Extracted from:

Functional Programming: A PragPub Anthology

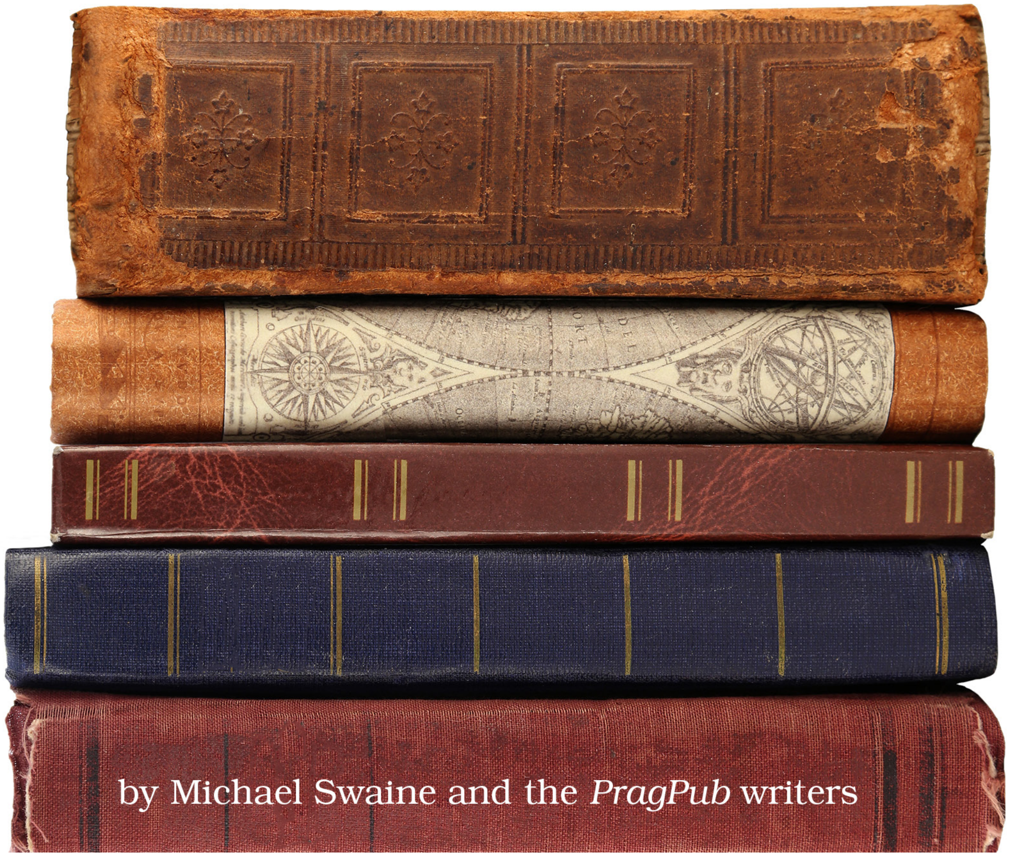
Exploring Clojure, Elixir, Haskell, Scala, and Swift

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Functional Programming: A *PragPub* Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift



by Michael Swaine and the *PragPub* writers

Functional Programming: A PragPub Anthology

Exploring Clojure, Elixir, Haskell, Scala, and Swift

Michael Swaine
and the PragPub writers

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Susannah Davidson Pfalzer
Indexing: Potomac Indexing, LLC
Copy Editor: Nicole Abramowitz
Layout: Gilson Graphics

Printed in the United States of America.
ISBN-13: 978-1-68050-233-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2017

CHAPTER 11

Getting Functional with Elixir

by Dave Thomas

In the preceding chapter, we looked at the basics of pattern matching and saw how it is universal in Elixir—it's the only way to bind a value to a variable or parameter. That may have seemed like an unusual way to start talking about functional programming, but it's very natural for Elixir. And pattern matching really shines when we apply it to functions, which we'll explore in depth in this chapter.

Anonymous Functions

Elixir has a nice set of built-in modules. One of these, Enum, lets you work on enumerable collections. One of its most commonly used functions is map, which applies a function to a collection, producing a new collection. Let's fire up the Elixir interactive shell, iex, and try it.

```
iex> Enum.map [2,4,6], fn val -> val * val end
[4,16,36]
```

The first argument we pass to map is the collection: in this case, a list of three integers. The second argument is an *anonymous function*.

Anonymous functions (I'm going to call them *fn*s from now on) are defined between the keywords fn and end. A right-facing arrow, ->, separates a list of zero or more parameters on the left from the body of the function on the right.

We passed map the function fn val -> val*val end. This fn takes a single parameter, val, and the body of the function multiplies that value by itself, implicitly returning the result.

A fn is just another Elixir value, so we also have written this code as:

```
iex> square = fn val -> val * val end
#Function<erl_eval.6.17052888>
iex> Enum.map [2,4,6], square
[4,16,36]
```

You can call fns using something like a regular function call:

```
iex> square.(5)
25
```

The period and the parentheses are required.

Boy, that's way too much typing, you say.

No it's not, and we don't reward whining around these parts anyway.

That said, Elixir does have a shortcut.

```
iex> Enum.map [2,4,6], &( &1 * &1 )
[4,16,36]
```

When Elixir sees a unary &, it knows that it needs to generate an anonymous function. The function will have from 1 to n parameters, denoted in the expression that follows as &1 to &*n*. So, &(&1*&1) is logically the same as fn p1 -> p1*p1 end, and &rem(&1,&2) becomes fn p1,p2 -> rem(p1,p2) end.

Because fns are just values, you can even write the code as:

```
iex> square = & &1 * &1
#Function<erl_eval.6.17052888>
iex> Enum.map [2,4,6], square
[4,16,36]
```

This is a fantastically useful shortcut, but there is a gotcha. When deciding what code to make the body of the fn, Elixir starts at the ampersand term and looks up the parse tree for the immediately enclosing expression. With &(&1*&1), it's the multiplication operator. With &rem(&1,&2), it's the function call to rem.

Named Functions

Anonymous functions tend to be used for callback work—they are typically short and localized in their use. Named functions, however, are where the real work gets done.

Named functions can only exist inside Elixir modules. Here's an example:

elixir/ascii.exs
```
defmodule AsciiDigit do
  def valid?(character) do
    character in ?0..?9
  end
end

IO.inspect AsciiDigit.valid? ?4    # => true
IO.inspect AsciiDigit.valid? ?a    # => false
```

To follow this code, you first have to know that the syntax ?x returns the integer character code for x (so ?0 is 48).

Our example defines a module called AsciiDigit containing a single function, valid?. This takes a character code and returns true if it is a digit from 0 to 9. We use the range operator .. to define the first and last valid character, and the in operator to check for inclusion.

As we saw in *Patterns and Transformations in Elixir*, Elixir supports pattern matching when determining which function to run. You can use def multiple times for the same function, each with a different pattern of parameters. Elixir will dynamically choose one where the parameters match the arguments passed.

Let's take another look at our Fibonacci function.

elixir/fib.exs
```
defmodule Fibonacci do

  def fib(0), do: 1
  def fib(1), do: 1
  def fib(n), do: fib(n-2)+fib(n-1)

end

Enum.map 0..10, &Fibonacci.fib(&1)   #=> [1,1,2,3,5,8,13,21,34,55,89]
```

Despite appearances, there's just one definition of the fib function in there. It just has three *heads*—three patterns of arguments that select different bodies.

The first two heads select the cases where the argument is 0 or 1. They use the abbreviated form of the body, do: expr to return 1. The third form is the recursive step. If neither of the first two match, the third one executes.

What happens if we pass our function a negative argument? Right now, it will loop until we run out of stack or patience—subtracting 1 or 2 from a negative number will never reach 0. Fortunately, Elixir has *guard clauses*, which allow us to put additional constraints on pattern matching.

elixir/fib1.exs

```
defmodule Fibonacci do

  def fib(0), do: 1
  def fib(1), do: 1
  def fib(n) when n > 1, do: fib(n-2)+fib(n-1)

end

Fibonacci.fib(10)   #=> 89
Fibonacci.fib(-10)
# => ** (FunctionClauseError) no function clause matching in Fibonacci.fib/1
```

Now, when we call fib with a negative number, Elixir can't find a function clause that matches, so it raises an exception. If you really wanted to, you could handle this case in code, giving a more application-specific error:

elixir/fib2.exs

```
defmodule Fibonacci do

  def fib(0), do: 1
  def fib(1), do: 1
  def fib(n) when is_integer(n) and  n > 1, do: fib(n-2)+fib(n-1)
  def fib(x), do: raise "Can't find fib(#{x})"

end

Fibonacci.fib(10)     #=> 89
Fibonacci.fib(-10)    #=> ** (RuntimeError) Can't find fib(-10)
Fibonacci.fib("cat")  #=> ** (RuntimeError) Can't find fib(cat)
```

We extended our guard clause to check that the parameter is an integer, and then added a fourth function head that accepts any parameter and reports an appropriate error.

But understanding how Elixir does functions isn't the same as understanding how to do *functional programming* in Elixir. As we are about to see.

A Practical Example

Most of the long number strings we deal with every day (credit card numbers, IMEI numbers in your phone, and so on) have a check digit. This is normally the final digit of the number, and it is calculated using some algorithm that combines all the previous digits. So, when you enter your credit card number, the web page can recalculate the check digit, and verify that it is the same as the last digit in the number you gave. It isn't a check against fraud; it's simply a quick way of picking up typos.

Probably the most widely used technique is the Luhn Algorithm.[1] It reverses the digits in the number, and splits them into two sets: digits at odd positions in the string, and digits at even positions. It sums the odd digits. For the even digits, it multiplies each by two. If the result is ten or more, it subtracts nine. It then sums all the results. Adding the sum of odd and even positions will yield a result that's divisible by ten for valid numbers.

When I first started with Elixir, my head was still full of conventional ways of doing things. As a result, I'd write something like the following:

elixir/nf1.ex

```
defmodule CheckDigit do

  import Enum

  def valid?(numbers) do
    numbers = reverse(numbers)
    numbers = map(numbers, fn char -> char - ?0 end)
    numbers = with_index(numbers)
    { odds, evens } =
        partition(numbers, fn {_digit, index} -> rem(index, 2) == 0 end)
    sum_odd = reduce odds, 0, fn {number, _index}, sum -> sum + number end
    sum_even = reduce evens, 0, fn {number, _index}, sum ->
      result = number * 2
      if result >= 10 do
        result - 9 + sum
      else
        result + sum
      end
    end
    rem(sum_odd + sum_even, 10) == 0
  end

end
```

Ugh! Let's step through it (hopefully you're wearing boots).

The Enum module has lots of functions for dealing with collections. We'll be using many of them in this code, so we import the module. This means we can write map instead of Enum.map.

Our valid? function is passed a list of UTF-8 digits. By coincidence, that's exactly what the single quoted string literal generates.

Using the description of the Luhn algorithm, we reverse the digits, and then convert the UTF representation to the actual integer value (so ?1, which is 41, gets mapped to 1). At this point, given the argument '123', we'd have a list of integers [3, 2, 1].

1. http://en.wikipedia.org/wiki/Luhn_algorithm

Now it gets messy. We need to partition the digits into those on an even position and those at an odd position. To prepare to do that, we use map, passing it the function fn number, index -> {number, index} end. This function takes the actual digit value, along with its index in the list, and maps it to a tuple containing each.

At this point, alarm bells should be ringing. This is just too damn hard. But we plow on, because that's what programmers do.

The partition function takes a collection and a function. It returns a tuple where the first element is a list of values for which the function returned true, and the second element is the rest of the values.

Now we have to sum the odd values. Whenever you need to reduce a collection to a single value, you'll probably want to use the reduce function. It takes the collection, an initial value, and a function. This function receives each element of the collection in turn, along with the current value. Whatever the function returns becomes the next current value. So, summing a list of numbers can be done with

```
Enum.reduce list, fn val, sum => val + sum end
# or
Enum.reduce list, &( &1 + &2 )
```

But what we have is a list of {value, index} tuples. This means we need to use pattern matching on the first parameter of the function to extract just the value. (The underscore in front of _index means we're ignoring this field.)

Summing the even numbers is similar, but we have to do the doubling, and the conversion of numbers ten or above.

At the end of all this, we can test this in iex. I'm using a standard Visa test credit card number here, so don't go booking a trip to Tahiti using it.

```
$ iex validate_cc.exs
iex> CheckDigit.valid? '4012888888881881'
true
iex> CheckDigit.valid? '0412888888881881'
false
```

Refactor to Functional Style

Our solution works, but the style isn't very functional. (That's a polite way of saying it's butt ugly.) To tidy it up, I look for places where there's clearly something wrong, and see if I can fix them.

The first problem I see is the first three lines. I'm transforming the given number into a reversed set of digits, each with an associated index.

The word *transform* is the clue. Functional programming is all about transforming data. It's so important that Elixir has a special operator, |>. This lets us build pipelines of functions, where each transforms the results of the previous. It lets us compose functionality.

Using the pipeline operator, we can rewrite the first three lines as

```
numbers
  |> reverse
  |> map(fn char -> char - ?0 end)
  |> map(fn digit, index -> {digit, index} end)
```

We take the original list and transform it by reversing it, then by converting character codes to integers, and then by adding the index.

The pipeline operator looks like magic, but it's actually quite simple. It takes the value of the expression on its left, and inserts it as the first argument of the function call on its right, shifting all the other arguments down.

Now the second ugliness is all this partitioning and summing. Our problem is that we're thinking imperatively, not functionally. We're telling Elixir each step of what to do, when instead we should be thinking of the specification of what we want and letting Elixir work out the details.

Think back to our Fibonacci example. There we implemented our specification as three function heads, which matched the two special cases and the one general case. Can we do the same here?

Rather than processing our list of digits one element at a time, what if we process it in twos? This means we're working with a pair of digits—the first will be the one at an odd position, the second at the even position. We know how to calculate the Luhn number for these two digits, and then we can add the result for them into the result for the remainder of the list. That's our recursive step.

When we finally empty the list, we will have calculated the required sum, so we can simply return it.

There's one other case to consider. If the list has an odd number of digits, then when we get to the end, we'll only have a single element. But we know that element is at an odd position, so we can simply add it to the accumulated sum and return the result.

So, here's the new version of our code:

elixir/nfx.ex
```
defmodule CheckDigit do

  import Enum, only: [ reverse: 1, map: 2 ]

  @doc """
  Determine if a sequence of digits is valid, assuming the last digit is
  a Luhn checksum. (http://en.wikipedia.org/wiki/Luhn_algorithm)
  """

  def valid?(numbers) when is_list(numbers) do
    numbers
      |> reverse
      |> map(&(&1 - ?0))
      |> sum
      |> rem(10) == 0
  end

  defp sum(digits), do: _sum(digits, 0)

  defp _sum([], sum), do: sum
  defp _sum([odd], sum), do: sum + odd
  defp _sum([odd, even | tail], sum) when even < 5 do
    _sum(tail, sum + odd + even*2)
  end
  defp _sum([odd, even | tail], sum) do
    _sum(tail, sum + odd + even*2 - 9)
  end

end
```

The pipeline at the top is now a lot simpler—there's no messing with indexes and no temporary variables. It reads like a code version of the spec.

The sum function is an example of a common pattern. We need to set the initial value of the thing we're summing, but we don't want the code that calls us to know about that detail, so we write a version of sum that just takes the numbers and then calls the actual implementation, passing in the list and a zero for the initial value. We could give the helper functions the same name, but I prefer using _sum to differentiate them. (Many Elixir programmers would have called them do_sum, but that always strikes me as too imperative.)

The _sum function has four heads:

- If the list is empty, we return the sum that we've been accumulating in the second parameter.
- If the list has one element, add its value to the sum so far and return it. This is the terminating condition for a list with an odd number of elements.

- Otherwise, we extract the first two elements from the list. This uses the pattern `[odd,even|tail]`. The first element is bound to `odd`, the second to `even`, and the remainder of the list is bound to `tail`.

  Looking back at the Luhn algorithm, we have two cases to consider. If the result of multiplying the even number by two is less than ten, then that's the number we add into the sum. We use a guard class to check for this.

- Otherwise, we have to subtract nine from the product. That's what the fourth function body does.

Notice how we're passing the updated sum around as the second parameter to the function—this is a universal pattern when you want to accumulate a value or values across a set of recursive function calls.

What's Different About This Code

When you write in a language such as Java, C#, or Ruby, you're working at a number of levels simultaneously. Part of your brain is thinking about the specification—what has to get done. The other part is thinking about the implementation—the nuts and bolts of how to do it. And that's where things often get bogged down.

But look at that last example. We're iterating over a set of digits. We're selecting those with odd or even positions. We're performing conditional calculations. We're summing the result. *And there isn't a single control structure in the program.* No ifs, no loops. The code pretty much reflects the specification of what we want to happen.

And that's one of the reasons I'm a fan of functional programming in general, and Elixir in particular.

The value of functional programming, though, shows up in parallel processing, so let's start getting parallel. In the next chapter, we'll see how we can use Elixir to run hundreds of thousands of processes, and how to coordinate their work.