Extracted from:

Seven More Languages in Seven Weeks

Languages That Are Shaping the Future

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Seven More Languages in Seven Weeks

Languages That Are Shaping the Future

Bruce A. Tate, Fred Daoud, Ian Dees, and Jack Moffitt

Foreword by José Valim

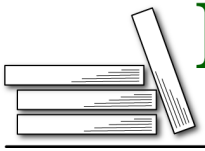
Edited by Jacquelyn Carter

Seven More Languages in Seven Weeks

Languages That Are Shaping the Future

Bruce A. Tate
Fred Daoud
Ian Dees
Jack Moffitt

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (indexer)
Liz Welch (copyeditor)
Dave Thomas (typesetter)
Janet Furlow (producer)
Ellie Callahan (support)

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-941222-15-7
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—November 2014

Day 1: Laying a Great Foundation

Our speed tour will focus on the three biggest influences on Elixir: Ruby, Lisp, and Erlang. Day 1 will show you where Ruby's influence begins and ends. I'll walk you through the basic building blocks of the language, while taking an informal look at operators, simple types, and expressions. Then, we'll look at functions and modules. Finally, we will work with collections of things and craft together some simple programs with recursion. That's a lot to handle, but to get to know this rich language, we'll have to move fast.

Day 2 will bring forth the strong Lisp influences on the abstract syntax tree (AST), the foundation for Elixir's macro system. We'll focus most of our attention on building a macro to represent a state machine in code.

We'll finish our tour by looking into Erlang influences in Day 3. The third day will a little shorter, because in Days 1 and 2 we have to lay a lot of language foundation to handle the rich macro material. We'll use our state machine in a concurrent, distributed application.

Rarely will you get the opportunity to explore so closely the influences of one language on another. It's going to be a long first day, so let's get started.

Installing Elixir

Elixir is a language based on Erlang (*Programming Erlang: Software for a Concurrent World [Arm07]*), which we covered in the first *Seven Languages* book (*Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages [Tat10]*). You'll need to install Erlang.[1] I'm using 17.1, and you'll need version 17.0 or later.

Next, you'll install the language and environment. Find them on the language's Getting Started page.[2] I used Homebrew, version 0.14, but everything should work on Elixir version 1.0. Syntax is changing quickly, so if you decide to use a later version, you'll need to pay attention to changes in syntax.

Once you've installed it all, fire up Interactive Elixir (iex) like this:

```
> iex
Erlang/OTP 17 [erts-6.1] [source] [64-bit] [smp:8:8]
[async-threads:10] [hipe] [kernel-poll:false]

Interactive Elixir (1.0) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

1. http://www.erlang.org/download.html
2. http://elixir-lang.org/getting_started/1.html

Solt's Ruby++, Right?

Since José Valim, creator of Elixir, was a member of the Ruby on Rails core team, many people viewed his new language with Ruby-colored glasses. Syntactically, you can see more than a coincidental similarity. Try to see what reminds you of Ruby:

```
iex> IO.puts "It's B-29s, bub."
It's B-29s, bub.
iex> 4
4
iex> 4 != 5
true
iex> 4 > 5 and 6 > 7
false
iex(2)> Enum.at [], 0
nil
iex> :atom
:atom
```

Like Ruby and many modern languages, Elixir programs are made up of simple data types, operators, and functions that roll up into expressions. The special values nil, true, and false all mean what you think, and are named just as they are on the Ruby side. Elixir also copies Ruby's syntax for symbols instead of Erlang's atoms.

```
iex> if 5 > 4, do: IO.puts "You wanted the truth!"
You wanted the truth!
:ok
iex> if nil, do: IO.puts "You wanted the truth!"
nil
:ok
```

:ok is a typical Elixir return code. Like Ruby, Elixir has do/end syntax for simple control structures. Like Ruby, Elixir also has one-line syntax for if expressions. Like Ruby, Elixir has so-called "truthy" expressions. nil and false are false; everything else is true. Strings have some familiar sugar, too:

```
iex> "Two plus two is #{2 + 2}"
"Two plus two is 4"
```

Elixir's string interpolation drops a string representation of an expression into the string you specify. There are other similarities to Ruby on the string side. They can contain escape sequences for unprintable characters such as newlines and tabs; Elixir allows for multiline representations called heredocs, and you'll find C-style sigils, a syntax for formatting literals.

No, Not Ruby

Although the syntax might be familiar to Ruby developers, under the hood, things are remarkably different. Elixir is a functional language. The base types are not objects, and the base types are immutable. You can't change a list or a tuple after you've defined it the first time.

It's best to think of Elixir as a language *whose syntax is influenced by Ruby*. The similarities end there. Take the = operator, for example:

```
iex> i = 5
5
iex> 10 = i
** (MatchError) no match of right hand side value: 5
```

It may look like an assignment here, but it's not. If you learned Erlang in *Seven Languages*, you recognize the = operator as a pattern match. Said another way, the interpreter asked the question "Do the values on the left side match the values on the right?" If necessary, the interpreter assigns unbound variables on the left to match values on the right. Let's push pattern matching a little further.

Tuples are collections of fixed size. You can have a two-tuple representing a city and state, like this:

```
iex> austin = {:austin, :tx}
{:austin, :tx}
iex> is_tuple {:a}
true
```

austin is a variable, and we assign a tuple with two atoms, :austin and :tx. Elixir makes the left side match the right by assigning {:austin, :tx} to the variable austin. In this case, we matched the whole tuple. We can also use matching to access both elements of the tuple individually, or using wildcards, we can access either element in isolation. This concept, called *destructuring*, is critical.

```
iex> austin = {:austin, :tx}
{:austin, :tx}
iex> {city, :tx} = austin
{:austin, :tx}
iex> city
:austin
iex> {city, :ok} = austin
** (MatchError) no match of right hand side value: {:austin, :tx}
iex> {_, big_state} = austin
{:austin, :tx}
iex> big_state
:tx
```

Nice. In this way, you'll use Elixir to trivially pack and unpack complex data structures just as you did this one.

So what was all of that noise about surly and opinionated?

In functional languages like Erlang, multiple assignment just won't work. You can assign a given variable a value exactly once. That practice means that these languages are immune from many of the problems related to mutable state or multiple assignment. To handle this language limitation, you'll see developers use different variable values on the left-hand side for each assignment, and keeping track of those changing values can be tedious and error prone as code evolves, like this:

```
...
Price = Catalog.lookup(Item)
Price2 = Price * Quantity
Price3 = Price2 + Price2 * Tax
...
```

Elixir's approach looks a little more like the imperative style of Ruby or Java:

```
...
price = Catalog.lookup(item)
price = price * quantity
price = price + price * tax
...
```

Some card-carrying Erlang developer now knows exactly what I mean by opinionated. In fact, his thoughts could be sliding into black rage because *functional programming should not allow reassignment*. We can only hope that he doesn't have adamantium blades for fingernails and the ability to respawn.

That code looks suspiciously like mutable state, but really, it's not. The compiler is playing a game here. The compiler marks each new `price` as `price'` internally, and for each subsequent access. In fact, the compiler is doing implicitly exactly what the original Erlang program does by hand. The result is that internally, there's no mutable state at all.

This language feature expresses an opinion. Does this trick actually make code more expressive, or does it take you down the slippery slope toward mutable state and obscure what's actually happening? Decide for yourself.

The primary Ruby influence, though, isn't mutability, or 'true's and 'nil's. It's *intelligent sugar*. You express powerful idioms in a way that communicates to both you and the compiler. The debate is how far syntactic sugar should go.

Elixir is about as much like Ruby as Java is like JavaScript. From here on, put Ruby out of your head completely, and enjoy the new path Elixir is cutting.

Writing Functions

So far, we've seen how some basic types and expressions work, and that the language relies heavily on pattern matching to accomplish basic tasks. It's time to add the basic building block of all functional languages, the function. Elixir has plenty of different options for declaring and consuming functions. We're going to start simple, with unnamed or *anonymous functions* and then ramp up to named functions in modules. We can assign a function to a variable like this:

```
iex> inc = fn(x) -> x + 1 end
#Function<6.80484245 in :erl_eval.expr/5>
iex> inc.(1)
2
```

When you invoke an anonymous function, you need a . character before your arguments. This double_call is a higher order function:

```
iex> double_call = fn(x, f) -> f.(f.(x)) end
#Function<12.80484245 in :erl_eval.expr/5>
iex> double_call.(2, inc)
4
```

As expected, we called inc.(inc.(2)) and got 4. As you might imagine, you'll be working with functions more than any other language construct. Here's a shorthand way for declaring a function to add two numbers:

```
iex> add = &(&1 + &2)
&Kernel.+/2
iex> add.(1, 2)
3
```

Beautiful. We just used &1 and &2 as placeholders for our arguments. Now that we have an add, we can use it to declare other functions that build on it.

```
iex> inc = &(add.(&1, 1))
 #Function<6.80484245 in :erl_eval.expr/5>
iex> inc.(1)
2
iex> dec = &(add.(&1, -1))
 #Function<6.80484245 in :erl_eval.expr/5>
iex> dec.(1)
0
```

inc and dec are examples of partially applied functions. As you learned in Elm, these functions take existing functions and apply only a subset of arguments to them. For example, inc is a partially applied function, applying the second argument and leaving the first unapplied.

Composing with Pipes

Functional programming is about building functions that work together. One of the most important compositions is running functions in sequence, matching up inputs and outputs. Let's express two steps forward and one step back with inc and dec:

```
iex> x = 10
10
iex> dec.(inc.(inc.(x)))
11
```

We started with x = 10. A step forward is an inc and a step back is dec. If you start from the inside and work your way out, you can see that we are actually doing inc, inc, and dec. But the intention is not clear. Let's remedy that.

```
iex> 10 |> inc.() |> inc.() |> dec.()
11
```

That's much clearer. These pipes work just like they do in Factor or Elm. Elixir evaluates the pipe from left to right, passing the expression on the left-hand side of the pipe as the first argument of the function on the right. If you had two named functions, inc and dec, you could strip away even more syntax with 10 |> inc |> inc |> dec. The pipe operator translates the obtuse inside-out representation to a simple and direct statement of what our program accomplishes. Clojure developers, think ->.

You'll find that the pipe operator is perhaps the most important operator in the language in the same way that Unix shell languages rely on the | operator. It allows you to express ideas in the same way that you're used to consuming information: from left to right, with inputs on the left contributing to the process on the right. You can make complex problems simpler by expressing them as a pipe of simpler functions.

As we continue to work with bigger and bigger building blocks in the language, we go from the function to the *module*. In the next section, we'll organize named functions into modules.

Using Modules

Elixir programmers group functions, macros, and other constructs into modules. Learning Elixir is easier if you think of a module definition as plain, old executable code rather than a series of function definitions. Take a look:

```
iex> defmodule Silly do
...>   IO.puts "Pointless existence"
...> end
Pointless existence
{:module, Silly, ..., :ok}
```

Notice that the compiler ran the module, and printed the expression Pointless existence. You can see that as Elixir is loading the module, it will just execute each line, in sequence. Most of the time, those lines will define other functions or modules.

For now, think of modules as specialized functions that generate code at compile time. defmodule is a macro that defines a module, and def is a macro that defines functions. Keep this knowledge in the back of your mind as we walk through these basic examples.

Named Functions

Let's create some modules to do elementary geometry. We'll start with some simple functions to compute the area and perimeter of a rectangle. You might start with a couple of functions that each take parameters h and w, like this:

```
defmodule Rectangle do
  def area(w, h), do: h * w
  def perimeter(w, h), do: 2 * (w + h)
end
```

We call the defmodule function, which defines a module. We provide a block that calls the def function twice, creating two functions within the module.

area and perimeter are functions you'd expect to apply to a rectangle, so we can improve on the API. Instead of passing in individual dimensions, let's pass in a tuple with two dimensions that represents our rectangle, like this:

elixir/geometry.exs

```
defmodule Rectangle do
  def area({h, w}), do: h * w

  def perimeter({h, w}) do
    2 * (h + w)
  end
end
```

That's much better. Now, the API is Rectangle.area(shape), where shape is a two-tuple that represents a rectangle with width and height. The API clearly expresses our intentions. We use pattern matching to pick off each dimension, and then use those dimensions in a calculation.