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

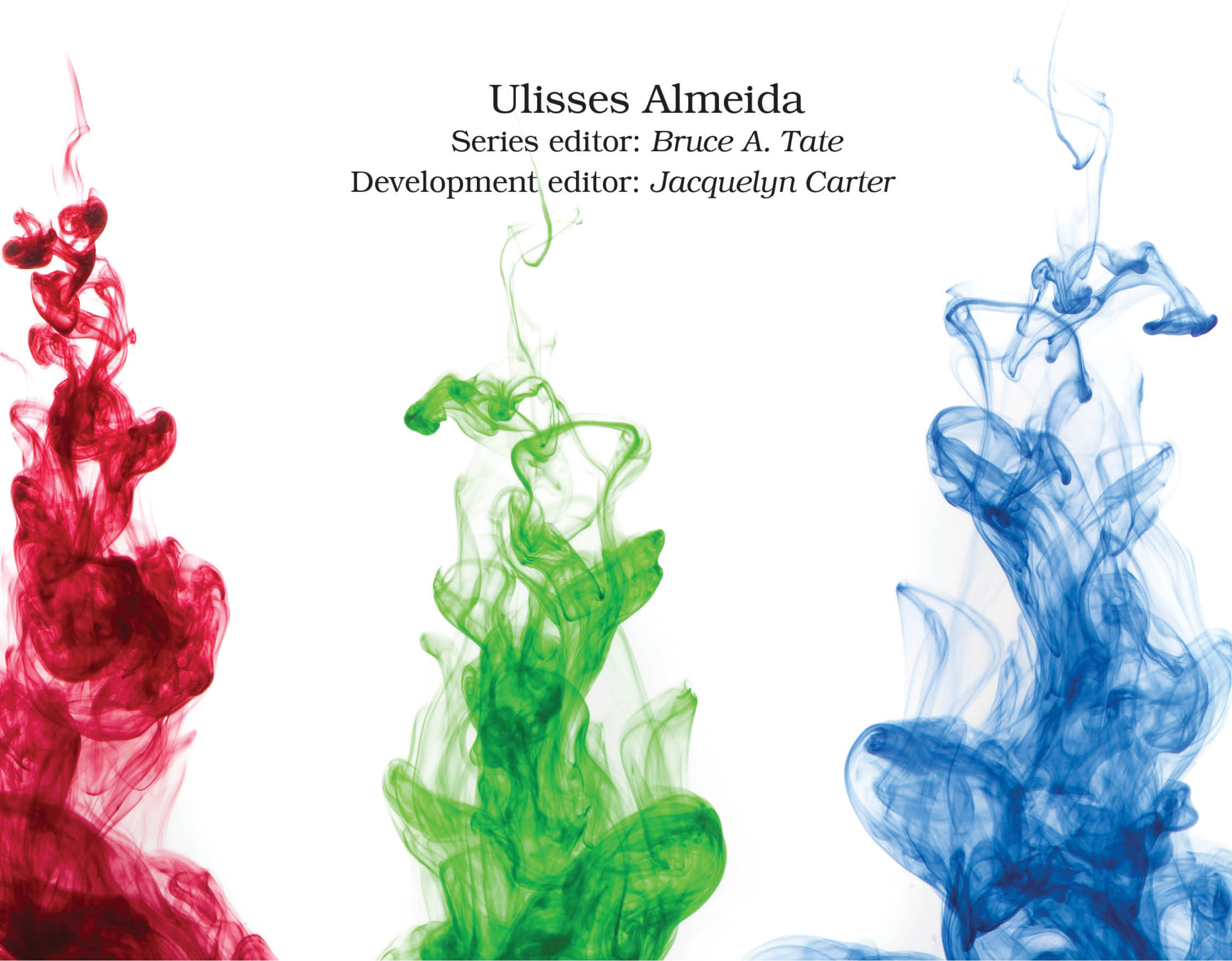
Learn Functional Programming with Elixir

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

Conquering Recursion

For many developers, recursive functions are one of the hardest things to understand when moving to functional programming. Finding the stop-condition clause and making the function call itself can be confusing. Functional programmers work more with recursive functions than developers in other paradigms, because recursive functions are the core of code repetition. If you want to become a functional programmer, it's important that you don't get stuck every time you face a recursive function.

There are two helpful techniques for solving problems using recursive functions: *decrease and conquer* and *divide and conquer*. We'll explore them next.

Decrease and Conquer

Decrease and conquer is a technique for reducing a problem to its simplest form and starting to solve it incrementally. By doing this, we find the most obvious solution to a tiny part of the problem. From there we start to *conquer* progressively, incrementing the problem step by step. Let's experiment with this approach using a well-known problem: the factorial.

The factorial of a number is the product of all positive integers less than or equal to it. If we want to know the factorial of 3, we use the `3 * 2 * 1` expression. Using the decrease-and-conquer strategy, the first step is to find the *base case*, the simplest factorial scenario. We'll use the base case to help solve more complex ones. Let's write it in a module, expecting a number from 0 to 4:

recursion/lib/factorial.ex
```
defmodule Factorial do
  def of(0), do: 1
  def of(1), do: 1
  def of(2), do: 2 * 1
  def of(3), do: 3 * 2 * 1
  def of(4), do: 4 * 3 * 2 * 1
end
```

The factorial for the number `4` is `4 * 3 * 2 * 1`. For the number `3`, it's `3 * 2 * 1`, and so on until we find the solution. The base scenario for the factorial is when the argument is `0` or `1`. We can't keep going because a factorial works only with positive numbers. We don't need calculations for `0` and `1` because the result is `1`. We can compile the code and try what we have done using IEx:

```
iex> c("factorial.ex")
iex> Factorial.of(0)
1
iex> Factorial.of(1)
```

```
1
iex> Factorial.of(4)
24
iex> Factorial.of(5)
** (FunctionClauseError) no function clause matching in Factorial.of/1
iex> Factorial.of(-1)
** (FunctionClauseError) no function clause matching in Factorial.of/1
```

We can't just take the biggest factorial number we want to calculate and write all the functions until we reach it. Let's take a closer look at the factorial of 3. With the expression 3 * 2 * 1, we can write it as 3 * (2 * 1) and it returns the same result. The expression (2 * 1) is the same body as the factorial of 2. Then, instead of writing (2 * 1), we can use a recursive function call. Let's rewrite this function, replacing the calculations with function calls:

recursion/lib/factorial.ex

```
defmodule Factorial do
  def of(0), do: 1
  def of(1), do: 1 * of(0)
  def of(2), do: 2 * of(1)
  def of(3), do: 3 * of(2)
  def of(4), do: 4 * of(3)
end
```

We're almost there. Now the pattern for the solution of the factorial is clear. For a given number, we multiply it with the solution of the factorial of the previous number. That's how we *conquer* the problem. Let's now rewrite these functions to apply the pattern we've discovered:

recursion/lib/factorial.ex

```
defmodule Factorial do
  def of(0), do: 1
  def of(n) when n > 0, do: n * of(n - 1)
end
```

We're done! We've created the solution for the factorial problem using recursion. We've used the guard clause n > 0 to ensure that only numbers greater than our base clause are permitted. That's how to use the decrease-and-conquer approach: First reduce the problem to find its base clause, then look out for the recursive call pattern in the problem we've reduced.

Divide and Conquer

The divide-and-conquer technique is about separating the problem into two or more parts that can be processed independently and can be combined in the end. This technique is not only useful for recursive algorithms; it helps with many other tasks in programming. For example, imagine that we need to build a news homepage that will contain the most recent articles, headlines, and sports and culture sections. If we try to fetch all this content from the database at once, the SELECT query would be hard to write and maintain. The best approach is to *divide* the query into smaller independent operations for each desired piece of content. Then, in the end, we can combine all the results from the database and *conquer* the solution of the news homepage. We can use the same approach with recursive functions. Let's experiment by creating a function that sorts a list.

Sorting functions are useful for displaying ascending or descending content, for creating a better visual experience for users, and for improving search algorithms. We want to build a function that receives a list and returns a list with items in ascending order. In functional programming, the data is immutable; we can't change the order of the values in a list. Instead, we need to build a new list. In the process of generating a new list, we must guarantee that it's sorted. Thinking about a function that does it all at once is hard; instead, we can *divide* the list in half. Now we have two lists to sort, but they're small. If we keep dividing, we'll end up with lists that each contain one element. Lists with one element are sorted! Then we need to merge these lists in a sorted way to finish the algorithm.

That's enough theory. Let's write our sorting function. First we need to learn how to divide a list in half. In Elixir the Enum.split/2 function generates two lists from one, splitting the items. Let's try it in an IEx session:

```
iex> Enum.split([:a, :b, :c], 1)
{[:a], [:b, :c]}
iex> Enum.split([:a, :b, :c], 2)
{[:a, :b], [:c]}
iex> Enum.split([:a, :b, :c], 3)
{[:a, :b, :c], []}
```

The Enum.split/2 returns a tuple with two lists, where the first list contains the number of elements that we specified in the function call. The rest of the list items go in the second list. To split it in half, we need to pass the median number of elements of a list. We'll use the Elixir function Enum.count/1 to calculate the total, and then divide it by two. Try it in your IEx:

```
iex> Enum.count([:a, :b, :c])
3
iex> Enum.count([:a, :b, :c, :d]) / 2
2.0
iex> Enum.count([:a, :b, :c]) / 2
1.5
```

It's a problem when the number of elements in a list is odd, because we can't pass floats to the split function or it will generate an error. We need an integer division here. We can do it using the Elixir Kernel.div/2 function. Try it:

```
iex> div(3, 2)
1
iex> div(4, 2)
2
```

Now we can combine all these functions to divide a list in half recursively. It's the first step in our sorting algorithm; the second step is to build a new list in a sorted way. Let's create the sorting function and put it in a Sort module. Type the following code in your sort.ex file:

recursion/lib/sort.ex

```
defmodule Sort do
  def ascending([]), do: []
  def ascending([a]), do: [a]
  def ascending(list) do
    half_size = div(Enum.count(list), 2)
    {list_a, list_b} = Enum.split(list, half_size)
    # We need to sort list_a and list_b
    # ascending(list_a)
    # ascending(list_b)
    # And merge them using some strategy
  end
end
```

We created an ascending function that only splits the lists, but will soon sort the elements. We created the stop-condition clauses for empty lists and lists containing one element by using pattern matching in the function argument. For lists with more than one item, we created a function clause that will divide it in two. We've used some Elixir built-in functions, such as Enum.split/2 and Enum.count/1, to help us focus on the sorting algorithm and not on list operations.

We divided the lists until they reached one element. Now we need to use these one-item lists to build a new sorted list. We need a merge function that will unify two lists by putting the smallest elements at the beginning of the list. This way, if we try to merge [9] and [5], the result will be [5, 9]. Since the arguments are sorted lists, we know that the first elements are the smallest.

Then we can extract the first item from both lists, compare them, and put the lower value in a new list. If we try to combine [5, 9] with [1, 2], it will be [1, 2, 5, 9]. Doing it recursively, we'll generate a sorted list in the end. Let's see how it works when merging [5, 9] and [1, 4, 5]:

```
merge([5, 9], [1, 4, 5])
[1 | merge([5, 9], [4, 5])]
[1, 4 | merge([5, 9], [5])]
[1, 4, 5 | merge([9], [5])]
[1, 4, 5, 5 | merge([9], [])]
[1, 4, 5, 5, 9]
```

Let's write the merge function that does this work for us:

recursion/lib/sort.ex

```
defp merge([], list_b), do: list_b
defp merge(list_a, []), do: list_a
defp merge([head_a | tail_a], list_b = [head_b | _]) when head_a <= head_b do
  [head_a | merge(tail_a, list_b)]
end
defp merge(list_a = [head_a | _], [head_b | tail_b]) when head_a > head_b do
  [head_b | merge(list_a, tail_b)]
end
```

The first two clauses are straightforward. If we try to merge an empty list with any other list, the result will be that other list. The clause head_a <= head_b means the first element of list_a is the smallest. Then we extract the first element of list_a and put it in the first spot in the new list using the expression [head_a | merge(tail_a, list_b)]. For the rest of the elements of the new list, we call merge recursively, passing the rest of the elements of the list a and passing the entire list_b. The clause head_a > head_b does the inverse operation, extracting and putting the first element of list_b in the new list's first spot.

Using the merge/2 function, we can now combine all the lists we've divided and build a new one. Let's add this call in our ascending function:

recursion/lib/sort.ex

```
def ascending([]), do: []
def ascending([a]), do: [a]
def ascending(list) do
  half_size = div(Enum.count(list), 2)
  {list_a, list_b} = Enum.split(list, half_size)
➤ merge(
➤   ascending(list_a),
➤   ascending(list_b)
➤ )
end
```

Before we pass the lists to the merge/2 function, we must ensure that lists are sorted. That's why we do a recursive call to the ascending function before the merge. It will recursively merge the divided lists like this:

```
merge(merge([9], [5]), merge(merge([1], [5]), [4]))
merge([5, 9], merge([1, 5], [4]))
merge([5, 9], [1, 4, 5])
[1, 4, 5, 5, 9]
```

In this sorting function, the recursive call for ascending can work independently; all the recursive calls are not connected with each other. For example, we can compute them in parallel, although in the end we need to join both results to present a sorted list using the merge function. We can try our Sort module using IEx:

```
iex> c("sort.ex")
iex> Sort.ascending([9, 5, 1, 5, 4])
[1, 4, 5, 5, 9]
iex> Sort.ascending([2, 2, 3, 1])
[1, 2, 2, 3]
iex> Sort.ascending(["c", "d", "a", "c"])
["a", "c", "c", "d"]
```

We did it! The sorting algorithm is working. This algorithm is known as the *merge sort*.[1] It's one of the most famous divide-and-conquer algorithms.

Divide and conquer, as you may have noticed, is very similar to decrease and conquer. The main difference is that while the *decrease* strategy is focused on reducing the problem until we find a base clause, the *divide* technique is about separating the problem into two or more parts. These parts can be processed independently and be combined in the end. As you can see, recursion does a lot of function calls and it may cause performance bottlenecks. In the next section you'll learn how create recursive functions that use your machine resources prudently.

1. https://en.wikipedia.org/wiki/Merge_sort