Table of Contents

Meta-Programming in Elixir

This file serves as your book's preface, a great place to describe your book's content and ideas.

Quote and unquote

An Elixir program can be represented by its own data structures. In this chapter, we will learn what those structures look like and how to compose them. The concepts learned in this chapter are the building blocks for macros, which we are going to take a deeper look at in the next chapter.

Quoting

The building block of an Elixir program is a tuple with three elements. For example, the function call `sum(1, 2, 3)` is represented internally as:

```
{:sum, [], [1, 2, 3]}
```

You can get the representation of any expression by using the `quote` macro:

```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

The first element is the function name, the second is a keyword list containing metadata and the third is the arguments list.

Operators are also represented as such tuples:

```
iex> quote do: 1 + 2
{:+, [context: Elixir, import: Kernel], [1, 2]}
```

Even a map is represented as a call to `%{}` :

```
iex> quote do: %{1 => 2}
{:%{}, [], [{1, 2}]}
```

Variables are also represented using such triplets, except the last element is an atom, instead of a list:

```
iex> quote do: x
{:x, [], Elixir}
```

When quoting more complex expressions, we can see that the code is represented in such tuples, which are often nested inside each other in a structure resembling a tree. Many languages would call such representations an Abstract Syntax Tree (AST). Elixir calls them quoted expressions:

```
iex> quote do: sum(1, 2 + 3, 4)
{:sum, [], [1, {:+, [context: Elixir, import: Kernel], [2, 3]}, 4]}
```

Sometimes when working with quoted expressions, it may be useful to get the textual code representation back. This can be done with `Macro.to_string/1`:

```
iex> Macro.to_string(quote do: sum(1, 2 + 3, 4))
"sum(1, 2 + 3, 4)"
```

In general, the tuples above are structured according to the following format:

```
{atom | tuple, list, list | atom}
```

- The first element is an atom or another tuple in the same representation;
- The second element is a keyword list containing metadata, like numbers and contexts;
- The third element is either a list of arguments for the function call or an atom. When this element is an atom, it means the tuple represents a variable.

Besides the tuple defined above, there are five Elixir literals that, when quoted, return themselves (and not a tuple). They are:

```
:sum         #=> Atoms
1.0          #=> Numbers
[1, 2]       #=> Lists
"strings"    #=> Strings
{key, value} #=> Tuples with two elements
```

Most Elixir code has a straight-forward translation to its underlying quoted expression. We recommend you try out different code samples and see what the results are. For example, what does `String.upcase(&quot;foo&quot;)` expand to? We have also learned that `if(true, do: :this, else: :that)` is the same as `if true do :this else :that end`. How does this affirmation hold with quoted expressions?

Unquoting

Quote is about retrieving the inner representation of some particular chunk of code. However, sometimes it may be necessary to inject some other particular chunk of code inside the representation we want to retrieve.

For example, imagine you have a variable `number` which contains the number you want to inject inside a quoted expression.

```
iex> number = 13
iex> Macro.to_string(quote do: 11 + number)
"11 + number"
```

That's not what we wanted, since the value of the `number` variable has not been injected and `number` has been quoted in the expression. In order to inject the *value* of the `number` variable, `unquote` has to be used inside the quoted representation:

```
iex> number = 13
iex> Macro.to_string(quote do: 11 + unquote(number))
"11 + 13"
```

`unquote` can even be used to inject function names:

```
iex> fun = :hello
iex> Macro.to_string(quote do: unquote(fun)(:world))
"hello(:world)"
```

In some cases, it may be necessary to inject many values inside a list. For example, imagine you have a list containing `[1, 2, 6]` and we want to inject `[3, 4, 5]` into it. Using `unquote` won't yield the desired result:

```
iex> inner = [3, 4, 5]
iex> Macro.to_string(quote do: [1, 2, unquote(inner), 6])
"[1, 2, [3, 4, 5], 6]"
```

That's when `unquote_splicing` becomes handy:

```
iex> inner = [3, 4, 5]
iex> Macro.to_string(quote do: [1, 2, unquote_splicing(inner), 6])
"[1, 2, 3, 4, 5, 6]"
```

Unquoting is very useful when working with macros. When writing macros, developers are able to receive code chunks and inject them inside other code chunks, which can be used to transform code or write code that generates code during compilation.

Escaping

As we saw at the beginning of this chapter, only some values are valid quoted expressions in Elixir. For example, a map is not a valid quoted expression. Neither is a tuple with four elements. However, such values *can* be expressed as a quoted expression:

```
iex> quote do: %{1 => 2}
{:%{}, [], [{1, 2}]}
```

In some cases, you may need to inject such *values* into *quoted expressions*. To do that, we need to first escape those values into quoted expressions with the help of `Macro.escape/1`:

```
iex> map = %{hello: :world}
iex> Macro.escape(map)
{:%{}, [], [hello: :world]}
```

Macros receive quoted expressions and must return quoted expressions. However, sometimes during the execution of a macro, you may need to work with values and making a distinction between values and quoted expressions will be required.

In other words, it is important to make a distinction between a regular Elixir value (like a list, a map, a process, a reference, etc) and a quoted expression. Some values, such as integers, atoms and strings, have a quoted expression equal to the value itself. Other values, like maps, need to be explicitly converted. Finally, values like functions and references cannot be converted to a quoted expression at all.

You can read more about `quote` and `unquote` in the `Kernel.SpecialForms` module. Documentation for `Macro.escape/1` and other functions related to quoted expressions can be found in the `Macro` module.

In this introduction we have laid the groundwork to finally write our first macro, so let's move to the next chapter.

Macros

Foreword

Even though Elixir attempts its best to provide a safe environment for macros, the major responsibility of writing clean code with macros falls on developers. Macros are harder to write than ordinary Elixir functions and it's considered to be bad style to use them when they're not necessary. So write macros responsibly.

Elixir already provides mechanisms to write your every day code in a simple and readable fashion by using its data structures and functions. Macros should only be used as a last resort. Remember that **explicit is better than implicit**. **Clear code is better than concise code.**

Our first macro

Macros in Elixir are defined via `defmacro/2`.

> For this chapter, we will be using files instead of running code samples in IEx. That's because the code samples will span multiple lines of code and typing them all in IEx can be counter-productive. You should be able to run the code samples by saving them into a `macros.exs` file and running it with `elixir macros.exs` or `iex macros.exs`.

In order to better understand how macros work, let's create a new module where we are going to implement `unless`, which does the opposite of `if`, as a macro and as a function:

```
defmodule Unless do
  def fun_unless(clause, expression) do
    if(!clause, do: expression)
  end

  defmacro macro_unless(clause, expression) do
    quote do
      if(!unquote(clause), do: unquote(expression))
    end
  end
end
```

The function receives the arguments and passes them to `if`. However, as we learned in the previous chapter, the macro will receive quoted expressions, inject them into the quote, and finally return another quoted expression.

Let's start `iex` with the module above:

```
$ iex macros.exs
```

And play with those definitions:

```
iex> require Unless
iex> Unless.macro_unless true, IO.puts "this should never be printed"
nil
iex> Unless.fun_unless true, IO.puts "this should never be printed"
"this should never be printed"
nil
```

Note that in our macro implementation, the sentence was not printed, although it was printed in our function implementation. That's because the arguments to a function call are evaluated before calling the function. However, macros do not evaluate their arguments. Instead, they receive the arguments as quoted expressions which are then transformed into other quoted expressions. In this case, we have rewritten our `unless` macro to become an `if` behind the scenes.

In other words, when invoked as:

```
Unless.macro_unless true, IO.puts "this should never be printed"
```

Our `macro_unless` macro received the following:

```
macro_unless(true, {{:., [], [{:aliases, [], [:IO]}, :puts]}, [], ["this should never
be printed"]})
```

And it then returned a quoted expression as follows:

```
{:if, [], [
  {:!, [], [true]},
  {{:., [], [IO, :puts], [], ["this should never be printed"]}}]}
```

We can actually verify that this is the case by using `Macro.expand_once/2`:

```
iex> expr = quote do: Unless.macro_unless(true, IO.puts "this should never be printed"
)
iex> res  = Macro.expand_once(expr, __ENV__)
iex> IO.puts Macro.to_string(res)
if(!true) do
  IO.puts("this should never be printed")
end
:ok
```

`Macro.expand_once/2` receives a quoted expression and expands it according to the current environment. In this case, it expanded/invoked the `Unless.macro_unless/2` macro and returned its result. We then proceeded to convert the returned quoted expression to a string and print it (we will talk about `__ENV__` later in this chapter).

That's what macros are all about. They are about receiving quoted expressions and transforming them into something else. In fact, `unless/2` in Elixir is implemented as a macro:

```
defmacro unless(clause, options) do
  quote do
    if(!unquote(clause), do: unquote(options))
  end
end
```

Constructs such as `unless/2`, `defmacro/2`, `def/2`, `defprotocol/2`, and many others used throughout this getting started guide are implemented in pure Elixir, often as a macros. This means that the constructs being used to build the language can be used by developers to extend the language to the domains they are working on.

We can define any function and macro we want, including ones that override the built-in definitions provided by Elixir. The only exceptions are Elixir special forms which are not implemented in Elixir and therefore cannot be overridden, the full list of special forms is available in `Kernel.SpecialForms`.

Macros hygiene

Elixir macros have late resolution. This guarantees that a variable defined inside a quote won't conflict with a variable defined in the context where that macro is expanded. For example:

```
defmodule Hygiene do
  defmacro no_interference do
    quote do: a = 1
  end
end

defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.no_interference
    a
  end
end

HygieneTest.go
# => 13
```

In the example above, even though the macro injects `a = 1`, it does not affect the variable `a` defined by the `go` function. If a macro wants to explicitly affect the context, it can use `var!`:

```
defmodule Hygiene do
  defmacro interference do
    quote do: var!(a) = 1
  end
end

defmodule HygieneTest do
  def go do
    require Hygiene
    a = 13
    Hygiene.interference
    a
  end
end

HygieneTest.go
# => 1
```

Variable hygiene only works because Elixir annotates variables with their context. For example, a variable `x` defined on line 3 of a module would be represented as:

```
{:x, [line: 3], nil}
```

However, a quoted variable is represented as:

```
defmodule Sample do
  def quoted do
    quote do: x
  end
end

Sample.quoted #=> {:x, [line: 3], Sample}
```

Notice that the third element in the quoted variable is the atom `Sample`, instead of `nil`, which marks the variable as coming from the `Sample` module. Therefore, Elixir considers these two variables as coming from different contexts and handles them accordingly.

Elixir provides similar mechanisms for imports and aliases too. This guarantees that a macro will behave as specified by its source module rather than conflicting with the target module where the macro is expanded. Hygiene can be bypassed under specific situations by using macros like `var!/2` and `alias!/2`, although one must be careful when using those as they directly change the user environment.

Sometimes variable names might be dynamically created. In such cases, `Macro.var/2` can be used to define new variables:

```
defmodule Sample do
  defmacro initialize_to_char_count(variables) do
    Enum.map variables, fn(name) ->
      var = Macro.var(name, nil)
      length = name |> Atom.to_string |> String.length
      quote do
        unquote(var) = unquote(length)
      end
    end
  end

  def run do
    initialize_to_char_count [:red, :green, :yellow]
    [red, green, yellow]
  end
end

> Sample.run #=> [3, 5, 6]
```

Take note of the second argument to `Macro.var/2`. This is the context being used and will determine hygiene as described in the next section.

The environment

When calling `Macro.expand_once/2` earlier in this chapter, we used the special form `__ENV__`.

`__ENV__` returns an instance of the `Macro.Env` struct which contains useful information about the compilation environment, including the current module, file and line, all variables defined in the current scope, as well as imports, requires and so on:

```
iex> __ENV__.module
nil
iex> __ENV__.file
"iex"
iex> __ENV__.requires
[IEx.Helpers, Kernel, Kernel.Typespec]
iex> require Integer
nil
iex> __ENV__.requires
[IEx.Helpers, Integer, Kernel, Kernel.Typespec]
```

Many of the functions in the `Macro` module expect an environment. You can read more about these functions in the docs for the `Macro` module and learn more about the compilation environment in the docs for `Macro.Env`.

Private macros

Elixir also supports private macros via `defmacrop`. As private functions, these macros are only available inside the module that defines them, and only at compilation time.

It is important that a macro is defined before its usage. Failing to define a macro before its invocation will raise an error at runtime, since the macro won't be expanded and will be translated to a function call:

```
iex> defmodule Sample do
...>  def four, do: two + two
...>  defmacrop two, do: 2
...> end
** (CompileError) iex:2: function two/0 undefined
```

Write macros responsibly

Macros are a powerful construct and Elixir provides many mechanisms to ensure they are used responsibly.

- Macros are hygienic: by default, variables defined inside a macro are not going to affect the user code. Furthermore, function calls and aliases available in the macro context are not going to leak into the user context.
- Macros are lexical: it is impossible to inject code or macros globally. In order to use a macro, you need to explicitly `require` or `import` the module that defines the macro.
- Macros are explicit: it is impossible to run a macro without explicitly invoking it. For example, some languages allow developers to completely rewrite functions behind the scenes, often via parse transforms or via some reflection mechanisms. In Elixir, a macro must be explicitly invoked in the caller during compilation time.
- Macros' language is clear: many languages provide syntax shortcuts for `quote` and `unquote`. In Elixir, we preferred to have them explicitly spelled out, in order to clearly delimit the boundaries of a macro definition and its quoted expressions.

Even with such guarantees, the developer plays a big role when writing macros responsibly. If you are confident you need to resort to macros, remember that macros are not your API. Keep your macro definitions short, including their quoted contents. For example, instead of writing a macro like this:

```
defmodule MyModule do
  defmacro my_macro(a, b, c) do
    quote do
      do_this(unquote(a))
      ...
      do_that(unquote(b))
      ...
      and_that(unquote(c))
    end
  end
end
```

write:

```
defmodule MyModule do
  defmacro my_macro(a, b, c) do
    quote do
      # Keep what you need to do here to a minimum
      # and move everything else to a function
      do_this_that_and_that(unquote(a), unquote(b), unquote(c))
    end
  end

  def do_this_that_and_that(a, b, c) do
    do_this(a)
    ...
    do_that(b)
    ...
    and_that(c)
  end
end
```

This makes your code clearer and easier to test and maintain, as you can invoke and test `do_this_that_and_that/3` directly. It also helps you design an actual API for developers that do not want to rely on macros.

With those lessons, we finish our introduction to macros. The next chapter is a brief discussion on DSLs that shows how we can mix macros and module attributes to annotate and extend modules and functions.

Domain Specific Languages

Foreword

Domain Specific Languages (DSL) allow developers to tailor their application to a particular domain. You don't need macros in order to have a DSL: every data structure and every function you define in your module is part of your Domain Specific Language.

For example, imagine we want to implement a Validator module which provides a data validation domain specific language. We could implement it using data structures, functions or macros. Let's see how those different DSLs would look like:

```
# 1\. data structures
import Validator
validate user, name: [length: 1..100],
               email: [matches: ~r/@/]

# 2\. functions
import Validator
user
|> validate_length(:name, 1..100)
|> validate_matches(:email, ~r/@/)

# 3\. macros + modules
defmodule MyValidator do
  use Validator
  validate_length :name, 1..100
  validate_matches :email, ~r/@/
end

MyValidator.validate(user)
```

Of all the approaches above, the first is definitely the most flexible. If our domain rules can be encoded with data structures, they are by far the easiest to compose and implement, as Elixir's standard library is filled with functions for manipulating different data types.

The second approach uses function calls which better suits more complex APIs (for example, if you need to pass many options) and reads nicely in Elixir thanks to the pipe operator.

The third approach, uses macros, and is by far the most complex. It will take more lines of code to implement, it is hard and expensive to test (compared to testing simple functions), and it limits how the user may use the library since all validations need to be defined inside a module.

To drive the point home, imagine you want to validate a certain attribute only if a given condition is met. We could easily achieve it with the first solution, by manipulating the data structure accordingly, or with the second solution by using conditionals (if/else) before invoking the function. However it is impossible to do so with the macros approach unless its DSL is augmented.

In other words:

```
data > functions > macros
```

That said, there are still cases where using macros and modules to build domain specific languages is useful. Since we have explored data structures and function definitions in the Getting Started guide, this chapter will explore how to use macros and module attributes to tackle more complex DSLs.

Building our own test case

The goal in this chapter is to build a module named `TestCase` that allows us to write the following:

```
defmodule MyTest do
  use TestCase

  test "arithmetic operations" do
    4 = 2 + 2
  end

  test "list operations" do
    [1, 2, 3] = [1, 2] ++ [3]
  end
end

MyTest.run
```

In the example above, by using `TestCase`, we can write tests using the `test` macro, which defines a function named `run` to automatically run all tests for us. Our prototype will simply rely on the match operator (`=`) as a mechanism to do assertions.

The `test` macro

Let's start by creating a module that simply defines and imports the `test` macro when used:

```
defmodule TestCase do
  # Callback invoked by `use`.
  #
  # For now it simply returns a quoted expression that
  # imports the module itself into the user code.
  @doc false
  defmacro __using__(_opts) do
    quote do
      import TestCase
    end
  end

  @doc """
  Defines a test case with the given description.

  ## Examples

      test "arithmetic operations" do
        4 = 2 + 2
      end

"""
  defmacro test(description, do: block) do
    function_name = String.to_atom("test " <> description)
    quote do
      def unquote(function_name)(), do: unquote(block)
    end
  end
end
```

Assuming we defined `TestCase` in a file named `tests.exs`, we can open it up by running `iex tests.exs` and define our first tests:

```
iex> defmodule MyTest do
...>   use TestCase
...>
...>   test "hello" do
...>     "hello" = "world"
...>   end
...> end
```

For now we don't have a mechanism to run tests, but we know that a function named "test hello" was defined behind the scenes. When we invoke it, it should fail:

```
iex> MyTest."test hello"()
** (MatchError) no match of right hand side value: "world"
```

Storing information with attributes

In order to finish our `TestCase` implementation, we need to be able to access all defined test cases. One way of doing this is by retrieving the tests at runtime via `__MODULE__.__info__(:functions)`, which returns a list of all functions in a given module. However, considering that we may want to store more information about each test besides the test name, a more flexible approach is required.

When discussing module attributes in earlier chapters, we mentioned how they can be used as temporary storage. That's exactly the property we will apply in this section.

In the `__using__/1` implementation, we will initialize a module attribute named `@tests` to an empty list, then store the name of each defined test in this attribute so the tests can be invoked from the `run` function.

Here is the updated code for the `TestCase` module:

```
defmodule TestCase do
  @doc false
  defmacro __using__(_opts) do
    quote do
      import TestCase

      # Initialize @tests to an empty list
      @tests []

      # Invoke TestCase.__before_compile__/1 before the module is compiled
      @before_compile TestCase
    end
  end

  @doc """
  Defines a test case with the given description.

  ## Examples

      test "arithmetic operations" do
        4 = 2 + 2
      end

"""
  defmacro test(description, do: block) do
    function_name = String.to_atom("test " <> description)
    quote do
      # Prepend the newly defined test to the list of tests
      @tests [unquote(function_name) | @tests]
      def unquote(function_name)(), do: unquote(block)
    end
  end

  # This will be invoked right before the target module is compiled
  # giving us the perfect opportunity to inject the `run/0` function
  @doc false
  defmacro __before_compile__(env) do
    quote do
      def run do
        Enum.each @tests, fn name ->
          IO.puts "Running #{name}"
          apply(__MODULE__, name, [])
        end
      end
    end
  end
end
```

By starting a new IEx session, we can now define our tests and run them:

```
iex> defmodule MyTest do
...>   use TestCase
...>
...>   test "hello" do
...>     "hello" = "world"
...>   end
...> end
iex> MyTest.run
Running test hello
** (MatchError) no match of right hand side value: "world"
```

Although we have overlooked some details, this is the main idea behind creating domain specific modules in Elixir. Macros enable us to return quoted expressions that are executed in the caller, which we can then use to transform code and store relevant information in the target module via module attributes. Finally, callbacks such as `@before_compile` allow us to inject code into the module when its definition is complete.

Besides `@before_compile`, there are other useful module attributes like `@on_definition` and `@after_compile`, which you can read more about in the docs for the `Module` module. You can also find useful information about macros and the compilation environment in the documentation for the `Macro` module and `Macro.Env`.