Extracted from:

Learn Functional Programming with Elixir

New Foundations for a New World

The Pragmatic Bookshelf

Raleigh, North Carolina

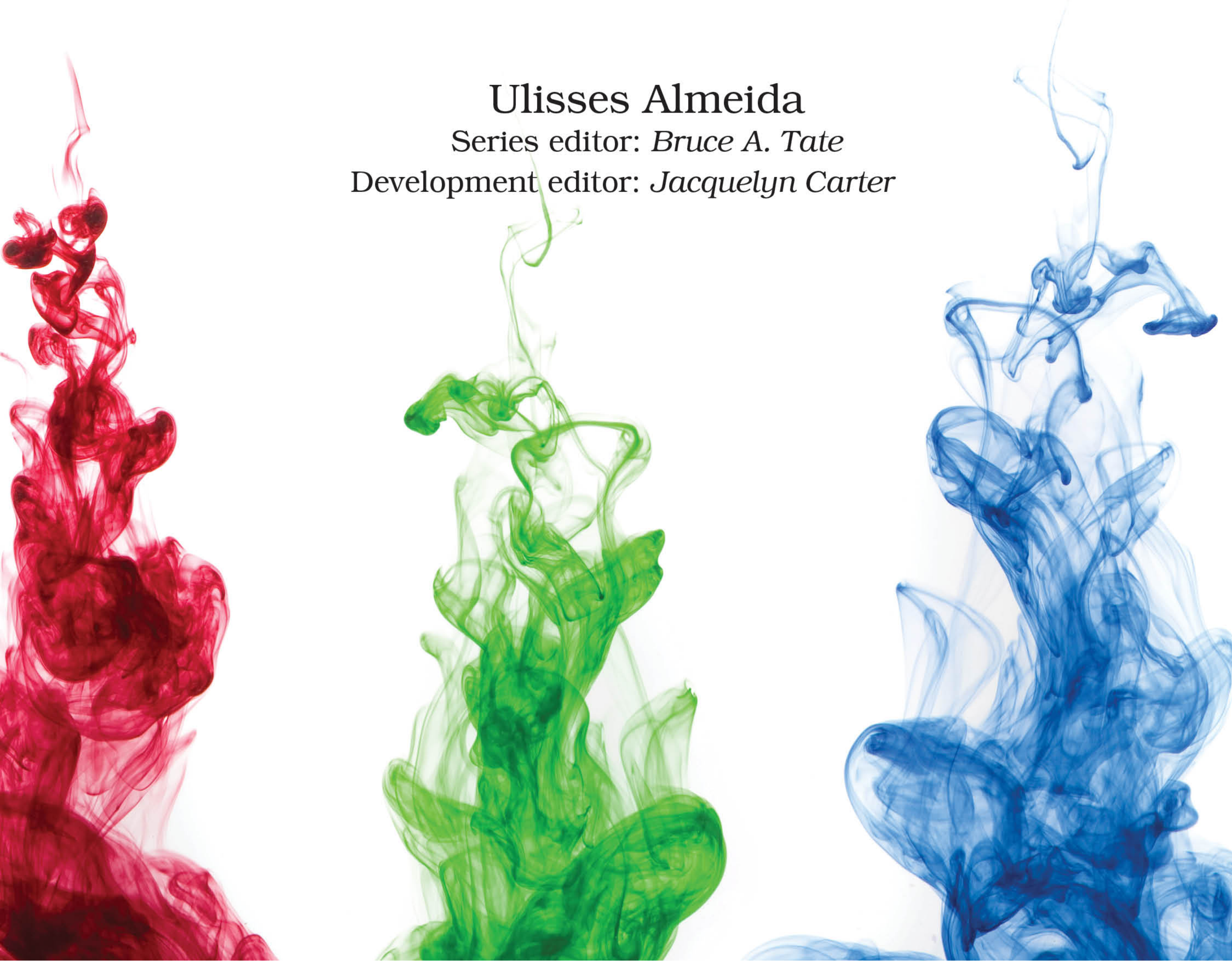
The Pragmatic Programmers

Your Elixir Source

Learn Functional Programming with Elixir

New Foundations for a New World

Ulisses Almeida
Series editor: *Bruce A. Tate*
Development editor: *Jacquelyn Carter*



Learn Functional Programming with Elixir

New Foundations for a New World

Ulisses Almeida

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Series editor: Bruce A. Tate
Copy Editor: Candace Cunningham, Nicole Abramowitz
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*.

Printed in the United States of America.
ISBN-13: 978-1-68050-245-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

Creating Higher-Order Functions for Lists

Using functions in variables, like with any other value, can be hard to remember for newcomers. To practice, we'll work with a subject familiar to us: lists. They're a useful data type and are present in almost any program we need to build. We've seen how to work with them using recursive functions, but if we stop and look again at all that code we've written, we'll see that they are a little bit repetitive and boring. We always have code that navigates through each item, and a stop condition when the list is empty. It's time to change it! We'll look at how to build higher-order functions that hide the tedious tasks and provide an interface for what matters. Let's start with the navigation routine.

Navigating Through Items of a List

A common task when working with lists is to travel through all the items and do some computation on them. The first higher-order function we'll create permits us to navigate a list by passing a function that will compute each item. Our first task is to create a variable that holds a list to test. Let's go back to our old fantasy friend Edwin and store some of his enchanted items in a variable. Open your IEx and type the code that will create the magic recipient:

```
iex> enchanted_items = [
  %{title: "Edwin's Longsword", price: 150},
  %{title: "Healing Potion", price: 60},
  %{title: "Edwin's Rope", price: 30},
  %{title: "Dragon's Spear", price: 100}
]
```

Now people are coming to the store and want to know the items' names. Let's create some code that prints that information. With this routine, Edwin can prepare more magic potions while the program states the items' names for the buyers. To do that, we need to navigate through each list element. In this chapter, we'll create several functions for lists, then create a module called MyList in a my_list.ex file and put all the functions there. The first function will be called each/2. Write the following code:

higher_order_functions/0/my_list.ex

```
defmodule MyList do
  def each([], _function), do: nil
  def each([head | tail], function) do
    function.(head)
    each(tail, function)
  end
end
```

The function receives two arguments: the first is the list that we'll navigate, and the second is a function that will be called, passing each element of the list. The stop-condition clause is called when the list is empty; then it does nothing. The other clause is called when the list has elements; then we use the code function.(head) to call the function received in the argument, passing an element of the list. It runs recursively when the list has multiple elements. Let's try it using our IEx:

```
iex> c("my_list.ex")
iex> MyList.each(enchanted_items, fn item -> IO.puts item.title end)
Edwin's Longsword
Healing Potion
Edwin's Rope
Dragon's Spear
```

We've used MyList.each/2 to navigate through each element of the list. The most interesting part is that when we use that function, we don't need to worry about stop conditions or recursion. All that complexity is hidden. We only need to pass the function that must be executed through each item. It's the same thing to say we have passed an *action* that will happen during the list navigation. We can use this function with different lists and change the result the way we like:

```
items = ["dogs", "cats", "flowers"]
iex> MyList.each(items, fn item -> IO.puts String.capitalize(item) end)
Dogs
Cats
Flowers
iex> MyList.each(items, fn item -> IO.puts String.upcase(item) end)
DOGS
CATS
FLOWERS
iex> MyList.each(items, fn item -> IO.puts String.length(item) end)
4
4
7
```

We've used the same collection and completed different tasks easily. It shows how higher-order functions are powerful for helping us reuse code and hide complexity.

Transforming Lists

Let's practice more. Another common task is generating new lists. We can reduce the complexity of this generation by creating a higher-order function. Let's imagine that the town where Edwin sells his items has increased the sales tax rate; now he needs to increase the price of his items by 10% in order to make the same profit. We need to generate a new list with the new prices. Let's go back to our module MyList and add this new function:

higher_order_functions/my_list.ex
```
def map([], _function), do: []
def map([head | tail], function) do
  [function.(head) | map(tail, function)]
end
```

The MyList.map/2 that we have created receives two arguments. The first is the list that we'll navigate and the second is the function that we're going to pass each item to and use its return to build a new list. The stop-condition clause is when we have an empty list. The other clause uses the list syntax to make a new list. On the new list head, we have the returning value of the given function. That function receives the current list head. On the new list tail, we have a recursive call of the map function. We created a function that generates a new list by applying some computation on each item. The map name is an inheritance of mathematics terminology that means transforming a set to another one. Let's see it in action:

```
iex> c("my_list.ex")
iex> increase_price = fn i -> %{title: i.title, price: i.price * 1.1} end
iex> MyList.map(enchanted_items, increase_price)
[%{price: 165.0, title: "Edwin's Longsword"},
 %{price: 66.0, title: "Healing Potion"},
 %{price: 33.0, title: "Edwin's Rope"},
 %{price: 110.00000000000001, title: "Dragon's Spear"}]
```

You can simplify increase_price by using Elixir's built-in higher-order function Kernel.update_in/2 to update a map. Take a look:

```
iex> increase_price = fn item -> update_in(item.price, &(&1 * 1.1)) end
iex> MyList.map(enchanted_items, increase_price)
[%{price: 165.0, title: "Edwin's Longsword"},
 %{price: 66.0, title: "Healing Potion"},
 %{price: 33.0, title: "Edwin's Rope"},
 %{price: 110.00000000000001, title: "Dragon's Spear"}]
```

The update_in/2 function is useful for updating a map without having to write all the keys to build a new one. We can use our map/2 function to transform any list we want. Try it:

```
items = ["dogs", "cats", "flowers"]
iex> MyList.map(items, &String.capitalize/1)
["Dogs", "Cats", "Flowers"]
iex> MyList.map(items, &String.upcase/1)
["DOGS", "CATS", "FLOWERS"]
iex> MyList.map(["45.50", "32.12", "86.0"], &String.to_float/1)
[45.5, 32.12, 86.0]
```

When we use the map, the task of transforming lists becomes easier. All the work of iterating and building a new list is hidden; we only need to think about the transformation on each item.

Reducing Lists to One Value

The next task is to create a function that transforms a list into one value. For example, it can be useful to discover how much income Edwin can have. To see it, we need to sum all his items' prices. Let's write a higher-order function that will make the job easier:

higher_order_functions/my_list.ex
```
def reduce([], acc, _function), do: acc
def reduce([head | tail], acc, function) do
  reduce(tail, function.(head, acc), function)
end
```

In the first argument, the MyList.reduce/3 function expects a list that will be navigated. The second parameter is an initial value to be accumulated during navigation. The third argument is a function that will be used to apply a computation on the list's item and the value accumulated, generating a new accumulated value. The first function clause is for empty lists. The second clause iterates recursively on each item, updating the accumulated value. Let's sum all of Edwin's items' prices using this function:

```
iex> c("my_list.ex")
iex> sum_price = fn item, sum -> item.price + sum end
iex> MyList.reduce(enchanted_items, 0, sum_price)
340
```

The initial value to accumulate the items' price is 0, then on each iteration reduce uses the sum_price function result to update the accumulated value. The sum_price function takes two parameters: the item of the list and the current accumulated value. We sum both values, and the result is the new accumulated value. We can use the reduce/3 function to work with any generic list we want. Try it:

```
iex> MyList.reduce([10, 5, 5, 10], 0, &+/2)
30
iex> MyList.reduce([5, 4, 3, 2, 1], 1, &*/2)
120
iex> MyList.reduce([100, 20, 400, 200], 100, &max/2)
400
iex> MyList.reduce([100, 20, 400, 200], 100, &min/2)
20
```

Using the reduce/3 function, we can focus only on the operation that accumulates the value. The work of iterating over each item by recursively updating accumulated values is hidden from us.

Filtering Items of a List

The last function we'll build for lists is very common and useful: filtering a list by applying some criteria. Going back to Edwin's shop, let's imagine the customers want to see only the products that cost less than 70 gold coins. We need to filter the shop items by applying the criteria *price less than 70*. When we're filtering, we're creating a new list with only the elements that pass the criteria.

Let's create the following function that will filter the items for us:

higher_order_functions/my_list.ex
```
def filter([], _function), do: []
def filter([head | tail], function) do
  if function.(head) do
    [head | filter(tail, function)]
  else
    filter(tail, function)
  end
end
```

The MyList.filter/2 function calls the given criteria function by passing each list item. If it returns a *falsy* value, it means the item should not be on the new list. Everything that is *truthy*, not nil or false, means it has passed the criteria and should be in the new list. For truthy or falsy cases, the function will keep building the new list and make a recursive call on its tail. Let's see how much easier it is now to filter list items:

```
iex> c("my_list.ex")
iex> MyList.filter(enchanted_items, fn item -> item.price < 70 end)
[%{price: 60, title: "Healing Potion"}, %{price: 30, title: "Edwin's Rope"}]
```

Using our higher-order function filter/2, we just need to pass a function that checks if the item's price is less than 70 gold coins. We can use that function to filter any list. Try it:

```
iex> MyList.filter(["a", "b", "c", "d"], &(&1 > "b"))
["c", "d"]
iex> MyList.filter([100, 200, 300, 400], &(&1 < 300))
[100, 200]
iex> MyList.filter(["Alex", "Mike", "Ana"], &String.starts_with?(&1, "A"))
["Alex", "Ana"]
iex> MyList.filter(["a@b", "t.t", "a@b.c"], &String.contains?(&1, "@"))
["a@b", "a@b.c"]
```

When we use the filter/2 function, it's clear which data and filtering criteria we need to apply. All the complexity of navigating through lists, filtering, building new lists, and recursing is hidden from us.

Using the Enum Module

The each, map, reduce, and filter list operations are useful. Almost all of the programming tasks you'll do with lists can benefit from these functions. Thanks to Elixir's core team, you don't need to write these higher-order functions every time you start a new Elixir project, because they're available in the Enum module. You wrote all these functions to understand how to create higher-order functions. From now on, you'll use them directly from the Enum module. Now we'll experiment with more useful higher-order functions from that module, starting with ones you've built. Open your IEx and try this:

```
iex> Enum.each(["dogs", "cats", "flowers"], &(IO.puts String.upcase(&1)))
DOGS
CATS
FLOWERS
iex> Enum.map(["dogs", "cats", "flowers"], &String.capitalize/1)
["Dogs", "Cats", "Flowers"]
iex> Enum.reduce([10, 5, 5, 10], 0, &+/2)
30
iex> Enum.filter(["a", "b", "c", "d"], &(&1 > "b"))
["c", "d"]
```

The Enum functions work like our homemade functions. The Enum module has many useful functions; it's easy to guess what they do from their names. Let's take a quick look:

```
iex> Enum.count(["dogs", "cats", "flowers"])
3
iex> Enum.uniq(["a", "a", "b", "b", "b", "c"])
["a", "b", "c"]
iex> Enum.sum([10, 5, 5, 10])
30
iex> Enum.sort(["c", "b", "d", "a"], &<=/2)
["a", "b", "c", "d"]
iex> Enum.sort(["c", "b", "d", "a"], &>=/2)
["d", "c", "b", "a"]
iex> Enum.member?([10, 20, 12], 10)
true
iex> Enum.join(["apples", "hot dogs", "flowers"], ", ")
"apples, hot dogs, flowers"
```

The count/1 function returns the total number of elements, and uniq/1 returns a new list without duplicated elements. sum/1 returns the sum of all numbers in a list, member?/2 checks if an item exists in a list, and join/2 combines the list items in one string. The sort/2 is a higher-order function that accepts a function comparing the elements in a list. The Enum functions work with any data type that respects the Enumerable *protocol.*[1] Take a look:

```
iex> upcase = fn {_key, value} -> String.upcase(value) end
iex> Enum.map(%{name: "willy", last_name: "wonka"}, upcase)
["WONKA", "WILLY"]
```

The map is a data type that implements the Enumerable protocol, so you can use it with the Enum module functions. On each iteration of a map structure, we have a tuple with two elements: one for the map key and the other for the value. We'll see more about protocols in Chapter 6, *Designing Your Elixir Applications*, on page ?.

In the Enum module, we also have useful and complex higher-order functions that take two functions in the argument. For example, Enum.group_by/3 receives a function that applies grouping criteria, and it takes a function that generates the values for each group. Let's try it with a list that contains medals and the players who earned them. Create the following medals variable:

```
iex> medals = [
  %{medal: :gold, player: "Anna"},
  %{medal: :silver, player: "Joe"},
  %{medal: :gold, player: "Zoe"},
  %{medal: :bronze, player: "Anna"},
  %{medal: :silver, player: "Anderson"},
  %{medal: :silver, player: "Peter"}
]
```

Now let's show the players that have won each type of medal. To do it, we need to group by medal type (gold, silver, or bronze), and for each group we need to build a list with players' names. Using recursive functions manually isn't easy, but using Enum.group_by/3 can be simple. Try it:

```
iex> Enum.group_by(medals, &(&1.medal), &(&1.player))
%{bronze: ["Anna"], gold: ["Anna", "Zoe"], silver: ["Joe", "Anderson", "Peter"]}
```

We've done a great operation in one line of code. The grouping-criteria function should return a value that will be used to group the items that have identical values. The anonymous function we passed &(&1.medal) returns the value of the medal; that can be :gold, :silver, or :bronze. Then the second function should return a value that goes in the list of each group. Next we use &(&1.player), which returns the player name. With this simple call, we've built a map that contains the players grouped by the medals they've won.

1. https://hexdocs.pm/elixir/Enumerable.html

The flexible and reusable functions of the Enum module are very common in daily tasks, so take time to read about the Enum module and play with its functions. It will help you create simple code since you're taking advantage of the facilities Elixir provides.