Metaprogramming + DSL Design in Elixir

by Adi Iyengar

About Me

- Adi Iyengar
  - GitHub: thebugcatcher
  - Twitter: aditya7iyengar
- Pronouns: He/Him
- Senior Software Engineer (Elixir, 5+ years)
- Loves catching bugs (software bugs)
- ❤️ TDD
- 💙 Elixir
- 💖 Theoretical Physics

Outline

- ➔ What is Metaprogramming
- ➔ When to use Metaprogramming
- ➔ Metaprogramming in Elixir
- ➔ Build a DSL in Elixir
- ➔ Questions

Metaprogramming: What is it?

Code that writes code!

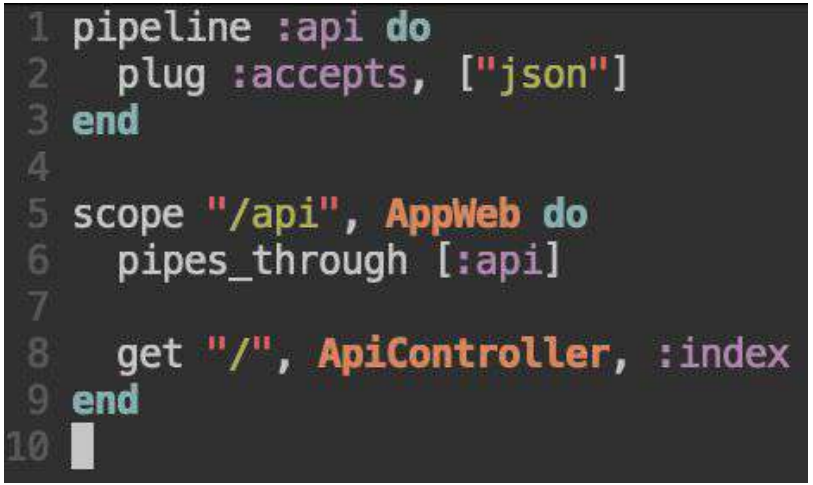
A program is a metaprogram if it:

- ➔ Generates Code
- ➔ Analyzes other code
- ➔ Stores information about other code
- ➔ Injects behavior into other code
- ➔ Treats other code as arguments or data

The language used to metaprogram is called the metalanguage.

Elixir's metalanguage is Elixir itself.. [***Reflectivity***]

Example: Phoenix Router Pipeline



Metaprogramming: Pros and Cons

Pros:

- ➔ Hides complexity of the implementation
- ➔ Increases developer productivity
- ➔ Automates and Standardizes tedious boilerplate code

Cons:

- ➔ Decreases transparency
- ➔ Increases *overall* code complexity

Example: Phoenix Router Pipeline

```
1  pipeline :api do
2    plug :accepts, ["json"]
3  end
4
5  scope "/api", AppWeb do
6    pipes_through [:api]
7
8    get "/", ApiController, :index
9  end
10
```

Metaprogramming: When to use it?

Use metaprogramming only when:

- ➔ You have exhausted all other options 🤷
- ➔ You have minimized the “meta” code (Separate interface from implementation)
- ➔ You have maximized its determinism via thorough unit and integration testing
- ➔ You have maximized its inspectability; The code should be debuggable
- ➔ The requirements for the DSL are less volatile; Less maintenance
- ➔ The cost of failure is manageable

Metaprogramming isn't evil, but it needs to be used thoughtfully.

Metaprogramming in Elixir

Three pillars of Metaprogramming in Elixir:

- ➔ Elixir representation of the Abstract Syntax Tree
  - ◆ *quoted expressions*
- ➔ Code/Behavior injection
  - ◆ *macros*
- ➔ Compile-time callbacks
  - ◆ *@before_compile, @after_compile and @on_definition*

Quoted Expressions

Elixir representation of the AST

quote/2 converts a block of code in Elixir to its AST representation

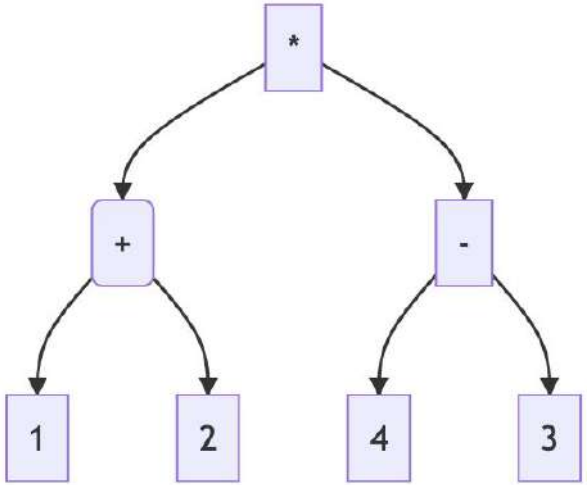
3 Element Tuple:

- ➔ Function
- ➔ Metadata
- ➔ Arguments

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

Quoted Expressions (AST)

```
iex> quote do: (1 + 2) * (4 - 3)
{:*, _metadata,
  [
    {:+, _metadata, [1, 2]},
    {:-, _metadata, [4, 3]}
  ]
}
```



Code.eval_quoted/3

Code.eval_quoted/3 can evaluate a quoted expression using a set of variable bindings and an environment.

It returns the final result with variable bindings after the evaluation.

```
iex> expr = quote do: 1 + 2
iex> Code.eval_quoted(expr)
{3, []}
```

Merging two quoted expressions

You can manually merge two quoted expressions by wrapping them in a 3 Element tuple with *:__block__* as the function.

```
iex> expr1 = quote do: 1 + a
iex> expr2 = quote do: a = 2
iex> expr = {:__block__, [], [expr2, expr1]}
iex> Code.eval_quoted(expr)
{3, [{{:a, Elixir}, 2}]}
```

Hygienic evaluation of quotes

Quoted Expressions are evaluated *hygienically.* This means variables don't leak across scopes, in and out of the quoted expression, upon evaluation.

So, anything defined inside an evaluated quoted expression doesn't conflict with the outer context.

And, anything defined outside the quoted expression doesn't conflict with the inner context.

```
iex> expr = quote do: b = 2
iex> Code.eval_quoted(expr)
{2, [{{:b, Elixir}, 2}]}
iex> b
** (CompileError) undefined function b/0
```

```
iex> a = 1
iex> expr = quote do: 1 + a
iex> Code.eval_quoted(expr)
warning: variable "a" does not exist ....
** (CompileError) undefined function a/0
```

var!/2 and unquote/2

To explicitly affect the context beyond the quoted expression boundary, we can use *var!/2* or *unquote/2*.

var!: evaluation of the quoted expression.

unquote: definition of the quoted expression.

```
iex> expr = quote do: 1 + var!(a)
{:+, _metadata,
 [
   1,
   {:var!, _metadata,
    [{:a, _metadata, Elixir}]}
 ]
}
iex> Code.eval_quoted(expr, [a: 2])
{3, [a: 2]}
```

```
iex> a = 2
iex> expr = quote do: 1 + unquote(a)
{:+, _metadata, [1, 2]}
iex> Code.eval_quoted(expr)
{3, []}
```

Code Injection (the bad way)

We can use *Code.eval_quoted/3* to inject code into a module at the time of its compilation.

```
defmodule Behavior do
  def behavior_ast do
    quote do
      def hello, do: "world"
    end
  end
end

defmodule Test do
  Code.eval_quoted(Behavior.behavior_ast(), [], __ENV__)
end

iex> Test.hello
"world"
```

Code Injection (the good way)

macro is the correct way of injecting code/behavior into another module at compile-time.

```elixir
defmodule Behavior do
  def behavior_ast do
    quote do
      def hello, do: "world"
    end
  end
end

defmodule Test do
  Code.eval_quoted(Behavior.behavior_ast(), [], __ENV__)
end

iex> Test.hello
"world"
```

```elixir
defmodule Behavior do
  defmacro behavior_ast do
    quote do
      def hello, do: "world"
    end
  end
end

defmodule Test do
  require Behavior
  Behavior.behavior_ast
end

iex> Test.hello
"world"
```

Code Injection (*use* keyword)

Elixir has a special macro *__using__/1* which can be invoked using the *use* keyword.

```elixir
defmodule Behavior do
  defmacro behavior_ast do
    quote do
      def hello, do: "world"
    end
  end
end

defmodule Test do
  require Behavior
  Behavior.behavior_ast
end

iex> Test.hello
"world"
```

```elixir
defmodule Behavior do
  defmacro __using__(_) do
    quote do
      def hello, do: "world"
    end
  end
end

defmodule Test do
  use Behavior
end

iex> Test.hello
"world"
```

Compile-time callbacks

Hook into the compilation of a module and change its behavior.

Elixir has 3 compile-time callbacks:

- ➔ *@before_compile*
- ➔ *@after_compile*
- ➔ *@on_definition*

@before_compile

- ➔ Invoked right before a module's bytecode is generated
- ➔ Takes the environment as the argument
- ➔ Needs to be defined in a different module

```elixir
defmodule Behavior do
  defmacro __using__(_) do
    quote do
      def hello, do: "world"
    end
  end
end

defmodule Test do
  use Behavior
end

iex> Test.hello
"world"
```

```elixir
defmodule Behavior do
  defmacro __before_compile__(_env) do
    quote do
      def hello, do: "world"
    end
  end
end

defmodule Test do
  @before_compile Behavior
end

iex> Test.hello
"world"
```

@after_compile

- ➔ Invoked after a module's bytecode is generated
- ➔ Takes the environment and bytecode as arguments
- ➔ Can be defined in the same module itself

```elixir
defmodule Test do
  @after_compile __MODULE__

  def __after_compile__(_env, _byte_code) do
    IO.puts "Compiled #{__MODULE__}"
  end
end

# On compilation it will print
Compiled Elixir.Test
```

@on_definition

- ➔ Invoked whenever a function/macro is defined in the current module
- ➔ Takes six arguments.
- ➔ Needs to be defined in a different module; can only be a function (no macros allowed)

```
defmodule OnDef do
  def __on_definition__(_, _, name, _, _, body) do
    IO.puts """
    Defining a function named #{name}
    with body:
    #{Macro.to_string(body)}
    """
  end
end

defmodule Test do
  @on_definition OnDef

  def hello, do: IO.puts "world"
end

# On compilation it will print
Defining a function named hello
with body:
     [do: IO.puts("world")]
```

Summary

- In Elixir, metaprogramming revolves around three constructs: quoted expressions, macros and compile-time callbacks.
- Quoted Expressions are Elixir representation of ASTs which are evaluated hygienically.
- To add dynamic behavior to them, use *var!* (evaluation-time) or *unquote* (definition-time).
- Macros are used to inject behavior using quoted expressions at compile-time.
- Compile-time callbacks are used to run tasks (or add behavior) by hooking into the compile-time of a module.
- Metaprogramming should be used carefully, as it makes code more complex.
- Use simple metaprogramming to make code digestible. A DSL is a good use case.

Let's build a DSL

- A simple DSL to compose music in Elixir
- Calls ALSA's *aplay* command to play a note
- Define a *sequence* of *notes*.
- A note needs to have a class (rest, C, D, E, F..), a modifier (*sharp* or *base*), octet, duration and volume (with defaults).
- A sequence can *embed notes* from other sequences.
- Very much inspired by the phoenix router DSL

```elixir
defmodule Music do
  use DSL

  sequence :intro do
    note :c, modifier: :sharp, octet: 4, duration: 0.5, volume: 50
    note :d, modifier: :base, octet: 4, duration: 0.5
    note :rest, octet: 0, duration: 0.5
    note :e, octet: 4, duration: 0.5
  end

  sequence :outro do
    note :c, octet: 4, modifier: :sharp, duration: 0.5
    note :d, octet: 4, modifier: :sharp, duration: 0.5
  end

  sequence :final do
    embed_notes :intro
    embed_notes :outro
  end
end

# Play a sequence using the function `play/1`
Music.play(:final)
```

Things already done

- *Note* module/struct, representing a note to be played, along with defaults.
  - `%Note{class: :a, modifier: :base, octet: 4}`
- *NotePlayer* module, which calls ALSA's *aplay* command.
  - Use `NotePlayer.play/1` function which takes a `%Note{}`
- Unit and Integration Tests for our DSL.
  - TDD!!
- Final (super awesome) track using the expected DSL.
  - This will only work once the DSL is done. *#Incentive*

TODO

- *DSL.__using__/1* macro
- *sequence/2* macro
  - A way to store a list of notes
- *note/2* macro inside *sequence/2*
  - Add to list of notes under current sequence
- *embed_notes/1* macro inside *sequence/2*
  - Add a list of notes from an existing sequence to current sequence
  - Track defined sequences
- *play/1* function which takes a sequence
  - Use *NotePlayer.play/1* to play a list of notes under a sequence

```elixir
defmodule Music do
  use DSL

  sequence :intro do
    note :c, modifier: :sharp, octet: 4, duration: 0.5, volume: 50
    note :d, modifier: :base, octet: 4, duration: 0.5
    note :rest, octet: 0, duration: 0.5
    note :e, octet: 4, duration: 0.5
  end

  sequence :outro do
    note :c, octet: 4, modifier: :sharp, duration: 0.5
    note :d, octet: 4, modifier: :sharp, duration: 0.5
  end

  sequence :final do
    embed_notes :intro
    embed_notes :outro
  end
end

# Play a sequence using the function `play/1`
Music.play(:final)
```

Let’s Code! 💻