Extracted from:

Learn Functional Programming with Elixir

New Foundations for a New World

The Pragmatic Bookshelf

Raleigh, North Carolina

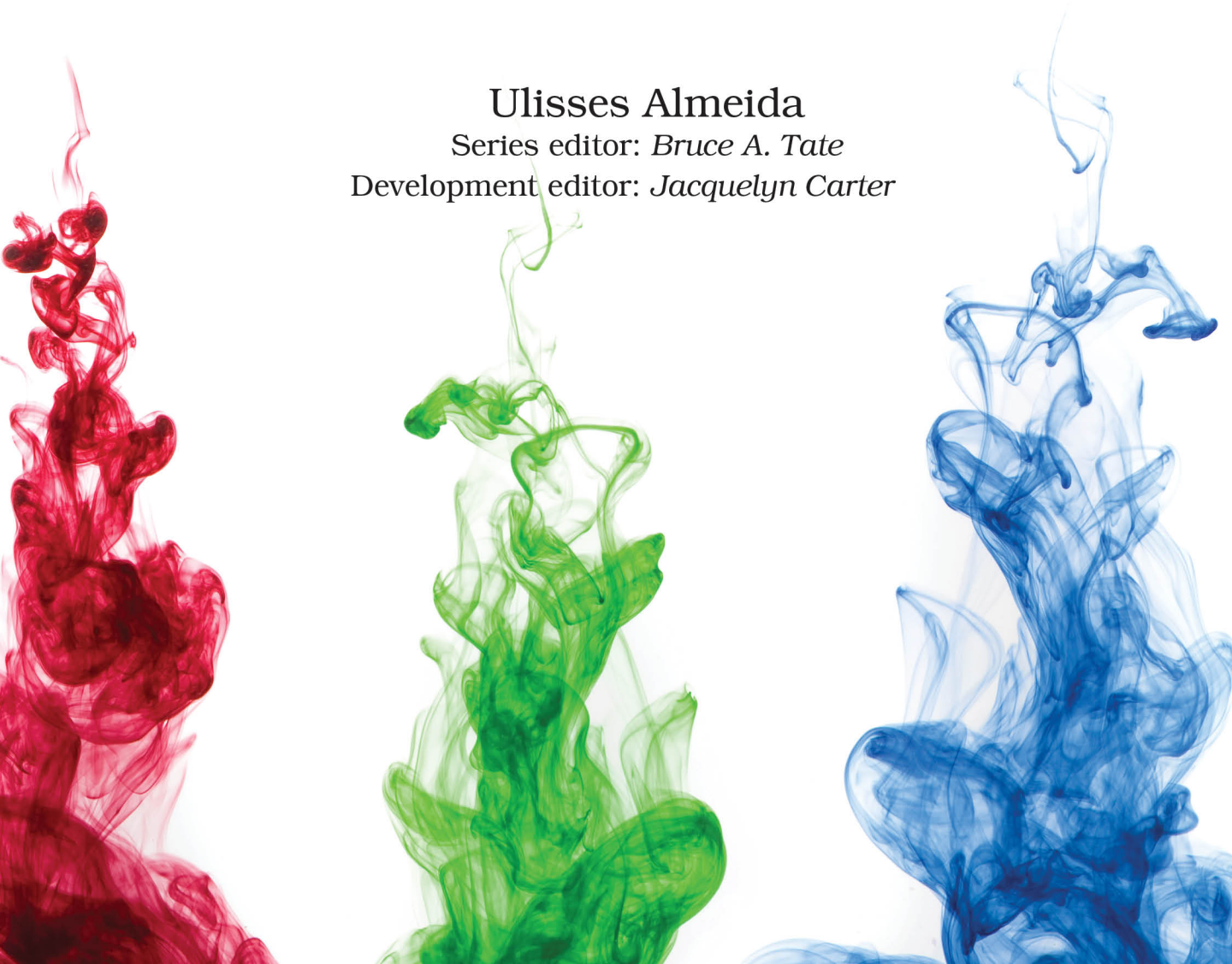
The Pragmatic Programmers

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*



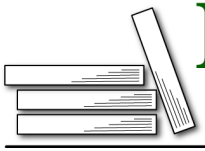
Learn Functional Programming with Elixir

New Foundations for a New World

Ulisses Almeida

The Pragmatic Bookshelf

Raleigh, North Carolina



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
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 Anonymous Functions

You can think of functions as subprograms of your program. They receive an input, do some computation, and then return an output. The function body is where we write expressions to do a computation. The last expression value in the function body is the function's output. Functions are useful for reusing expressions. Let's start with a simple example in which we'll build messages to say hello to Ana, John, and the world. Try typing this in your IEx:

```
iex> "Hello, Mary!"
"Hello, Mary!"
iex> "Hello, John!"
"Hello, John!"
iex> "Hello, World!"
"Hello, World!"
```

If we want to say hello to Alice and Mike, we could copy and paste the message and replace the names. But instead we can create a function to make it easier to say hello to anything we want. First, we need to identify the things that change in the messages. In the preceding example, we can see that the only thing that changes is the name of the person or group we want to say hello to. We can write an expression that separates the name from the message. Try it:

```
iex> name = "Alice"
iex> "Hello, " <> name <> "!"
"Hello, Alice!"
```

We created the name variable that represents something that can change. Then we used the <> operator to join the strings with the name variable. To transform these expressions into a function, we transform the name variable in a parameter and the string concatenation in a function body. Let's take a look at the function-creation syntax. Try it in your IEx:

```
iex> hello = fn name -> "Hello, " <> name <> "!" end
iex> hello.("Ana")
"Hello, Ana!"
iex> hello.("John")
"Hello, John!"
iex> hello.("World")
"Hello, World!"
```

We created a function and bound it to a variable called hello. Then we invoked that using the *dot* operator and passing values inside the parentheses. We can invoke that function with different values in the argument. These types of functions are called *anonymous functions* in Elixir because they have no global name and must be bound to a variable to be reused. They are useful for creating functions on the fly. (They are also known as *lambdas* and are the only type of function in lambda calculus.)

Now let's go step by step through how we have defined the function:

1. The fn indicates the beginning of the function.
2. The name is the function's parameter. A function's parameters are internal function variables that force whoever is invoking the function to supply them with values. When calling a function we need to pass the values in the same order the parameters were defined.
3. We have the -> operator, which indicates the following expression will be the body of a function clause.
4. The function body is the expression "Hello, " <> name <> "!". The return value is the value of the last expression. In this example, there's only one expression, so the value of that expression will be returned.
5. The end marks the end of the function definition.

Elixir gives developers the power of redefining some of the language's basic functions and blocks by using metaprogramming. However, the fn and end combination is an Elixir special form. Special forms are basic building blocks that cannot be overridden by the developer. They'll always work in the same

way no matter the framework or library that you're using in your application. You can see more details about special forms in Elixir's documentation.[4]

You can replace the <> operator with Elixir's expressive string-interpolation syntax:

```
iex> hello = fn name -> "Hello, #{name}!" end
iex> hello.("Ana")
"Hello, Ana!"
```

All the expressions inside of the brackets in the #{} code will be evaluated and coerced to a string. Here's an example:

```
iex> "1 + 1 = #{1+1}"
"1 + 1 = 2"
```

We commonly use anonymous functions for simple operations, and most of them will be on one line. But we can create them with multiple lines; just break the line after the -> operator:

```
iex> greet = fn name ->
...>   greetings = "Hello, #{name}"
...>   "#{greetings}! Enjoy your stay."
...> end
#Function<6.99386804/1 in :erl_eval.expr/5>
```

We can also create functions without arguments. We just need to omit them:

```
iex> one_plus_one = fn -> 1 + 1 end
iex> one_plus_one.()
2
```

We can create functions with multiple arguments, too, by separating them with commas:

```
iex> total_price = fn price, quantity -> price * quantity end
iex> total_price.(5, 6)
30
```

We've used commas to separate the parameters price and quantity. Elixir has a limit of 255 parameters in a function. That's enough for any application. However, it's good maintenance practice to keep the number of parameters below five. A higher number of parameters can be a good indication that you need a data structure—tuples, lists, structs, or maps—or you need to split your function into smaller ones.

4. https://hexdocs.pm/elixir/Kernel.SpecialForms.html

Functions as First-Class Citizens

The first time I read the term *first-class citizens*, I found it funny because I imagined a bunch of functions flying first class to Europe. But it means the opposite. When we say in programming that functions are first-class citizens, we mean that they are like any other value. It's an important feature that came from lambda calculus.

In Elixir, functions are values of type function. Let's build a function that expects a function:

```
iex> total_price = fn price, fee -> price + fee.(price) end
```

The function total_price receives two arguments; one is a number that will represent the price. The fee parameter expects a function. We'll call the given function, passing the price. The final result of the function is the result of the price plus the result of the fee function. Now, let's build some fee functions:

```
iex> flat_fee = fn price -> 5 end
iex> proportional_fee = fn price -> price * 0.12 end
```

Now we can try these functions all together:

```
iex> total_price.(1000, flat_fee)
1005
iex> total_price.(1000, proportional_fee)
1120.0
```

We first call the total_price function, passing the flat_fee, and then we call total_price another time, passing the proportional_fee function. In this example, we have passed a function in an argument like any other value. Functions are the actions in the program. Passing or returning actions in functions is what makes functional programming so different from imperative programming. We'll explore it more in Chapter 5, *Using Higher-Order Functions*, on page ?.

Sharing Values Without Using Arguments

We can share values with functions using *closures*. A closure has access to variable values both inside and outside of the code block. In Elixir we can create an anonymous function and pass it a code block with the values of the variables that were defined outside of it. It's useful to be able to share values with functions when you can't control the functions' invocation, since you can't pass values to functions' parameters. You can't control function calls specially when you use functions that take other functions as arguments. For example, we can use Elixir's spawn to start a process and execute a function asynchronously. The spawn will invoke the given function asynchronously,

and we can't pass arguments to it. One way to share values with that function is by taking advantage of the closure:

```
iex> message = "Hello, World!"
iex> say_hello = fn -> Process.sleep(1000); IO.puts(message) end
iex> spawn(say_hello)
"Hello, World!"
```

The function say_hello remembered the value of the message variable and printed the message on the console using IO.puts after one second using Process.sleep. We used the printing and sleeping commands on the same line using the semicolon. (The commands are named functions, and we'll see these types of functions in detail in the next section.) We have shared values with say_hello without using arguments. This is possible because closures remember all the free variables that were referenced in the lexical scope in which they were created. Free variables? Lexical scope? Let's see what these terms mean.

Hey, We Have a Side Effect Here

In this section, we used a say_hello function. It calls IO.puts, displaying a message in our console session. The console and our program are different entities. When a function interacts with anything that is external, it's vulnerable to external problems. We say that function has side effects; it's impure. We'll discuss pure and impure functions in detail in Chapter 7, *Handling Impure Functions*, on page ?.

A scope is a part of a program—a code block, for example. The lexical scope is related to the visibility of the variables in the code where they were defined. When you use a variable in a function definition, the compiler will analyze your code reading upwards and will bind the variable to the closest definition. Everything defined before and outside of a function's scope is the upward scope. Try this example:

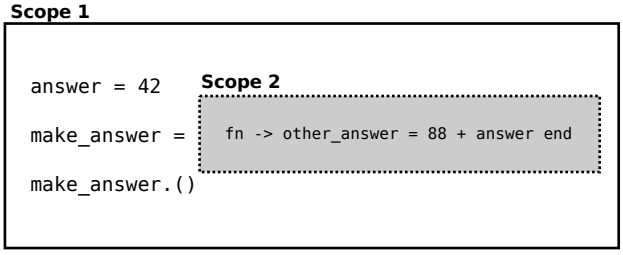
```
iex> answer = 42
iex> make_answer = fn -> other_answer = 88 + answer end
iex> make_answer.()
130
iex> other_answer
** (CompileError) iex:4: undefined function other_answer/0
iex> answer = 0
iex> make_answer.()
130
```

The function make_answer references the variable answer; the compiler will go to the upward scope and find the answer definition. When we try to call other_answer outside of the function's scope, the program will generate an error. That's

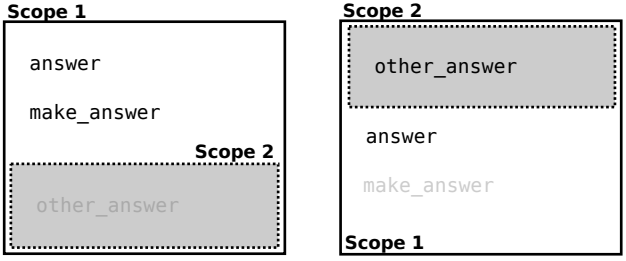
because other_answer exists only inside of the make_answer function's scope, not outside of it. It's like a one-way mirror: the inner scope can *see* the variables outside, but not vice versa.

Also note the unaffected make_answer result after we assign a new value to answer. When we define a function referencing a variable outside of the function's scope, we bind the current value and it will be immutable. That's why when answer has a new value, it doesn't affect the make_answer function's result.

The following diagram illustrates how scopes work. The white box is the scope of the IEx shell, while the gray box is the scope of the anonymous function make_answer.



We can see each code block has his own space. The next diagram shows that each code block we create has a space that the code outside can't see into. But the code inside the space can see the variables defined outside and reference them. The gray shading color of the variable indicates that variable is not visible by the scope. The following diagram shows each scope's variable visibility:



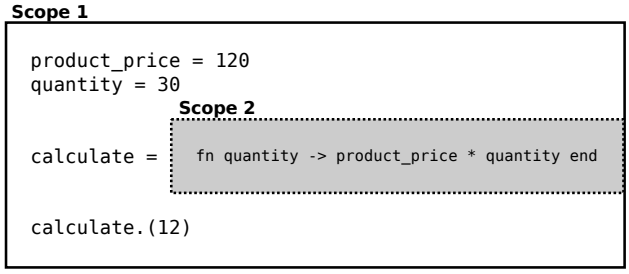
The outer scope can't see the variables defined inside of the anonymous function. The anonymous function can only see the variables defined before its own definition. That's why the anonymous function can't see the make_answer variable: it was defined after the function-creation expression.

With an understanding of how lexical scope works, we can now discuss free and bound variables. Inside of a function, a variable is bound when it is

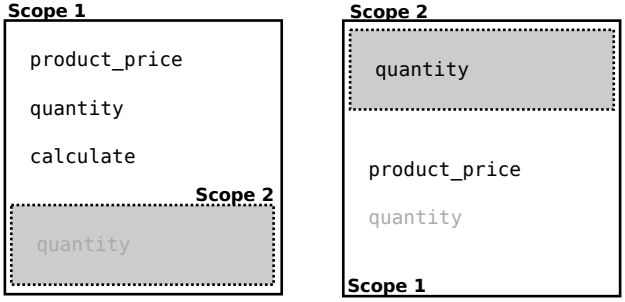
defined as a function's parameter or a local variable in a function's body; otherwise, it's free. Let's test the closure:

```
iex> product_price = 200
iex> quantity = 2
iex> calculate = fn quantity -> product_price * quantity end
iex> calculate.(4)
800
```

We've defined the variable quantity, but the function calculate has a parameter with the same name. This means the variable is bound, and its value will not be remembered. product_price is free, but it doesn't exist in the calculate parameter although it's referenced in the body. Therefore, the product_price value will be remembered no matter where the execution happens. The following diagram illustrates the scopes' definitions:



We can see the variables' visibility on each scope:



We can clearly see now that the quantity parameter defined in the inner scope has higher precedence than the variable with the same name defined in the outer scope. The outer variable quantity is shadowed by the quantity parameter in the calculate function. Variable shadowing isn't good practice because it creates confusion about the variable's value, generating code that is hard to understand. Avoid this! That's how closures work in Elixir: we can share values with functions without using arguments.