Clojure For The Brave and True Part II – Chapter 5 Functional Programming

Agung Tuanany

2025-09-17 Wed

So far, you've focused on becoming familiar with the tools that Clojure provides: immutable data structures, functions, abstractions, and so on. In this chapter, you'll learn how to think about your programming tasks in a way that makes the best use of those tools. You'll begin integrating your experience into a new functional programming mindset.

The core concepts you'll learn include: what pure functions are and why they're useful; how to work with immutable data structures and why they're superior to their mutable cousins; how disentangling[1] data and functions gives you more power and flexibility; and why it's powerful to program to a small set of data abstractions. Once you shove all this knowledge into your brain matter, you'll have an entirely new approach to a problem solving.

After going over these topics, you'll put everything you've learned to use by writing a terminal-based game inspired by an ancient, mystic mind-training device found in Cracker Barrel restaurants across America: Peg Thing!

Pure Functions: What and Why

Except for *println* and *rand*, all the functions you've used up till now have been pure functions. What makes them pure functions, and why does it matter? A function is pure if it meets two qualifications:

- It always return the same result if given the same arguments. This is called *referential transparency*[2], and you can add it to your list of $5 programming terms.

[1]to separate things that have become joined or confused

[2]an expression can always be replaced by its resulting value without changing the program's overall outcome or behavior

- It can't cause any side effects. That is, the function can't make any changes that are observable outside the function itself–for example, by changing an externally accessible mutable object or writing of a file.

These qualities make it easier for you to reason about programs because the functions are completely isolated, unable to impact other parts of your system. When you use them, you don't have to ask yourself, ”What could I break by calling this function?” They're also consistent: you'll never need to figure out why passing a function the same arguments result in different return values, because that will never happens.

Pure functions are a stable and problem free as arithmetic (when was the last time you fretted over adding two numbers?). They're stupendous little bricks of functionality that you can confidently use as the foundation of your program. Let'slook at referential transparency and lack of side effects in more detail to see exactly what they are and how they're helpful.

Pure Functions Are Referentially Transparent

To return the same result when called with the same argument, pure function rely only on [1] their own arguments and [2] immutable values to determine their return value. Mathematical functions, for example, are referentially transparent:

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

If a function relies on a immutable values, it's referentially transparent. The string *”, Daniel-san”* is immutable, so the following function is also referentially transparent:

```
(defn wisdom
  [words]
  (str words, ", Daniel-san"))

(wisdom "Always bathe of Fridays")
;; => Always bathe of Fridays, Daniel-san
```

By contrast, the following functions do not yield the same result with the same arguments, therefore, they are not referentially transparent. Any function that relies on a random number generator cannot be referentially transparent:

```
(defn year-end-evaluation
  []
  (if (> (rand) 0.5)
    "You get a raise!"
    "Better luck next year"))

(defn year-end-evaluation
  []
  (let [random-value (rand)]  ; Generate a random value and store it in a variable
    (if (> random-value 0.5)
      (str "You get a raise! (Random value: " random-value ")")
      (str "Better luck next year (Random value: " random-value ")"))))

(year-end-evaluation)
```

If your function reads from a file, it's not referentially transparent because the file's contents can change. The following function, /analyze-file, is not referentially transparent, but the function analysis is:

```
(defn analyze-file
  [filename]
  (analysis (slurp filename)))

(defn analysis
  [text]
  (str "Character count: " (count text)))
```

When using a referentially transparent function, you never have to consider what possible external conditions could affect the return value of the function. This is especially important if your function calls. In both cases, you can rest easy knowing changes to external conditions won't cause your code to break.

Another way to think about this is that reality is largely referentially transparent. If you think of gravity as a function, then gravitational force is the return value of calling that function on two objects. Therefore, the next time you're in a programming interview, you can demonstrate your functional programming knowledge by knocking everything off your interviewer's desk. (This also demonstrates that you know how to apply a function over a collection.)

Pure Functions Have No Side Effects

To perform a side effect is to change the association between a name and its value within a given scope. Here is an example in JavaScript:

```
var haplessObject = {
    emotion: "Carefree!"
};

var evilMutator = function(object){
    object.emotion = "So emo : '(";
}

evilMutator(haplessObject);
haplessObject.emotion;
// => "So emo : '("
```

Of course, your program has to have some side effects. It writes to a disk, which changes the association between a filename and a collection of disk sector; it changes the RGB values of your monitor's pixels; and so on. Otherwise, there had be no point in running it.

Side effects are potentially harmful, however, because they introduce uncertainty about what the names in your code are referring to. This leads to situations where it's very difficult to trace why and how a name came to be associated with a value, which makes it hard to debug the program. When you call a functions that doesn't have side effects, you only have to consider the relationship between the input and the output. You don't have to worry about other changes that could be rippling through your system.

Function with side effects, on the other hand, place more of a burden on your mind grapes: now you have to worry about how the world is affected when you call the function. Not only that, but every function that depends on a side-effecting function gets infected by this worry; it, too, becomes another component that requires extra care and thought as you build your program.

If you have significant experience with a language like Ruby or JavaScript you've probably run into this problem. As an object gets passed around, its attributes somehow change, and you can't figure out why. Then you have to buy a new computer because you've chucked yours out the window. If you've read anything about object-oriented design, you know that a lot of writing has been devoted to strategies for managing state and reducing side effects for just this reason.

For all these reasons, it's a good idea to look for ways to limit the use of side effects in your code. Lucky for you, Clojure makes your job easier by going to great lengths to limit side effects–all of its core data structures are immutable. You cannot change them in place, no matter how hard you try! However, if you're unfamiliar with immutable data structures, you might feel like your favorite tools has been taken from you. How can you do anything without side effects? Well, that's what the next section is all about! How about that segue, eh? Eh?

Living with Immutable Data Structures

Immutable data structure structures ensure that your code won't have side effects. As you now know with all your heart, this is a good thing. but how do you get anything done without side effects?

Recursion Instead of for/while

Raise your hand if you've ever written something like this in JavaScript:

```
var wreslers = getAlligatorWrestLers();
var totalBites = 0;
var l = wrestlers.length;

for(var i=0; i < l; i++){
    totalBites += wrestlers[i].timeBitten;
}

var allPatients = getArkhamPatients();
var analyzedPatients = [];
var l = allPatients.length;

for(var i=0; i < l; i++){
    if(allPatients[i].analyzed){
        analyzedPatients.push(allPatients[i]);
    }
}
```

Notice that both examples induce side effects on the looping variable *i*, as well as a variable outside the loop (*totalBites* in the first example and *analyzedPatients* in the second). Using side effects this way–mutating *internal* variables–is pretty much harmless. You're creating new values, as opposed to changing an object you've received from elsewhere in your program.

But Clojure's core data structure don't even allow these harmless mutations. So what can you do instead? First, ignore the fact that you could easily use *map* and *reduce* to accomplish the preceding work. In these situations–iterating over some collection to build a result–the functional alternative to mutation is recursion.

Let's look at the first example, building a sum. Clojure has no assignment operator. You can't associative a new value with a name without creating a new scope:

```
(def great-baby-name "Rosanthony")
great-baby-name
;; =>  Rosanthony

(let [great-baby-name "Bloodthunder"]
  great-baby-name)
;; => Bloodthunder

great-baby-name
;; => Rosanthony
```

In this example, you first bind the name *great-baby-name* to *"Rosanthony"* within the global scope. Next, you introduce a new scope with *let*. Within that scope, you bind *great-baby-name* to *"Bloodthunder"*. Once Clojure finishes evaluating the *let* expression, you're back in the global scope, and *great-baby-name* evaluates to *"Rosanthony"* once again.

Clojure lets you work around this apparent limitation with recursion. The following example show the general approach to recursive problem solving:

```
(defn sum
  ([vals] (sum vals 0))  ;; [L-1]
  ([vals accumulating-total]
   (if (empty? vals)  ;; [L-2]
     accumulating-total
     (sum (rest vals) (+ (first vals) accumulating-total)))))
```

This function takes two arguments, a collection to process *(vals)* and an accumulator *(accumulating-total)*, and it uses arity overloading (covered in Chapter 3) to provide a default value of *0* for *accumulating-total* at [L-1].

Like all recursive solutions, this function check the argument it's processing against a base condition. In this case, we checks whether *vals* is empty at [L-2]. If it is, we know that we've processed all the elements in the collection, so we return *accumulating-total*.

If *vals* isn't empty, it means we're still working our way through the sequence, so we recursively call *sum* passing it two arguments: the *tail* of *vals* with *(rest vals)* and the sum of the first element of *vals* plus the accumulating total with *(+ (first vals) accumulating-total)*. In this way, we build up *accumulating-total* and at the same time reduce *vals* until it reaches the base case of an empty collection.

Here's what the recursive function call might look like if we separate out each time it recurs:

```
(defn sum
  ([vals] (sum vals 0))  ;; [L-1]
  ([vals accumulating-total]
   (if (empty? vals)  ;; [L-2]
     accumulating-total
     (sum (rest vals) (+ (first vals) accumulating-total)))))

(sum [39 5 1])  ;; single-arity body calls two-arity body
(sum [39 5 1] 0) ;; '0' is default value
(sum [5 1] 39)
(sum [1] 44)
(sum [] 45) ;; base case is reached, so return accumulating-total
;; => 45
```

Each recursive call to *sum* creates a new scope where *vals* and *accumulating-total* are bound to different values, all without needing alter the values originally passed to the function or perform any internal mutation. As you can see, you can get along fine without mutation.

Note that you should generally use *recur* when doing recursion for performance reasons. The reason is that Clojure doesn't provide tail call optimization. A topic I will never bring up again! (Check out this URL for more information *tail$_{call}$*/.) So here's how you'd do this with *recur*:

```
(defn sum
  ([vals] (sum vals 0))  ;; [L-1]
  ([vals accumulating-total]
   (if (empty? vals)  ;; [L-2]
     accumulating-total
     (recur (rest vals) (+ (first vals) accumulating-total)))))
```

Using *recur* isn't that important if you're recursively operating on a small collection. But if your collection contains thousands or millions values, you will definitely need to whip our *recur* so you don't blow up your program with a stack overflow.

Once last thing! You might be saying, "Wait a minute–what if I end up creating thousands of intermediate values? Doesn't this cause the program to thrash because of garbage collection or whatever?"

Very good question, eagle-eyed-reader! The answer is no! the reason is that, behind the scenes, Clojure's immutable data structures are implemented using *structural sharing*, which is totally beyond the scope of this book. It's kind of like Git! Read this great article if you want to know more understanding-persistent-vector.

Function Composition Instead of Attribute Mutation

Another way you might be used to using mutation is to build up the final state of an object. In the following Ruby example, the *GlamourShotCaption* object uses mutation to clean input by removing trailing spaces and capitalizing "lol":

```
class GlamourShotCaption
  attr_reader :text

  def initialize(text)
    @text = text
    clean!
  end

  private

  def clean!
    text.strip!
    text.gsub!(/lol/, "LOL")
  end
end

best = GlamourShotCaption.new("My boa constrictor is so sassy lol")
```

```
puts best.text

# => My boa constrictor is so sassy LOL
```

In this code, the class *GlamourShotCaption* encapsulated the knowledge of how to clean a glamour shot caption. On creating a *GlamourShotCaption* object, you assign text to an instance variable and progressively mutate it.

Listing 5-1 shows you might do this in Clojure:

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

(defn clean
  [text]

  (s/replace (s/trim text) #"lol" "LOL"))

(clean "My boa constrictor is so sassy lol!")
;; =>"My boa constrictor is so sassy LOL"
```

Figure 1: Listing 5-1: Using function composition to modify glamour shot caption

In the first line, we use *require* to access the string function library (I'll discuss this function and related concept in Chapter 6). Otherwise, the code is easy peasy. No mutation required. Instead of progressively mutating an object, the clean function works by passing an immutable value, text, to a pure function, *s/trim*, which returns an immutable value (*"My boa constrictor is so sassy lol!"; the spaces at the end of the string have been trimmed.) That value is then passed to the pure function /s/replace*, which returns another immutable value (/"My boa constrictor is so sassy LOL!").

Combining functions like this–so that the return value of one function is passed as an argument to another–is called *function composition.* In fact, this isn't so different from the previous example, which used recursion, because recursion continually passes the result of a function to another function; it just happens to be the same function. In general, functional programming encourages you to build more complex functions by combining simpler functions.

This comparison also starts to reveal some limitations of object-oriented programming (OOP). In OOP, one of the main purposes of classes is to protect against unwanted modification of private data–something that isn't necessary with immutable data structures. You also have to tightly couple methods with classes, thus limiting the reuseability of the methods. In the Ruby example, you have to do extra work to reuse the *clean!* method. In Clojure, *clean* will work on any string at all. By both [a] decoupling function and data, [b] programming to a small set of abstractions, you end up with more reusable, composable code. You gain power and lose nothing.

Going beyond immediately practical concerns, the differences between the way you write object-oriented and functional code point to a deeper difference between the two mindset. Programming is about manipulating data for your own nefarious purposes (as much as you can say it's about anything). In OOP, you think about data as something you can embody in an object, and you poke and prod it until it looks right. During this process, your original data is lost forever unless you're very careful about preserving it. By contrast, in functional programming you think of data as unchanging, and you derive new data from existing data. During this process, the original data remains safe and sound. In the preceding Clojure example, the original caption doesn't get modified. It's safe in the same way that numbers are safe when you add them together; you don't somehow transform 4 into 7 when you add 3 to it.

Cool Things to Do with Pure Functions

You can derive new functions from existing functions it the same way that you derive new data from existing data. You've already seen one function, partial, that creates new functions. This section introduces you to two more functions, *comp* and *memoize*, which rely on referential transparency, immutability, or both.

comp

It's always safe to compose pure functions like we just did in the previous section, because you only need to worry about their input/output relationship. Composing functions is so common that Clojure provides a function, *comp*, for creating a new function from the composition of any numbers of functions. Here's simple example:

```
((comp inc *) 2 3)
;; => 7
```

Here, you create an anonymous function by composing the *inc* and *** functions. Then, you immediately apply this function to the arguments *2* and *3*. The function multiplies the numbers 2 and 3 and then increments the result. Using math notation, you'd say that, in general, using *comp* on the function $f_1, f_2, ...f_n$ creates a new function g such that $g(x_1, x_2, ...x_n)$

memoize

Peg Thing

Playing

Code Organization

Creating the Board

Moving Pegs

Rendering and Printing the Board

Player Interaction

Summary

Exercises