Meta-Programming In Elixir

Plataformatec

July 2014

1 Quote and unquote

An Elixir program can be represented by its own data structures. In this chapter, we will learn how 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 the next chapter.

1.1 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:

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

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

```
1 x> quote do: sum(1, 2, 3)
2 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:

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

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

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

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

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

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

```
x> 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`:

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

In general, the tuples above follow the following format:

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

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

```
um          #=> Atoms
0           #=> Numbers
,2]         #=> Lists
trings"     #=> Strings
ey, value}  #=> Tuples with two elements
```

Most of Elixir code has straight-forward translation to its underlying quoted expression. We recommend you to try out different code samples and see what gets out of it. For example, what `String.upcase("foo")` expands 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?

1.2 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. The number can be injected into the quoted representation by using `unquote`:

```
x> number = 13
x> Macro.to_string(quote do: 11 + unquote(number))
1 + 13"
```

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

```
x> fun = :hello
x> Macro.to_string(quote do: unquote(fun)(:world))
ello(: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:

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

That's when `unquote_splicing` becomes handy:

```
x> inner = [3, 4, 5]
x> 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, being able to transform code or write code that generates code during compilation.

1.3 Escaping

As we have seen 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. Nor a tuple with four elements. However, those values *can* be expressed as a quoted expression:

```
x> 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`:

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

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

In other words, it is important to make a distinction in between a regular Elixir value (like a list, a map, a process, a reference, etc) and a quoted expression. Some values like integers, atoms and strings have a quoted expression equal to the value itself. Other values like maps needs to be explicitly converted. Finally, some 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 quoted expressions related functions can be found in the `Macro` module.

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

2 Macros

Macros can be defined in Elixir using `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`.

2.1 Our first macro

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:

```
fmodule 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
d
```

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

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

```
$ iex macros.exs
```

And play with those definitions:

```
x> require Unless
x> Unless.macro_unless true, IO.puts "this should never be printed"
l
x> Unless.fun_unless true, IO.puts "this should never be printed"
his should never be printed"
l
```

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 the 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:

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

Our `macro_unless` macro received the following:
{% raw %}

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

{% endraw %}
And it then return a quoted expression as follows:
{% raw %}

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

{% endraw %}
We can actually verify this is the case by using `Macro.expand_once/2`:

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

`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__` still 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:

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

Not only `unless/2`, `defmacro/2`, `def/2`, `defprotocol/2` and many constructs 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`.

2.2 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:

```
fmodule Hygiene do
defmacro no_interference do
  quote do: a = 1
end
d

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

gieneTest.go
=> 13
```

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

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

fmodule HygieneTest do
def go do
```

```
  require Hygiene
  a = 13
  Hygiene.interference
  a
end
d

gieneTest.go
=> 1
```

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

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

However, a quoted variable is represented as:

```
fmodule Sample do
def quoted do
  quote do: x
end
d

mple.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 those two variables come from different contexts and handle them accordingly.

Elixir provides similar mechanisms for imports and aliases too. This guarantees macros 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 these cases, `Macro.var/2` can be used to define new variables:

```
fmodule Sample do
defmacro initialize_to_char_count(variables) do
  Enum.map variables, fn(name) ->
    var = Macro.var(name, nil)
    value = Atom.to_string(name)|> 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
d

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

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

2.3 The environment

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

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

Let's give it a try:

```
x> __ENV__.module
l
x> __ENV__.file
ex"
x> __ENV__.requires
Ex.Helpers, Kernel, Kernel.Typespec]
x> require Integer
l
x> __ENV__.requires
Ex.Helpers, Integer, Kernel, Kernel.Typespec]
```

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

2.4 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:

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

2.5 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 the macro are not going to affect the user code. Furthermore, functions 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. Before using 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;
- Macros' language is clear: many languages provide syntax shortcuts for `quote` and `unquote`. In Elixir, we preferred to have those explicitly spelled out, in order to clearly delimit the boundaries of a macro definition and its quoted expressions;

Even if Elixir attempts its best to provide a safe environment, the major responsibility still falls on the developers. That's why the first rule of the macro club is **write macros responsibly**. 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. Elixir already provides elegant mechanisms to write your every day code and macros should be saved as last resort.

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

3 Domain Specific Languages

Domain Specific Languages allows developers to specialize their application to a particular domain. There are many language features that come together to aid developers to write Domain Specific Languages and in this chapter we will focus on just one of them.

In particular, we will focus on how macros and module attributes can be used together to create domain specific modules, that are focused on solving one particular example. To showcase our implementation, we will write a very simple test case implementation.

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

```
fmodule MyTest do
use TestCase

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

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

Test.run
```

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

3.1 The `test` macro

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

```
fmodule TestCase do
# Callback invoked when 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 = binary_to_atom("test " <> description)
  quote do
    def unquote(function_name)(), do: unquote(block)
  end
end
d
```

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

```
x> 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, so we can invoke it and it should fail:

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

3.2 Storing information with attributes

In order to finish our `TestCase` implementation, we need to be able to retrieve all defined test cases. One way of doing such 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 with each tests beyond the test name, are more flexible approach is required.

When discussing module attributes in early chapters, we have discussed how they can be used as temporary storage and that's exactly what we will do in this section.

In the `__using__/1` implementation, we will initialize a module attribute named `@tests` to an empty list, then store each defined test in this attribute until we compile into into a `run` function.

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

```
1  fmodule TestCase do
2  @doc false
3  defmacro __using__(_opts) do
4    quote do
5      import TestCase
6
7      # Initialize @tests to an empty list
8      @tests []
9
10     # Invoke TestCase.__before_compile__/1 before the module is compiled
11     @before_compile TestCase
12   end
13 end
14
15 @doc """
16 Defines a test case with the given description.
17
18 ## Examples
19
20     test "arithmetic operations" do
21       4 = 2 + 2
22     end
23
24 """
25 defmacro test(description, do: block) do
26   function_name = binary_to_atom("test " <> description)
27   quote do
28     # Prepend the newly defined test to the list of tests
29     @tests [unquote(function_name)|@tests]
30     def unquote(function_name)(), do: unquote(block)
31   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
d
```

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

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

Although we have glanced over some details, the bulk of how we can create domain specific modules in Elixir is here. Macros allows us to return quoted expressions that are executed in the caller, which in turn we use to transform code and to store relevant information in the target module via module attributes. Finally, with callbacks such as `@before_compile`, we have the perfect opportunity 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` and you can more about read them in the docs for the `Module` module. You can also find useful documentation about macros in the `Macro` module and about the compilation environment in `Macro.Env`.