a man. You can grow up never having touched a computer and still become a skilled programmer. Yeah, it's harder–and yeah, people will give you shit, but that's not your fault and has nothing to do with your *ability* or your *right* to do what you love. All it takes to be a good engineer, scientist, or mathematician is your curiosity, your passion, the right teaching material, and putting in the hours.

There's nothing in this guide that's just for lesbian grandmas or just for mixed-race kids; bros, you're welcome here too. There's nothing dumbed down. We're gonna go as deep into the ideas of programming as I know how to go, and we're gonna do it with everyone on board.

No matter who you are or who people *think* you are, this guide is for you.

Why Clojure?

This book is about how to program. We'll be learning in Clojure, which is a modern dialect of a very old family of computer languages, called Lisp. You'll find that many of this book's ideas will translate readily to other languages; though they may be expressed in different ways.

We're going to explore the nature of syntax, metalanguages, values, references, mutation, control flow, and concurrency. Many languages leave these ideas implicit in the language construction, or don't have a concept of metalanguages or concurrency at all. Clojure makes these ideas explicit, first-class language constructs.

At the same time, we're going to defer or omit any serious discussion of static type analysis, hardware, and performance. This is not to say that these ideas aren't *important*; just that they don't fit well within this particular narrative arc. For a deep exploration of type theory I recommend a study in Haskell, and for a better understanding of underlying hardware, learning C and an assembly language will undoubtedly help.

In more general terms, Clojure is a well-rounded language. It offers broad library support and runs on multiple operating systems. Clojure performance is not terrific, but is orders of magnitude faster than Ruby, Python, or Javascript. Unlike some faster languages, Clojure emphasizes *safety* in its type system and approach to parallelism, making it easier to write correct multithreaded programs. Clojure is *concise*, requiring very little code to express complex operations. It offers a REPL and dynamic type system: ideal for beginners to experiment with, and well-suited for manipulating complex data structures. A consistently designed standard library and full-featured set of core datatypes rounds out the Clojure toolbox.

Finally, there are some drawbacks. As a compiled language, Clojure is much slower to start than a scripting language; this makes it unsuitable for writing small scripts for interactive use. Clojure is also *not* well-suited for high-performance numeric operations. Though it is possible, you have to jump through hoops to achieve performance comparable with Java. I'll do my best to call out these constraints and shortcomings as we proceed through the text.

With that context out of the way, let's get started by installing Clojure!

Getting set up

First, you'll need a Java Virtual Machine, or JVM, and its associated development tools, called the JDK. This is the software which *runs* a Clojure program. If you're on Windows, install Oracle JDK 1.7. If you're on OS X or Linux, you may already have a JDK installed. In a terminal, try:

```
which javac
```

If you see something like

```
/usr/bin/javac
```

Then you're good to go. If you don't see any output from that command, install the appropriate Oracle JDK 1.7 for your operating system, or whatever JDK your package manager has available.

When you have a JDK, you'll need Leiningen, the Clojure build tool. If you're on a Linux or OS X computer, the instructions below should get you going right away. If you're on Windows, see the Leiningen page for an installer. If you get stuck, you might want to start with a primer on command line basics.

```
mkdir -p ~/bin
cd ~/bin
curl -O https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein
chmod a+x lein
```

Leiningen automatically handles installing Clojure, finding libraries from the internet, and building and running your programs. We'll create a new Leiningen project to play around in:

```
cd
lein new scratch
```

This creates a new directory in your homedir, called `scratch`. If you see `command not found` instead, it means the directory `~/bin` isn't registered with your terminal as a place to search for programs. To fix this, add the line

```
export PATH="$PATH":~/bin
```

to the file `.bash_profile` in your home directory, then run `source ~/.bash_profile`. Re-running `lein new scratch` should work.

Let's enter that directory, and start using Clojure itself:

```
cd scratch
lein repl
```

The structure of programs

When you type `lein repl` at the terminal, you'll see something like this:

```
aphyr@waterhouse:~/scratch$ lein repl
nREPL server started on port 45413
REPL-y 0.2.0
Clojure 1.5.1
    Docs: (doc function-name-here)
          (find-doc "part-of-name-here")
  Source: (source function-name-here)
 Javadoc: (javadoc java-object-or-class-here)
```

```
    Exit: Control+D or (exit) or (quit)

user=>
```

This is an interactive Clojure environment called a REPL, for “Read, Evaluate, Print Loop”. It’s going to *read* a program we enter, run that program, and print the results. REPLs give you quick feedback, so they’re a great way to explore a program interactively, run tests, and prototype new ideas.

Let’s write a simple program. The simplest, in fact. Type “nil”, and hit enter.

```
user=> nil
nil
```

`nil` is the most basic value in Clojure. It represents emptiness, nothing-doing, not-a-thing. The absence of information.

```
user=> true
true
user=> false
false
```

`true` and `false` are a pair of special values called *Booleans*. They mean exactly what you think: whether a statement is true or false. `true`, `false`, and `nil` form the three poles of the Lisp logical system.

```
user=> 0
0
```

This is the number zero. Its numeric friends are `1`, `-47`, `1.2e-4`, `1/3`, and so on. We might also talk about *strings*, which are chunks of text surrounded by double quotes:

```
user=> "hi there!"
"hi there!"
```

`nil`, `true`, `0`, and `"hi there!"` are all different types of *values*; the nouns of programming. Just as one could say “House.” in English, we can write a program like `"hello, world"` and it evaluates to itself: the string `"hello world"`. But most sentences aren’t just about stating the existence of a thing; they involve *action*. We need *verbs*.

```
user=> inc
#<core$inc clojure.core$inc@6f7ef41c>
```

This is a verb called `inc`–short for “increment”. Specifically, `inc` is a *symbol* which *points* to a verb: `#<core$inc clojure.core$inc@6f7ef41c>`– just like the word “run” is a *name* for the *concept* of running.

There’s a key distinction here–that a signifier, a reference, a label, is not the same as the signified, the referent, the concept itself. If you write the word “run” on paper, the ink means nothing by itself. It’s just a symbol. But in the mind of a reader, that symbol takes on *meaning*; the idea of running.

Unlike the number 0, or the string “hi”, symbols are references to other values. when Clojure evaluates a symbol, it looks up that symbol’s meaning. Look up `inc`, and you get `#<core $inc clojure.core$inc@6f7ef41c>`.

Can we refer to the symbol itself, *without* looking up its meaning?

```
user=> 'inc
inc
```

Yes. The single quote `'` *escapes* a sentence. In programming languages, we call sentences `expressions` or `statements`. A quote says “Rather than *evaluating* this expression’s text, simply return the text itself, unchanged.” Quote a symbol, get a symbol. Quote a number, get a number. Quote anything, and get it back exactly as it came in.

```
user=> '123
123
user=> '"foo"
"foo"
user=> '(1 2 3)
(1 2 3)
```

A new kind of value, surrounded by parentheses: the *list*. LISP originally stood for LISt Processing, and lists are still at the core of the language. In fact, they form the most basic way to compose expressions, or sentences. A list is a single expression which has *multiple parts*. For instance, this list contains three elements: the numbers 1, 2, and 3. Lists can contain anything: numbers, strings, even other lists:

```
user=> '(nil "hi")
(nil "hi")
```

A list containing two elements: the number 1, and a second list. That list contains two elements: the number 2, and another list. *That* list contains two elements: 3, and an empty list.

```
user=> '(1 (2 (3 ())))
(1 (2 (3 ())))
```

You could think of this structure as a tree–which is a provocative idea, because *languages* are like trees too: sentences are comprised of clauses, which can be nested, and each clause may have subjects modified by adjectives, and verbs modified by adverbs, and so on. “Lindsay, my best friend, took the dog which we found together at the pound on fourth street, for a walk with her mother Michelle.”

```
Took
  Lindsay
    my best friend
  the dog
    which we found together
      at the pound
        on fourth street
    for a walk
      with her mother
        Michelle
```

But let’s try something simpler. Something we know how to talk about. “Increment the number zero.” As a tree:

```
Increment
  the number zero
```

We have a symbol for incrementing, and we know how to write the number zero. Let's combine them in a list:

```
clj=> '(inc 0)
(inc 0)
```

A basic sentence. Remember, since it's quoted, we're talking about the tree, the text, the expression, by itself. Absent interpretation. If we remove the single-quote, Clojure will *interpret* the expression:

```
user=> (inc 0)
1
```

Incrementing zero yields one. And if we wanted to increment *that* value?

```
Increment
  increment
    the number zero
```

```
user=> (inc (inc 0))
2
```

A sentence in Lisp is a list. It starts with a verb, and is followed by zero or more objects for that verb to act on. Each part of the list can *itself* be another list, in which case that nested list is evaluated first, just like a nested clause in a sentence. When we type

```
(inc (inc 0))
```

Clojure first looks up the meanings for the symbols in the code:

```
(#<core$inc clojure.core$inc@6f7ef41c>
  (#<core$inc clojure.core$inc@6f7ef41c>
    0))
```

Then evaluates the innermost list `(inc 0)`, which becomes the number 1:

```
(#<core$inc clojure.core$inc@6f7ef41c>
 1)
```

Finally, it evaluates the outer list, incrementing the number 1:

```
2
```

Every list starts with a verb. Parts of a list are evaluated from left to right. Innermost lists are evaluated before outer lists.

```
(+ 1 (- 5 2) (+ 3 4))
(+ 1 3       (+ 3 4))
(+ 1 3       7)
11
```

That's it.

The entire grammar of Lisp: the structure for every expression in the language. We transform expressions by *substituting* meanings for symbols, and obtain some result. This is the core of the Lambda Calculus, and it is the theoretical basis for almost all computer languages. Ruby, Javascript, C, Haskell; all languages express the text of their programs in different ways, but internally all construct a *tree* of expressions. Lisp simply makes it explicit.

Review

We started by learning a few basic nouns: numbers like `5`, strings like `"cat"`, and symbols like `inc` and `+`. We saw how quoting makes the difference between an *expression* itself and the thing it *evaluates* to. We discovered symbols as *names* for other values, just like how words represent concepts in any other language. Finally, we combined lists to make trees, and used those trees to represent a program.

With these basic elements of syntax in place, it's time to expand our vocabulary with new verbs and nouns; learning to represent more complex values and transform them in different ways.

Chapter 2: Basic types

We've learned the basics of Clojure's syntax and evaluation model. Now we'll take a tour of the basic nouns in the language.

Types

We've seen a few different values already–for instance, `nil`, `true`, `false`, `1`, `2.34`, and `"meow"`. Clearly all these things are *different* values, but some of them seem more alike than others.

For instance, `1` and `2` are *very* similar numbers; both can be added, divided, multiplied, and subtracted. `2.34` is also a number, and acts very much like 1 and 2, but it's not quite the same. It's got *decimal* points. It's not an *integer*. And clearly `true` is *not* very much like a number. What is true plus one? Or false divided by 5.3? These questions are poorly defined.

We say that a *type* is a group of values which work in the same way. It's a *property* that some values share, which allows us to organize the world into sets of similar things. 1 + 1 and 1 + 2 use *the same addition*, which adds together integers. Types also help us *verify* that a program makes sense: that you can only add together numbers, instead of adding numbers to porcupines.

Types can overlap and intersect each other. Cats are animals, and cats are fuzzy too. You could say that a cat is a *member* (or sometimes "instance"), of the fuzzy and animal types. But there are fuzzy things like moss which *aren't* animals, and animals like alligators that aren't fuzzy in the slightest.

Other types completely subsume one another. All tabbies are housecats, and all housecats are felidae, and all felidae are animals. Everything which is true of an animal is automatically true of a housecat. Hierarchical types make it easier to write programs which don't need to know all the specifics of every value; and conversely, to create new types in terms of others. But they can also get in the way of the programmer, because not every useful classification (like "fuzziness") is purely hierarchical. Expressing overlapping types in a hierarchy can be tricky.

Every language has a *type system*; a particular way of organizing nouns into types, figuring out which verbs make sense on which types, and relating types to one another. Some languages are strict, and others more relaxed. Some emphasize hierarchy, and others a more ad-hoc view of the world. We call Clojure's type system *strong* in that operations on improper types are simply not allowed: the program will explode if asked to subtract a dandelion. We also say that Clojure's types are *dynamic* because they are enforced when the program is run, instead of when the program is first read by the computer.

We'll learn more about the formal relationships between types later, but for now, keep this in the back of your head. It'll start to hook in to other concepts later.

Integers

Let's find the type of the number 3:

```
user=> (type 3)
java.lang.Long
```

So 3 is a `java.lang.Long`, or a “Long”, for short. Because Clojure is built on top of Java, many of its types are plain old Java types.

Longs, internally, are represented as a group of sixty-four binary digits (ones and zeroes), written down in a particular pattern called signed two’s complement representation. You don’t need to worry about the specifics–there are only two things to remember about longs. First, longs use one bit to store the sign: whether the number is positive or negative. Second, the other 63 bits represent the *size* of the number. That means the biggest number you can represent with a long is 2^{63} - 1 (the minus one is because of the number 0), and the smallest long is -2^{63}.

How big is 2^{63} - 1?

```
user=> Long/MAX_VALUE
9223372036854775807
```

That’s a reasonably big number. Most of the time, you won’t need anything bigger, but… what if you did? What happens if you add one to the biggest Long?

```
user=> (inc Long/MAX_VALUE)

ArithmeticException integer overflow
  clojure.lang.Numbers.throwIntOverflow (Numbers.java:1388)
```

An error occurs! This is Clojure telling us that something went wrong. The type of error was an `ArithmeticException`, and its message was “integer overflow”, meaning “this type of number can’t hold a number that big”. The error came from a specific *place* in the source code of the program: `Numbers.java`, on line 1388. That’s a part of the Clojure source code. Later, we’ll learn more about how to unravel error messages and find out what went wrong.

The important thing is that Clojure’s type system *protected* us from doing something dangerous; instead of returning a corrupt value, it aborted evaluation and returned an error.

If you *do* need to talk about really big numbers, you can use a BigInt: an arbitrary-precision integer. Let’s convert the biggest Long into a BigInt, then increment it:

```
user=> (inc (bigint Long/MAX_VALUE))
9223372036854775808N
```

Notice the N at the end? That’s how Clojure writes arbitrary-precision integers.

```
user=> (type 5N)
clojure.lang.BigInt
```

There are also smaller numbers.

```
user=> (type (int 0))
java.lang.Integer
user=> (type (short 0))
java.lang.Short
user=> (type (byte 0))
```

```
java.lang.Byte
```

Integers are half the size of Longs; they store values in 32 bits. Shorts are 16 bits, and Bytes are 8. That means their biggest values are 2^{31}-1, 2^{15}-1, and 2^{7}-1, respectively.

```
user=> Integer/MAX_VALUE
2147483647
user=> Short/MAX_VALUE
32767
user=> Byte/MAX_VALUE
127
```

Fractional numbers

To represent numbers *between* integers, we often use floating-point numbers, which can represent small numbers with fine precision, and large numbers with coarse precision. Floats use 32 bits, and Doubles use 64. Doubles are the default in Clojure.

```
user=> (type 1.23)
java.lang.Double
user=> (type (float 1.23))
java.lang.Float
```

Floating point math is complicated, and we won't get bogged down in the details just yet. The important thing to know is floats and doubles are *approximations*. There are limits to their correctness:

```
user=> 0.99999999999999999
1.0
```

To represent fractions exactly, we can use the *ratio* type:

```
user=> (type 1/3)
clojure.lang.Ratio
```

Mathematical operations

The exact behavior of mathematical operations in Clojure depends on their types. In general, though, Clojure aims to *preserve* information. Adding two longs returns a long; adding a double and a long returns a double.

```
user=> (+ 1 2)
3
user=> (+ 1 2.0)
3.0
```

`3` and `3.0` are *not* the same number; one is a long, and the other a double. But for most purposes, they're equivalent, and Clojure will tell you so:

```
user=> (= 3 3.0)
false
user=> (== 3 3.0)
true
```

`=` asks whether all the things that follow are equal. Since floats are approximations, `=` considers them different from integers. `==` also compares things, but a little more loosely: it considers integers equivalent to their floating-point representations.

We can also subtract with `-`, multiply with `*`, and divide with `/`.

```
user=> (- 3 1)
2
user=> (* 1.5 3)
4.5
user=> (/ 1 2)
1/2
```

Putting the verb *first* in each list allows us to add or multiply more than one number in the same step:

```
user=> (+ 1 2 3)
6
user=> (* 2 3 1/5)
6/5
```

Subtraction with more than 2 numbers subtracts all later numbers from the first. Division divides the first number by all the rest.

```
user=> (- 5 1 1 1)
2
user=> (/ 24 2 3)
4
```

By extension, we can define useful interpretations for numeric operations with just a *single* number:

```
user=> (+ 2)
2
user=> (- 2)
-2
user=> (* 4)
4
user=> (/ 4)
1/4
```

We can also add or multiply a list of no numbers at all, obtaining the additive and multiplicative identities, respectively. This might seem odd, especially coming from other languages, but we'll see later that these generalizations make it easier to reason about higher-level numeric operations.

```
user=> (+)
0
user=> (*)
1
```

Often, we want to ask which number is bigger, or if one number falls between two others. `<=` means "less than or equal to", and asserts that all following values are in order from smallest to biggest.

```
user=> (<= 1 2 3)
true
user=> (<= 1 3 2)
```

```
false
```

`<` means “strictly less than”, and works just like `<=`, except that no two values may be equal.

```
user=> (<= 1 1 2)
true
user=> (< 1 1 2)
false
```

Their friends `>` and `>=` mean “greater than” and “greater than or equal to”, respectively, and assert that numbers are in descending order.

```
user=> (> 3 2 1)
true
user=> (> 1 2 3)
false
```

Also commonly used are `inc` and `dec`, which add and subtract one to a number, respectively:

```
user=> (inc 5)
6
user=> (dec 5)
4
```

One final note: equality tests can take more than 2 numbers as well.

```
user=> (= 2 2 2)
true
user=> (= 2 2 3)
false
```

Strings

We saw that strings are text, surrounded by double quotes, like `"foo"`. Strings in Clojure are, like Longs, Doubles, and company, backed by a Java type:

```
user=> (type "cat")
java.lang.String
```

We can make almost *anything* into a string with `str`. Strings, symbols, numbers, booleans; every value in Clojure has a string representation. Note that `nil`’s string representation is `""`; an empty string.

```
user=> (str "cat")
"cat"
user=> (str 'cat)
"cat"
user=> (str 1)
"1"
user=> (str true)
"true"
user=> (str '(1 2 3))
"(1 2 3)"
user=> (str nil)
""
```

`str` can also *combine* things together into a single string, which we call “concatenation”.

```
user=> (str "meow " 3 " times")
```

```
"meow 3 times"
```

To look for patterns in text, we can use a regular expression, which is a tiny language for describing particular arrangements of text. `re-find` and `re-matches` look for occurrences of a regular expression in a string. To find a cat:

```
user=> (re-find #"cat" "mystic cat mouse")
"cat"
user=> (re-find #"cat" "only dogs here")
nil
```

That `#"..."` is Clojure's way of writing a regular expression.

With `re-matches`, you can extract particular parts of a string which match an expression. Here we find two strings, separated by a `:`. The parentheses mean that the regular expression should *capture* that part of the match. We get back a list containing the part of the string that matched the first parentheses, followed by the part that matched the second parentheses.

```
user=> (rest (re-matches #"(.+):(.+)" "mouse:treat"))
("mouse" "treat")
```

Regular expressions are a powerful tool for searching and matching text, especially when working with data files. Since regexes work the same in most languages, you can use any guide online to learn more. It's not something you have to master right away; just learn specific tricks as you find you need them. For a deeper guide, try Fitzgerald's Introducing Regular Expressions.

Booleans and logic

Everything in Clojure has a sort of charge, a truth value, sometimes called "truthiness". `true` is positive and `false` is negative. `nil` is negative, too.

```
user=> (boolean true)
true
user=> (boolean false)
false
user=> (boolean nil)
false
```

Every other value in Clojure is positive.

```
user=> (boolean 0)
true
user=> (boolean 1)
true
user=> (boolean "hi there")
true
user=> (boolean str)
true
```

If you're coming from a C-inspired language, where 0 is considered false, this might be a bit surprising. Likewise, in much of POSIX, 0 is considered success and nonzero values are failures. Lisp allows no such confusion: the only negative values are `false` and `nil`.

We can reason about truth values using `and`, `or`, and `not`. `and` returns the first negative value, or the last value if all are truthy.

```
user=> (and true false true)
false
user=> (and true true true)
true
user=> (and 1 2 3)
3
```

Similarly, `or` returns the first positive value.

```
user=> (or false 2 3)
2
user=> (or false nil)
nil
```

And `not` inverts the logical sense of a value:

```
user=> (not 2)
false
user=> (not nil)
true
```

We'll learn more about Boolean logic when we start talking about *control flow*; the way we alter evaluation of a program and express ideas like "if I'm a cat, then meow incessantly".

Symbols

We saw symbols in the previous chapter; they're bare strings of characters, like `foo` or `+`.

```
user=> (class 'str)
clojure.lang.Symbol
```

Symbols can have either short or full names. The short name is used to refer to things locally. The *fully qualified* name is used to refer unambiguously to a symbol from anywhere. If I were a symbol, my name would be "Kyle", and my full name "Kyle Kingsbury."

Symbol names are separated with a `/`. For instance, the symbol `str` is also present in a family called `clojure.core`; the corresponding full name is `clojure.core/str`.

```
user=> (= str clojure.core/str)
true
user=> (name 'clojure.core/str)
"str"
```

When we talked about the maximum size of an integer, that was a fully-qualified symbol, too.

```
(type 'Integer/MAX_VALUE)
clojure.lang.Symbol
```

The job of symbols is to *refer* to things, to *point* to other values. When evaluating a program, symbols are looked up and replaced by their corresponding values. That's not the only use of symbols, but it's the most common.

Keywords

Closely related to symbols and strings are *keywords*, which begin with a `:`. Keywords are like strings in that they're made up of text, but are specifically intended for use as *labels* or

identifiers. These *aren't* labels in the sense of symbols: keywords aren't replaced by any other value. They're just names, by themselves.

```
user=> (type :cat)
clojure.lang.Keyword
user=> (str :cat)
":cat"
user=> (name :cat)
"cat"
```

As labels, keywords are most useful when paired with other values in a collection, like a *map*. Keywords can also be used as verbs to *look up specific values* in other data types. We'll learn more about keywords shortly.

Lists

A collection is a group of values. It's a *container* which provides some structure, some framework, for the things that it holds. We say that a collection contains *elements*, or *members*. We saw one kind of collection–a list–in the previous chapter.

```
user=> '(1 2 3)
(1 2 3)
user=> (type '(1 2 3))
clojure.lang.PersistentList
```

Remember, we *quote* lists with a `'` to prevent them from being evaluated. You can also construct a list using `list`:

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

Lists are comparable just like every other value:

```
user=> (= (list 1 2) (list 1 2))
true
```

You can modify a list by `conj`oining an element onto it:

```
user=> (conj '(1 2 3) 4)
(4 1 2 3)
```

We added 4 to the list–but it appeared at the *front*. Why? Internally, lists are stored as a *chain* of values: each link in the chain is a tiny box which holds the value and a connection to the next link. This data structure, called a linked list, offers immediate access to the first element.

```
user=> (first (list 1 2 3))
1
```

But getting to the second element requires an extra hop down the chain

```
user=> (second (list 1 2 3))
2
```

and the third element a hop after that, and so on.

```
user=> (nth (list 1 2 3) 2)
```

```
3
```

`nth` gets the element of an ordered collection at a particular *index*. The first element is index 0, the second is index 1, and so on.

This means that lists are well-suited for small collections, or collections which are read in linear order, but are slow when you want to get arbitrary elements from later in the list. For fast access to every element, we use a *vector*.

Vectors

Vectors are surrounded by square brackets, just like lists are surrounded by parentheses. Because vectors *aren't* evaluated like lists are, there's no need to quote them:

```
user=> [1 2 3]
[1 2 3]
user=> (type [1 2 3])
clojure.lang.PersistentVector
```

You can also create vectors with `vector`, or change other structures into vectors with `vec`:

```
user=> (vector 1 2 3)
[1 2 3]
user=> (vec (list 1 2 3))
[1 2 3]
```

`conj` on a vector adds to the *end*, not the *start*:

```
user=> (conj [1 2 3] 4)
[1 2 3 4]
```

Our friends `first`, `second`, and `nth` work here too; but unlike lists, `nth` is *fast* on vectors. That's because internally, vectors are represented as a very broad tree of elements, where each part of the tree branches into 32 smaller trees. Even very large vectors are only a few layers deep, which means getting to elements only takes a few hops.

In addition to `first`, you'll often want to get the *remaining* elements in a collection. There are two ways to do this:

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

`rest` and `next` both return "everything but the first element". They differ only by what happens when there are no remaining elements:

```
user=> (rest [1])
()
user=> (next [1])
nil
```

`rest` returns logical true, `next` returns logical false. Each has their uses, but in almost every case they're equivalent–I interchange them freely.

We can get the final element of any collection with `last`:

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

And figure out how big the vector is with `count`:

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

Because vectors are intended for looking up elements by index, we can also use them directly as *verbs*:

```
user=> ([:a :b :c] 1)
:b
```

So we took the vector containing three keywords, and asked “What’s the element at index 1?” Lisp, like most (but not all!) modern languages, counts up from *zero*, not one. Index 0 is the first element, index 1 is the second element, and so on. In this vector, finding the element at index 1 evaluates to `:b`.

Finally, note that vectors and lists containing the same elements are considered equal in Clojure:

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

In almost all contexts, you can consider vectors, lists, and other sequences as interchangeable. They only differ in their performance characteristics, and in a few data-structure-specific operations.

Sets

Sometimes you want an unordered collection of values; especially when you plan to ask questions like “does the collection have the number 3 in it?” Clojure, like most languages, calls these collections *sets*.

```
user=> #{:a :b :c}
#{:a :c :b}
```

Sets are surrounded by `#{...}`. Notice that though we gave the elements `:a`, `:b`, and `:c`, they came out in a different order. In general, the order of sets can shift at any time. If you want a particular order, you can ask for it as a list or vector:

```
user=> (vec #{:a :b :c})
[:a :c :b]
```

Or ask for the elements in sorted order:

```
(sort #{:a :b :c})
(:a :b :c)
```

`conj` on a set adds an element:

```
user=> (conj #{:a :b :c} :d)
#{:a :c :b :d}
user=> (conj #{:a :b :c} :a)
```

```
#{:a :c :b}
```

Sets never contain an element more than once, so `conj`ing an element which is already present does nothing. Conversely, one removes elements with `disj`:

```
user=> (disj #{"hornet" "hummingbird"} "hummingbird")
#{"hornet"}
```

The most common operation with a set is to check whether something is inside it. For this we use `contains?`.

```
user=> (contains? #{1 2 3} 3)
true
user=> (contains? #{1 2 3} 5)
false
```

Like vectors, you can use the set *itself* as a verb. Unlike `contains?`, this expression returns the element itself (if it was present), or `nil`.

```
user=> (#{1 2 3} 3)
3
user=> (#{1 2 3} 4)
nil
```

You can make a set out of any other collection with `set`.

```
user=> (set [:a :b :c])
#{:a :c :b}
```

Maps

The last collection on our tour is the *map*: a data structure which associates *keys* with *values*. In a dictionary, the keys are words and the definitions are the values. In a library, keys are call signs, and the books are values. Maps are indexes for looking things up, and for representing different pieces of named information together. Here's a cat:

```
user=> {:name "mittens" :weight 9 :color "black"}
{:weight 9, :name "mittens", :color "black"}
```

Maps are surrounded by braces `{...}`, filled by alternating keys and values. In this map, the three keys are `:name`, `:color`, and `:weight`, and their values are `"mittens"`, `"black"`, and 9, respectively. We can look up the corresponding value for a key with `get`:

```
user=> (get {"cat" "meow" "dog" "woof"} "cat")
"meow"
user=> (get {:a 1 :b 2} :c)
nil
```

`get` can also take a *default* value to return instead of nil, if the key doesn't exist in that map.

```
user=> (get {:glinda :good} :wicked :not-here)
:not-here
```

Since lookups are so important for maps, we can use a map as a verb directly:

```
user=> ({"amlodipine" 12 "ibuprofen" 50} "ibuprofen")
```

```
50
```

And conversely, keywords can *also* be used as verbs, which look themselves up in maps:

```
user=> (:raccoon {:weasel "queen" :raccoon "king"})
"king"
```

You can add a value for a given key to a map with `assoc`.

```
user=> (assoc {:bolts 1088} :camshafts 3)
{:camshafts 3 :bolts 1088}
user=> (assoc {:camshafts 3} :camshafts 2)
{:camshafts 2}
```

Assoc adds keys if they aren't present, and *replaces* values if they're already there. If you associate a value onto `nil`, it creates a new map.

```
user=> (assoc nil 5 2)
{5 2}
```

You can combine maps together using `merge`, which yields a map containing all the elements of *all* given maps, preferring the values from later ones.

```
user=> (merge {:a 1 :b 2} {:b 3 :c 4})
{:c 4, :a 1, :b 3}
```

Finally, to remove a value, use `dissoc`.

```
user=> (dissoc {:potatoes 5 :mushrooms 2} :mushrooms)
{:potatoes 5}
```

Putting it all together

All these collections and types can be combined freely. As software engineers, we model the world by creating a particular *representation* of the problem in the program. Having a rich set of values at our disposal allows us to talk about complex problems. We might describe a person:

```
{:name "Amelia Earhart"
 :birth 1897
 :death 1939
 :awards {"US"    #{"Distinguished Flying Cross" "National Women's Hall of Fame"}
          "World" #{"Altitude record for Autogyro" "First to cross Atlantic twice"}}}
```

Or a recipe:

```
{:title "Chocolate chip cookies"
 :ingredients {"flour"           [(+ 2 1/4) :cup]
               "baking soda"     [1   :teaspoon]
               "salt"            [1   :teaspoon]
               "butter"          [1   :cup]
               "sugar"           [3/4 :cup]
               "brown sugar"     [3/4 :cup]
               "vanilla"         [1   :teaspoon]
               "eggs"            2
               "chocolate chips" [12  :ounce]}}
```

Or the Gini coefficients of nations, as measured over time:

```
{"Afghanistan" {2008 27.8}
```

```
 "Indonesia"   {2008 34.1 2010 35.6 2011 38.1}
 "Uruguay"     {2008 46.3 2009 46.3 2010 45.3}}
```

In Clojure, we *compose* data structures to form more complex values; to talk about bigger ideas. We use operations like `first`, `nth`, `get`, and `contains?` to extract specific information from these structures, and modify them using `conj`, `disj`, `assoc`, `dissoc`, and so on.

We started this chapter with a discussion of *types*: groups of similar objects which obey the same rules. We learned that bigints, longs, ints, shorts, and bytes are all integers, that doubles and floats are approximations to decimal numbers, and that ratios represent fractions exactly. We learned the differences between strings for text, symbols as references, and keywords as short labels. Finally, we learned how to compose, alter, and inspect collections of elements. Armed with the basic nouns of Clojure, we're ready to write a broad array of programs.

I'd like to conclude this tour with one last type of value. We've inspected dozens of types so far– but what what happens when you turn the camera on itself?

```
user=> (type type)
clojure.core$type
```

What *is* this `type` thing, exactly? What *are* these verbs we've been learning, and where do they come from? This is the central question of chapter three: functions.

Chapter 3: Functions

We left off last chapter with a question: what *are* verbs, anyway? When you evaluate `(type :mary-poppins)`, what really happens?

```
user=> (type :mary-poppins)
clojure.lang.Keyword
```

To understand how `type` works, we'll need several new ideas. First, we'll expand on the notion of symbols as references to other values. Then we'll learn about functions: Clojure's verbs. Finally, we'll use the Var system to explore and change the definitions of those functions.

Let bindings

We know that symbols are names for things, and that when evaluated, Clojure replaces those symbols with their corresponding values. `+`, for instance, is a symbol which points to the verb `#<core$_PLUS_ clojure.core$_PLUS_@12992c>`.

```
user=> +
#<core$_PLUS_ clojure.core$_PLUS_@12992c>
```

When you try to use a symbol which has no defined meaning, Clojure refuses:

```
user=> cats

CompilerException java.lang.RuntimeException: Unable to resolve symbol: cats in this context, co
```

But we can define a meaning for a symbol within a specific expression, using `let`.

```
user=> (let [cats 5] (str "I have " cats " cats."))
"I have 5 cats."
```

The `let` expression first takes a vector of *bindings*: alternating symbols and values that those symbols are *bound* to, within the remainder of the expression. "Let the symbol `cats` be 5, and construct a string composed of `"I have "`, `cats`, and `" cats"`.

Let bindings apply only within the let expression itself. They also override any existing definitions for symbols at that point in the program. For instance, we can redefine addition to mean subtraction, for the duration of a `let`:

```
user=> (let [+ -] (+ 2 3))
-1
```

But that definition doesn't apply outside the let:

```
user=> (+ 2 3)
5
```

We can also provide *multiple* bindings. Since Clojure doesn't care about spacing, alignment, or newlines, I'll write this on multiple lines for clarity.

```
user=> (let [person   "joseph"
             num-cats 186]
         (str person " has " num-cats " cats!"))
```

```
"joseph has 186 cats!"
```

When multiple bindings are given, they are evaluated in order. Later bindings can use previous bindings.

```
user=> (let [cats 3
             legs (* 4 cats)]
         (str legs " legs all together"))
"12 legs all together"
```

So fundamentally, `let` defines the meaning of symbols within an expression. When Clojure evaluates a `let`, it replaces all occurrences of those symbols in the rest of the `let` expression with their corresponding values, then evaluates the rest of the expression.

Functions

We saw in chapter one that Clojure evaluates lists by *substituting* some other value in their place:

```
user=> (inc 1)
2
```

`inc` takes any number, and is replaced by that number plus one. That sounds an awful lot like a let:

```
user=> (let [x 1] (+ x 1))
2
```

If we bound `x` to `5` instead of `1`, this expression would evaluate to `6`. We can think about `inc` like a let expression, but without particular values provided for the symbols.

```
(let [x] (+ x 1))
```

We can't actually evaluate this program, because there's no value for `x` yet. It could be 1, or 4, or 1453. We say that `x` is *unbound*, because it has no binding to a particular value. This is the nature of the *function*: an expression with unbound symbols.

```
user=> (fn [x] (+ x 1))
#<user$eval293$fn__294 user$eval293$fn__294@663fc37>
```

Does the name of that function remind you of anything?

```
user=> inc
#<core$inc clojure.core$inc@16bc0b3c>
```

Almost all verbs in Clojure are functions. Functions represent unrealized computation: expressions which are not yet evaluated, or incomplete. This particular function works just like `inc`: it's an expression which has a single unbound symbol, `x`. When we *invoke* the function with a particular value, the expressions in the function are evaluated with `x` bound to that value.

```
user=> (inc 2)
3
user=> ((fn [x] (+ x 1)) 2)
3
```

We say that `x` is this functions *argument*, or *parameter*. When Clojure evaluates `(inc 2)`, we say that `inc` is *called* with `2`, or that `2` is *passed* to `inc`. The result of that *function invocation* is the function's *return value*. We say that `(inc 2)` *returns* `3`.

Fundamentally, functions describe the relationship between arguments and return values: given `1`, return `2`. Given `2`, return `3`, and so on. Let bindings describe a similar relationship, but with a specific set of values for those arguments. `let` is evaluated immediately, whereas `fn` is evaluated *later*, when bindings are provided.

There's a shorthand for writing functions, too: `#(+ % 1)` is equivalent to `(fn [x] (+ x 1))`. `%` takes the place of the first argument to the function. You'll sometime see `%1`, `%2`, etc. used for the first argument, second argument, and so on.

```
user=> (let [burrito #(list "beans" % "cheese")]
         (burrito "carnitas"))
("beans" "carnitas" "cheese")
```

Since functions exist to *defer* evaluation, there's no sense in creating and invoking them in the same expression as we've done here. What we want is to give *names* to our functions, so they can be recombined in different ways.

```
user=> (let [twice (fn [x] (* 2 x))]
         (+ (twice 1)
            (twice 3)))
8
```

Compare that expression to an equivalent, expanded form:

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

The name `twice` is gone, and in its place is the same sort of computation–`(* 2 something)`–written twice. While we *could* represent our programs as a single massive expression, it'd be impossible to reason about. Instead, we use functions to compact redundant expressions, by isolating common patterns of computation. Symbols help us re-use those functions (and other values) in more than one place. By giving the symbols meaningful names, we make it easier to reason about the structure of the program as a whole; breaking it up into smaller, understandable parts.

This is core pursuit of software engineering: organizing expressions. Almost every programming language is in search of the right tools to break apart, name, and recombine expressions to solve large problems. In Clojure we'll see one particular set of tools for composing programs, but the underlying ideas will transfer to many other languages.

Vars

We've used `let` to define a symbol within an expression, but what about the default meanings of `+`, `conj`, and `type`? Are they also `let` bindings? Is the whole universe one giant `let`?

Well, not exactly. That's one way to think about default bindings, but it's brittle. We'd need to wrap our whole program in a new `let` expression every time we wanted to change the meaning of a symbol. And moreover, once a `let` is defined, there's no way to change it. If

we want to redefine symbols for *everyone*–even code that we didn't write–we need a new construct: a *mutable* variable.

```
user=> (def cats 5)
#'user/cats
user=> (type #'user/cats)
clojure.lang.Var
```

`def` *defines* a type of value we haven't seen before: a var. Vars, like symbols, are references to other values. When evaluated, a symbol pointing to a var is replaced by the var's corresponding value:

```
user=> user/cats
5
```

`def` also *binds* the symbol `cats` (and its globally qualified equivalent `user/cats`) to that var.

```
user=> user/cats
5
user=> cats
5
```

When we said in chapter one that `inc`, `list`, and friends were symbols that pointed to functions, that wasn't the whole story. The symbol `inc` points to the var `#'inc`, which in turn points to the function `#<core$inc clojure.core$inc@16bc0b3c>`. We can see the intermediate var with `resolve`:

```
user=> 'inc
inc ; the symbol
user=> (resolve 'inc)
#'clojure.core/inc ; the var
user=> (eval 'inc)
#<core$inc clojure.core$inc@16bc0b3c> ; the value
```

Why two layers of indirection? Because unlike the symbol, we can *change* the meaning of a Var for everyone, globally, at any time.

```
user=> (def astronauts [])
#'user/astronauts
user=> (count astronauts)
0
user=> (def astronauts ["Sally Ride" "Guy Bluford"])
#'user/astronauts
user=> (count astronauts)
2
```

Notice that `astronauts` had *two* distinct meanings, depending on *when* we evaluated it. After the first `def`, astronauts was an empty vector. After the second `def`, it had one entry.

If this seems dangerous, you're a smart cookie. Redefining names in this way changes the meaning of expressions *everywhere* in a program, without warning. Expressions which relied on the value of a Var could suddenly take on new, possibly incorrect, meanings. It's a powerful tool for experimenting at the REPL, and for updating a running program, but it can have unexpected consequences. Good Clojurists use `def` to set up a program initially, and only change those definitions with careful thought.

Totally redefining a Var isn't the only option. There are safer, controlled ways to change the meaning of a Var within a particular part of a program, which we'll explore later.

Defining functions

Armed with *def*, we're ready to create our own named functions in Clojure.

```
user=> (def half (fn [number] (/ number 2)))
#'user/half
user=> (half 6)
3
```

Creating a function and binding it to a var is so common that it has its own form: `defn`, short for `def fn`.

```
user=> (defn half [number] (/ number 2))
#'user/half
```

Functions don't have to take an argument. We've seen functions which take zero arguments, like `(+)`.

```
user=> (defn half [] 1/2)
#'user/half
user=> (half)
1/2
```

But if we try to use our earlier form with one argument, Clojure complains that the *arity*–the number of arguments to the function–is incorrect.

```
user=> (half 10)

ArityException Wrong number of args (1) passed to: user$half
  clojure.lang.AFn.throwArity (AFn.java:437)
```

To handle *multiple* arities, functions have an alternate form. Instead of an argument vector and a body, one provides a series of lists, each of which starts with an argument vector, followed by the body.

```
user=> (defn half
         ([]  1/2)
         ([x] (/ x 2)))
user=> (half)
1/2
user=> (half 10)
5
```

Multiple arguments work just like you expect. Just specify an argument vector of two, or three, or however many arguments the function takes.

```
user=> (defn add
         [x y]
         (+ x y))
#'user/add
user=> (add 1 2)
3
```

Some functions can take *any* number of arguments. For that, Clojure provides `&`, which slurps up all remaining arguments as a list:

```
user=> (defn vargs
         [x y & more-args]
         {:x    x
          :y    y
          :more more-args})
#'user/vargs
user=> (vargs 1)

ArityException Wrong number of args (1) passed to: user$vargs
  clojure.lang.AFn.throwArity (AFn.java:437)
user=> (vargs 1 2)
{:x 1, :y 2, :more nil}
user=> (vargs 1 2 3 4 5)
{:x 1, :y 2, :more (3 4 5)}
```

Note that `x` and `y` are mandatory, though there don't have to be any remaining arguments.

To keep track of what arguments a function takes, why the function exists, and what it does, we usually include a *docstring*. Docstrings help fill in the missing context around functions, to explain their assumptions, context, and purpose to the world.

```
(defn launch
  "Launches a spacecraft into the given orbit by initiating a
   controlled on-axis burn. Does not automatically stage, but
   does vector thrust, if the craft supports it."
  [craft target-orbit]
  "OK, we don't know how to control spacecraft yet.")
```

Docstrings are used to automatically generate documentation for Clojure programs, but you can also access them from the REPL.

```
user=> (doc launch)
-------------------------
user/launch
([craft target-orbit])
   Launches a spacecraft into the given orbit by initiating a
   controlled on-axis burn. Does not automatically stage, but
   does vector thrust, if the craft supports it.
nil
```

`doc` tells us the full name of the function, the arguments it accepts, and its docstring. This information comes from the `#'launch` var's *metadata*, and is saved there by `defn`. We can inspect metadata directly with the `meta` function:

```
(meta #'launch)
{:arglists ([craft target-
orbit]), :ns #<Namespace user>, :name launch!, :column 1, :doc "Launches a spacecraft into the
 given orbit.", :line 1, :file "NO_SOURCE_PATH"}
```

There's some other juicy information in there, like the file the function was defined in and which line and column it started at, but that's not particularly useful since we're in the REPL, not a file. However, this *does* hint at a way to answer our motivating question: how does the `type` function work?

How does type work?

We know that `type` returns the type of an object:

```
user=> (type 2)
java.lang.long
```

And that `type`, like all functions, is a kind of object with its own unique type:

```
user=> type
#<core$type clojure.core$type@39bda9b9>
user=> (type type)
clojure.core$type
```

This tells us that `type` is a particular *instance*, at memory address `39bda9b9`, of the type `clojure.core$type`. `clojure.core` is a namespace which defines the fundamentals of the Clojure language, and `$type` tells us that it's named `type` in that namespace. None of this is particularly helpful, though. Maybe we can find out more about the `clojure.core$type` by asking what its *supertypes* are:

```
user=> (supers (type type))
#{clojure.lang.AFunction clojure.lang.IMeta java.util.concurrent.Callable clojure.lang.Fn clojur
```

This is a set of all the types that include `type`. We say that `type` is an *instance* of `clojure.lang.AFunction`, or that it *implements* or *extends* `java.util.concurrent.Callable`, and so on. Since it's a member of `clojure.lang.IMeta` it has metadata, and since it's a member of clojure.lang.AFn, it's a function. Just to double check, let's confirm that `type` is indeed a function:

```
user=> (fn? type)
true
```

What about its documentation?

```
user=> (doc type)
-------------------------
clojure.core/type
([x])
  Returns the :type metadata of x, or its Class if none
nil
```

Ah, that's helpful. `type` can take a single argument, which it calls `x`. If it has `:type` metadata, that's what it returns. Otherwise, it returns the class of `x`. Let's take a deeper look at `type`'s metadata for more clues.

```
user=> (meta #'type)
{:ns #<Namespace clojure.core>, :name type, :arglists ([x]), :column 1, :added "1.0", :static tr     :do
 the :type metadata of x, or its Class if none", :line 3109, :file "clojure/core.clj"}
```

Look at that! This function was first added to Clojure in version 1.0, and is defined in the file `clojure/core.clj`, on line 3109. We could go dig up the Clojure source code and read its definition there–or we could ask Clojure to do it for us:

```
user=> (source type)
(defn type
  "Returns the :type metadata of x, or its Class if none"
  {:added "1.0"
   :static true}
  [x]
```

```
  (or (get (meta x) :type) (class x)))
nil
```

Aha! Here, at last, is how `type` works. It's a function which takes a single argument `x`, and returns either `:type` from its metadata, or `(class x)`.

We can delve into any function in Clojure using these tools:

```
user=> (source +)
(defn +
  "Returns the sum of nums. (+) returns 0. Does not auto-promote
  longs, will throw on overflow. See also: +'"
  {:inline (nary-inline 'add 'unchecked_add)
   :inline-arities >1?
   :added "1.2"}
  ([] 0)
  ([x] (cast Number x))
  ([x y] (. clojure.lang.Numbers (add x y)))
  ([x y & more]
     (reduce1 + (+ x y) more)))
nil
```

Almost every function in a programming language is made up of other, simpler functions. `+`, for instance, is defined in terms of `cast`, `add`, and `reduce1`. Sometimes functions are defined in terms of themselves. `+` uses itself twice in this definition; a technique called *recursion*.

At the bottom, though, are certain fundamental constructs below which you can go no further. Core axioms of the language. Lisp calls these "special forms". `def` and `let` are special forms (well–almost: `let` is a thin wrapper around `let*`, which is a special form) in Clojure. These forms are defined by the core implementation of the language, and are not reducible to other Clojure expressions.

```
user=> (source def)
Source not found
```

Some Lisps are written *entirely* in terms of a few special forms, but Clojure is much less pure. Many functions bottom out in Java functions and types, or, for CLJS, in terms of Javascript. Any time you see an expression like `(. clojure.lang.Numbers (add x y))`, there's Java code underneath. Below Java lies the JVM, which might be written in C or C++, depending on which one you use. And underneath C and C++ lie more libraries, the operating system, assembler, microcode, registers, and ultimately, electrons flowing through silicon.

A well-designed language *isolates* you from details you don't need to worry about, like which logic gates or registers to use, and lets you focus on the task at hand. Good languages also need to allow escape hatches for performance or access to dangerous functionality, as we saw with Vars. You can write entire programs entirely in terms of Clojure, but sometimes, for performance or to use tools from other languages, you'll rely on Java. The Clojure code is easy to explore with `doc` and `source`, but Java can be more opaque–I usually rely on the java source files and online documentation.

Review

We've seen how `let` associates names with values in a particular expression, and how Vars allow for *mutable* bindings which apply universally. and whose definitions can change over time.

We learned that Clojure verbs are functions, which express the general shape of an expression but with certain values *unbound*. Invoking a function *binds* those variables to specific values, allowing evaluation of the function to proceed.

Functions *decompose* programs into simpler pieces, expressed in terms of one another. Short, meaningful names help us understand what those functions (and other values) mean.

Finally, we learned how to introspect Clojure functions with `doc` and `source`, and saw the definition of some basic Clojure functions. The Clojure cheatsheet gives a comprehensive list of the core functions in the language, and is a great starting point when you have to solve a problem but don't know what functions to use.

We'll see a broad swath of those functions in Chapter 4: Sequences.

My thanks to Zach Tellman, Kelly Sommers, and Michael R Bernstein for reviewing drafts of this chapter.

Chapter 4: Sequences

In Chapter 3, we discovered functions as a way to *abstract* expressions; to rephrase a particular computation with some parts missing. We used functions to transform a single value. But what if we want to apply a function to *more than one* value at once? What about sequences?

For example, we know that `(inc 2)` increments the number 2. What if we wanted to increment *every number* in the vector `[1 2 3]`, producing `[2 3 4]`?

```
user=> (inc [1 2 3])
ClassCastException clojure.lang.PersistentVector cannot be cast to java.lang.Number
  clojure.lang.Numbers.inc (Numbers.java:110)
```

Clearly `inc` can only work on numbers, not on vectors. We need a different kind of tool.

A direct approach

Let's think about the problem in concrete terms. We want to increment each of three elements: the first, second, and third. We know how to get an element from a sequence by using nth, so let's start with the first number, at index 0:

```
user=> (def numbers [1 2 3])
#'user/numbers
user=> (nth numbers 0)
1
user=> (inc (nth numbers 0))
2
```

So there's the first element incremented. Now we can do the second:

```
user=> (inc (nth numbers 1))
3
user=> (inc (nth numbers 2))
4
```

And it should be straightforward to combine these into a vector…

```
user=> [(inc (nth numbers 0)) (inc (nth numbers 1)) (inc (nth numbers 2))]
[2 3 4]
```

Success! We've incremented each of the numbers in the list! How about a list with only two elements?

```
user=> (def numbers [1 2])
#'user/numbers
user=> [(inc (nth numbers 0)) (inc (nth numbers 1)) (inc (nth numbers 2))]

IndexOutOfBoundsException   clojure.lang.PersistentVector.arrayFor (PersistentVector.java:107)
```

Shoot. We tried to get the element at index 2, but *couldn't*, because `numbers` only has indices 0 and 1. Clojure calls that "index out of bounds".

We could just leave off the third expression in the vector; taking only elements 0 and 1. But the problem actually gets much worse, because we'd need to make this change *every* time we

wanted to use a different sized vector. And what of a vector with 1000 elements? We'd need 1000 `(inc (nth numbers ...))` expressions! Down this path lies madness.

Let's back up a bit, and try a slightly smaller problem.

Recursion

What if we just incremented the *first* number in the vector? How would that work? We know that `first` finds the first element in a sequence, and `rest` finds all the remaining ones.

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

So there's the *pieces* we'd need. To glue them back together, we can use a function called `cons`, which says "make a list beginning with the first argument, followed by all the elements in the second argument".

```
user=> (cons 1 [2])
(1 2)
user=> (cons 1 [2 3])
(1 2 3)
user=> (cons 1 [2 3 4])
(1 2 3 4)
```

OK so we can split up a sequence, increment the first part, and join them back together. Not so hard, right?

```
(defn inc-first [nums]
  (cons (inc (first nums))
        (rest nums)))
user=> (inc-first [1 2 3 4])
(2 2 3 4)
```

Hey, there we go! First element changed. Will it work with any length list?

```
user=> (inc-first [5])
(6)
user=> (inc-first [])

NullPointerException   clojure.lang.Numbers.ops (Numbers.java:942)
```

Shoot. We can't increment the first element of this empty vector, because it doesn't *have* a first element.

```
user=> (first [])
nil
user=> (inc nil)

NullPointerException   clojure.lang.Numbers.ops (Numbers.java:942)
```

So there are really *two* cases for this function. If there is a first element in `nums`, we'll increment it as normal. If there's *no* such element, we'll return an empty list. To express this kind of conditional behavior, we'll use a Clojure special form called `if`:

```
user=> (doc if)
-------------------------
```

```
if
  (if test then else?)
Special Form
  Evaluates test. If not the singular values nil or false,
  evaluates and yields then, otherwise, evaluates and yields else. If
  else is not supplied it defaults to nil.

  Please see http://clojure.org/special_forms#if
```

To confirm our intuition:

```
user=> (if true :a :b)
:a
user=> (if false :a :b)
:b
```

Seems straightforward enough.

```
(defn inc-first [nums]
  (if (first nums)
    ; If there's a first number, build a new list with cons
    (cons (inc (first nums))
          (rest nums))
    ; If there's no first number, just return an empty list
    (list)))

user=> (inc-first [])
()
user=> (inc-first [1 2 3])
(2 2 3)
```

Success! Now we can handle *both* cases: empty sequences, and sequences with things in them. Now how about incrementing that *second* number? Let's stare at that code for a bit.

```
(rest nums)
```

Hang on. That list– `(rest nums)` –that's a list of numbers too. What if we… used our inc-first function on *that* list, to increment *its* first number? Then we'd have incremented both the first *and* the second element.

```
(defn inc-more [nums]
  (if (first nums)
    (cons (inc (first nums))
          (inc-more (rest nums)))
    (list)))
user=> (inc-more [1 2 3 4])
(2 3 4 5)
```

Odd. That didn't just increment the first two numbers. It incremented *all* the numbers. We fell into the *complete* solution entirely by accident. What happened here?

Well first we… yes, we got the number one, and incremented it. Then we stuck that onto `(inc-first [2 3 4])`, which got two, and incremented it. Then we stuck that two onto `(inc-first [3 4])`, which got three, and then we did the same for four. Only *that* time around, at the very end of the list, `(rest [4])` would have been *empty*. So when we went to get the first number of the empty list, we took the *second* branch of the `if`, and returned the empty list.

Having reached the *bottom* of the function calls, so to speak, we zip back up the chain. We can imagine this function turning into a long string of `cons` calls, like so:

```
(cons 2 (cons 3 (cons 4 (cons 5 '()))))
(cons 2 (cons 3 (cons 4 '(5))))
(cons 2 (cons 3 '(4 5)))
(cons 2 '(3 4 5))
'(2 3 4 5)
```

This technique is called *recursion*, and it is a fundamental principle in working with collections, sequences, trees, or graphs… any problem which has small parts linked together. There are two key elements in a recursive program:

1. Some part of the problem which has a known solution
2. A relationship which connects one part of the problem to the next

Incrementing the elements of an empty list returns the empty list. This is our *base case*: the ground to build on. Our *inductive* case, also called the *recurrence relation*, is how we broke the problem up into incrementing the *first* number in the sequence, and incrementing all the numbers in the *rest* of the sequence. The `if` expression bound these two cases together into a single function; a function *defined in terms of itself*.

Once the initial step has been taken, *every* step can be taken.

```
user=> (inc-more [1 2 3 4 5 6 7 8 9 10 11 12])
(2 3 4 5 6 7 8 9 10 11 12 13)
```

This is the beauty of a recursive function; folding an unbounded stream of computation over and over, onto itself, until only a single step remains.

Generalizing from inc

We set out to increment every number in a vector, but nothing in our solution actually depended on `inc`. It just as well could have been `dec`, or `str`, or `keyword`. Let's *parameterize* our `inc-more` function to use *any* transformation of its elements:

```
(defn transform-all [f xs]
  (if (first xs)
    (cons (f (first xs))
          (transform-all f (rest xs)))
    (list)))
```

Because we could be talking about *any* kind of sequence, not just numbers, we've named the sequence `xs`, and its first element `x`. We also take a function `f` as an argument, and that function will be applied to each `x` in turn. So not only can we increment numbers…

```
user=> (transform-all inc [1 2 3 4])
(2 3 4 5)
```

…but we can turn strings to keywords…

```
user=> (transform-all keyword ["bell" "hooks"])
(:bell :hooks)
```

…or wrap every element in a list:

```
user=> (transform-all list [:codex :book :manuscript])
```

```
((:codex) (:book) (:manuscript))
```

In short, this function expresses a sequence in which each element is some function applied to the corresponding element in the underlying sequence. This idea is so important that it has its own name, in mathematics, Clojure, and other languages. We call it `map`.

```
user=> (map inc [1 2 3 4])
(2 3 4 5)
```

You might remember maps as a datatype in Clojure, too–they're dictionaries that relate keys to values.

```
{:year  1969
 :event "moon landing"}
```

The *function* `map` relates one sequence to another. The `type` map relates keys to values. There is a deep symmetry between the two: maps are usually sparse, and the relationships between keys and values may be arbitrarily complex. The map function, on the other hand, usually expresses the *same* type of relationship, applied to a series of elements in *fixed order*.

Building sequences

Recursion can do more than just `map`. We can use it to expand a single value into a sequence of values, each related by some function. For instance:

```
(defn expand [f x count]
  (if (pos? count)
    (cons x (expand f (f x) (dec count)))))
```

Our base case is `x` itself, followed by the sequence beginning with `(f x)`. That sequence in turn expands to `(f (f x))`, and then `(f (f (f x)))`, and so on. Each time we call `expand`, we count down by one using `dec`. Once the count is zero, the `if` returns `nil`, and evaluation stops. If we start with the number 0 and use inc as our function:

```
user=> user=> (expand inc 0 10)
(0 1 2 3 4 5 6 7 8 9)
```

Clojure has a more general form of this function, called `iterate`.

```
user=> (take 10 (iterate inc 0))
(0 1 2 3 4 5 6 7 8 9)
```

Since this sequence is *infinitely* long, we're using `take` to select only the first 10 elements. We can construct more complex sequences by using more complex functions:

```
user=> (take 10 (iterate (fn [x] (if (odd? x) (+ 1 x) (/ x 2))) 10))
(10 5 6 3 4 2 1 2 1 2)
```

Or build up strings:

```
user=> (take 5 (iterate (fn [x] (str x "o")) "y"))
("y" "yo" "yoo" "yooo" "yoooo")
```

`iterate` is extremely handy for working with infinite sequences, and has some partners in crime. `repeat`, for instance, constructs a sequence where every element is the same.

```
user=> (take 10 (repeat :hi))
(:hi :hi :hi :hi :hi :hi :hi :hi :hi :hi)
user=> (repeat 3 :echo)
(:echo :echo :echo)
```

And its close relative `repeatedly` simply calls a function `(f)` to generate an infinite sequence of values, over and over again, without any relationship between elements. For an infinite sequence of random numbers:

```
user=> (rand)
0.9002678382322784
user=> (rand)
0.12375594203332863
user=> (take 3 (repeatedly rand))
(0.44442397843046755 0.33668691162169784 0.18244875487846746)
```

Notice that calling `(rand)` returns a different number each time. We say that `rand` is an *impure* function, because it cannot simply be replaced by the same value every time. It does something different each time it's called.

There's another very handy sequence function specifically for numbers: `range`, which generates a sequence of numbers between two points. `(range n)` gives n successive integers starting at 0. `(range n m)` returns integers from n to m-1. `(range n m step)` returns integers from n to m, but separated by `step`.

```
user=> (range 5)
(0 1 2 3 4)
user=> (range 2 10)
(2 3 4 5 6 7 8 9)
user=> (range 0 100 5)
(0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95)
```

To extend a sequence by repeating it forever, use `cycle`:

```
user=> (take 10 (cycle [1 2 3]))
(1 2 3 1 2 3 1 2 3 1)
```

Transforming sequences

Given a sequence, we often want to find a *related* sequence. `map`, for instance, applies a function to each element–but has a few more tricks up its sleeve.

```
user=> (map (fn [n vehicle] (str "I've got " n " " vehicle "s"))
         [0 200 9]
         ["car" "train" "kiteboard"])
("I've got 0 cars" "I've got 200 trains" "I've got 9 kiteboards")
```

If given multiple sequences, `map` calls its function with one element from each sequence in turn. So the first value will be `(f 0 "car")`, the second `(f 200 "train")`, and so on. Like a zipper, map folds together corresponding elements from multiple collections. To sum three vectors, column-wise:

```
user=> (map + [1 2 3]
              [4 5 6]
```

```
              [7 8 9])
(12 15 18)
```

If one sequence is bigger than another, map stops at the end of the smaller one. We can exploit this to combine finite and infinite sequences. For example, to number the elements in a vector:

```
user=> (map (fn [index element] (str index ". " element))
            (iterate inc 0)
            ["erlang" "ruby" "haskell"])
("0. erlang" "1. ruby" "2. haskell")
```

Transforming elements together with their indices is so common that Clojure has a special function for it: `map-indexed`:

```
user=> (map-indexed (fn [index element] (str index ". " element))
                    ["erlang" "ruby" "haskell"])
("0. erlang" "1. ruby" "2. haskell")
```

You can also tack one sequence onto the end of another, like so:

```
user=> (concat [1 2 3] [:a :b :c] [4 5 6])
(1 2 3 :a :b :c 4 5 6)
```

Another way to combine two sequences is to riffle them together, using `interleave`.

```
user=> (interleave [:a :b :c] [1 2 3])
(:a 1 :b 2 :c 3)
```

And if you want to insert a specific element between each successive pair in a sequence, try `interpose`:

```
user=> (interpose :and [1 2 3 4])
(1 :and 2 :and 3 :and 4)
```

To reverse a sequence, use `reverse`.

```
user=> (reverse [1 2 3])
(3 2 1)
user=> (reverse "woolf")
(\f \l \o \o \w)
```

Strings are sequences too! Each element of a string is a *character*, written `\f`. You can rejoin those characters into a string with `apply str`:

```
user=> (apply str (reverse "woolf"))
"floow"
```

…and break strings up into sequences of chars with `seq`.

```
user=> (seq "sato")
(\s \a \t \o)
```

To randomize the order of a sequence, use `shuffle`.

```
user=> (shuffle [1 2 3 4])
[3 1 2 4]
user=> (apply str (shuffle (seq "abracadabra")))
"acaadabrrab"
```

Subsequences

We've already seen `take`, which selects the first n elements. There's also `drop`, which removes the first n elements.

```
user=> (range 10)
(0 1 2 3 4 5 6 7 8 9)
user=> (take 3 (range 10))
(0 1 2)
user=> (drop 3 (range 10))
(3 4 5 6 7 8 9)
```

And for slicing apart the other end of the sequence, we have `take-last` and `drop-last`:

```
user=> (take-last 3 (range 10))
(7 8 9)
user=> (drop-last 3 (range 10))
(0 1 2 3 4 5 6)
```

`take-while` and `drop-while` work just like `take` and `drop`, but use a function to decide when to cut.

```
user=> (take-while pos? [3 2 1 0 -1 -2 10])
(3 2 1)
```

In general, one can cut a sequence in twain by using `split-at`, and giving it a particular index. There's also `split-with`, which uses a function to decide when to cut.

```
(split-at 4 (range 10))
[(0 1 2 3) (4 5 6 7 8 9)]
user=> (split-with number? [1 2 3 :mark 4 5 6 :mark 7])
[(1 2 3) (:mark 4 5 6 :mark 7)]
```

Notice that because indexes start at zero, sequence functions tend to have predictable numbers of elements. `(split-at 4)` yields *four* elements in the first collection, and ensures the second collection *begins at index four*. `(range 10)` has ten elements, corresponding to the first ten indices in a sequence. `(range 3 5)` has two (since 5 - 3 is two) elements. These choices simplify the definition of recursive functions as well.

We can select particular elements from a sequence by applying a function. To find all positive numbers in a list, use `filter`:

```
user=> (filter pos? [1 5 -4 -7 3 0])
(1 5 3)
```

`filter` looks at each element in turn, and includes it in the resulting sequence *only* if `(f element)` returns a truthy value. Its complement is `remove`, which only includes those elements where `(f element)` is `false` or `nil`.

```
user=> (remove string? [1 "turing" :apple])
(1 :apple)
```

Finally, one can group a sequence into chunks using `partition`, `partition-all`, or `partition-by`. For instance, one might group alternating values into pairs:

```
user=> (partition 2 [:cats 5 :bats 27 :crocodiles 0])
((:cats 5) (:bats 27) (:crocodiles 0))
```

Or separate a series of numbers into negative and positive runs:

```
(user=> (partition-by neg? [1 2 3 2 1 -1 -2 -3 -2 -1 1 2])
((1 2 3 2 1) (-1 -2 -3 -2 -1) (1 2))
```

Collapsing sequences

After transforming a sequence, we often want to collapse it in some way; to derive some smaller value. For instance, we might want the number of times each element appears in a sequence:

```
user=> (frequencies [:meow :mrrrow :meow :meow])
{:meow 3, :mrrrow 1}
```

Or to group elements by some function:

```
user=> (pprint (group-by :first [{:first "Li"    :last "Zhou"}
                                 {:first "Sarah" :last "Lee"}
                                 {:first "Sarah" :last "Dunn"}
                                 {:first "Li"    :last "O'Toole"}]))
{"Li"    [{:last "Zhou", :first "Li"}   {:last "O'Toole", :first "Li"}],
 "Sarah" [{:last "Lee", :first "Sarah"} {:last "Dunn", :first "Sarah"}]}
```

Here we've taken a sequence of people with first and last names, and used the `:first` keyword (which can act as a function!) to look up those first names. `group-by` used that function to produce a *map* of first names to lists of people–kind of like an index.

In general, we want to *combine* elements together in some way, using a function. Where `map` treated each element independently, reducing a sequence requires that we bring some information along. The most general way to collapse a sequence is `reduce`.

```
user=> (doc reduce)
-------------------------
clojure.core/reduce
([f coll] [f val coll])
  f should be a function of 2 arguments. If val is not supplied,
  returns the result of applying f to the first 2 items in coll, then
  applying f to that result and the 3rd item, etc. If coll contains no
  items, f must accept no arguments as well, and reduce returns the
  result of calling f with no arguments.  If coll has only 1 item, it
  is returned and f is not called.  If val is supplied, returns the
  result of applying f to val and the first item in coll, then
  applying f to that result and the 2nd item, etc. If coll contains no
  items, returns val and f is not called.
```

That's a little complicated, so we'll start small. We need a function, `f`, which combines successive elements of the sequence. `(f state element)` will return the state for the *next* invocation of `f`. As `f` moves along the sequence, it carries some changing state with it. The final state is the return value of `reduce`.

```
user=> (reduce + [1 2 3 4])
10
```

`reduce` begins by calling `(+ 1 2)`, which yields the state `3`. Then it calls `(+ 3 3)`, which yields `6`. Then `(+ 6 4)`, which returns `10`. We've taken a function over *two* elements, and used it to combine *all* the elements. Mathematically, we could write:

```
1 + 2 + 3 + 4
    3 + 3 + 4
```

```
        6 + 4
           10
```

So another way to look at `reduce` is like sticking a function *between* each pair of elements. To see the reducing process in action, we can use `reductions`, which returns a sequence of all the intermediate states.

```
user=> (reductions + [1 2 3 4])
(1 3 6 10)
```

Oftentimes we include a *default* state to start with. For instance, we could start with an empty set, and add each element to it as we go along:

```
user=> (reduce conj #{} [:a :b :b :b :a :a])
#{:a :b}
```

Reducing elements into a collection has its own name: `into`. We can conj `[key value]` vectors into a map, for instance, or build up a list:

```
user=> (into {} [[:a 2] [:b 3]])
{:a 2, :b 3}
user=> (into (list) [1 2 3 4])
(4 3 2 1)
```

Because elements added to a list appear at the *beginning*, not the end, this expression reverses the sequence. Vectors `conj` onto the end, so to emit the elements in order, using `reduce`, we might try:

```
user=> (reduce conj [] [1 2 3 4 5])
(reduce conj [] [1 2 3 4 5])
[1 2 3 4 5]
```

Which brings up an interesting thought: this looks an awful lot like `map`. All that's missing is some kind of transformation applied to each element.

```
(defn my-map [f coll]
  (reduce (fn [output element]
            (conj output (f element)))
          []
          coll))
user=> (my-map inc [1 2 3 4])
[2 3 4 5]
```

Huh. `map` is just a special kind of `reduce`. What about, say, `take-while`?

```
(defn my-take-while [f coll]
  (reduce (fn [out elem]
            (if (f elem)
              (conj out elem)
              (reduced out)))
          []
          coll))
```

We're using a special function here, `reduced`, to indicate that we've completed our reduction *early* and can skip the rest of the sequence.

```
user=> (my-take-while pos? [2 1 0 -1 0 1 2])
[2 1]
```

`reduce` really is the uberfunction over sequences. Almost any operation on a sequence can be expressed in terms of a reduce–though for various reasons, many of the Clojure sequence functions are not written this way. For instance, `take-while` is *actually* defined like so:

```
user=> (source take-while)
(defn take-while
  "Returns a lazy sequence of successive items from coll while
  (pred item) returns true. pred must be free of side-effects."
  {:added "1.0"
   :static true}
  [pred coll]
  (lazy-seq
   (when-let [s (seq coll)]
       (when (pred (first s))
         (cons (first s) (take-while pred (rest s)))))))
```

There's a few new pieces here, but the structure is *essentially* the same as our initial attempt at writing `map`. When the predicate matches the first element, cons the first element onto `take-while`, applied to the rest of the sequence. That `lazy-seq` construct allows Clojure to compute this sequence *as required*, instead of right away. It defers execution to a later time.

Most of Clojure's sequence functions are lazy. They don't do anything until needed. For instance, we can increment every number from zero to infinity:

```
user=> (def infseq (map inc (iterate inc 0)))
#'user/infseq
user=> (realized? infseq)
false
```

That function returned *immediately*. Because it hasn't done any work yet, we say the sequence is *unrealized*. It doesn't increment any numbers at all until we ask for them:

```
user=> (take 10 infseq)
(1 2 3 4 5 6 7 8 9 10)
user=> (realized? infseq)
true
```

Lazy sequences also *remember* their contents, once evaluated, for faster access.

Putting it all together

We've seen how recursion generalizes a function over *one* thing into a function over *many* things, and discovered a rich landscape of recursive functions over sequences. Now let's use our knowledge of sequences to solve a more complex problem: find the sum of the products of consecutive pairs of the first 1000 odd integers.

First, we'll need the integers. We can start with 0, and work our way up to infinity. To save time printing an infinite number of integers, we'll start with just the first 10.

```
user=> (take 10 (iterate inc 0))
(0 1 2 3 4 5 6 7 8 9)
```

Now we need to find only the ones which are odd. Remember, `filter` pares down a sequence to only those elements which pass a test.

```
user=> (take 10 (filter odd? (iterate inc 0)))
```

```
(1 3 5 7 9 11 13 15 17 19)
```

For consecutive pairs, we want to take `[1 3 5 7 ...]` and find a sequence like `([1 3] [3 5] [5 7] ...)`. That sounds like a job for `partition`:

```
user=> (take 3 (partition 2 (filter odd? (iterate inc 0))))
((1 3) (5 7) (9 11))
```

Not quite right–this gave us non-overlapping pairs, but we wanted overlapping ones too. A quick check of `(doc partition)` reveals the `step` parameter:

```
user=> (take 3 (partition 2 1 (filter odd? (iterate inc 0))))
((1 3) (3 5) (5 7))
```

Now we need to find the product for each pair. Given a pair, multiply the two pieces together… yes, that sounds like `map`:

```
user=> (take 3 (map (fn [pair] (* (first pair) (second pair)))
                    (partition 2 1 (filter odd? (iterate inc 0)))))
(3 15 35)
```

Getting a bit unwieldy, isn't it? Only one final step: sum all those products. We'll adjust the `take` to include the first 1000, not the first 3, elements.

```
user=> (reduce +
               (take 1000
                     (map (fn [pair] (* (first pair) (second pair)))
                          (partition 2 1
                                     (filter odd?
                                             (iterate inc 0))))))
1335333000
```

The sum of the first thousand products of consecutive pairs of the odd integers starting at 0. See how each part leads to the next? This expression looks a lot like the way we phrased the problem in English–but both English and Lisp expressions are sort of backwards, in a way. The part that *happens first* appears *deepest*, *last*, in the expression. In a chain of reasoning like this, it'd be nicer to write it in order.

```
user=> (->> 0
            (iterate inc)
            (filter odd?)
            (partition 2 1)
            (map (fn [pair]
                   (* (first pair) (second pair))))
            (take 1000)
            (reduce +))
1335333000
```

Much easier to read: now everything flows in order, from top to bottom, and we've flattened out the deeply nested expressions into a single level. This is how object-oriented languages structure their expressions: as a chain of function invocations, each acting on the previous value.

But how is this possible? Which expression gets evaluated first? `(take 1000)` isn't even a valid call–where's its second argument? How are *any* of these forms evaluated?

What kind of arcane function *is* `->>`?

All these mysteries, and more, in Chapter 5: Macros.

Problems

1. Write a function to find out if a string is a palindrome–that is, if it looks the same forwards and backwards.
2. Find the number of ‘c’s in “abracadabra”.
3. Write your own version of `filter`.
4. Find the first 100 prime numbers: 2, 3, 5, 7, 11, 13, 17, ….

Chapter 5: Macros

In Chapter 1, I asserted that the grammar of Lisp is uniform: every expression is a list, beginning with a verb, and followed by some arguments. Evaluation proceeds from left to right, and every element of the list must be evaluated *before* evaluating the list itself. Yet we just saw, at the end of Sequences, an expression which seemed to *violate* these rules.

Clearly, this is not the whole story.

Macroexpansion

There is another phase to evaluating an expression; one which takes place before the rules we've followed so far. That process is called *macro-expansion*. During macro-expansion, the *code itself* is restructured according to some set of rules–rules which you, the programmer, can define.

```
(defmacro ignore
  "Cancels the evaluation of an expression, returning nil instead."
  [expr]
  nil)
user=> (ignore (+ 1 2))
nil
```

`defmacro` looks a lot like `defn`: it has a name, an optional documentation string, an argument vector, and a body–in this case, just `nil`. In this case, it looks like it simply ignored the expr `(+ 1 2)` and returned `nil`–but it's actually deeper than that. `(+ 1 2)` was *never evaluated at all*.

```
user=> (def x 1)
#'user/x
user=> x
1
user=> (ignore (def x 2))
nil
user=> x
1
```

`def` should have defined `x` to be `2` *no matter what*–but that never happened. At macroexpansion time, the expression `(ignore (+ 1 2))` was *replaced* by the expression `nil`, which was then evaluated to `nil`. Where functions rewrite *values*, macros rewrite *code*.

To see these different layers in play, let's try a macro which reverses the order of arguments to a function.

```
(defmacro rev [fun & args]
  (cons fun (reverse args)))
```

This macro, named `rev`, takes one mandatory argument: a function. Then it takes any number of arguments, which are collected in the list `args`. It constructs a new list, starting with the function, and followed by the arguments, in reverse order.

First, we macro-expand:

```
user=> (macroexpand '(rev str "hi" (+ 1 2)))
```

```
(str (+ 1 2) "hi")
```

So the `rev` macro took `str` as the function, and `"hi"` and `(+ 1 2)` as the arguments; then constructed a new list with the same function, but the arguments reversed. When we *evaluate* that expression, we get:

```
user=> (eval (macroexpand '(rev str "hi" (+ 1 2))))
"3hi"
```

`macroexpand` takes an expression and returns that expression with all macros expanded. `eval` takes an expression and evaluates it. When you type an unquoted expression into the REPL, Clojure macroexpands, then evaluates. Two stages–the first transforming *code*, the second transforming *values*.

Across languages

Some languages have a *metalanguage*: a language for extending the language itself. In C, for example, macros are implemented by the C preprocessor, which has its own syntax for defining expressions, matching patterns in the source code's text, and replacing that text with other text. But that preprocessor is *not* C–it is a separate language entirely, with special limitations. In Clojure, the metalanguage is *Clojure itself*–the full power of the language is available to restructure programs. This is called a *procedural* macro system. Some Lisps, like Scheme, use a macro system based on templating expressions, and still others use more powerful models like *f-expressions*–but that's a discussion for a later time.

There is another key difference between Lisp macros and many other macro systems: in Lisp, the macros operate on *expressions*: the data structure of the code itself. Because Lisp code is *written* explicitly as a data structure, a tree made out of lists, this transformation is natural. You can *see* the structure of the code, which makes it easy to reason about its transformation. In the C preprocessor, macros operate only on *text*: there is no understanding of the underlying syntax. Even in languages like Scala which have syntactic macros, the fact that the code looks *nothing like* the syntax tree makes it cumbersome to truly restructure expressions.

When people say that Lisp's syntax is "more elegant", or "more beautiful", or "simpler", this is part of what they they mean. By choosing to represent the program directly as a a data structure, we make it much easier to define complex transformations of code itself.

Defining new syntax

What kind of transformations are best expressed with macros?

Most languages encode special syntactic forms–things like "define a function", "call a function", "define a local variable", "if this, then that", and so on. In Clojure, these are called *special forms*. `if` is a special form, for instance. Its definition is built into the language core itself; it cannot be reduced into smaller parts.

```
(if (< 3 x)
  "big"
  "small")
```

Or in Javascript:

```
if (3 < x) {
  return "big";
} else {
  return "small";
}
```

In Javascript, Ruby, and many other languages, these special forms are *fixed*. You cannot define your own syntax. For instance, one cannot define `or` in a language like JS or Ruby: it must be defined *for* you by the language author.

In Clojure, `or` is just a macro.

```
user=> (source or)
(defmacro or
  "Evaluates exprs one at a time, from left to right. If a form
  returns a logical true value, or returns that value and doesn't
  evaluate any of the other expressions, otherwise it returns the
  value of the last expression. (or) returns nil."
  {:added "1.0"}
  ([] nil)
  ([x] x)
  ([x & next]
      `(let [or# ~x]
         (if or# or# (or ~@next)))))
nil
```

That `` ` `` operator–that's called *syntax-quote*. It works just like regular quote–preventing evaluation of the following list–but with a twist: we can escape the quoting rule and substitute in regularly evaluated expressions using *unquote* (`~`), and *unquote-splice* (`~@`). Think of a syntax-quoted expression like a *template* for code, with some parts filled in by evaluated forms.

```
user=> (let [x 2] `(inc x))
(clojure.core/inc user/x)
user=> (let [x 2] `(inc ~x))
(clojure.core/inc 2)
```

See the difference? `~x` *substitutes* the value of x, instead of using `x` as an unevaluated symbol. This code is essentially just shorthand for something like

```
user=> (let [x 2] (list 'clojure.core/inc x))
(inc 2)
```

… where we explicitly constructed a new list with the quoted symbol `'inc` and the current value of `x`. Syntax quote just makes it easier to read the code, since the quoted and expanded expressions have similar shapes.

The `~@` unquote splice works just like `~`, except it explodes a list into *multiple* expressions in the resulting form:

```
user=> `(foo ~[1 2 3])
(user/foo [1 2 3])
user=> `(foo ~@[1 2 3])
(user/foo 1 2 3)
```

`~@` is particularly useful when a function or macro takes an *arbitrary* number of arguments. In the definition of `or`, it's used to expand `(or a b c)` *recursively*.

```
user=> (pprint (macroexpand '(or a b c d)))
(let*
 [or__3943__auto__ a]
 (if or__3943__auto__ or__3943__auto__ (clojure.core/or b c d)))
```

We're using `pprint` (for "pretty print") to make this expression easier to read. `(or a b c d)` is defined in terms of *if*: if the first element is truthy we return it; otherwise we evaluate `(or b c d)` instead, and so on.

The final piece of the puzzle here is that weirdly named symbol: `or__3943__auto__`. That variable was *automatically generated* by Clojure, to prevent *conflicts* with an existing variable name. Because macros rewrite code, they have to be careful not to interfere with local variables, or it could get very confusing. Whenever we need a new variable in a macro, we use `gensym` to *generate a new symbol*.

```
user=> (gensym "hi")
hi326
user=> (gensym "hi")
hi329
user=> (gensym "hi")
hi332
```

Each symbol is different! If we tack on a `#` to the end of a symbol in a syntax-quoted expression, it'll be expanded to a particular gensym:

```
user=> `(let [x# 2] x#)
(clojure.core/let [x__339__auto__ 2] x__339__auto__)
```

Note that you can always escape this safety feature if you *want* to override local variables. That's called *symbol capture*, or an *anaphoric* or *unhygenic* macro. To override local symbols, just use `~'foo` instead of `foo#`.

With all the pieces on the board, let's compare the `or` macro and its expansion:

```
(defmacro or
  "Evaluates exprs one at a time, from left to right. If a form
  returns a logical true value, or returns that value and doesn't
  evaluate any of the other expressions, otherwise it returns the
  value of the last expression. (or) returns nil."
  {:added "1.0"}
  ([] nil)
  ([x] x)
  ([x & next]
      `(let [or# ~x]
         (if or# or# (or ~@next)))))

user=> (pprint (clojure.walk/macroexpand-all
                 '(or (mossy? stone) (cool? stone) (wet? stone))))
(let*
 [or__3943__auto__ (mossy? stone)]
 (if
  or__3943__auto__
  or__3943__auto__
  (let*
   [or__3943__auto__ (cool? stone)]
   (if or__3943__auto__ or__3943__auto__ (wet? stone)))))
```

See how the macro's syntax-quoted `(let ...` has the same shape as the resulting code? `or#` is expanded to a variable named `or__3943__auto__`, which is bound to the expression

`(mossy? stone)`. If that variable is truthy, we return it. Otherwise, we (and here's the recursive part) rebind `or__3943__auto__` to `(cool? stone)` and try again. If *that* fails, we fall back to evaluating `(wet? stone)`–thanks to the base case, the single-argument form of the `or` macro.

Control flow

We've seen that `or` is a macro written in terms of the special form `if`–and because of the way the macro is structured, it does *not* obey the normal execution order. In `(or a b c)`, only `a` is evaluated first–then, only if it is `false` or `nil`, do we evaluate `b`. This is called *short-circuiting*, and it works for `and` as well.

Changing the order of evaluation in a language is called *control flow*, and lets programs make decisions based on varying circumstances. We've already seen `if`:

```
user=> (if (= 2 2) :a :b)
:a
```

`if` takes a predicate and two expressions, and only evaluates one of them, depending on whether the predicate evaluates to a truthy or falsey value. Sometimes you want to evaluate *more than one* expression in order. For this, we have `do`.

```
user=> (if (pos? -5)
         (prn "-5 is positive")
         (do
           (prn "-5 is negative")
           (prn "Who would have thought?")))
"-5 is negative"
"Who would have thought?"
nil
```

`prn` is a function which has a *side effect*: it prints a message to the screen, and returns `nil`. We wanted to print *two* messages, but `if` only takes a single expression per branch–so in our false branch, we used `do` to wrap up two `prn`s into a single expression, and evaluate them in order. `do` returns the value of the final expression, which happens to be `nil` here.

When you only want to take one branch of an `if`, you can use `when`:

```
user=> (when false
         (prn :hi)
         (prn :there))
nil
user=> (when true
         (prn :hi)
         (prn :there))
:hi
:there
nil
```

Because there is only one path to take, `when` takes any number of expressions, and evaluates them only when the predicate is truthy. If the predicate evaluates to `nil` or `false`, `when` does not evaluate its body, and returns `nil`.

Both `when` and `if` have complementary forms, `when-not` and `if-not`, which simply invert the sense of their predicate.

```
user=> (when-not (number? "a string")
         :here)
:here
user=> (if-not (vector? (list 1 2 3))
         :a
         :b)
:a
```

Often, you want to perform some operation, and if it's truthy, re-use that value without recomputing it. For this, we have `when-let` and `if-let`. These work just like `when` and `let` combined.

```
user=> (when-let [x (+ 1 2 3 4)]
         (str x))
"10"
user=> (when-let [x (first [])]
         (str x))
nil
```

`while` evaluates an expression so long as its predicate is truthy. This is generally useful only for side effects, like `prn` or `def`; things that change the state of the world.

```
user=> (def x 0)
#'user/x
user=> (while (< x 5)
  #_=>   (prn x)
  #_=>   (def x (inc x)))
0
1
2
3
4
nil
```

`cond` (for "conditional") is like a multiheaded `if`: it takes *any number* of test/expression pairs, and tries each test in turn. The first test which evaluates truthy causes the following expression to be evaluated; then `cond` returns that expression's value.

```
user=> (cond
  #_=>   (= 2 5) :nope
  #_=>   (= 3 3) :yep
  #_=>   (= 5 5) :cant-get-here
  #_=>   :else   :a-default-value)
:yep
```

If you find yourself making several similar decisions based on a value, try `condp`, for "cond with predicate". For instance, we might categorize a number based on some ranges:

```
(defn category
  "Determines the Saffir-Simpson category of a hurricane, by wind speed in meters/sec"
  [wind-speed]
  (condp <= wind-speed
    70 :F5
    58 :F4
    49 :F3
    42 :F2
       :F1)) ; Default value
user=> (category 10)
:F1
user=> (category 50)
:F3
user=> (category 100)
```

```
:F5
```

`condp` generates code which combines the predicate `<=` with each number, and the value of `wind-speed`, like so:

```
(if (<= 70 wind-speed) :F5
  (if (<= 58 wind-speed) :F4
    (if (<= 49 wind-speed) :F3
      (if (<= 42 wind-speed) :F2
        :F1))))
```

Specialized macros like `condp` are less commonly used than `if` or `when`, but they still play an important role in simplifying repeated code. They clarify the meaning of complex expressions, making them easier to read and maintain.

Finally, there's `case`, which works a little bit like a map of keys to values–only the values are *code*, to be evaluated. You can think of `case` like `(condp = ...)`, trying to match an expression to a particular branch for which it is equal.

```
(defn with-tax
  "Computes the total cost, with tax, of a purchase in the given state."
  [state subtotal]
  (case state
    :WA (* 1.065 subtotal)
    :OR subtotal
    :CA (* 1.075 subtotal)
    ; ... 48 other states ...
    subtotal)) ; a default case
```

Unlike `cond` and `condp`, `case` does *not* evaluate its tests in order. It jumps *immediately* to the matching expression. This makes `case` much faster when there are many branches to take–at the cost of reduced generality.

Recursion

Previously, we defined recursive functions by having those functions call themselves explicitly.

```
(defn sum [numbers]
  (if-let [n (first numbers)]
    (+ n (sum (rest numbers)))
    0))
user=> (sum (range 10))
45
```

But this approach breaks down when we have the function call itself *deeply*, over and over again.

```
user=> (sum (range 100000))

StackOverflowError   clojure.core/range/fn--4269 (core.clj:2664)
```

Every time you call a function, the arguments for that function are stored in memory, in a region called *the stack*. They remain there for as long as the function is being called–including any deeper function calls.

```
    (+ n (sum (rest numbers)))
```

In order to add `n` and `(sum (rest numbers))`, we have to call `sum` *first*–while holding onto the memory for `n` and `numbers`. We can't re-use that memory until *every single recursive call* has completed. Clojure complains, after tens of thousands of stack frames are in use, that it has run out of space in the stack and can allocate no more.

But consider this variation on `sum`:

```
(defn sum
  ([numbers]
   (sum 0 numbers))
  ([subtotal numbers]
   (if-let [n (first numbers)]
     (recur (+ subtotal n) (rest numbers))
     subtotal)))
user=> (sum (range 100000))
4999950000
```

We've added an additional parameter to the function. In its two-argument form, `sum` now takes an accumulator, `subtotal`, which represents the count so far. In addition, `recur` has taken the place of `sum`. Notice, however, that the final expression to be evaluated is not `+`, but `sum` (viz `recur`) itself. We don't need to hang on to any of the variables in this function any more, because the final return value won't depend on them. `recur` hints to the Clojure compiler that we *don't need* to hold on to the stack, and can re-use that space for other things. This is called a *tail-recursive* function, and it requires only a single stack frame no matter how deep the recursive calls go.

Use `recur` wherever possible. It requires much less memory and is much faster than the explicit recursion.

You can also use `recur` within the context of the `loop` macro, where it acts just like an unnamed recursive function with initial values provided. Think of it, perhaps, like a recursive `let`.

```
user=> (loop [i 0
              nums []]
         (if (< 10 i)
           nums
           (recur (inc i) (conj nums i))))
[0 1 2 3 4 5 6 7 8 9 10]
```

Laziness

In chapter 4 we mentioned that most of the sequences in Clojure, like `map`, `filter`, `iterate`, `repeatedly`, and so on, were *lazy*: they did not evaluate any of their elements until required. This too is provided by a macro, called `lazy-seq`.

```
(defn integers
  [x]
  (lazy-seq
    (cons x (integers (inc x)))))
user=> (def xs (integers 0))
#'user/xs
```

This sequence does not terminate; it is *infinitely* recursive. Yet it returned instantaneously. `lazy-seq` interrupted that recursion and restructured it into a sequence which constructs elements only when they are requested.

```
user=> (take 10 xs)
(0 1 2 3 4 5 6 7 8 9)
```

When using `lazy-seq` and its partner `lazy-cat`, you don't have to use `recur`–or even be tail-recursive. The macros interrupt each level of recursion, preventing stack overflows.

You can also delay evaluation of some expressions until later, using `delay` and `deref`.

```
user=> (def x (delay
                (prn "computing a really big number!")
                (last (take 10000000 (iterate inc 0)))))
#'user/x ; Did nothing, returned immediately
user=> (deref x)
"computing a really big number!" ; Now we have to wait!
9999999
```

List comprehensions

Combining recursion and laziness is the *list comprehension* macro, `for`. In its simplest form, `for` works like `map`:

```
user=> (for [x (range 10)] (- x))
(0 -1 -2 -3 -4 -5 -6 -7 -8 -9)
```

Like `let`, `for` takes a vector of `bindings`. Unlike `let`, however, `for` binds its variables to *each possible combination of elements in their corresponding sequences*.

```
user=> (for [x [1 2 3]
             y [:a :b]]
         [x y])
([1 :a] [1 :b] [2 :a] [2 :b] [3 :a] [3 :b])
```

"For each x in the sequence `[1 2 3]`, and for each `y` in the sequence `[:a :b]`, find all `[x y]` pairs." Note that the rightmost variable `y` iterates the fastest.

Like most sequence functions, the `for` macro yields lazy sequences. You can filter them with `take`, `filter`, et al like any other sequence. Or you can use `:while` to tell `for` when to stop, or `:when` to filter out combinations of elements.

```
(for [x     (range 5)
      y     (range 5)
      :when (and (even? x) (odd? y))]
  [x y])
([0 1] [0 3] [2 1] [2 3] [4 1] [4 3])
```

Clojure includes a rich smörgåsbord of control-flow constructs; we'll meet new ones throughout the book.

The threading macros

Sometimes you want to *thread* a computation through several expressions, like a chain. Object-oriented languages like Ruby or Java are well-suited to this style:

```
1.9.3p385 :004 > (0..10).select(&:odd?).reduce(&:+)
25
```

Start with the range `0` to `10`, then call `select` on that range, with the function `odd?`. Finally, take *that* sequence of numbers, and reduce it with the `+` function.

The Clojure threading macros do the same by restructuring a sequence of expressions, inserting each expression as the first (or final) argument in the next expression.

```
user=> (pprint (clojure.walk/macroexpand-all
         '(->> (range 10) (filter odd?) (reduce +))))
(reduce + (filter odd? (range 10)))
user=> (->> (range 10) (filter odd?) (reduce +))
25
```

`->>` took `(range 10)` and inserted it at the end of `(filter odd?)`, forming `(filter odd? (range 10))`. Then it took *that* expression and inserted it at the end of `(reduce +)`. In essence, `->>` *flattens and reverses* a nested chain of operations.

`->`, by contrast, inserts each form in as the *first* argument in the following expression.

```
user=> (pprint (clojure.walk/macroexpand-all
         '(-> {:proton :fermion} (assoc :photon :boson) (assoc :neutrino :fermion))))
(assoc (assoc {:proton :fermion} :photon :boson) :neutrino :fermion)
user=> (-> {:proton :fermion}
           (assoc :photon :boson)
           (assoc :neutrino :fermion))
{:neutrino :fermion, :photon :boson, :proton :fermion}
```

Clojure isn't just `function-oriented` in its syntax; it can be object-oriented, and stack-oriented, and array-oriented, and so on–and *mix all of these styles freely, in a controlled way*. If you don't like the way the language fits a certain problem, you can write a macro which defines a *new* language, specifically for that subproblem.

`cond`, `condp` and `case`, for example, express a language for branching based on predicates. `->`, `->>`, and `doto` express object-oriented and other expression-chaining languages.

— core.match is a set of macros which express powerful *pattern-matching* and substitution languages.
— core.logic expresses syntax for *logic programming*, for finding values which satisfy complex constraints.
— core.async restructures Clojure code into *asynchronous* forms so they can do many things at once.
— For those with a twisted sense of humor, Swiss Arrows extends the threading macros into evil–but delightfully concise!–forms.

We'll see a plethora of macros, from simple to complex, through the course of this book. Each one shares the common pattern of *simplifying code*; reducing tangled or verbose expressions into something more concise, more meaningful, better suited to the problem at hand.

When to use macros

While it's important to be aware of the purpose and behavior of the macro system, you don't need to write your own macros to be productive with Clojure. For now, you'll be just fine writing

code which uses the existing macros in the language. When you *do* need to delve deeper, come back to this guide and experiment. It'll take some time to sink in.

First, know that writing macros is *tricky*, even for experts. It requires you to think at two levels simultaneously, and to be mindful of the distinction between *expression* and underlying *evaluation*. Writing a macro is essentially extending the language, the compiler, the syntax and evaluation model of Clojure, by restructuring *arbitrary* expressions into ones the evaluation system understands. This is hard, and it'll take practice to get used to.

In addition, Clojure macros come with some important restrictions. Because they're expanded prior to evaluation, macros are invisible to functions. They can't be composed functionally–you can't `(map or ...)`, for instance.

So in general, if you *can* solve a problem without writing a macro, *don't write one*. It'll be easier to debug, easier to understand, and easier to compose later. Only reach for macros when you need *new syntax*, or when performance demands the code be transformed at compile time.

When you do write a macro, consider its scope carefully. Keep the transformation simple; and do as much in normal functions as possible. Provide an escape hatch where possible, by doing most of the work in a function, and writing a small wrapper macro which calls that function. Finally, remember the distinction between *code* and what that code *evaluates to*. Use `let` whenever a value is to be re-used, to prevent it being evaluated twice by accident.

For a deeper exploration of Clojure macros in a real-world application, try Language Power.

Review

In Chapter 4, deeply nested expressions led to the desire for a *simpler*, *more direct* expression of a chain of sequence operations. We learned that the Clojure compiler first *expands* expressions before evaluating them, using macros–special functions which take code and return other code. We used macros to define the short-circuiting `or` operator, and followed that with a tour of basic control flow, recursion, laziness, list comprehensions, and chained expressions. Finally, we learned a bit about when and how to write our own macros.

Throughout this chapter we've brushed against the idea of *side effects*: things which change the outside world. We might change a var with `def`, or print a message to the screen with `prn`. Real languages must model a continually shifting universe, which leads us to Chapter Six: Side effects and state.

Problems

1. Using the control flow constructs we've learned, write a `schedule` function which, given an hour of the day, returns what you'll be doing at that time. `(schedule 18)`, for me, returns `:dinner`.

2. Using the threading macros, find how many numbers from 0 to 9999 are palindromes: identical when written forwards and backwards. `121` is a palindrome, as is `7447` and `5`, but not `12` or `953`.

3. Write a macro `id` which takes a function and a list of args: `(id f a b c)`, and returns an expression which calls that function with the given args: `(f a b c)`.

4. Write a macro `log` which uses a var, `logging-enabled`, to determine whether or not to print an expression to the console at compile time. If `logging-enabled` is false, `(log :hi)` should macroexpand to `nil`. If `logging-enabled` is true, `(log :hi)` should macroexpand to `(prn :hi)`. Why would you want to do this check during *compilation*, instead of when running the program? What might you *lose*?

5. (Advanced) Using the `rationalize` function, write a macro `exact` which rewrites any use of `+`, `-`, `*`, or `/` to force the use of *ratios* instead of floating-point numbers. `(* 2452.45 100)` returns `245244.99999999997`, but `(exact (* 2452.45 100))` should return `245245N`

Chapter 6: State

Previously: Macros.

Most programs encompass *change*. People grow up, leave town, fall in love, and take new names. Engines burn through fuel while their parts wear out, and new ones are swapped in. Forests burn down and their logs become nurseries for new trees. Despite these changes, we say “She’s still Nguyen”, “That’s my motorcycle”, “The same woods I hiked through as a child.”

Identity is a skein we lay across the world of immutable facts; a single entity which encompasses change. In programming, identities unify different values over time. Identity types are *mutable references* to *immutable values*.

In this chapter, we’ll move from immutable references to complex concurrent transactions. In the process we’ll get a taste of *concurrency* and *parallelism*, which will motivate the use of more sophisticated identity types. These are not easy concepts, so don’t get discouraged. You don’t have to understand this chapter fully to be a productive programmer, but I do want to hint at *why* things work this way. As you work with state more, these concepts will solidify.

Immutability

The references we’ve used in `let` bindings and function arguments are *immutable*: they never change.

```
user=> (let [x 1]
         (prn (inc x))
         (prn (inc x)))
2
2
```

The expression `(inc x)` did not *alter* `x`: `x` remained `1`. The same applies to strings, lists, vectors, maps, sets, and most everything else in Clojure:

```
user=> (let [x [1 2]]
         (prn (conj x :a))
         (prn (conj x :b)))
[1 2 :a]
[1 2 :b]
```

Immutability also extends to `let` bindings, function arguments, and other symbols. Functions *remember* the values of those symbols at the time the function was constructed.

```
(defn present
  [gift]
  (fn [] gift))

user=> (def green-box (present "clockwork beetle"))
#'user/green-box
user=> (def red-box (present "plush tiger"))
#'user/red-box
user=> (red-box)
"plush tiger"
user=> (green-box)
"clockwork beetle"
```

The `present` function *creates a new function*. That function takes no arguments, and always returns the gift. Which gift? Because `gift` is not an argument to the inner function, it refers to the value from the *outer function body*. When we packaged up the red and green boxes, the functions we created carried with them a memory of the `gift` symbol's value.

This is called *closing over* the `gift` variable; the inner function is sometimes called *a closure*. In Clojure, new functions close over *all* variables except their arguments–the arguments, of course, will be provided when the function is invoked.

Delays

Because functions *close over* their arguments, they can be used to *defer* evaluation of expressions. That's how we introduced functions originally–like `let` expressions, but with a number (maybe zero!) of symbols *missing*, to be filled in at a later time.

```
user=> (do (prn "Adding") (+ 1 2))
"Adding"
3
user=> (def later (fn [] (prn "Adding") (+ 1 2)))
#'user/later
user=> (later)
"Adding"
3
```

Evaluating `(def later ...)` did *not* evaluate the expressions in the function body. Only when we invoked the function `later` did Clojure print `"Adding"` to the screen, and return `3`. This is the basis of *concurrency*: evaluating expressions outside their normal, sequential order.

This pattern of deferring evaluation is so common that there's a standard macro for it, called `delay`:

```
user=> (def later (delay (prn "Adding") (+ 1 2)))
#'user/later
user=> later
#<Delay@2dd31aac: :pending>
user=> (deref later)
"Adding"
3
```

Instead of a function, `delay` creates a special type of Delay object: an identity which *refers* to expressions which should be evaluated later. We extract, or *dereference*, the value of that identity with `deref`. Delays follow the same rules as functions, closing over lexical scope–because `delay` actually macroexpands into an anonymous function.

```
user=> (source delay)
(defmacro delay
  "Takes a body of expressions and yields a Delay object that will
  invoke the body only the first time it is forced (with force or deref/@), and
  will cache the result and return it on all subsequent force
  calls. See also - realized?"
  {:added "1.0"}
  [& body]
    (list 'new 'clojure.lang.Delay (list* `^{:once true} fn* [] body)))
```

Why the `Delay` object instead of a plain old function? Because unlike function invocation, delays only evaluate their expressions *once*. They remember their value, after the first evaluation, and return it for every successive `deref`.

```
user=> (deref later)
3
user=> (deref later)
3
```

By the way, there's a shortcut for `(deref something)`: the wormhole operator `@`:

```
user=> @later ; Interpreted as (deref later)
3
```

Remember how `map` returned a sequence immediately, but didn't actually perform any computation until we asked for elements? That's called *lazy* evaluation. Because delays are lazy, we can avoid doing expensive operations until they're really needed. Like an IOU, we use delays when we aren't ready to do something just yet, but when someone calls in the favor, we'll make sure it happens.

Futures

What if we wanted to *opportunistically* defer computation? Modern computers have multiple cores, and operating systems let us share a core between two tasks. It would be great if we could use that multitasking ability to say, "I don't need the result of evaluating these expressions *yet*, but I'd like it *later*. Could you start working on it in the meantime?"

Enter the *future*: a delay which is evaluated *in parallel*. Like delays, futures return immediately, and give us an *identity* which will point to the value of the last expression in the future–in this case, the value of `(+ 1 2)`.

```
user=> (def x (future (prn "hi") (+ 1 2)))
"hi"
#'user/x
user=> (deref x)
3
```

Notice how the future printed "hi" right away. That's because futures are evaluated in a new *thread*. On multicore computers, two threads can run in *parallel*, on different cores the same time. When there are more threads than cores, the cores *trade off* running different threads. Both parallel and non-parallel evaluation of threads are *concurrent* because expressions from different threads can be evaluated out of order.

```
user=> (dotimes [i 5] (future (prn i)))
14

3
0
2
nil
```

Five threads running at once. Notice that the thread printing `1` didn't even get to move to a new line before `4` showed up–then both threads wrote new lines at the same time. There are

techniques to control this concurrent execution so that things happen in some well-defined sequence, like agents and locks, but we'll discuss those later.

Just like delays, we can deref a future as many times as we want, and the expressions are only evaluated once.

```
user=> (def x (future (prn "hi") (+ 1 2)))
#'user/x"hi"

user=> @x
3
user=> @x
3
```

Futures are the most generic parallel construct in Clojure. You can use futures to do CPU-intensive computation faster, to wait for multiple network requests to complete at once, or to run housekeeping code periodically.

Promises

Delays *defer* evaluation, and futures *parallelize* it. What if we wanted to defer something we *don't even have yet*? To hand someone an empty box and, later, before they open it, sneak in and replacing its contents with an actual gift? Surely I'm not the only one who does birthday presents this way.

```
user=> (def box (promise))
#'user/box
user=> box
#<core$promise$reify__6310@1d7762e: :pending>
```

This box is *pending* a value. Like futures and delays, if we try to open it, we'll get *stuck* and have to wait for something to appear inside:

```
user=> (deref box)
```

But unlike futures and delays, this box won't be filled automatically. Hold the `Control` key and hit `c` to give up on trying to open that package. Nobody else is in this REPL, so we'll have to buy our own presents.

```
user=> (deliver box :live-scorpions!)
#<core$promise$reify__6310@1d7762e: :live-scorpions!>
user=> (deref box)
:live-scorpions!
```

Wow, that's a *terrible* gift. But at least there's something there: when we dereference the box, it opens immediately and live scorpions skitter out. Can we get a do-over? Let's try a nicer gift.

```
user=> (deliver box :puppy)
nil
user=> (deref box)
:live-scorpions!
```

Like delays and futures, there's no going back on our promises. Once delivered, a promise *always* refers to the same value. This is a simple identity type: we can set it to a value once, and read it as many times as we want. `promise` is also a *concurrency primitive*: it guarantees that

any attempt to read the value will *wait* until the value has been written. We can use promises to *synchronize* a program which is being evaluated concurrently–for instance, this simple card game:

```
user=> (def card (promise))
#'user/card
user=> (def dealer (future
                     (Thread/sleep 5000)
                     (deliver card [(inc (rand-int 13))
                                    (rand-nth [:clubs :spades :hearts :diamonds])])))
#'user/dealer
user=> (deref card)
[5 :diamonds]
```

In this program, we set up a `dealer` thread which waits for five seconds (5000 milliseconds), then delivers a random card. While the dealer is sleeping, we try to deref our card–and have to wait until the five seconds are up. Synchronization and identity in one package.

Where delays are lazy, and futures are parallel, promises are concurrent *without specifying how the evaluation occurs*. We control exactly when and how the value is delivered. You can think of both delays and futures as being built atop promises, in a way.

Vars

So far the identities we've discussed have referred (eventually) to a *single* value, but the real world needs names that refer to *different* values at different points in time. For this, we use *vars*.

We've touched on vars before–they're transparent mutable references. Each var has a value associated with it, and that value can change over time. When a var is evaluated, it is replaced by its *present* value transparently–everywhere in the program.

```
user=> (def x :mouse)
#'user/x
user=> (def box (fn [] x))
#'user/box
user=> (box)
:mouse
user=> (def x :cat)
#'user/x
user=> (box)
:cat
```

The `box` function closed over `x`–but calling `(box)` returned *different* results depending on the current value of `x`. Even though the *var* `x` remained unchanged throughout this example, the *value associated with that var* did change!

Using mutable vars allows us to write programs which we can redefine as we go along.

```
user=> (defn decouple [glider]
  #_=>   (prn "bolts released"))
#'user/decouple
user=> (defn launch [glider]
  #_=>   (decouple glider)
  #_=>   (prn glider "away!"))
#'user/launch
user=> (launch "albatross")
"bolts released"
"albatross" "away!"
nil
```

```
user=> (defn decouple [glider]
  #_=>   (prn "tether released"))
#'user/decouple
user=> (launch "albatross")
"tether released"
"albatross" "away!"
```

A reference which is the same everywhere is called a *global variable*, or simply a *global*. But vars have an additional trick up their sleeve: with a *dynamic* var, we can override their value only within the scope of a particular function call, and nowhere else.

```
user=> (def ^:dynamic *board* :maple)
#'user/*board*
```

`^:dynamic` tells Clojure that this var can be overridden in one particular scope. By convention, dynamic variables are named with asterisks around them–this reminds us, as programmers, that they are likely to change. Next, we define a function that uses that dynamic var:

```
user=> (defn cut [] (prn "sawing through" *board*))
#'user/cut
```

Note that `cut` closes over the var `*board*`, but not the *value* :maple. Every time the function is invoked, it looks up the *current* value of `*board*`.

```
user=> (cut)
"sawing through" :maple
nil
user=> (binding [*board* :cedar] (cut))
"sawing through" :cedar
nil
user=> (cut)
"sawing through" :maple
```

Like `let`, the `binding` macro assigns a value to a name–but where `fn` and `let` create immutable *lexical scope*, `binding` creates *dynamic scope*. The difference? Lexical scope is constrained to the literal text of the `fn` or `let` expression–but dynamic scope propagates *through function calls*.

Within the `binding` expression, and in every function called from that expression, and every function called from *those* functions, and so on, `*board*` has the value `:cedar`. Outside the `binding` expression, the value is still `:maple`. This safety property holds even when the program is executed in multiple threads: only the thread which evaluated the `binding` expression uses that value. Other threads are unaffected.

While we use `def` all the time in the REPL, in real programs you should only mutate vars sparingly. They're intended for naming functions, important bits of global data, and for tracking the *environment* of a program–like where to print messages with `prn`, which database to talk to, and so on. Using vars for mutable program state is a recipe for disaster, as we're about to see.

Atoms

Vars can be read, set, and dynamically bound–but they aren't easy to *evolve*. Imagine building up a set of integers:

```
user=> (def xs #{})
#'user/xs
user=> (dotimes [i 10] (def xs (conj xs i)))
user=> xs
#{0 1 2 3 4 5 6 7 8 9}
```

For each number from 0 to 9, we take the current set of numbers `xs`, add a particular number `i` to that set, and redefine `xs` as the result. This is a common idiom in imperative language like C, Ruby, Javascript, or Java–all variables are mutable by default.

```
ImmutableSet xs = new ImmutableSet();
for (int i = 0; i++; i < 10) {
  xs = xs.add(i);
}
```

It seems straightforward enough, but there are serious problems lurking here. Specifically, this program is not *thread safe*.

```
user=> (def xs #{})
user=> (dotimes [i 10] (future (def xs (conj xs i))))
#'user/xs
nil
user=> xs
#{1 4 5 7}
```

This program runs 10 threads in parallel, and each reads the current value of `xs`, adds its particular number, and defines `xs` to be that new set of numbers. This read-modify-update process assumed that all updates would be *consecutive*–not *concurrent*. When we allowed the program to do two read-modify-updates at the same time, updates were lost.

1. Thread 2 read `#{0 1}`
2. Thread 3 read `#{0 1}`
3. Thread 2 wrote `#{0 1 2}`
4. Thread 3 wrote `#{0 1 3}`

This interleaving of operations allowed the number `2` to slip through the cracks. We need something stronger–an identity which supports safe transformation from one state to another. Enter *atoms.

```
user=> (def xs (atom #{}))
#'user/xs
user=> xs
#<Atom@30bb8cc9: #{}>
```

The initial value of this atom is `#{}`. Unlike vars, atoms are not transparent. When evaluated, they don't return their underlying values–but notice that when printed, the current value is hiding inside. To get the current value out of an atom, we have to use `deref` or `@`.

```
user=> (deref xs)
#{}
user=> @xs
```

```
#{}
```

Like vars, atoms can be set to a particular value–but instead of `def`, we use `reset!`. The exclamation point (sometimes called a *bang*) is there to remind us that this function *modifies* the state of its arguments–in this case, changing the value of the atom.

```
user=> (reset! xs :foo)
:foo
user=> xs
#<Atom@30bb8cc9: :foo>
```

Unlike vars, atoms can be safely *updated* using `swap!`. `swap!` uses a pure function which takes the current value of the atom and returns a *new* value. Under the hood, Clojure does some tricks to ensure that these updates are *linearizable*, which means:

1. All updates with `swap! complete in what *appears* to be a single consecutive order.
2. The effect of a swap! never takes place before calling `swap!`.
3. The effect of a swap! is visible to everyone once swap! returns.

```
user=> (def x (atom 0))
#'user/x
user=> (swap! x inc)
1
user=> (swap! x inc)
2
```

The first `swap!` reads the value `0`, calls `(inc 0)` to obtain `1`, and writes `1` back to the atom. Each call to `swap!` returns the value that was just written.

We can pass additional arguments to the function `swap!` calls. For instance, `(swap! x + 5 6)` will call `(+ x 5 6)` to find the new value. Now we have the tools to correct our parallel program from earlier:

```
user=> (def xs (atom #{}))
#'user/xs
user=> (dotimes [i 10] (future (swap! xs conj i)))
nil
user=> @xs
#{0 1 2 3 4 5 6 7 8 9}
```

Note that the function we use to update an atom must be *pure*–must not mutate any state–because when resolving conflicts between multiple threads, Clojure might need to call the update function more than once. Clojure's reliance on immutable datatypes, immutable variables, and pure functions *enables* this approach to linearizable mutability. Languages which emphasize mutable datatypes need to use other constructs.

Atoms are the workhorse of Clojure state. They're lightweight, safe, fast, and flexible. You can use atoms with any immutable datatype–for instance, a map to track complex state. Reach for an atom whenever you want to update a single thing over time.

Refs

Atoms are a great way to represent state, but they are only linearizable *individually*. Updates to an atom aren't well-ordered with respect to other atoms, so if we try to update more than one atom at once, we could see the same kinds of bugs that we did with vars.

For multi-identity updates, we need a stronger safety property than single-atom linearizability. We want *serializability*: a global order. For this, Clojure has an identity type called a *Ref*.

```
user=> (def x (ref 0))
#'user/x
user=> x
#<Ref@1835d850: 0>
```

Like all identity types, refs are dereferencable:

```
user=> @x
0
```

But where atoms are updated individually with `swap!`, refs are updated in *groups* using `dosync` transactions. Just as we `reset!` an atom, we can set refs to new values using `ref-set`–but unlike atoms, we can change more than one ref at once.

```
user=> (def x (ref 0))
user=> (def y (ref 0))
user=> (dosync
         (ref-set x 1)
         (ref-set y 2))
2
user=> [@x @y]
[1 2]
```

The equivalent of `swap!`, for a ref, is `alter`:

```
user=> (def x (ref 1))
user=> (def y (ref 2))
user=> (dosync
         (alter x + 2)
         (alter y inc))
3
user=> [@x @y]
[3 3]
```

All `alter` operations within a `dosync` take place atomically–their effects are never interleaved with other transactions. If it's OK for an operation to take place out of order, you can use `commute` instead of `alter` for a performance boost:

```
user=> (dosync
         (commute x + 2)
         (commute y inc))
```

These updates are *not* guaranteed to take place in the same order–but if all our transactions are equivalent, we can *relax* the ordering constraints. x + 2 + 3 is equal to x + 3 + 2, so we can do the additions in either order. That's what *commutative* means: the same result from all orders. It's a weaker, but faster kind of safety property.

Finally, if you want to read a value from one ref and use it to update another, use `ensure` instead of `deref` to perform a *strongly consistent read*–one which is guaranteed to take place in the same logical order as the `dosync` transaction itself. To add `y`'s current value to `x`, use:

```
user=> (dosync
         (alter x + (ensure y)))
```

Refs are a powerful construct, and make it easier to write complex transactional logic safely. However, that safety comes at a cost: refs are typically an order of magnitude slower to update than atoms.

Use refs only where you need to update multiple pieces of state independently–specifically, where different transactions need to work with distinct but *partly overlapping* pieces of state. If there's no overlap between updates, use distinct atoms. If all operations update the same identities, use a single atom to hold a map of the system's state. If a system requires complex interlocking state spread throughput the program–that's when to reach for refs.

Summary

We moved beyond immutable programs into the world of *changing state*–and discovered the challenges of concurrency and parallelism. Where symbols provide immutable and transparent names for values objects, Vars provide *mutable* transparent names. We also saw a host of anonymous identity types for different purposes: delays for lazy evaluation, futures for parallel evaluation, and promises for arbitrary handoff of a value. Updates to vars are unsafe, so atoms and refs provide linearizable and serializable identities where transformations are *safe*.

Where reading a symbol or var is *transparent*–they evaluate directly to their current values–reading these new identity types requires the use of `deref`. Delays, futures, and promises *block*: deref must wait until the value is ready. This allows synchronization of concurrent threads. Atoms and refs, by contrast, can be read immediately at any time–but *updating* their values should occur within a `swap!` or `dosync` transaction, respectively.

Type Mutability Reads Updates Evaluation Scope Symbol Immutable Transparent Lexical Var Mutable Transparent Unrestricted Global/Dynamic Delay Mutable Blocking Once only Lazy Future Mutable Blocking Once only Parallel Promise Mutable Blocking Once only Atom Mutable Nonblocking Linearizable Ref Mutable Nonblocking Serializable

State is undoubtedly the hardest part of programming, and this chapter probably felt overwhelming! On the other hand, we're now equipped to solve serious problems. We'll take a break to apply what we've learned through practical examples, in Chapter Seven: Logistics.

Exercises

Finding the sum of the first 10000000 numbers takes about 1 second on my machine:

```
user=> (defn sum [start end] (reduce + (range start end)))
user=> (time (sum 0 1e7))
"Elapsed time: 1001.295323 msecs"
49999995000000
```

1. Use `delay` to compute this sum lazily; show that it takes no time to return the delay, but roughly 1 second to `deref`.

2. We can do the computation in a new thread directly, using `(.start (Thread. (fn [] (sum 0 1e7))))`–but this simply runs the `(sum)` function and discards the results. Use

a promise to hand the result back out of the thread. Use this technique to write your own version of the `future` macro.

3. If your computer has two cores, you can do this expensive computation twice as fast by splitting it into two parts: `(sum 0 (/ 1e7 2))`, and `(sum (/ 1e7 2) 1e7)`, then adding those parts together. Use `future` to do both parts at once, and show that this strategy gets the same answer as the single-threaded version, but takes roughly half the time.

4. Instead of using `reduce`, store the sum in an atom and use two futures to add each number from the lower and upper range to that atom. Wait for both futures to complete using `deref`, then check that the atom contains the right number. Is this technique faster or slower than `reduce`? Why do you think that might be?

5. Instead of using a lazy list, imagine two threads are removing tasks from a pile of work. Our work pile will be the list of all integers from 0 to 10000:

```
user=> (def work (ref (apply list (range 1e5))))
user=> (take 10 @work)
(0 1 2 3 4 5 6 7 8 9)
```

And the sum will be a ref as well:

```
user=> (def sum (ref 0))
```

Write a function which, in a `dosync` transaction, removes the first number in `work` and adds it to `sum`. Then, in two futures, call that function over and over again until there's no work left. Verify that `@sum` is `4999950000`. Experiment with different combinations of `alter` and `commute`–if both are correct, is one faster? Does using `deref` instead of `ensure` change the result?

Chapter 7: Logistics

Previously, we covered state and mutability.

Up until now, we've been programming primarily at the REPL. However, the REPL is a limited tool. While it lets us explore a problem interactively, that interactivity comes at a cost: changing an expression requires retyping the entire thing, editing multi-line expressions is awkward, and our work vanishes when we restart the REPL–so we can't share our programs with others, or run them again later. Moreover, programs in the REPL are hard to organize. To solve large problems, we need a way of writing programs *durably*–so they can be read and evaluated later.

In addition to the code itself, we often want to store *ancillary* information. *Tests* verify the correctness of the program. *Resources* like precomputed databases, lookup tables, images, and text files provide other data the program needs to run. There may be *documentation*: instructions for how to use and understand the software. A program may also depend on code from *other* programs, which we call *libraries*, *packages*, or *dependencies*. In Clojure, we have a standardized way to bind together all these parts into a single directory, called a *project*.

Project structure

We created a project at the start of this book by using Leiningen, the Clojure project tool.

```
$ lein new scratch
```

`scratch` is the name of the project, and also the name of the directory where the project's files live. Inside the project are a few files.

```
$ cd scratch; ls
doc  project.clj  README.md  resources  src  target  test
```

`project.clj` defines the project: its name, its version, dependencies, and so on. Notice the name of the project (`scratch`) comes first, followed by the version (`0.1.0-SNAPSHOT`). `-SNAPSHOT` versions are for development; you can change them at any time, and any projects which depend on the snapshot will pick up the most recent changes. A version which does *not* end in `-SNAPSHOT` is fixed: once published, it always points to the same version of the project. This allows projects to specify precisely which projects they depend on. For example, scratch's `project.clj` says scratch depends on `org.clojure/clojure` version `1.5.1`.

```
(defproject scratch "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"] ])
```

`README.md` is the first file most people open when they look at a new project, and Lein generates a generic readme for you to fill in later. We call this kind of scaffolding or example a "stub"; it's just there to remind you what sort of things to write yourself. You'll notice the readme includes the name of the project, some notes on what it does and how to use it, a copyright notice where your name should go, and a license, which sets the legal terms for the use of the

project. By default, Leiningen suggests the Eclipse Public License, which allows everyone to use and modify the software, so long as they preserve the license information.

The `doc` directory is for documentation; sometimes hand-written, sometimes automatically generated from the source code. `resources` is for additional files, like images. `src` is where Clojure code lives, and `test` contains the corresponding tests. Finally, `target` is where Leiningen stores compiled code, built packages, and so on.

Namespaces

Every lein project starts out with a stub namespace containing a simple function. Let's take a look at that namespace now–it lives in `src/scratch/core.clj`:

```
(ns scratch.core)

(defn foo
  "I don't do a whole lot."
  [x]
  (println x "Hello, World!"))
```

The first part of this file defines the *namespace* we'll be working in. The `ns` macro lets the Clojure compiler know that all following code belongs in the `scratch.core` namespace. Remember, `scratch` is the name of our project. `scratch.core` is for the core functions and definitions of the scratch project. As projects expand, we might add new namespaces to *separate* our work into smaller, more understandable pieces. For instance, Clojure's primary functions live in `clojure.core`, but there are auxiliary functions for string processing in `clojure.string`, functions for interoperating with Java's input-output system in `clojure.java.io`, for printing values in `clojure.pprint`, and so on.

`def`, `defn`, and peers always work in the scope of a particular *namespace*. The function `foo` in `scratch.core` is *different* from the function `foo` in `scratch.pad`.

```
scratch.foo=> (ns scratch.core)
nil
scratch.core=> (def foo "I'm in core")
#'scratch.core/foo
scratch.core=> (ns scratch.pad)
nil
scratch.pad=> (def foo "I'm in pad!")
#'scratch.pad/foo
```

Notice the full names of these vars are different: `scratch.core/foo` vs `scratch.pad/foo`. You can always refer to a var by its fully qualified name: the namespace, followed by a slash `/`, followed by the short name.

Inside a namespace, symbols resolve to variables which are defined in that namespace. So in `scratch.pad`, `foo` refers to `scratch.pad/foo`.

```
scratch.pad=> foo
"I'm in pad!"
```

Namespaces automatically include `clojure.core` by default; which is where all the standard functions, macros, and special forms come from. `let`, `defn`, `filter`, `vector`, etc: all live in

`clojure.core`, but are automatically *included* in new namespaces so we can refer to them by their short names.

Notice that the values for `foo` we defined in `scratch.pad` and `scratch.core` aren't available in other namespaces, like `user`.

```
scratch.pad=> (ns user)
nil
user=> foo

CompilerException java.lang.RuntimeException: Unable to resolve symbol: foo in this context, cor
```

To access things from other namespaces, we have to *require* them in the namespace definition.

```
user=> (ns user (:require [scratch.core]))
nil
user=> scratch.core/foo
"I'm in core"
```

The `:require` part of the `ns` declaration told the compiler that the `user` namespace needed access to `scratch.core`. We could then refer to the fully qualified name `scratch.core/foo`.

Often, writing out the full namespace is cumbersome–so you can give a short alias for a namespace like so:

```
user=> (ns user (:require [scratch.core :as c]))
nil
user=> c/foo
"I'm in core"
```

The `:as` directive indicates that anywhere we write `c/something`, the compiler should expand that to `scratch.core/something`. If you plan on using a var from another namespace often, you can *refer* it to the local namespace–which means you may omit the namespace qualifier entirely.

```
user=> (ns user (:require [scratch.core :refer [foo]]))
nil
user=> foo
"I'm in core"
```

You can refer functions into the current namespace by listing them: `[foo bar ...]`. Alternatively, you can suck in *every* function from another namespace by saying `:refer :all`:

```
user=> (ns user (:require [scratch.core :refer :all]))
nil
user=> foo
"I'm in core"
```

Namespaces *control complexity* by isolating code into more understandable, related pieces. They make it easier to read code by keeping similar things together, and unrelated things apart. By making dependencies between namespaces explicit, they make it clear how groups of functions relate to one another.

If you've worked with Erlang, Modula-2, Haskell, Perl, or ML, you'll find namespaces analogous to *modules* or *packages*. Namespaces are often large, encompassing hundreds of functions; and most projects use only a handful of namespaces.

By contrast, object-oriented programming languages like Java, Scala, Ruby, and Objective C organize code in *classes*, which combine *names* and *state* in a single construct. Because all functions in a class operate on the same state, object-oriented languages tend to have *many* classes with *fewer* functions in each. It's not uncommon for a typical Java project to define hundreds or thousands of classes containing only one or two functions each. If you come from an object-oriented language, it can feel a bit unusual to combine so many functions in a single scope–but because functional programs isolate state differently, this is *normal*. If, on the other hand, you move *to* an object-oriented language after Clojure, remember that OO languages compose differently. Objects with hundreds of functions are usually considered unwieldy and should be split into smaller pieces.

Code and tests

It's perfectly fine to test small programs in the REPL. We've written and refined hundreds of functions that way: by calling the function and seeing what happens. However, as programs grow in scope and complexity, testing them by hand becomes harder and harder. If you change the behavior of a function which ten other functions rely on, you may have to re-test *all ten* by hand. In real programs, a small change can alter thousands of distinct behaviors, all of which should be verified.

Wherever possible, we want to *automate* software tests–making the test itself *another program*. If the test suite runs in a matter of seconds, we can make changes freely–re-running the tests continuously to verify that everything still works.

As a simple example, let's write and test a single function in `src/scratch/core.clj`. How about exponentiation–raising a number to the given power?

```
(ns scratch.core)

(defn pow
  "Raises base to the given power. For instance, (pow 3 2) returns three squared, or nine."
  [base power]
  (apply * (repeat base power)))
```

So we *repeat* the base *power* times, then call `*` with that sequence of bases to multiply them all together. Seems straightforward enough. Now we need to test it.

By default, all lein projects come with a simple test stub. Let's see it in action by running `lein test`.

```
aphyr@waterhouse:~/scratch$ lein test

lein test scratch.core-test

lein test :only scratch.core-test/a-test

FAIL in (a-test) (core_test.clj:7)
FIXME, I fail.
expected: (= 0 1)
  actual: (not (= 0 1))
```

```
Ran 1 tests containing 1 assertions.
1 failures, 0 errors.
Tests failed.
```

A *failure* is when a test returns the wrong value. An *error* is when a test throws an exception. In this case, the test named `a-test`, in the file `core_test.clj`, on line 7, failed. That test expected zero to be equal to one–but found that 0 and 1 were (in point of fact) not equal. Let's take a look at that test, in `test/scratch/core_test.clj`.

```
(ns scratch.core-test
  (:require [clojure.test :refer :all]
            [scratch.core :refer :all]))

(deftest a-test
  (testing "FIXME, I fail."
    (is (= 0 1))))
```

These tests live in a namespace too! Notice that namespaces and file names match up: `scratch.core` lives in `src/scratch/core.clj`, and `scratch.core-test` lives in `test/scratch/core_test.clj`. Dashes (`-`) in namespaces correspond to underscores (`_`) in filenames, and dots (`.`) correspond to directory separators (`/`).

The `scratch.core-test` namespace is responsible for testing things in `scratch.core`. Notice that it requires two namespaces: `clojure.test`, which provides testing functions and macros, and `scratch.core`, which is the namespace we want to test.

Then we define a test using `deftest`. `deftest` takes a symbol as a test name, and then any number of expressions to evaluate. We can use `testing` to split up tests into smaller pieces. If a test fails, `lein test` will print out the enclosing `deftest` and `testing` names, to make it easier to figure out what went wrong.

Let's change this test so that it passes. 0 should equal 0.

```
(deftest a-test
  (testing "Numbers are equal to themselves, right?"
    (is (= 0 0))))
```

```
aphyr@waterhouse:~/scratch$ lein test

lein test scratch.core-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

Wonderful! Now let's test the `pow` function. I like to start with a really basic case and work my way up to more complicated ones. 1^1 is 1, so:

```
(deftest pow-test
  (testing "unity"
    (is (= 1 (pow 1 1)))))
```

```
aphyr@waterhouse:~/scratch$ lein test

lein test scratch.core-test

Ran 1 tests containing 1 assertions.
```

```
0 failures, 0 errors.
```

Excellent. How about something harder?

```
(deftest pow-test
  (testing "unity"
    (is (= 1 (pow 1 1))))

  (testing "square integers"
    (is (= 9 (pow 3 2)))))
```

```
aphyr@waterhouse:~/scratch$ lein test

lein test scratch.core-test

lein test :only scratch.core-test/pow-test

FAIL in (pow-test) (core_test.clj:10)
square integers
expected: (= 9 (pow 3 2))
  actual: (not (= 9 8))

Ran 1 tests containing 2 assertions.
1 failures, 0 errors.
Tests failed.
```

That's odd. 3^2 should be 9, not 8. Let's double-check our code in the REPL. `base` was 3, and `power` was 2, so…

```
user=> (repeat 3 2)
(2 2 2)
user=> (* 2 2 2)
8
```

Ah, there's the problem. We're mis-using `repeat`. Instead of repeating 3 twice, we repeated 2 thrice.

```
user=> (doc repeat)
-------------------------
clojure.core/repeat
([x] [n x])
  Returns a lazy (infinite!, or length n if supplied) sequence of xs.
```

Let's redefine `pow` with the correct arguments to `repeat`:

```
(defn pow
  "Raises base to the given power. For instance, (pow 3 2) returns three
  squared, or nine."
  [base power]
  (apply * (repeat power base)))
```

How about $0^{0?}$ By convention, mathematicians define 0^0 as 1.

```
(deftest pow-test
  (testing "unity"
    (is (= 1 (pow 1 1))))

  (testing "square integers"
    (is (= 9 (pow 3 2))))

  (testing "0^0"
```

```
    (is (= 1 (pow 0 0)))))
```

```
aphyr@waterhouse:~/scratch$ lein test

lein test scratch.core-test

Ran 1 tests containing 3 assertions.
0 failures, 0 errors.
```

Hey, what do you know? It works! But *why*?

```
user=> (repeat 0 0)
()
```

What happens when we call `*` with an *empty* list of arguments?

```
user=> (*)
1
```

Remember when we talked about how the zero-argument forms of `+`, and `*` made some definitions simpler? This is one of those times. We didn't have to define a special exception for zero powers because `(*)` returns the multiplicative identity 1, by convention.

Exploring data

The last bit of logistics we need to talk about is *working with other people's code*. Clojure projects, like most modern programming environments, are built to work together. We can use libraries to parse data, solve mathematical problems, render graphics, perform simulations, talk to robots, or predict the weather. As a quick example, I'd like to imagine that you and I are public-health researchers trying to identify the best location for an ad campaign to reduce drunk driving.

The FBI's Uniform Crime Reporting database tracks the annual tally of different types of arrests, broken down by county–but the data files provided by the FBI are a mess to work with. Helpfully, Matt Aliabadi has helpfully organized the UCR's somewhat complex format into nice, normalized files in a data format called JSON, and made them available on Github. Let's download the most recent year's normalized data, and save it in the `scratch` directory.

What's *in* this file, anyway? Let's take a look at the first few lines using `head`:

```
aphyr@waterhouse:~/scratch$ head 2008.json
[
  {
    "icpsr_study_number": null,
    "icpsr_edition_number": 1,
    "icpsr_part_number": 1,
    "icpsr_sequential_case_id_number": 1,
    "fips_state_code": "01",
    "fips_county_code": "001",
    "county_population": 52417,
    "number_of_agencies_in_county": 3,
```

This is a data format called JSON, and it looks a lot like Clojure's data structures. That's the start of a vector on the first line, and the second line starts a map. Then we've got string keys like `"icpsr_study_number"`, and values which look like `null` (`nil`), numbers, or strings.

But in order to *work* with this file, we'll need to *parse* it into Clojure data structures. For that, we can use a JSON parsing library, like Cheshire.

Cheshire, like most Clojure libraries, is published on an internet repository called Clojars. To include it in our scratch project, all we have to do is add open `project.clj` in a text editor, and add a line specifying that we want to use a particular version of Cheshire.

```
(defproject scratch "0.1.0-SNAPSHOT"
  :description "Just playing around"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.5.1"]
                 [cheshire "5.3.1"]])
```

Now we'll exit the REPL with Control+D (^D), and restart it with `lein repl`. Leiningen, the Clojure package manager, will automatically download Cheshire from Clojars and make it available in the new REPL session.

Now let's figure out how to parse the JSON file. Looking at Cheshire's README shows an example that looks helpful:

```
;; parse some json and get keywords back
(parse-string "{\"foo\":\"bar\"}" true)
;; => {:foo "bar"}
```

So Cheshire includes a parse-string function which can take a string and return a data structure. How can we get a string out of a file? Using `slurp`:

```
user=> (use 'cheshire.core)
nil
user=> (parse-string (slurp "2008.json"))
...
```

Woooow, that's a lot of data! Let's chop it down to something more manageable. How about the first entry?

```
user=> (first (parse-string (slurp "2008.json")))
{"syntheticdrug_salemanufacture" 1, "all_other_offenses_except_traffic" 900, "arson" 3, ...}
user=> (-> "2008.json" slurp parse-string first)
```

It'd be nicer if this data used keywords instead of strings for its keys. Let's use the second argument to Chesire's `parse-string` to convert all the keys in maps to keywords.

```
user=> (first (parse-string (slurp "2008.json") true))
{:other_assaults 288, :gambling_all_other 0, :arson 3, ... :drunkenness 108}
```

Since we're going to be working with this dataset over and over again, let's bind it to a variable for easy re-use.

```
user=> (def data (parse-string (slurp "2008.json") true))
#'user/data
```

Now we've got a big long vector of counties, each represented by a map–but we're just interested in the *DUIs* of each one. What does that look like? Let's *map* each county to its `:driving_under_influence`.

```
user=> (->> data (map :driving_under_influence))
(198 1095 114 98 135 4 122 587 204 53 177 ...
```

What's the most any county has ever reported?

```
user=> (->> data (map :driving_under_influence) (apply max))
45056
```

45056 counts in one year? Wow! What about the second-worst county? The easiest way to find the *top n* counties is to *sort* the list, then look at the final elements.

```
user=> (->> data (map :driving_under_influence) sort (take-last 10))
(8589 10432 10443 10814 11439 13983 17572 18562 26235 45056)
```

So the top 10 counties range from 8549 counts to 45056 counts. What's the *most common* count? Clojure comes with a built-in function called `frequencies` which takes a sequence of elements, and returns a map from each element to how many times it appeared in the sequence.

```
user=> (->> data (map :driving_under_influence) frequencies)
{0 227, 1024 1, 45056 1, 32 15, 2080 1, 64 12 ...
```

Now let's take those [drunk-driving, frequency] pairs and sort them by key to produce a *histogram*. `sort-by` takes a function to apply to each element in the collection–in this case, a key-value pair–and returns something that can be sorted, like a number. We'll choose the `key` function to extract the key from each key-value pair, effectively sorting the counties by number of reported incidents.

```
user=> (->> data (map :driving_under_influence) frequencies (sort-by key) pprint)
([0 227]
 [1 24]
 [2 17]
 [3 20]
 [4 17]
 [5 24]
 [6 23]
 [7 23]
 [8 17]
 [9 19]
 [10 29]
 [11 20]
 [12 18]
 [13 21]
 [14 25]
 [15 13]
 [16 18]
 [17 16]
 [18 17]
 [19 11]
 [20 8]
 ...
```

So a ton of counties (227 out of 3172 total) report no drunk driving; a few hundred have one incident, a moderate number have 10-20, and it falls off from there. This is a common sort of shape in statistics; often a hallmark of an exponential distribution.

How about the 10 worst counties, all the way out on the end of the curve?

```
user=> (->> data (map :driving_under_influence) frequencies (sort-by key) (take-
last 10) pprint)
([8589 1]
 [10432 1]
 [10443 1]
 [10814 1]
 [11439 1]
 [13983 1]
 [17572 1]
 [18562 1]
 [26235 1]
 [45056 1])
```

So it looks like 45056 is high, but there are 8 other counties with tens of thousands of reports too. Let's back up to the original dataset, and sort it by DUIs:

```
user=> (->> data (sort-by :driving_under_influence) (take-last 10) pprint)
({:other_assaults 3096,
  :gambling_all_other 3,
  :arson 106,
  :have_stolen_property 698,
  :syntheticdrug_salemanufacture 0,
  :icpsr_sequential_case_id_number 220,
  :drug_abuse_salemanufacture 1761,
  ...
```

What we're looking for is the county names, but it's a little hard to read these enormous maps. Let's take a look at just the keys that define each county, and see which ones might be useful. We'll take this list of counties, map each one to a list of its keys, and concatenate those lists together into one big long list. `mapcat` maps and concatenates in a single step. We expect the same keys to show up over and over again, so we'll remove duplicates by merging all those keys `into` a `sorted-set`.

```
user=> (->> data (sort-by :driving_under_influence) (take-
last 10) (mapcat keys) (into (sorted-set)) pprint)
#{:aggravated_assaults :all_other_offenses_except_traffic :arson
  :auto_thefts :bookmaking_horsesport :burglary :county_population
  :coverage_indicator :curfew_loitering_laws :disorderly_conduct
  :driving_under_influence :drug_abuse_salemanufacture
  :drug_abuse_violationstotal :drug_possession_other
  :drug_possession_subtotal :drunkenness :embezzlement
  :fips_county_code :fips_state_code :forgerycounterfeiting :fraud
  :gambling_all_other :gambling_total :grand_total
  :have_stolen_property :icpsr_edition_number :icpsr_part_number
  :icpsr_sequential_case_id_number :icpsr_study_number :larceny
  :liquor_law_violations :marijuana_possession
  :marijuanasalemanufacture :multicounty_jurisdiction_flag :murder
  :number_of_agencies_in_county :numbers_lottery
  :offenses_against_family_child :opiumcocaine_possession
  :opiumcocainesalemanufacture :other_assaults :otherdang_nonnarcotics
  :part_1_total :property_crimes :prostitutioncomm_vice :rape :robbery
  :runaways :sex_offenses :suspicion :synthetic_narcoticspossession
  :syntheticdrug_salemanufacture :vagrancy :vandalism :violent_crimes
  :weapons_violations}
```

Ah, `:fips_county_code` and `:fips_state_code` look promising. Let's compact the dataset to just drunk driving and those codes, using `select-keys`.

```
user=> (->> data (sort-by :driving_under_influence) (take-last 10) (map #(select-
keys % [:driving_under_influence :fips_county_code :fips_state_code])) pprint)
({:fips_state_code "06",
  :fips_county_code "067",
```

```
  :driving_under_influence 8589}
 {:fips_state_code "48",
  :fips_county_code "201",
  :driving_under_influence 10432}
 {:fips_state_code "32",
  :fips_county_code "003",
  :driving_under_influence 10443}
 {:fips_state_code "06",
  :fips_county_code "065",
  :driving_under_influence 10814}
 {:fips_state_code "53",
  :fips_county_code "033",
  :driving_under_influence 11439}
 {:fips_state_code "06",
  :fips_county_code "071",
  :driving_under_influence 13983}
 {:fips_state_code "06",
  :fips_county_code "059",
  :driving_under_influence 17572}
 {:fips_state_code "06",
  :fips_county_code "073",
  :driving_under_influence 18562}
 {:fips_state_code "04",
  :fips_county_code "013",
  :driving_under_influence 26235}
 {:fips_state_code "06",
  :fips_county_code "037",
  :driving_under_influence 45056})
```

Now we're getting somewhere–but we need a way to interpret these state and county codes. Googling for "FIPS" led me to Wikipedia's account of the FIPS county code system, and the NOAA's ERDDAP service, which has a table mapping FIPS codes to state and county names. Let's save that file as fips.json.

Now we'll slurp that file into the REPL and parse it, just like we did with the crime dataset.

```
user=> (def fips (parse-string (slurp "fips.json") true))
```

Let's take a quick look at the structure of this data:

```
user=> (keys fips)
(:table)
user=> (keys (:table fips))
(:columnNames :columnTypes :rows)
user=> (->> fips :table :columnNames)
["FIPS" "Name"]
```

Great, so we expect the rows to be a list of FIPS code and Name.

```
user=> (->> fips :table :rows (take 3) pprint)
(["02000" "AK"]
 ["02013" "AK, Aleutians East"]
 ["02016" "AK, Aleutians West"])
```

Perfect. Now that's we've done some exploratory work in the REPL, let's shift back to an editor. Create a new file in `src/scratch/crime.clj`:

```
(ns scratch.crime
  (:require [cheshire.core :as json]))

(def fips
  "A map of FIPS codes to their county names."
  (->> (json/parse-string (slurp "fips.json") true)
       :table
```

```
       :rows
       (into {})))
```

We're just taking a snippet we wrote in the REPL–parsing the FIPS dataset–and writing it down for use as a part of a bigger program. We use `(into {})` to convert the sequence of `[fips, name]` pairs into a map, just like we used `into (sorted-set)` to merge a list of keywords into a set earlier. `into` works just like `conj`, repeated over and over again, and is an incredibly useful tool for building up collections of things.

Back in the REPL, let's check if that worked:

```
user=> (use 'scratch.crime :reload)
nil
user=> (fips "10001")
"DE, Kent"
```

Remember, maps act like functions in Clojure, so we can use the `fips` map to look up the names of counties efficiently.

We also have to load the UCR crime file–so let's split that load-and-parse code into its own function:

```
(defn load-json
  "Given a filename, reads a JSON file and returns it, parsed, with keywords."
  [file]
  (json/parse-string (slurp file) true))

(def fips
  "A map of FIPS codes to their county names."
  (->> "fips.json"
       load-json
       :table
       :rows
       (into {})))
```

Now we can re-use `load-json` to load the UCR crime file.

```
(defn most-duis
  "Given a JSON filename of UCR crime data for a particular year, finds the
  counties with the most DUIs."
  [file]
  (->> file
       load-json
       (sort-by :driving_under_influence)
       (take-last 10)
       (map #(select-keys % [:driving_under_influence
                             :fips_county_code
                             :fips_state_code]))))
```

```
user=> (use 'scratch.crime :reload) (pprint (most-duis "2008.json"))
nil
({:fips_state_code "06",
  :fips_county_code "067",
  :driving_under_influence 8589}
 {:fips_state_code "48",
  :fips_county_code "201",
  :driving_under_influence 10432}
 {:fips_state_code "32",
  :fips_county_code "003",
  :driving_under_influence 10443}
 {:fips_state_code "06",
  :fips_county_code "065",
```

```
  :driving_under_influence 10814}
 {:fips_state_code "53",
  :fips_county_code "033",
  :driving_under_influence 11439}
 {:fips_state_code "06",
  :fips_county_code "071",
  :driving_under_influence 13983}
 {:fips_state_code "06",
  :fips_county_code "059",
  :driving_under_influence 17572}
 {:fips_state_code "06",
  :fips_county_code "073",
  :driving_under_influence 18562}
 {:fips_state_code "04",
  :fips_county_code "013",
  :driving_under_influence 26235}
 {:fips_state_code "06",
  :fips_county_code "037",
  :driving_under_influence 45056})
```

Almost there. We need to join together the state and county FIPS codes into a single string, because that's how `fips` represents the county code:

```
(defn fips-code
  "Given a county (a map with :fips_state_code and :fips_county_code keys),
   returns the five-digit FIPS code for the county, as a string."
  [county]
  (str (:fips_state_code county) (:fips_county_code county)))
```

Let's write a quick test in `test/scratch/crime_test.clj` to verify that function works correctly:

```
(ns scratch.crime-test
  (:require [clojure.test :refer :all]
            [scratch.crime :refer :all]))

(deftest fips-code-test
  (is (= "12345" (fips-code {:fips_state_code "12" :fips_county_code "345"}))))
```

```
aphyr@waterhouse:~/scratch$ lein test scratch.crime-test

lein test scratch.crime-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

Great. Note that `lein test some-namespace` runs only the tests in that particular namespace. For the last step, let's take the `most-duis` function and, using `fips` and `fips-code`, construct a map of county names to DUI reports.

```
(defn most-duis
  "Given a JSON filename of UCR crime data for a particular year, finds the
  counties with the most DUIs."
  [file]
  (->> file
       load-json
       (sort-by :driving_under_influence)
       (take-last 10)
       (map (fn [county]
              [(fips (fips-code county))
               (:driving_under_influence county)]))
       (into {})))
```

```
user=> (use 'scratch.crime :reload) (pprint (most-duis "2008.json"))
nil
{"CA, Orange" 17572,
 "CA, San Bernardino" 13983,
 "CA, Los Angeles" 45056,
 "CA, Riverside" 10814,
 "NV, Clark" 10443,
 "WA, King" 11439,
 "AZ, Maricopa" 26235,
 "CA, San Diego" 18562,
 "TX, Harris" 10432,
 "CA, Sacramento" 8589}
```

Our question is, at least in part, answered: Los Angeles and Maricopa counties, in California and Arizona, have the most reports of drunk driving out of any counties in the 2008 Uniform Crime Reporting database. These might be good counties for a PSA campaign. California has either lots of drunk drivers, or aggressive enforcement, or both! Remember, this only tells us about *reports* of crimes; not the crimes themselves. Numbers vary based on how the state enforces the laws!

```
(ns scratch.crime
  (:require [cheshire.core :as json]))

(defn load-json
  "Given a filename, reads a JSON file and returns it, parsed, with keywords."
  [file]
  (json/parse-string (slurp file) true))

(def fips
  "A map of FIPS codes to their county names."
  (->> "fips.json"
       load-json
       :table
       :rows
       (into {})))

(defn fips-code
  "Given a county (a map with :fips_state_code and :fips_county_code keys),
  returns the five-digit FIPS code for the county, as a string."
  [county]
  (str (:fips_state_code county) (:fips_county_code county)))

(defn most-duis
  "Given a JSON filename of UCR crime data for a particular year, finds the
  counties with the most DUIs."
  [file]
  (->> file
       load-json
       (sort-by :driving_under_influence)
       (take-last 10)
       (map (fn [county]
              [(fips (fips-code county))
               (:driving_under_influence county)]))
       (into {})))
```

Recap

In this chapter, we expanded beyond transient programs written in the REPL. We learned how *projects* combine static resources, code, and tests into a single package, and how projects can relate to one another through *dependencies*. We learned the basics of Clojure's namespace system, which isolates distinct chunks of code from one another, and how to include definitions

from one namespace in another via `require` and `use`. We learned how to write and run *tests* to verify our code's correctness, and how to move seamlessly between the repl and code in `.clj` files. We made use of Cheshire, a Clojure library published on Clojars, to parse JSON– a common data format. Finally, we brought together our knowledge of Clojure's basic grammar, immutable data structures, core functions, sequences, threading macros, and vars to explore a real-world problem.

Exercises

1. `most-duis` tells us about the raw number of reports, but doesn't account for differences in county population. One would naturally expect counties with more people to have more crime! Divide the `:driving_under_influence` of each county by its `:county_population` to find a *prevalence* of DUIs, and take the top ten counties based on prevalence. How should you handle counties with a population of zero?

2. How do the prevalence counties compare to the original counties? Expand most-duis to return vectors of `[county-name, prevalence, report-count, population]` What are the populations of the high-prevalence counties? Why do you suppose the data looks this way? If you were leading a public-health campaign to reduce drunk driving, would you target your intervention based on *report count* or *prevalence*? Why?

3. We can *generalize* the `most-duis` function to handle *any* type of crime. Write a function `most-prevalent` which takes a file and a field name, like `:arson`, and finds the counties where that field is most often reported, per capita.

4. Write a test to verify that `most-prevalent` is correct.

Chapter 8: Modeling

Previously: Logistics

Until this point in the book, we've dealt primarily in specific details: what an expression is, how math works, which functions apply to different data structures, and where code lives. But programming, like speaking a language, painting landscapes, or designing turbines, is about more than the *nuts and bolts* of the trade. It's knowing how to *combine* those parts into a cohesive whole–and this is a skill which is difficult to describe formally. In this part of the book, I'd like to work with you on an integrative tour of one particular problem: modeling a rocket in flight.

We're going to reinforce our concrete knowledge of the standard library by using maps, sequences, and math functions together. At the same time, we're going to practice how to represent a complex system; decomposing a problem into smaller parts, naming functions and variables, and writing tests.

So you want to go to space

First, we need a representation of a craft. The obvious properties for a rocket are its dry mass (how much it weighs without fuel), fuel mass, position, velocity, and time. We'll create a new file in our scratch project–`src/scratch/rocket.clj`–to talk about spacecraft.

For starters, let's pattern our craft after an Atlas V launch vehicle. We'll represent everything in SI units–kilograms, meters, newtons, etc. The Atlas V carries 627,105 lbs of LOX/RP-1 fuel, and a total mass of 334,500 kg gives only 50,050 kg of mass which *isn't* fuel. It develops 4152 kilonewtons of thrust and runs for 253 seconds, with a specific impulse (effectively, exhaust velocity) of 3.05 kilometers/sec. Real rockets develop varying amounts of thrust depending on the atmosphere, but we'll pretend it's constant in our simulation.

```
(defn atlas-v
 []
  {:dry-mass  50050
   :fuel-mass 284450
   :time 0
   :isp 3050
   :max-fuel-rate (/ 284450 253)
   :max-thrust 4.152e6})
```

How heavy is the craft?

```
(defn mass
  "The total mass of a craft."
  [craft]
  (+ (:dry-mass craft) (:fuel-mass craft)))
```

What about the position and velocity? We could represent them in Cartesian coordinates–x, y, and z–or we could choose spherical coordinates: a radius from the planet and angle from the pole and 0 degrees longitude. I've got a hunch that spherical coordinates will be easier for position, but accelerating the craft will be simplest in in x, y, and z terms. The center of the

planet is a natural choice for the coordinate system's origin (0, 0, 0). We'll choose z along the north pole, and x and y in the plane of the equator.

Let's define a space center where we launch from–let's say it's initially on the equator at y=0. To figure out the x coordinate, we'll need to know how far the space center is from the center of the earth. The earth's equatorial radius is ~6378 kilometers.

```
(def earth-equatorial-radius
 "Radius of the earth, in meters"
  6378137)
```

How fast is the surface moving? Well the earth's day is 86,400 seconds long,

```
(def earth-day
  "Length of an earth day, in seconds."
  86400)
```

which means a given point on the equator has to go 2 * pi * equatorial radius meters in earth-day seconds:

```
(def earth-equatorial-speed
  "How fast points on the equator move, relative to the center of the earth,
  in meters/sec."
  (/ (* 2 Math/PI earth-equatorial-radius)
     earth-day))
```

So our space center is on the equator (z=0), at y=0 by choice, which means x is the equatorial radius. Since the earth is spinning, the space center is moving at earth-equatorial-speed in the y direction–and not changing at all in x or z.

```
(def initial-space-center
  "The initial position and velocity of the launch facility"
  {:time     0
   :position {:x earth-equatorial-radius
              :y 0
              :z 0}
   :velocity {:x 0
              :y earth-equatorial-speed
              :z 0}})
```

`:position` and `:velocity` are both vectors, in the sense that they describe a position, or a direction, in terms of x, y, and z components. This is a *different* kind of vector than a Clojure vector, like `[1 2 3]`. We're actually representing these logical vectors as Clojure *maps*, with `:x`, `:y`, and `:z` keys, corresponding to the distance along the x, y, and z directions, from the center of the earth. Throughout this chapter, I'll mainly use the term *coordinates* to talk about these structures, to avoid confusion with Clojure vectors.

Now let's create a function which positions our craft on the launchpad at time 0. We'll just *merge* the spacecraft's with the initial space center, overwriting the craft's time and space coordinates.

```
(defn prepare
  "Prepares a craft for launch from an equatorial space center."
  [craft]
  (merge craft initial-space-center))
```

Forces

Gravity continually pulls the spacecraft towards the center of the Earth, accelerating it by 9.8 meters/second every second. To figure out what direction is towards the Earth, we'll need the angles of a spherical coordinate system. We'll use the trigonometric functions from java.lang.Math.

```
(defn magnitude
  "What's the radius of a given set of cartesian coordinates?"
  [c]
  ; By the Pythagorean theorem...
  (Math/sqrt (+ (Math/pow (:x c) 2)
                (Math/pow (:y c) 2)
                (Math/pow (:z c) 2))))

(defn cartesian->spherical
  "Converts a map of Cartesian coordinates :x, :y, and :z to spherical coordinates :r, :theta,
 and :phi."
  [c]
  (let [r (magnitude c)]
    {:r r
     :theta (Math/acos (/ (:z c) r))
     :phi   (Math/atan (/ (:y c) (:x c)))}))

(defn spherical->cartesian
  "Converts spherical to Cartesian coordinates."
  [c]
  {:x (* (:r c) (Math/sin (:theta c)) (Math/cos (:phi c)))
   :y (* (:r c) (Math/sin (:theta c)) (Math/sin (:phi c)))
   :z (* (:r c) (Math/cos (:phi c)))})
```

With those angles in mind, computing the gravitational acceleration is easy. We just take the spherical coordinates of the spacecraft, and replace the radius with the total force due to gravity. Then we can transform that spherical force back into Cartesian coordinates.

```
(def g "Acceleration of gravity in meters/s^2" -9.8)

(defn gravity-force
  "The force vector, each component in Newtons, due to gravity."
  [craft]
  ; Since force is mass times acceleration...
  (let [total-force (* g (mass craft))]
    (-> craft
        ; Now we'll take the craft's position
        :position
        ; in spherical coordinates,
        cartesian->spherical
        ; replace the radius with the gravitational force...
        (assoc :r total-force)
        ; and transform back to Cartesian-land
        spherical->cartesian)))
```

Rockets produce thrust by consuming fuel. Let's say the fuel consumption is always the maximum, until we run out:

```
(defn fuel-rate
  "How fast is fuel, in kilograms/second, consumed by the craft?"
  [craft]
  (if (pos? (:fuel-mass craft))
    (:max-fuel-rate craft)
    0))
```

Now that we know how much fuel is being consumed, we can compute the force the rocket engine develops. That force is simply the mass consumed per second times the exhaust velocity–which is the specific impulse `:isp`. We'll ignore atmospheric effects.

```
(defn thrust
  "How much force, in newtons, does the craft's rocket engines exert?"
  [craft]
  (* (fuel-rate craft) (:isp craft)))
```

Cool. What about the direction of thrust? Just for grins, let's keep the rocket pointing entirely along the x axis.

```
(defn engine-force
  "The force vector, each component in Newtons, due to the rocket engine."
  [craft]
  (let [t (thrust craft)]
    {:x t
     :y 0
     :z 0}))
```

The total force on a craft is just the sum of gravity and thrust. To sum these maps together, we'll need a way to sum the x, y, and z components independently. Clojure's `merge-with` function combines common fields in maps using a function, so this is surprisingly straightforward.

```
(defn total-force
  "Total force on a craft."
  [craft]
  (merge-with + (engine-force craft)
                (gravity-force craft)))
```

The acceleration of a craft, by Newton's second law, is force divided by mass. This one's a little trickier; given `{:x 1 :y 2 :z 4}` we want to apply a function–say, multiplication by a factor, to each number. Since maps are sequences of key/value pairs…

```
user=> (seq {:x 1 :y 2 :z 3})
([:z 3] [:y 2] [:x 1])
```

… and we can build up new maps out of key/value pairs using `into`…

```
user=> (into {} [[:x 4] [:y 5]])
{:x 4, :y 5}
```

… we can write a function `map-values` which works like `map`, but affects the values of a map data structure.

```
(defn map-values
  "Applies f to every value in the map m."
  [f m]
  (into {}
        (map (fn [pair]
               [(key pair) (f (val pair))])
             m)))
```

And that allows us to define a `scale` function which *scales* a set of coordinates by some factor:

```
(defn scale
  "Multiplies a map of x, y, and z coordinates by the given factor."
  [factor coordinates]
  (map-values (partial * factor) coordinates))
```

What's that `partial` thing? It's a function which *takes a function*, and some arguments, and *returns a new function*. What does the new function do? It calls the original function, with the arguments passed to `partial`, followed by any arguments passed to the new function. In short, `(partial * factor)` returns a function that takes any number, and multiplies it by `factor`.

So to divide each component of the force vector by the mass of the craft:

```
(defn acceleration
  "Total acceleration of a craft."
  [craft]
  (let [m (mass craft)]
    (scale (/ m) (total-force craft))))
```

Note that `(/ m)` returns 1/m. Our scale function can do double-duty as both multiplication and division.

With the acceleration and fuel consumption all figured out, we're ready to *apply those changes over time*. We'll write a function which takes the rocket at a particular time, and returns a version of it `dt` seconds later.

```
(defn step
  [craft dt]
  (assoc craft
         ; Time advances by dt seconds
         :t         (+ dt (:t craft))
         ; We burn some fuel
         :fuel-mass (- (:fuel-mass craft) (* dt (fuel-rate craft)))
         ; Our position changes based on our velocity
         :position  (merge-with + (:position craft)
                                  (scale dt (:velocity craft)))
         ; And our velocity changes based on our acceleration
         :velocity  (merge-with + (:velocity craft)
                                  (scale dt (acceleration craft)))))
```

OK. Let's save the `rocket.clj` file, load that code into the REPL, and fire it up.

```
user=> (use 'scratch.rocket :reload)
nil
```

`use` is like a shorthand for `(:require ... :refer :all)`. We're passing `:reload` because we want the REPL to re-read the file. Notice that in `ns` declarations, the namespace name `scratch.rocket` is *unquoted*–but when we call `use` or `require` at the repl, we quote the namespace name.

```
user=> (atlas-v)
{:dry-mass 50050, :fuel-mass 284450, :time 0, :isp 3050, :max-fuel-rate 284450/253, :max-
thrust 4152000.0}
```

Launch

Let's prepare the rocket. We'll use `pprint` to print it in a more readable form.

```
user=> (-> (atlas-v) prepare pprint)
{:velocity {:x 0, :y 463.8312116386399, :z 0},
 :position {:x 6378137, :y 0, :z 0},
 :dry-mass 50050,
 :fuel-mass 284450,
 :time 0,
```

```
 :isp 3050,
 :max-fuel-rate 284450/253,
 :max-thrust 4152000.0}
```

Great; there it is on the launchpad. Wow, even “standing still”, it’s moving at 463 meters/sec because of the earth’s rotation! That means *you and I* are flying through space at almost half a kilometer every second! Let’s step forward one second in time.

```
user=> (-> (atlas-v) prepare (step 1) pprint)

NullPointerException   clojure.lang.Numbers.ops (Numbers.java:942)
```

In evaluating this expression, Clojure reached a point where it could not continue, and aborted execution. We call this error an *exception*, and the process of aborting *throwing* the exception. Clojure backs up to the function which *called* the function that threw, then the function which called *that* function, and so on, all the way to the top-level expression. The REPL finally intercepts the exception, prints an error to the console, and stashes the exception object in a special variable `*e`.

In this case, we know that the exception in question was a `NullPointerException`, which occurs when a function received `nil` unexpectedly. This one came from `clojure.lang.Numbers.ops`, which suggests some sort of math was involved. Let’s use `pst` to find out where it came from.

```
user=> (pst *e)
NullPointerException
    clojure.lang.Numbers.ops (Numbers.java:942)
    clojure.lang.Numbers.add (Numbers.java:126)
    scratch.rocket/step (rocket.clj:125)
    user/eval1478 (NO_SOURCE_FILE:1)
    clojure.lang.Compiler.eval (Compiler.java:6619)
    clojure.lang.Compiler.eval (Compiler.java:6582)
    clojure.core/eval (core.clj:2852)
    clojure.main/repl/read-eval-print--6588/fn--6591 (main.clj:259)
    clojure.main/repl/read-eval-print--6588 (main.clj:259)
    clojure.main/repl/fn--6597 (main.clj:277)
    clojure.main/repl (main.clj:277)
    clojure.tools.nrepl.middleware.interruptible-eval/evaluate/
fn--589 (interruptible_eval.clj:56)
```

This is called a *stack trace*: the *stack* is the context of the program at each function call. It traces the path the computer took in evaluating the expression, from the bottom to the top. At the bottom is the REPL, and Clojure compiler. Our code begins at `user/eval1478`–that’s the compiler’s name for the expression we just typed. That function called `scratch.rocket/step`, which in turn called `Numbers.add`, and that called `Numbers.ops`. Let’s start by looking at the last function *we* wrote before calling into Clojure’s standard library: the `step` function, in `rocket.clj`, on line `125`.

```
123  (assoc craft
124         ; Time advances by dt seconds
125         :t         (+ dt (:t craft))
```

Ah; we named the time field `:time` earlier, not `:t`. Let’s replace `:t` with `:time`, save the file, and reload.

```
user=> (use 'scratch.rocket :reload)
```

```
nil
user=> (-> (atlas-v) prepare (step 1) pprint)
{:velocity {:x 0.45154055666826215, :y 463.8312116386399, :z -9.8},
 :position {:x 6378137, :y 463.8312116386399, :z 0},
 :dry-mass 50050,
 :fuel-mass 71681400/253,
 :time 1,
 :isp 3050,
 :max-fuel-rate 284450/253,
 :max-thrust 4152000.0}
```

Look at that! Our position is unchanged (because our velocity was zero), but our *velocity* has shifted. We're now moving… wait, -9.8 meters per second *south*? That can't be right. Gravity points *down*, not sideways. Something must be wrong with our spherical coordinate system. Let's write a test in `test/scratch/rocket_test.clj` to explore.

```
(ns scratch.rocket-test
  (:require [clojure.test :refer :all]
            [scratch.rocket :refer :all]))

(deftest spherical-coordinate-test
  (let [pos {:x 1 :y 2 :z 3}]
    (testing "roundtrip"
      (is (= pos (-> pos cartesian->spherical spherical->cartesian))))))
```

```
aphyr@waterhouse:~/scratch$ lein test

lein test scratch.core-test

lein test scratch.rocket-test

lein test :only scratch.rocket-test/spherical-coordinate-test

FAIL in (spherical-coordinate-test) (rocket_test.clj:8)
roundtrip
expected: (= pos (-> pos cartesian->spherical spherical->cartesian))
  actual: (not (= {:z 3, :y 2, :x 1} {:x 1.0, :y 1.9999999999999996, :z 1.6733200530681513}))

Ran 2 tests containing 4 assertions.
1 failures, 0 errors.
Tests failed.
```

Definitely wrong. Looks like something to do with the z coordinate, since x and y look OK. Let's try testing a point on the north pole:

```
(deftest spherical-coordinate-test
  (testing "spherical->cartesian"
    (is (= (spherical->cartesian {:r 2
                                  :phi 0
                                  :theta 0})
           {:x 0.0 :y 0.0 :z 2.0})))

  (testing "roundtrip"
    (let [pos {:x 1.0 :y 2.0 :z 3.0}]
      (is (= pos (-> pos cartesian->spherical spherical->cartesian))))))
```

That checks out OK. Let's try some values in the repl.

```
user=> (cartesian->spherical {:x 0.00001 :y 0.00001 :z 2.0})
{:r 2.00000000005, :theta 7.071068104411588E-6, :phi 0.7853981633974483}
user=> (cartesian->spherical {:x 1 :y 2 :z 3})
{:r 3.7416573867739413, :theta 0.6405223126794245, :phi 1.1071487177940904}
user=> (spherical->cartesian (cartesian->spherical {:x 1 :y 2 :z 3}))
{:x 1.0, :y 1.9999999999999996, :z 1.6733200530681513}
```

```
user=> (cartesian->spherical {:x 1 :y 2 :z 0})
{:r 2.23606797749979, :theta 1.5707963267948966, :phi 1.1071487177940904}
user=> (cartesian->spherical {:x 1 :y 1 :z 0})
{:r 1.4142135623730951, :theta 1.5707963267948966, :phi 0.7853981633974483}
```

Oh, wait, that looks odd. `{:x 1 :y 1 :z 0}` is on the equator: phi–the angle from the pole–should be pi/2 or ~1.57, and theta–the angle around the equator–should be pi/4 or 0.78. Those coordinates are reversed! Double-checking our formulas with Wolfram MathWorld shows that we mixed up phi and theta! Let's redefine `cartesian->polar` correctly.

```
(defn cartesian->spherical
  "Converts a map of Cartesian coordinates :x, :y, and :z to spherical
  coordinates :r, :theta, and :phi."
  [c]
  (let [r (Math/sqrt (+ (Math/pow (:x c) 2)
                        (Math/pow (:y c) 2)
                        (Math/pow (:z c) 2)))]
    {:r     r
     :phi   (Math/acos (/ (:z c) r))
     :theta (Math/atan (/ (:y c) (:x c)))}))
```

```
aphyr@waterhouse:~/scratch$ lein test

lein test scratch.core-test

lein test scratch.rocket-test

Ran 2 tests containing 5 assertions.
0 failures, 0 errors.
```

Great. Now let's check the rocket trajectory again.

```
user=> (-> (atlas-v) prepare (step 1) pprint)
{:velocity
 {:x 0.45154055666826204,
  :y 463.8312116386399,
  :z -6.000769315822031E-16},
 :position {:x 6378137, :y 463.8312116386399, :z 0},
 :dry-mass 50050,
 :fuel-mass 71681400/253,
 :time 1,
 :isp 3050,
 :max-fuel-rate 284450/253,
 :max-thrust 4152000.0}
```

This time, our velocity is increasing in the +x direction, at half a meter per second. We have liftoff!

Flight

We have a function that can move the rocket forward by one small step of time, but we'd like to understand the rocket's trajectory as a *whole*; to see *all* positions it will take. We'll use *iterate* to construct a lazy, infinite sequence of rocket states, each one constructed by stepping forward from the last.

```
(defn trajectory
  [dt craft]
  "Returns all future states of the craft, at dt-second intervals."
  (iterate #(step % 1) craft))
```

```
user=> (->> (atlas-v) prepare (trajectory 1) (take 3) pprint)
({:velocity {:x 0, :y 463.8312116386399, :z 0},
  :position {:x 6378137, :y 0, :z 0},
  :dry-mass 50050,
  :fuel-mass 284450,
  :time 0,
  :isp 3050,
  :max-fuel-rate 284450/253,
  :max-thrust 4152000.0}
 {:velocity
  {:x 0.45154055666826204,
   :y 463.8312116386399,
   :z -6.000769315822031E-16},
  :position {:x 6378137, :y 463.8312116386399, :z 0},
  :dry-mass 50050,
  :fuel-mass 71681400/253,
  :time 1,
  :isp 3050,
  :max-fuel-rate 284450/253,
  :max-thrust 4152000.0}
 {:velocity
  {:x 0.9376544222659078,
   :y 463.83049896253056,
   :z -1.2001538631644O6E-15},
  :position
  {:x 6378137.451540557,
   :y 927.6624232772798,
   :z -6.000769315822031E-16},
  :dry-mass 50050,
  :fuel-mass 71396950/253,
  :time 2,
  :isp 3050,
  :max-fuel-rate 284450/253,
  :max-thrust 4152000.0})
```

Notice that each map is like a frame of a movie, playing at one frame per second. We can make the simulation more or less accurate by raising or lowering the framerate–adjusting the parameter fed to `trajectory`. For now, though, we'll stick with one-second intervals.

How high above the surface is the rocket?

```
(defn altitude
  "The height above the surface of the equator, in meters."
  [craft]
  (-> craft
      :position
      cartesian->spherical
      :r
      (- earth-equatorial-radius)))
```

Now we can explore the rocket's path as a series of altitudes over time:

```
user=> (->> (atlas-v) prepare (trajectory 1) (map altitude) (take 10) pprint)
(0.0
 0.016865378245711327
 0.519002066925168
 1.540983198210597
 3.117615718394518
 5.283942770212889
 8.075246102176607
 11.52704851794988
 15.675116359256208
 20.555462017655373)
```

The million dollar question, though, is whether the rocket breaks orbit.

```
(defn above-ground?
  "Is the craft at or above the surface?"
  [craft]
  (<= 0 (altitude craft)))

(defn flight
  "The above-ground portion of a trajectory."
  [trajectory]
  (take-while above-ground? trajectory))

(defn crashed?
  "Does this trajectory crash into the surface before 100 hours are up?"
  [trajectory]
  (let [time-limit (* 100 3600)] ; 1 hour
    (not (every? above-ground?
                 (take-while #(<= (:time %) time-limit) trajectory)))))

(defn crash-time
  "Given a trajectory, returns the time the rocket impacted the ground."
  [trajectory]
  (:time (last (flight trajectory))))

(defn apoapsis
  "The highest altitude achieved during a trajectory."
  [trajectory]
  (apply max (map altitude trajectory)))

(defn apoapsis-time
  "The time of apoapsis"
  [trajectory]
  (:time (apply max-key altitude (flight trajectory))))
```

If the rocket goes below ground, we know it crashed. If the rocket stays in orbit, the trajectory will never end. That makes it a bit tricky to tell whether the rocket is in a stable orbit or not, because we can't ask about every element, or the last element, of an infinite sequence: it'll take infinite time to evaluate. Instead, we'll assume that the rocket *should* crash within the first, say, 100 hours; if it makes it past that point, we'll assume it made orbit successfully. With these functions in hand, we'll write a test in `test/scratch/rocket_test.clj` to see whether or not the launch is successful:

```
(deftest makes-orbit
  (let [trajectory (->> (atlas-v)
                        prepare
                        (trajectory 1))]

    (when (crashed? trajectory)
      (println "Crashed at" (crash-time trajectory) "seconds")
      (println "Maximum altitude" (apoapsis trajectory)
               "meters at"        (apoapsis-time trajectory) "seconds"))

    ; Assert that the rocket eventually made it to orbit.
    (is (not (crashed? trajectory)))))
```

```
aphyr@waterhouse:~/scratch$ lein test scratch.rocket-test

lein test scratch.rocket-test
Crashed at 982 seconds
Maximum altitude 753838.039645385 meters at 532 seconds

lein test :only scratch.rocket-test/makes-orbit

FAIL in (makes-orbit) (rocket_test.clj:26)
expected: (not (crashed? trajectory))
  actual: (not (not true))
```

```
Ran 2 tests containing 3 assertions.
1 failures, 0 errors.
Tests failed.
```

We made it to an altitude of 750 kilometers, and crashed 982 seconds after launch. We're gonna need a bigger boat.

Stage II

The Atlas V isn't big enough to make it into orbit on its own. It carries a second stage, the Centaur), which is much smaller and uses more efficient engines.

```
(defn centaur
  "The upper rocket stage.
  http://en.wikipedia.org/wiki/Centaur_(rocket_stage)
  http://www.astronautix.com/stages/cenaurde.htm"
  []
  {:dry-mass  2361
   :fuel-mass 13897
   :isp       4354
   :max-fuel-rate (/ 13897 470)})
```

The Centaur lives inside the Atlas V main stage. We'll re-write `atlas-v` to take an *argument*: its next stage.

```
(defn atlas-v
  "The full launch vehicle. http://en.wikipedia.org/wiki/Atlas_V"
  [next-stage]
  {:dry-mass  50050
   :fuel-mass 284450
   :isp 3050
   :max-fuel-rate (/ 284450 253)
   :next-stage next-stage})
```

Now, in our tests, we'll construct the rocket like so:

```
  (let [trajectory (->> (atlas-v (centaur))
                        prepare
                        (trajectory 1))]
```

When we exhaust the fuel reserves of the primary stage, we'll de-couple the main booster from the Centaur. In terms of our simulation, the Atlas V will be *replaced* by its next stage, the Centaur. We'll write a function `stage` which separates the vehicles when ready:

```
(defn stage
  "When fuel reserves are exhausted, separate stages. Otherwise, return craft
  unchanged."
  [craft]
  (cond
    ; Still fuel left
    (pos? (:fuel-mass craft))
    craft

    ; No remaining stages
    (nil? (:next-stage craft))
    craft

    ; Stage!
    :else
    (merge (:next-stage craft)
```

```
           (select-keys craft [:time :position :velocity])))))
```

We're using `cond` to handle three distinct cases: where there's fuel remaining in the craft, where there is no stage to separate, and when we're ready for stage separation. Separation is easy: we simply return the next stage of the current craft, with the current craft's time, position, and velocity merged in.

Finally, we'll have to update our `step` function to take into account the possibility of stage separation.

```
(defn step
  [craft dt]
  (let [craft (stage craft)]
    (assoc craft
           ; Time advances by dt seconds
           :time      (+ dt (:time craft))
           ; We burn some fuel
           :fuel-mass (- (:fuel-mass craft) (* dt (fuel-rate craft)))
           ; Our position changes based on our velocity
           :position  (merge-with + (:position craft)
                                  (scale dt (:velocity craft)))
           ; And our velocity changes based on our acceleration
           :velocity  (merge-with + (:velocity craft)
                                  (scale dt (acceleration craft))))))
```

Same as before, only now we call `stage` prior to the physics simulation. Let's try a launch.

```
aphyr@waterhouse:~/scratch$ lein test scratch.rocket-test

lein test scratch.rocket-test
Crashed at 2415 seconds
Maximum altitude 4598444.289945109 meters at 1446 seconds

lein test :only scratch.rocket-test/makes-orbit

FAIL in (makes-orbit) (rocket_test.clj:27)
expected: (not (crashed? trajectory))
  actual: (not (not true))

Ran 2 tests containing 3 assertions.
1 failures, 0 errors.
Tests failed.
```

Still crashed–but we increased our apoapsis from 750 kilometers to 4,598 kilometers. That's plenty high, but we're still not making orbit. Why? Because we're going straight up, then straight back down. To orbit, we need to move *sideways*, around the earth.

Orbital insertion

Our spacecraft is shooting upwards, but in order to remain in orbit around the earth, it has to execute a *second* burn: an orbital injection maneuver. That injection maneuver is also called a *circularization burn* because it turns the orbit from an ascending parabola into a circle (or something roughly like it). We don't need to be precise about circularization–any trajectory that doesn't hit the planet will suffice. All we have to do is burn towards the horizon, once we get high enough.

To do that, we'll need to enhance the rocket's control software. We assumed that the rocket would always thrust in the +x direction; but now we'll need to thrust in multiple directions. We'll

break up the engine force into two parts: `thrust` (how hard the rocket motor pushes) and `orientation` (which determines the direction the rocket is pointing.)

```
(defn unit-vector
  "Scales coordinates to magnitude 1."
  [coordinates]
  (scale (/ (magnitude coordinates)) coordinates))

(defn engine-force
  "The force vector, each component in Newtons, due to the rocket engine."
  [craft]
  (scale (thrust craft) (unit-vector (orientation craft))))
```

We're taking the orientation of the craft–some coordinates–and scaling it to be of length one with `unit-vector`. Then we're scaling the orientation vector by the thrust, returning a *thrust vector*.

As we go back and redefine parts of the program, you might see an error like

```
Exception in thread "main" java.lang.RuntimeException: Unable to resolve symbol: unit-vector
 in this context, compiling:(scratch/rocket.clj:69:11)
    at clojure.lang.Compiler.analyze(Compiler.java:6380)
    at clojure.lang.Compiler.analyze(Compiler.java:6322)
```

This is a stack trace from the Clojure compiler. It indicates that in `scratch/rocket.clj`, on line `69`, column `11`, we used the symbol `unit-vector`–but it didn't have a meaning at that point in the program. Perhaps `unit-vector` is defined *below* that line. There are two ways to solve this.

1. Organize your functions so that the simple ones come first, and those that depend on them come later. Read this way, namespaces tell a story, progressing from smaller to bigger, more complex problems.

2. Sometimes, ordering functions this way is impossible, or would put related ideas too far apart. In this case you can `(declare unit-vector)` near the top of the namespace. This tells Clojure that `unit-vector` isn't defined *yet*, but it'll come later.

Now that we've broken up `engine-force` into `thrust` and `orientation`, we have to control the thrust properly for our two burns. We'll start by defining the times for the initial ascent and circularization burn, expressed as vectors of start and end times, in seconds.

```
(def ascent
  "The start and end times for the ascent burn."
  [0 3000])

(def circularization
  "The start and end times for the circularization burn."
  [4000 1000])
```

Now we'll change the thrust by adjusting the rate of fuel consumption. Instead of burning at maximum until running out of fuel, we'll execute two distinct burns.

```
(defn fuel-rate
  "How fast is fuel, in kilograms/second, consumed by the craft?"
  [craft]
  (cond
    ; Out of fuel
    (<= (:fuel-mass craft) 0)
```

```
    0

    ; Ascent burn
    (<= (first ascent) (:time craft) (last ascent))
    (:max-fuel-rate craft)

    ; Circularization burn
    (<= (first circularization) (:time craft) (last circularization))
    (:max-fuel-rate craft)

    ; Shut down engines otherwise
    :else 0))
```

We're using `cond` to express four distinct possibilities: that we've run out of fuel, executing either of the two burns, or resting with the engines shut down. Because the comparison function `<=` takes any number of arguments and asserts that they occur in order, expressing intervals like "the time is between the first and last times in the ascent" is easy.

Finally, we need to determine the *direction* to burn in. This one's gonna require some tricky linear algebra. You don't need to worry about the specifics here–the goal is to find out what direction the rocket should burn to fly towards the horizon, in a circle around the planet. We're doing that by taking the rocket's velocity vector, and *flattening out* the velocity towards or away from the planet. All that's left is the direction the rocket is flying *around* the earth.

```
(defn dot-product
  "Finds the inner product of two x, y, z coordinate maps.
  See http://en.wikipedia.org/wiki/Dot_product."
  [c1 c2]
  (+ (* (:x c1) (:x c2))
     (* (:y c1) (:y c2))
     (* (:z c1) (:z c2))))

(defn projection
  "The component of coordinate map a in the direction of coordinate map b.
  See http://en.wikipedia.org/wiki/Vector_projection."
  [a b]
  (let [b (unit-vector b)]
    (scale (dot-product a b) b)))

(defn rejection
  "The component of coordinate map a *not* in the direction of coordinate map
  b."
  [a b]
  (let [a' (projection a b)]
    {:x (- (:x a) (:x a'))
     :y (- (:y a) (:y a'))
     :z (- (:z a) (:z a'))}))
```

With the mathematical underpinnings ready, we'll define the orientation control software:

```
(defn orientation
  "What direction is the craft pointing?"
  [craft]
  (cond
    ; Initially, point along the *position* vector of the craft--that is
    ; to say, straight up, away from the earth.
    (<= (first ascent) (:time craft) (last ascent))
    (:position craft)

    ; During the circularization burn, we want to burn *sideways*, in the
    ; direction of the orbit. We'll find the component of our velocity
    ; which is aligned with our position vector (that is to say, the vertical
    ; velocity), and subtract the vertical component. All that's left is the
    ; *horizontal* part of our velocity.
```

```
    (<= (first circularization) (:time craft) (last circularization))
    (rejection (:velocity craft) (:position craft))

    ; Otherwise, just point straight ahead.
    :else (:velocity craft)))
```

For the ascent burn, we'll push straight away from the planet. For circularization, we use the `rejection` function to find the part of the velocity around the planet, and thrust in that direction. By default, we'll leave the rocket pointing in the direction of travel.

With these changes made, the rocket should execute two burns. Let's re-run the tests and see.

```
aphyr@waterhouse:~/scratch$ lein test scratch.rocket-test

lein test scratch.rocket-test

Ran 2 tests containing 3 assertions.
0 failures, 0 errors.
```

We finally did it! We're *rocket scientists*!

Review

```
(ns scratch.rocket)

;; Linear algebra for {:x 1 :y 2 :z 3} coordinate vectors.

(defn map-values
  "Applies f to every value in the map m."
  [f m]
  (into {}
        (map (fn [pair]
               [(key pair) (f (val pair))])
             m)))

(defn magnitude
  "What's the radius of a given set of cartesian coordinates?"
  [c]
  ; By the Pythagorean theorem...
  (Math/sqrt (+ (Math/pow (:x c) 2)
                (Math/pow (:y c) 2)
                (Math/pow (:z c) 2))))

(defn scale
  "Multiplies a map of x, y, and z coordinates by the given factor."
  [factor coordinates]
  (map-values (partial * factor) coordinates))

(defn unit-vector
  "Scales coordinates to magnitude 1."
  [coordinates]
  (scale (/ (magnitude coordinates)) coordinates))

(defn dot-product
  "Finds the inner product of two x, y, z coordinate maps. See
  http://en.wikipedia.org/wiki/Dot_product"
  [c1 c2]
  (+ (* (:x c1) (:x c2))
     (* (:y c1) (:y c2))
     (* (:z c1) (:z c2))))

(defn projection
  "The component of coordinate map a in the direction of coordinate map b.
  See http://en.wikipedia.org/wiki/Vector_projection."
  [a b]
```

```
  (let [b (unit-vector b)]
    (scale (dot-product a b) b)))

(defn rejection
  "The component of coordinate map a *not* in the direction of coordinate map
  b."
  [a b]
  (let [a' (projection a b)]
    {:x (- (:x a) (:x a'))
     :y (- (:y a) (:y a'))
     :z (- (:z a) (:z a'))}))

;; Coordinate conversion

(defn cartesian->spherical
  "Converts a map of Cartesian coordinates :x, :y, and :z to spherical
  coordinates :r, :theta, and :phi."
  [c]
  (let [r (magnitude c)]
    {:r     r
     :phi   (Math/acos (/ (:z c) r))
     :theta (Math/atan (/ (:y c) (:x c)))}))

(defn spherical->cartesian
  "Converts spherical to Cartesian coordinates."
  [c]
  {:x (* (:r c) (Math/cos (:theta c)) (Math/sin (:phi c)))
   :y (* (:r c) (Math/sin (:theta c)) (Math/sin (:phi c)))
   :z (* (:r c) (Math/cos (:phi c)))})

;; The earth

(def earth-equatorial-radius
  "Radius of the earth, in meters"
  6378137)

(def earth-day
  "Length of an earth day, in seconds."
  86400)

(def earth-equatorial-speed
  "How fast points on the equator move, relative to the center of the earth, in
  meters/sec."
  (/ (* 2 Math/PI earth-equatorial-radius)
     earth-day))

(def g "Acceleration of gravity in meters/s^2" -9.8)

(def initial-space-center
  "The initial position and velocity of the launch facility"
  {:time     0
   :position {:x earth-equatorial-radius
              :y 0
              :z 0}
   :velocity {:x 0
              :y earth-equatorial-speed
              :z 0}})


;; Craft

(defn centaur
  "The upper rocket stage.
  http://en.wikipedia.org/wiki/Centaur_(rocket_stage)
  http://www.astronautix.com/stages/cenaurde.htm"
  []
  {:dry-mass  2361
   :fuel-mass 13897
   :isp       4354
```

```
   :max-fuel-rate (/ 13897 470)})

(defn atlas-v
  "The full launch vehicle. http://en.wikipedia.org/wiki/Atlas_V"
  [next-stage]
  {:dry-mass  50050
   :fuel-mass 284450
   :isp 3050
   :max-fuel-rate (/ 284450 253)
   :next-stage next-stage})

;; Flight control

(def ascent
  "The start and end times for the ascent burn."
  [0 300])

(def circularization
  "The start and end times for the circularization burn."
  [400 1000])

(defn orientation
  "What direction is the craft pointing?"
  [craft]
  (cond
    ; Initially, point along the *position* vector of the craft--that is
    ; to say, straight up, away from the earth.
    (<= (first ascent) (:time craft) (last ascent))
    (:position craft)

    ; During the circularization burn, we want to burn *sideways*, in the
    ; direction of the orbit. We'll find the component of our velocity
    ; which is aligned with our position vector (that is to say, the vertical
    ; velocity), and subtract the vertical component. All that's left is the
    ; *horizontal* part of our velocity.
    (<= (first circularization) (:time craft) (last circularization))
    (rejection (:velocity craft) (:position craft))

    ; Otherwise, just point straight ahead.
    :else (:velocity craft)))

(defn fuel-rate
  "How fast is fuel, in kilograms/second, consumed by the craft?"
  [craft]
  (cond
    ; Out of fuel
    (<= (:fuel-mass craft) 0)
    0

    ; Ascent burn
    (<= (first ascent) (:time craft) (last ascent))
    (:max-fuel-rate craft)

    ; Circularization burn
    (<= (first circularization) (:time craft) (last circularization))
    (:max-fuel-rate craft)

    ; Shut down engines otherwise
    :else 0))

(defn stage
  "When fuel reserves are exhausted, separate stages. Otherwise, return craft
  unchanged."
  [craft]
  (cond
    ; Still fuel left
    (pos? (:fuel-mass craft))
    craft
```

```
    ; No remaining stages
    (nil? (:next-stage craft))
    craft

    ; Stage!
    :else
    (merge (:next-stage craft)
           (select-keys craft [:time :position :velocity])))))

;; Dynamics

(defn thrust
  "How much force, in newtons, does the craft's rocket engines exert?"
  [craft]
  (* (fuel-rate craft) (:isp craft)))

(defn mass
  "The total mass of a craft."
  [craft]
  (+ (:dry-mass craft) (:fuel-mass craft)))

(defn gravity-force
  "The force vector, each component in Newtons, due to gravity."
  [craft]
  ; Since force is mass times acceleration...
  (let [total-force (* g (mass craft))]
    (-> craft
        ; Now we'll take the craft's position
        :position
        ; in spherical coordinates,
        cartesian->spherical
        ; replace the radius with the gravitational force...
        (assoc :r total-force)
        ; and transform back to Cartesian-land
        spherical->cartesian)))

(declare altitude)

(defn engine-force
  "The force vector, each component in Newtons, due to the rocket engine."
  [craft]
;  Debugging; useful for working through trajectories in detail.
;  (println craft)
;  (println "t   " (:time craft) "alt" (altitude craft) "thrust" (thrust craft))
;  (println "fuel" (:fuel-mass craft))
;  (println "vel " (:velocity craft))
;  (println "ori " (unit-vector (orientation craft)))
  (scale (thrust craft) (unit-vector (orientation craft))))

(defn total-force
  "Total force on a craft."
  [craft]
  (merge-with + (engine-force craft)
              (gravity-force craft)))

(defn acceleration
  "Total acceleration of a craft."
  [craft]
  (let [m (mass craft)]
    (scale (/ m) (total-force craft))))

(defn step
  [craft dt]
  (let [craft (stage craft)]
    (assoc craft
           ; Time advances by dt seconds
           :time      (+ dt (:time craft))
           ; We burn some fuel
           :fuel-mass (- (:fuel-mass craft) (* dt (fuel-rate craft)))
```

```
           ; Our position changes based on our velocity
           :position  (merge-with + (:position craft)
                                  (scale dt (:velocity craft)))
           ; And our velocity changes based on our acceleration
           :velocity  (merge-with + (:velocity craft)
                                  (scale dt (acceleration craft))))))

;; Launch and flight

(defn prepare
  "Prepares a craft for launch from an equatorial space center."
  [craft]
  (merge craft initial-space-center))

(defn trajectory
  [dt craft]
  "Returns all future states of the craft, at dt-second intervals."
  (iterate #(step % 1) craft))

;; Analyzing trajectories

(defn altitude
  "The height above the surface of the equator, in meters."
  [craft]
  (-> craft
      :position
      cartesian->spherical
      :r
      (- earth-equatorial-radius)))

(defn above-ground?
  "Is the craft at or above the surface?"
  [craft]
  (<= 0 (altitude craft)))

(defn flight
  "The above-ground portion of a trajectory."
  [trajectory]
  (take-while above-ground? trajectory))

(defn crashed?
  "Does this trajectory crash into the surface before 10 hours are up?"
  [trajectory]
  (let [time-limit (* 10 3600)] ; 10 hours
    (not (every? above-ground?
                 (take-while #(<= (:time %) time-limit) trajectory)))))

(defn crash-time
  "Given a trajectory, returns the time the rocket impacted the ground."
  [trajectory]
  (:time (last (flight trajectory))))

(defn apoapsis
  "The highest altitude achieved during a trajectory."
  [trajectory]
  (apply max (map altitude (flight trajectory))))

(defn apoapsis-time
  "The time of apoapsis"
  [trajectory]
  (:time (apply max-key altitude (flight trajectory))))
```

As written here, our first non-trivial program tells a story–though a *different* one than the process of exploration and refinement that brought the rocket to orbit. It builds from small, abstract ideas: linear algebra and coordinates; physical constants describing the universe for the simulation; and the basic outline of the spacecraft. Then we define the software controlling the

rocket; the times for the burns, how much to thrust, in what direction, and when to separate stages. Using those control functions, we build a *physics engine* including gravity and thrust forces, and use Newton's second law to build a basic Euler Method solver. Finally, we analyze the trajectories the solver produces to answer key questions: how high, how long, and did it explode?

We used Clojure's immutable data structures–mostly maps–to represent the state of the universe, and defined *pure functions* to interpret those states and construct new ones. Using `iterate`, we projected a single state forward into an infinite timeline of the future–evaluated as demanded by the analysis functions. Though we pay a performance penalty, immutable data structures, pure functions, and lazy evaluation make simulating complex systems easier to reason about.

Had we written this simulation in a different language, different techniques might have come into play. In Java, C++, or Ruby, we would have defined a hierarchy of datatypes called *classes*, each one representing a small piece of state. We might define a `Craft` type, and created subtypes `Atlas` and `Centaur`. We'd create a `Coordinate` type, subdivided into `Cartesian` and `Spherical`, and so on. The types add complexity and rigidity, but also prevent mis-spellings, and can prevent us from interpreting, say, coordinates as craft or vice-versa.

To move the system forward in a language emphasizing *mutable* data structures, we would have updated the time and coordinates of a single craft in-place. This introduces additional complexity, because many of the changes we made depended on the current values of the craft. To ensure the correct ordering of calculations, we'd scatter temporary variables and explicit copies throughout the code, ensuring that functions didn't see inconsistent pictures of the craft state. The mutable approach would likely be faster, but would still demand some copying of data, and sacrifice clarity.

More *imperative* languages place less emphasis on laziness, and make it harder to express ideas like `map` and `take`. We might have simulated the trajectory for some fixed time, constructing a history of all the intermediate results we needed, then analyzed it by moving explicitly from slot to slot in that history, checking if the craft had crashed, and so on.

Across all these languages, though, some ideas remain the same. We solve big problems by breaking them up into smaller ones. We use data structures to represent the state of the system, and functions to alter that state. Comments and docstrings clarify the *story* of the code, making it readable to others. Tests ensure the software is correct, and allow us to work piecewise towards a solution.

Exercises

1. We know the spacecraft reached orbit, but we have no idea what that orbit *looks* like. Since the trajectory is infinite in length, we can't ask about the *entire* history using `max`–but we know that all orbits have a high and low point. At the highest point, the difference between successive altitudes changes from increasing to decreasing, and at the lowest point, the difference between successive altitudes changes from decreasing to increasing. Using this technique, refine the `apoapsis` function to find the highest point using that *inflection* in

altitudes–and write a corresponding `periapsis` function that finds the lowest point in the orbit. Display both periapsis and apoapsis in the test suite.

2. We assumed the force of gravity resulted in a constant 9.8 meter/second/second acceleration towards the earth, but in the real world, gravity falls off with the inverse square law. Using the mass of the earth, mass of the spacecraft, and Newton's constant, refine the gravitational force used in this simulation to take Newton's law into account. How does this affect the apoapsis?

3. We ignored the atmosphere, which exerts drag on the craft as it moves through the air. Write a basic air-density function which falls off with altitude. Make some educated guesses as to how much drag a real rocket experiences, and assume that the drag force is proportional to the square of the rocket's velocity. Can your rocket still reach orbit?

4. Notice that the periapsis and apoapsis of the rocket are *different*. By executing the circularization burn carefully, can you make them the same–achieving a perfectly circular orbit? One way to do this is to pick an orbital altitude and velocity of a known satellite–say, the International Space Station–and write the control software to match that velocity at that altitude.

Chapter 9: Debugging

Writing software can be an exercise in frustration. Useless error messages, difficult-to-reproduce bugs, missing stacktrace information, obscure functions without documentation, and unmaintained libraries all stand in our way. As software engineers, our most useful skill isn't so much *knowing how to solve a problem* as *knowing how to explore a problem that we haven't seen before*. Experience is important, but even experienced engineers face unfamiliar bugs every day. When a problem doesn't bear a resemblance to anything we've seen before, we fall back on *general cognitive strategies* to explore–and ultimately solve–the problem.

There's an excellent book by the mathematician George Polya: How to Solve It, which tries to catalogue how successful mathematicians approach unfamiliar problems. When I catch myself banging my head against a problem for more than a few minutes, I try to back up and consider his principles. Sometimes, just taking the time to slow down and reflect can get me out of a rut.

1. Understand the problem.
2. Devise a plan.
3. Carry out the plan
4. Look back

Seems easy enough, right? Let's go a little deeper.

Understanding the problem

Well *obviously* there's a problem, right? The program failed to compile, or a test spat out bizarre numbers, or you hit an unexpected exception. But try to dig a little deeper than that. Just having a careful description of the problem can make the solution obvious.

> Our audit program detected that users can double-withdraw cash from their accounts.

What does your program do? Chances are your program is large and complex, so try to *isolate* the problem as much as possible. Find *preconditions* where the error holds.

> The problem occurs after multiple transfers between accounts.

Identify specific lines of code from the stacktrace that are involved, specific data that's being passed around. Can you find a particular function that's misbehaving?

> The balance transfer function sometimes doesn't increase or decrease the account values correctly.

What are that function's inputs and outputs? Are the inputs what you expected? What did you expect the result to be, given those arguments? It's not enough to know "it doesn't work"–you

need to know exactly what *should* have happened. Try to find conditions where the function works correctly, so you can map out the boundaries of the problem.

> Trying to transfer $100 from A to B works as expected, as does a transfer of $50 from B to A. Running a million random transfers between accounts, sequentially, results in correct balances. The problem only seems to happen in production.

If your function–or functions it calls–uses mutable state, like an agent, atom, or ref, the value of those references matters too. This is why you should avoid mutable state wherever possible: each mutable variable introduces another dimension of possible behaviors for your program. Print out those values when they're read, and after they're written, to get a description of what the function is actually doing. I am a huge believer in sprinkling `(prn x)` throughout one's code to print how state evolves when the program runs.

> Each balance is stored in a separate atom. When two transfers happen at the same time involving the same accounts, the new value of one or both atoms may not reflect the transfer correctly.

Look for *invariants*: properties that should always be true of a program. Devise a test to look for where those invariants are broken. Consider each individual step of the program: does it preserve all the invariants you need? If it doesn't, what ensures those invariants are restored correctly?

> The total amount of money in the system should be constant–but sometimes changes!

Draw diagrams, and invent a notation to talk about the problem. If you're accessing fields in a vector, try drawing the vector as a set of boxes, and drawing the fields it accesses, step by step on paper. If you're manipulating a tree, draw one! Figure out a way to write down the state of the system: in letters, numbers, arrows, graphs, whatever you can dream up.

```
Transferring $5 from A to B in transaction 1, and $5 from B to A in transaction 2:

Transaction  |  A  |  B
-------------+-----+-----
txn1 read    |  10 |  10   ; Transaction 1 sees 10, 10
txn1 write A |   5 |  10   ; A and B now out-of-sync
txn2 read    |   5 |  10   ; Transaction 2 sees 5, 10
txn1 write B |   5 |  15   ; Transaction 1 completes
txn2 write A |  10 |  15   ; Transaction 2 writes based on out-of-sync read
txn2 write B |   5 |  5    ; Should have been 10, 10!
```

This doesn't *solve* the problem, but helps us *explore* the problem in depth. Sometimes this makes the solution obvious–other times, we're just left with a pile of disjoint facts. Even if things *look* jumbled-up and confusing, don't despair! Exploring gives the brain the pieces; it'll link them together over time.

Armed with a detailed *description* of the problem, we're much better equipped to solve it.

Devise a plan

Our brains are excellent pattern-matchers, but not that great at tracking abstract logical operations. Try changing your viewpoint: rotating the problem into a representation that's a little more tractable for your mind. Is there a similar problem you've seen in the past? Is this a well-known problem?

Make sure you know how to *check* the solution. With the problem isolated to a single function, we can write a test case that verifies the account balances are correct. Then we can experiment freely, and have some confidence that we've actually found a solution.

Can you solve a *related* problem? If only concurrent transfers trigger the problem, could we solve the issue by ensuring transactions never take place concurrently–e.g. by wrapping the operation in a lock? Could we solve it by *logging* all transactions, and replaying the log? Is there a simpler variant of the problem that might be tractable–maybe one that always *overcounts*, but never *undercounts*?

Consider your assumptions. We rely on layers of abstraction in writing software–that changing a variable is atomic, that lexical variables don't change, that adding 1 and 1 always gives 2. Sometimes, parts of the computer *fail* to guarantee those abstractions hold. The CPU might–very rarely–fail to divide numbers correctly. A library might, for supposedly valid input, spit out a bad result. A numeric algorithm might fail to converge, and spit out wrong numbers. To avoid questioning *everything*, start in your own code, and work your way down to the assumptions themselves. See if you can devise tests that check the language or library is behaving as you expect.

Can you avoid solving the problem altogether? Is there a library, database, or language feature that does transaction management for us? Is integrating that library worth the reduced complexity in our application?

We're not mathematicians; we're engineers. Part theorist, yes, but also part mechanic. Some problems take a more abstract approach, and others are better approached by tapping it with a wrench and checking the service manual. If other people have solved your problem already, using their solution can be much simpler than devising your own.

Can you think of a way to get more diagnostic information? Perhaps we could log more data from the functions that are misbehaving, or find a way to dump and replay transactions from the live program. Some problems *disappear* when instrumented; these are the hardest to solve, but also the most rewarding.

Combine key phrases in a Google search: the name of the library you're using, the type of exception thrown, any error codes or log messages. Often you'll find a StackOverflow result, a mailing list post, or a Github issue that describes your problem. This works well when you know the technical terms for your problem–in our case, that we're performing a *atomic*, *transactional* transfer between two variables. Sometimes, though, you don't *know* the established names for your problem, and have to resort to blind queries like "variables out of sync" or "overwritten data"–which are much more difficult.

When you get stuck exploring on your own, try asking for help. Collect your description of the problem, the steps you took, and what you expected the program to do. Include any stacktraces or error messages, log files, and the smallest section of source code required to reproduce the

problem. Also include the versions of software used–in Clojure, typically the JVM version (`java -version`), Clojure version (`project.clj`), and any other relevant library versions.

If the project has a Github page or public issue tracker, like Jira, you can try filing an issue there. Here's a particularly well-written issue filed by a user on one of my projects. Note that this user included installation instructions, the command they ran, and the stacktrace it printed. The more specific a description you provide, the easier it is for someone else to understand your problem and help!

Sometimes you need to talk through a problem interactively. For that, I prefer IRC–many projects have a channel on the Freenode IRC network where you can ask basic questions. Remember to be respectful of the channel's time; there may be hundreds of users present, and they have to sort through everything you write. Paste your problem description into a *pastebin* like Gist, then mention the link in IRC with a short–say a few sentences–description of the problem. I try asking in a channel devoted to a specific library or program first, then back off to a more general channel, like #clojure. There's no need to ask "Can I ask a question" first–just jump in.

Since the transactional problem we've been exploring seems like a general issue with atoms, I might ask in #clojure

```
aphyr > Hi! Does anyone know the right way to change multiple atoms at the same time?
aphyr > This function and test case (http://gist.github.com/...) seems to double-
        or under-count when invoked concurrently.
```

Finally, you can join the project's email list, and ask your question there. Turnaround times are longer, but you'll often find a more in-depth response to your question via email. This applies especially if you and the maintainer are in different time zones, or if they're busy with life. You can also ask specific problems on StackOverflow or other message boards; users there can be incredibly helpful.

Remember, other engineers are taking time away from their work, family, friends, and hobbies to help you. It's always polite to give them time to answer first–they may have other priorities. A sincere thank-you is always appreciated–as is paying it forward by answering other users' questions on the list or channel!

Dealing with abuse

Sadly, some women, LGBT people, and so on experience harassment on IRC or in other discussion circles. They may be asked inappropriate personal questions, insulted, threatened, assumed to be straight, to be a man, and so on. Sometimes other users will attack questioners for inexperience. Exclusion can be overt ("Read the fucking docs, faggot!") or more subtle ("Hey dudes, what's up?"). It only takes one hurtful experience this to sour someone on an entire community.

If this happens to you, place your own well-being first. You are *not* obligated to fix anyone else's problems, or to remain in a social context that makes you uncomfortable.

That said, be aware the other people in a channel may not share your culture. English may not be their main language, or they may have said something hurtful without realizing its impact.

Explaining how the comment made you feel can jar a well-meaning but unaware person into reconsidering their actions.

Other times, people are just *mean*–and it only takes one to ruin everybody's day. When this happens, you can appeal to a moderator. On IRC, moderators are sometimes identified by an `@` sign in front of their name; on forums, they may have a special mark on their username or profile. Large projects may have an official policy for reporting abuse on their website or in the channel topic. If there's no policy, try asking whoever seems in charge for help. Most projects have a primary maintainer or community manager with the power to mute or ban malicious users.

Again, these ways of dealing with abuse are *optional*. You have no responsibility to provide others with endless patience, and it is not your responsibility to fix a toxic culture. You can always log off and try something else. There are many communities which will welcome and support you–it may just take a few tries to find the right fit.

If you don't find community, you can *build* it. Starting your own IRC channel, mailing list, or discussion group with a few friends can be a great way to help each other learn in a supportive environment. And if trolls ever come calling, you'll be able to ban them personally.

Now, back to problem-solving.

Execute the plan

Sometimes we can make a quick fix in the codebase, test it by hand, and move on. But for more serious problems, we'll need a more involved process. I always try to get a reproducible test suite–one that runs in a matter of seconds–so that I can continually check my work.

Persist. Many problems require grinding away for some time. Mix blind experimentation with sitting back and planning. Periodically re-evaluate your work–have you made progress? Identified a sub-problem that can be solved independently? Developed a new notation?

If you get stuck, try a new tack. Save your approach as a comment or using `git stash`, and start fresh. Maybe using a different concurrency primitive is in order, or rephrasing the data structure entirely. Take a reading break and review the documentation for the library you're trying to use. Read the *source code* for the functions you're calling–even if you don't understand exactly what it does, it might give you clues to how things work under the hood.

Bounce your problem off a friend. Grab a sheet of paper or whiteboard, describe the problem, and work through your thinking with that person. Their understanding of the problem might be totally off-base, but can still give you valuable insight. Maybe they know exactly what the problem is, and can point you to a solution in thirty seconds!

Finally, take a break. Go home. Go for a walk. Lift heavy, run hard, space out, drink with your friends, practice music, read a book. Just before sleep, go over the problem once more in your head; I often wake up with a new algorithm or new questions burning to get out. Your unconscious mind can come up with unexpected insights if given time *away* from the problem!

Some folks swear by time in the shower, others by hiking, or with pen and paper in a hammock. Find what works for you! The important thing seems to be giving yourself *away* from struggling with the problem.

Look back

Chances are you'll know as soon as your solution works. The program compiles, transactions generate the correct amounts, etc. Now's an important time to *solidify* your work.

Bolster your tests. You may have made the problem *less likely*, but not actually solved it. Try a more aggressive, randomized test; one that runs for longer, that generates a broader class of input. Try it on a copy of the production workload before deploying your change.

Identify *why* the new system works. Pasting something in from StackOverflow may get you through the day, but won't help you solve similar problems in the future. Try to really understand *why* the program went wrong, and how the new pieces work together to prevent the problem. Is there a more general underlying problem? Could you generalize your technique to solve a related problem? If you'll encounter this type of issue frequently, could you build a function or library to help build other solutions?

Document the solution. Write down your description of the problem, and why your changes fix it, as comments in the source code. Use that same description of the solution in your commit message, or attach it as a comment to the resources you used online, so that other people can come to the same understanding.

Debugging Clojure

With these general strategies in mind, I'd like to talk specifically about the debugging *Clojure* code–especially understanding its *stacktraces*. Consider this simple program for baking cakes:

```
(ns scratch.debugging)

(defn bake
  "Bakes a cake for a certain amount of time, returning a cake with a new
  :tastiness level."
  [pie temp time]
  (assoc pie :tastiness
         (condp (* temp time) <
           400 :burned
           350 :perfect
           300 :soggy)))
```

And in the REPL

```
user=> (bake {:flavor :blackberry} 375 10.25)

ClassCastException java.lang.Double cannot be cast to clojure.lang.IFn  scratch.debugging/
bake (debugging.clj:8)
```

This is not particularly helpful. Let's print a full stacktrace using `pst`:

```
user=> (pst)
ClassCastException java.lang.Double cannot be cast to clojure.lang.IFn
    scratch.debugging/bake (debugging.clj:8)
    user/eval1223 (form-init4495957503656407289.clj:1)
    clojure.lang.Compiler.eval (Compiler.java:6619)
    clojure.lang.Compiler.eval (Compiler.java:6582)
    clojure.core/eval (core.clj:2852)
    clojure.main/repl/read-eval-print--6588/fn--6591 (main.clj:259)
    clojure.main/repl/read-eval-print--6588 (main.clj:259)
    clojure.main/repl/fn--6597 (main.clj:277)
    clojure.main/repl (main.clj:277)
```

```
    clojure.tools.nrepl.middleware.interruptible-eval/evaluate/
fn--591 (interruptible_eval.clj:56)
    clojure.core/apply (core.clj:617)
    clojure.core/with-bindings* (core.clj:1788)
```

The first line tells us the *type* of the error: a `ClassCastException`. Then there's some explanatory text: we can't cast a `java.lang.Double` to a `clojure.lang.IFn`. The indented lines show the functions that led to the error. The first line is the deepest function, where the error actually occurred: the `bake` function in the `scratch.debugging` namespace. In parentheses is the file name (`debugging.clj`) and line number (`8`) from the code that caused the error. Each following line shows the function that *called* the previous line. In the REPL, our code is invoked from a special function compiled by the REPL itself–with an automatically generated name like `user/eval1223`, and that function is invoked by the Clojure compiler, and the REPL tooling. Once we see something like `Compiler.eval` at the repl, we can generally skip the rest.

As a general rule, we want to look at the *deepest* (earliest) point in the stacktrace *that we wrote*. Sometimes an error will arise from deep within a library or Clojure itself–but it was probably *invoked* by our code somewhere. We'll skim down the lines until we find our namespace, and start our investigation at that point.

Our case is simple: `bake.clj`, on line 8, seems to be the culprit.

```
        (condp (* temp time) <
```

Now let's consider the error itself: `ClassCastException: java.lang.Double cannot be cast to clojure.lang.IFn`. This implies we had a `Double` and tried to cast it to an `IFn`–but what does "cast" mean? For that matter, what's a `Double`, or an `IFn`?

A quick google search for java.lang.Double reveals that it's a *class* (a Java type) with some basic documentation. "The Double class wraps a value of the primitive type `double` in an object" is not particularly informative–but the "class hierarchy" at the top of the page shows that a `Double` is a kind of `java.lang.Number`. Let's experiment at the REPL:

```
user=> (type 4)
java.lang.Long
user=> (type 4.5)
java.lang.Double
```

Indeed: decimal numbers in Clojure appear to be doubles. One of the expressions in that `condp` call was probably a decimal. At first we might suspect the literal values `300`, `350`, or `400`–but those are `Long`s, not `Doubles`. The only `Double` we passed in was the time duration `10.25`–which appears in `condp` as `(* temp time)`. That first argument was a `Double`, but *should* have been an IFn.

What the heck is an IFn? Its source code has a comment:

> IFn provides complete access to invoking any of Clojure's API's. You can also access any other library written in Clojure, after adding either its source or compiled form to the classpath.

So IFn has to do with *invoking* Clojure's API. Ah– `Fn` probably stands for *function*–and this class is chock full of things like `invoke(Object arg1, Object arg2)`. That suggests that IFn is about *calling functions*. And the `I`? Google suggests it's a Java convention for an *interface*–whatever that is. Remember, we don't have to understand *everything*–just enough to get by. There's plenty to explore later.

Let's check our hypothesis in the repl:

```
user=> (instance? clojure.lang.IFn 2.5)
false
user=> (instance? clojure.lang.IFn conj)
true
user=> (instance? clojure.lang.IFn (fn [x] (inc x)))
true
```

So `Doubles` aren't IFns–but Clojure built-in functions, and anonymous functions, both are. Let's double-check the docs for `condp` again:

```
user=> (doc condp)
-------------------------
clojure.core/condp
([pred expr & clauses])
Macro
  Takes a binary predicate, an expression, and a set of clauses.
  Each clause can take the form of either:

  test-expr result-expr

  test-expr :>> result-fn

  Note :>> is an ordinary keyword.

  For each clause, (pred test-expr expr) is evaluated. If it returns
  logical true, the clause is a match. If a binary clause matches, the
  result-expr is returned, if a ternary clause matches, its result-fn,
  which must be a unary function, is called with the result of the
  predicate as its argument, the result of that call being the return
  value of condp. A single default expression can follow the clauses,
  and its value will be returned if no clause matches. If no default
  expression is provided and no clause matches, an
  IllegalArgumentException is thrown.clj
```

That's a lot to take in! No wonder we got it wrong! We'll take it slow, and look at the arguments.

```
(condp (* temp time) <
```

Our `pred` was `(* temp time)` (a `Double`), and our `expr` was the comparison function `<`. For each clause, `(pred test-expr expr)` is evaluated, so that would expand to something like

```
((* temp time) 400 <)
```

Which evaluates to something like

```
(123.45 400 <)
```

But this isn't a valid Lisp program! It starts with a number, not a function. We should have written `(< 123.45 400)`. Our arguments are backwards!

```
(defn bake
```

```
  "Bakes a cake for a certain amount of time, returning a cake with a new
  :tastiness level."
  [pie temp time]
  (assoc pie :tastiness
         (condp < (* temp time)
           400 :burned
           350 :perfect
           300 :soggy)))
```

```
user=> (use 'scratch.debugging :reload)
nil
user=> (bake {:flavor :chocolate} 375 10.25)
{:tastiness :burned, :flavor :chocolate}
user=> (bake {:flavor :chocolate} 450 0.8)
{:tastiness :perfect, :flavor :chocolate}
```

Mission accomplished! We read the stacktrace as a *path* to a part of the program where things went wrong. We identified the deepest part of that path in *our* code, and looked for a problem there. We discovered that we had reversed the arguments to a function, and after some research and experimentation in the REPL, figured out the right order.

An aside on types: some languages have a *stricter* type system than Clojure's, in which the types of variables are explicitly declared in the program's source code. Those languages can detect type errors–when a variable of one type is used in place of another, incompatible, type–and offer more precise feedback. In Clojure, the compiler does not generally enforce types at compile time, which allows for significant flexibility–but requires more rigorous testing to expose these errors.

Higher order stacktraces

The stacktrace shows us a *path* through the program, moving downwards through functions. However, that path may not be straightforward. When data is handed off from one part of the program to another, the stacktrace may not show the *origin* of an error. When *functions* are handed off from one part of the program to another, the resulting traces can be tricky to interpret indeed.

For instance, say we wanted to make some picture frames out of wood, but didn't know how much wood to buy. We might sketch out a program like this:

```
(defn perimeter
  "Given a rectangle, returns a vector of its edge lengths."
  [rect]
  [(:x rect)
   (:y rect)
   (:z rect)
   (:y rect)])

(defn frame
  "Given a mat width, and a photo rectangle, figure out the size of the frame
  required by adding the mat width around all edges of the photo."
  [mat-width rect]
  (let [margin (* 2 rect)]
    {:x (+ margin (:x rect))
     :y (+ margin (:y rect))}))

(def failure-rate
  "Sometimes the wood is knotty or we screw up a cut. We'll assume we need a
  spare segment once every 8."
  1/8)
```

```
(defn spares
  "Given a list of segments, figure out roughly how many of each distinct size
  will go bad, and emit a sequence of spare segments, assuming we screw up
  `failure-rate` of them."
  [segments]
  (->> segments
       ; Compute a map of each segment length to the number of
       ; segments we'll need of that size.
       frequencies
       ; Make a list of spares for each segment length,
       ; based on how often we think we'll screw up.
       (mapcat (fn [ [segment n]]
                 (repeat (* failure-rate n)
                         segment)))))

(def cut-size
  "How much extra wood do we need for each cut? Let's say a mitred cut for a
  1-inch frame needs a full inch."
  1)

(defn total-wood
  [mat-width photos]
  "Given a mat width and a collection of photos, compute the total linear
  amount of wood we need to buy in order to make frames for each, given a
  2-inch mat."
  (let [segments (->> photos
                      ; Convert photos to frame dimensions
                      (map (partial frame mat-width))
                      ; Convert frames to segments
                      (mapcat perimeter))]

    ; Now, take segments
    (->> segments
         ; Add the spares
         (concat (spares segments))
         ; Include a cut between each segment
         (interpose cut-size)
         ; And sum the whole shebang.
         (reduce +))))

(->> [{:x 8
       :y 10}
      {:x 10
       :y 8}
      {:x 20
       :y 30}]
     (total-wood 2)
     (println "total inches:"))
```

Running this program yields a curious stacktrace. We'll print the *full* trace (not the shortened one that comes with `pst`) for the last exception `*e` with the `.printStackTrace` function.

```
user=> (.printStackTrace *e)
java.lang.ClassCastException: clojure.lang.PersistentArrayMap cannot be cast to
 java.lang.Number, compiling:(scratch/debugging.clj:73:23)
    at clojure.lang.Compiler.load(Compiler.java:7142)
    at clojure.lang.RT.loadResourceScript(RT.java:370)
    at clojure.lang.RT.loadResourceScript(RT.java:361)
    at clojure.lang.RT.load(RT.java:440)
    at clojure.lang.RT.load(RT.java:411)
        ...
    at java.lang.Thread.run(Thread.java:745)
Caused by: java.lang.ClassCastException: clojure.lang.PersistentArrayMap cannot be cast to
 java.lang.Number
    at clojure.lang.Numbers.multiply(Numbers.java:146)
    at clojure.lang.Numbers.multiply(Numbers.java:3659)
    at scratch.debugging$frame.invoke(debugging.clj:26)
```

```
    at clojure.lang.AFn.applyToHelper(AFn.java:156)
    at clojure.lang.AFn.applyTo(AFn.java:144)
    at clojure.core$apply.invoke(core.clj:626)
    at clojure.core$partial$fn__4228.doInvoke(core.clj:2468)
    at clojure.lang.RestFn.invoke(RestFn.java:408)
    at clojure.core$map$fn__4245.invoke(core.clj:2557)
    at clojure.lang.LazySeq.sval(LazySeq.java:40)
    at clojure.lang.LazySeq.seq(LazySeq.java:49)
    at clojure.lang.RT.seq(RT.java:484)
    at clojure.core$seq.invoke(core.clj:133)
    at clojure.core$map$fn__4245.invoke(core.clj:2551)
    at clojure.lang.LazySeq.sval(LazySeq.java:40)
    at clojure.lang.LazySeq.seq(LazySeq.java:49)
    at clojure.lang.RT.seq(RT.java:484)
    at clojure.core$seq.invoke(core.clj:133)
    at clojure.core$apply.invoke(core.clj:624)
    at clojure.core$mapcat.doInvoke(core.clj:2586)
    at clojure.lang.RestFn.invoke(RestFn.java:423)
    at scratch.debugging$total_wood.invoke(debugging.clj:62)
        ...
```

First: this trace has *two parts*. The top-level error (a `CompilerException`) appears first, and is followed by the exception that *caused* the `CompilerException`: a `ClassCastException`. This makes the stacktrace read somewhat out of order, since the deepest part of the trace occurs in the *first* line of the *last* exception. We read `C B A` then `F E D`. This is an old convention in the Java language, and the cause of no end of frustration.

Notice that this representation of the stacktrace is less friendly than `(pst)`. We're seeing the Java Virtual Machine (JVM)'s internal representation of Clojure functions, which look like `clojure.core$partial$fn__4228.doInvoke`. This corresponds to the namespace `clojure.core`, in which there is a function called `partial`, inside of which is an *anonymous* function, here named `fn__4228`. Calling a Clojure function is written, in the JVM, as `.invoke` or `.doInvoke`.

So: the root cause was a `ClassCastException`, and it tells us that Clojure expected a `java.lang.Number`, but found a `PersistentArrayMap`. We might guess that `PersistentArrayMap` is something to do with the map data structure, which we used in this program:

```
user=> (type {:x 1})
clojure.lang.PersistentArrayMap
```

And we'd be right. We can also tell, by reading down the stacktrace looking for our `scratch.debugging` namespace, where the error took place: `scratch.debugging$frame`, on line `26`.

```
  (let [margin (* 2 rect)]
```

There's our multiplication operation `*`, which we might assume expands to `clojure.lang.Numbers.multiply`. But the *path* to the error is odd.

```
                (->> photos
                     ; Convert photos to frame dimensions
                     (map (partial frame mat-width))
```

In `total-wood`, we call `(map (partial frame mat-width) photos)` right away, so we'd expect the stacktrace to go from `total-wood` to `map` to `frame`. But this is *not*

what happens. Instead, `total-wood` invokes something called `RestFn`–a piece of Clojure plumbing–which in turn calls `mapcat`.

```
at clojure.core$mapcat.doInvoke(core.clj:2586)
    at clojure.lang.RestFn.invoke(RestFn.java:423)
    at scratch.debugging$total_wood.invoke(debugging.clj:62)
```

Why doesn't `total-wood` call `map` first? Well it *did*–but `map` doesn't actually apply its function to anything in the `photos` vector when invoked. Instead, it returns a *lazy* sequence–one which applies `frame` only when elements are asked for.

```
user=> (type (map inc (range 10)))
clojure.lang.LazySeq
```

Inside each `LazySeq` is a box containing a function. When you ask a `LazySeq` for its first value, it calls that function to return a new sequence–and *that's* when `frame` gets invoked. What we're seeing in this stacktrace is the `LazySeq` internal machinery at work–`mapcat` asks it for a value, and the LazySeq asks `map` to generate that value.

```
at clojure.core$partial$fn__4228.doInvoke(core.clj:2468)
    at clojure.lang.RestFn.invoke(RestFn.java:408)
    at clojure.core$map$fn__4245.invoke(core.clj:2557)
    at clojure.lang.LazySeq.sval(LazySeq.java:40)
    at clojure.lang.LazySeq.seq(LazySeq.java:49)
    at clojure.lang.RT.seq(RT.java:484)
    at clojure.core$seq.invoke(core.clj:133)
    at clojure.core$map$fn__4245.invoke(core.clj:2551)
    at clojure.lang.LazySeq.sval(LazySeq.java:40)
    at clojure.lang.LazySeq.seq(LazySeq.java:49)
    at clojure.lang.RT.seq(RT.java:484)
    at clojure.core$seq.invoke(core.clj:133)
    at clojure.core$apply.invoke(core.clj:624)
    at clojure.core$mapcat.doInvoke(core.clj:2586)
    at clojure.lang.RestFn.invoke(RestFn.java:423)
    at scratch.debugging$total_wood.invoke(debugging.clj:62)
```

In fact we pass through `map`'s laziness *twice* here: a quick peek at `(source mapcat)` shows that it expands into a `map` call itself, and then there's a *second* map: the one we created in in `total-wood`. Then an odd thing happens–we hit something called `clojure.core$partial$fn__4228`.

```
  (map (partial frame mat-width) photos)
```

The `frame` function takes two arguments: a mat width and a photo. We wanted a function that takes just *one* argument: a photo. `(partial frame mat-width)` took `mat-width` and generated a *new function* which takes one arg–call it `photo`–and calls `(frame mad-width photo)`. That automatically generated function, returned by `partial`, is what `map` uses to generate new elements of its sequence on demand.

```
user=> (partial + 1)
#<core$partial$fn__4228 clojure.core$partial$fn__4228@243634f2>
user=> ((partial + 1) 4)
5
```

That's why we see control flow through `clojure.core$partial$fn__4228` (an anonymous function defined inside `clojure.core/partial`) on the way to `frame`.

```
Caused by: java.lang.ClassCastException: clojure.lang.PersistentArrayMap cannot be cast to
 java.lang.Number
    at clojure.lang.Numbers.multiply(Numbers.java:146)
    at clojure.lang.Numbers.multiply(Numbers.java:3659)
    at scratch.debugging$frame.invoke(debugging.clj:26)
    at clojure.lang.AFn.applyToHelper(AFn.java:156)
    at clojure.lang.AFn.applyTo(AFn.java:144)
    at clojure.core$apply.invoke(core.clj:626)
    at clojure.core$partial$fn__4228.doInvoke(core.clj:2468)
```

And there's our suspect! `scratch.debugging/frame`, at line `26`. To return to that line again:

```
  (let [margin (* 2 rect)]
```

`*` is a multiplication, and `2` is obviously a number, but `rect`… `rect` is a map here. Aha! We meant to multiply the `mat-width` by two, not the rectangle.

```
(defn frame
  "Given a mat width, and a photo rectangle, figure out the size of the frame
  required by adding the mat width around all edges of the photo."
  [mat-width rect]
  (let [margin (* 2 mat-width)]
    {:x (+ margin (:x rect))
     :y (+ margin (:y rect))}))
```

I believe we've fixed the bug, then. Let's give it a shot!

The unbearable lightness of nil

There's one more bug lurking in this program. This one's stacktrace is short.

```
user=> (use 'scratch.debugging :reload)

CompilerException java.lang.NullPointerException, compiling:(scratch/debugging.clj:73:23)
user=> (pst)
CompilerException java.lang.NullPointerException, compiling:(scratch/debugging.clj:73:23)
    clojure.lang.Compiler.load (Compiler.java:7142)
    clojure.lang.RT.loadResourceScript (RT.java:370)
    clojure.lang.RT.loadResourceScript (RT.java:361)
    clojure.lang.RT.load (RT.java:440)
    clojure.lang.RT.load (RT.java:411)
    clojure.core/load/fn--5066 (core.clj:5641)
    clojure.core/load (core.clj:5640)
    clojure.core/load-one (core.clj:5446)
    clojure.core/load-lib/fn--5015 (core.clj:5486)
    clojure.core/load-lib (core.clj:5485)
    clojure.core/apply (core.clj:626)
    clojure.core/load-libs (core.clj:5524)
Caused by:
NullPointerException
    clojure.lang.Numbers.ops (Numbers.java:961)
    clojure.lang.Numbers.add (Numbers.java:126)
    clojure.core/+ (core.clj:951)
    clojure.core.protocols/fn--6086 (protocols.clj:143)
    clojure.core.protocols/fn--6057/G--6052--6066 (protocols.clj:19)
    clojure.core.protocols/seq-reduce (protocols.clj:27)
    clojure.core.protocols/fn--6078 (protocols.clj:53)
    clojure.core.protocols/fn--6031/G--6026--6044 (protocols.clj:13)
    clojure.core/reduce (core.clj:6287)
    scratch.debugging/total-wood (debugging.clj:69)
    scratch.debugging/eval1560 (debugging.clj:81)
    clojure.lang.Compiler.eval (Compiler.java:6703)
```

On line 69, `total-wood` calls `reduce`, which dives through a series of functions from `clojure.core.protocols` before emerging in `+`: the function we passed to `reduce`. Reduce is trying to combine two elements from its collection of wood segments using `+`, but one of them was `nil`. Clojure calls this a `NullPointerException`. In `total-wood`, we constructed the sequence of segments this way:

```
  (let [segments (->> photos
                      ; Convert photos to frame dimensions
                      (map (partial frame mat-width))
                      ; Convert frames to segments
                      (mapcat perimeter))]

    ; Now, take segments
    (->> segments
         ; Add the spares
         (concat (spares segments))
         ; Include a cut between each segment
         (interpose cut-size)
         ; And sum the whole shebang.
         (reduce +))))
```

Where did the `nil` value come from? The stacktrace *doesn't say*, because the sequence `reduce` is traversing didn't have any problem *producing* the `nil`. `reduce` asked for a value and the sequence happily produced a `nil`. We only had a problem when it came time to *combine* the `nil` with the next value, using `+`.

A stacktrace like this is something like a murder mystery: we know the program died in the reducer, that it was shot with a `+`, and the bullet was a `nil`–but we don't know where the bullet came from. The trail runs cold. We need *more forensic information*–more hints about the `nil`'s origin–to find the culprit.

Again, this is a class of error largely preventable with static type systems. If you have worked with a statically typed language in the past, it may be interesting to consider that almost every Clojure function takes `Option[A]` and does something more-or-less sensible, returning `Option[B]`. Whether the error propagates as a `nil` or an `Option`, there can be similar difficulties in localizing the cause of the problem.

Let's try printing out the state as `reduce` goes along:

```
    (->> segments
         ; Add the spares
         (concat (spares segments))
         ; Include a cut between each segment
         (interpose cut-size)
         ; And sum the whole shebang.
         (reduce (fn [acc x] (prn acc x) (+ acc x))))))
```

```
user=> (use 'scratch.debugging :reload)
12 1
13 14
27 1
28 nil

CompilerException java.lang.NullPointerException, compiling:(scratch/debugging.clj:73:56)
```

Not every value is nil! There's a `14` there which looks like a plausible segment for a frame, and two one-inch buffers from `cut-size`. We can rule out `interpose` because it inserts

a `1` every time, and that `1` reduces correctly. But where's that `nil` coming from? Is from `segments` or `(spares segments)`?

```
  (let [segments (->> photos
                      ; Convert photos to frame dimensions
                      (map (partial frame mat-width))
                      ; Convert frames to segments
                      (mapcat perimeter))]

    (prn :segments segments)
```

```
user=> (use 'scratch.debugging :reload)
:segments (12 14 nil 14 14 12 nil 12 24 34 nil 34)
```

It is present in `segments`. Let's trace it backwards through the sequence's creation. It'd be handy to have a function like `prn` that *returned* its input, so we could spy on values as they flowed through the `->>` macro.

```
(defn spy
  [& args]
  (apply prn args)
  (last args))
```

```
  (let [segments (->> photos
                      ; Convert photos to frame dimensions
                      (map (partial frame mat-width))
                      (spy :frames)
                      ; Convert frames to segments
                      (mapcat perimeter))]
```

```
user=> (use 'scratch.debugging :reload)
:frames ({:x 12, :y 14} {:x 14, :y 12} {:x 24, :y 34})
:segments (12 14 nil 14 14 12 nil 12 24 34 nil 34)
```

Ah! So the frames are intact, but the *perimeters* are bad. Let's check the `perimeter` function:

```
(defn perimeter
  "Given a rectangle, returns a vector of its edge lengths."
  [rect]
  [(:x rect)
   (:y rect)
   (:z rect)
   (:y rect)])
```

Spot the typo? We wrote `:z` instead of `:x`. Since the frame didn't have a `:z` field, it returned `nil`! That's the origin of our `NullPointerException`. With the bug fixed, we can re-run and find:

```
user=> (use 'scratch.debugging :reload)
total inches: 319
```

Whallah!

Recap

As we solve more and more problems, we get faster at debugging–at skipping over irrelevant log data, figuring out exactly what input was at fault, knowing what terms to search for, and

developing a network of peers and mentors to ask for help. But when we encounter unexpected bugs, it can help to fall back on a family of problem-solving tactics.

We explore the problem thoroughly, localizing it to a particular function, variable, or set of inputs. We identify the boundaries of the problem, carving away parts of the system that work as expected. We develop new notation, maps, and diagrams of the problem space, precisely characterizing it in a variety of modes.

With the problem identified, we search for extant solutions–or related problems others have solved in the past. We trawl through issue trackers, mailing list posts, blogs, and forums like Stackoverflow, or, for more theoretical problems, academic papers, Mathworld, and Wikipedia, etc. If searching reveals nothing, we try rephrasing the problem, relaxing the constraints, adding debugging statements, and solving smaller subproblems. When all else fails, we ask for help from our peers, or from the community in IRC, mailing lists, and so on, or just take a break.

We learned to explore Clojure stacktraces as a trail into our programs, leading to the place where an error occurred. But not all paths are linear, and we saw how lazy operations and higher-order functions create inversions and intermediate layers in the stacktrace. Then we learned how to debug values that were *distant* from the trace, by adding logging statements and working our way closer to the origin.

Programming languages and us, their users, are engaged in a continual dialogue. We may speak more formally, verbosely, with many types and defensive assertions–or we may speak quickly, generally, in fuzzy terms. The more precise we are with the specifications of our program's types, the more the program can assist us when things go wrong. Conversely, those specifications *harden* our programs into strong but *rigid* forms, and rigid structures are harder to bend into new shapes.

In Clojure we strike a more dynamic balance: we speak in generalities, but we pay for that flexibility. Our errors are harder to trace to their origins. While the Clojure compiler can warn us of some errors, like mis-spelled variable names, it cannot (without a library like core.typed) tell us when we have incorrectly assumed an object will be of a certain type. Even very rigid languages, like Haskell, cannot identify some errors, like reversing the arguments to a subtraction function. *Some* tests are always necessary, though types are a huge boon.

No matter what language we write in, we use a balance of types and tests to *validate* our assumptions, both when the program is compiled and when it is run.

The errors that arise in compilation or runtime aren't *rebukes* so much as *hints*. Don't despair! They point the way towards understanding one's program in more detail–though the errors may be cryptic. Over time we get better at reading our language's errors and making our programs more robust.