Intro to Elixir macro

@tony612

Overview

- What's macro?
- Why macro?
- How is macro implemented in Elixir?
- How to use?
- When to use?

What’s macro?

Macro in Lisp

A macro is an ordinary piece of Lisp code that operates on another piece of putative Lisp code, translating it into (a version closer to) executable Lisp.

http://cl-cookbook.sourceforge.net/macros.html

Macro in Elixir

A macro is an ordinary piece of Elixir code that operates on another piece of quoted Elixir code, translating it into quoted Elixir.

defmacro

```
defmodule MyLogic do
  defmacro unless(expr, opts) do
    quote do
      if !unquote(expr), unquote(opts)
    end
  end
end

require MyLogic
MyLogic.unless false do
  IO.puts "It works"
end
```

Elixir code to AST

```
sum(1, 2, 3)
=>
{:sum, [], [1, 2, 3]}

x
=>
{:x, [], Elixir}

%{1 => 2}
=>
{:%{}, [], [{1, 2}]}

Foo.run(1, 2)
=>
{{:.,
  [], [{:__aliases__, [alias: false], [:Foo]}, :run]},
 [], [1, 2]}
```

quote

```
quote do
  defmodule Foo do
    def foo do
      1
    end
  end
end
{:defmodule, [context: Elixir, import: Kernel],
 [{:__aliases__, [alias: false], [:Foo]},
  [do: {:def, [context: Elixir, import: Kernel],
    [{:foo, [context: Elixir], Elixir}, [do: 1]]}]]}
```

Metaprogramming

(wikipedia)

- Computer programs have the ability to treat programs as their data
- A program can be designed to read, generate, analyse or transform other programs

Why macro?

Metaprogramming

(wikipedia)

- Minimize the amount of code to express a solution, and thus reducing the development time
- Move computations from run-time to compile-time
- Generate code using compile time computations

less code, more effective

```elixir
defmodule MyRouter do
  use Plug.Router

  plug :match        # middleware
  plug :dispatch     # middleware

  get "/hello" do    # logic for a route
    send_resp(conn, 200, "world")
  end
end
```

compute at compile-time

```
# config.yml
web_port: 8080

Config.get(:web_port)
=>
8080
```

(not a good example)

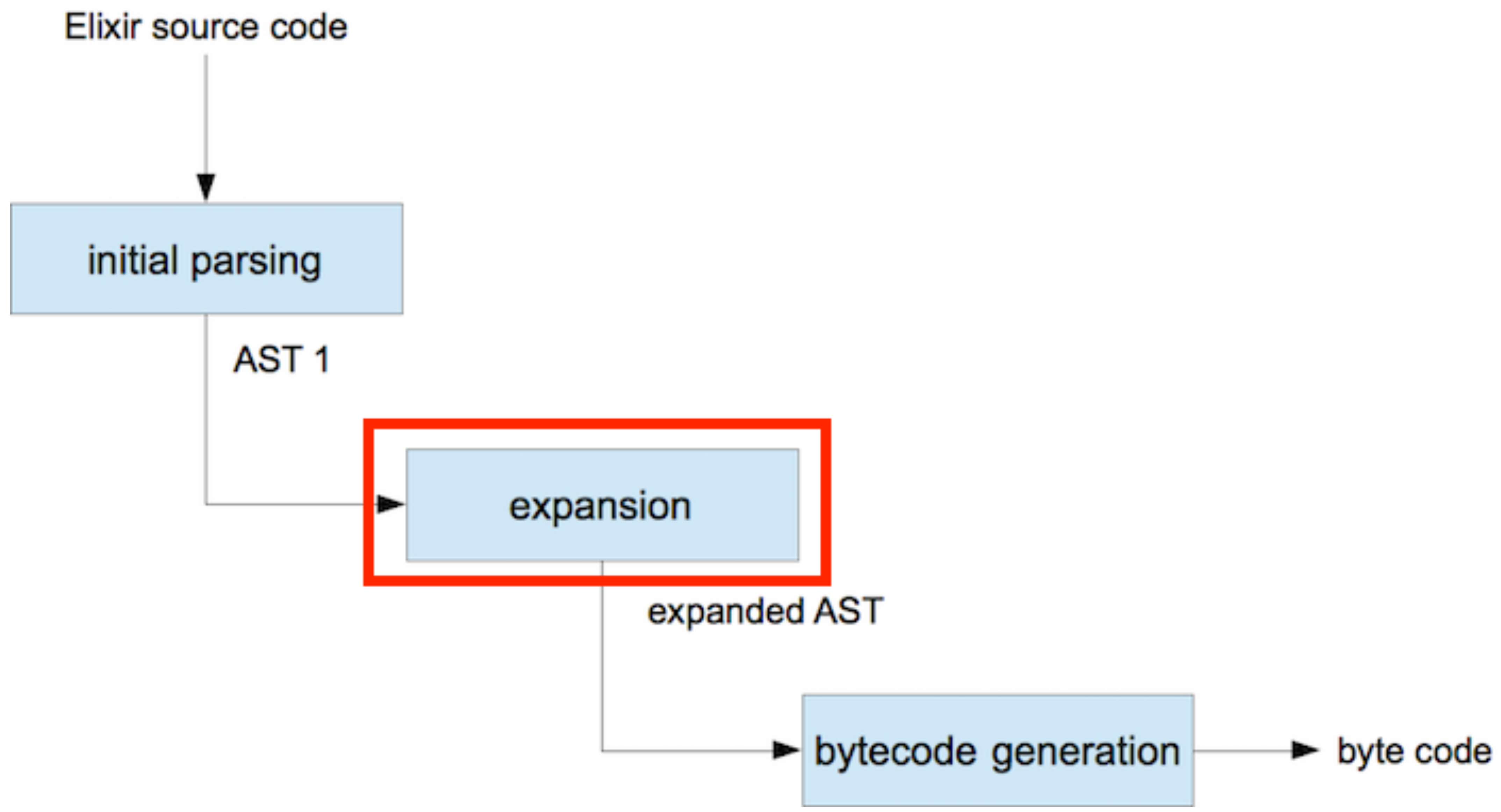
Generate code

```
iex> String.upcase("Elixir Shànghǎi")
"ELIXIR SHÀNGHǍI"
```

https://github.com/elixir-lang/elixir/blob/master/lib/elixir/unicode/unicode.ex#L376

How is macro implemented in Elixir?

Compiling of Elixir



An example

```
defmodule Foo do
  defmacro add1(x) do
    quote do
      x + 1
    end
  end
end

Foo.add1(2)
```

1. Code to AST

```
quote do: Foo.add1(2)
=>
{
  {:.,
    [], [{:__aliases__, [alias: false],
[:Foo]}, :add1]},
  [], [2]}
```

2. Macro expansion

```
require Foo
Macro.expand(quote(do: Foo.add1(2)), __ENV__)
=>
{:+, [context: Foo, import: Kernel],
 [{:x, [], Foo}, 1]}
```

```
quote do: Foo.add1(2)
=>
{
  {:.,
    [], [{:__aliases__, [alias: false], [:Foo]}, :add1]},
  [], [2]}
```

AST to code

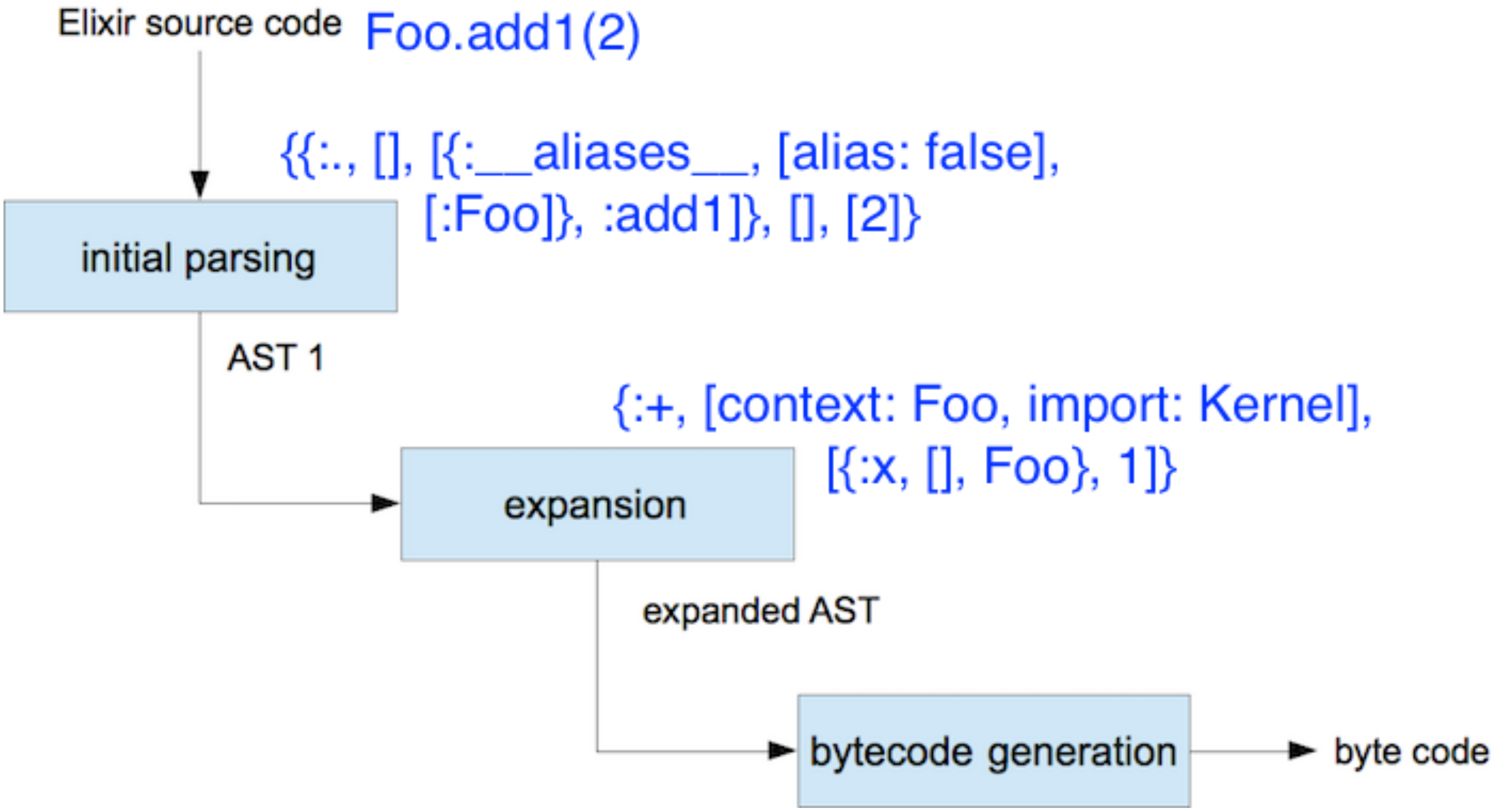
```
Macro.to_string({:+, [context: Foo, import: Kernel],
  [{:x, [], Foo}, 1]})
=>
"x + 1"
```

So our code changes after expansion

```
Foo.add1(2)

=>

x + 1
```



```
Foo.add1(2)
```

```
{{:., [], [{:__aliases__, [alias: false],
    [:Foo]}, :add1]}, [], [2]}
```

```
{:+, [context: Foo, import: Kernel],
    [{:x, [], Foo}, 1]}
```

How to use?

1. Think of the final code(after expansion)

2. Define macros to generate the code

3. Write and run unit tests(or run in iex)

4. (Repeat)

Example: a simple FSM lib using DSL

Usage of the lib

```elixir
defmodule Door do
  trans :push, "closed", "opened"
  trans :pull, "opened", "closed"
end

state0 = "closed"
state1 = Door.push(state0) # "opened"
state2 = Door.pull(state1) # "closed"
```

The expanded code

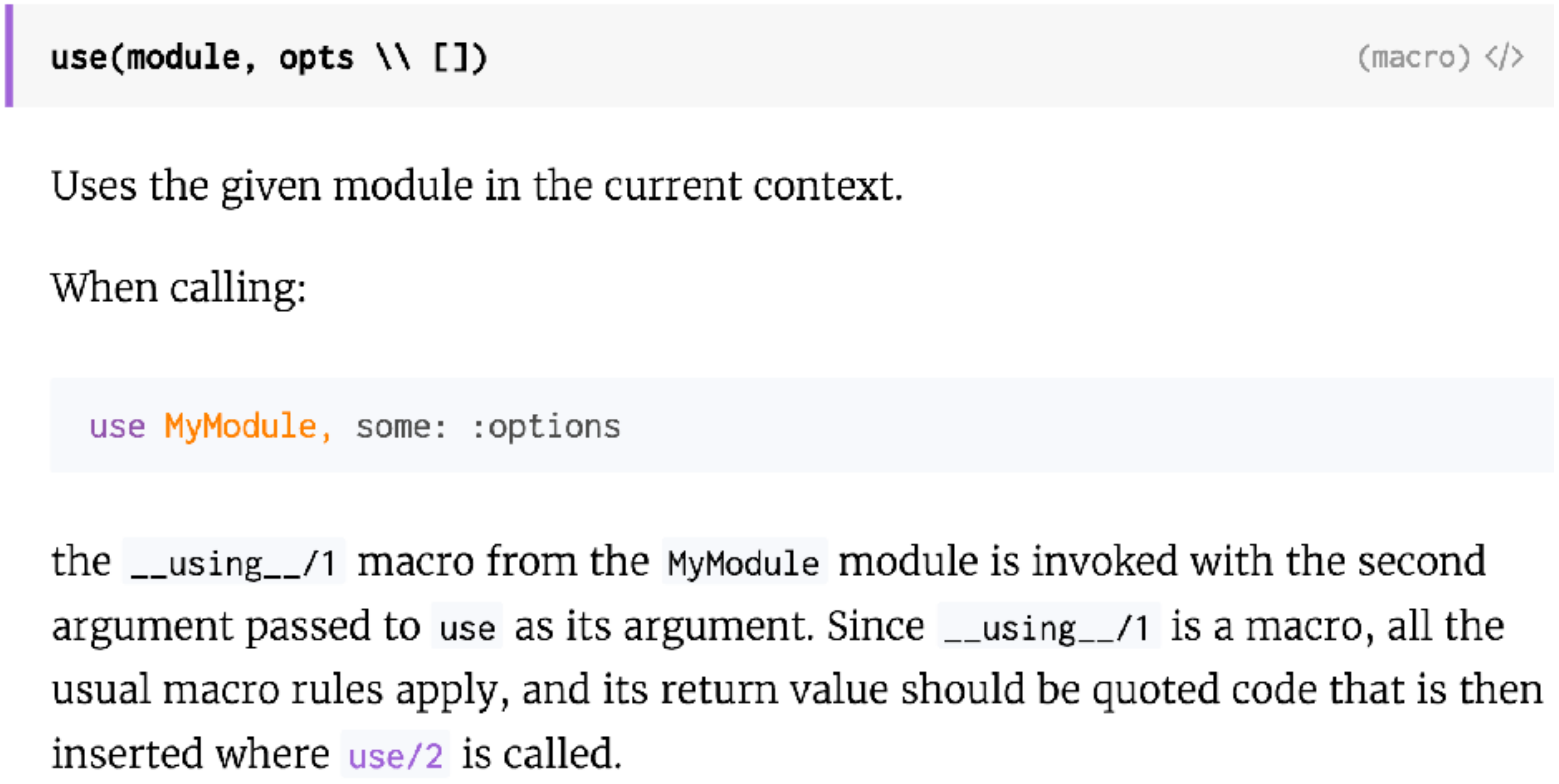
```
defmodule Door do
  def push("closed") do
    "opened"
  end

  def pull("opened") do
    "closed"
  end
end
```

So we need

1. The ability to inject macro trans
2. Use trans to generate functions

use to inject a module



```
use(module, opts \\ [])
```

(macro) </>

Uses the given module in the current context.

When calling:

```
use MyModule, some: :options
```

the `__using__/1` macro from the `MyModule` module is invoked with the second argument passed to `use` as its argument. Since `__using__/1` is a macro, all the usual macro rules apply, and its return value should be quoted code that is then inserted where `use/2` is called.

https://hexdocs.pm/elixir/Kernel.html#use/2

```elixir
defmodule Door do
  use FSM
  trans :push, "closed", "opened"
  trans :pull, "opened", "closed"
end

# expanded =>

defmodule Door do
  import FSM.DSL, only: [trans: 3]
  # ...
end
```

```elixir
defmodule FSM do
  defmacro __using__(opts) do
    quote do
      import FSM.DSL, only: [trans: 3]
    end
  end
end

defmodule FSM.DSL do
  defmacro trans(name, from, to) do
    # ...
  end
end
```

Implement trans in FSM.DSL

```
defmacro trans(name, from, to) do
  quote do
    def name(from) do
      to
    end
  end
end
```

This is wrong!

Generated code will be

```
defmodule Door do
  def name(from) do
    to
  end

  def name(from) do
    to
  end
end
```

This is wrong!

unquote to compute expression before macro expansion

```
defmacro trans(name, from, to) do
  quote do
    def unquote(name)(unquote(from)) do
      unquote(to)
    end
  end
end
```

The code can be run in iex

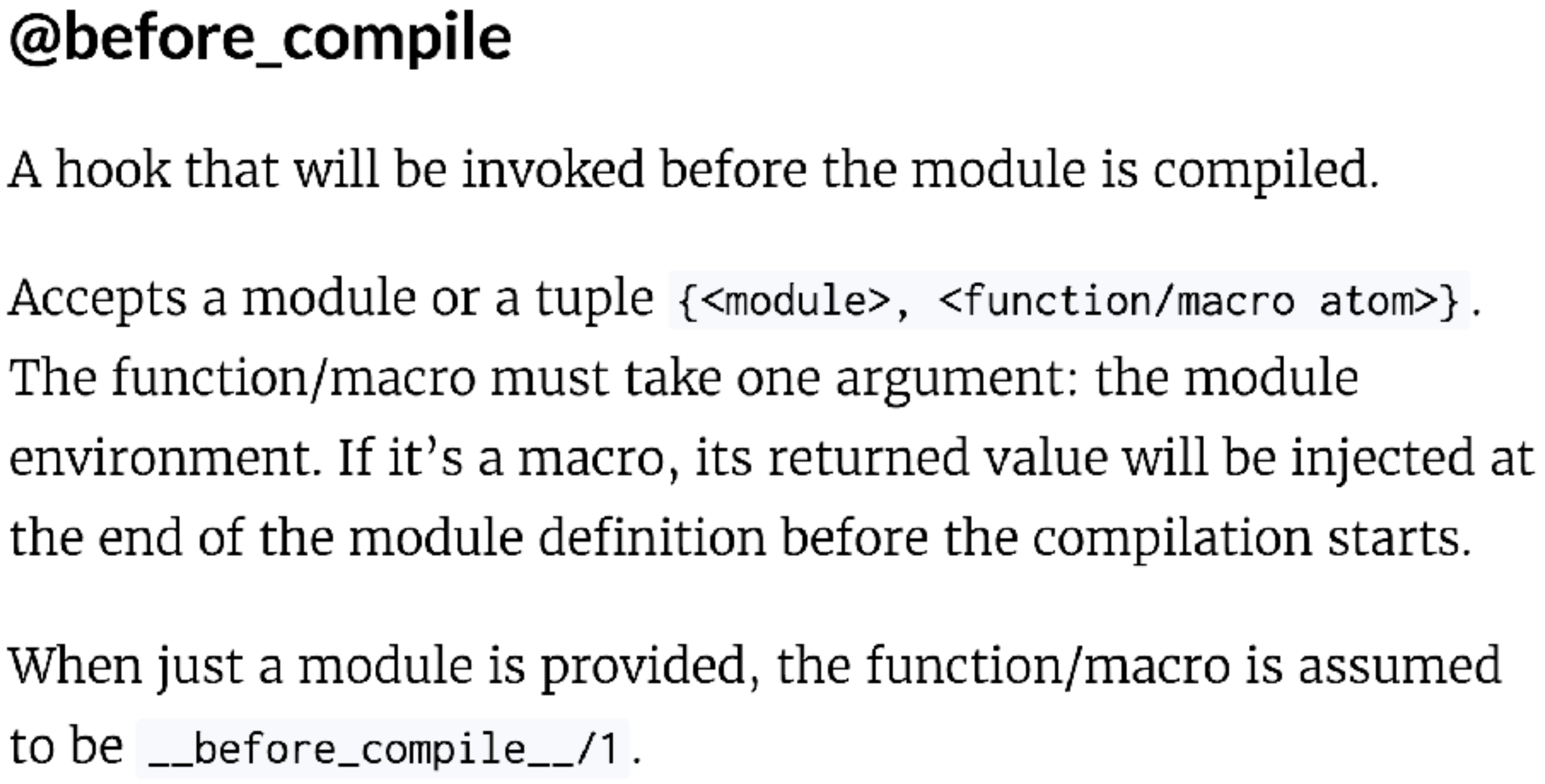
https://gist.github.com/tony612/720287cc2f8701e2b6bcabe294212a17#file-fsm1-exs

An advanced feature:
List all events of the FSM

```elixir
defmodule Door do
  use FSM
  trans :push, "closed", "opened"
  trans :pull, "opened", "closed"
end

Door.__events__ # => [:push, :pull]
```

- We can use module attributes to save all events
- But module attributes can’t be accessed in runtime
- So we need a function to save the attributes in a module just before compiling(compiled to bytes code)



@before_compile

A hook that will be invoked before the module is compiled.

Accepts a module or a tuple `{<module>, <function/macro atom>}`. The function/macro must take one argument: the module environment. If it's a macro, its returned value will be injected at the end of the module definition before the compilation starts.

When just a module is provided, the function/macro is assumed to be `__before_compile__/1`.

https://hexdocs.pm/elixir/Module.html#module-compile-callbacks

```elixir
defmodule Door do
  @before_compile FSM.DSL

  # ...

  # expanded
  def __events__ do
    # ...
  end
end
```

```elixir
defmodule FSM do
  defmacro __using__(opts) do
    quote do
      @before_compile FSM.DSL
    end
  end
end

defmodule FSM.DSL do
  defmacro __before_compile__(env) do
    quote do
      def __events__ do
        # ...
      end
    end
  end
end
```

Use module attributes to list events

```
defmodule FSM do
  defmacro __using__(opts) do
    quote do
      Module.register_attribute(__MODULE__, :events,
        accumulate: true)
    end
  end
end

defmacro trans(name, from, to) do
  quote do
    @events unquote(name)
    # ...
  end
end
```

…and define __events__ before compiling

```
defmodule FSM.DSL do
  defmacro __before_compile__(env) do
    fields = Module.get_attribute(env.module, :events)
    quote do
      def __events__ do
        unquote(Enum.reverse(fields))
      end
    end
  end
end
```

The code can be run in iex

https://gist.github.com/tony612/720287cc2f8701e2b6bcabe294212a17#file-fsm2-exs

When to use macro?

- When you need the benefits of macros
- When functions can’t solve your problems
- When you write a lib and want to provide “friendly” API
- Macros should only be used as a last resort

Write macros responsibly and keep your macro definitions short

http://elixir-lang.org/getting-started/meta/macros.html#write-macros-responsibly

“Explicit is better than implicit. Clear code is better than concise code.”

http://elixir-lang.org/getting-started/meta/macros.html

Q&A

Many thanks to @aquarhead
for reviewing