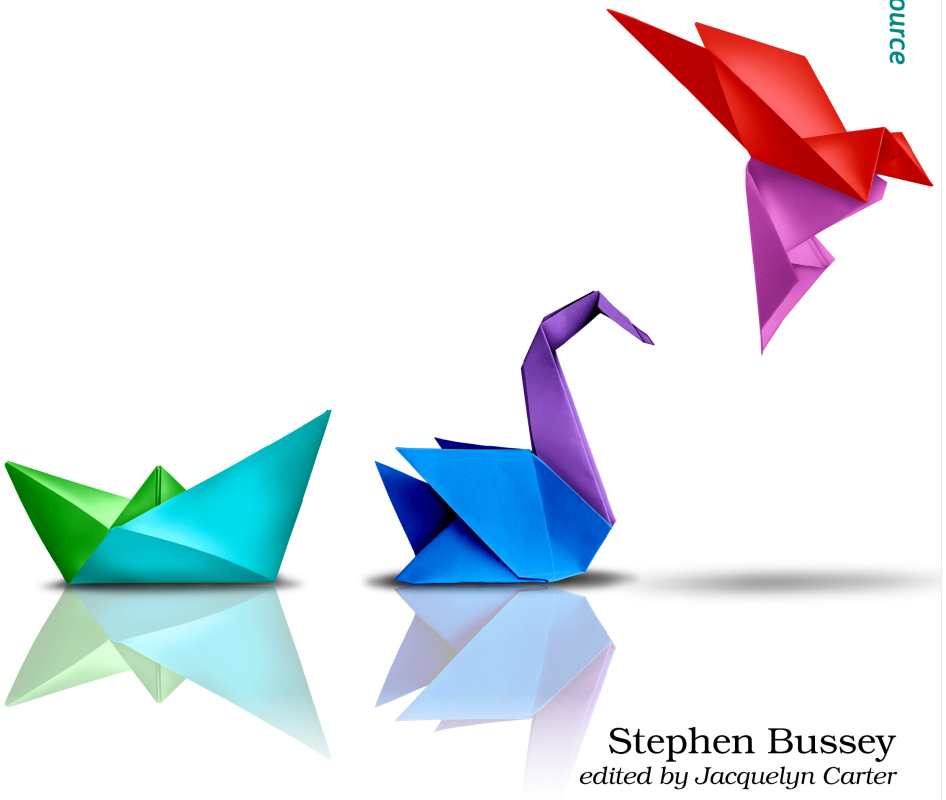
The Pragmatic Programmers

Your Elixir Source

From Ruby to Elixir

Unleash the Full Potential of Functional Programming



Stephen Bussey

edited by Jacquelyn Carter

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Pattern matching will change the way that you write code. It's a simple—yet extremely powerful—feature that's built into the foundation of Elixir. Pattern matching is used in function definitions, variable assignments, and control flows—it's a core port of the language's design. And once you master it, you won't want to go back.

In Ruby (and most languages), the core structures for control flow are if statements. This is easy to use if you want to check whether a value is one of two candidate values—true or false. It becomes cumbersome if you want to check whether a value is one of many possible values—based on string contents, array values, map keys, and so on. Elixir has if statements, but it also has something more powerful.

In this chapter, you'll see how Elixir's case statement completely replaces if and switch statements. Elixirists use case statements a lot, so it's good to get comfortable with it. Luckily, it's also simple. You just need to know how pattern matching works. The syntax of pattern matching is simple, but its roots run deep and it takes a little bit of practice to get used to. This chapter builds up slowly so that you have everything you need to be confident with patterns.

We'll start by looking at the most basic forms of pattern matching. Then, you'll see how case statements are used in control flow. Finally, we'll combine everything to see how pattern matching affects function definitions and makes recursive functions much easier to write.

Pattern matching is a game changer, so let's dive in!

Pattern Matching Basics

Elixir doesn't have a normal assignment operator. In most languages, the = operator is used for simple `left = "value"` statements. In Elixir, this operator is called the match operator, and it initiates pattern matching.

Pattern matching is implemented by the BEAM, so it's baked into the runtime of the language. Some optimizations make it efficient even for a large number of pattern clauses. So, you can use pattern matching without worrying about a negative performance impact on your application.

In this section, we'll go over different pattern-matching syntaxes for basic data types, lists, maps, tuples, and more.

Match Basic Types

Let's start with the most basic syntax for pattern matching. Open a new IEx session and type the following:

```
$ iex
iex> 1 = 1
1

iex> a_number = 1
1

iex> 1 = a_number
1
```

This first example is seemingly simple, but `1 = 1` is rather unusual. In Ruby, you can only have variables on the left side of `=`. Clearly, that's not the case here.

To evaluate a pattern mentally, execute the right side and then compare the result with the left side. Assign any variables that are on the left side. If the patterns don't match, then you get a `MatchError`:

```
iex> 1 = 2
** (MatchError) no match of right hand side value: 2
```

Variable assignment works just as it does in Ruby. Values are reassigned when they're on the left side:

```
iex> a_number = 1
iex> a_number = 2
iex> a_number
2
```

It's important to understand that this doesn't transform the data of the variable. It simply reassigns the variable to a different value.

One thing that trips up many new Elixir programmers is that you cannot have function calls inside of the pattern.

```
iex> 1 + 1 = 2
CompileError: cannot invoke remote function :erlang.+/2 inside a match
```

```
iex>
defmodule Local do
  def call do
    test() = 1
  end

  defp test do
    1
  end
end
```

```
** (CompileError): cannot find or invoke local test/0 inside match.
   Only macros can be invoked in a match and they must be defined
   before their invocation. Called as: test()
```

This is a useful error message. It tells us about our coding error and also lets us know that some functions (macros) can be invoked in a match clause.

It's not common to write macro-based match functions yourself, but you'll frequently use ones provided by Elixir. Besides lists and maps—which we will cover next—string concatenation is commonly used. Here's an example that uses string concatenation (<>) in a match clause:

```
iex> "store:" <> data_command = "store:Widget:process"
"store:Widget:process"
```

```
iex> data_command
"Widget:process"
```

The <> appears on the left side of the = symbol, and a variable is used where a string part would be. Elixir pattern-matches the string and extracts the relevant text into the data_command variable.

This is a powerful way to split apart text without calling String.split/2. But it's not without its limitations. The variable must always be the last part of the concatenation. You can do this:

```
iex> "text" <> ":" <> number = "text:7"
iex> number
"7"
```

But you can't do this:

```
iex> "text" <> symbol <> number = "text:7"
** (ArgumentError) the left argument of <> operator inside a match
   should always be a literal binary because its size can't be
   verified. Got: symbol
```

Even with this limitation, it's still extremely useful.

Let's explore other powerful pattern-matching forms. We'll look at lists, maps, and tuples next.

Match Data Structures

Lists, tuples, and maps are fully compatible with pattern matching. You'll commonly use this in two ways. The first is to extract data structure components into variables so you can operate on them. The second is to check if an input matches a certain structure, as part of control flow.

This section focuses on extracting the components of data structures. Open a new IEx session and type the following:

```
$ iex
iex> [a] = [1]
iex> a
1

iex> {:ok, result} = {:ok, "my result"}
iex> result
"my result"

iex> [a, 2, c] = [1, 2, 3]
iex> {a, c}
{1, 3}

iex> [a] = [1, 2]
** (MatchError) no match of right hand side value: [1, 2]
```

Lists and tuples can be matched on an exact-position basis. Each position on the right and left must have compatible patterns. You can even separate a data structure into multiple variables, like `a` and `c` in the previous code. If your structure doesn't match the pattern provided, you get a `MatchError`.

Parts of a list are matched with the `|` and `++` operators:

```
iex> [head | tail] = [1, 2, 3, 4]
iex> head
1

iex> tail
[2, 3, 4]
```

```
iex> [first, second | rest] = [1, 2, 3, 4]
iex> {first, second}
{1, 2}

iex> [first, second] ++ rest = [1, 2, 3, 4]
iex> {first, second}
{1, 2}
```

The | operator is used inside of the list brackets to represent the beginning of the list. One or more elements can be matched at a time.

The ++ operator is used to capture the concatenation of two lists. It's less common to see this syntax, though.

One thing you'll notice from these match clauses is that they behave exactly like their function versions. This makes the syntax intuitive to use. If you can use <> or [|] in your code, then you can use it in a pattern match clause. For example:

```
iex> [1, 2] ++ [3, 4]
[1, 2, 3, 4]

iex> [1, 2] ++ rest = [1, 2, 3]
iex> rest
[3]
```

In this example, we're able to use the ++ operator as a function (left ++ right) and as a match.

Pattern matching with maps is also intuitive, as in the following example:

```
iex> %{a: a} = %{a: 1, b: 2}
iex> a
1

iex> %{a: 1, b: nil} = %{a: 1, b: 2}
** (MatchError) no match of right hand side value: %{a: 1, b: 2}

iex> %{list: [%{a: ["a"]}, %{b: [b]}]} = %{list: [%{a: ["a"]}, %{b: ["b"]}]}
iex> b
"b"
```

This example is a bit dense, but it shows you that the complexity of the match clause isn't limited as long as it uses valid syntax.

Map matching behaves differently than lists because maps don't have to perfectly match. In the first example, the left side doesn't mention the `b` key at all. Maps are loosely matched when the key isn't specified. This is useful in practice because you often extract a few keys of a map. Here's a simple example to demonstrate this:

```
iex> [] = [1]
** (MatchError) no match of right hand side value: [1]

iex> %{} = %{a: 1}
%{a: 1}
```

The left side of the list match is empty, and it doesn't match the right side list. The left side of the map match is empty, but it still matches the right side map. Maps are loosely matched, but lists are strictly matched.

Let's see how we can reference existing values in a pattern match clause.

Pinned Values

You are not limited to only assigning variables in a pattern match. The pin operator lets you use the value of an existing variable inside of your pattern match. This is most useful in test suites, where you want to guarantee that different values match inside of a data structure.

Prepend the variable with the ^ symbol to pin its value. Let's see this in action:

```
iex> var = :match
iex> ^var = :match
:match

iex> ^var = :no_match
** (MatchError) no match of right hand side value: :no_match

iex> [^var, second] = [:match, :other]
iex> second
:other
```

Pinned values are strictly enforced, so the result must perfectly match or you'll receive an error. This is intuitive for most data types, but be careful with maps:

```
iex> map = %{a: 1}
iex> ^map = %{a: 1}
%{a: 1}

iex> ^map = %{a: 1, b: 2}
** (MatchError) no match of right hand side value: %{a: 1, b: 2}
```

The map didn't exactly equal the pinned value, so a MatchError was thrown.

You might be wondering what happens if you use a variable twice on the left side of a match. This isn't considered a pinned value, but it behaves similarly.

```
iex> {x, x} = {1, 1}
iex> x
1

iex> {x, x} = {1, 2}
** (MatchError) no match of right hand side value: {1, 2}
```

The duplicated variable must be equal in all positions of the match clause. Otherwise, you'll get a MatchError.

Now that you have the basics of pattern matching, let's see how it can be used to control the flow of a program.