Introducing Elixir

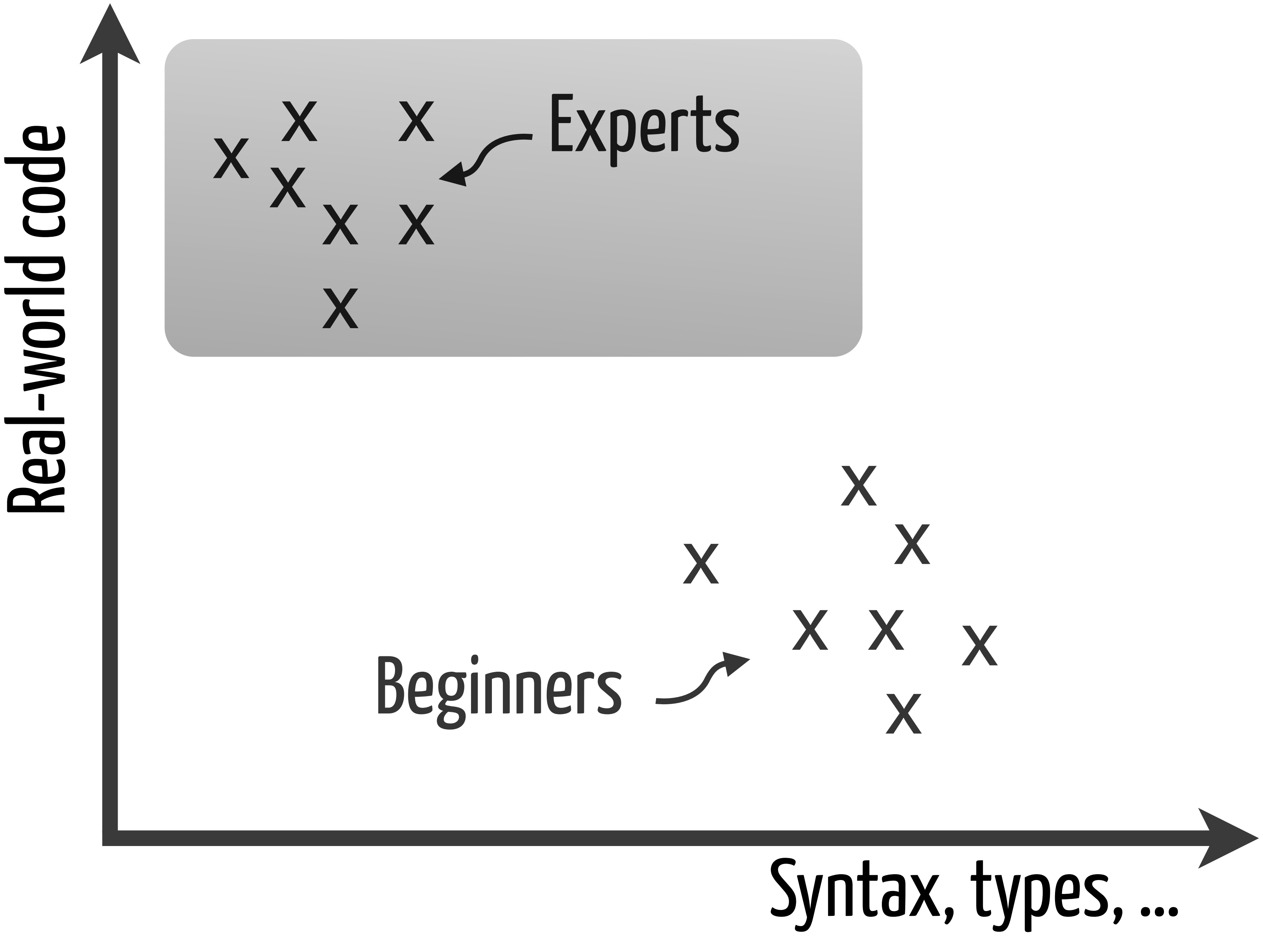
Dave Thomas
[@+]pragdave
dave@pragprog.com

Pragmatic Bookshelf

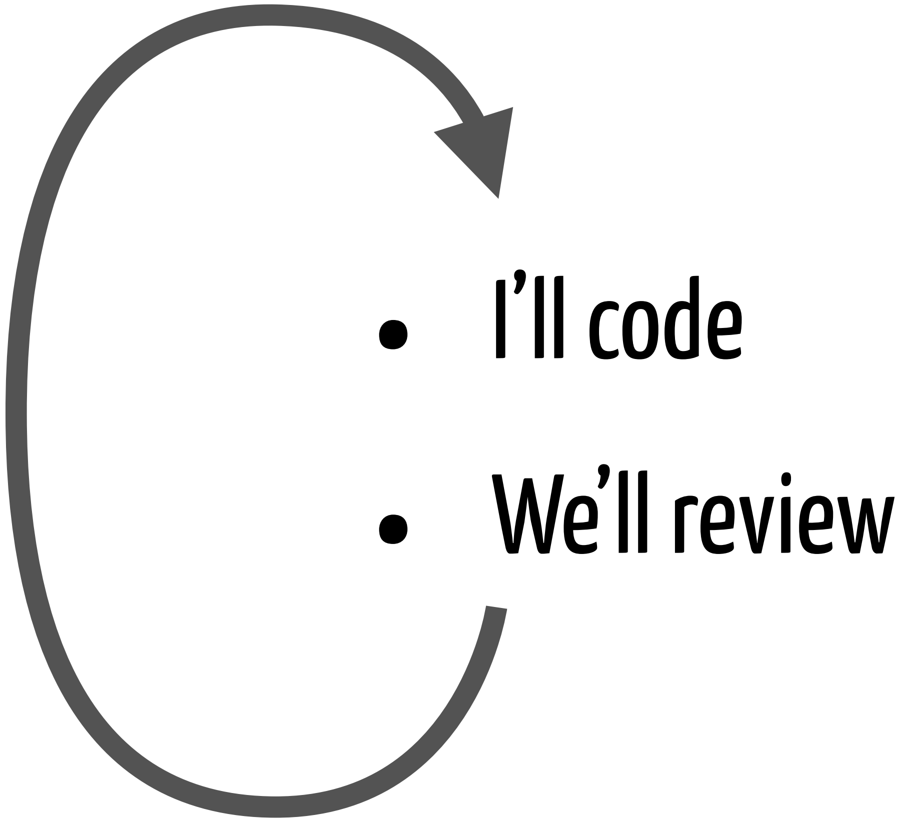
Goals

- Skip the frustrating “beginner” stage
- Get experience writing Elixir code
- Look briefly at more advanced stuff
- Have fun

Plan



Plan



- I'll code
- We'll review

You interact!!!

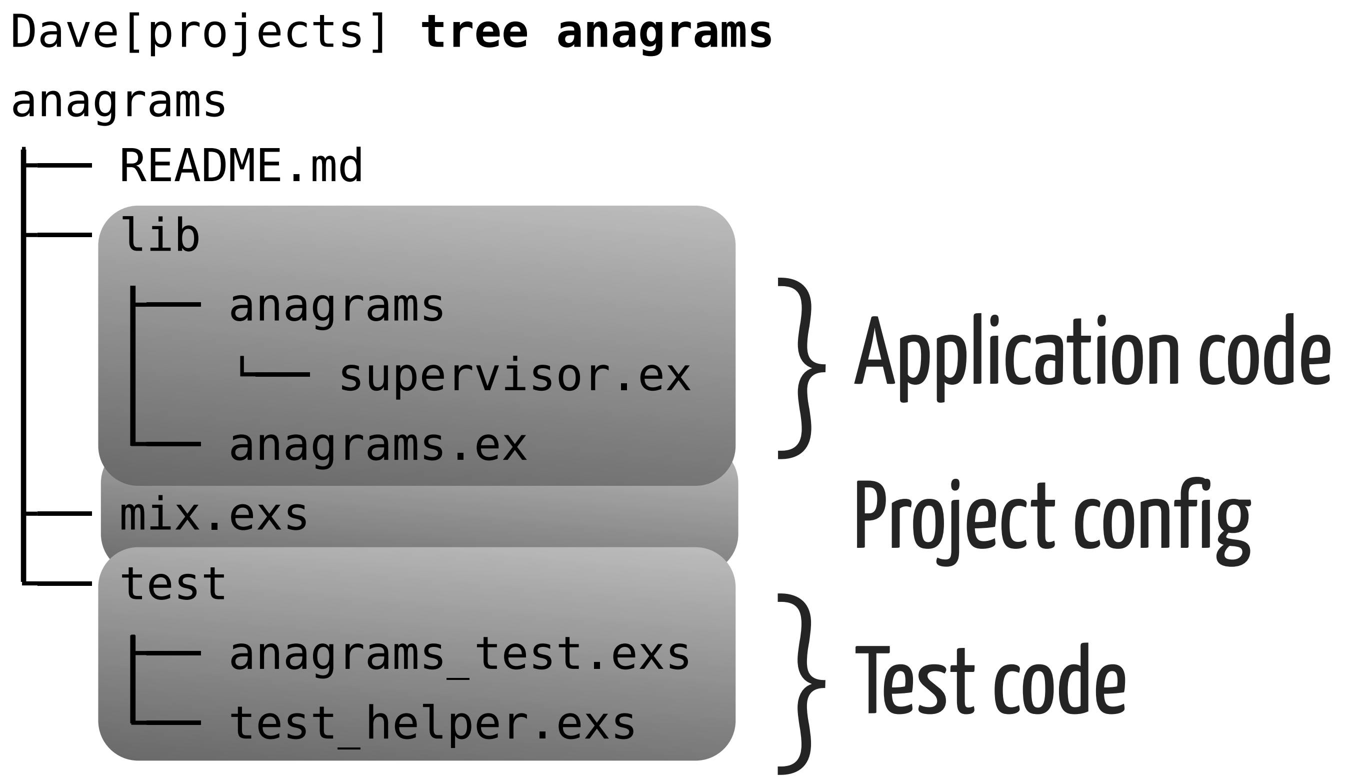
Start New Project

mix: The Elixir Project Tool

- Create new projects
- Automate projects
- Manage dependencies
- Run tests

```
Dave[projects] mix --help
mix                 # Run the default task (current: mix run)
mix archive         # Archive this project into a .ez file
mix clean           # Clean generated application files
mix cmd             # Executes the given command
mix compile         # Compile source files
mix deps            # List dependencies and their status
mix deps.clean      # Remove the given dependencies' files
mix deps.compile    # Compile dependencies
mix deps.get        # Get all out of date dependencies
mix deps.unlock     # Unlock the given dependencies
mix deps.update     # Update the given dependencies
mix do              # Executes the tasks separated by comma
mix escriptize      # Generates an escript for the project
mix help            # Print help information for tasks
mix local           # List local tasks
mix local.install   # Install a task or an archive locally
mix local.rebar     # Install rebar locally
mix local.uninstall # Uninstall local tasks or archives
mix new             # Creates a new Elixir project
mix run             # Run the given file or expression
mix test            # Run a project's tests
```

```
Dave[projects] mix new anagrams
* creating README.md
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/anagrams.ex
* creating lib/anagrams
* creating lib/anagrams/supervisor.ex
* creating test
* creating test/test_helper.exs
* creating test/anagrams_test.exs

Your mix project was created successfully.
You can use mix to compile it, test it, and more:

    cd anagrams
    mix compile
    mix test

Run `mix help` for more information.
```



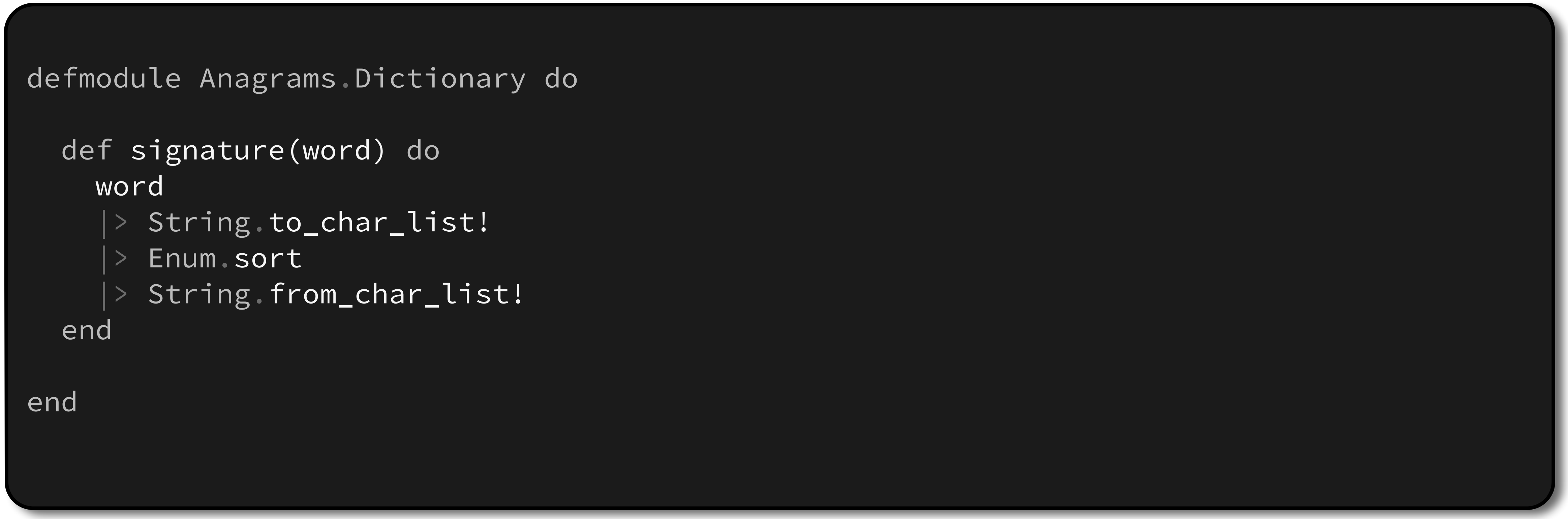
```
Dave[projects] tree anagrams
anagrams
├── README.md
├── lib
│   ├── anagrams
│   │   └── supervisor.ex
│   └── anagrams.ex
├── mix.exs
└── test
    ├── anagrams_test.exs
    └── test_helper.exs
```

Start Writing Code

```
Dave[projects] tree anagrams
anagrams
├── README.md
├── lib
│   ├── anagrams
│   │   └── dictionary.ex
│   │   └── supervisor.ex
│   └── anagrams.ex
├── mix.exs
└── test
    ├── anagrams_test.exs
    └── test_helper.exs
```

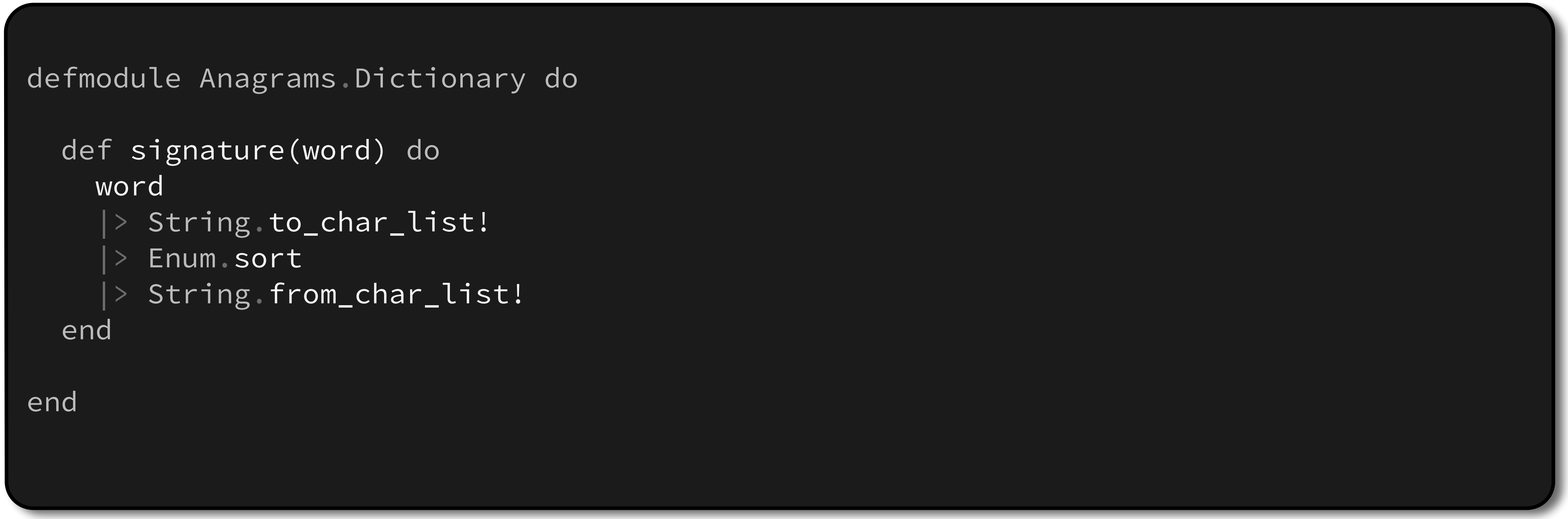
file path: anagrams/dictionary.ex

```
defmodule Anagrams.Dictionary do

  def signature(word) do
    word
     |> String.to_char_list!
     |> Enum.sort
     |> String.from_char_list!
  end

end
```



```
defmodule Anagrams.Dictionary do

  def signature(word) do
    word
     |> String.to_char_list!
     |> Enum.sort
     |> String.from_char_list!
  end

end
```

- defmodule, def, etc are Elixir macros
- take two parameters: name and do block
- do is actually syntactic sugar for a keyword parameter do:



```
defmodule Anagrams.Dictionary do

  def signature(word) do
    word
     |> String.to_char_list!
     |> Enum.sort
     |> String.from_char_list!
  end

end
```

```
def the_answer do
  42
end
```

```
def the_answer, do:42
```

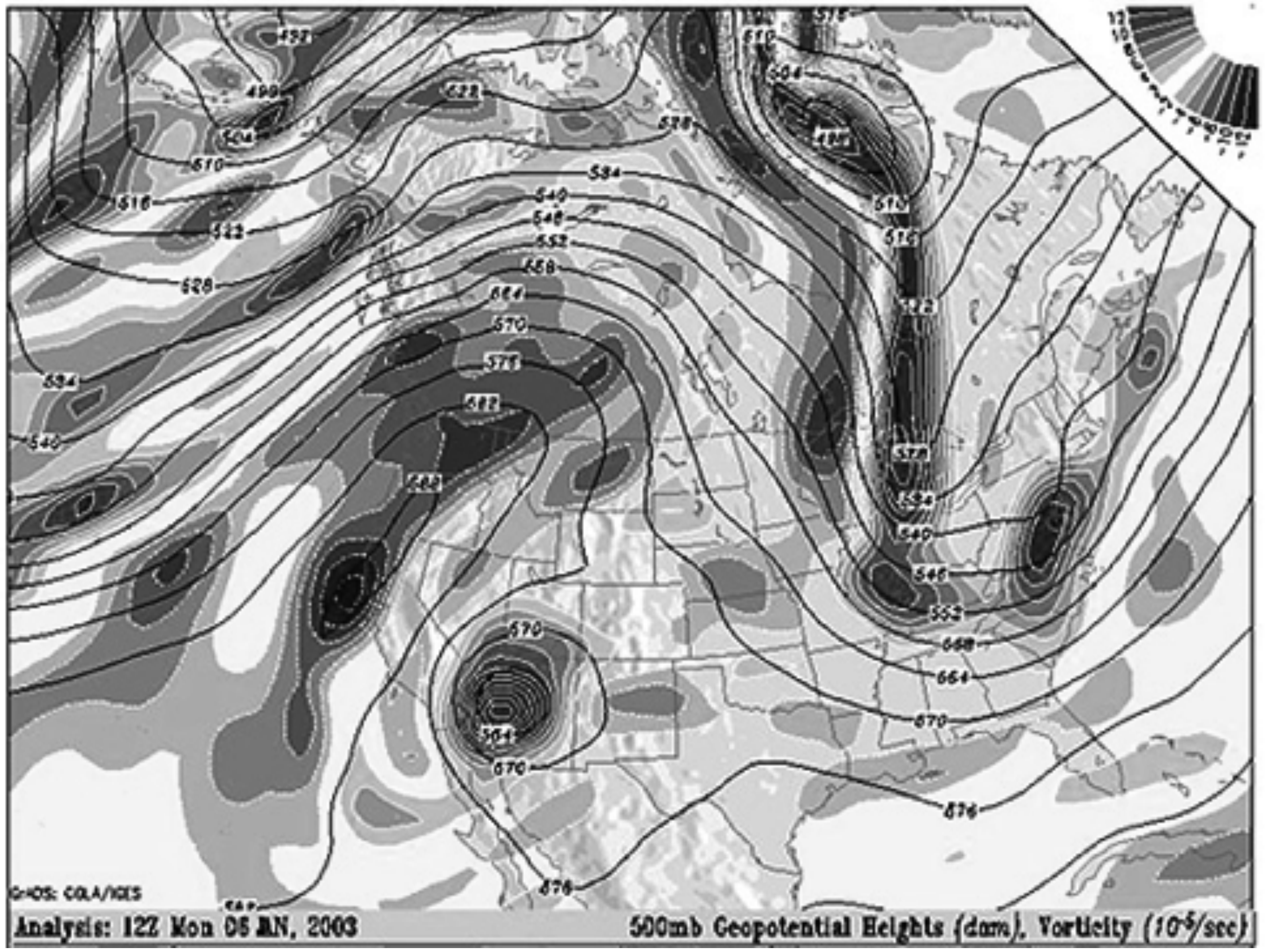
What is Functional Programming?

Functions

-10..10 ➙ sin ➙

Functions

sensor data → analysis →


GrADS: COLA/IGES
Analysis: 12Z Mon 06 JAN, 2003
500mb Geopotential Heights (dam), Vorticity (10-5/sec)

https://www.fas.org/irp/imint/docs/rst/Sect14/Sect14_1c.html

Functions

orders ➙ fulfillment ➙

What is Functional Programming?

Transformation

- take a word
- convert to list of characters
- sort the list
- reassemble into a string

```
String.from_char_list!(Enum.sort(String.to_char_list!(word)))
```

```
chars = String.to_char_list!(word)
sorted = Enum.sort(chars)
String.from_char_list!(sorted)
```

```
word
|> String.to_char_list!
|> Enum.sort
|> String.from_char_list!
```

|>

```
word |> String.to_char_list!
```

↓

```
String.to_char_list!(word)
```

|>

```
expr |> func(a, b, c)
```

↓

```
func(expr, a, b, c)
```

|>

```
expr |> f1(a) |> f2(b)
```

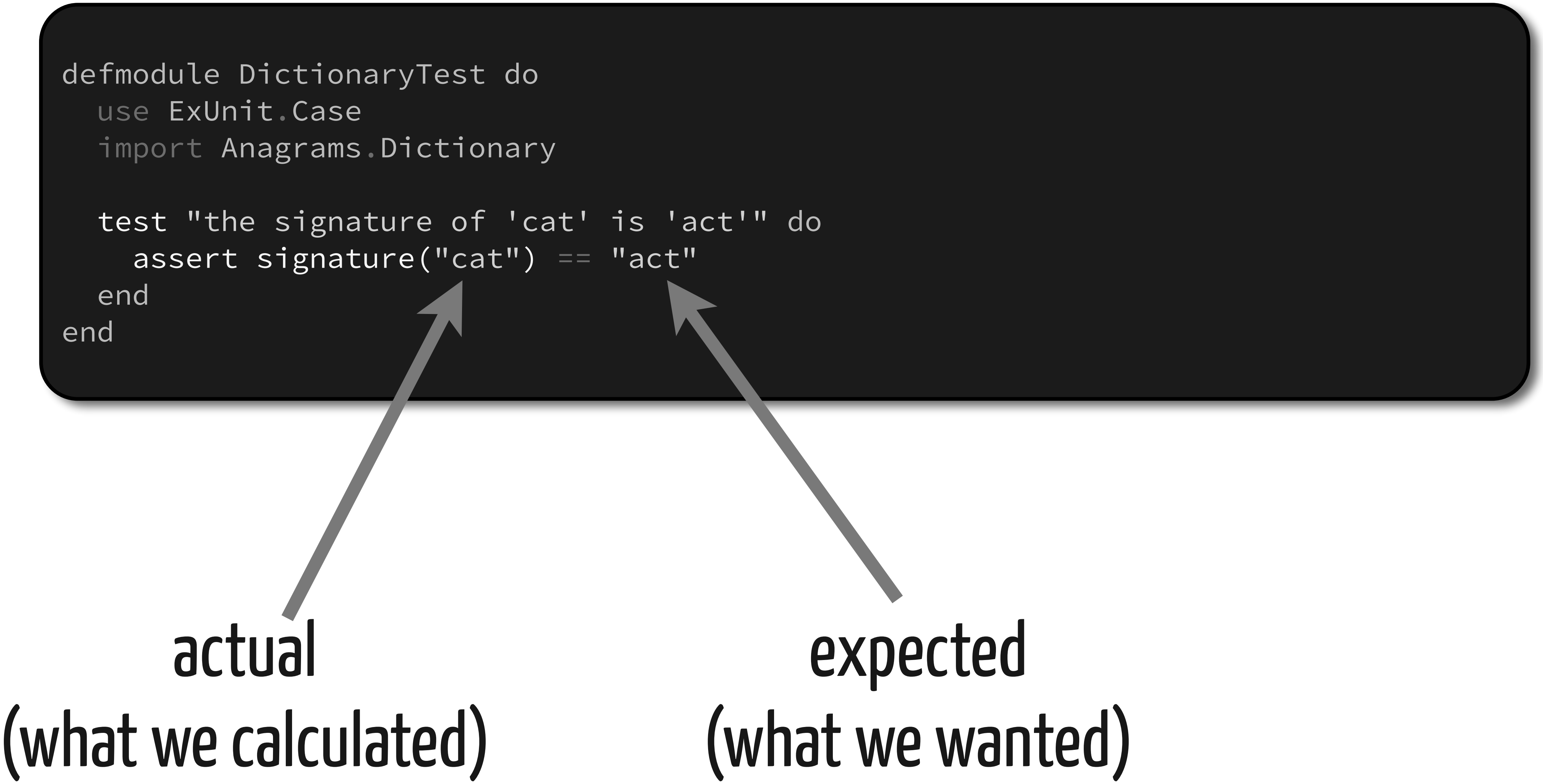
↓

```
f2(f1(expr, a), b)
```

|>

```
word
|> String.to_char_list!
|> Enum.sort
|> String.from_char_list!
```

Testing



```
defmodule DictionaryTest do
  use ExUnit.Case
  import Anagrams.Dictionary

  test "the signature of 'cat' is 'act'" do
    assert signature("cat") == "act"
  end
end
```

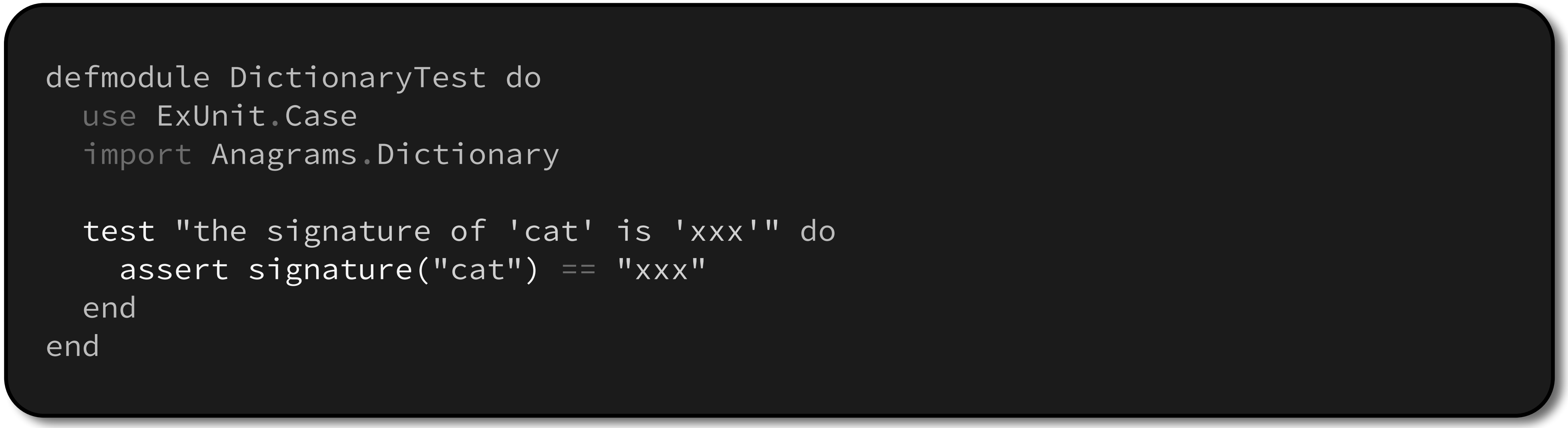
actual
(what we calculated)

expected
(what we wanted)

```
$ mix test
Compiled lib/anagrams/dictionary.ex
Generated anagrams.app
.

Finished in 0.09 seconds (0.09s on load, 0.00s on tests)
5 tests, 0 failures
```

```
defmodule DictionaryTest do
  use ExUnit.Case
  import Anagrams.Dictionary

  test "the signature of 'cat' is 'xxx'" do
    assert signature("cat") == "xxx"
  end
end
```

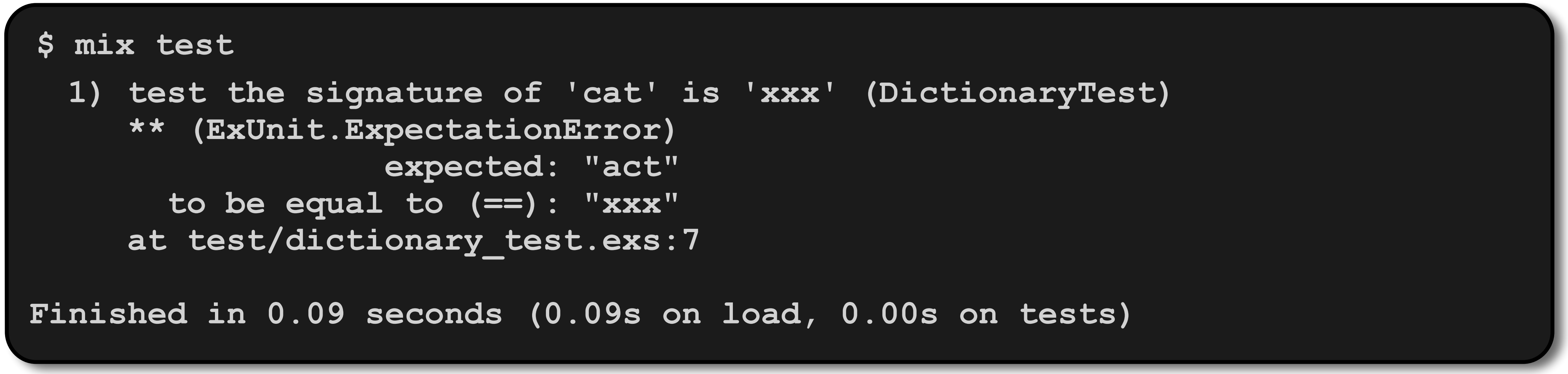
```
$ mix test
  1) test the signature of 'cat' is 'xxx' (DictionaryTest)
     ** (ExUnit.ExpectationError)
                  expected: "act"
        to be equal to (==): "xxx"
     at test/dictionary_test.exs:7

Finished in 0.09 seconds (0.09s on load, 0.00s on tests)
```



```
defmodule DictionaryTest do
  use ExUnit.Case
  import Anagrams.Dictionary

  test "the signature of 'cat' is 'xxx'" do
    assert signature("cat") == "xxx"
  end
end
```

How?

`assert` is an Elixir macro which overrides `==` `>`, `<` etc



```
$ mix test
  1) test the signature of 'cat' is 'xxx' (DictionaryTest)
     ** (ExUnit.ExpectationError)
                 expected: "act"
       to be equal to (==): "xxx"
     at test/dictionary_test.exs:7

Finished in 0.09 seconds (0.09s on load, 0.00s on tests)
```

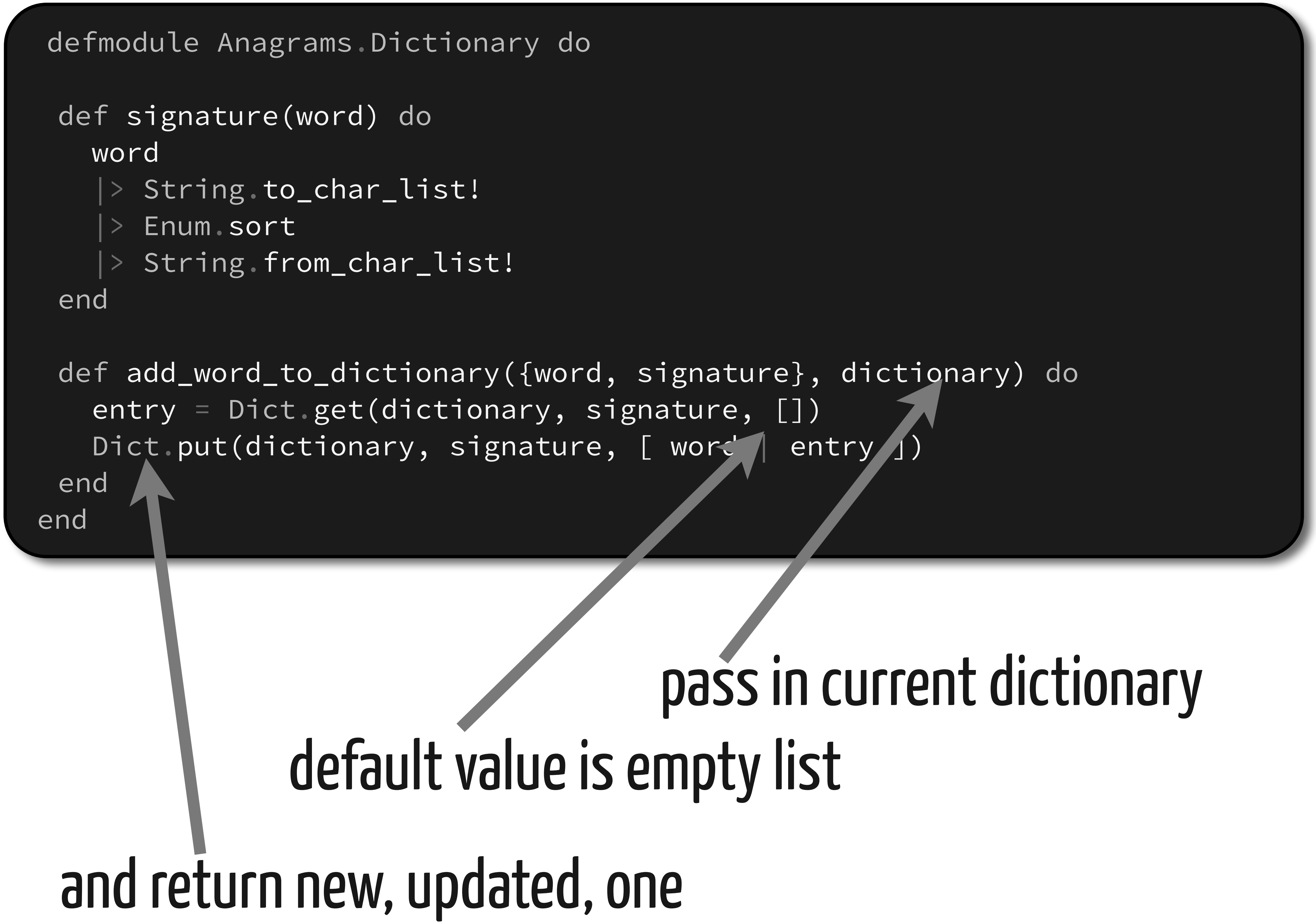
```
test “another example” do
  assert 2 > 1 + 3
end
```

```
$ mix test
  1) test another example (DictionaryTest)
     ** (ExUnit.ExpectationError)
              expected: 2
       to be more than: 4
     at test/some_test.exs:7
```

Build Dictionary

```
defmodule Anagrams.Dictionary do

  def signature(word) do
    word
    |> String.to_char_list!
    |> Enum.sort
    |> String.from_char_list!
  end

  def add_word_to_dictionary({word, signature}, dictionary) do
    entry = Dict.get(dictionary, signature, [])
    Dict.put(dictionary, signature, [ word | entry ])
  end
end
```

pass new word and signature as a tuple



```
defmodule Anagrams.Dictionary do

 def signature(word) do
   word
   |> String.to_char_list!
   |> Enum.sort
   |> String.from_char_list!
 end

 def add_word_to_dictionary({word, signature}, dictionary) do
   entry = Dict.get(dictionary, signature, [])
   Dict.put(dictionary, signature, [ word | entry ])
 end
end
```

pass in current dictionary

default value is empty list

and return new, updated, one

And Test It

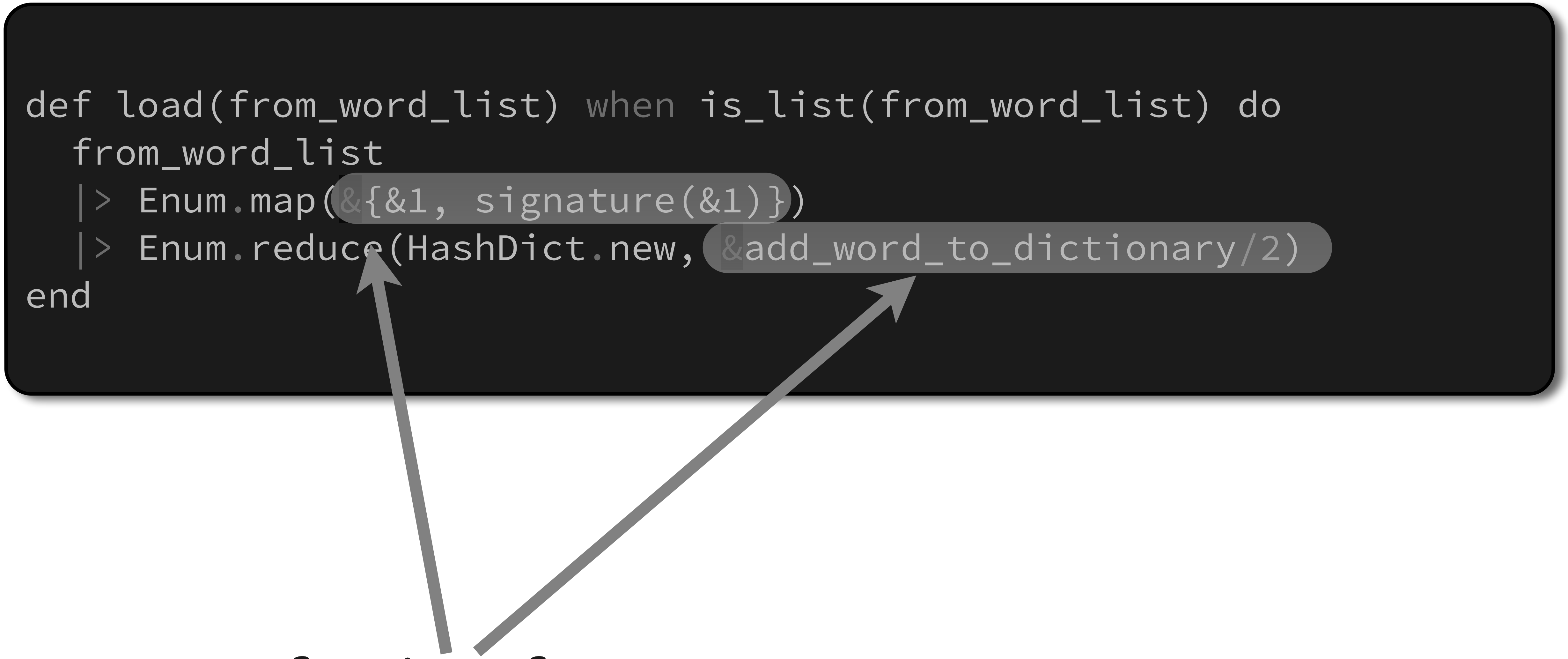
```
defmodule Anagrams.Dictionary do

  def signature(word) do
    word
    |> String.to_char_list!
    |> Enum.sort
    |> String.from_char_list!
  end

  def add_word_to_dictionary({word, signature}, dictionary) do
    entry = Dict.get(dictionary, signature, [])
    Dict.put(dictionary, signature, [ word | entry ])
  end
end
```

```
test "adding a word to a dictionary that doesn't have the signature creates a new entry" do
  dict = HashDict.new [ { "dgo", ["dog"] } ]
  dict = add_word_to_dictionary({"cat", "act"}, dict)
  assert Dict.has_key?(dict, "dgo")
  assert Dict.has_key?(dict, "act")
  assert Dict.get(dict, "act") == [ "cat" ]
end

test "adding a word to a dictionary that does have the signature adds to the entry" do
  dict = HashDict.new [ { "act", ["tac"] } ]
  dict = add_word_to_dictionary({"cat", "act"}, dict)
  assert Dict.has_key?(dict, "act")
  assert Dict.get(dict, "act") == [ "cat", "tac" ]
end
```

Dict vs. HashDict

- The Dict module defines the protocol, and provides default behaviours
- The HashDict module is a particular implementation (as is ListDict)
- In general, create using the specific implementation, but access using the general protocol

Load Words From List



```
def load(from_word_list) when is_list(from_word_list) do
  from_word_list
  |> Enum.map(&{&1, signature(&1)})
  |> Enum.reduce(HashDict.new, &add_word_to_dictionary/2)
end
```

function reference

Anonymous Functions

- fn params -> body end

```
iex> Enum.map 1..5, fn val -> val * 7 end

[7, 14, 21, 28, 35]

iex(2)> Enum.reduce 1..5, fn val, acc -> val + acc end

15

iex(3)> Enum.map [ "cat", "horse", "aardvark" ], fn word -> String.length(word) end

[3, 5, 8]
```

Anonymous Functions

- &(expression with &1, &2..)

```
iex> Enum.map 1..5, &(&1 * 7)

[7, 14, 21, 28, 35]

iex(2)> Enum.reduce 1..5, &(&1 + &2)

15

iex(3)> Enum.map [ "cat", "horse", "aardvark" ], &(String.length(&1))

[3, 5, 8]
```

Anonymous Functions

- &{ tuple with &1, &2.. }
  &[list with &1, &2..]

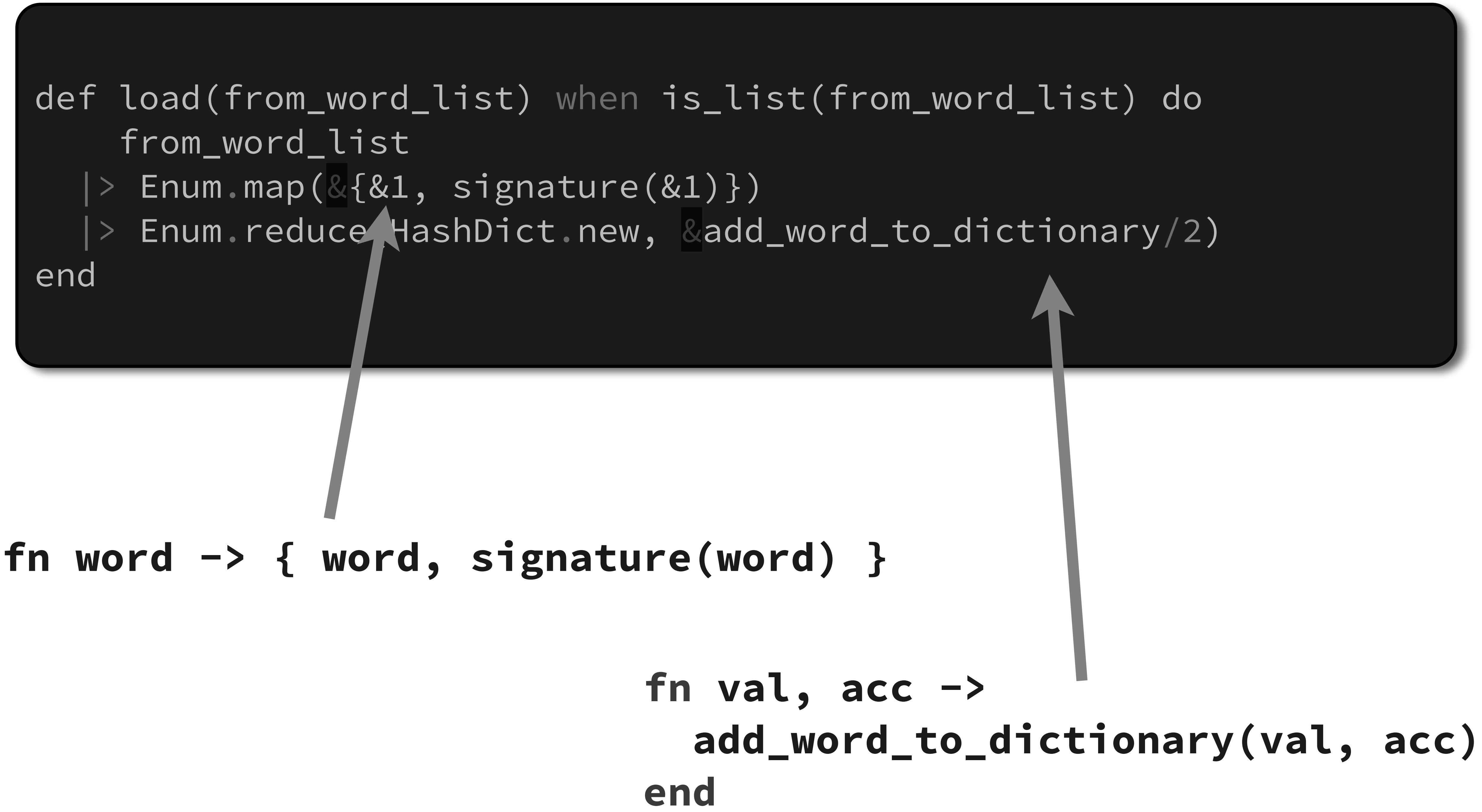
```
iex(2)> Enum.map 1..5, &{ :value, &1 }

[value: 1, value: 2, value: 3, value: 4, value: 5]


iex(3)> Enum.reduce 1..5, 0, &[ &1, &2 ]

[5, [4, [3, [2, [1, 0]]]]]
```

Anonymous Functions

- &func_name/arity

```
iex(3)> Enum.map [ "cat", "horse", "aardvark" ], &(String.length(&1))
[3, 5, 8]

iex(3)> Enum.map [ "cat", "horse", "aardvark" ], &String.length/1
[3, 5, 8]
```

function with arity “1”

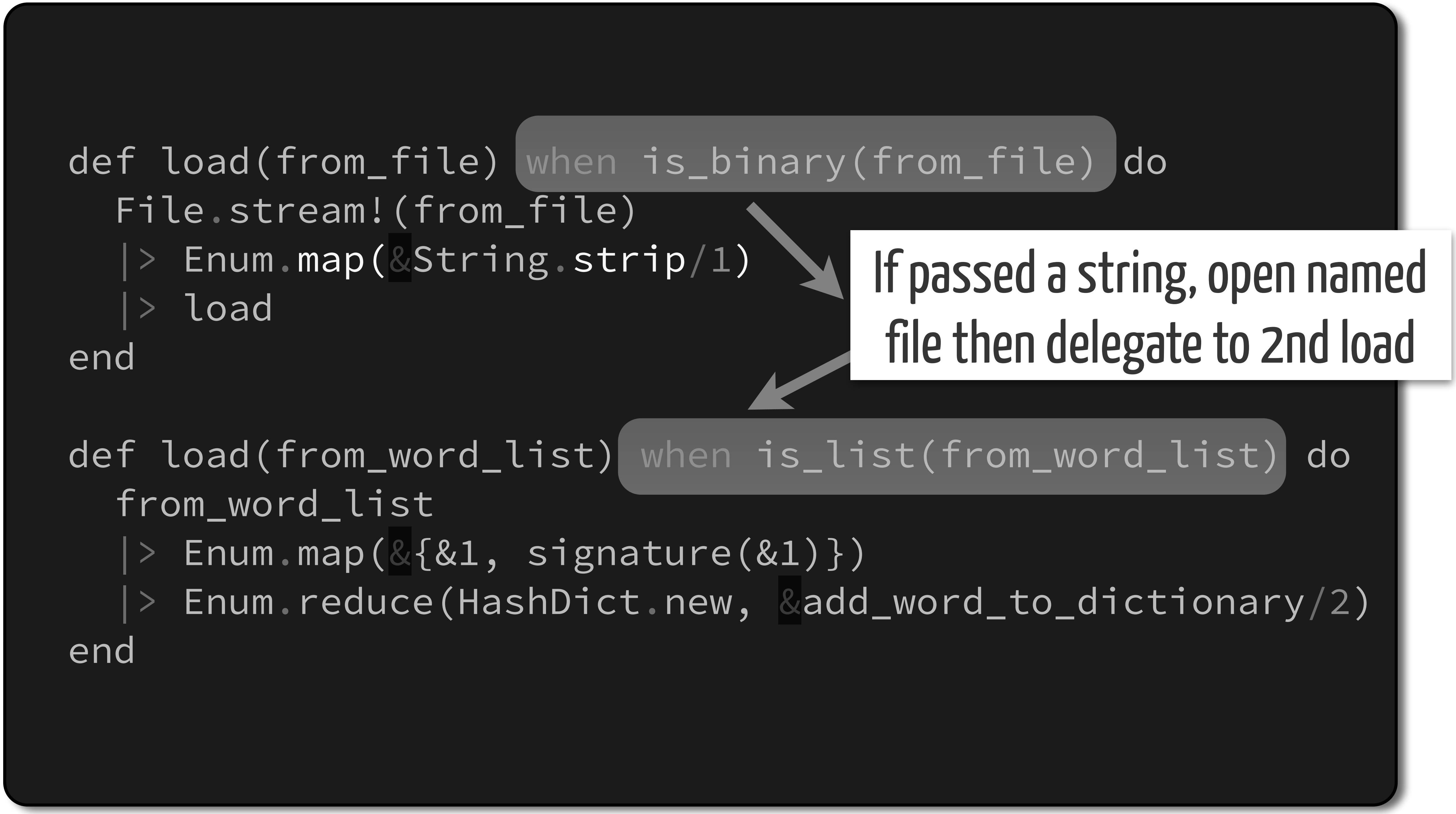


```
def load(from_word_list) when is_list(from_word_list) do
    from_word_list
  |> Enum.map(&{&1, signature(&1)})
  |> Enum.reduce(HashDict.new, &add_word_to_dictionary/2)
end
```

```
fn word -> { word, signature(word) }
```

```
fn val, acc ->
  add_word_to_dictionary(val, acc)
end
```

Load from File



```
def load(from_file) when is_binary(from_file) do
  File.stream!(from_file)
  |> Enum.map(&String.strip/1)
  |> load
end

def load(from_word_list) when is_list(from_word_list) do
  from_word_list
  |> Enum.map(&{&1, signature(&1)})
  |> Enum.reduce(HashDict.new, &add_word_to_dictionary/2)
end
```

If passed a string, open named file then delegate to 2nd load

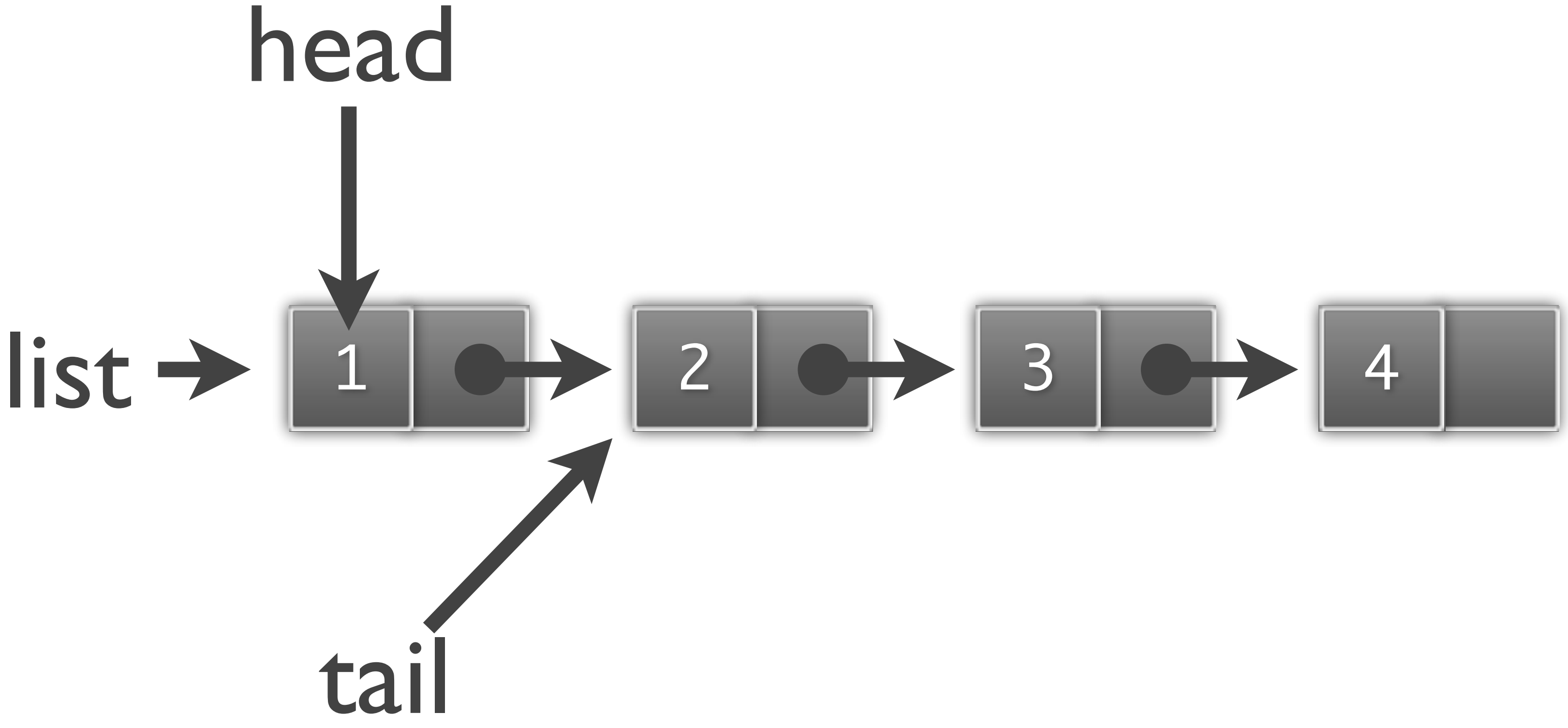
```
defmodule Anagrams.Dictionary do

  def load(from_file) when is_binary(from_file) do
    File.stream!(from_file)
    |> Enum.map(&String.strip/1)
    |> load
  end

  def load(from_word_list) when is_list(from_word_list) do
    from_word_list
    |> Enum.map(&{&1, signature(&1)})
    |> Enum.reduce(HashDict.new, &add_word_to_dictionary/2)
  end

  def lookup(dictionary, word) do
    Dict.get(dictionary, signature(word), "No anagrams found")
  end

  def signature(word) do
    word
    |> String.to_char_list!
    |> Enum.sort
    |> String.from_char_list!
  end

  def add_word_to_dictionary({word, signature}, dictionary) do
    entry = Dict.get(dictionary, signature, [])
    Dict.put(dictionary, signature, [ word | entry ])
  end

end
```

```
dave[anagrams] iex -S mix

Erlang R16B (erts-5.10.1) …
Interactive Elixir (0.11.2-dev)…

iex> import Anagrams.Dictionary
nil

iex> dict = load "/usr/share/dict/words"
#HashDict<[{"ehhloprtwy", ["helpworthy"]},
  {"eenoorvvw", ["overwoven"]},
  {"ccdeilnotuvy", ["conductively"]} …

iex> lookup dict, "retsina"
["stearin", "starnie", "stainer",
 "restain", "eranist", "asterin"]

iex> lookup dict, "aardvark"
["aardvark"]

iex> lookup dict, "xyzzy"
"No anagrams found"

^C ^C  (yes, it’s ugly)
```

Lists, Tuples, Binaries, CharLists, and Strings

Lists

[1, 2, 3, 4]

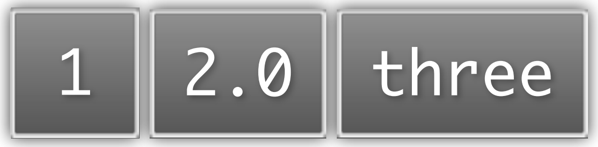


Lists

- Adding to front, and removing head is O(1)
- Everything else is O(n)

Tuples

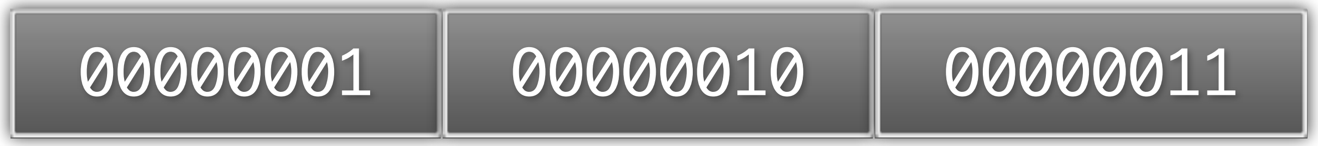
- { 1, 2.0, “three” }
- like a record structure



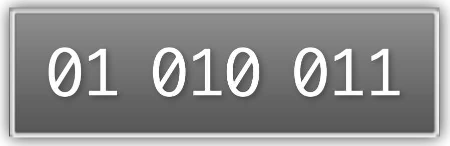
- size fixed at creation.
- access is O(1)

Binaries

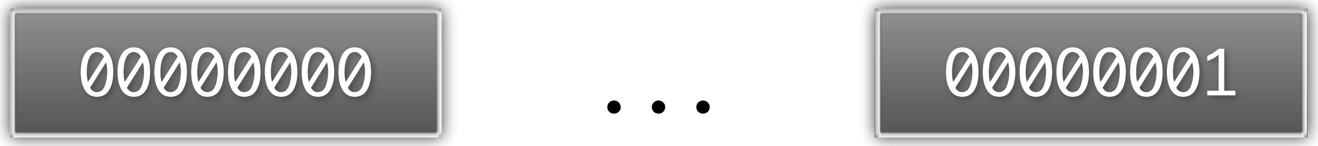
```
iex> b = << 1, 2, 3 >>
<<1, 2, 3>>
iex> size b
3
```



```
iex> b = << 1 :: 2, 2 :: 3, 3 :: 3 >>
"S"
```



```
iex> b = << 1 :: 32 >>
<<0, 0, 0, 1>>
```



Strings

```
iex> b = << 1 :: 2, 2 :: 3, 3 :: 3 >>
"S"
```

???

- A string is a binary containing UTF-8 codepoints
- Elixir displays a binary as characters iff its contents are valid code points

Strings

```
iex(11)> << 67, 65, 84 >>
"CAT"

iex(17)> <<74, 111, 115, 195, 169 >>
“José”

Iex(18)> "José" == <<74, 111, 115, 195, 169 >>
true
```

U+00E9   (0xc3a9)

Double-quoted string
generates a binary

Character Lists

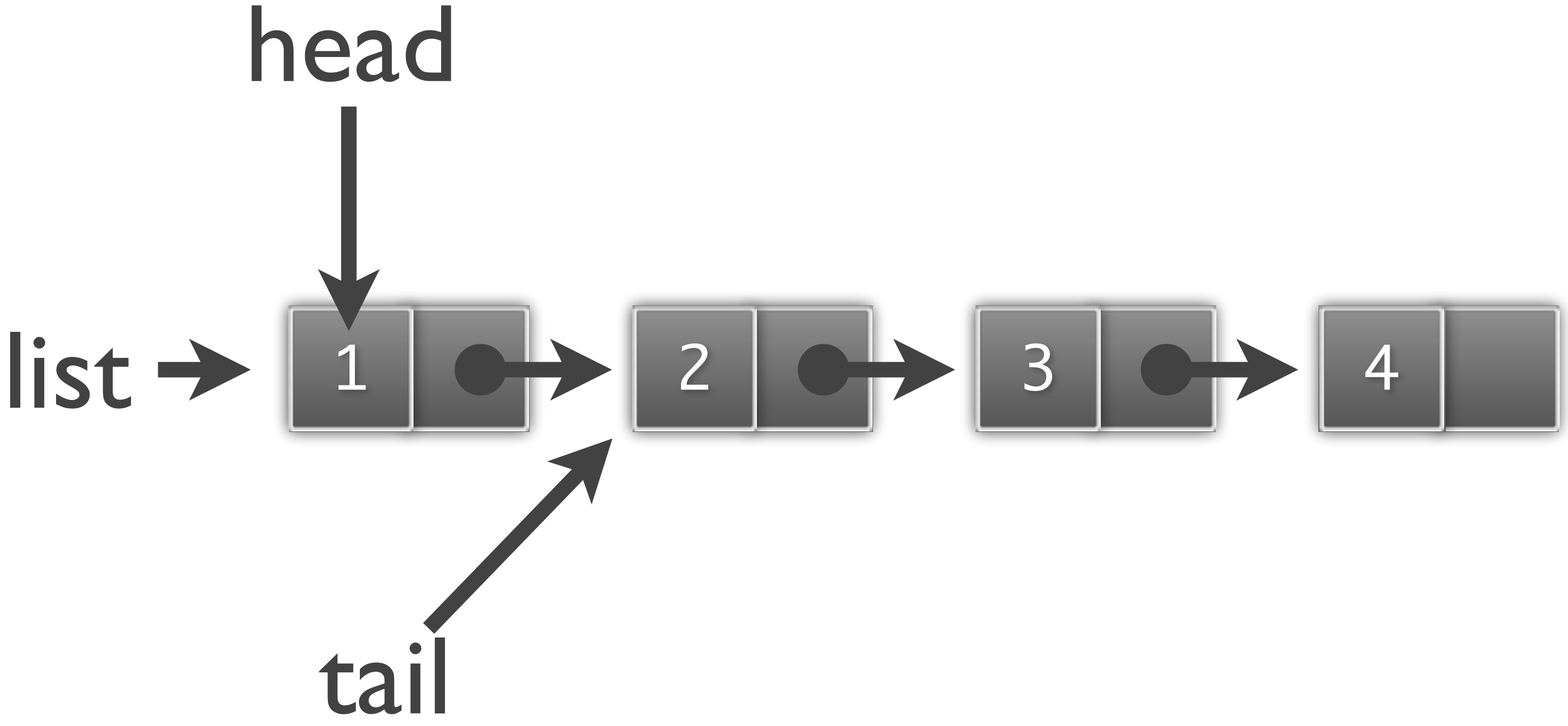
```
iex(11)> [ 67, 65, 84 ]
'CAT'

iex(17)> [ 74, 111, 115, 195, 169 ]
'José'
```

Single-quoted string
generates a charlist

Pattern Matching and Lists

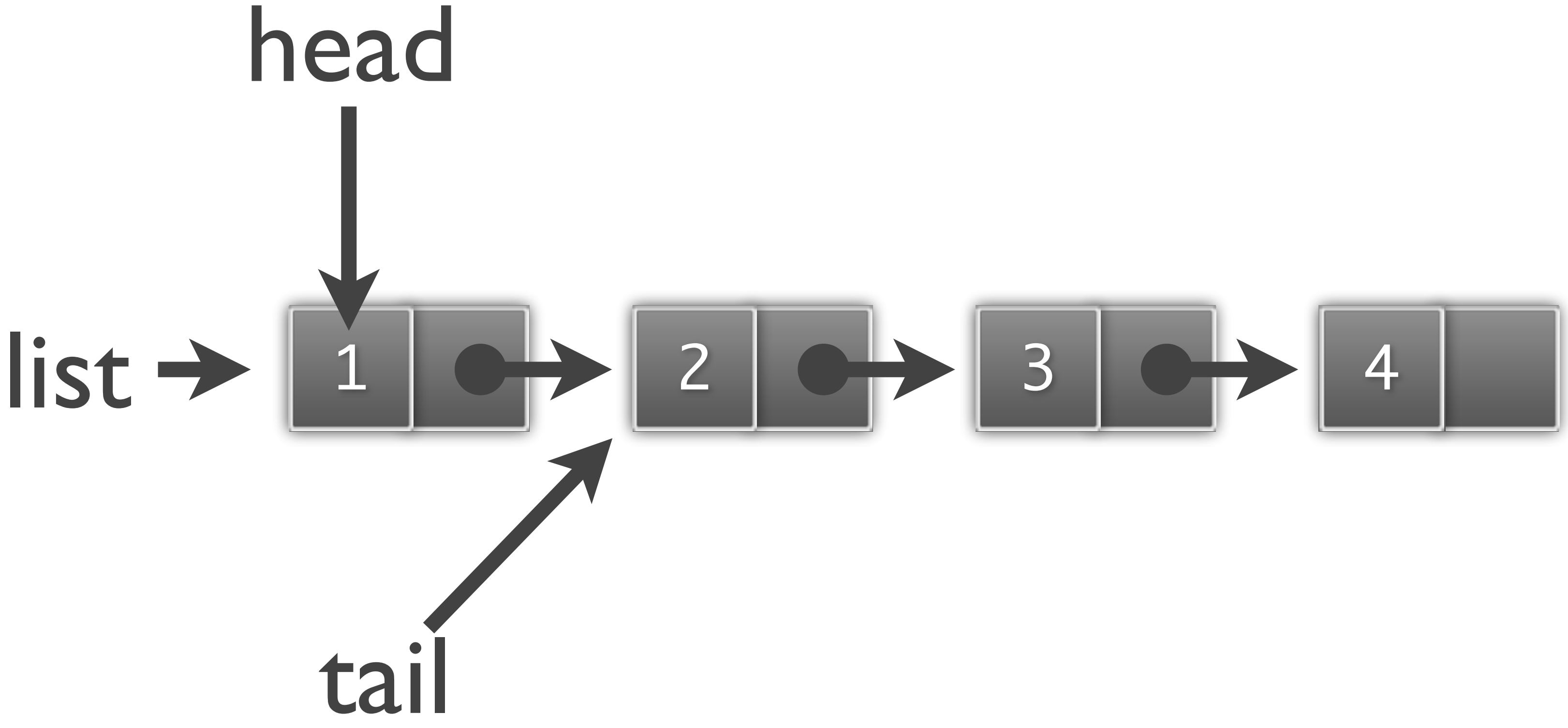
Lists

[1, 2, 3, 4]



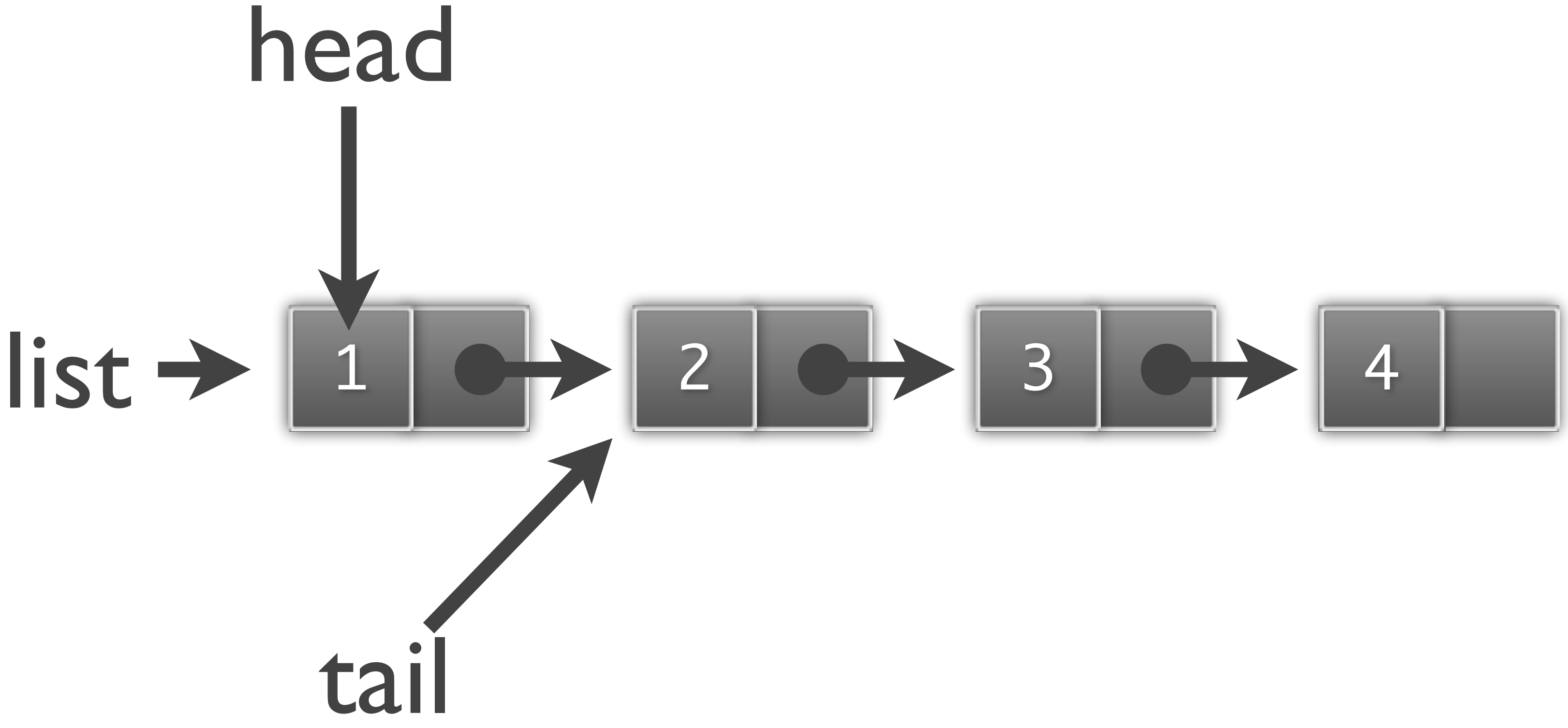
Lists

List with a head of 1 and a tail of [2, 3, 4] (a list)



Lists

[1 | [2, 3, 4]]



Lists

[1 | [2, 3, 4]]

[head | tail]

Lists and Pattern Matching

[head | tail]

```
iex> list = [ 1,2,3,4,5]
[1, 2, 3, 4, 5]

iex> [head | tail ] = list
[1, 2, 3, 4, 5]

iex> head
1

iex> tail
[2,3,4,5]
```

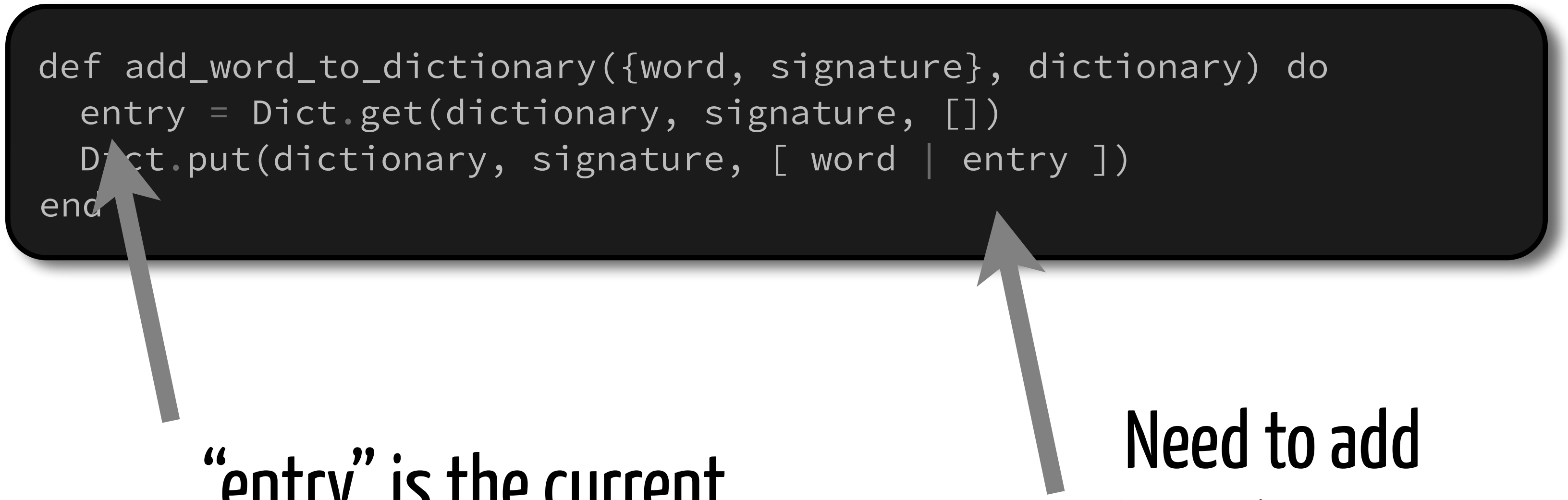
Lists and Pattern Matching

```
iex>  [ head | tail ] = [ 99 ]
'c'

iex> head
99

iex> tail
[]
```

```
defmodule Anagrams.Dictionary do

  def load(from_file) when is_binary(from_file) do
    File.stream!(from_file)
    |> Enum.map(&String.strip/1)
    |> load
  end

  def load(from_word_list) when is_list(from_word_list) do
    from_word_list
    |> Enum.map(&{&1, signature(&1)})
    |> Enum.reduce(HashDict.new, &add_word_to_dictionary/2)
  end

  def lookup(dictionary, word) do
    Dict.get(dictionary, signature(word), "No anagrams found")
  end

  def signature(word) do
    word
    |> String.to_char_list!
    |> Enum.sort
    |> String.from_char_list!
  end
```

```
def add_word_to_dictionary({word, signature}, dictionary) do
  entry = Dict.get(dictionary, signature, [])
  Dict.put(dictionary, signature, [ word | entry ])
end
```

```
end
```



```
def add_word_to_dictionary({word, signature}, dictionary) do
  entry = Dict.get(dictionary, signature, [])
  Dict.put(dictionary, signature, [ word | entry ])
end
```

“entry” is the current list of anagrams for a particular signature

(entry for signature “abt” was [“bat”])

Need to add “tab” (current word) to the list

[word | entry]

[“tab” | [“ bat”]]

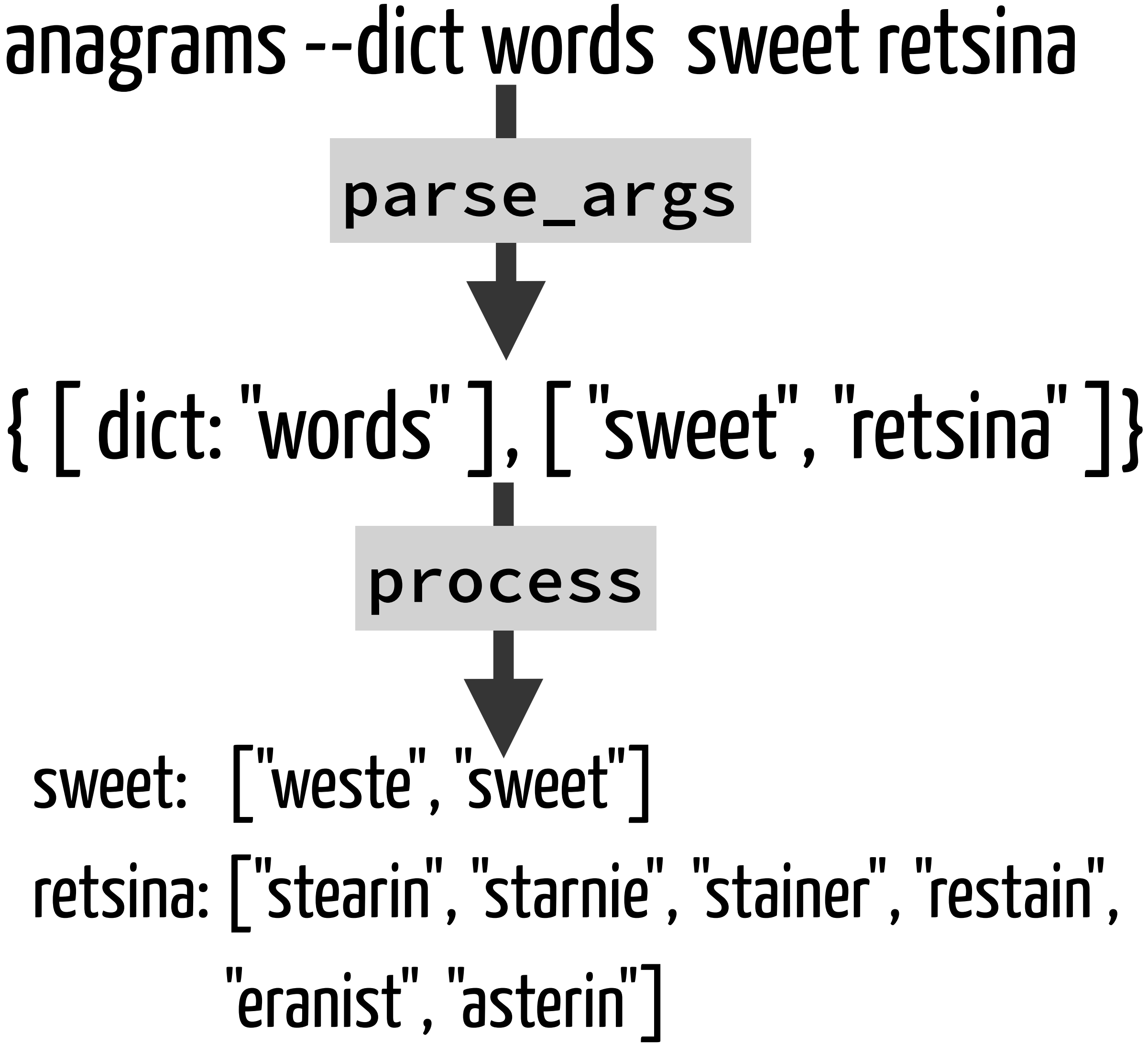
[“tab”, “ bat”]

Command Line Interface

The Idea

- We write all our interesting code “headless”
- We add interfaces when we need them
  - command line
  - web
  - ...
- Decouple from the start

```
dave[anagrams] ./anagrams --dict words sweet retsina
loading dictionary took 5.530942s
sweet: ["weste", "sweet"]
retsina: ["stearin", "starnie", "stainer", "restain", "eranist", "asterin"]
```



anagrams --dict words sweet retsina

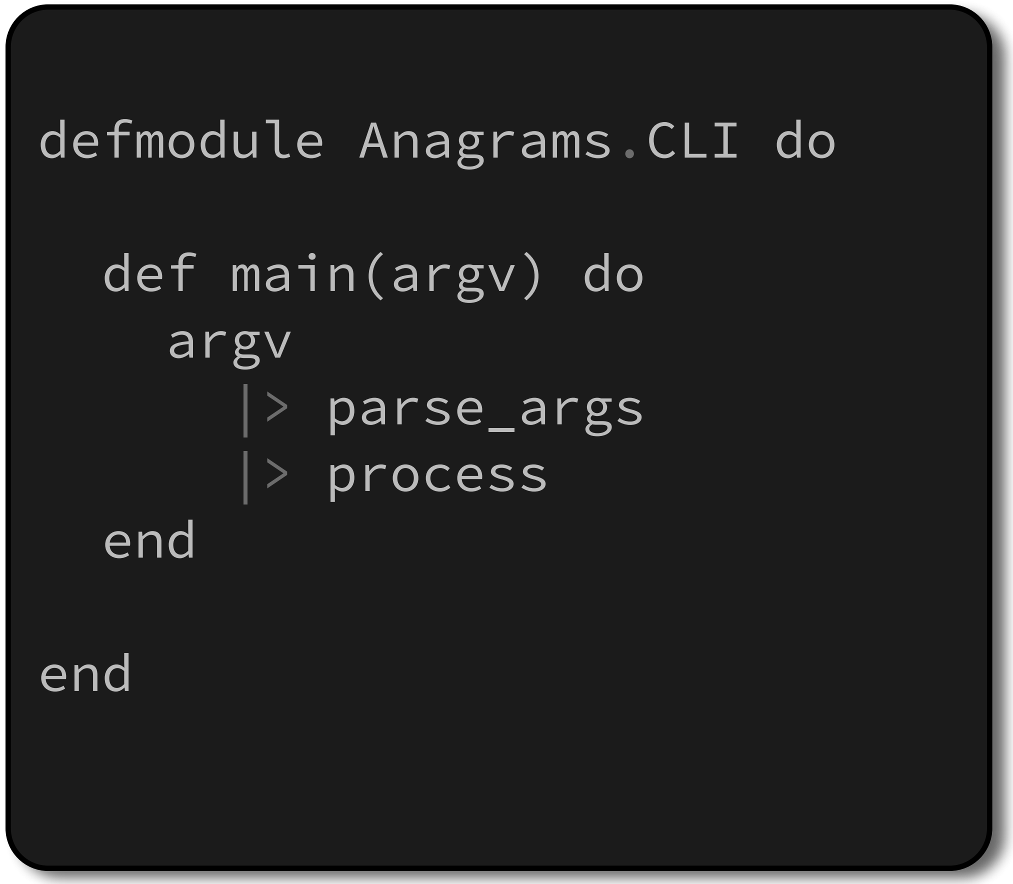
`parse_args`

{ [dict: "words"], ["sweet", "retsina"]}

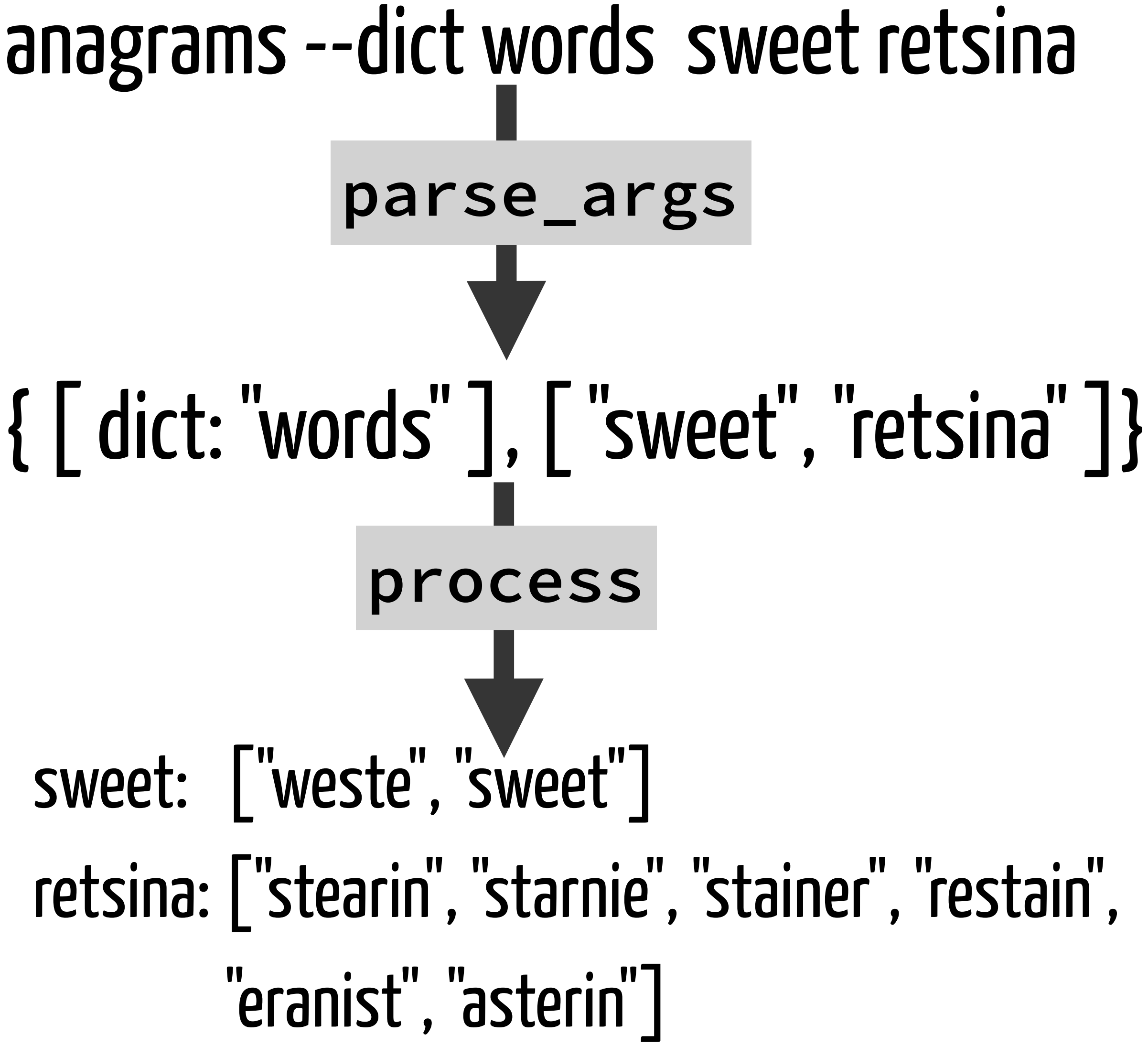
`process`

```
sweet:   ["weste", "sweet"]
retsina: ["stearin", "starnie", "stainer", "restain",
          "eranist", "asterin"]
```

Transformation!



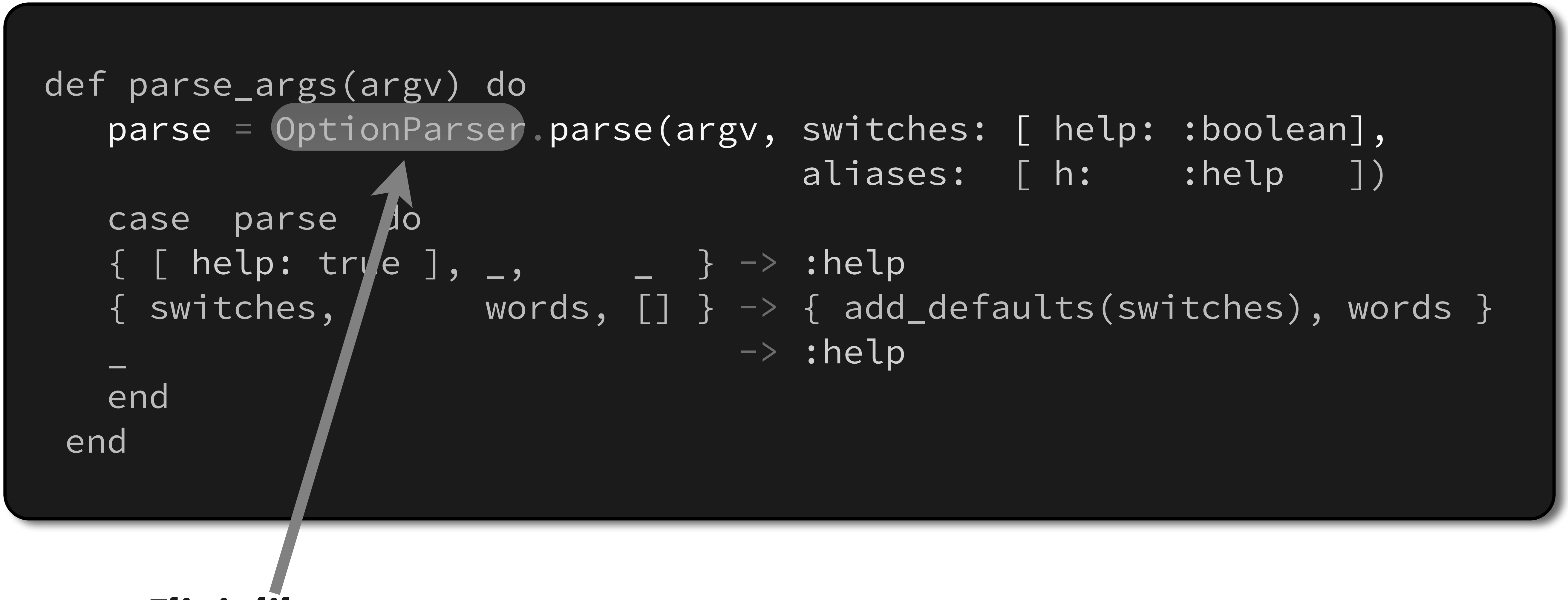
```
defmodule Anagrams.CLI do

  def main(argv) do
    argv
      |> parse_args
      |> process
  end

end
```



anagrams --dict words sweet retsina

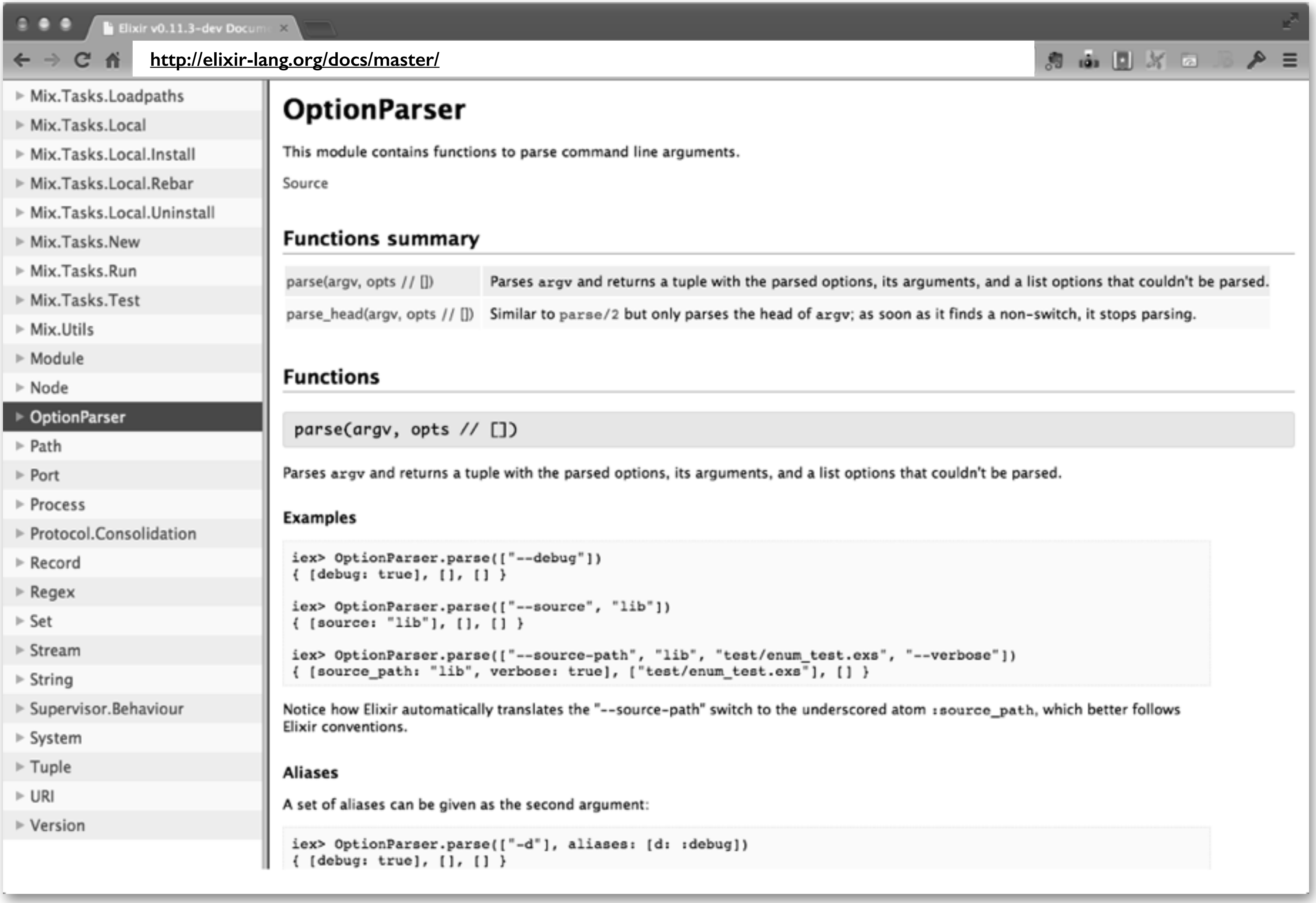
{ [dict: "words"], ["sweet", "retsina"]}

sweet: ["weste", "sweet"]
retsina: ["stearin", "starnie", "stainer", "restain",
          "eranist", "asterin"]



```
def parse_args(argv) do
   parse = OptionParser.parse(argv, switches: [ help: :boolean],
                                    aliases:  [ h:    :help   ])
   case  parse  do
   { [ help: true ], _,      _  } -> :help
   { switches,       words, [] } -> { add_defaults(switches), words }
   _                             -> :help
   end
 end
```

Elixir library



http://elixir-lang.org/docs/master/

OptionParser

This module contains functions to parse command line arguments.

Source

Functions summary

parse(argv, opts // [])	Parses `argv` and returns a tuple with the parsed options, its arguments, and a list options that couldn't be parsed.
parse_head(argv, opts // [])	Similar to `parse/2` but only parses the head of `argv`; as soon as it finds a non-switch, it stops parsing.

Functions

```
parse(argv, opts // [])
```

Parses `argv` and returns a tuple with the parsed options, its arguments, and a list options that couldn't be parsed.

Examples

```
iex> OptionParser.parse(["--debug"])
{ [debug: true], [], [] }

iex> OptionParser.parse(["--source", "lib"])
{ [source: "lib"], [], [] }

iex> OptionParser.parse(["--source-path", "lib", "test/enum_test.exs", "--verbose"])
{ [source_path: "lib", verbose: true], ["test/enum_test.exs"], [] }
```

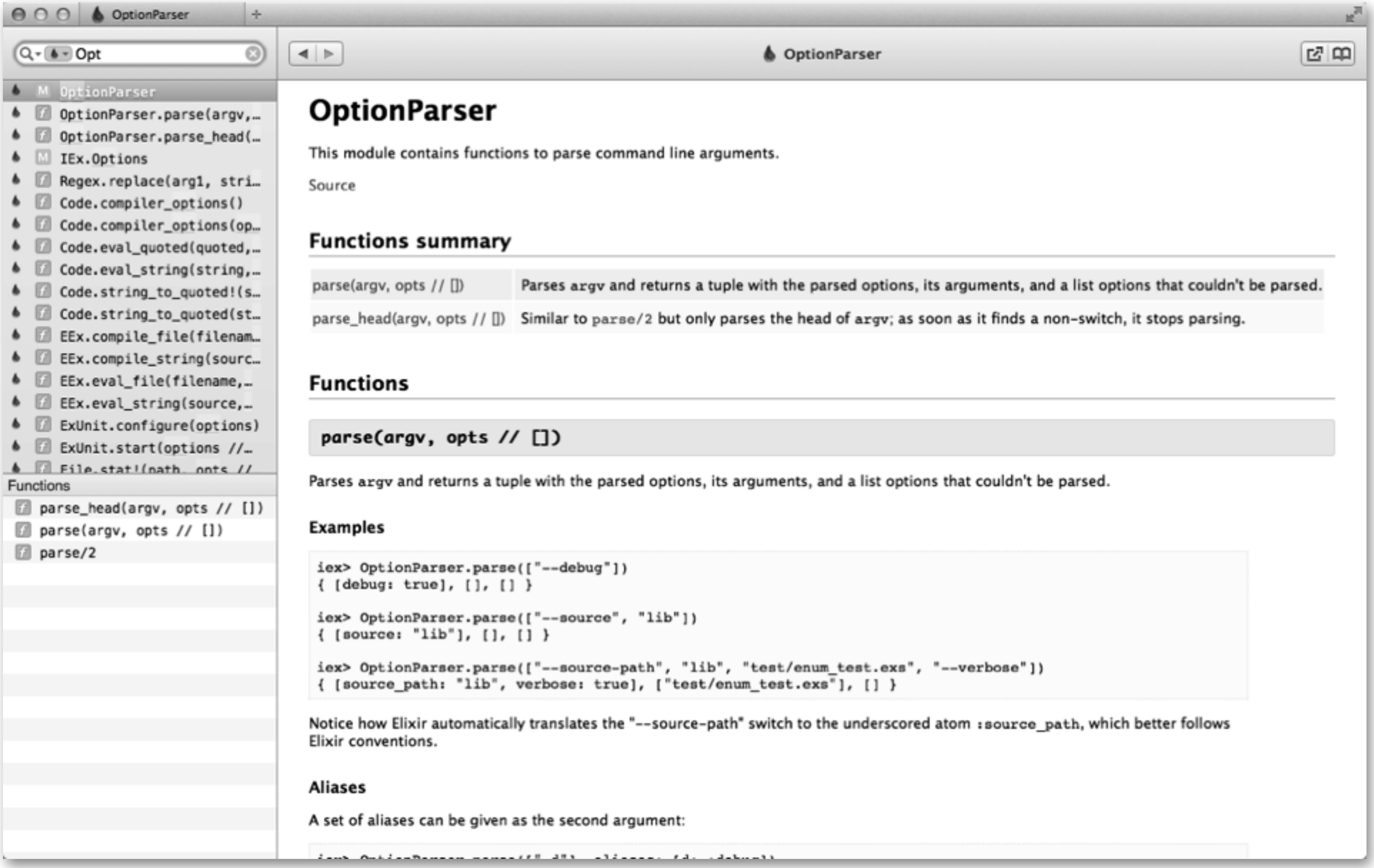
Notice how Elixir automatically translates the "--source-path" switch to the underscored atom `:source_path`, which better follows Elixir conventions.

Aliases

A set of aliases can be given as the second argument:

```
iex> OptionParser.parse(["-d"], aliases: [d: :debug])
{ [debug: true], [], [] }
```

Dash for OS X


OptionParser
Opt
OptionParser
OptionParser.parse(argv,…
OptionParser.parse_head(…
IEx.Options
Regex.replace(arg1, stri…
Code.compiler_options()
Code.compiler_options(op…
Code.eval_quoted(quoted,…
Code.eval_string(string,…
Code.string_to_quoted!(s…
Code.string_to_quoted(st…
EEx.compile_file(filenam…
EEx.compile_string(sourc…
EEx.eval_file(filename,…
EEx.eval_string(source,…
ExUnit.configure(options)
ExUnit.start(options //…
File.stat!(path, opts //
Functions
parse_head(argv, opts // [])
parse(argv, opts // [])
parse/2
OptionParser

OptionParser

This module contains functions to parse command line arguments.

Source

Functions summary

parse(argv, opts // [])	Parses `argv` and returns a tuple with the parsed options, its arguments, and a list options that couldn't be parsed.
parse_head(argv, opts // [])	Similar to `parse/2` but only parses the head of `argv`; as soon as it finds a non-switch, it stops parsing.

Functions

```
parse(argv, opts // [])
```

Parses `argv` and returns a tuple with the parsed options, its arguments, and a list options that couldn't be parsed.

Examples

```
iex> OptionParser.parse(["--debug"])
{ [debug: true], [], [] }

iex> OptionParser.parse(["--source", "lib"])
{ [source: "lib"], [], [] }

iex> OptionParser.parse(["--source-path", "lib", "test/enum_test.exs", "--verbose"])
{ [source_path: "lib", verbose: true], ["test/enum_test.exs"], [] }
```

Notice how Elixir automatically translates the "--source-path" switch to the underscored atom `:source_path`, which better follows Elixir conventions.

Aliases

A set of aliases can be given as the second argument:

```
iex> OptionParser.parse(["-d"], aliases: [d: :debug])
```

```
def parse_args(argv) do
   parse = OptionParser.parse(argv, switches: [ help: :boolean],
                                    aliases:  [ h:    :help   ])

   case  parse  do
   { [ help: true ], _,     _  } -> :help
   { switches,       words, [] } -> { add_defaults(switches), words }
   _                             -> :help
   end
 end
```

Atoms, keyword lists,
and funky parameters

```
iex> keywords = [ name: "dave", location: "Texas" ]
[name: "dave", location: "Texas"]

iex> inspect keywords
[name: "dave", location: "Texas"]

iex> inspect keywords, raw: true
[{:name, "dave"}, {:location, "Texas"}]
```

- Keyword list is a list of { :symbol, value } tuples
- In list context: atom: value

  is a shortcut for: { :atom, value }

```
iex> list = [ {:name, "Andy"}, {:location, "Raleigh"} ]
[name: "Andy", location: "Raleigh"]

iex> inspect list, raw: true
[{:name, "Andy"}, {:location, "Raleigh"}]
```

- iex pretty-prints lists of {:atom, value} as [atom: value, ...]
- use inspect thing, raw: true to see real form

- Pass a keyword list to a function

```
iex> inspect([feet: 6.1, meter: 1.86])
"[feet: 6.1, meter: 1.86]"
```

- If it is the last parameter, can leave off the [ ]

```
iex> inspect( feet: 6.1, meter: 1.86 )
"[feet: 6.1, meter: 1.86]"
```

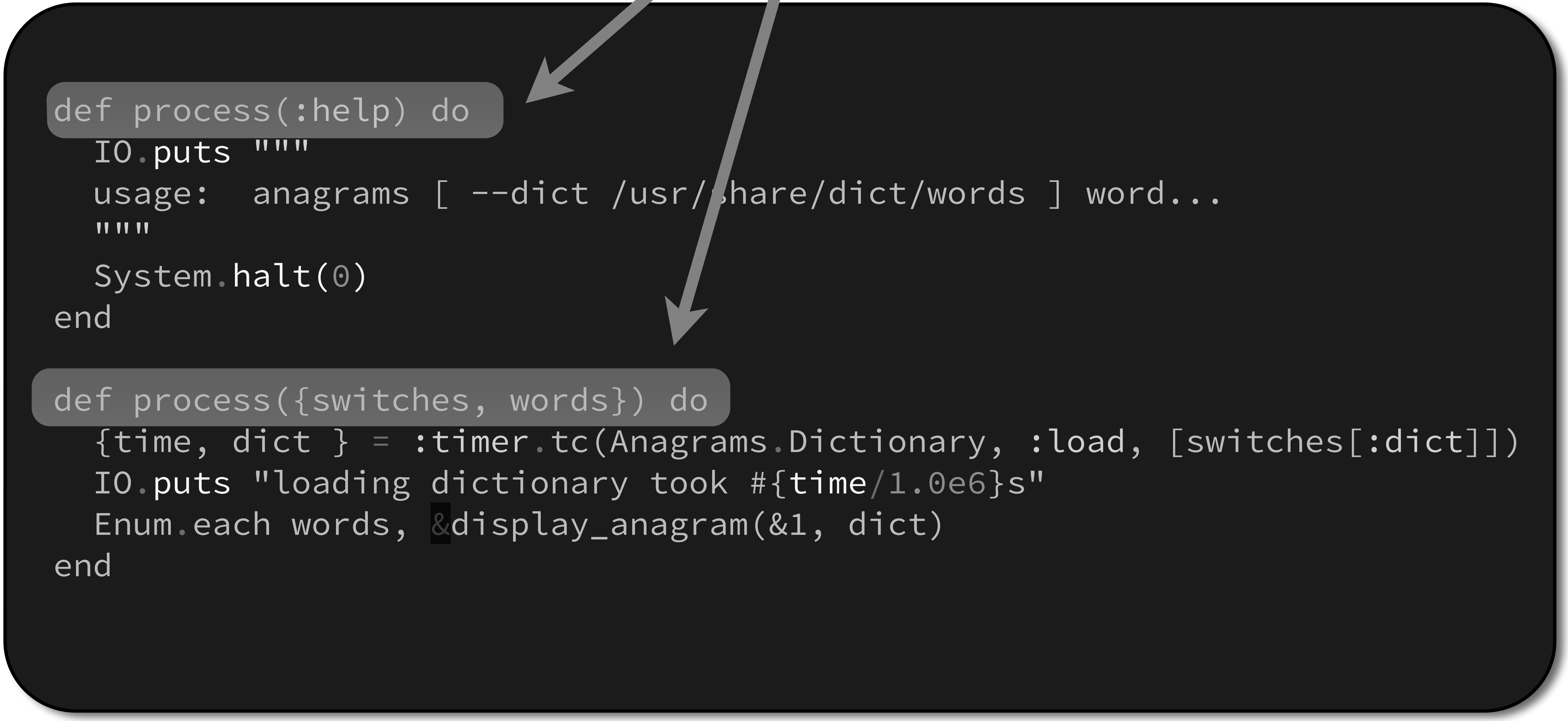
- And parentheses are optional on function calls

```
iex> inspect feet: 6.1, meter: 1.86
"[feet: 6.1, meter: 1.86]"
```

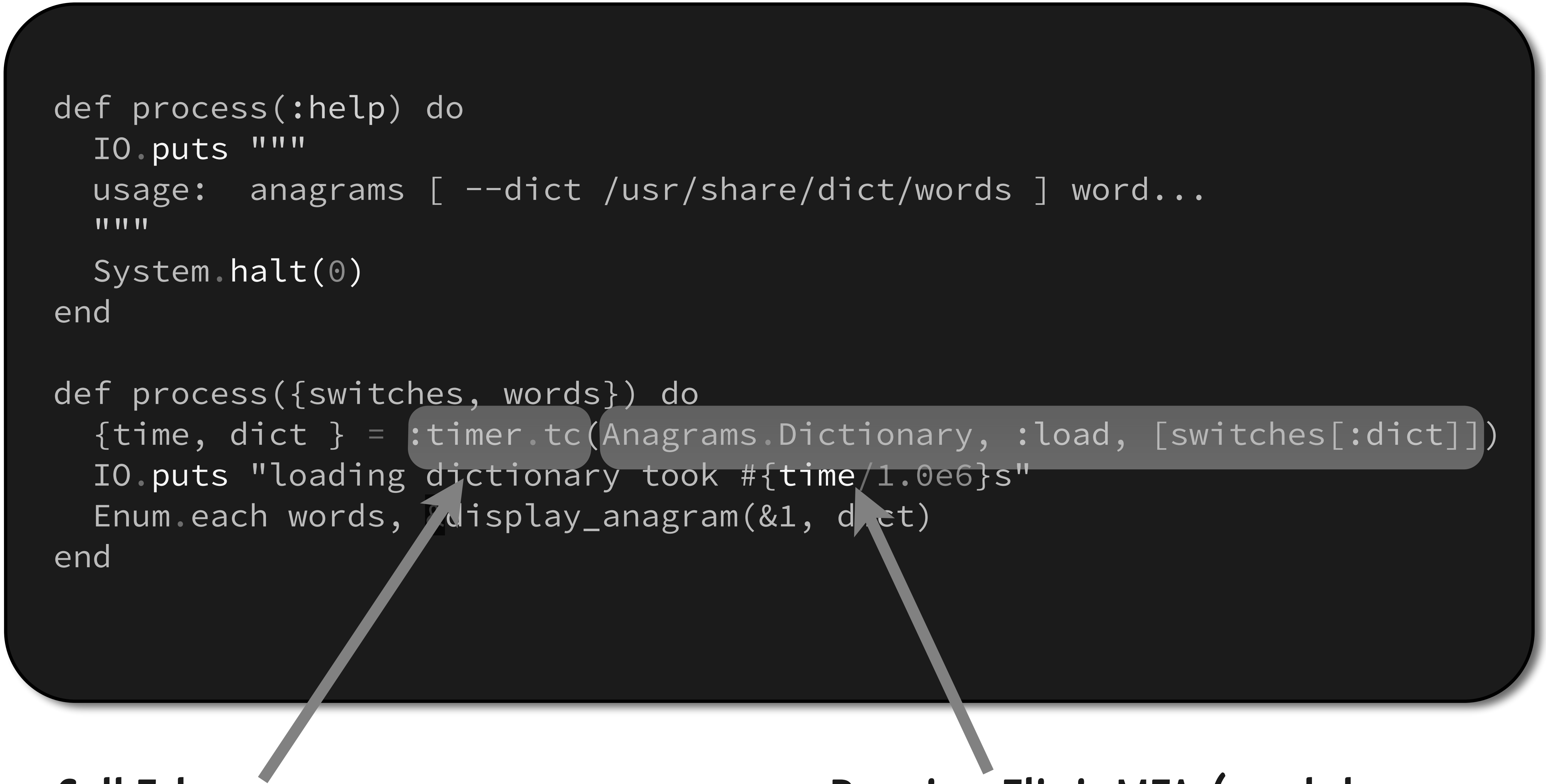
```
def main(argv) do
  argv
  |> parse_args
  |> process
end

def parse_args(argv) do
  parse = OptionParser.parse(argv, switches: [ help: :boolean],
                                   aliases:  [ h:    :help   ])
  case  parse  do
  { [ help: true ], _,      _  } -> :help
  { switches,       words, [] } -> { add_defaults(switches), words }
  _                              -> :help
  end
end
```

Return value of parse_args.
Gets passed to process()

Pattern match to select function to run



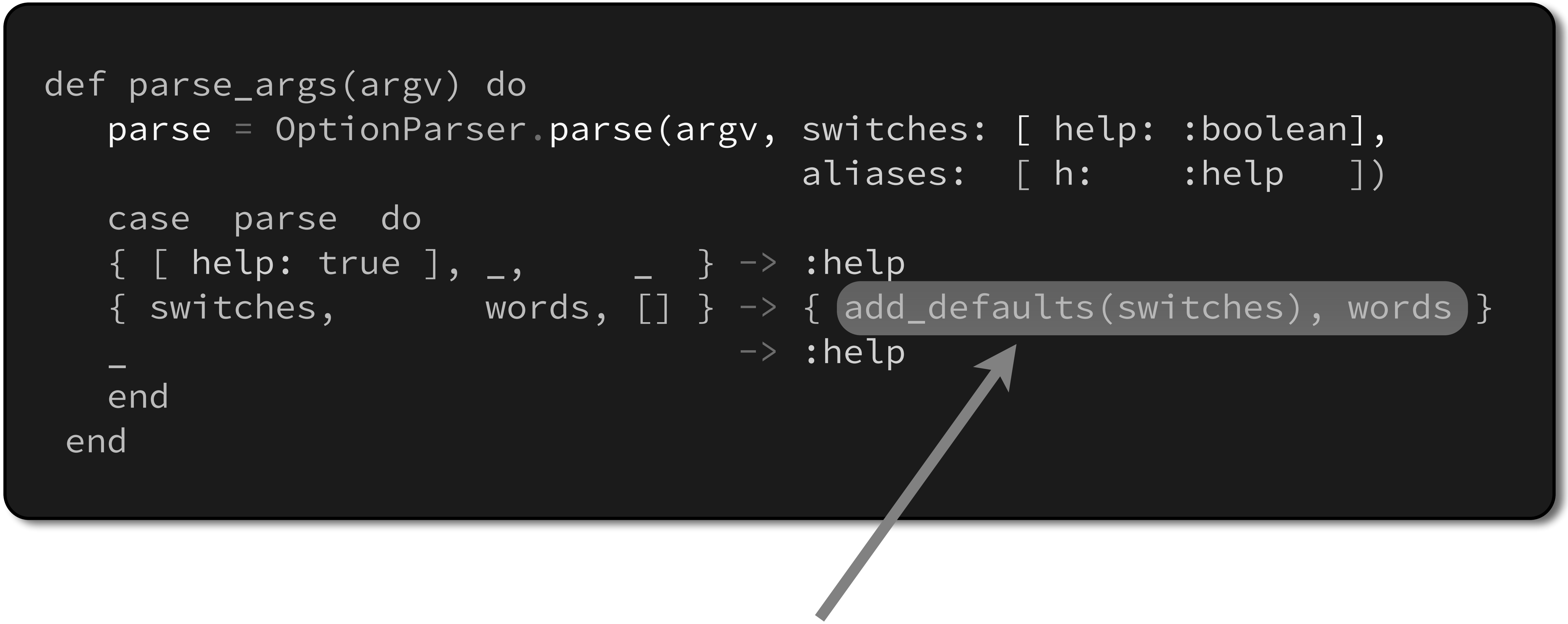
```
def process(:help) do
  IO.puts """
  usage:  anagrams [ --dict /usr/share/dict/words ] word...
  """
  System.halt(0)
end

def process({switches, words}) do
  {time, dict } = :timer.tc(Anagrams.Dictionary, :load, [switches[:dict]])
  IO.puts "loading dictionary took #{time/1.0e6}s"
  Enum.each words, &display_anagram(&1, dict)
end
```



```
def process(:help) do
  IO.puts """
  usage:  anagrams [ --dict /usr/share/dict/words ] word...
  """
  System.halt(0)
end

def process({switches, words}) do
  {time, dict } = :timer.tc(Anagrams.Dictionary, :load, [switches[:dict]])
  IO.puts "loading dictionary took #{time/1.0e6}s"
  Enum.each words, &display_anagram(&1, dict)
end
```

Call Erlang
library function

Passing Elixir MFA (module,
function name, arguments)



```
def parse_args(argv) do
   parse = OptionParser.parse(argv, switches: [ help: :boolean],
                                    aliases:  [ h:    :help   ])
   case  parse  do
   { [ help: true ], _,      _  } -> :help
   { switches,       words, [] } -> { add_defaults(switches), words }
   _                              -> :help
   end
 end
```

Supply defaults for any
options not supplied by user

```
defmodule Anagrams.CLI do

  @default_switches [ dict: "/usr/share/dict/words" ]

  def add_defaults(switches) do
    Dict.merge(@default_switches, switches)
  end

end
```

Module attribute.
(Same as Erlang.
A bit like Ruby constants)

```
defmodule SomeModule do
  @value 123
  def func1, do: IO.puts @value
  @value "cat"
  def func2, do: IO.puts @value
end
SomeModule.func1    #=> 123
SomeModule.func2    #=> "cat"
```

By default, current value used.

Can be accumulated instead.

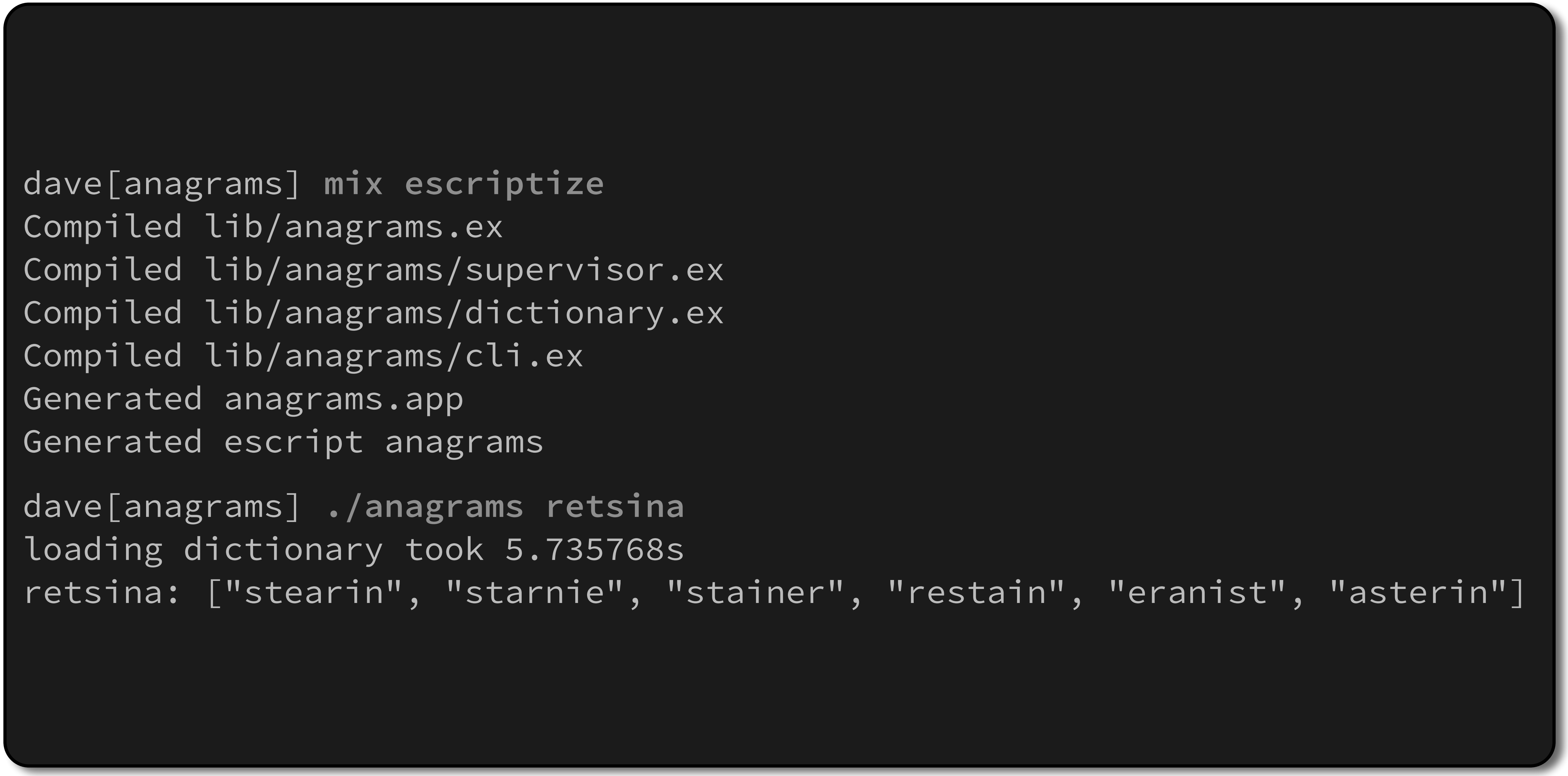
Make an Executable

```
defmodule Anagrams.Mixfile do
  use Mix.Project

  def project do
    [ app: :anagrams,
      version: "0.0.1",
      elixir: "~> 0.11.2-dev",
      escript_main_module: Anagrams.CLI,
      deps: deps ]
  end

  # Configuration for the OTP application
  def application do
    [mod: { Anagrams, [] }]
  end

  defp deps do
    []
  end
end
```

mix.exs defaults project attributes, applications, and dependencies

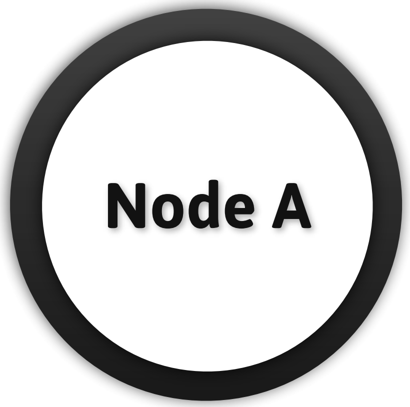
```
defmodule Anagrams.Mixfile do
  use Mix.Project

  def project do
    [ app: :anagrams,
      version: "0.0.1",
      elixir: "~> 0.11.2-dev",
      escript_main_module: Anagrams.CLI,
      deps: deps ]
  end

  # Configuration for the OTP application
  def application do
    [mod: { Anagrams, [] }]
  end

  defp deps do
    []
  end
end
```

Specify “entry point”. Program is run by calling main() function in this module



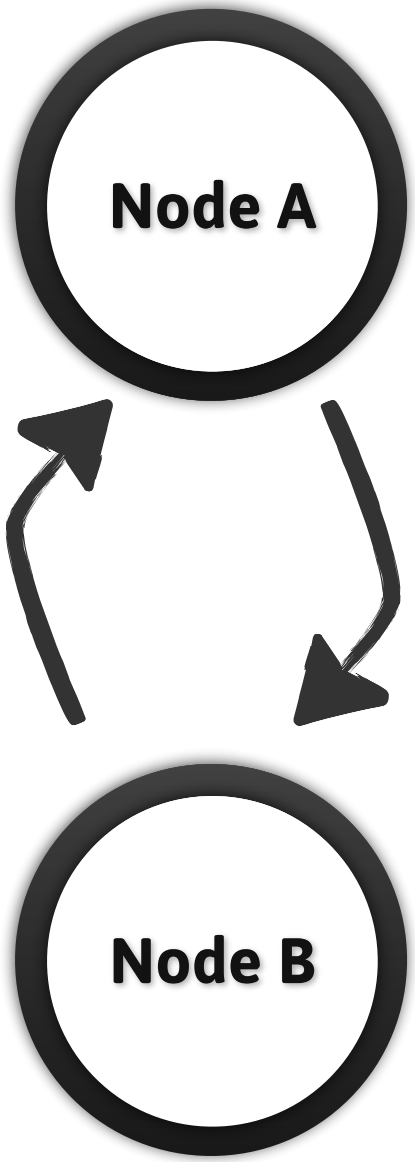
```
dave[anagrams] mix escriptize
Compiled lib/anagrams.ex
Compiled lib/anagrams/supervisor.ex
Compiled lib/anagrams/dictionary.ex
Compiled lib/anagrams/cli.ex
Generated anagrams.app
Generated escript anagrams

dave[anagrams] ./anagrams retsina
loading dictionary took 5.735768s
retsina: ["stearin", "starnie", "stainer", "restain", "eranist", "asterin"]
```

Make a Server

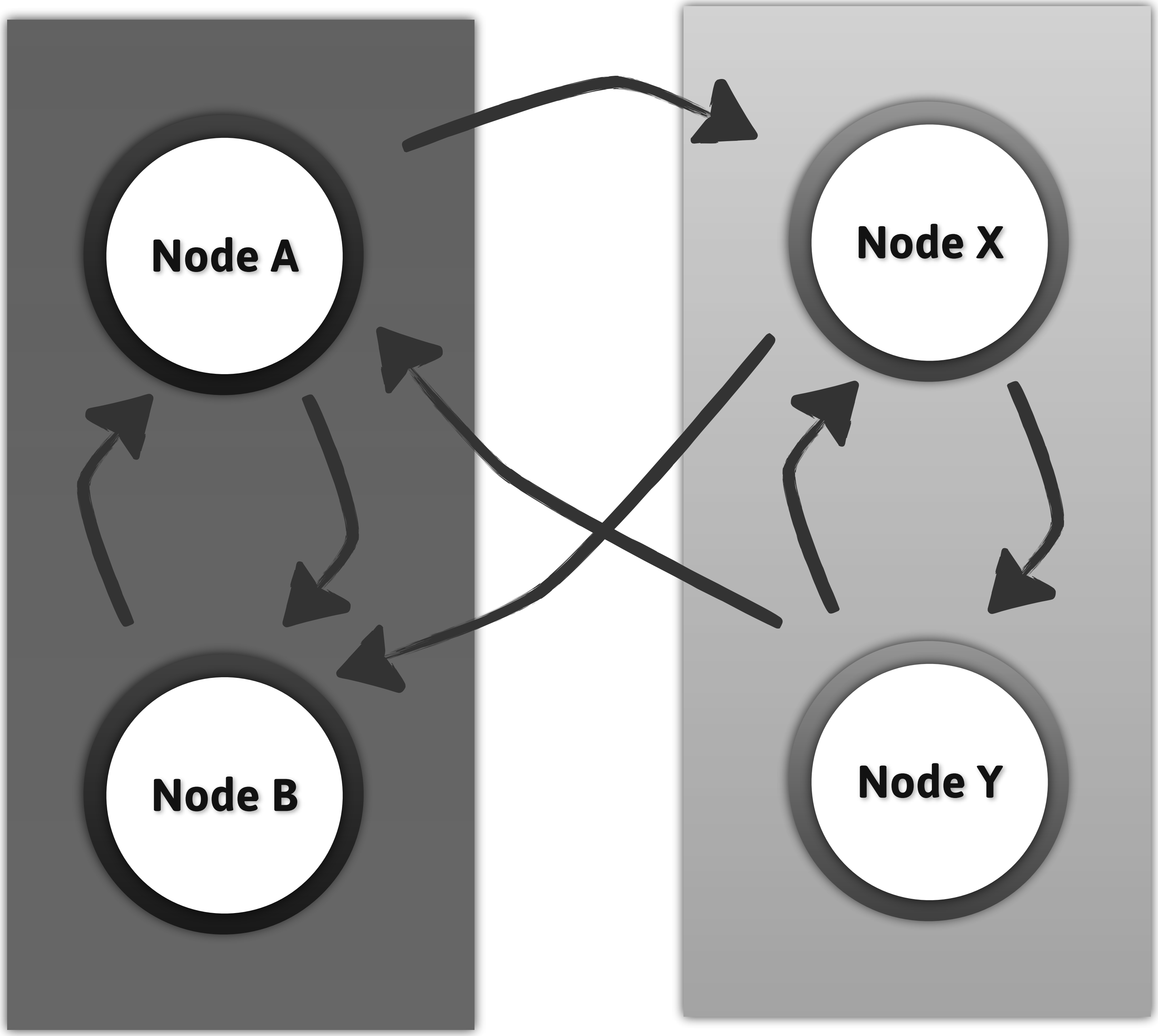


Node A

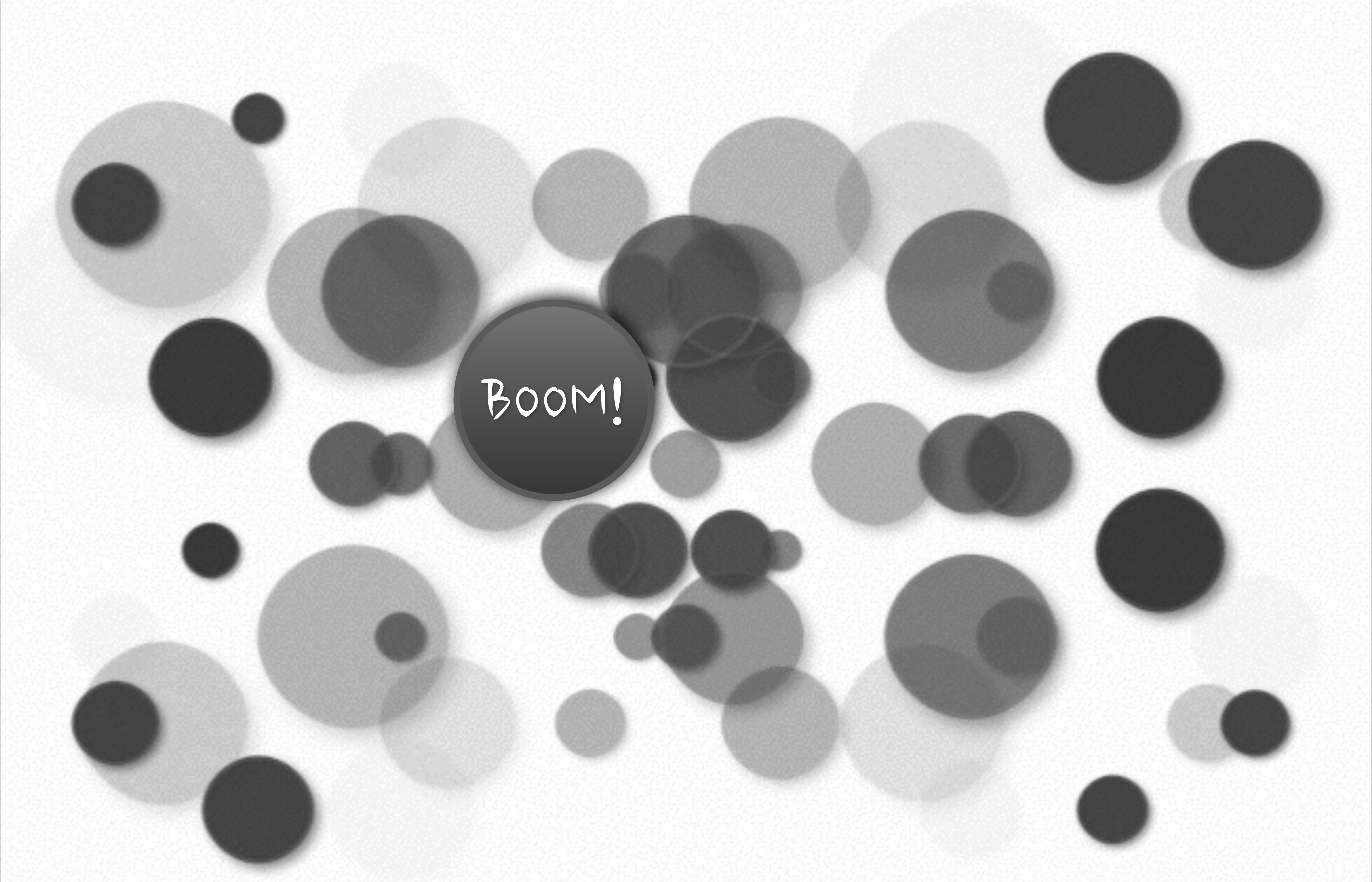
- Instance of the Erlang VM
- Can run on multiple processors/cores
- Local



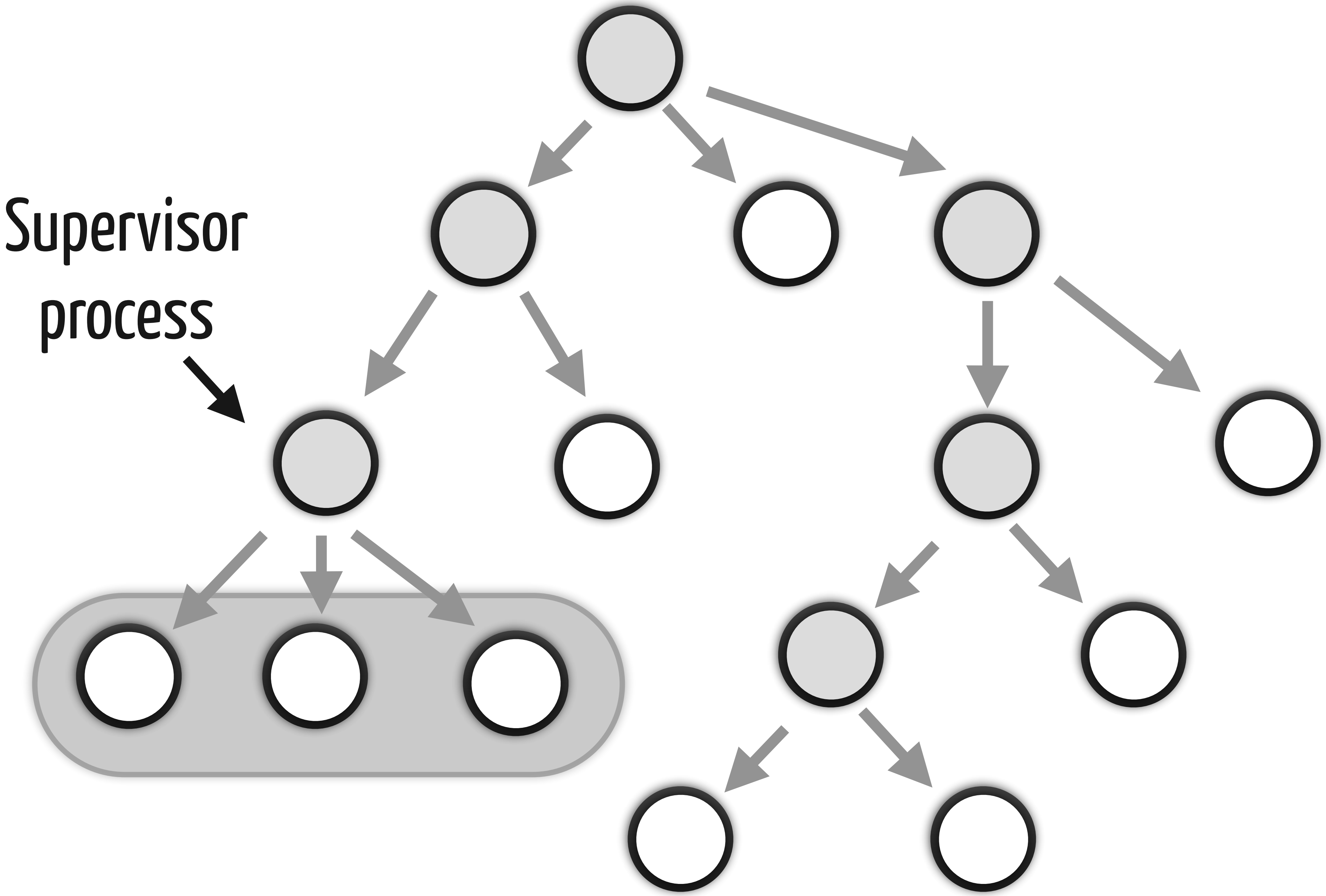
- Spawn and monitor processes on any node
- Message passing and monitoring transparent

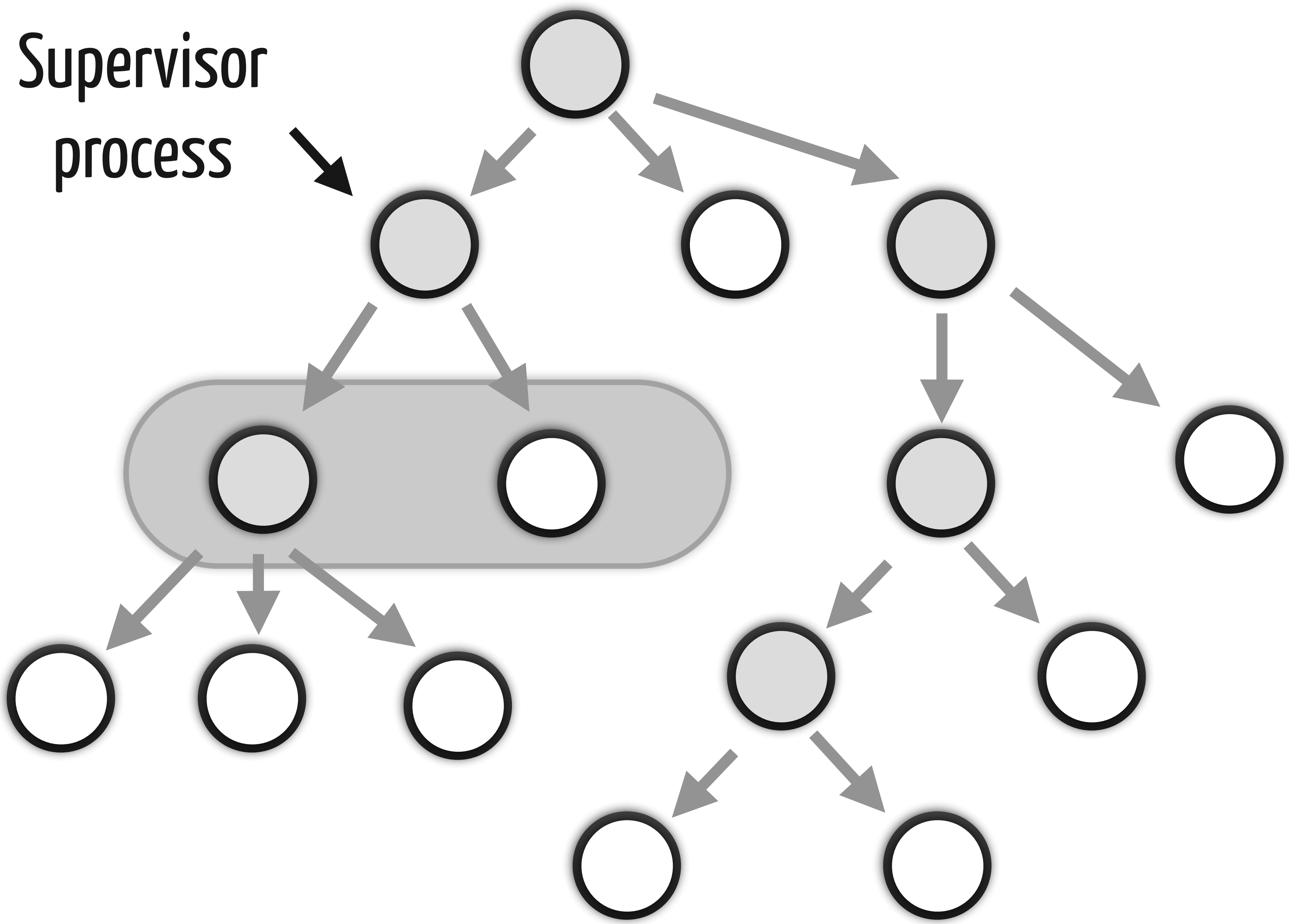


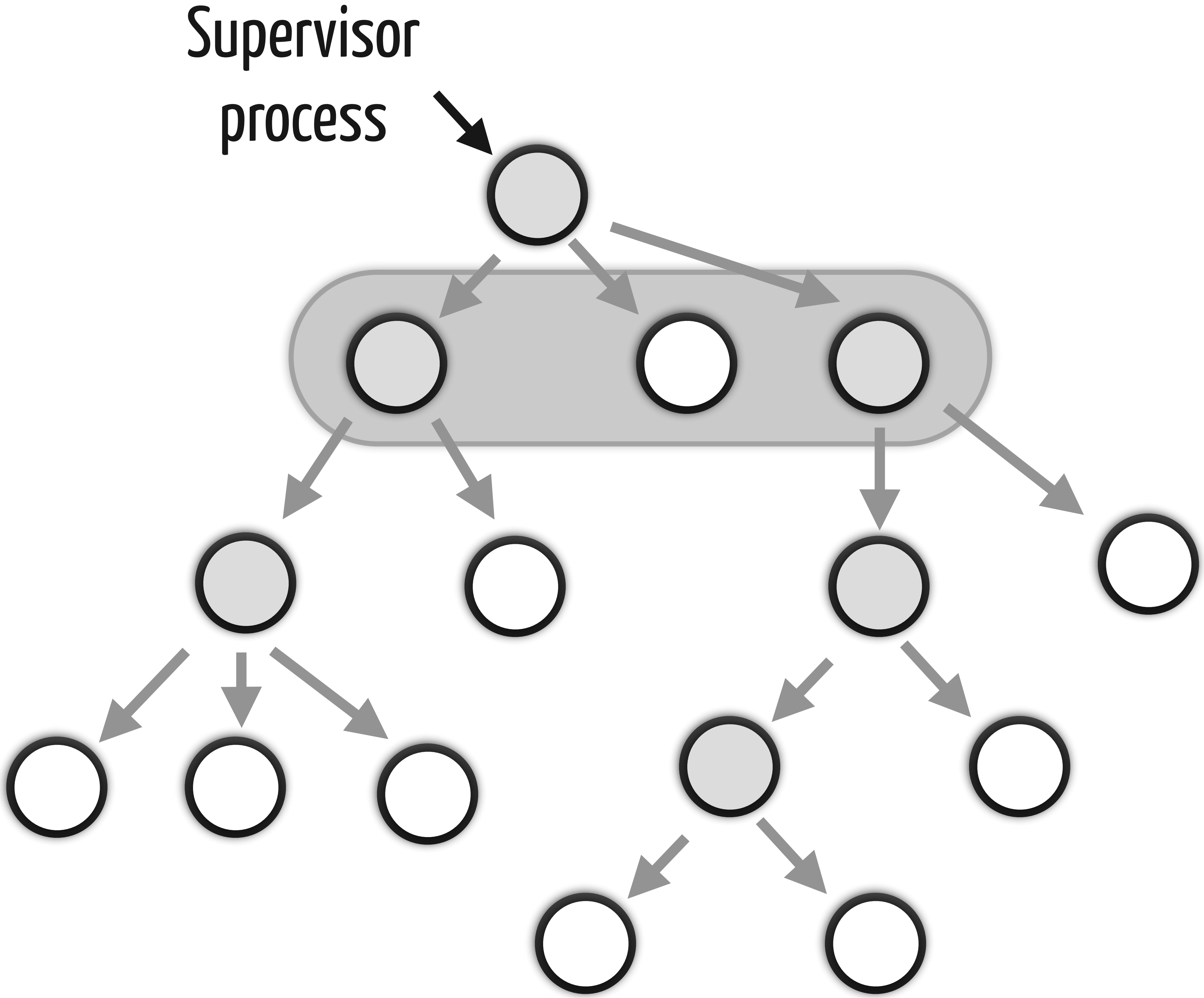
- Between machines
- Across networks


Boom!

Supervision Tree



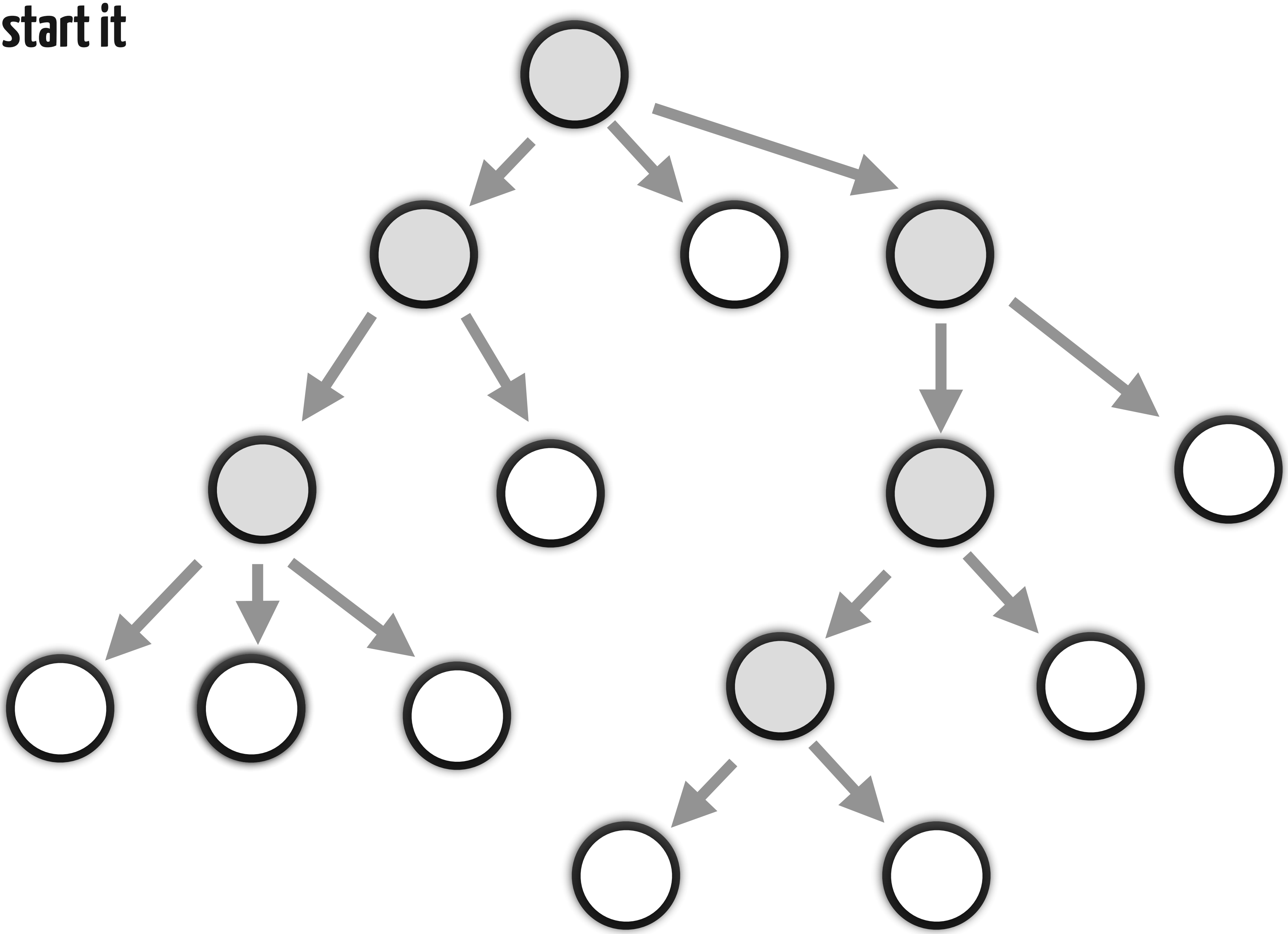

Supervisor
process



Supervisor
process

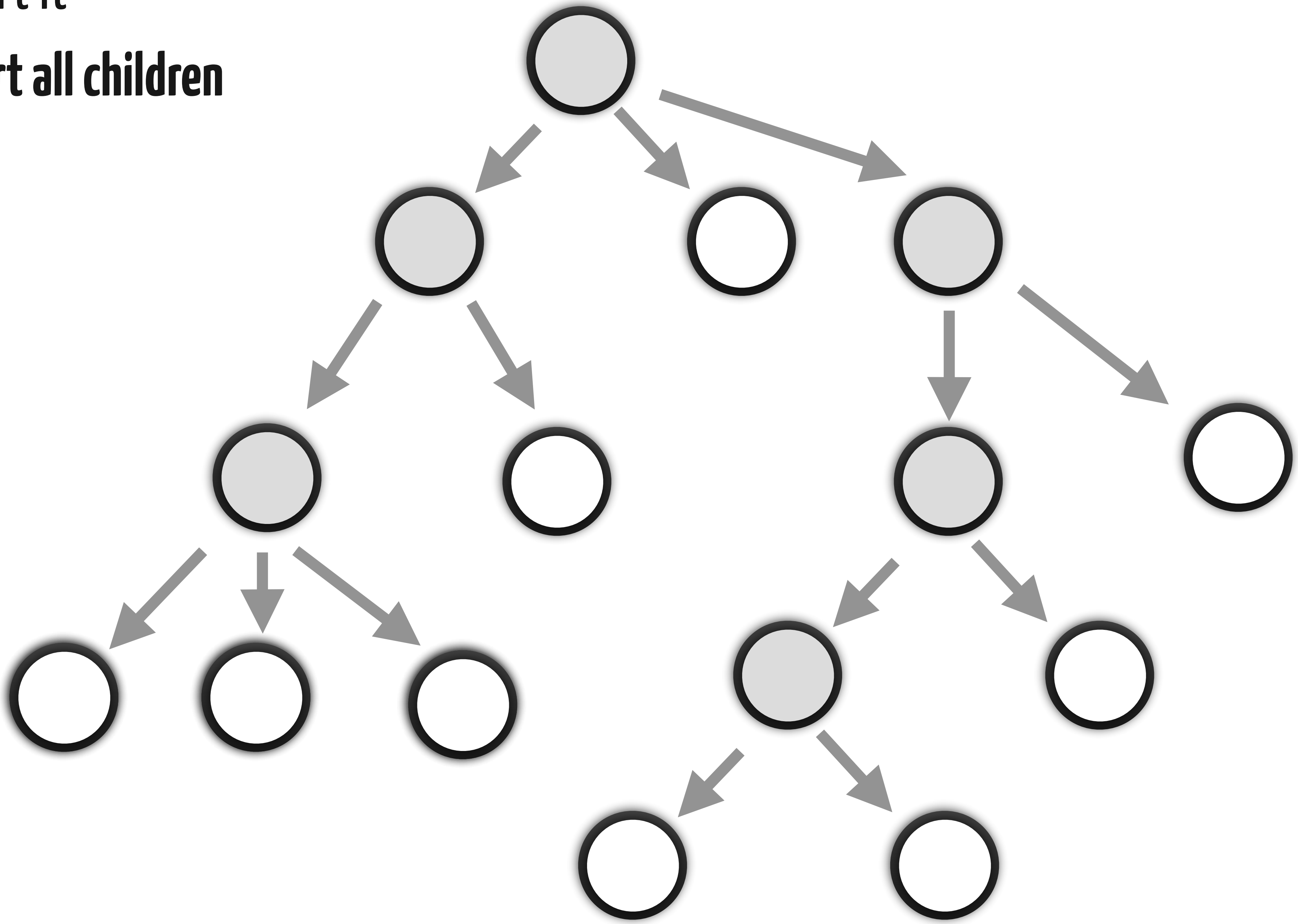
When child dies:

- **restart it**



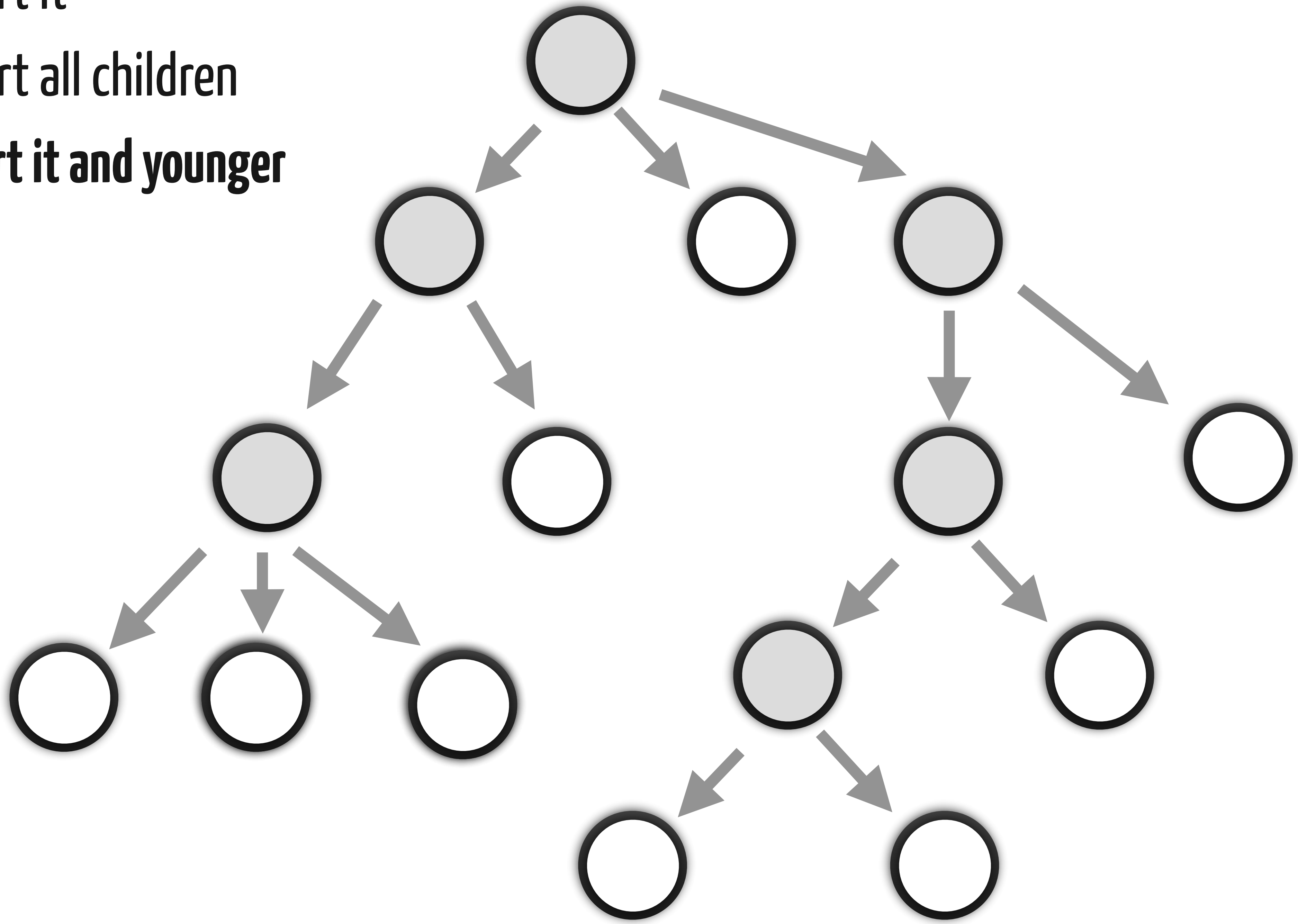
When child dies:

- restart it
- **restart all children**



When child dies:

- restart it
- restart all children
- **restart it and younger**

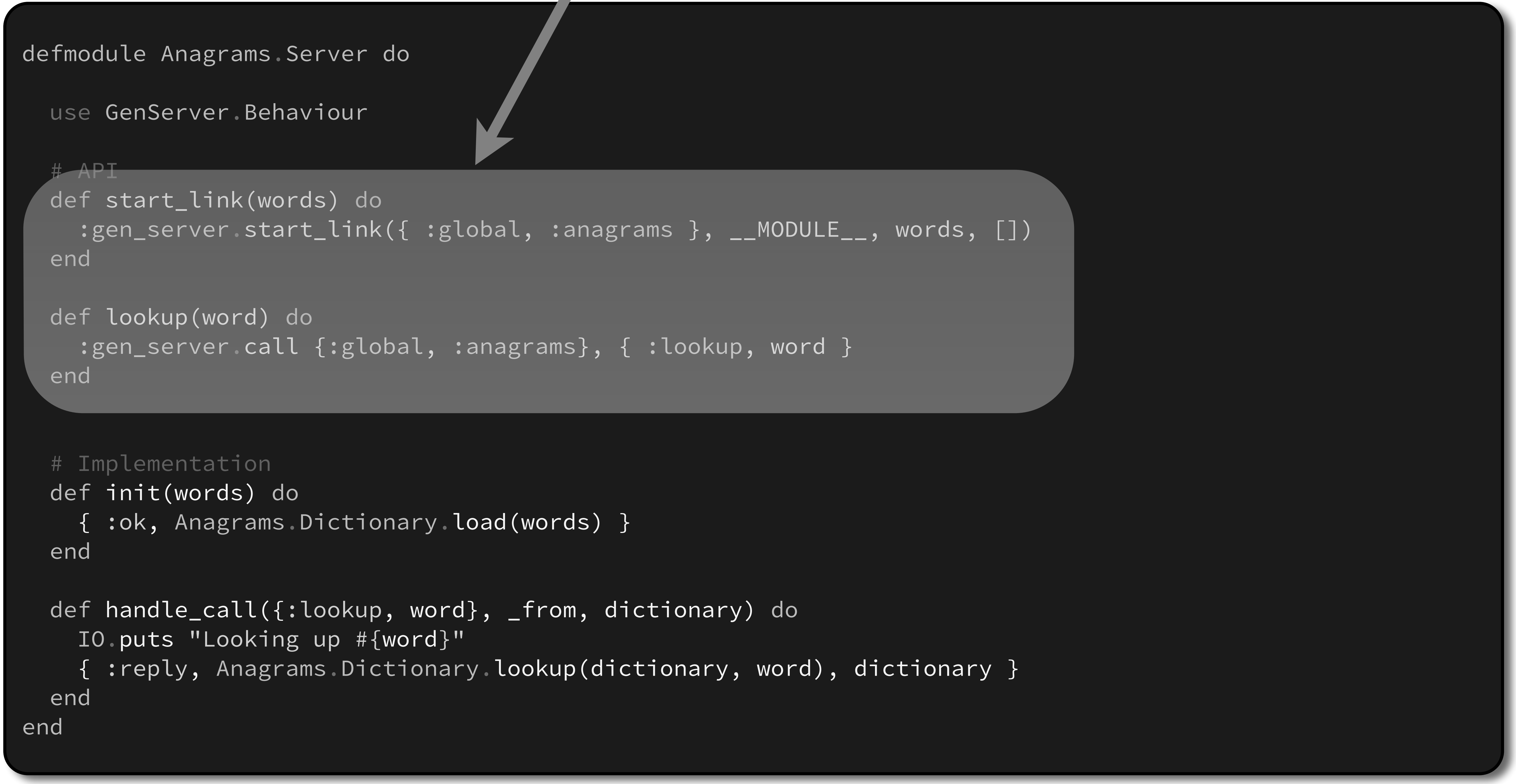


Code Server

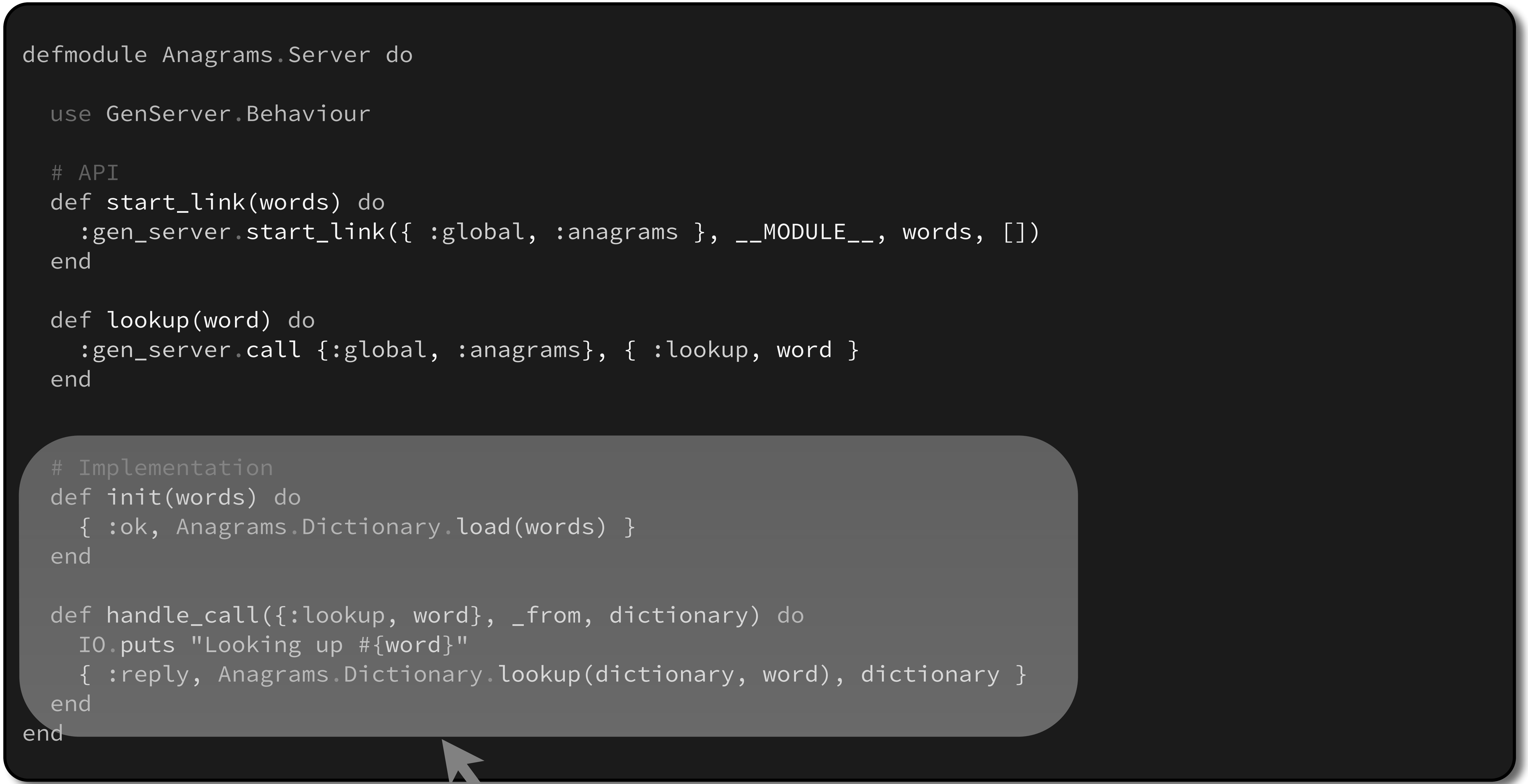
introduce another module
to this one

```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```

These functions are the API.
They run in the caller's process



```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```

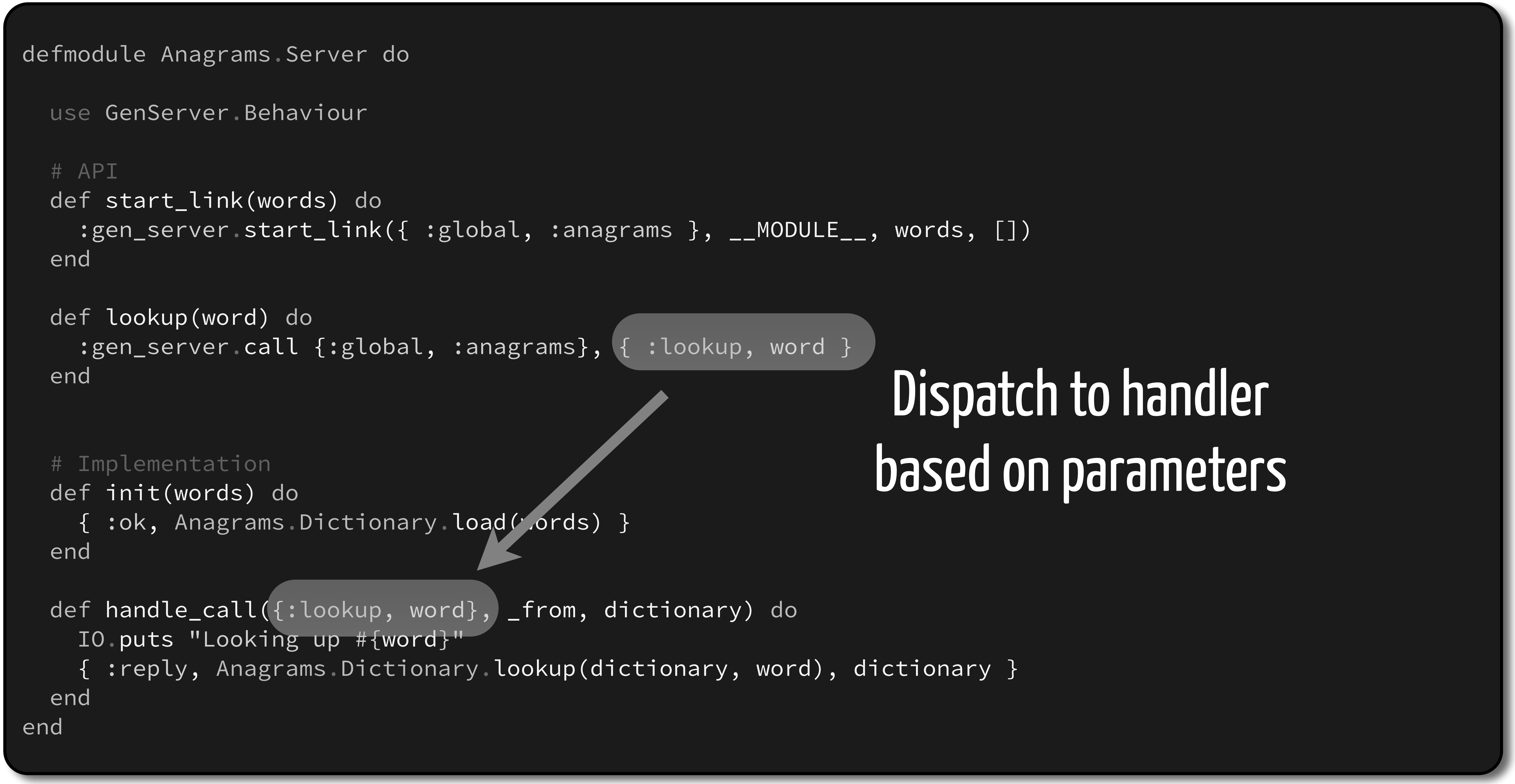


```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```

These callback functions are invoked by gen_server, and run in the server's process

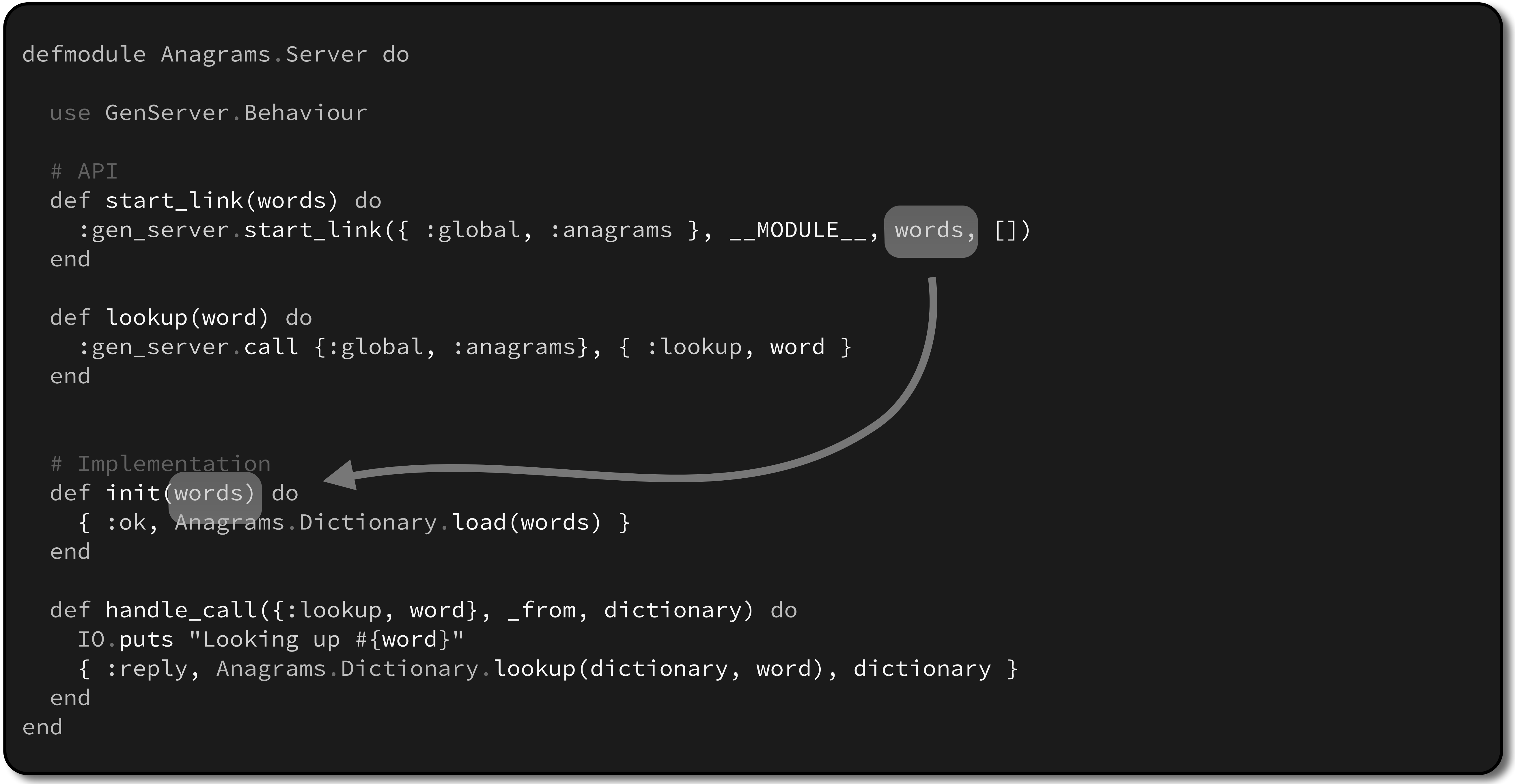
```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```

Calls our Dictionary module,
but from server process

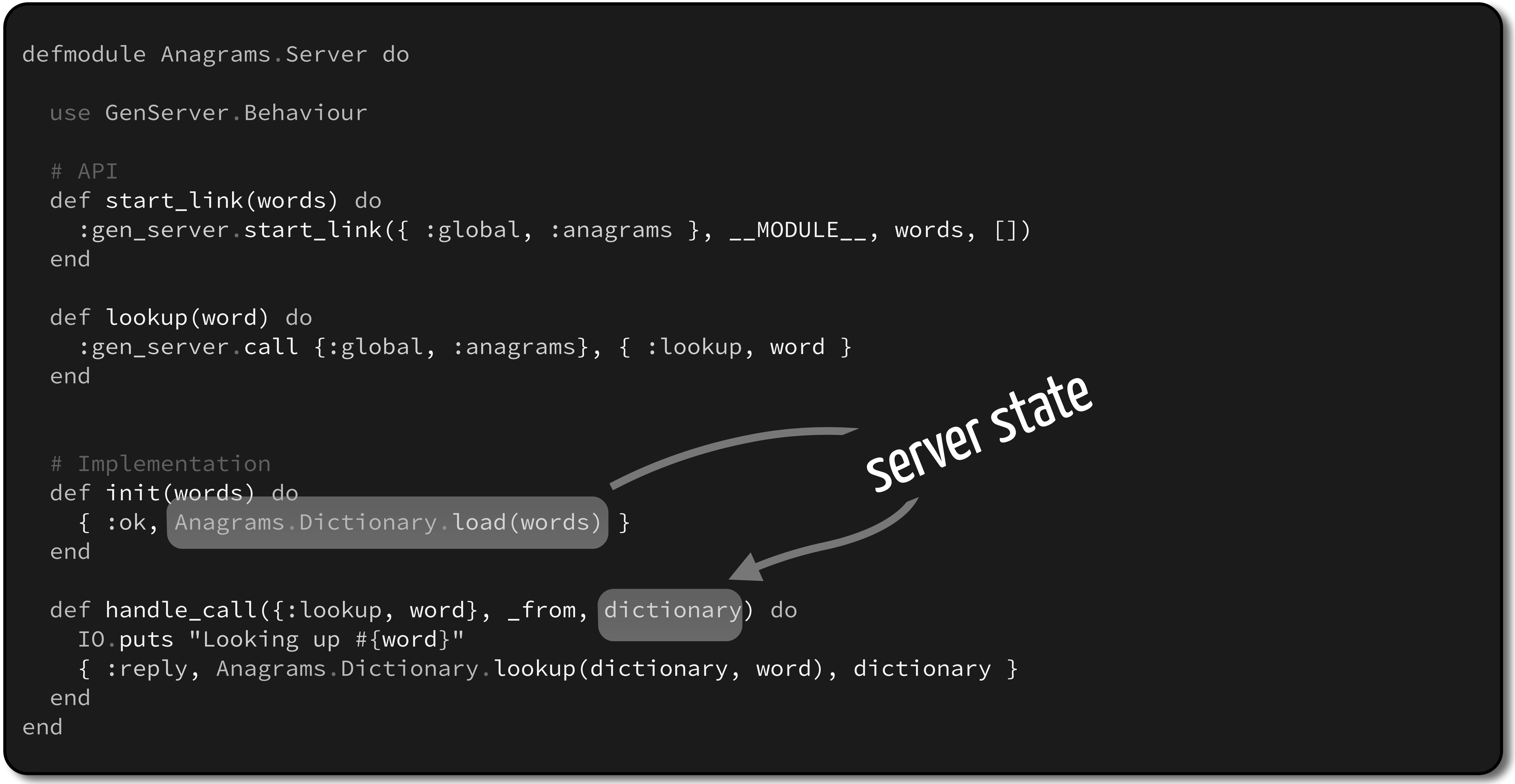


```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```

Dispatch to handler based on parameters

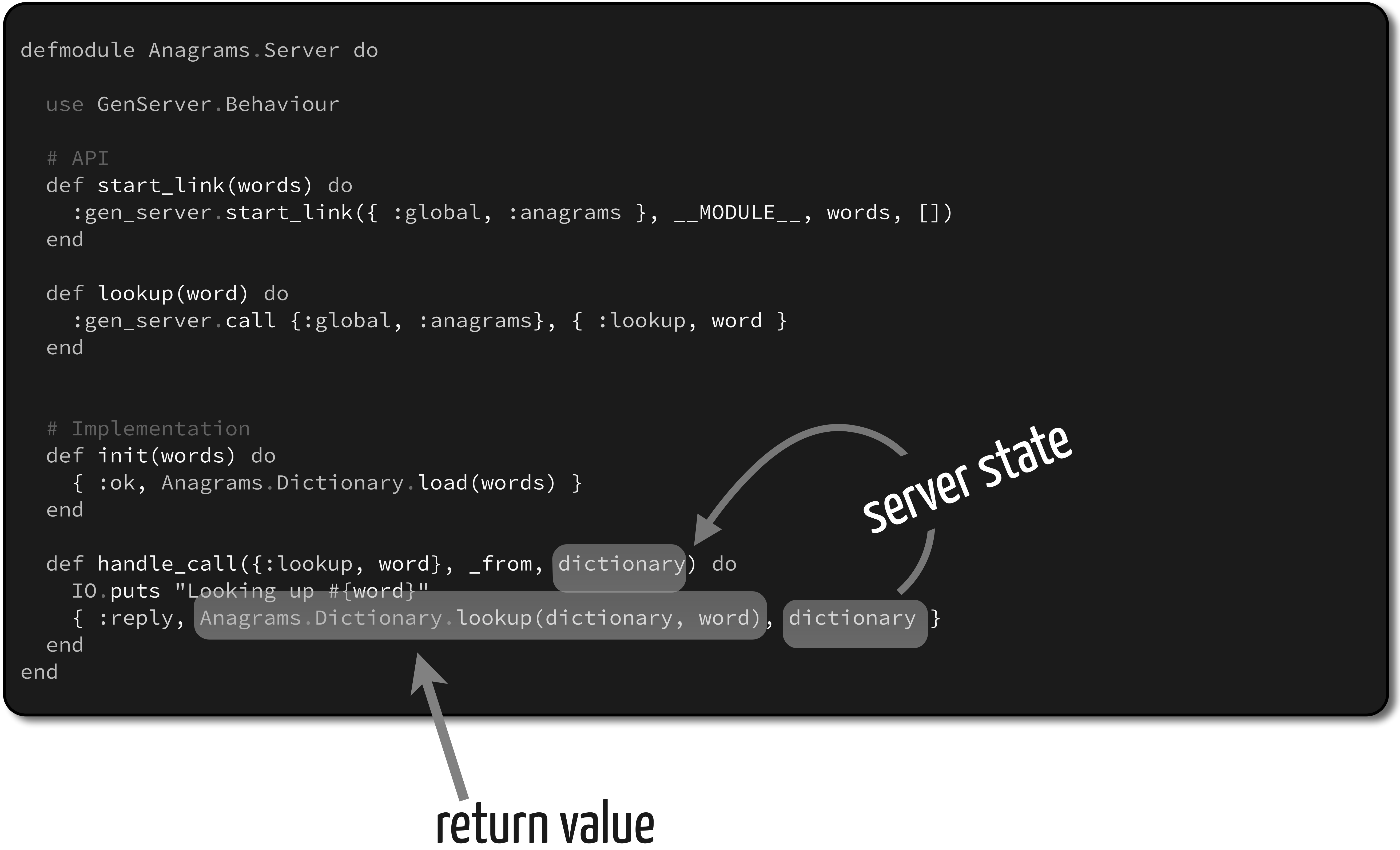


```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```



```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```

server state



```
defmodule Anagrams.Server do

  use GenServer.Behaviour

  # API
  def start_link(words) do
    :gen_server.start_link({ :global, :anagrams }, __MODULE__, words, [])
  end

  def lookup(word) do
    :gen_server.call {:global, :anagrams}, { :lookup, word }
  end


  # Implementation
  def init(words) do
    { :ok, Anagrams.Dictionary.load(words) }
  end

  def handle_call({:lookup, word}, _from, dictionary) do
    IO.puts "Looking up #{word}"
    { :reply, Anagrams.Dictionary.lookup(dictionary, word), dictionary }
  end
end
```

server state

return value

Supervising

Starts supervisor process.
Runs in caller's process.

```
defmodule Anagrams.Supervisor do
  use Supervisor.Behaviour

  def start_link do
    :supervisor.start_link(__MODULE__, [])
  end

  def init([]) do
    children = [
      worker(Anagrams.Server, [ "/usr/share/dict/words" ])
    ]

    supervise(children, strategy: :one_for_one)
  end
end
```

```
defmodule Anagrams.Supervisor do
  use Supervisor.Behaviour

  def start_link do
    :supervisor.start_link(__MODULE__, [])
  end

  def init([]) do
    children = [
      worker(Anagrams.Server, [ "/usr/share/dict/words" ])
    ]

    supervise(children, strategy: :one_for_one)
  end
end
```

Starts child processes.
Specifies supervisor strategy.

```
defmodule Anagrams.Supervisor do
  use Supervisor.Behaviour

  def start_link do
    :supervisor.start_link(__MODULE__, [])
  end

  def init([]) do
    children = [
      worker(Anagrams.Server, [ "/usr/share/dict/words" ])
    ]

    supervise(children, strategy: :one_for_one)
  end
end
```

Parameter(s) passed
to workers

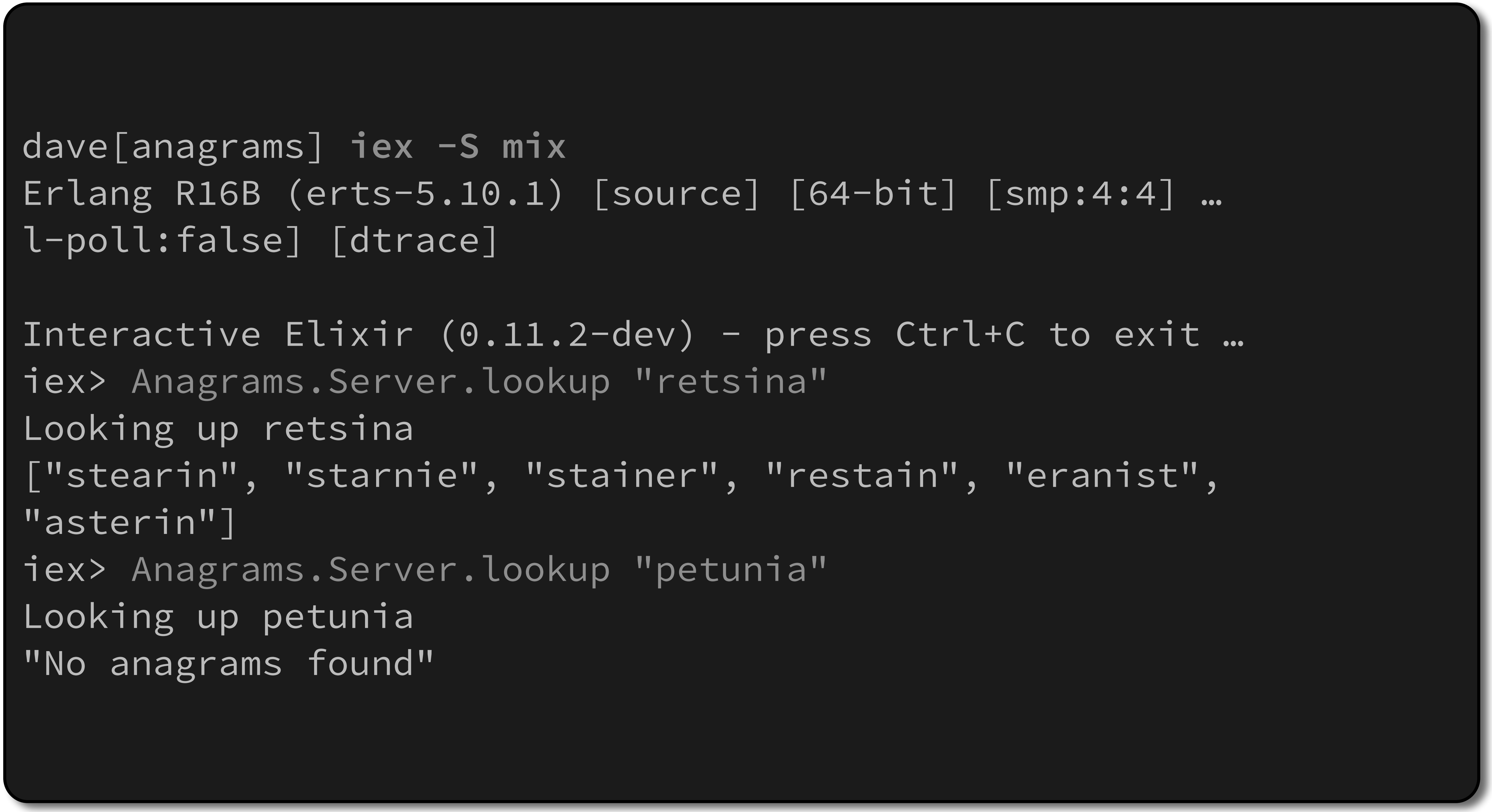
Set it Running

```
Dave[projects] tree anagrams
anagrams
├── README.md
├── lib
│   ├── anagrams
│   │   ├── cli.ex
│   │   ├── dictionary.ex
│   │   └── supervisor.ex
│   └── anagrams.ex
├── mix.exs
└── test
    ├── anagrams_test.exs
    └── test_helper.exs
```

Generated by “mix new anagrams”

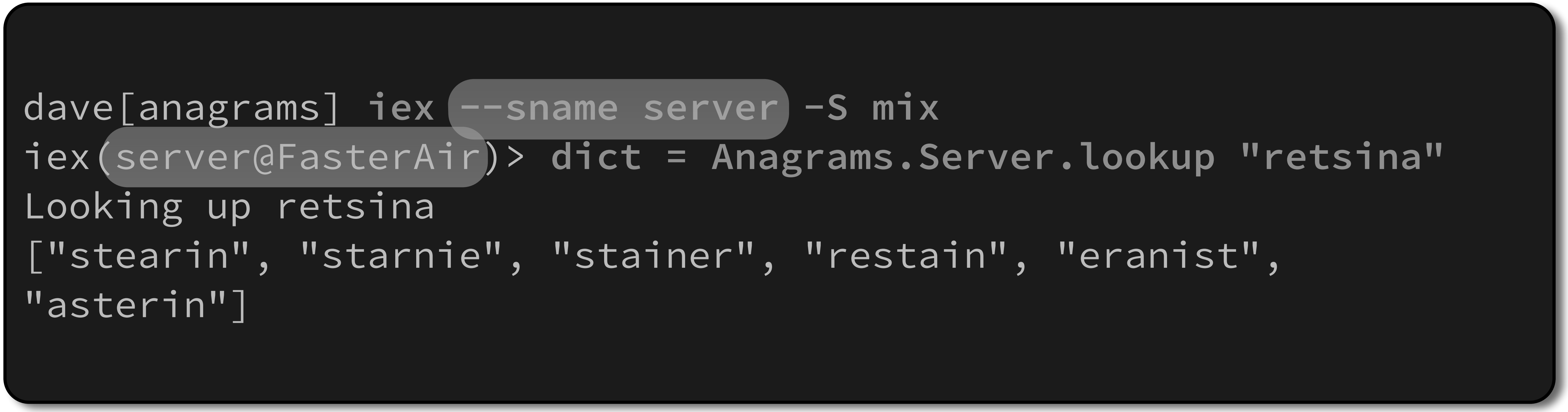
```
defmodule Anagrams do
  use Application.Behaviour

  def start(_type, _args) do
    Anagrams.Supervisor.start_link
  end

end
```

Can also pass in arguments (for example from command line)

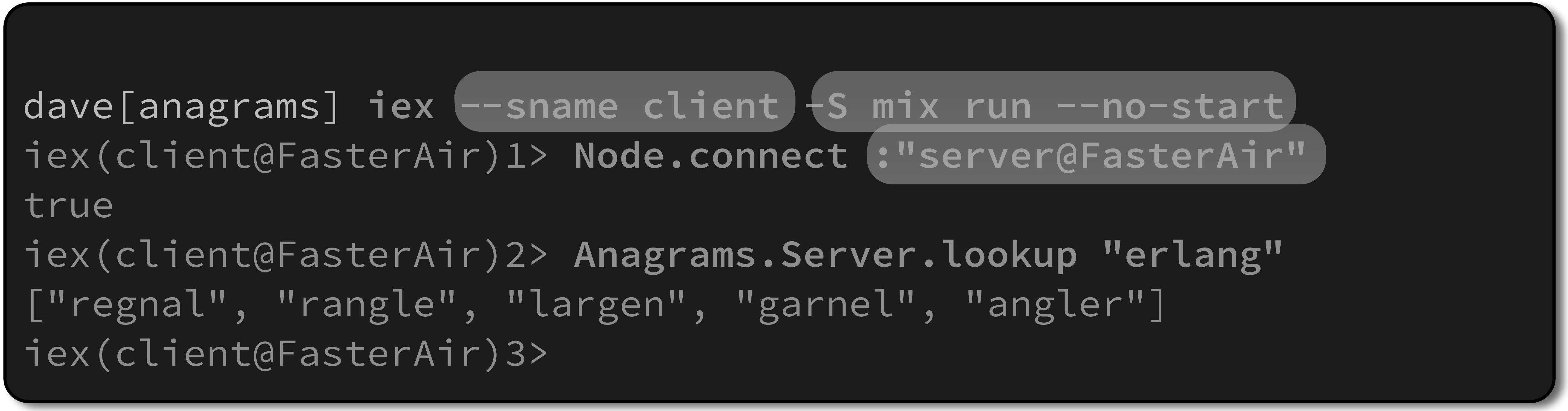


```
dave[anagrams] iex -S mix
Erlang R16B (erts-5.10.1) [source] [64-bit] [smp:4:4] …
l-poll:false] [dtrace]

Interactive Elixir (0.11.2-dev) - press Ctrl+C to exit …
iex> Anagrams.Server.lookup "retsina"
Looking up retsina
["stearin", "starnie", "stainer", "restain", "eranist",
"asterin"]
iex> Anagrams.Server.lookup "petunia"
Looking up petunia
"No anagrams found"
```

Make it Distributed



```
dave[anagrams] iex --sname server -S mix
iex(server@FasterAir)> dict = Anagrams.Server.lookup "retsina"
Looking up retsina
["stearin", "starnie", "stainer", "restain", "eranist",
"asterin"]
```

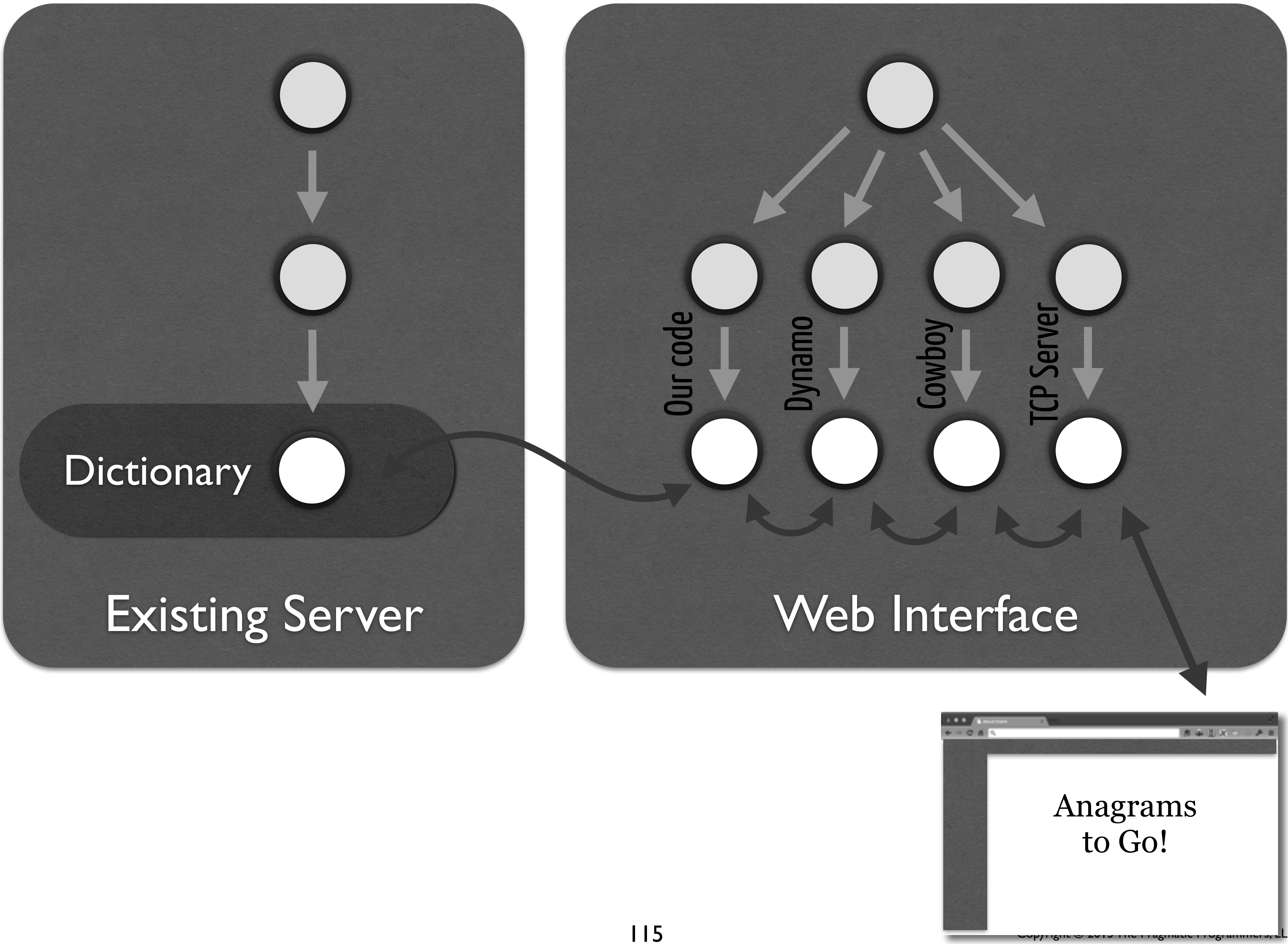


```
dave[anagrams] iex --sname client -S mix run --no-start
iex(client@FasterAir)1> Node.connect :"server@FasterAir"
true
iex(client@FasterAir)2> Anagrams.Server.lookup "erlang"
["regnal", "rangle", "largen", "garnel", "angler"]
iex(client@FasterAir)3>
```

To The Web!

Erlang/Elixir Web Options

- Low-level TCP to high-level frameworks
- HTTOption is a good web client
- Cowboy is a good web server
- Higher level Elixir frameworks: Dynamo, Weber...
    - all changing rapidly



Dictionary

Existing Server

Our code

Dynamo

Cowboy

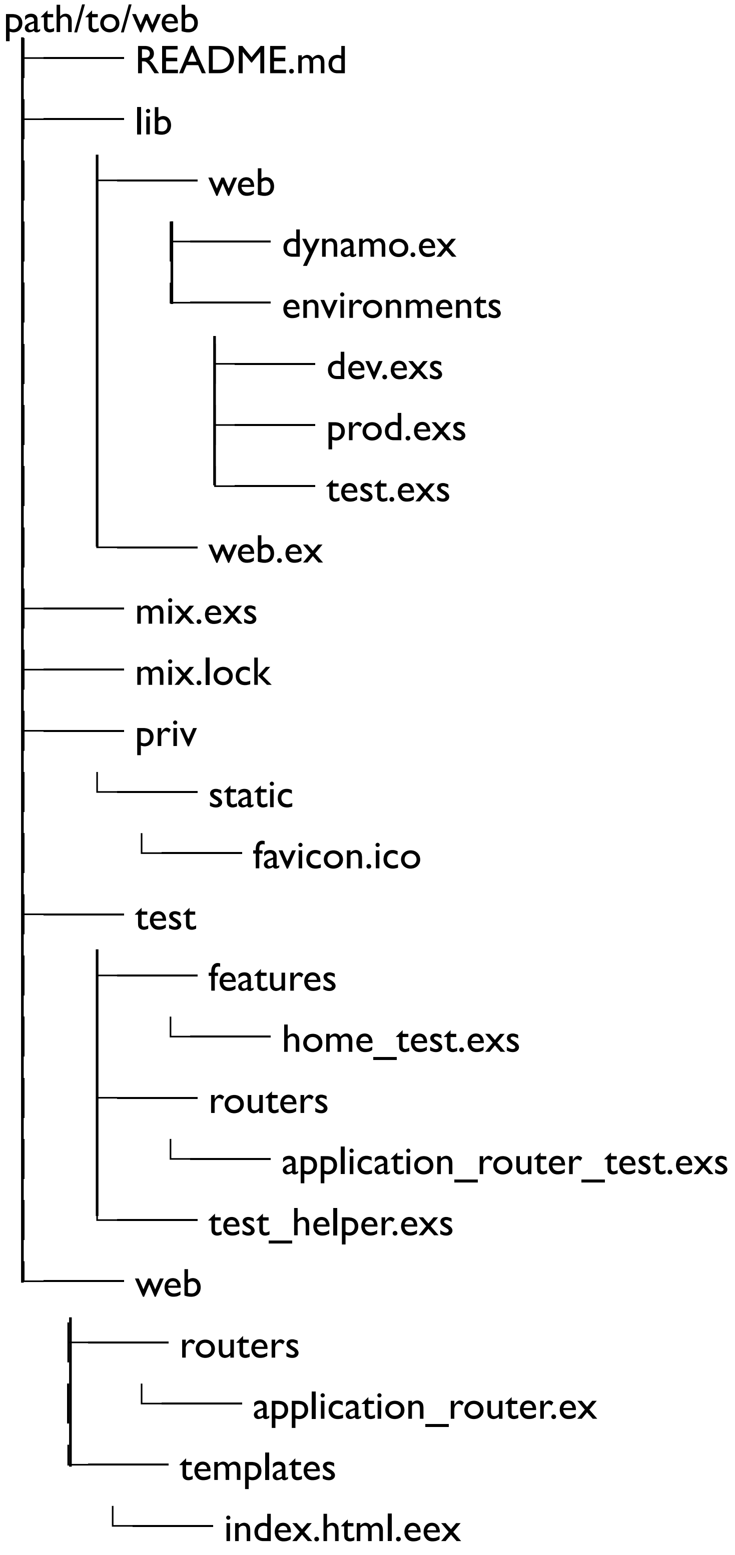
TCP Server

Web Interface

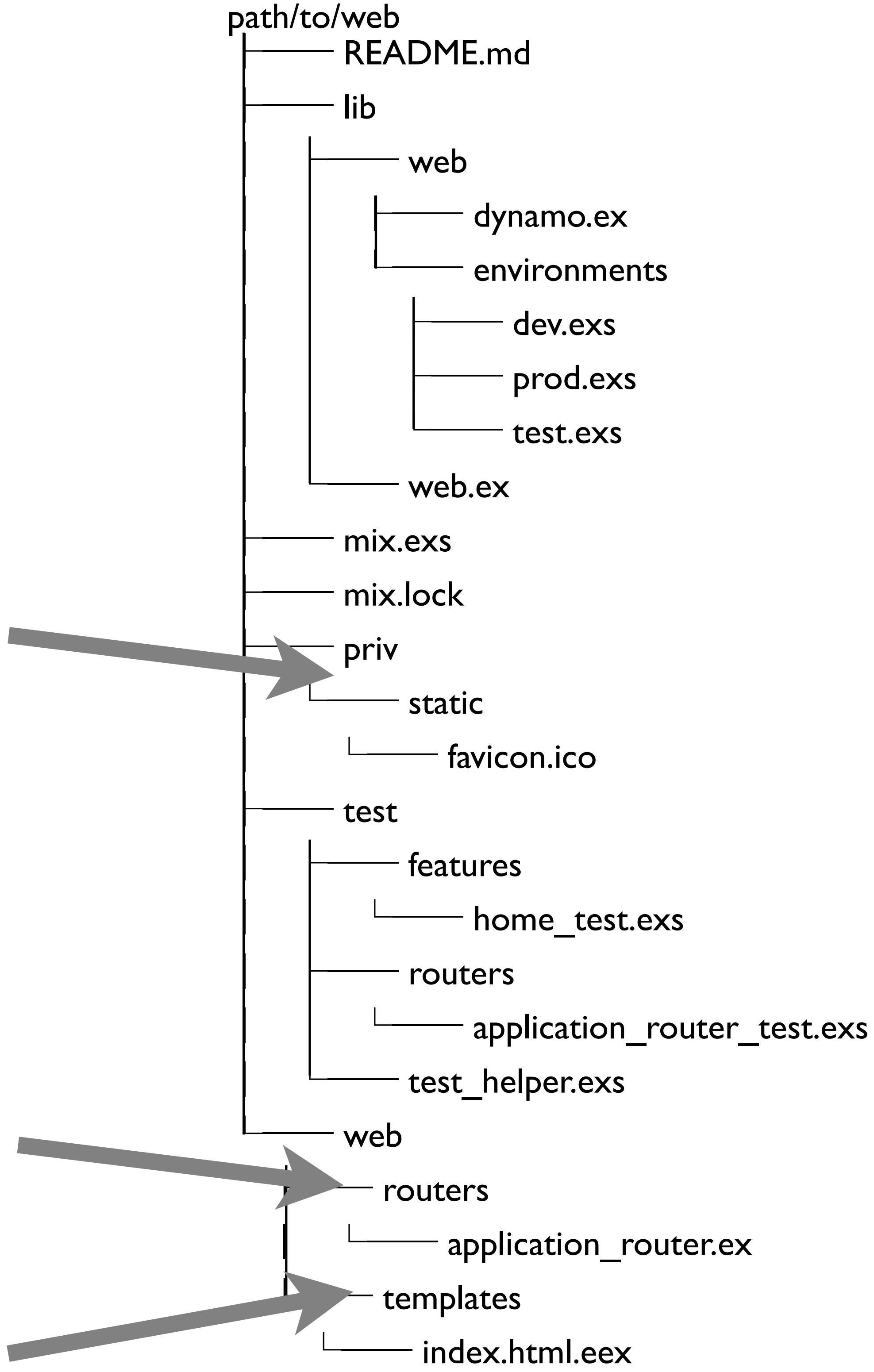
Anagrams
to Go!

Create a Dynamo App

```
Dave[projects] mix dynamo path/to/web
dave[Play/dynamo] mix dynamo ~/tmp/xx/web
* creating README.md
* creating .gitignore
* creating mix.lock
* creating mix.exs
* creating web
* creating web/routers
* creating web/routers/application_router.ex
* creating web/templates
* creating web/templates/index.html.eex
* creating lib
* creating lib/web.ex
* creating lib/web
* creating lib/web/dynamo.ex
* creating lib/web/environments
* creating lib/web/environments/dev.exs
* creating lib/web/environments/test.exs
* creating lib/web/environments/prod.exs
* creating priv
* creating priv/static
* creating priv/static/favicon.ico
* creating test
* creating test/test_helper.exs
* creating test/features
* creating test/features/home_test.exs
* creating test/routers
* creating test/routers/application_router_test.exs
```



```
path/to/web
├── README.md
├── lib
│   ├── web
│   │   ├── dynamo.ex
│   │   └── environments
│   │       ├── dev.exs
│   │       ├── prod.exs
│   │       └── test.exs
│   └── web.ex
├── mix.exs
├── mix.lock
├── priv
│   └── static
│       └── favicon.ico
├── test
│   ├── features
│   │   └── home_test.exs
│   ├── routers
│   │   └── application_router_test.exs
│   └── test_helper.exs
└── web
    ├── routers
    │   └── application_router.ex
    └── templates
        └── index.html.eex
```



Static content

Entry points for request handling

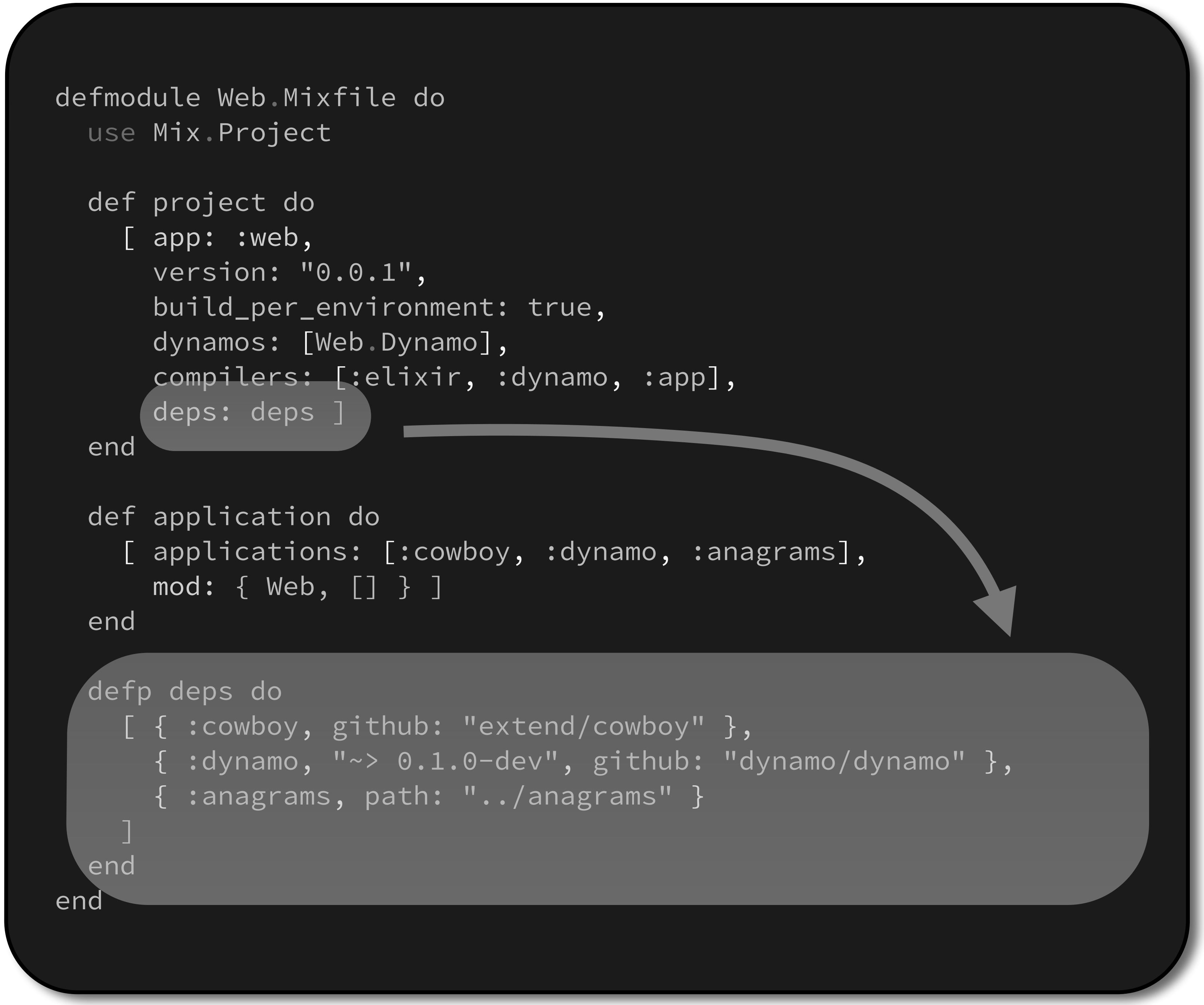
Response templates

```
path/to/web
├── README.md
├── lib
│   ├── web
│   │   ├── dynamo.ex
│   │   └── environments
│   │       ├── dev.exs
│   │       ├── prod.exs
│   │       └── test.exs
│   └── web.ex
├── mix.exs
├── mix.lock
├── priv
│   └── static
│       └── favicon.ico
├── test
│   ├── features
│   │   └── home_test.exs
│   ├── routers
│   │   └── application_router_test.exs
│   └── test_helper.exs
└── web
    ├── routers
    │   └── application_router.ex
    └── templates
        └── index.html.eex
```

Dependencies for Web Server

Declare Dependencies

- dynamo
- cowboy
- our anagram server



```
defmodule Web.Mixfile do
  use Mix.Project

  def project do
    [ app: :web,
      version: "0.0.1",
      build_per_environment: true,
      dynamos: [Web.Dynamo],
      compilers: [:elixir, :dynamo, :app],
      deps: deps ]
  end

  def application do
    [ applications: [:cowboy, :dynamo, :anagrams],
      mod: { Web, [] } ]
  end

  defp deps do
    [ { :cowboy, github: "extend/cowboy" },
      { :dynamo, "~> 0.1.0-dev", github: "dynamo/dynamo" },
      { :anagrams, path: "../anagrams" }
    ]
  end
end
```

Declare Dependencies

mix deps.get

loads self-contained applications into deps/ directory

```
defmodule Web.Mixfile do
  use Mix.Project

  def project do
    [ app: :web,
      version: "0.0.1",
      build_per_environment: true,
      dynamos: [Web.Dynamo],
      compilers: [:elixir, :dynamo, :app],
      deps: deps ]
  end

  def application do
    [ applications: [:cowboy, :dynamo, :anagrams],
      mod: { Web, [] } ]
  end

  defp deps do
    [ { :cowboy, github: "extend/cowboy" },
      { :dynamo, "~> 0.1.0-dev", github: "dynamo/dynamo" },
      { :anagrams, path: "../anagrams" }
    ]
  end
end
```

Start Applications

```
defmodule Web.Mixfile do
  use Mix.Project

  def project do
    [ app: :web,
      version: "0.0.1",
      build_per_environment: true,
      dynamos: [Web.Dynamo],
      compilers: [:elixir, :dynamo, :app],
      deps: deps ]
  end

  def application do
    [ applications: [:cowboy, :dynamo, :anagrams],
      mod: { Web, [] } ]
  end

  defp deps do
    [ { :cowboy, github: "extend/cowboy" },
      { :dynamo, "~> 0.1.0-dev", github: "dynamo/dynamo" },
      { :anagrams, path: "../anagrams" }
    ]
  end
end
```

starts each app
just like our
gen_server app

Write Routing Code and Templates

```
defmodule ApplicationRouter do
  use Dynamo.Router

  prepare do
    conn.fetch([:params])     # preload parameters on each request
  end

  get "/" do
   render conn, "index.html"  # static content on "/"
  end

  get "/:word" do             # otherwise take content as a word to look up
    word = conn.params[:word]
    anagram_list(conn, word, Anagrams.Server.lookup(word))
  end

 # Helpers...

  def anagram_list(conn, _, msg) when is_binary(msg) do
    conn.resp_body(msg)
  end

  def anagram_list(conn, word, list) do
    conn
    .assign(:word, word)
    .assign(:anagrams, list)
    |> render"anagrams.html"
  end

end
```

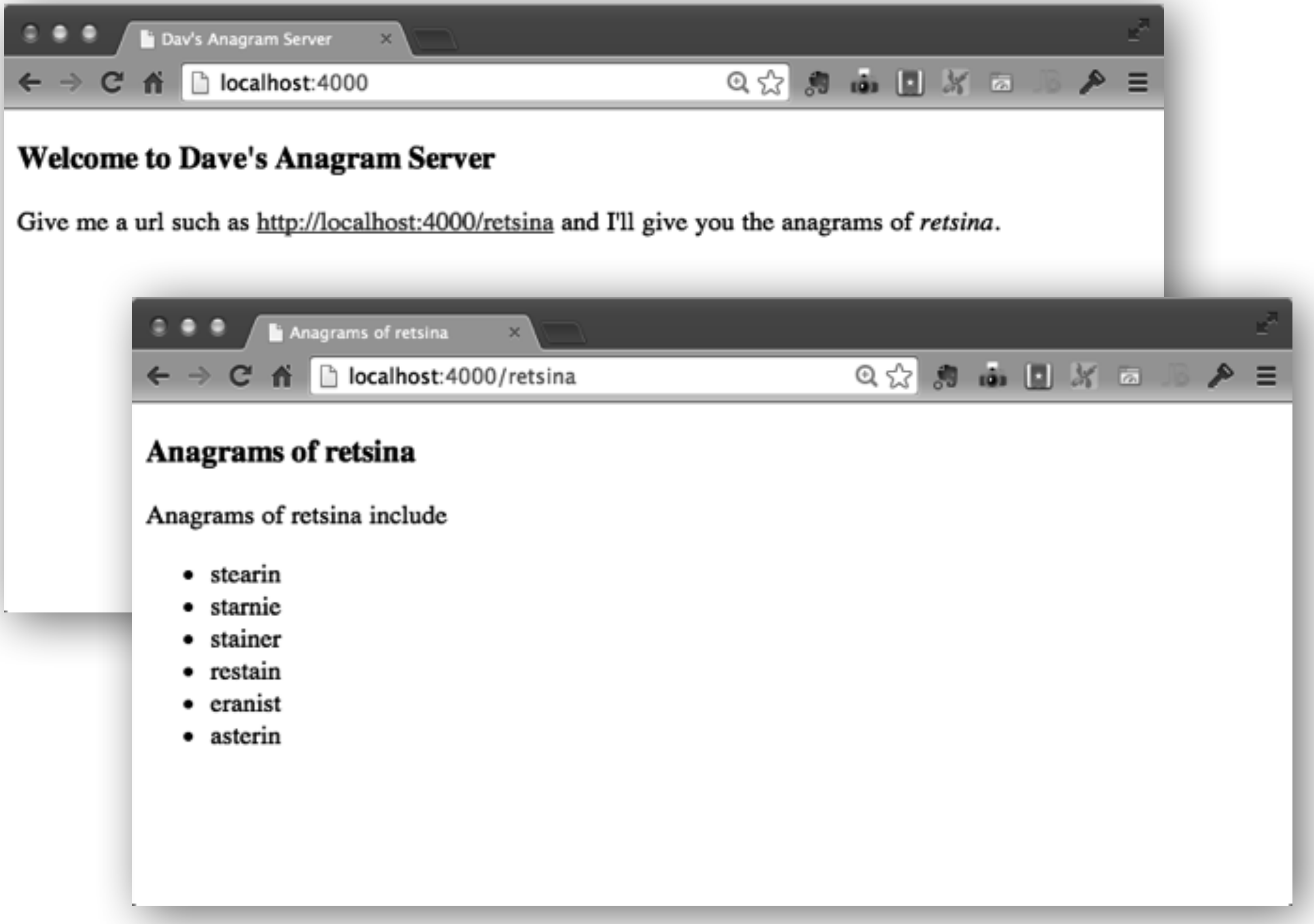
web/web/templates/index.html.eex

```
<!DOCTYPE HTML>
<html>
<head>
  <title>Dave's Anagram Server</title>
  <link rel="shortcut icon" href="/static/favicon.ico" />
</head>
<body>
  <h3>Welcome to Dave's Anagram Server</h3>

  <p>
    Give me a url such as
    <a href="retsina">http://localhost:4000/retsina</a>
    and I'll give you the anagrams of <em>retsina</em>.
  </p>
</body>
</html>
```

web/web/templates/anagrams.html.eex

```
 <!DOCTYPE HTML>
<html>
  <head>
    <title>Anagrams of <%= @word %></title>
  </head>
  <body>
    <h3>Anagrams of <%= @word %></h3>
    <p>
       Anagrams of <%= @word %> include
    </p>
    <ul>
       <%= lc anagram inlist @anagrams do %>
       <li><%= anagram %></li>
       <% end %>
    </ul>
  </body>
</html>
```

```
$ mix server
```



Welcome to Dave's Anagram Server

Give me a url such as http://localhost:4000/retsina and I'll give you the anagrams of *retsina*.

Anagrams of retsina

Anagrams of retsina include

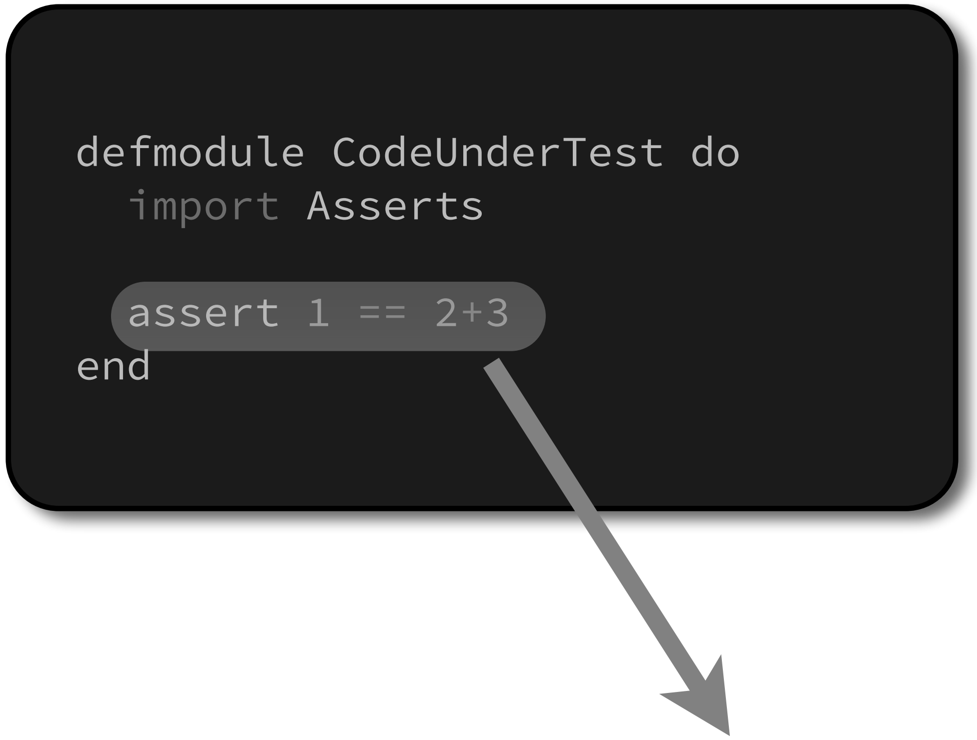
- stearin
- starnie
- stainer
- restain
- eranist
- asterin

Macros

```
defmodule CodeUnderTest do
  import Asserts

  assert 1 == 2+3
end
```

```
Expected 1 (1)

to equal 2 + 3 (5)
```

- “assert” macro receives expression “1 == 2 + 3”
- evaluates as expression
- reports errors based on both values and source



```
defmodule CodeUnderTest do
  import Asserts

  assert 1 == 2+3
end
```

- Intermediate value of code is tuples and lists.
- This is what is passed to macros.
- Can also create using `quote`

```
iex> quote do 1 == 2+3 end
{:==, [...],
      [ 1,
        {:+, [...], [2, 3]}
      ]
}
```

Macro

- defmacro name(args) do … end
- receives internal representation of args
- expected to return an internal representation of code to be inserted at point of call

Log Expression Execution

```
defmodule Example do
  defmacro log(expr) do
    IO.inspect expr
    expr
  end
end

defmodule UseExample do
  import Example

  IO.puts "Result = #{log 1 + 2}"

end
```

```
{:+, [line: 30], [1, 2]}
Result = 3
```

```
defmodule Example do
  defmacro log(expr) do
    IO.inspect expr
    expr
  end
end

defmodule UseExample do
  import Example

  IO.puts "Result = #{
                        IO.inspect {:+, [line: 30], [1, 2]}
                        1 + 2
                     }"
end
```

{:+, [line: 30], [1, 2]}

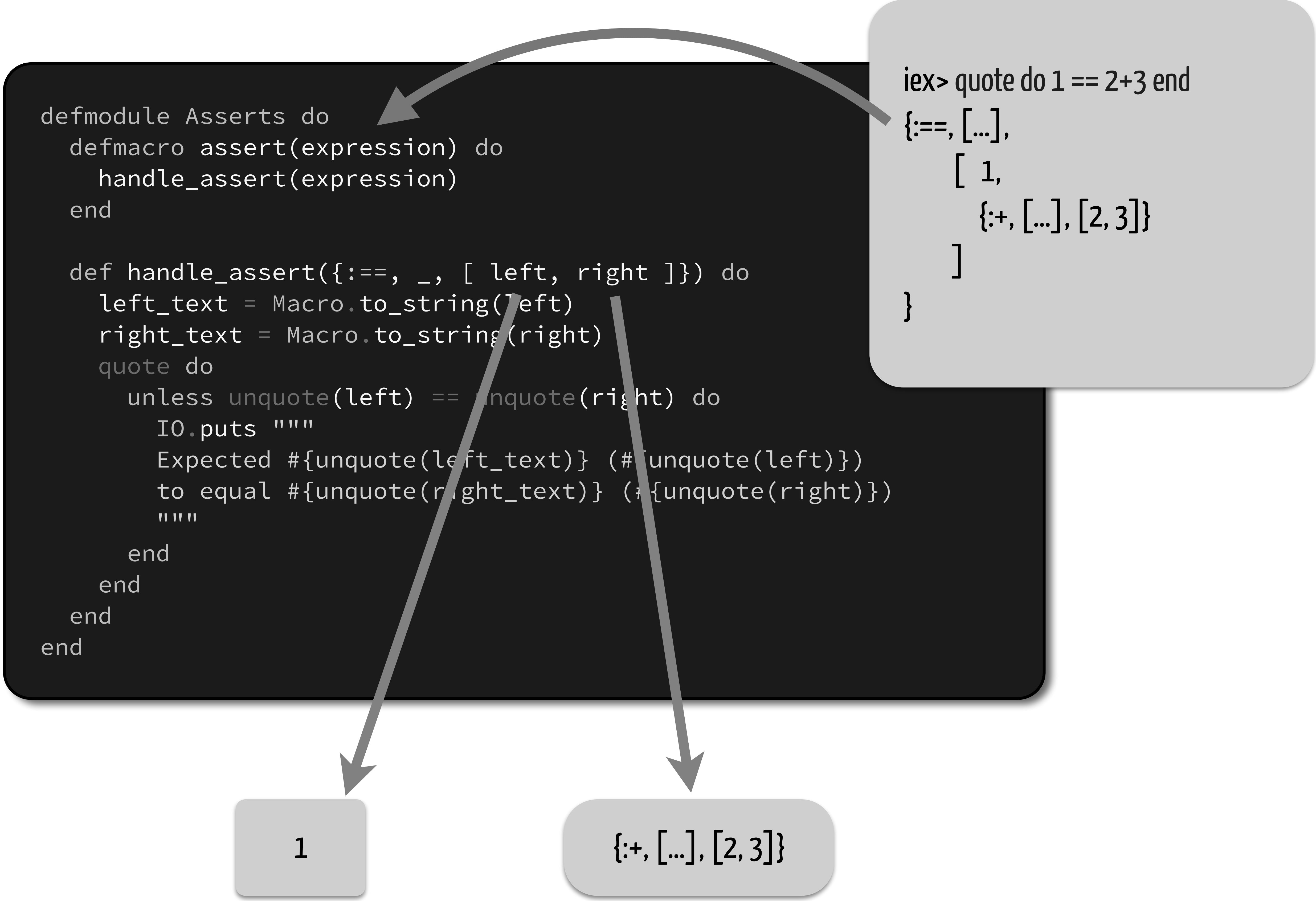
Result = 3

```
defmodule Example do
  defmacro log(expr) do
    source = Macro.to_string(expr)
    quote do
      IO.puts "#{unquote(source)} = #{unquote(expr)}"
      unquote(expr)
    end
  end
end

defmodule UseExample do
  import Example

  IO.puts "Result = #{log 1 + 2}"

end
```

```
1 + 2 = 3
Result = 3
```

- quote takes source code and returns internal representation
- unquote takes representation of code and inserts it as it source

A Sketch of the Assert macro



```
defmodule Asserts do
  defmacro assert(expression) do
    handle_assert(expression)
  end

  def handle_assert({:==, _, [ left, right ]}) do
    left_text = Macro.to_string(left)
    right_text = Macro.to_string(right)
    quote do
      unless unquote(left) == unquote(right) do
        IO.puts """
        Expected #{unquote(left_text)} (#{unquote(left)})
        to equal #{unquote(right_text)} (#{unquote(right)})
        """
      end
    end
  end
end
```

```
iex> quote do 1 == 2+3 end
{:==, [...],
     [ 1,
       {:+, [...], [2, 3]}
     ]
}
```

```
defmodule Asserts do
  defmacro assert(expression) do
    handle_assert(expression)
  end

  def handle_assert({:==, _, [ left, right ]}) do
    left_text = Macro.to_string(left)
    right_text = Macro.to_string(right)
    quote do
      unless unquote(left) == unquote(right) do
        IO.puts """
        Expected #{unquote(left_text)} (#{unquote(left)})
        to equal #{unquote(right_text)} (#{unquote(right)})
        """
      end
    end
  end
end
```

Expected 1 (1)
to equal 2 + 3 (5)

More Stuff...

We Didn’t Cover

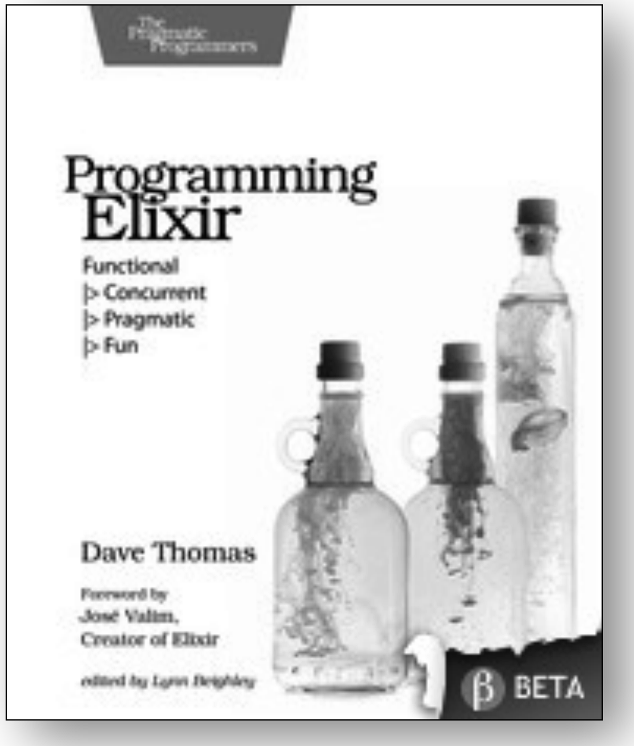
- Records—give names to fields in tuples
- Protocols
  — specify common behaviour between modules
  — implement “polymorphism”
- Use and __using__
- Behaviours

We Didn’t Cover

- Libraries (Elixir and Erlang)
- gen_fsm and gen_event
- Deployment (evolving, but Heroku support today)
- The future (Erlang R17 maps, two-stage compilation, etc)

Resources

- elixir-lang.org
- groups.google.com/forum/#!forum/elixir-lang-talk
- elixir-fountain.com



Getting Started in Functional Programming
Early Release
RAW & UNEDITED
Introducing
Elixir
O'REILLY®
Simon St. Laurent & J. David Eisenberg

elixir sips
with Josh Adams

Learn elixir with two short videos each week.

YOU!

Introducing Elixir

Thank you

Dave Thomas
[@+]pragdave
dave@pragprog.com

Pragmatic
Bookshelf

Thank you