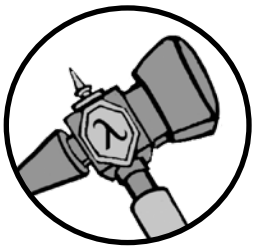
3

DO THINGS: A CLOJURE CRASH COURSE



It's time to learn how to actually *do things* with Clojure! Hot damn! Although you've undoubtedly heard of Clojure's awesome concurrency support and other stupendous features, Clojure's most salient characteristic is that it is a Lisp. In this chapter, you'll explore the elements that compose this Lisp core: syntax, functions, and data. Together they will give you a solid foundation for representing and solving problems in Clojure.

After laying this groundwork, you will be able to write some super important code. In the last section, you'll tie everything together by creating a model of a hobbit and writing a function to hit it in a random spot. Super! Important!

As you move through the chapter, I recommend that you type the examples in a REPL and run them. Programming in a new language is a skill, and just like yodeling or synchronized swimming, you have to practice to learn it. By the way, *Synchronized Swimming for Yodelers for the Brave and True* will be published in August of 20never. Keep an eye out for it!

Syntax

Clojure's syntax is simple. Like all Lisps, it employs a uniform structure, a handful of special operators, and a constant supply of parentheses delivered from the parenthesis mines hidden beneath the Massachusetts Institute of Technology, where Lisp was born.

Forms

All Clojure code is written in a uniform structure. Clojure recognizes two kinds of structures:

- Literal representations of data structures (like numbers, strings, maps, and vectors)
- Operations

We use the term *form* to refer to valid code. I'll also sometimes use *expression* to refer to Clojure forms. But don't get too hung up on the terminology. Clojure *evaluates* every form to produce a value. These literal representations are all valid forms:

```
1
"a string"
["a" "vector" "of" "strings"]
```

Your code will rarely contain free-floating literals, of course, because they don't actually do anything on their own. Instead, you'll use literals in operations. Operations are how you *do things*. All operations take the form *opening parenthesis, operator, operands, closing parenthesis*:

```
(operator operand1 operand2 ... operandn)
```

Notice that there are no commas. Clojure uses whitespace to separate operands, and it treats commas as whitespace. Here are some example operations:

```
(+ 1 2 3)
; => 6

(str "It was the panda " "in the library " "with a dust buster")
; => "It was the panda in the library with a dust buster"
```

In the first operation, the operator `+` adds the operands `1`, `2`, and `3`. In the second operation, the operator `str` concatenates three strings to form a new string. Both are valid forms. Here's something that is not a form because it doesn't have a closing parenthesis:

```
(+
```

Clojure's structural uniformity is probably different from what you're used to. In other languages, different operations might have different structures depending on the operator and the operands. For example, JavaScript employs a smorgasbord of infix notation, dot operators, and parentheses:

```
1 + 2 + 3
"It was the panda ".concat("in the library ", "with a dust buster")
```

Clojure's structure is very simple and consistent by comparison. No matter which operator you're using or what kind of data you're operating on, the structure is the same.

Control Flow

Let's look at three basic control flow operators: `if`, `do`, and `when`. Throughout the book you'll encounter more, but these will get you started.

if

This is the general structure for an `if` expression:

```
(if boolean-form
  then-form
  optional-else-form)
```

A Boolean form is just a form that evaluates to a truthy or falsey value. You'll learn about truthiness and falsiness in the next section. Here are a couple of `if` examples:

```
(if true
  "By Zeus's hammer!"
  "By Aquaman's trident!")
; => "By Zeus's hammer!"

(if false
  "By Zeus's hammer!"
  "By Aquaman's trident!")
; => "By Aquaman's trident!"
```

The first example returns `"By Zeus's hammer!"` because the Boolean form evaluates to `true`, a truthy value, and the second example returns `"By Aquaman's trident!"` because its Boolean form, `false`, evaluates to a falsey value.

You can also omit the `else` branch. If you do that and the Boolean expression is false, Clojure returns `nil`, like this:

```
(if false
  "By Odin's Elbow!")
; => nil
```

Notice that `if` uses operand position to associate operands with the `then` and `else` branches: the first operand is the `then` branch, and the second operand is the (optional) `else` branch. As a result, each branch can have only one form. This is different from most languages. For example, you can write this in Ruby:

```
if true
  doer.do_thing(1)
  doer.do_thing(2)
else
  other_doer.do_thing(1)
  other_doer.do_thing(2)
end
```

To get around this apparent limitation, you can use the `do` operator.

do

The `do` operator lets you *wrap up* multiple forms in parentheses and run each of them. Try the following in your REPL:

```
(if true
  (do (println "Success!")
      "By Zeus's hammer!")
  (do (println "Failure!")
      "By Aquaman's trident!"))
; => Success!
; => "By Zeus's hammer!"
```

This operator lets you do multiple things in each of the `if` expression's branches. In this case, two things happen: `Success!` is printed in the REPL, and `"By Zeus's hammer!"` is returned as the value of the entire `if` expression.

when

The `when` operator is like a combination of `if` and `do`, but with no `else` branch. Here's an example:

```
(when true
  (println "Success!")
  "abra cadabra")
```

```
; => Success!
; => "abra cadabra"
```

Use `when` if you want to do multiple things when some condition is true, and you always want to return `nil` when the condition is false.

nil, true, false, Truthiness, Equality, and Boolean Expressions

Clojure has `true` and `false` values. `nil` is used to indicate *no value* in Clojure. You can check if a value is `nil` with the appropriately named `nil?` function:

```
(nil? 1)
; => false

(nil? nil)
; => true
```

Both `nil` and `false` are used to represent logical falsiness, whereas all other values are logically truthy. *Truthy* and *falsey* refer to how a value is treated in a Boolean expression, like the first expression passed to `if`:

```
(if "bears eat beets"
  "bears beets Battlestar Galactica")
; => "bears beets Battlestar Galactica"

(if nil
  "This won't be the result because nil is falsey"
  "nil is falsey")
; => "nil is falsey"
```

In the first example, the string `"bears eat beets"` is considered truthy, so the `if` expression evaluates to `"bears beets Battlestar Galactica"`. The second example shows a falsey value as falsey.

Clojure's equality operator is `=`:

```
(= 1 1)
; => true

(= nil nil)
; => true

(= 1 2)
; => false
```

Some other languages require you to use different operators when comparing values of different types. For example, you might have to use some kind of special string equality operator made just for strings. But you don't need anything weird or tedious like that to test for equality when using Clojure's built-in data structures.

Clojure uses the Boolean operators `or` and `and`. `or` returns either the first truthy value or the last value. `and` returns the first falsey value or, if no values are falsey, the last truthy value. Let's look at `or` first:

```
(or false nil :large_I_mean_venti :why_cant_I_just_say_large)
; => :large_I_mean_venti

(or (= 0 1) (= "yes" "no"))
; => false

(or nil)
; => nil
```

In the first example, the return value is `:large_I_mean_venti` because it's the first truthy value. The second example has no truthy values, so `or` returns the last value, which is `false`. In the last example, once again no truthy values exist, and `or` returns the last value, which is `nil`. Now let's look at `and`:

```
(and :free_wifi :hot_coffee)
; => :hot_coffee

(and :feelin_super_cool nil false)
; => nil
```

In the first example, `and` returns the last truthy value, `:hot_coffee`. In the second example, `and` returns `nil`, which is the first falsey value.

Naming Values with def

You use `def` to *bind* a name to a value in Clojure:

```
(def failed-protagonist-names
  ["Larry Potter" "Doreen the Explorer" "The Incredible Bulk"])

failed-protagonist-names
; => ["Larry Potter" "Doreen the Explorer" "The Incredible Bulk"]
```

In this case, you're binding the name `failed-protagonist-names` to a vector containing three strings (you'll learn about vectors in "Vectors" on page 45).

Notice that I'm using the term *bind*, whereas in other languages you'd say you're *assigning* a value to a *variable*. Those other languages typically encourage you to perform multiple assignments to the same variable.

For example, in Ruby you might perform multiple assignments to a variable to build up its value:

```
severity = :mild
error_message = "OH GOD! IT'S A DISASTER! WE'RE "
if severity == :mild
  error_message = error_message + "MILDLY INCONVENIENCED!"
else
  error_message = error_message + "DOOOOOOOMED!"
end
```

You might be tempted to do something similar in Clojure:

```
(def severity :mild)
(def error-message "OH GOD! IT'S A DISASTER! WE'RE ")
(if (= severity :mild)
  (def error-message (str error-message "MILDLY INCONVENIENCED!"))
  (def error-message (str error-message "DOOOOOOOMED!")))
```

However, changing the value associated with a name like this can make it harder to understand your program's behavior because it's more difficult to know which value is associated with a name or why that value might have changed. Clojure has a set of tools for dealing with change, which you'll learn about in Chapter 10. As you learn Clojure, you'll find that you'll rarely need to alter a name/value association. Here's one way you could write the preceding code:

```
(defn error-message
  [severity]
  (str "OH GOD! IT'S A DISASTER! WE'RE "
       (if (= severity :mild)
         "MILDLY INCONVENIENCED!"
         "DOOOOOOOMED!")))

(error-message :mild)
; => "OH GOD! IT'S A DISASTER! WE'RE MILDLY INCONVENIENCED!"
```

Here, you create a function, `error-message`, which accepts a single argument, `severity`, and uses that to determine which string to return. You then call the function with `:mild` for the severity. You'll learn all about creating functions in "Functions" on page 48; in the meantime, you should treat `def` as if it's defining constants. In the next few chapters, you'll learn how to work with this apparent limitation by embracing the functional programming paradigm.

Data Structures

Clojure comes with a handful of data structures that you'll use the majority of the time. If you're coming from an object-oriented background, you'll be surprised at how much you can do with the seemingly basic types presented here.

All of Clojure's data structures are immutable, meaning you can't change them in place. For example, in Ruby you could do the following to reassign the failed protagonist name at index 0:

```
failed_protagonist_names = [
  "Larry Potter",
  "Doreen the Explorer",
  "The Incredible Bulk"
]
failed_protagonist_names[0] = "Gary Potter"

failed_protagonist_names
# => [
#   "Gary Potter",
#   "Doreen the Explorer",
#   "The Incredible Bulk"
# ]
```

Clojure has no equivalent for this. You'll learn more about why Clojure was implemented this way in Chapter 10, but for now it's fun to learn just how to do things without all that philosophizing. Without further ado, let's look at numbers in Clojure.

Numbers

Clojure has pretty sophisticated numerical support. I won't spend much time dwelling on the boring technical details (like coercion and contagion), because that will get in the way of *doing things*. If you're interested in said boring details, check out the documentation at *http://clojure.org/data_structures#Data%20Structures-Numbers*. Suffice it to say, Clojure will merrily handle pretty much anything you throw at it.

In the meantime, we'll work with integers and floats. We'll also work with ratios, which Clojure can represent directly. Here's an integer, a float, and a ratio, respectively:

```
93
1.2
1/5
```

Strings

Strings represent text. The name comes from the ancient Phoenicians, who one day invented the alphabet after an accident involving yarn. Here are some examples of string literals:

```
"Lord Voldemort"
"\"He who must not be named\""
"\"Great cow of Moscow!\" - Hermes Conrad"
```

Notice that Clojure only allows double quotes to delineate strings. `'Lord Voldemort'`, for example, is not a valid string. Also notice that Clojure doesn't have string interpolation. It only allows concatenation via the `str` function:


```
(def name "Chewbacca")
(str "\"Uggllglglglglglglglll\" - " name)
; => "Uggllglglglglglglglll" - Chewbacca
```

Maps

Maps are similar to dictionaries or hashes in other languages. They're a way of associating some value with some other value. The two kinds of maps in Clojure are hash maps and sorted maps. I'll only cover the more basic hash maps. Let's look at some examples of map literals. Here's an empty map:

```
{}
```

In this example, `:first-name` and `:last-name` are keywords (I'll cover those in the next section):

```
{:first-name "Charlie"
 :last-name "McFishwich"}
```

Here we associate `"string-key"` with the `+` function:

```
{"string-key" +}
```

Maps can be nested:

```
{:name {:first "John" :middle "Jacob" :last "Jingleheimerschmidt"}}
```

Notice that map values can be of any type—strings, numbers, maps, vectors, even functions. Clojure don't care!

Besides using map literals, you can use the `hash-map` function to create a map:

```
(hash-map :a 1 :b 2)
; => {:a 1 :b 2}
```

You can look up values in maps with the `get` function:

```
(get {:a 0 :b 1} :b)
; => 1

(get {:a 0 :b {:c "ho hum"}} :b)
; => {:c "ho hum"}
```

In both of these examples, we asked `get` for the value of the `:b` key in the given map—in the first case it returns `1`, and in the second case it returns the nested map `{:c "ho hum"}`.

`get` will return `nil` if it doesn't find your key, or you can give it a default value to return, such as `"unicorns?"`:

```
(get {:a 0 :b 1} :c)
; => nil

(get {:a 0 :b 1} :c "unicorns?")
; => "unicorns?"
```

The `get-in` function lets you look up values in nested maps:

```
(get-in {:a 0 :b {:c "ho hum"}} [:b :c])
; => "ho hum"
```

Another way to look up a value in a map is to treat the map like a function with the key as its argument:

```
({:name "The Human Coffeepot"} :name)
; => "The Human Coffeepot"
```

Another cool thing you can do with maps is use keywords as functions to look up their values, which leads to the next subject, keywords.

Keywords

Clojure keywords are best understood by seeing how they're used. They're primarily used as keys in maps, as you saw in the preceding section. Here are some more examples of keywords:

```
:a
:rumplestiltsken
:34
:_?
```

Keywords can be used as functions that look up the corresponding value in a data structure. For example, you can look up `:a` in a map:

```
(:a {:a 1 :b 2 :c 3})
; => 1
```

This is equivalent to:

```
(get {:a 1 :b 2 :c 3} :a)
; => 1
```

You can provide a default value, as with `get`:

```
(:d {:a 1 :b 2 :c 3} "No gnome knows homes like Noah knows")
; => "No gnome knows homes like Noah knows"
```

Using a keyword as a function is pleasantly succinct, and Real Clojurists do it all the time. You should do it too!

Vectors

A vector is similar to an array, in that it's a 0-indexed collection. For example, here's a vector literal:

```
[3 2 1]
```

Here we're returning the 0th element of a vector:

```
(get [3 2 1] 0)
; => 3
```

Here's another example of getting by index:

```
(get ["a" {:name "Pugsley Winterbottom"} "c"] 1)
; => {:name "Pugsley Winterbottom"}
```

You can see that vector elements can be of any type, and you can mix types. Also notice that we're using the same `get` function as we use when looking up values in maps.

You can create vectors with the `vector` function:

```
(vector "creepy" "full" "moon")
; => ["creepy" "full" "moon"]
```

You can use the `conj` function to add additional elements to the vector. Elements are added to the *end* of a vector:

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

Vectors aren't the only way to store sequences; Clojure also has *lists*.

Lists

Lists are similar to vectors in that they're linear collections of values. But there are some differences. For example, you can't retrieve list elements with `get`. To write a list literal, just insert the elements into parentheses and use a single quote at the beginning:

```
'(1 2 3 4)
; => (1 2 3 4)
```

Notice that when the REPL prints out the list, it doesn't include the single quote. We'll come back to why that happens later, in Chapter 7. If you want to retrieve an element from a list, you can use the `nth` function:

```
(nth '(:a :b :c) 0)
; => :a

(nth '(:a :b :c) 2)
; => :c
```

I don't cover performance in detail in this book because I don't think it's useful to focus on it until after you've become familiar with a language. However, it's good to know that using `nth` to retrieve an element from a list is slower than using `get` to retrieve an element from a vector. This is because Clojure has to traverse all *n* elements of a list to get to the *n*th, whereas it only takes a few hops at most to access a vector element by its index.

List values can have any type, and you can create lists with the `list` function:

```
(list 1 "two" {3 4})
; => (1 "two" {3 4})
```

Elements are added to the *beginning* of a list:

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

When should you use a list and when should you use a vector? A good rule of thumb is that if you need to easily add items to the beginning of a sequence or if you're writing a macro, you should use a list. Otherwise, you should use a vector. As you learn more, you'll get a good feel for when to use which.

Sets

Sets are collections of unique values. Clojure has two kinds of sets: hash sets and sorted sets. I'll focus on hash sets because they're used more often. Here's the literal notation for a hash set:

```
#{"kurt vonnegut" 20 :icicle}
```

You can also use `hash-set` to create a set:

```
(hash-set 1 1 2 2)
; => #{1 2}
```

Note that multiple instances of a value become one unique value in the set, so we're left with a single `1` and a single `2`. If you try to add a value

to a set that already contains that value (such as `:b` in the following code), it will still have only one of that value:

```
(conj #{:a :b} :b)
; => #{:a :b}
```

You can also create sets from existing vectors and lists by using the `set` function:

```
(set [3 3 3 4 4])
; => #{3 4}
```

You can check for set membership using the `contains?` function, by using `get`, or by using a keyword as a function with the set as its argument. `contains?` returns `true` or `false`, whereas `get` and keyword lookup will return the value if it exists, or `nil` if it doesn't.

Here's how you'd use `contains?`:

```
(contains? #{:a :b} :a)
; => true

(contains? #{:a :b} 3)
; => false

(contains? #{nil} nil)
; => true
```

Here's how you'd use a keyword:

```
(:a #{:a :b})
; => :a
```

And here's how you'd use `get`:

```
(get #{:a :b} :a)
; => :a

(get #{:a nil} nil)
; => nil

(get #{:a :b} "kurt vonnegut")
; => nil
```

Notice that using `get` to test whether a set contains `nil` will always return `nil`, which is confusing. `contains?` may be the better option when you're testing specifically for set membership.

Simplicity

You may have noticed that the treatment of data structures so far doesn't include a description of how to create new types or classes. The reason is that Clojure's emphasis on simplicity encourages you to reach for the built-in data structures first.

If you come from an object-oriented background, you might think that this approach is weird and backward. However, what you'll find is that your data does not have to be tightly bundled with a class for it to be useful and intelligible. Here's an epigram loved by Clojurists that hints at the Clojure philosophy:

> It is better to have 100 functions operate on one data structure
> than 10 functions on 10 data structures.
> —Alan Perlis

You'll learn more about this aspect of Clojure's philosophy in the coming chapters. For now, keep an eye out for the ways that you gain code reusability by sticking to basic data structures.

This concludes our Clojure data structures primer. Now it's time to dig in to functions and learn how to use these data structures!

Functions

One of the reasons people go nuts over Lisps is that these languages let you build programs that behave in complex ways, yet the primary building block—the function—is so simple. This section initiates you into the beauty and elegance of Lisp functions by explaining the following:

- Calling functions
- How functions differ from macros and special forms
- Defining functions
- Anonymous functions
- Returning functions

Calling Functions

By now you've seen many examples of function calls:

```
(+ 1 2 3 4)
(* 1 2 3 4)
(first [1 2 3 4])
```

Remember that all Clojure operations have the same syntax: opening parenthesis, operator, operands, closing parenthesis. *Function call* is just another term for an operation where the operator is a function or a *function expression* (an expression that returns a function).

This lets you write some pretty interesting code. Here's a function expression that returns the + (addition) function:

```
(or + -)
; => #<core$_PLUS_ clojure.core$_PLUS_@76dace31>
```

That return value is the string representation of the addition function. Because the return value of or is the first truthy value, and here the addition function is truthy, the addition function is returned. You can also use this expression as the operator in another expression:

```
((or + -) 1 2 3)
; => 6
```

Because (or + -) returns +, this expression evaluates to the sum of 1, 2, and 3, returning 6.

Here are a couple more valid function calls that each return 6:

```
((and (= 1 1) +) 1 2 3)
; => 6

((first [+ 0]) 1 2 3)
; => 6
```

In the first example, the return value of and is the first falsey value or the last truthy value. In this case, + is returned because it's the last truthy value, and is then applied to the arguments 1 2 3, returning 6. In the second example, the return value of first is the first element in a sequence, which is + in this case.

However, these aren't valid function calls, because numbers and strings aren't functions:

```
(1 2 3 4)
("test" 1 2 3)
```

If you run these in your REPL, you'll get something like this:

```
ClassCastException java.lang.String cannot be cast to clojure.lang.IFn
user/eval728 (NO_SOURCE_FILE:1)
```

You're likely to see this error many times as you continue with Clojure: *<x> cannot be cast to clojure.lang.IFn* just means that you're trying to use something as a function when it's not.

Function flexibility doesn't end with the function expression! Syntactically, functions can take any expressions as arguments—including *other functions*. Functions that can either take a function as an argument or return a function are called *higher-order functions*. Programming languages with higher-order functions are said to support *first-class functions* because you can treat functions as values in the same way you treat more familiar data types like numbers and vectors.

Take the `map` function (not to be confused with the map data structure), for instance. `map` creates a new list by applying a function to each member of a collection. Here, the `inc` function increments a number by 1:

```
(inc 1.1)
; => 2.1

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

(Note that `map` doesn't return a vector, even though we supplied a vector as an argument. You'll learn why in Chapter 4. For now, just trust that this is okay and expected.)

Clojure's support for first-class functions allows you to build more powerful abstractions than you can in languages without them. Those unfamiliar with this kind of programming think of functions as allowing you to generalize operations over data instances. For example, the `+` function abstracts addition over any specific numbers.

By contrast, Clojure (and all Lisps) allows you to create functions that generalize over processes. `map` allows you to generalize the process of transforming a collection by applying a function—any function—over any collection.

The last detail that you need know about function calls is that Clojure evaluates all function arguments recursively before passing them to the function. Here's how Clojure would evaluate a function call whose arguments are also function calls:

```
(+ (inc 199) (/ 100 (- 7 2)))
(+ 200 (/ 100 (- 7 2))) ; evaluated "(inc 199)"
(+ 200 (/ 100 5)) ; evaluated (- 7 2)
(+ 200 20) ; evaluated (/ 100 5)
220 ; final evaluation
```

The function call kicks off the evaluation process, and all subforms are evaluated before applying the `+` function.

Function Calls, Macro Calls, and Special Forms

In the previous section, you learned that function calls are expressions that have a function expression as the operator. The two other kinds of expressions are *macro calls* and *special forms*. You've already seen a couple of special forms: definitions and `if` expressions.

You'll learn everything there is to know about macro calls and special forms in Chapter 7. For now, the main feature that makes special forms "special" is that, unlike function calls, *they don't always evaluate all of their operands*.

Take `if`, for example. This is its general structure:

```
(if boolean-form
  then-form
  optional-else-form)
```

Now imagine you had an `if` statement like this:

```
(if good-mood
  (tweet walking-on-sunshine-lyrics)
  (tweet mopey-country-song-lyrics))
```

Clearly, in an `if` expression like this, we want Clojure to evaluate only one of the two branches. If Clojure evaluated both `tweet` function calls, your Twitter followers would end up very confused.

Another feature that differentiates special forms is that you can't use them as arguments to functions. In general, special forms implement core Clojure functionality that just can't be implemented with functions. Clojure has only a handful of special forms, and it's pretty amazing that such a rich language is implemented with such a small set of building blocks.

Macros are similar to special forms in that they evaluate their operands differently from function calls, and they also can't be passed as arguments to functions. But this detour has taken long enough; it's time to learn how to define functions!

Defining Functions

Function definitions are composed of five main parts:

- `defn`
- Function name
- A docstring describing the function (optional)
- Parameters listed in brackets
- Function body

Here's an example of a function definition and a sample call of the function:

```
❶ (defn too-enthusiastic
❷   "Return a cheer that might be a bit too enthusiastic"
❸   [name]
❹   (str "OH. MY. GOD! " name " YOU ARE MOST DEFINITELY LIKE THE BEST "
    "MAN SLASH WOMAN EVER I LOVE YOU AND WE SHOULD RUN AWAY SOMEWHERE"))

(too-enthusiastic "Zelda")
; => "OH. MY. GOD! Zelda YOU ARE MOST DEFINITELY LIKE THE BEST MAN SLASH WOMAN
EVER I LOVE YOU AND WE SHOULD RUN AWAY SOMEWHERE"
```

At ❶, `too-enthusiastic` is the name of the function, and it's followed by a descriptive docstring at ❷. The parameter, `name`, is given at ❸, and the function body at ❹ takes the parameter and does what it says on the tin—returns a cheer that might be a bit too enthusiastic.

Let's dive deeper into the docstring, parameters, and function body.

The Docstring

The *docstring* is a useful way to describe and document your code. You can view the docstring for a function in the REPL with `(doc` *`fn-name`*`)`—for example, `(doc map)`. The docstring also comes into play if you use a tool to generate documentation for your code.

Parameters and Arity

Clojure functions can be defined with zero or more parameters. The values you pass to functions are called *arguments*, and the arguments can be of any type. The number of parameters is the function's *arity*. Here are some function definitions with different arities:

```
(defn no-params
  []
  "I take no parameters!")
(defn one-param
  [x]
  (str "I take one parameter: " x))
(defn two-params
  [x y]
  (str "Two parameters! That's nothing! Pah! I will smoosh them "
  "together to spite you! " x y))
```

In these examples, `no-params` is a 0-arity function, `one-param` is 1-arity, and `two-params` is 2-arity.

Functions also support *arity overloading*. This means that you can define a function so a different function body will run depending on the arity. Here's the general form of a multiple-arity function definition. Notice that each arity definition is enclosed in parentheses and has an argument list:

```
(defn multi-arity
  ;; 3-arity arguments and body
  ([first-arg second-arg third-arg]
     (do-things first-arg second-arg third-arg))
  ;; 2-arity arguments and body
  ([first-arg second-arg]
     (do-things first-arg second-arg))
  ;; 1-arity arguments and body
  ([first-arg]
     (do-things first-arg)))
```

Arity overloading is one way to provide default values for arguments. In the following example, `"karate"` is the default argument for the `chop-type` parameter:

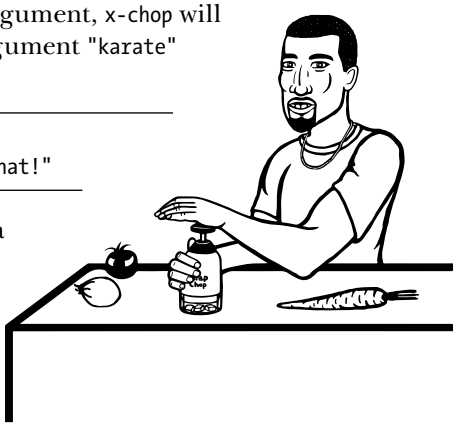
```
(defn x-chop
  "Describe the kind of chop you're inflicting on someone"
  ([name chop-type]
     (str "I " chop-type " chop " name "! Take that!"))
  ([name]
     (x-chop name "karate")))
```

If you call `x-chop` with two arguments, the function works just as it would if it weren't a multiple-arity function:

```
(x-chop "Kanye West" "slap")
; => "I slap chop Kanye West! Take that!"
```

If you call `x-chop` with only one argument, `x-chop` will actually call itself with the second argument `"karate"` supplied:

```
(x-chop "Kanye East")
; => "I karate chop Kanye East! Take that!"
```



It might seem unusual to define a function in terms of itself like this. If so, great! You're learning a new way to do things!

You can also make each arity do something completely unrelated:

```
(defn weird-arity
  ([]
     "Destiny dressed you this morning, my friend, and now Fear is
     trying to pull off your pants. If you give up, if you give in,
     you're gonna end up naked with Fear just standing there laughing
     at your dangling unmentionables! - the Tick")
  ([number]
     (inc number)))
```

The 0-arity body returns a wise quote, and the 1-arity body increments a number. Most likely, you wouldn't want to write a function like this, because it would be confusing to have two function bodies that are completely unrelated.

Clojure also allows you to define variable-arity functions by including a *rest parameter*, as in "put the rest of these arguments in a list with the following name." The rest parameter is indicated by an ampersand (&), as shown at ❶:

```
(defn codger-communication
  [whippersnapper]
  (str "Get off my lawn, " whippersnapper "!!!"))

(defn codger
❶   [& whippersnappers]
  (map codger-communication whippersnappers))

(codger "Billy" "Anne-Marie" "The Incredible Bulk")
; => ("Get off my lawn, Billy!!!"
      "Get off my lawn, Anne-Marie!!!"
      "Get off my lawn, The Incredible Bulk!!!")
```

As you can see, when you provide arguments to variable-arity functions, the arguments are treated as a list. You can mix rest parameters with normal parameters, but the rest parameter has to come last:

```
(defn favorite-things
  [name & things]
  (str "Hi, " name ", here are my favorite things: "
       (clojure.string/join ", " things)))

(favorite-things "Doreen" "gum" "shoes" "kara-te")
; => "Hi, Doreen, here are my favorite things: gum, shoes, kara-te"
```

Finally, Clojure has a more sophisticated way of defining parameters, called *destructuring*, which deserves its own subsection.

Destructuring

The basic idea behind destructuring is that it lets you concisely bind names to values within a collection. Let's look at a basic example:

```
;; Return the first element of a collection
(defn my-first
  [[first-thing]] ; Notice that first-thing is within a vector
  first-thing)

(my-first ["oven" "bike" "war-axe"])
; => "oven"
```

Here, the `my-first` function associates the symbol `first-thing` with the first element of the vector that was passed in as an argument. You tell `my-first` to do this by placing the symbol `first-thing` within a vector.

That vector is like a huge sign held up to Clojure that says, "Hey! This function is going to receive a list or a vector as an argument. Make my life easier by taking apart the argument's structure for me and associating

meaningful names with different parts of the argument!" When destructuring a vector or list, you can name as many elements as you want and also use rest parameters:

```
(defn chooser
  [[first-choice second-choice & unimportant-choices]]
  (println (str "Your first choice is: " first-choice))
  (println (str "Your second choice is: " second-choice))
  (println (str "We're ignoring the rest of your choices. "
                "Here they are in case you need to cry over them: "
                (clojure.string/join ", " unimportant-choices))))

(chooser ["Marmalade", "Handsome Jack", "Pigpen", "Aquaman"])
; => Your first choice is: Marmalade
; => Your second choice is: Handsome Jack
; => We're ignoring the rest of your choices. Here they are in case \
     you need to cry over them: Pigpen, Aquaman
```

Here, the rest parameter `unimportant-choices` handles any number of additional choices from the user after the first and second.

You can also destructure maps. In the same way that you tell Clojure to destructure a vector or list by providing a vector as a parameter, you destructure maps by providing a map as a parameter:

```
(defn announce-treasure-location
❶   [{lat :lat lng :lng}]
  (println (str "Treasure lat: " lat))
  (println (str "Treasure lng: " lng)))

(announce-treasure-location {:lat 28.22 :lng 81.33})
; => Treasure lat: 100
; => Treasure lng: 50
```

Let's look at the line at ❶ in more detail. This is like telling Clojure, "Yo! Clojure! Do me a flava and associate the name `lat` with the value corresponding to the key `:lat`. Do the same thing with `lng` and `:lng`, okay?"

We often want to just break keywords out of a map, so there's a shorter syntax for that. This has the same result as the previous example:

```
(defn announce-treasure-location
  [{:keys [lat lng]}]
  (println (str "Treasure lat: " lat))
  (println (str "Treasure lng: " lng)))
```

You can retain access to the original map argument by using the `:as` keyword. In the following example, the original map is accessed with `treasure-location`:

```
(defn receive-treasure-location
  [{:keys [lat lng] :as treasure-location}]
```

```
  (println (str "Treasure lat: " lat))
  (println (str "Treasure lng: " lng))

  ;; One would assume that this would put in new coordinates for your ship
  (steer-ship! treasure-location))
```

In general, you can think of destructuring as instructing Clojure on how to associate names with values in a list, map, set, or vector. Now, on to the part of the function that actually does something: the function body!

Function Body

The function body can contain forms of any kind. Clojure automatically returns the last form evaluated. This function body contains just three forms, and when you call the function, it spits out the last form, `"joe"`:

```
(defn illustrative-function
  []
  (+ 1 304)
  30
  "joe")

(illustrative-function)
; => "joe"
```

Here's another function body, which uses an `if` expression:

```
(defn number-comment
  [x]
  (if (> x 6)
    "Oh my gosh! What a big number!"
    "That number's OK, I guess"))

(number-comment 5)
; => "That number's OK, I guess"

(number-comment 7)
; => "Oh my gosh! What a big number!"
```

All Functions Are Created Equal

One final note: Clojure has no privileged functions. `+` is just a function, `-` is just a function, and `inc` and `map` are just functions. They're no better than the functions you define yourself. So don't let them give you any lip!

More important, this fact helps demonstrate Clojure's underlying simplicity. In a way, Clojure is very dumb. When you make a function call, Clojure just says, "`map`? Sure, whatever! I'll just apply this and move on." It doesn't care what the function is or where it came from; it treats all functions the same. At its core, Clojure doesn't give two burger flips about addition, multiplication, or mapping. It just cares about applying functions.

As you continue to program with Clojure, you'll see that this simplicity is ideal. You don't have to worry about special rules or syntax for working with different functions. They all work the same!

Anonymous Functions

In Clojure, functions don't need to have names. In fact, you'll use *anonymous* functions all the time. How mysterious! You create anonymous functions in two ways. The first is to use the `fn` form:

```
(fn [param-list]
  function body)
```

Looks a lot like `defn`, doesn't it? Let's try a couple of examples:

```
(map (fn [name] (str "Hi, " name))
     ["Darth Vader" "Mr. Magoo"])
; => ("Hi, Darth Vader" "Hi, Mr. Magoo")

((fn [x] (* x 3)) 8)
; => 24
```

You can treat `fn` nearly identically to the way you treat `defn`. The parameter lists and function bodies work exactly the same. You can use argument destructuring, rest parameters, and so on. You could even associate your anonymous function with a name, which shouldn't come as a surprise (if that does come as a surprise, then . . . Surprise!):

```
(def my-special-multiplier (fn [x] (* x 3)))
(my-special-multiplier 12)
; => 36
```

Clojure also offers another, more compact way to create anonymous functions. Here's what an anonymous function looks like:

```
#(* % 3)
```

Whoa, that looks weird. Go ahead and apply that weird-looking function:

```
(#(* % 3) 8)
; => 24
```

Here's an example of passing an anonymous function as an argument to map:

```
(map #(str "Hi, " %)
     ["Darth Vader" "Mr. Magoo"])
; => ("Hi, Darth Vader" "Hi, Mr. Magoo")
```

This strange-looking style of writing anonymous functions is made possible by a feature called *reader macros*. You'll learn all about those in Chapter 7. Right now, it's okay to learn how to use just these anonymous functions.

You can see that this syntax is definitely more compact, but it's also a little odd. Let's break it down. This kind of anonymous function looks a lot like a function call, except that it's preceded by a hash mark, `#`:

```
;; Function call
(* 8 3)

;; Anonymous function
#(* % 3)
```

This similarity allows you to more quickly see what will happen when this anonymous function is applied. "Oh," you can say to yourself, "this is going to multiply its argument by three."

As you may have guessed by now, the percent sign, `%`, indicates the argument passed to the function. If your anonymous function takes multiple arguments, you can distinguish them like this: `%1`, `%2`, `%3`, and so on. `%` is equivalent to `%1`:

```
(#(str %1 " and " %2) "cornbread" "butter beans")
; => "cornbread and butter beans"
```

You can also pass a rest parameter with `%&`:

```
(#(identity %&) 1 "blarg" :yip)
; => (1 "blarg" :yip)
```

In this case, you applied the identity function to the rest argument. Identity returns the argument it's given without altering it. Rest arguments are stored as lists, so the function application returns a list of all the arguments.

If you need to write a simple anonymous function, using this style is best because it's visually compact. On the other hand, it can easily become unreadable if you're writing a longer, more complex function. If that's the case, use `fn`.

Returning Functions

By now you've seen that functions can return other functions. The returned functions are *closures*, which means that they can access all the variables that were in scope when the function was created. Here's a standard example:

```
(defn inc-maker
  "Create a custom incrementor"
  [inc-by]
  #(+ % inc-by))

(def inc3 (inc-maker 3))

(inc3 7)
; => 10
```

Here, `inc-by` is in scope, so the returned function has access to it even when the returned function is used outside `inc-maker`.

Pulling It All Together

Okay! It's time to use your newfound knowledge for a noble purpose: smacking around hobbits! To hit a hobbit, you'll first model its body parts. Each body part will include its relative size to indicate how likely it is that that part will be hit. To avoid repetition, the hobbit model will include only entries for *left foot*, *left ear*, and so on. Therefore, you'll need a function to fully symmetrize the model, creating *right foot*, *right ear*, and so forth. Finally, you'll create a function that iterates over the body parts and randomly chooses the one hit. Along the way, you'll learn about a few new Clojure tools: `let` expressions, loops, and regular expressions. Fun!

The Shire's Next Top Model

For our hobbit model, we'll eschew such hobbit characteristics as joviality and mischievousness and focus only on the hobbit's tiny body. Here's the hobbit model:

```
(def asym-hobbit-body-parts [{:name "head" :size 3}
                             {:name "left-eye" :size 1}
                             {:name "left-ear" :size 1}
                             {:name "mouth" :size 1}
                             {:name "nose" :size 1}
                             {:name "neck" :size 2}
                             {:name "left-shoulder" :size 3}
                             {:name "left-upper-arm" :size 3}
                             {:name "chest" :size 10}
                             {:name "back" :size 10}
                             {:name "left-forearm" :size 3}
                             {:name "abdomen" :size 6}
                             {:name "left-kidney" :size 1}
                             {:name "left-hand" :size 2}
                             {:name "left-knee" :size 2}
                             {:name "left-thigh" :size 4}
                             {:name "left-lower-leg" :size 3}
                             {:name "left-achilles" :size 1}
                             {:name "left-foot" :size 2}])
```

This is a vector of maps. Each map has the name of the body part and relative size of the body part. (I know that only anime characters have eyes one-third the size of their head, but just go with it, okay?)

Conspicuously missing is the hobbit's right side. Let's fix that. Listing 3-1 is the most complex code you've seen so far, and it introduces some new ideas. But don't worry, because we'll examine it in great detail.

```
(defn matching-part
  [part]
  {:name (clojure.string/replace (:name part) #"^left-" "right-")
   :size (:size part)})

(defn symmetrize-body-parts
  "Expects a seq of maps that have a :name and :size"
  [asym-body-parts]
  (loop [remaining-asym-parts asym-body-parts
         final-body-parts []]
    (if (empty? remaining-asym-parts)
      final-body-parts
      (let [[part & remaining] remaining-asym-parts]
        (recur remaining
               (into final-body-parts
                     (set [part (matching-part part)])))))))
```

Listing 3-1: The `matching-part` and `symmetrize-body-parts` functions

When we call the function `symmetrize-body-parts` on `asym-hobbit-body-parts`, we get a fully symmetrical hobbit:

```
(symmetrize-body-parts asym-hobbit-body-parts)
; => [{:name "head", :size 3}
      {:name "left-eye", :size 1}
      {:name "right-eye", :size 1}
      {:name "left-ear", :size 1}
      {:name "right-ear", :size 1}
      {:name "mouth", :size 1}
      {:name "nose", :size 1}
      {:name "neck", :size 2}
      {:name "left-shoulder", :size 3}
      {:name "right-shoulder", :size 3}
      {:name "left-upper-arm", :size 3}
      {:name "right-upper-arm", :size 3}
      {:name "chest", :size 10}
      {:name "back", :size 10}
      {:name "left-forearm", :size 3}
      {:name "right-forearm", :size 3}
      {:name "abdomen", :size 6}
      {:name "left-kidney", :size 1}
      {:name "right-kidney", :size 1}
      {:name "left-hand", :size 2}
      {:name "right-hand", :size 2}
      {:name "left-knee", :size 2}
      {:name "right-knee", :size 2}
      {:name "left-thigh", :size 4}
      {:name "right-thigh", :size 4}
      {:name "left-lower-leg", :size 3}
      {:name "right-lower-leg", :size 3}
```

```
{:name "left-achilles", :size 1}
{:name "right-achilles", :size 1}
{:name "left-foot", :size 2}
{:name "right-foot", :size 2}]
```

Let's break down this code!

let

In the mass of craziness in Listing 3-1, you can see a form of the structure (`let ...`). Let's build up an understanding of `let` one example at a time, and then examine the full example from the program once we're familiar with all the pieces.

`let` binds names to values. You can think of `let` as short for *let it be*, which is also a beautiful Beatles song about programming. Here's an example:

```
(let [x 3]
  x)
; => 3

(def dalmatian-list
  ["Pongo" "Perdita" "Puppy 1" "Puppy 2"])
(let [dalmatians (take 2 dalmatian-list)]
  dalmatians)
; => ("Pongo" "Perdita")
```

In the first example, you bind the name `x` to the value `3`. In the second, you bind the name `dalmatians` to the result of the expression (`take 2 dalmatian-list`), which was the list (`"Pongo" "Perdita"`). `let` also introduces a new *scope*:

```
(def x 0)
(let [x 1] x)
; => 1
```

Here, you first bind the name `x` to the value `0` using `def`. Then, `let` creates a new scope in which the name `x` is bound to the value `1`. I think of scope as the context for what something refers to. For example, in the phrase "please clean up these butts," *butts* means something different depending on whether you're working in a maternity ward or on the custodial staff of a cigarette manufacturers convention. In this code snippet, you're saying, "I want `x` to be `0` in the global context, but within the context of this `let` expression, it should be `1`."

You can reference existing bindings in your `let` binding:

```
(def x 0)
(let [x (inc x)] x)
; => 1
```

In this example, the x in (inc x) refers to the binding created by (def x 0). The resulting value is 1, which is then bound to the name x within a new scope created by let. Within the confines of the let form, x refers to 1, not 0.

You can also use rest parameters in let, just like you can in functions:

```
(let [[pongo & dalmatians] dalmatian-list]
  [pongo dalmatians])
; => ["Pongo" ("Perdita" "Puppy 1" "Puppy 2")]
```

Notice that the value of a let form is the last form in its body that is evaluated. let forms follow all the destructuring rules introduced in "Calling Functions" on page 48. In this case, [pongo & dalmatians] destructured dalmatian-list, binding the string "Pongo" to the name pongo and the list of the rest of the dalmatians to dalmatians. The vector [pongo dalmatians] is the last expression in let, so it's the value of the let form.

let forms have two main uses. First, they provide clarity by allowing you to name things. Second, they allow you to evaluate an expression only once and reuse the result. This is especially important when you need to reuse the result of an expensive function call, like a network API call. It's also important when the expression has side effects.

Let's have another look at the let form in our symmetrizing function so we can understand exactly what's going on:

```
(let [[part & remaining] remaining-asym-parts]
  (recur remaining
         (into final-body-parts
               (set [part (matching-part part)]))))
```

This code tells Clojure, "Create a new scope. Within it, associate part with the first element of remaining-asym-parts. Associate remaining with the rest of the elements in remaining-asym-parts."

As for the body of the let expression, you'll learn about the meaning of recur in the next section. The function call

```
(into final-body-parts
  (set [part (matching-part part)]))
```

first tells Clojure, "Use the set function to create a set consisting of part and its matching part. Then use the function into to add the elements of that set to the vector final-body-parts." You create a set here to ensure you're adding unique elements to final-body-parts because part and (matching-part part) are sometimes the same thing, as you'll see in the upcoming section on regular expressions. Here's a simplified example:

```
(into [] (set [:a :a]))
; => [:a]
```

First, (set [:a :a]) returns the set #{:a}, because sets don't contain duplicate elements. Then (into [] #{:a}) returns the vector [:a].

Back to let: notice that part is used multiple times in the body of the let. If we used the original expressions instead of using the names part and remaining, it would be a mess! Here's an example:

```
(recur (rest remaining-asym-parts)
       (into final-body-parts
             (set [(first remaining-asym-parts) (matching-part (first
remaining-asym-parts))])))
```

So, let is a handy way to introduce local names for values, which helps simplify the code.

loop

In our symmetrize-body-parts function we use loop, which provides another way to do recursion in Clojure. Let's look at a simple example:

```
(loop [iteration 0]
  (println (str "Iteration " iteration))
  (if (> iteration 3)
    (println "Goodbye!")
    (recur (inc iteration))))
; => Iteration 0
; => Iteration 1
; => Iteration 2
; => Iteration 3
; => Iteration 4
; => Goodbye!
```

The first line, loop [iteration 0], begins the loop and introduces a binding with an initial value. On the first pass through the loop, iteration has a value of 0. Next, it prints a short message. Then, it checks the value of iteration. If the value is greater than 3, it's time to say Goodbye. Otherwise, we recur. It's as if loop creates an anonymous function with a parameter named iteration, and recur allows you to call the function from within itself, passing the argument (inc iteration).

You could in fact accomplish the same thing by just using a normal function definition:

```
(defn recursive-printer
  ([]
     (recursive-printer 0))
  ([iteration]
     (println iteration)
     (if (> iteration 3)
       (println "Goodbye!")
       (recursive-printer (inc iteration)))))
(recursive-printer)
```

```
; => Iteration 0
; => Iteration 1
; => Iteration 2
; => Iteration 3
; => Iteration 4
; => Goodbye!
```

But as you can see, this is a bit more verbose. Also, `loop` has much better performance. In our symmetrizing function, we'll use `loop` to go through each element in the asymmetrical list of body parts.

Regular Expressions

Regular expressions are tools for performing pattern matching on text. The literal notation for a regular expression is to place the expression in quotes after a hash mark:

```
#"regular-expression"
```

In the function `matching-part` in Listing 3-1, `clojure.string/replace` uses the regular expression `#"^left-"` to match strings starting with `"left-"` in order to replace `"left-"` with `"right-"`. The carat, `^`, is how the regular expression signals that it will match the text `"left-"` only if it's at the beginning of the string, which ensures that something like `"cleft-chin"` won't match. You can test this with `re-find`, which checks whether a string matches the pattern described by a regular expression, returning the matched text or `nil` if there is no match:

```
(re-find #"^left-" "left-eye")
; => "left-"

(re-find #"^left-" "cleft-chin")
; => nil

(re-find #"^left-" "wongleblart")
; => nil
```

Here are a couple of examples of `matching-part` using a regex to replace `"left-"` with `"right-"`:

```
(defn matching-part
  [part]
  {:name (clojure.string/replace (:name part) #"^left-" "right-")
   :size (:size part)})
(matching-part {:name "left-eye" :size 1})
; => {:name "right-eye" :size 1}]

(matching-part {:name "head" :size 3})
; => {:name "head" :size 3}]
```

Notice that the name `"head"` is returned as is.

Symmetrizer

Now let's go back to the full symmetrizer and analyze it in more detail:

```
(def asym-hobbit-body-parts [{:name "head" :size 3}
                             {:name "left-eye" :size 1}
                             {:name "left-ear" :size 1}
                             {:name "mouth" :size 1}
                             {:name "nose" :size 1}
                             {:name "neck" :size 2}
                             {:name "left-shoulder" :size 3}
                             {:name "left-upper-arm" :size 3}
                             {:name "chest" :size 10}
                             {:name "back" :size 10}
                             {:name "left-forearm" :size 3}
                             {:name "abdomen" :size 6}
                             {:name "left-kidney" :size 1}
                             {:name "left-hand" :size 2}
                             {:name "left-knee" :size 2}
                             {:name "left-thigh" :size 4}
                             {:name "left-lower-leg" :size 3}
                             {:name "left-achilles" :size 1}
                             {:name "left-foot" :size 2}])


(defn matching-part
  [part]
  {:name (clojure.string/replace (:name part) #"^left-" "right-")
   :size (:size part)})

❶ (defn symmetrize-body-parts
    "Expects a seq of maps that have a :name and :size"
    [asym-body-parts]
❷   (loop [remaining-asym-parts asym-body-parts
           final-body-parts []]
❸     (if (empty? remaining-asym-parts)
        final-body-parts
❹       (let [[part & remaining] remaining-asym-parts]
❺         (recur remaining
                 (into final-body-parts
                       (set [part (matching-part part)])))))))
```

The `symmetrize-body-parts` function (starting at ❶) employs a general strategy that is common in functional programming. Given a sequence (in this case, a vector of body parts and their sizes), the function continuously splits the sequence into a *head* and a *tail*. Then it processes the head, adds it to some result, and uses recursion to continue the process with the tail.

We begin looping over the body parts at ❷. The tail of the sequence will be bound to `remaining-asym-parts`. Initially, it's bound to the full sequence passed to the function: `asym-body-parts`. We also create a result sequence, `final-body-parts`; its initial value is an empty vector.

If `remaining-asym-parts` is empty at ❸, that means we've processed the entire sequence and can return the result, `final-body-parts`. Otherwise, at ❹ we split the list into a head, `part`, and tail, `remaining`.

At ❺, we recur with `remaining`, a list that gets shorter by one element on each iteration of the loop, and the `(into)` expression, which builds our vector of symmetrized body parts.

If you're new to this kind of programming, this code might take some time to puzzle out. Stick with it! Once you understand what's happening, you'll feel like a million bucks!

Better Symmetrizer with reduce

The pattern of *process each element in a sequence and build a result* is so common that there's a built-in function for it called `reduce`. Here's a simple example:

```
;; sum with reduce
(reduce + [1 2 3 4])
; => 10
```

This is like telling Clojure to do this:

```
(+ (+ (+ 1 2) 3) 4)
```

The `reduce` function works according to the following steps:

1. Apply the given function to the first two elements of a sequence. That's where `(+ 1 2)` comes from.
2. Apply the given function to the result and the next element of the sequence. In this case, the result of step 1 is `3`, and the next element of the sequence is `3` as well. So the final result is `(+ 3 3)`.
3. Repeat step 2 for every remaining element in the sequence.

`reduce` also takes an optional initial value. The initial value here is `15`:

```
(reduce + 15 [1 2 3 4])
```

If you provide an initial value, `reduce` starts by applying the given function to the initial value and the first element of the sequence rather than the first two elements of the sequence.

One detail to note is that, in these examples, `reduce` takes a collection of elements, `[1 2 3 4]`, and returns a single number. Although programmers often use `reduce` this way, you can also use `reduce` to return an even larger collection than the one you started with, as we're trying to do with `symmetrize-body-parts`. `reduce` abstracts the task "process a collection

and build a result," which is agnostic about the type of result returned. To further understand how `reduce` works, here's one way that you could implement it:

```
(defn my-reduce
  ([f initial coll]
   (loop [result initial
          remaining coll]
     (if (empty? remaining)
       result
       (recur (f result (first remaining)) (rest remaining)))))
  ([f [head & tail]]
   (my-reduce f head tail)))
```

We could reimplement our symmetrizer as follows:

```
(defn better-symmetrize-body-parts
  "Expects a seq of maps that have a :name and :size"
  [asym-body-parts]
  (reduce (fn [final-body-parts part]
            (into final-body-parts (set [part (matching-part part)])))
          []
          asym-body-parts))
```

Groovy! One immediately obvious advantage of using `reduce` is that you write less code overall. The anonymous function you pass to `reduce` focuses only on processing an element and building a result. The reason is that `reduce` handles the lower-level machinery of keeping track of which elements have been processed and deciding whether to return a final result or to recur.

Using `reduce` is also more expressive. If readers of your code encounter `loop`, they won't be sure exactly what the loop is doing without reading all of the code. But if they see `reduce`, they'll immediately know that the purpose of the code is to process the elements of a collection to build a result.

Finally, by abstracting the `reduce` process into a function that takes another function as an argument, your program becomes more composable. You can pass the `reduce` function as an argument to other functions, for example. You could also create a more generic version of `symmetrize-body-parts`, say, `expand-body-parts`. This could take an *expander* function in addition to a list of body parts and would let you model more than just hobbits. For example, you could have a spider expander that could multiply the numbers of eyes and legs. I'll leave it up to you to write that because I am evil.

Hobbit Violence

My word, this is truly Clojure for the Brave and True! To put the capstone on your work, here's a function that determines which part of a hobbit is hit:

```
(defn hit
  [asym-body-parts]
```

```
(let [sym-parts (❶better-symmetrize-body-parts asym-body-parts)
      ❷body-part-size-sum (reduce + (map :size sym-parts))
      target (rand body-part-size-sum)]
  ❸(loop [[part & remaining] sym-parts
          accumulated-size (:size part)]
    (if (> accumulated-size target)
      part
      (recur remaining (+ accumulated-size (:size (first remaining)))))))
```

`hit` works by taking a vector of asymmetrical body parts, symmetrizing it at ❶, and then summing the sizes of the parts at ❷. Once we sum the sizes, it's like each number from 1 through `body-part-size-sum` corresponds to a body part; 1 might correspond to the left eye, and 2, 3, 4 might correspond to the head. This makes it so when you hit a body part (by choosing a random number in this range), the likelihood that a particular body part is hit will depend on the size of the body part.

Finally, one of these numbers is randomly chosen, and then we use `loop` at ❸ to find and return the body part that corresponds to the number. The loop does this by keeping track of the accumulated sizes of parts that we've checked and checking whether the accumulated size is greater than the target. I visualize this process as lining up the body parts with a row of numbered slots. After I line up a body part, I ask myself, "Have I reached the target yet?" If I have, that means the body part I just lined up was the one hit. Otherwise, I just keep lining up those parts.

For example, say that your list of parts is *head*, *left eye*, and *left hand*, like in Figure 3-1. After taking the first part, the head, the accumulated size is 3. The body part is hit if the accumulated size is greater than the target, so if the target is 0, 1, or 2, then the head was hit. Otherwise, you take the next part, the left eye, and increase the accumulated size to 4, yielding a hit if the target is 3. Similarly, the left hand gets hit if the target is 4 or 5.

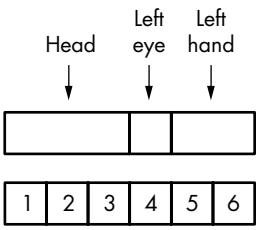


Figure 3-1: Body parts correspond to ranges of numbers and get hit if the target falls within that range.

Here are some sample runs of the `hit` function:

```
(hit asym-hobbit-body-parts)
; => {:name "right-upper-arm", :size 3}

(hit asym-hobbit-body-parts)
; => {:name "chest", :size 10}

(hit asym-hobbit-body-parts)
; => {:name "left-eye", :size 1}
```

Oh my god, that poor hobbit! You monster!

Summary

This chapter gave you a whirlwind tour of how to *do stuff* in Clojure. You now know how to represent information using strings, numbers, maps, keywords, vectors, lists, and sets, and how to name these representations with `def` and `let`. You've learned about how flexible functions are and how to create your own functions. Also, you've been introduced to Clojure's philosophy of simplicity, including its uniform syntax and its emphasis on using large libraries of functions on primitive data types.

Chapter 4 will take you through a detailed examination of Clojure's core functions, and Chapter 5 explains the functional programming mindset. This chapter has shown you how to write Clojure code—the next two will show you how to write Clojure *well*.

At this point I recommend, with every fiber of my being, that you start writing code. There is no better way to solidify your Clojure knowledge. The Clojure Cheat Sheet (*http://clojure.org/cheatsheet/*) is a great reference that lists all the built-in functions that operate on the data structures covered in this chapter.

The following exercises will really tickle your brain. If you'd like to test your new skills even more, try some Project Euler challenges at *http://www.projecteuler.net/*. You could also check out 4Clojure (*http://www.4clojure.com/problems/*), an online set of Clojure problems designed to test your knowledge. Just write something!

Exercises

These exercises are meant to be a fun way to test your Clojure knowledge and to learn more Clojure functions. The first three can be completed using only the information presented in this chapter, but the last three will require you to use functions that haven't been covered so far. Tackle the last three if you're really itching to write more code and explore Clojure's standard library. If you find the exercises too difficult, revisit them after reading Chapters 4 and 5—you'll find them much easier.

1. Use the `str`, `vector`, `list`, `hash-map`, and `hash-set` functions.
2. Write a function that takes a number and adds 100 to it.
3. Write a function, `dec-maker`, that works exactly like the function `inc-maker` except with subtraction:

```
(def dec9 (dec-maker 9))
(dec9 10)
; => 1
```

4. Write a function, `mapset`, that works like `map` except the return value is a set:

```
(mapset inc [1 1 2 2])
; => #{2 3}
```

5. Create a function that's similar to `symmetrize-body-parts` except that it has to work with weird space aliens with radial symmetry. Instead of two eyes, arms, legs, and so on, they have five.
6. Create a function that generalizes `symmetrize-body-parts` and the function you created in Exercise 5. The new function should take a collection of body parts and the number of matching body parts to add. If you're completely new to Lisp languages and functional programming, it probably won't be obvious how to do this. If you get stuck, just move on to the next chapter and revisit the problem later.