Extracted from:

Getting Clojure

Build Your Functional Skills One Idea at a Time

This PDF file contains pages extracted from *Getting Clojure*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2018 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Getting Clojure

Build Your Functional Skills One Idea at a Time

Russ Olsen

edited by Michael Swaine

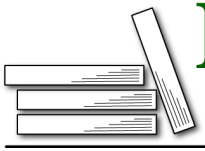
Getting Clojure

Build Your Functional Skills One Idea at a Time

Russ Olsen

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Michael Swaine
Copy Editor: Candace Cunningham
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2018 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-68050-300-5
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—May 2018

To

Mikey

Felicia

Jackson

Charlie & Jennifer

Tim & Emily & Evan

Nicholas & Jonathan

Meg & Alex & Zachary

and

Scott Downie

The future is in your hands.

CHAPTER 6

Functional Things

Even people who don't know very much about Clojure tend to know one thing about it: Clojure is a functional programming language. But what does it mean for a programming language to be functional? After all, no matter if they call it a function or a method or a subroutine, all mainstream programming languages come equipped with some kind of *call it with arguments and get back a result* thing. What makes functional languages like Clojure different?

In this chapter we're going to take a hard look at the thing that makes Clojure a functional programming language: the idea that functions are first-class values, values that you can manipulate with the language. Along the way we'll discover that much of the power of Clojure comes not from writing functions that do things, but in writing functions that create functions that do things. If all this sounds a bit like science fiction, well, prepare for light speed.

Functions Are Values

Let's start our adventures in functional programming by imagining that we have decided to add price and genre to the maps we've been using to keep track of our books, like this:

functional/examples.clj

```
(def dracula {:title "Dracula"
              :author "Stoker"
              :price 1.99
              :genre :horror})
```

Further, let's imagine that we need to write some code to distinguish the books based on an arbitrary price:

```
(defn cheap? [book]
  (when (<= (:price book) 9.99)
    book))
```

• Click HERE to purchase this book now. discuss

```
(defn pricey? [book]
  (when (> (:price book) 9.99)
    book))

(cheap? dracula)        ; Yes!
(pricey? dracula)       ; No!
```

or the genre:

```
(defn horror? [book]
  (when (= (:genre book) :horror)
    book))

(defn adventure? [book]
  (when (= (:genre book) :adventure)
    book))

(horror? dracula)       ; Yes!
(adventure? dracula)    ; Nope!
```

The only halfway interesting thing about these functions is that they take advantage of Clojure's truthy logic and return `nil` when the book fails the test, and the book map itself—which is truthy—when it passes.

We might also be interested in combinations of price and genre:

```
(defn cheap-horror? [book]
  (when (and (cheap? book)
             (horror? book))
    book))

(defn pricy-adventure? [book]
  (when (and (pricey? book)
             (adventure? book))
    book))
```

We could write functions like this all day. What about cheap books by some author or the expensive books entitled *Possession*?

Possession

It turns out there's a remarkable number of novels called *Possession*, with at least a dozen in print as I write this.

The key—and unfortunate—word here is *write*. When you are building real systems you don't want to spend your time writing these kinds of combinations by hand. What you want is to code the basic operations and then create the combinations dynamically. Fortunately, all you need to get out of the hand-coding business is to realize that in Clojure functions have something in common with numbers and strings and Booleans and vectors. Like these more mundane things, *functions are values*.

This means that when you evaluate the name of a function you've defined with defn, perhaps like this:

```
cheap?
```

you will see something like this:

```
#object[user$cheap_QMARK_ 0x71454b9d "user$cheap_QMARK_@71454b9d"]
```

The #object[user$cheap_QMARK_..."] is the semi-intelligible string that gets output when Clojure tries to print the function that knows a cheap book from an expensive one. You can also bind that function value to another symbol:

```
(def reasonably-priced? cheap?)
```

Do that, and reasonably-priced? is now an alternate name for our thrifty function:

```
(reasonably-priced? dracula)        ; Yes!
```

You can also pass function values to other functions. To take a silly example, we could do this:

```
(defn run-with-dracula [f]
  (f dracula))
```

run-with-dracula does exactly what the name suggests: it evaluates a function with the dracula value as an argument. Which function? The one that you pass to run-with-dracula:

```
(run-with-dracula pricey?)      ; Nope.

(run-with-dracula horror?)      ; Yes!
```

More practically, this idea of functions as values gives us an easy way of combining our predicates:

```
(defn both? [first-predicate-f second-predicate-f book]
  (when (and (first-predicate-f book)
             (second-predicate-f book))
    book))

(both? cheap? horror? dracula)     ; Yup!

(both? pricey? adventure? dracula) ; Nope!
```

The only difference between the more general-purpose both? function and the very specific cheap-horror? is that both? lets you pass in your pair of predicate functions, which means you can use it to run your books by any two predicates you can cook up.

Functions on the Fly

There's something else you can do with functional values: you can manufacture new ones, on the fly. In much the same way you can make a new number with (+ 2 3) or (* 5 x), you can use fn to create new functions. Here, for example, we manufacture a new function with fn, one that doubles its argument:

```
(fn [n] (* 2 n))
```

As you can see from the example, using fn is a lot like using defn, except that you leave out the name. Like defn, fn creates a new function, essentially a packaged bit of code. The difference between fn and defn is that fn doesn't bind its newborn bundle of code to a name; you just get the function value. So what can you do with a function value? Anything you can do with any other value. You can, for example, print it:

```
(println "A function:" (fn [n] (* 2 n)))
```

or bind it to a symbol:

```
(def double-it (fn [n] (* 2 n)))
```

and, most importantly, call it:

```
(double-it 10)          ; Gives you 20.
((fn [n] (* 2 n)) 10)   ; Also gives you 20.
```

Returning to our book example, here is a nameless function that does the same thing as cheap?:

```
(fn [book]
  (when (<= (:price book) 9.99)
    book))
```

Armed with fn, we can write functions that produce functions:

```
(defn cheaper-f [max-price]
  (fn [book]
    (when (<= (:price book) max-price)
      book)))
```

It's important to understand just how meta we've gone here: cheaper-f is a function that produces a whole family of bargain-spotting functions, each with its own idea of what constitutes a bargain.

```
;; Define some helpful functions.

(def real-cheap? (cheaper-f 1.00))
(def kind-of-cheap? (cheaper-f 1.99))
(def marginally-cheap? (cheaper-f 5.99))
```

• Click HERE to purchase this book now. discuss

```
;; And use them.

(real-cheap? dracula)          ; Nope.
(kind-of-cheap? dracula)       ; Yes.
(marginally-cheap? dracula)    ; Indeed.
```

If this all looks less than spectacular, look again. The thing to note is that a function produced by fn picks up and remembers the parameters around when the fn was run. So in the last example, the function produced when you call (cheaper-f 1.00) will remember that max-price is 1.00 while the function produced by (cheaper-f 5.99) will remember max-price as 5.99.

Going a step further, we can write a function that manufactures both?-like functions:

```
(defn both-f [predicate-f-1 predicate-f-2]
  (fn [book]
    (when (and (predicate-f-1 book) (predicate-f-2 book))
      book)))
```

With both-f we can then build a whole family of book-discriminating functions:

```
(def cheap-horror? (both-f cheap? horror?))

(def real-cheap-adventure? (both-f real-cheap? adventure?))

(def real-cheap-horror? (both-f real-cheap? horror?))
```

And then go up yet another level of meta:

```
(def cheap-horror-possession?
  (both-f cheap-horror?
    (fn [book] (= (:title book) "Possession"))))
```

This idea of a function grabbing and remembering the bindings that existed when the function was born is called a *closure*. We say that the function *closes* over the scope in which it was defined. More than anything else, the twin ideas of functions as values and *closure* are at the heart of what makes Clojure the programming language it is, and might explain the name as well.

A Functional Toolkit

Since so much of Clojure programming revolves around creating, combining, and using functions, it's unsurprising that the language provides a fair number of functions aimed at easing the job.

Take, for example, the apply function. It tackles the surprisingly common situation where you have a function and the arguments that you want to call that function with *in a collection*. In other words, instead of having this:

```
(+ 1 2 3 4)          ; Gives you 10.
```

what if you had the function (+ in this case) and the arguments, like this:

```
(def the-function +)
(def args [1 2 3 4])
```

Enter apply. You supply a function and a collection of arguments, and apply will call that function with the arguments, returning the result. Armed with apply we can get the job done like this:

```
(apply the-function args)   ; (the-function args0 args1 args2 ...)
```

The apply function is particularly useful for converting from one kind of value to another. Thus, if you have a vector like this:

```
(def v ["The number " 2 " best selling " "book."])
```

you can use the combination of apply and str to turn it into a string:

```
;; More or less the same as:
;; (str "The number " 2 " best selling " "book.")
(apply str v)
```

or apply and list to turn it into a list:

```
;; More or less the same as:
;; (list "The number " 2 " best selling " "book.")
(apply list v)
```

and then back into a vector:

```
(apply vector (apply list v))
```

Another incredibly useful function is partial. It's called partial because it partially fills in the arguments for an existing function, producing a new function of fewer arguments in the process. For example, Clojure includes a function called inc that adds one to the number you pass in, so that (inc 1) gives you 2 and (inc 41) is 42. It's easy enough to cook up your own version of inc:

```
(defn my-inc [n] (+ 1 n))
```

But consider that my-inc is simply filling in the first argument of + with 1. Which is exactly the kind of thing that partial does:

```
(def my-inc (partial + 1))
```

Returning to our book example, we can use partial to rework and simplify our cheapness-discriminating functions:

```
(defn cheaper-than [max-price book]
  (when (<= (:price book) max-price)
    book))
```

```
(def real-cheap? (partial cheaper-than 1.00))
(def kind-of-cheap? (partial cheaper-than 1.99))
(def marginally-cheap? (partial cheaper-than 5.99))
```

Each call to partial there is giving us back a new function that—when called—calls cheaper-than with one of the prices as the first argument.

Another handy function-producing function that comes packaged with Clojure is complement. With complement every day is opposite day. complement wraps the function that you supply with a call to not, producing a new function that is, well, the complement of the original. For example, earlier we wrote adventure?, which could tell adventure books from those of other genres:

```
(defn adventure? [book]
  (when (= (:genre book) :adventure)
    book))
```

But what if we needed a function that looked for nonadventure books? Clearly we could write it by hand:

```
(defn not-adventure? [book] (not (adventure? book)))
```

But we did say we were trying to get out of the hand-coding business, so instead we turn to complement:

```
(def not-adventure? (complement adventure?))
```

As I say, complement produces a function that returns the truthy negation of the function that you pass to complement.

One more example of a function-generating function is every-pred. It combines predicate functions into a single function that *ands* them all together. With every-pred we can dispense with our home-grown both-f:

```
(def cheap-horror? (every-pred cheap? horror?))
```

Even better, every-pred will take any number of arguments, so that this:

```
(def cheap-horror-possession?
  (every-pred
    cheap?
    horror?
    (fn [book] (= (:title book) "Possession"))))
```

will do exactly what you want it to do.

Function Literals

Another way that Clojure comes to your aid in creating new functions is to supply an alternate, minimalistic syntax for defining them. So for those

moments when even the sleek lines of fn seem like too much syntactical overhead, you can use a function literal: just a # followed by the function body, wrapped in the usual parentheses. Here, for example, are the guts of adventure? recast as a function literal:

```
#(when (= (:genre %1) :adventure) %1)
```

Note there are no named arguments in function literals; instead they use the very shell script-ish notation of %1 to stand for the first argument, %2 for the second argument, and so on. So if we needed a function that would double a number, we might use partial or we might do this:

```
#(* 2 %1)
```

Or if we need a function to add three numbers together, we might cook this up:

```
#(+ %1 %2 %3)
```

There are a few things to keep in mind about function literals, or *lambdas*, as they are sometimes known. First, function literals and fn produce exactly the same kind of thing (a function value); the only difference is the syntax.

Second, remember that Clojure infers the number of arguments that your literally defined function takes from the highest-numbered argument in the function body. Thus, if we modified our *double the number* function into this:

```
#(* 2 %11)
```

we would end up with a (very inconvenient) function that takes 11 arguments and ignores the first 10.

Finally, function literals have a special feature aimed directly at the very common case of creating a one-argument function. If the function you're building takes a single argument, you can use plain old %—without a number—for the one and only argument. So a minimal version of our number doubler would be as follows:

```
#(* % 2)
```

The trade-off between defining a full-blown named function with defn and using the streamlined fn or a completely stripped-down function literal is one of those familiar software-engineering choices. If you're going to be reusing the function, then by all means use defn and give it a name. Giving your function a name is also worthwhile if the name will help you (and those who come later) understand some intricate bit of code. You also probably will want to use defn on lengthy functions to visually break up the code.

• Click HERE to purchase this book now. discuss

On the other hand, fn and function literals are wonderful when you're cooking up short, single-use functions and when you need to take advantage of a closure to pick up some values.

The choice between fn and function literals centers on complexity and number of arguments. Lean toward function literals for really short, simple functions. If, for example, you need a function to double a number, then by all means write #(* % 2). Conversely, lean toward fn if you have a longer function, and especially one that takes more than a very few arguments. Examples aside, no one really writes function literals that have a %11.

In the Wild

And now we have the answer to the opening question of this chapter: the thing that makes Clojure a functional programming language is that you do basic things by writing functions and you do more sophisticated things by treating the functions as values—values that you can pass around and call and combine.

Possibly the best demonstration of the *functions are values* idea can be found inside the machinery of defn itself. defn is just a thin layer over def and fn. So when you define a new function with defn, perhaps this:

```
(defn say-welcome [what]
  (println "Welcome to" what "!"))
```

what gets evaluated is something like this:

```
(def say-welcome
  (fn [what] (println "Welcome to" what "!")))
```

As the name suggests, defn is def plus fn.

If you are not used to the idea, functional values can seem a bit special and magical, the kind of technique you would use only in extreme circumstances. Not so; in Clojure they are just part of the everyday programming landscape.

Take, for example, the mundane update function. As the name suggests, you use update to modify values, specifically the values inside of a map.

You Can't Modify That Map

To be precise, update produces a new map that's a lot like the input map, only different. But I'm getting as tired of writing that as you are of reading it.

So if we wanted to record that we've sold another copy of a book, we might write this:

```
;; Start with 1,000 copies sold.
(def book {:title "Emma" :copies 1000})
;; Now we have 1,001.
(def new-book (update book :copies inc))
```

As you can see, update takes three parameters: the map, the key whose value you want to update, and *a function* to do the updating. Your function will get called with the old value of the key (in this case 1000) and the map you get back will be just like the old map, except that the key will have the result of evaluating the function.

If you happen to have nested maps, you can reach for the slightly less mundane update-in function, which works like update but will also let you drill down through several layers of maps using a pathlike vector of keys:

```
(def by-author
  {:name "Jane Austen"
   :book {:title "Emma" :copies 1000}})
(def new-by-author (update-in by-author [:book :copies] inc))
```

But to see how much you can do with functional values, look no further than Ring,[1] the popular Clojure library that helps you build web applications.

To build a web application with Ring you first need to utter the proper incantation to load Ring (we'll talk about require in Chapter 9, *Namespaces*, on page ?):

```
(ns ring-example.core
  (:require [ring.adapter.jetty :as jetty]))
```

Then you create a function that takes in an HTTP request—in the form of a map—and returns a response, also in the form of a map, like this:

```
(defn handler [request]
  {:status 200
   :headers {"Content-Type" "text/html"}
   :body "Hello from your web application!"})
```

Having written your function, you now need to tell Ring that *this* is the function Ring should look to when a web request comes in. You can do that by passing the handler function to Ring's run-jetty function, which kicks off a simple web server called Jetty:

```
(defn -main []
  (jetty/run-jetty handler {:port 8080}))
```

1. https://github.com/ring-clojure/ring

And now your handler function will get called for requests on port 8080.

Aside from plain handlers, Ring applications also commonly use *middleware*. Middleware are functions that take a handler function as a parameter and return a new handler function. Ring programmers use middleware to layer additional features onto their handlers. For example, we might define a middleware function that logs the response:

```
(defn log-value
  "Log the message and the value. Returns the value."
  [msg value]
  (println msg value)
  value)

(defn wrap-logging
  "Return a function that logs the response."
  [msg handler]
  (fn [request]
    (log-value msg (handler request))))
```

and a second handler to specify the content type:

```
(defn wrap-content-type
  "Return a function that sets the response content type."
  [handler content-type]
  (fn [request]
    (assoc-in
      (handler request)
      [:headers "Content-Type"]
      content-type)))
```

As I say, middleware functions take in a handler—a function—and return another handler. The new handler typically runs the old handler while adding its own goodness along the way. Our first middleware function, wrap-logging, runs the handler function passed to it, prints the response, and then returns the response. The second middleware function does something more interesting: it adds a header (for the content type) to the response.

Assoc-in?

You may have noticed that the content-type handler in the example uses a function called assoc-in. This function is a lot like assoc in that it adds a new key/value association to a map. The difference is that you pass assoc-in a vector of keys and it will go spelunking down through multiple levels of maps for you. To put it another way, in the same way that update-in is the multistory version of update, assoc-in is the multistory version of assoc.

Traditionally Ring applications call the final, fully wrapped handler the app, short for application. So this is how we set up our final app and kick off Ring:

```
(defn handler [request]
  {:status 200
   :body "Hello from your web application!"})

(def app
  (wrap-logging
    "Final response:"
    (wrap-content-type handler "text/html")))
```

You can get a feeling for the power of the *functions as values* view of the world by noting that in the preceding example we're logging the final response—that is, the response after wrap-content-type has had its say. But with a little rearranging we can log the response before the content type gets added:

```
(def app
  (wrap-content-type
    (wrap-logging  "Initial response:" handler)
    "text/html"))
```

or we can log both:

```
(def app
  (wrap-logging
    "Final response:"
    (wrap-content-type
      (wrap-logging  "Initial response:" handler)
      "text/html")))
```

This last bit of code is a great example of the power of functional programming. It assembles four separate functions, three of them dynamically generated, into a working whole that is greater than the sum of its parts.

Staying Out of Trouble

Going from the simple-minded idea of functions as something you write and call manually to the *functions as values* idea has some interesting implications. Chief among these is that you don't always know the exact context in which your function will be called. Since functions are values, they can get passed around and evaluated any number of times:

```
(defn execute-that-function-three-times [your-function]
  (your-function)
  (your-function)
  (your-function))
```

Or they might get called sometime later. For example, we might use Thread/sleep to wait 372 milliseconds before calling your function:

```
(defn execute-that-function-later [your-function]
  (Thread/sleep 372)      ; Pause for 372 ms.
  (your-function))
```

Or it might never get called:

```
(defn execute-that-function-never [your-function]
  (+ 2 2))
```

Or it might get called in some odd combination:

```
(defn some-odd-combination [your-function]
  (execute-that-function-three-times
    #(execute-that-function-later your-function)))
```

Given all this, the functional programmer's Prime Directive is simple: try to write functions that *don't care* about the context in which they are called. In practice this means you should avoid writing functions that rely on or generate side effects. In functional programming, the best functions are the ones that look only at their arguments and produce only their return value. They don't read, create, or delete files; they don't roll the current time or date into their answer; and they certainly don't consult the user for input. They just look at their arguments and come up with a result. We even have an appropriately positive term for functions that follow these rules. We call them *pure functions*.

The good news is that pure functions are not hard to write. In fact, take out the printfs that we've sprinkled here and there, and all the functions we've written in this chapter are indeed pure. From adventure? to cheap-horror?, we've managed—without even trying—to write functions that look only at their arguments to come up with a return value. The goal of writing pure functions also explains the immutability of Clojure's data structures: by disallowing in-place modification of vectors and maps and all the rest, Clojure outlaws a whole class of side effects.

Note that the directive is to *try* to write pure functions. Much of the value that we programmers generate comes out in side effects—we read or write or delete the file, we update the database, or we increment the hit count on a web page. The only thing wrong with side effects is that functions that depend on them aren't the easy-to-assemble building blocks that pure functions are. So we try to write pure functions when we can. Because life is a lot easier without side effects.

• Click HERE to purchase this book now. discuss

Wrapping Up

In this chapter we had our first look at the deeper ideas of functional programming. We saw that in Clojure functions are values—values that you can create, bind to names, and pass around. We also saw that closures allow you to create custom-tailored functions that remember the bindings that existed when they were created. We also looked at some of the helpers that Clojure provides to aid you in the task of creating just the function you need for the task at hand. Finally, we took a quick look at the idea of a pure function: a function that neither relies on nor generates side effects.

Now that you understand what makes Clojure a functional language, it's time to turn to one of the stickiest issues of any programming language: naming things.