Rob Howard @damncabbage

A Shallow Dip Into Dialyzer

1920 x 1080

80 Familiar Pro

55 Inconso.f({a:b});

43 Inconso.f({a:b});

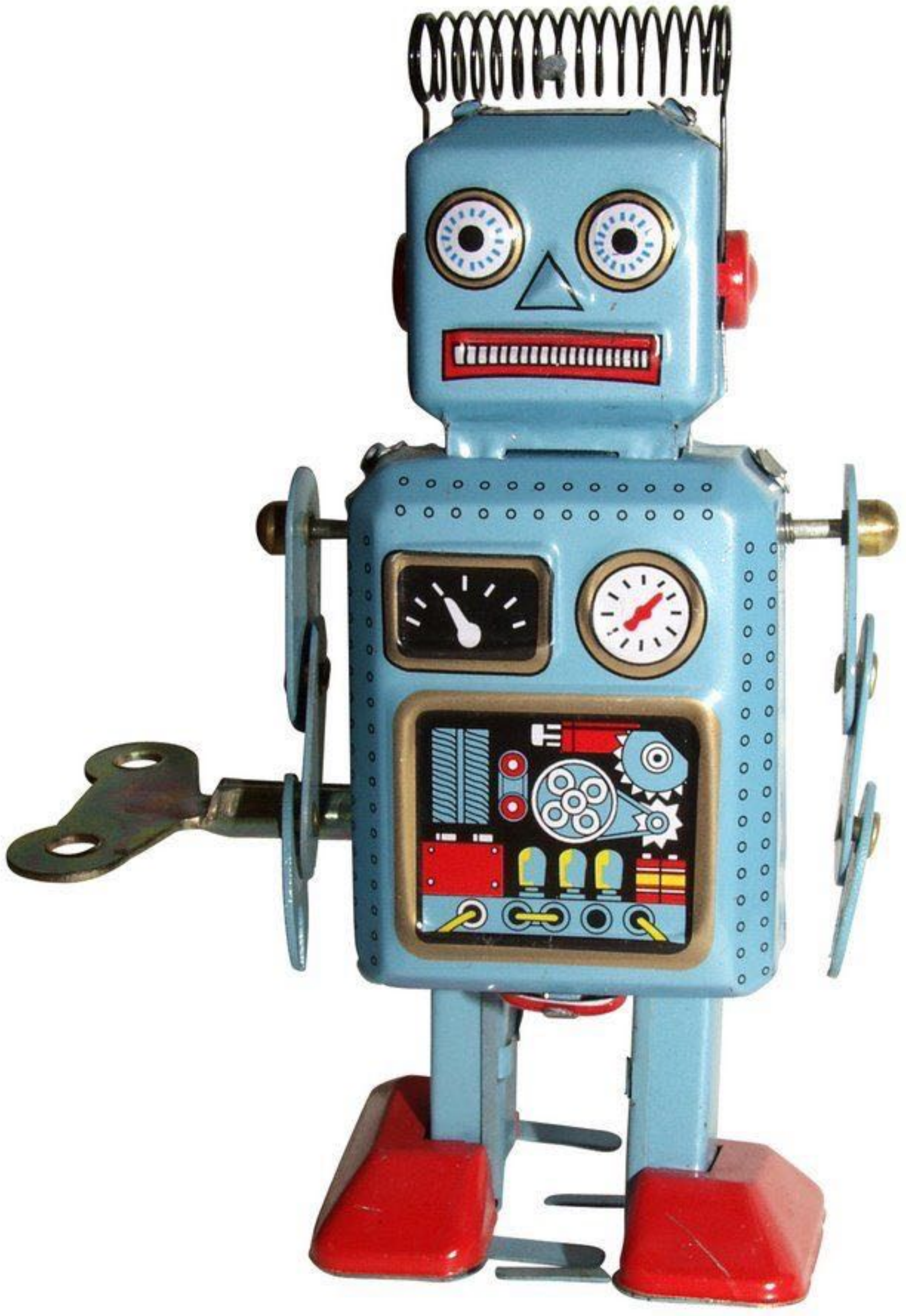
35 Open Sans

Famil Inconso.f() Open

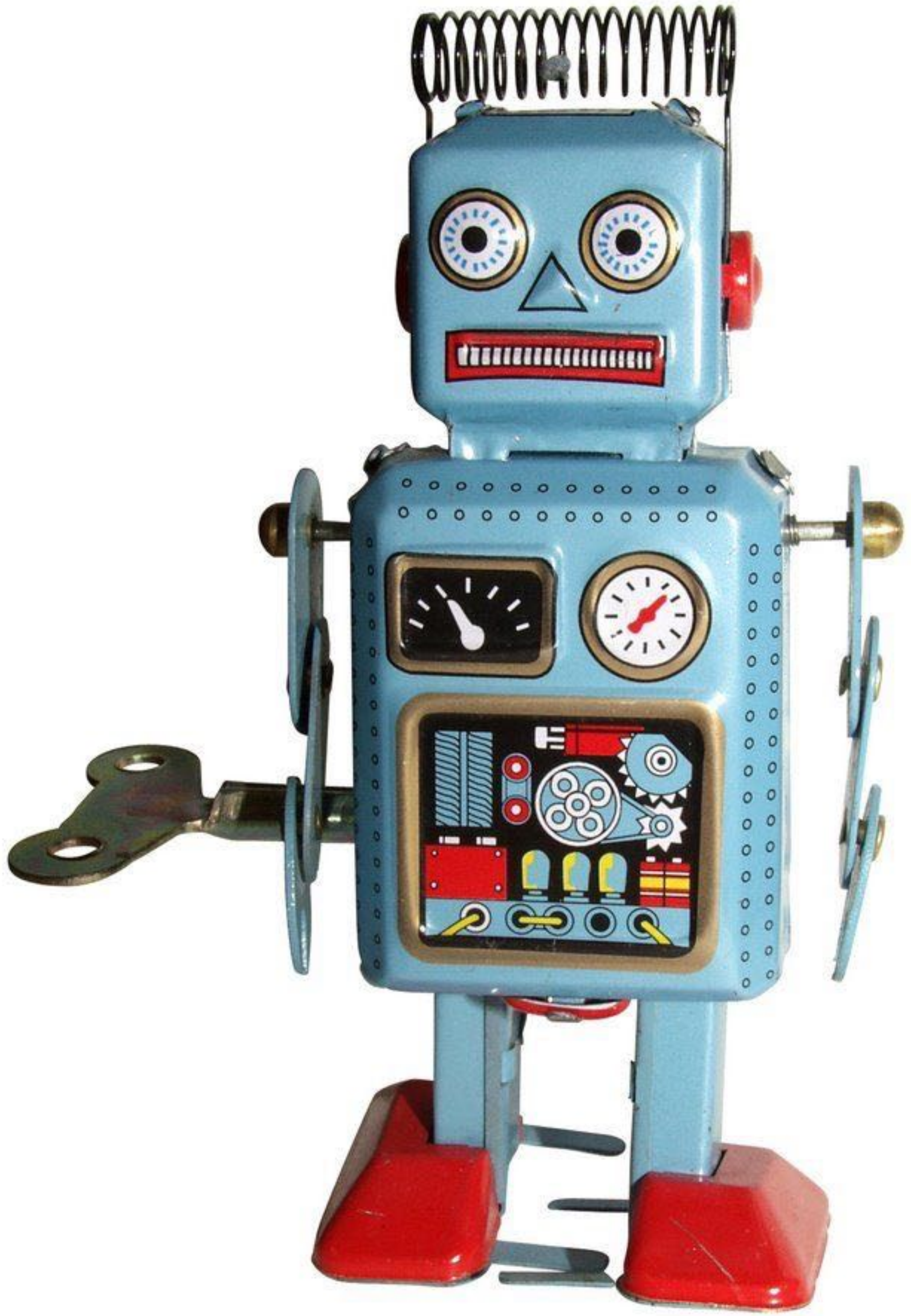
Famil Inconso.f() Open

An Elixir Safety Harness:

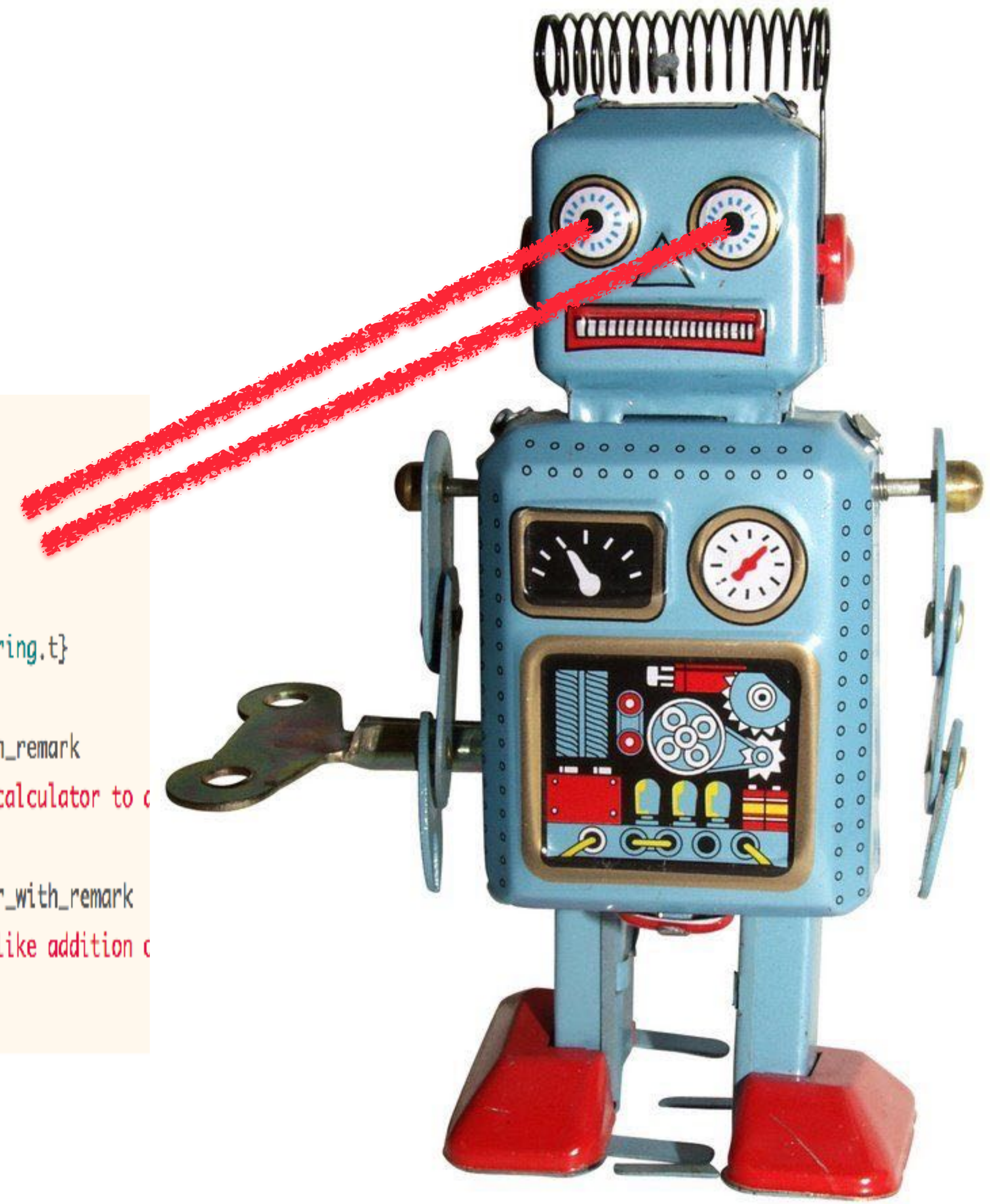
Dialyzer

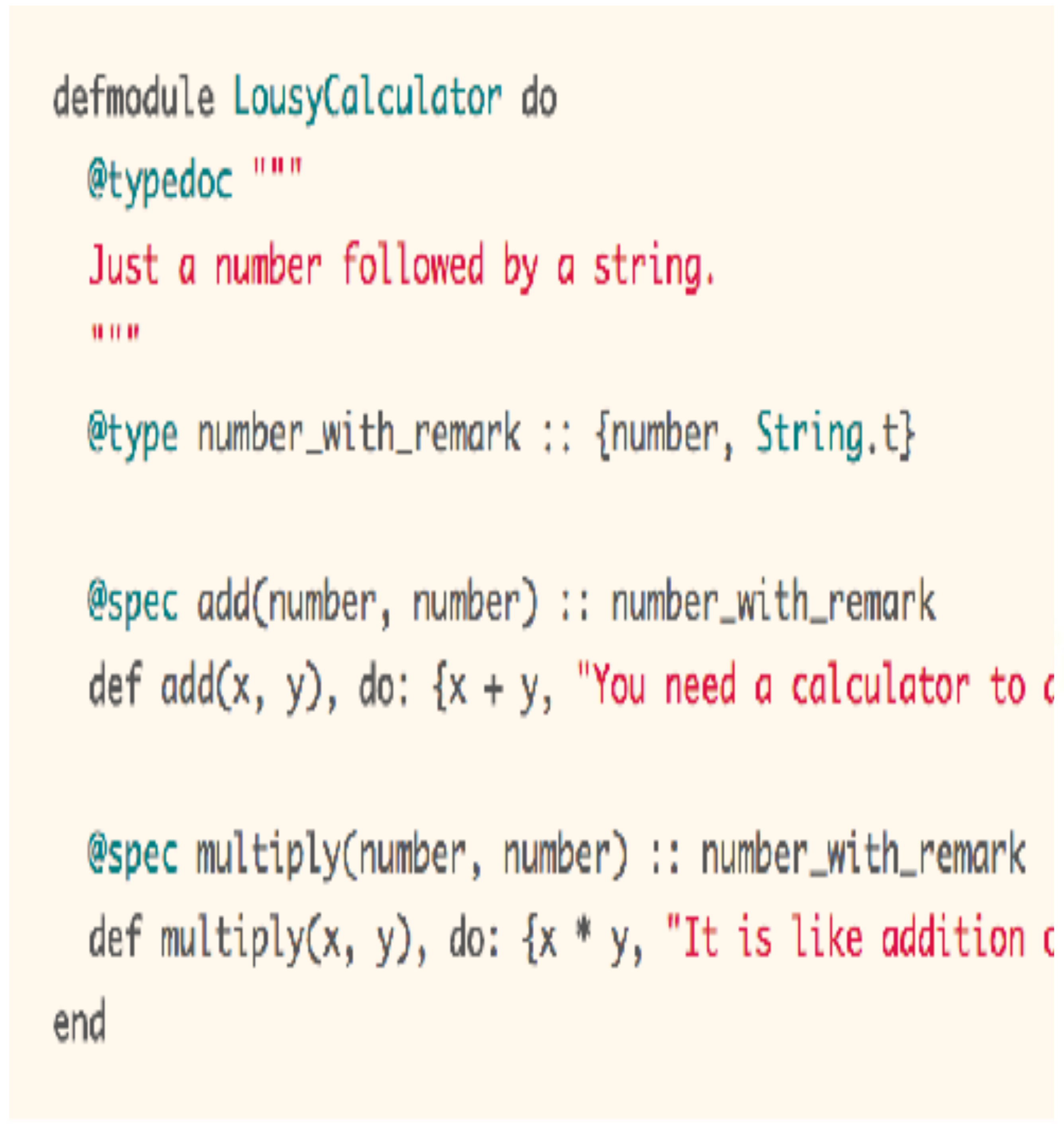


```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """
  @type number_with_remark :: {number, String.t}

  @spec add(number, number) :: number_with_remark
  def add(x, y), do: {x + y, "You need a calculator to

  @spec multiply(number, number) :: number_with_remark
  def multiply(x, y), do: {x * y, "It is like addition
end
```

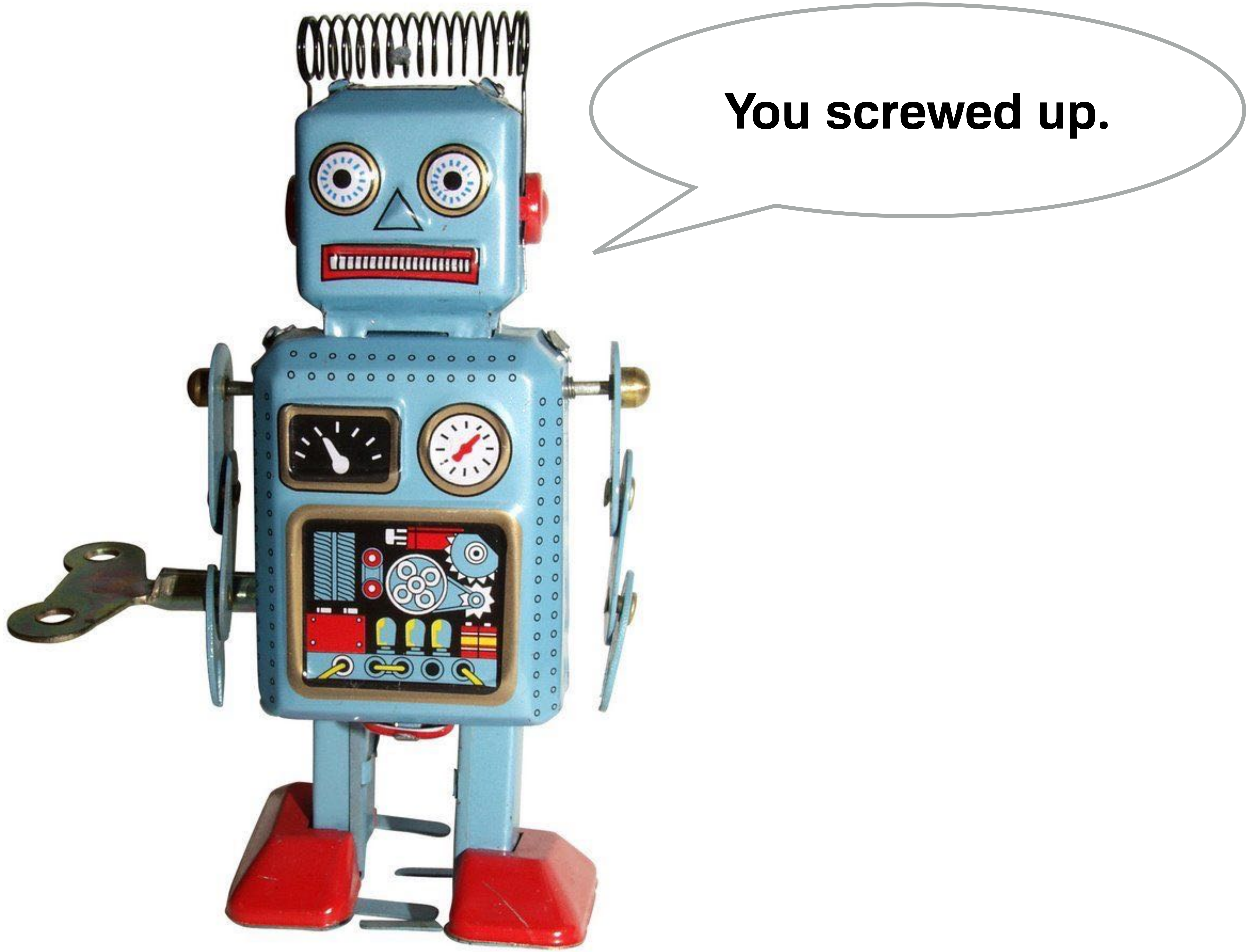


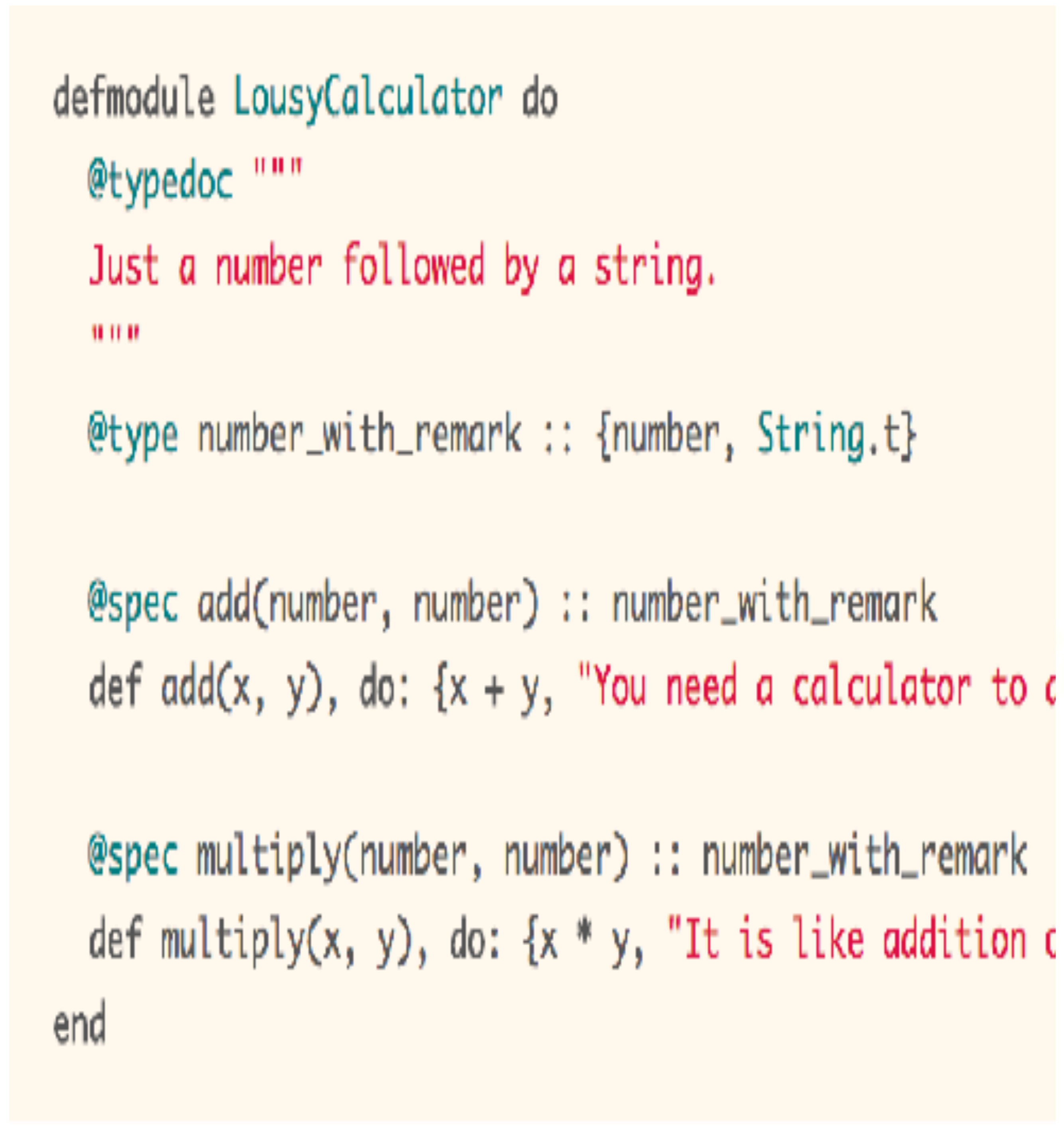
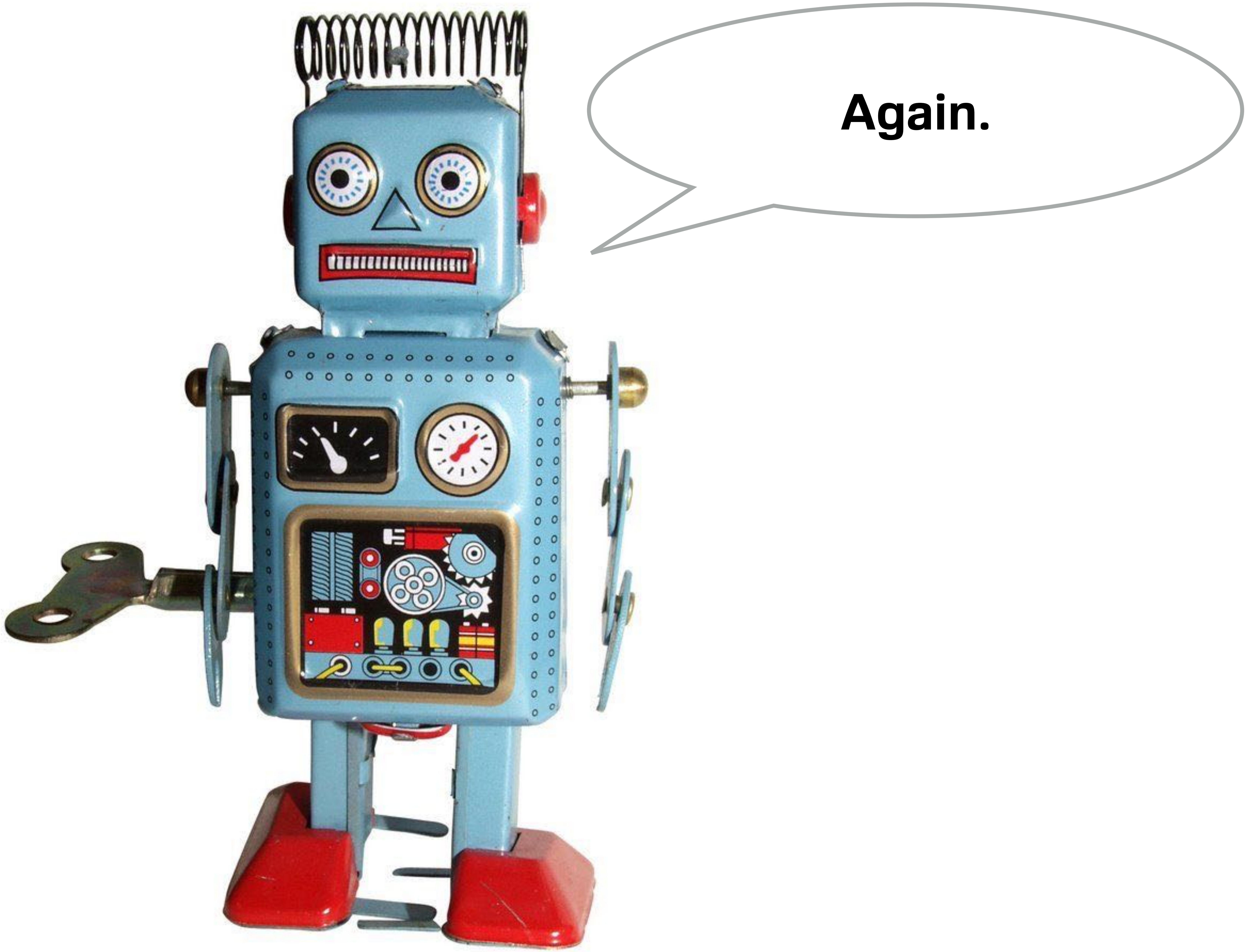
```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """
  @type number_with_remark :: {number, String.t}

  @spec add(number, number) :: number_with_remark
  def add(x, y), do: {x + y, "You need a calculator to

  @spec multiply(number, number) :: number_with_remark
  def multiply(x, y), do: {x * y, "It is like addition
end
```

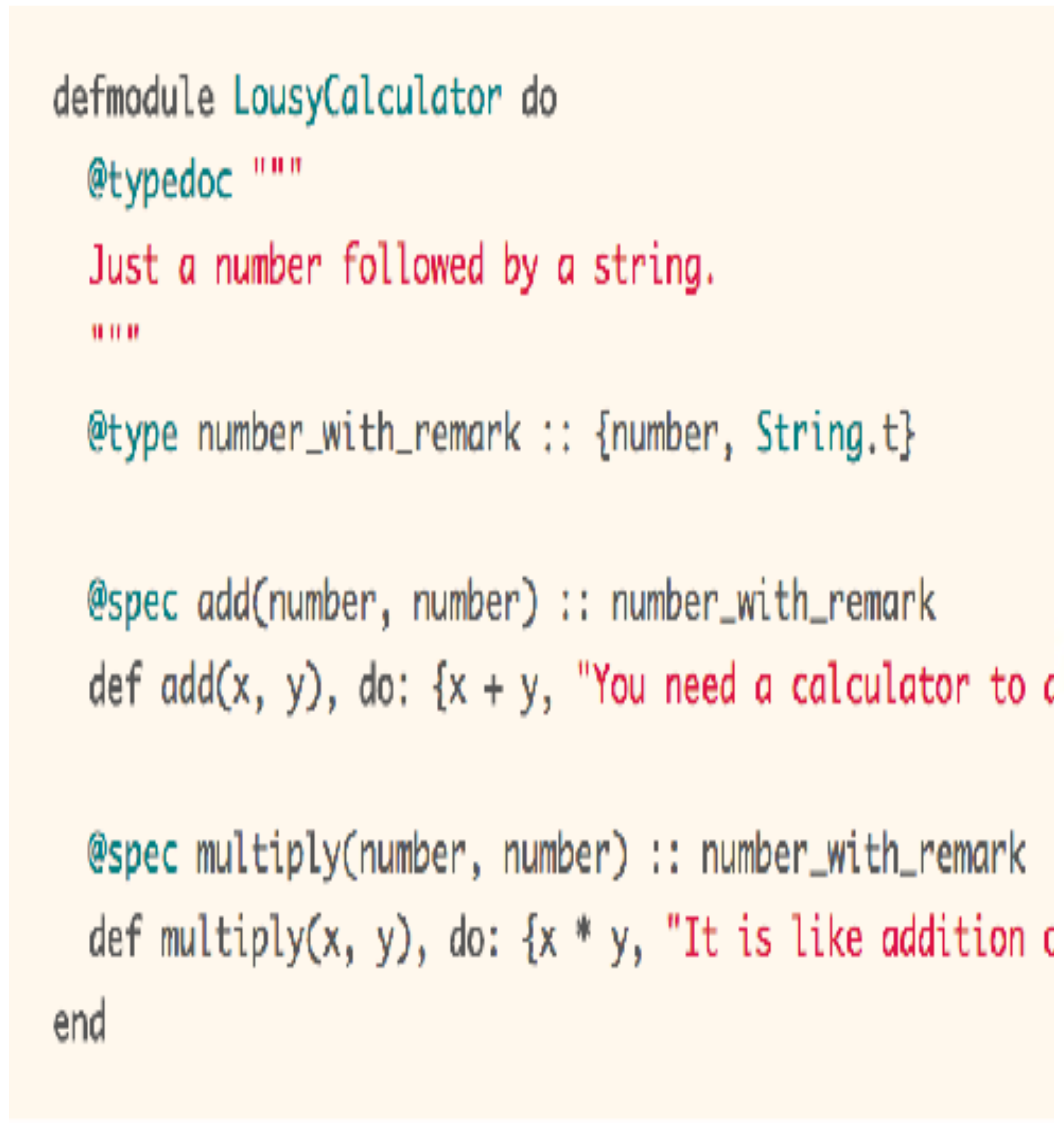




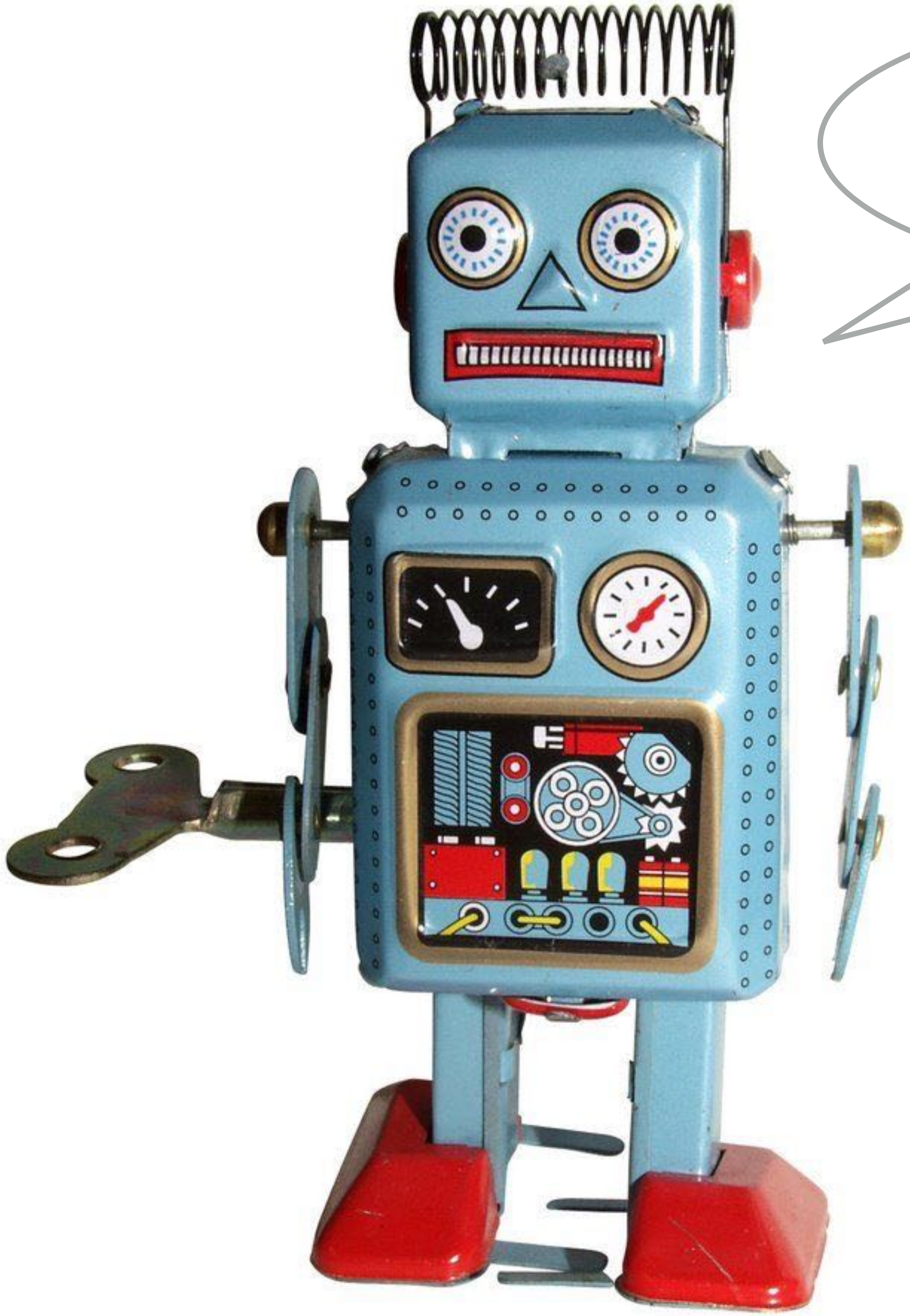
```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """
  @type number_with_remark :: {number, String.t}

  @spec add(number, number) :: number_with_remark
  def add(x, y), do: {x + y, "You need a calculator to d

  @spec multiply(number, number) :: number_with_remark
  def multiply(x, y), do: {x * y, "It is like addition o
end
```




```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """
  @type number_with_remark :: {number, String.t}

  @spec add(number, number) :: number_with_remark
  def add(x, y), do: {x + y, "You need a calculator to d

  @spec multiply(number, number) :: number_with_remark
  def multiply(x, y), do: {x * y, "It is like addition o
end
```




```
defmodule LousyCalculator do
  @typedoc """
  Just a number followed by a string.
  """

  @type number_with_remark :: {number, String.t}


  @spec add(number, number) :: number_with_remark
  def add(x, y), do: {x + y, "You need a calculator to c


  @spec multiply(number, number) :: number_with_remark
  def multiply(x, y), do: {x * y, "It is like addition c
end
```



Just like with everything else you do.

An Elixir Safety Harness:

Dialyzer

Rob Howard
@damncabbage
http://robhoward.id.au

Name

Name

Foo

Bar

Yurp

Traffic-Light Colour

Traffic-Light Colour

Red

Yellow Green

Boolean

Boolean

`True` `False`

Integer

Integer

24601

7

1

-5

String

```
""
```

String

```
"hello world"
```

```
"This is a
sentence."
```

String

""

"hello world"

"This is a
sentence."

```
"If music be the food of love, play on
Give me excess of it, that, surfeiting
The appetite may sicken, and so die.
That strain again! it had a dying fall
O, it came o'er my ear like the sweet
That breathes upon a bank of violets,"
Stealing and giving odour! Enough; no
'Tis not so sweet now as it was before
O spirit of love! how quick and fresh
That, notwithstanding thy capacity
Receiveth as the sea, nought enters th
Of what validity and pitch soe'er,
```

""

String

"hello world"

"This is a sentence."

"如果音乐是爱的食物，那就玩吧；
给我多余的，即，
食欲可能会让人生病，所以死亡。
再次感染！ 它有一个垂死的秋天：
哦，它像我甜美的声音一样出现在我
那呼吸在紫罗兰银行，“偷窃和给予气
“现在不像以前那么甜。
爱的精神！ 你是多么快速和新鲜的艺
尽管你有能力
作为海洋，无所不在，

Elixir / Erlang

Types

Boring Types

`boolean()`

`true` `false`

`integer()`

24601

7

1

-5

atom()

:jessie

:frank

:red

:x

Functions

```
(integer()) :: integer()
```

```
(integer()) :: integer()
```

```
(integer()) :: integer()
```

```
(integer()) :: integer()
```

(integer()) :: integer()

```
def double(number) do
  number * 2
end
```

```
(integer()) :: integer()
```

```
def double(number) do
  number * 2
end
```

```
(integer()) :: integer()
```

```
def double(number) do
  number * 2
end
```

```
(integer()) :: integer()
```

```
def double(number) do
  number * 2
end
```

(integer()) :: integer()

```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

Why?

(integer()) :: integer()

```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```elixir
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```

```elixir
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  double(num)
end)
# => [2, 6000, 4]
```

```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

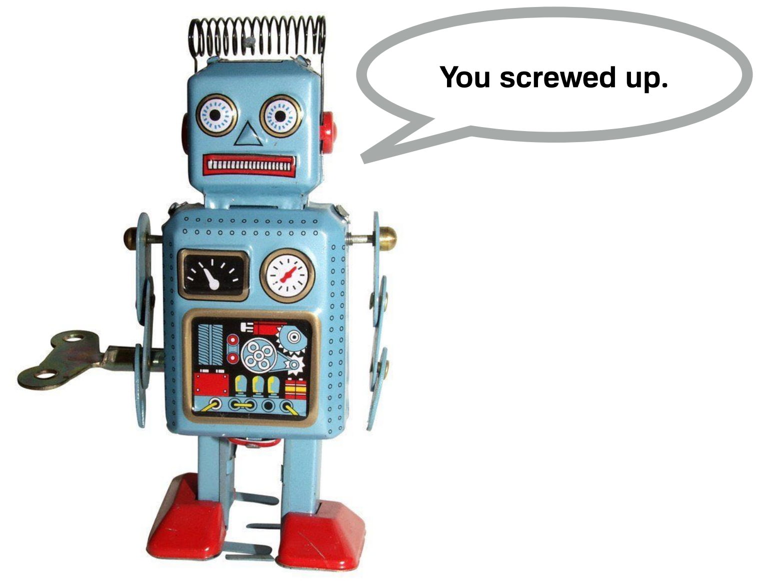
```elixir
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```

```elixir
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```elixir
numbers = [1, 3000, 2, :kaboom]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```

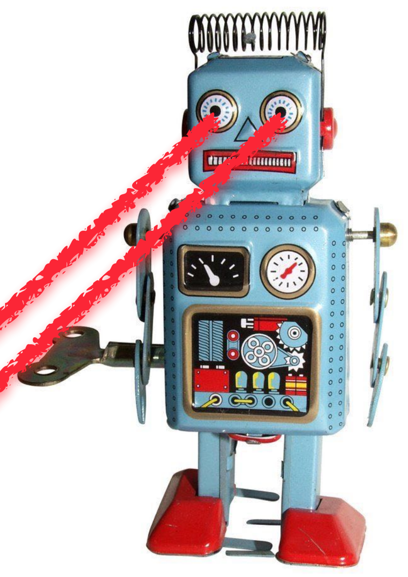
```elixir
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2, :kaboom]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```



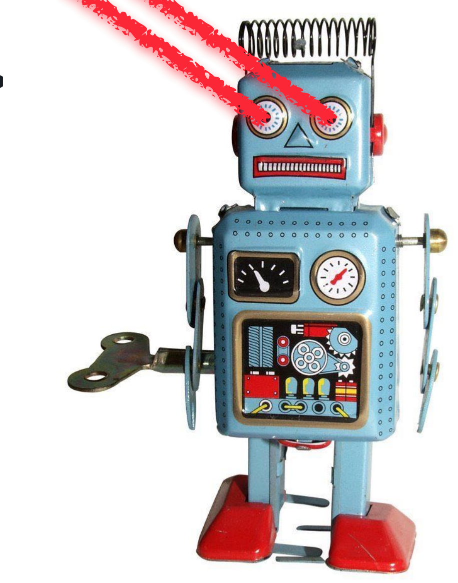
```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2, :kaboom]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```



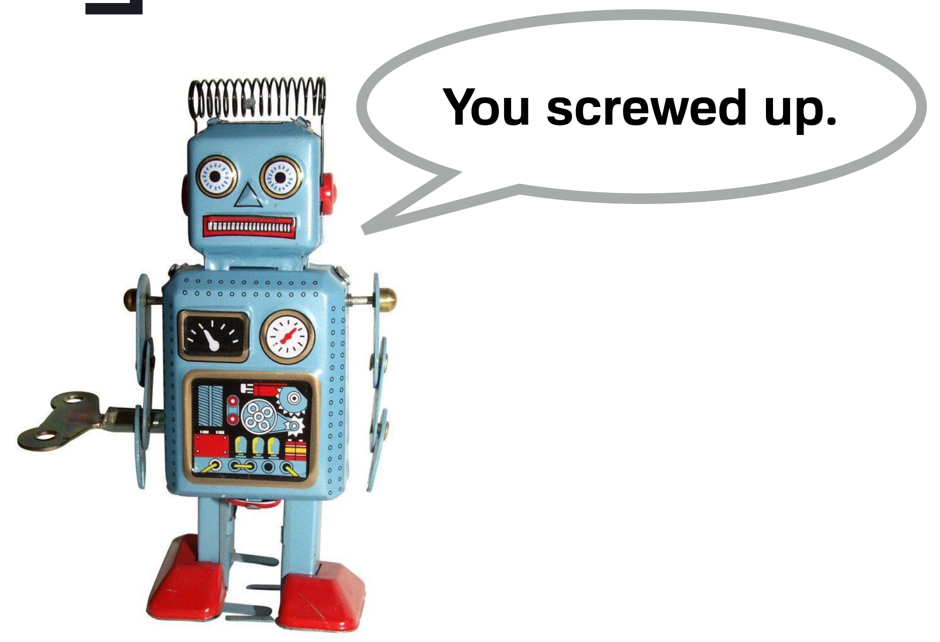
```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2, :kaboom]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```



```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2, :kaboom]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```



```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2, :kaboom]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```

```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2, :kaboom]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```

```
@spec double(integer()) :: integer()
def double(number) do
  number * 2
end
```

```
** (ArithmeticError) bad argument
   in arithmetic expression
```

i am aware this seems silly

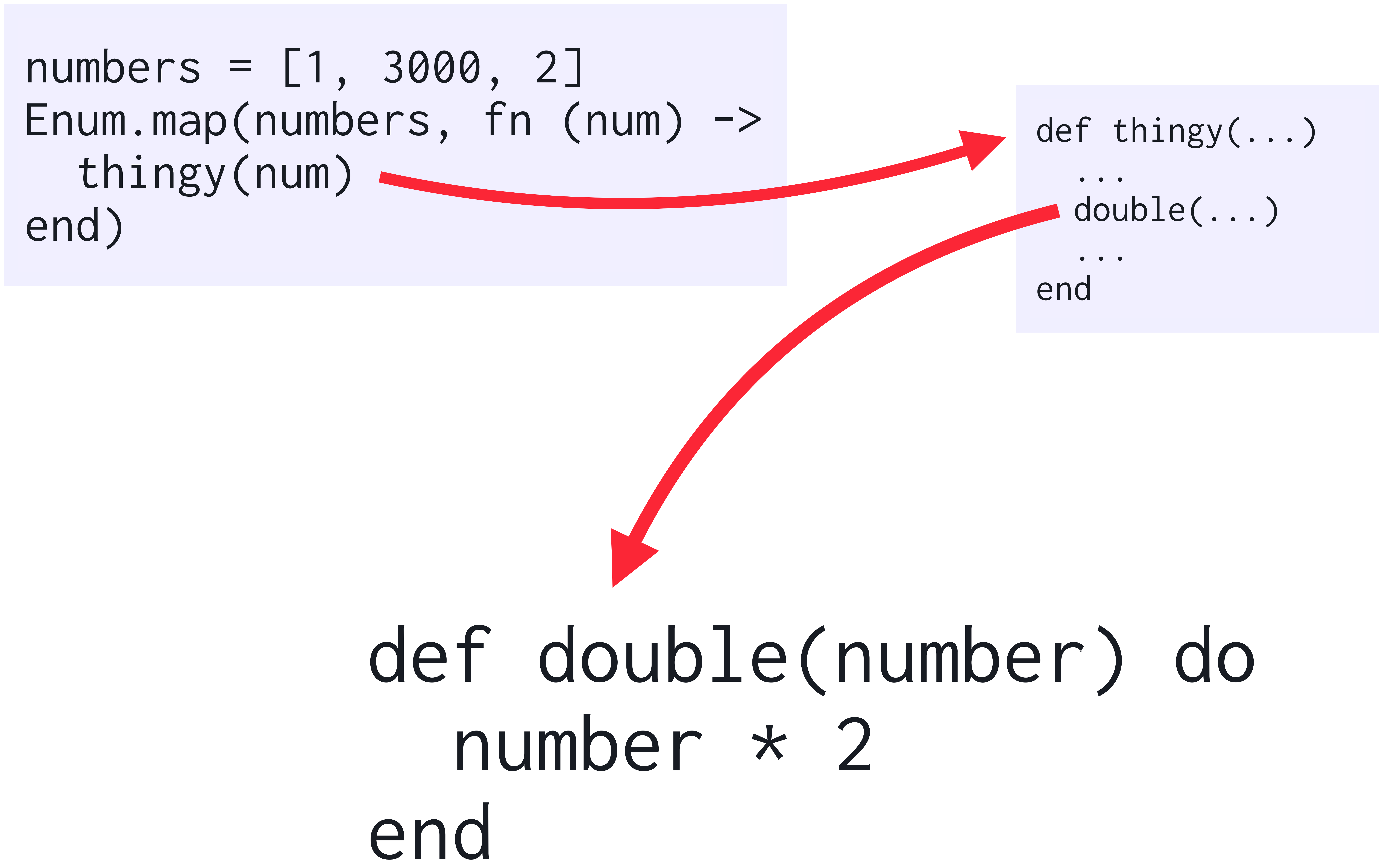
Keeping Focus

```
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  double(num)
end)
```

$$\big\downarrow$$

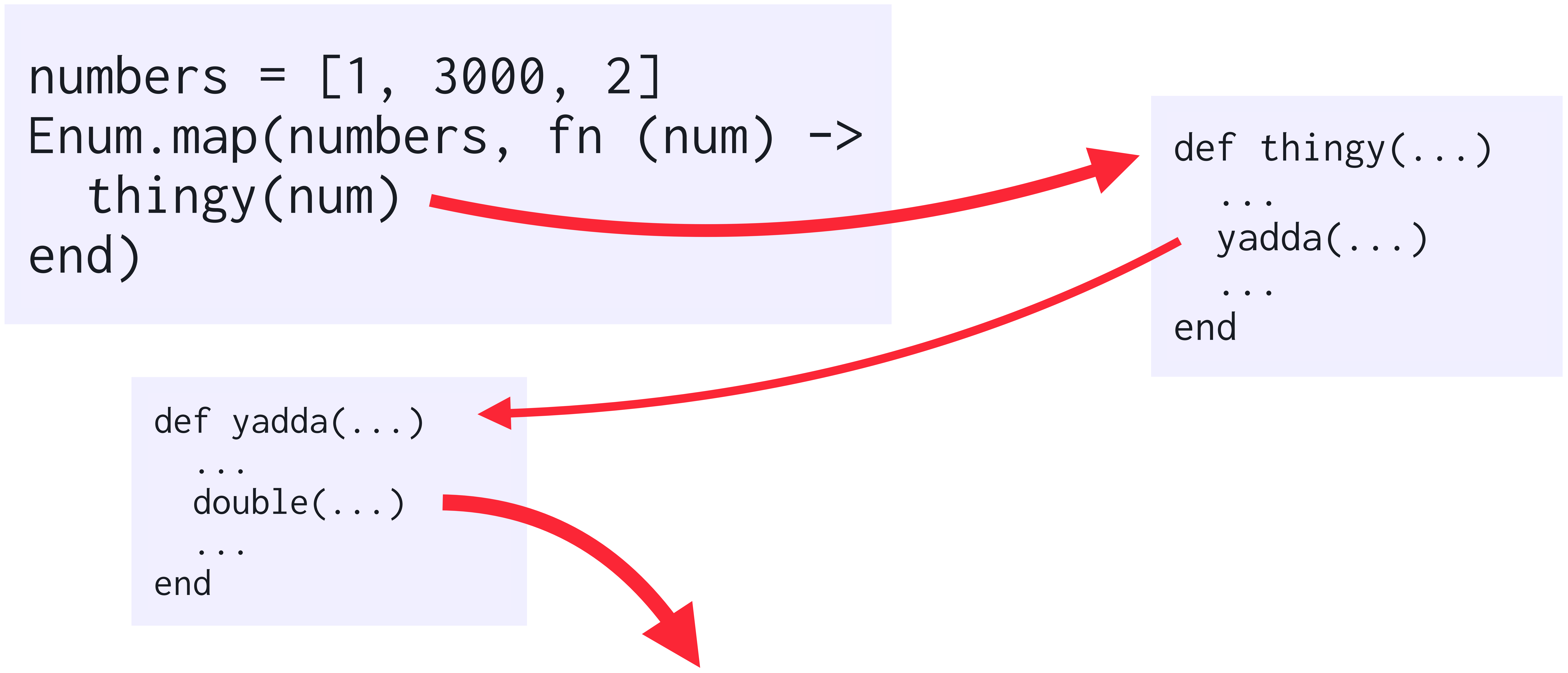
```
def double(number) do
  number * 2
end
```



```
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  thingy(num)
end)
```

```
def thingy(...)
  ...
  double(...)
  ...
end
```

```
def double(number) do
  number * 2
end
```



```
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  thingy(num)
end)
```

```
def thingy(...)
  ...
  yadda(...)
  ...
end
```

```
def yadda(...)
  ...
  double(...)
  ...
end
```

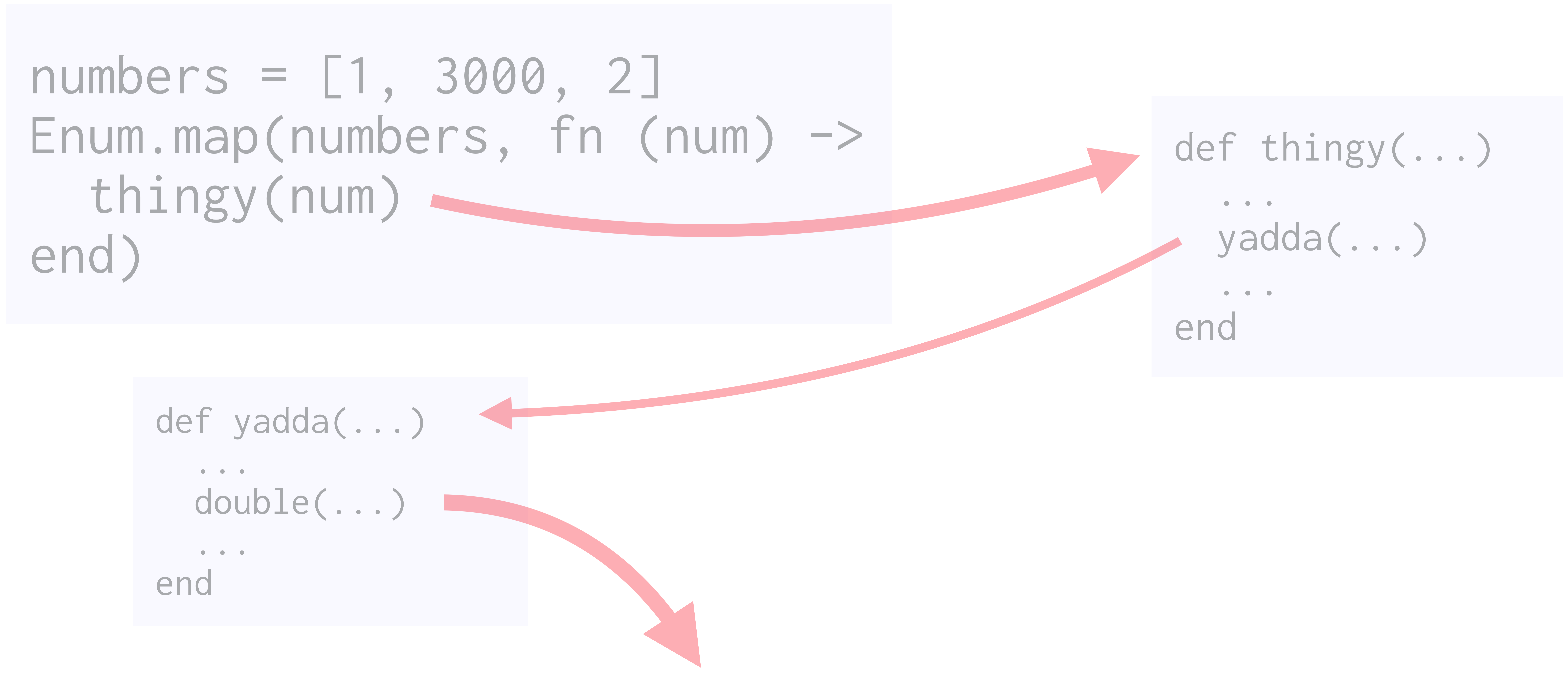
```
def double(number) do
  number * 2
end
```

```
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  thingy(num)
end)
```

```
def thingy(...)
  ...
  yadda(...)
  ...
end
```

```
def yadda(...)
  ...
  double(...)
  ...
end
```

```
def double(number) do
  number * 2
end
```



```
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  thingy(num)
end)
```

```
def thingy(...)
  ...
  yadda(...)
  ...
end
```

```
def yadda(...)
  ...
  double(...)
  ...
end
```

```
def double(number) do
  number * 2
end
```

MORE
Elixir / Erlang
Types

`integer()`

24601

7

1

-5

Compositions

(Things nested inside other things.)

```
{}
(empty tuple)
```

```
{}
```

```
{}
(empty tuple)
```

```
{}
```

im so alone

```
{atom(), integer()}
```

`{atom(), integer()}`

```
{:blue, 1}

{:red, 3}

{:green, 3}
```

Unions

(Things made up of other things.)

```
integer() | boolean()
```

integer() | boolean()

true

53

false

-350

7

```
integer() | boolean()
```

```
    true            53
                          false
-350          7
```

integer() | boolean()

true

53

false

-350

7

Literals

:red

(only :red, that's it)

:red

:red

(only :red, that's it)

:red

im so alone

```
:red | :yellow | :green
```

`:red | :yellow | :green`

`:red`

`:green`

`:yellow`

Let's Put It Together

```
  {:ok, integer()}
| {:error, atom()}
```

```
{:ok, integer()}
| {:error, atom()}
```

```
{:ok, 5}          {:error,
                   :not_found}

{:ok, 0}          {:error,
                   :permission_denied}
```

```
{:ok, integer()}
| {:error, atom()}
```

```
{:ok, 5}          {:error,
                   :not_found}

{:ok, 0}          {:error,
                   :permission_denied}
```

```
{:ok, integer()}
| {:error, atom()}
```

```
{:ok, 5}          {:error,
                    :not_found}

{:ok, 0}          {:error,
                    :permission_denied}
```

```
{:ok, integer()}
| {:error, atom()}
```

```
{:ok, 5}        {:error,
                 :not_found}

{:ok, 0}     {:error,
               :permission_denied}
```

```
  {:ok, integer()}
| {:error, atom()}
```

```
  {:ok, integer()}
| {:error, (
       :not_found
     | :permission_denied
   )}
   {:ok, integer()}
 | {:error, (
         :not_found
       | :permission_denied
     )}
```

```
{:ok, 5}
{:ok, 0}
{:error,
 :not_found}
{:error,
 :permission_denied}
```

Odds & Ends

`list(...)`

```
list(boolean())
```

```
                        []
[false, true]
     [true, false, false]
```

```
list(integer())

                              []
  [1, 2, 3]
        [1,2,3,4,5,6,7,-7]
```

`map()`

map()

```
%{:foo => 123}
%{"abc" => true}
%{1 => :red, 2 => :blue}
```

```
%{:age => integer()}
```

```
%{:age => integer()}
```

```
%{:age => 30}
%{:age => 1}
%{:name => "Frank", :age => 82}
```

```
%{:age => integer()}
```

```
%{:age => 1}
```

```
%{:age => 30}
```

```
%{:name => "Frank", :age => 82}
```

any( )

```
any()
```

<literally anything at all goes here>

```
String.t()
```

""

String.t()

"hello world"

"This is a
sentence."

"如果音乐是爱的食物，那就玩吧；
给我多余的，即，
食欲可能会让人生病，所以死亡。
再次感染！ 它有一个垂死的秋天：
哦， 它像我甜美的声音一样出现在我的耳
那呼吸在紫罗兰银行，“偷窃和给予气味
“现在不像以前那么甜。
爱的精神！ 你是多么快速和新鲜的艺术

String.t()

""

"hello world"

"This is a sentence."

"如果音乐是爱的食物，那就玩吧；
给我多余的，即，
食欲可能会让人生病，所以死亡。
再次感染！ 它有一个垂死的秋天：
哦，它像我甜美的声音一样出现在我的耳
那呼吸在紫罗兰银行，“偷窃和给予气味
“现在不像以前那么甜。
爱的精神！ 你是多么快速和新鲜的艺术

```
String.t()
integer()
atom()
boolean()
```

```
String.t()
integer()
atom()
boolean()
```

```
String.t
integer
atom
boolean
```

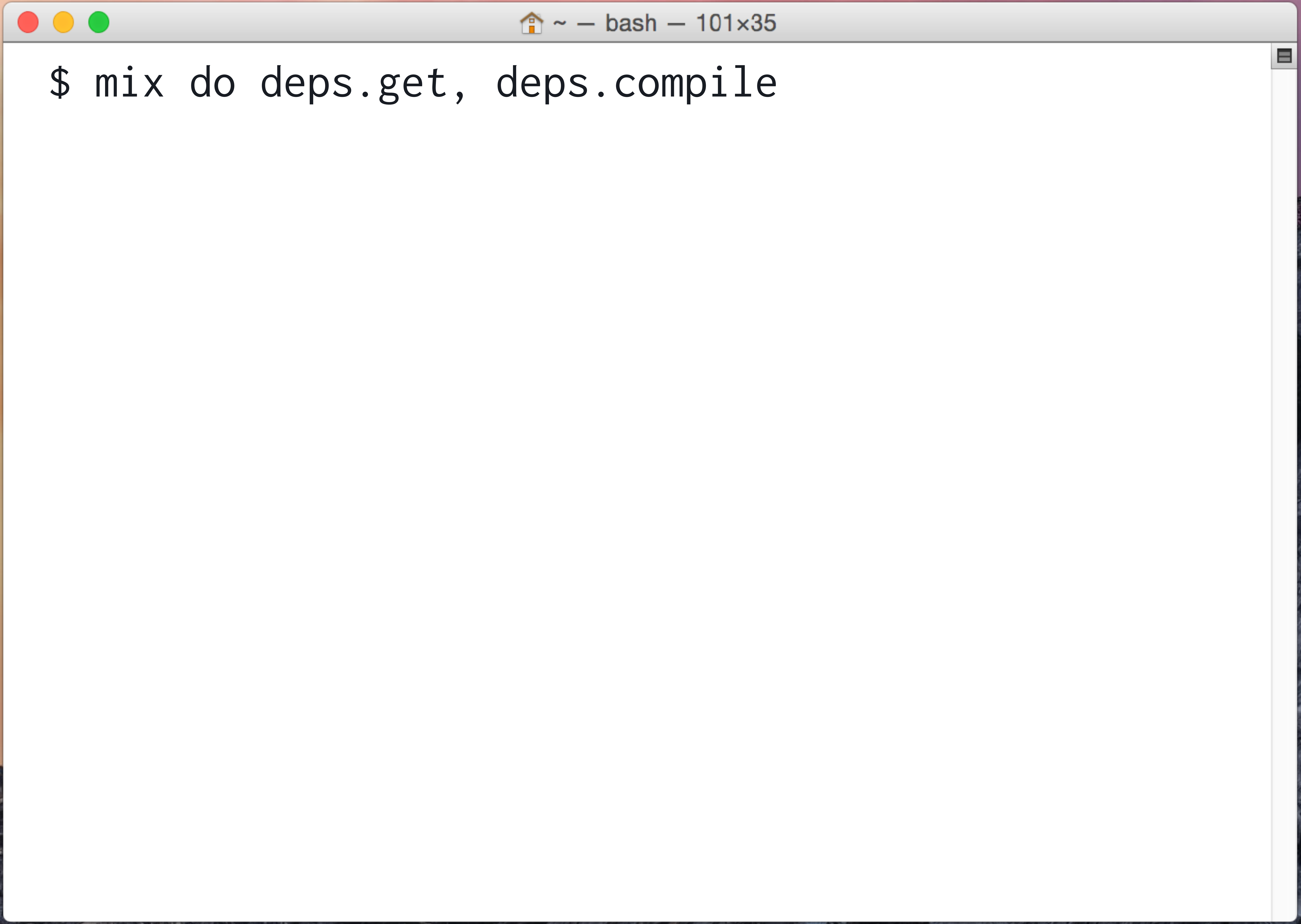
```
String.t()
integer()
atom()
boolean()
```

```
type :: any()                   # the top type, the set of all terms
      | none()                  # the bottom type, contains no terms
      | atom()
      | map()                   # any map
      | pid()                   # process identifier
      | port()
      | reference()
      | struct()                # any struct
      | tuple()                 # tuple of any size

                                ## Numbers
      | float()
      | integer()
      | neg_integer()           # ..., -3, -2, -1
      | non_neg_integer()       # 0, 1, 2, 3, ...
      | pos_integer()           # 1, 2, 3, ...

                                                    ## Lists
      | list(type)                                  # proper list ([]-terminated)
      | nonempty_list(type)                         # non-empty proper list
```

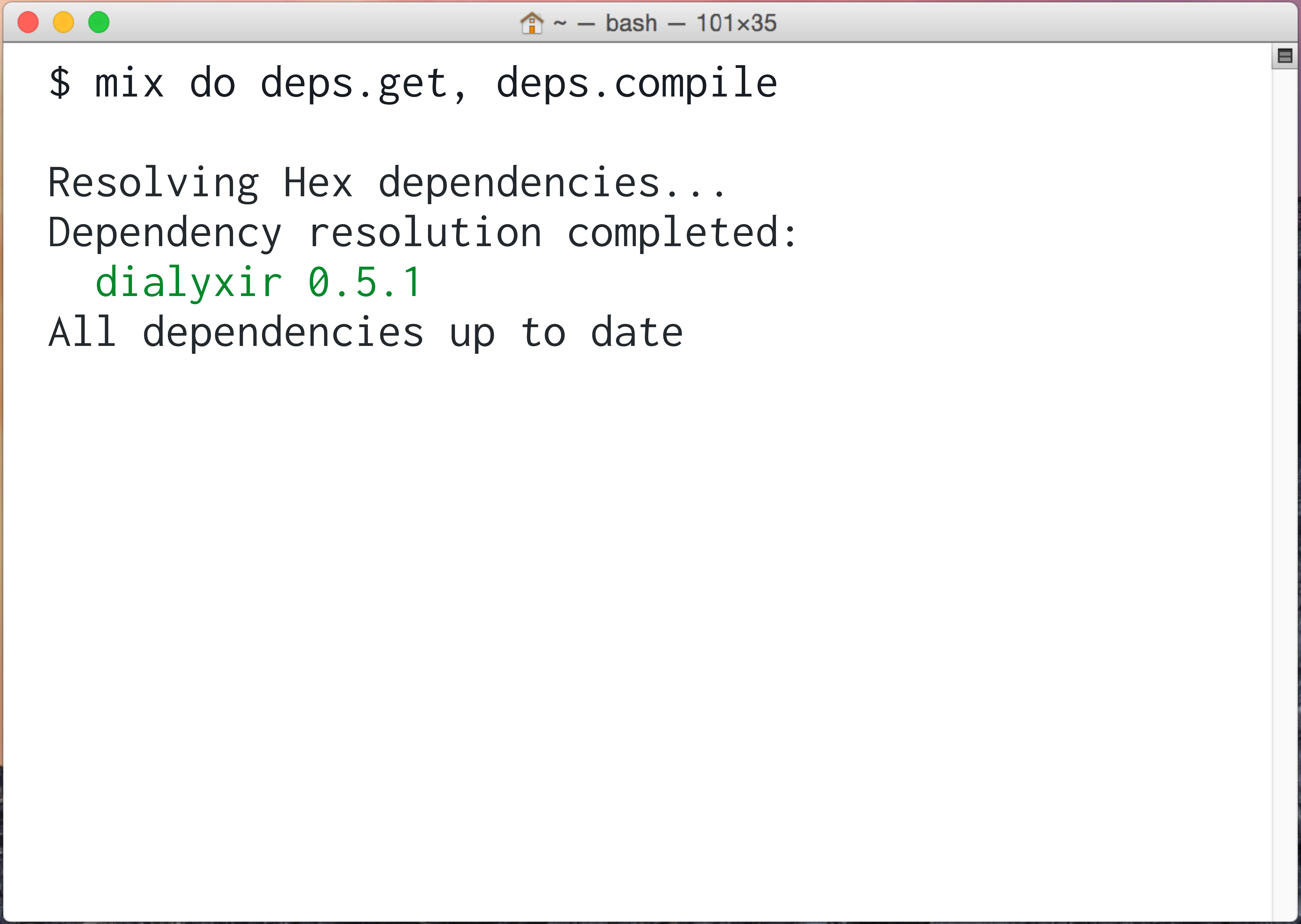
Dialyzer

```elixir
# mix.exs
defmodule HelloWorld.MixProject do
  use Mix.Project

  # ...

  defp deps do
    [
      {:dialyxir, "~> 0.4", only: [:dev]}
    ]
  end
end
```

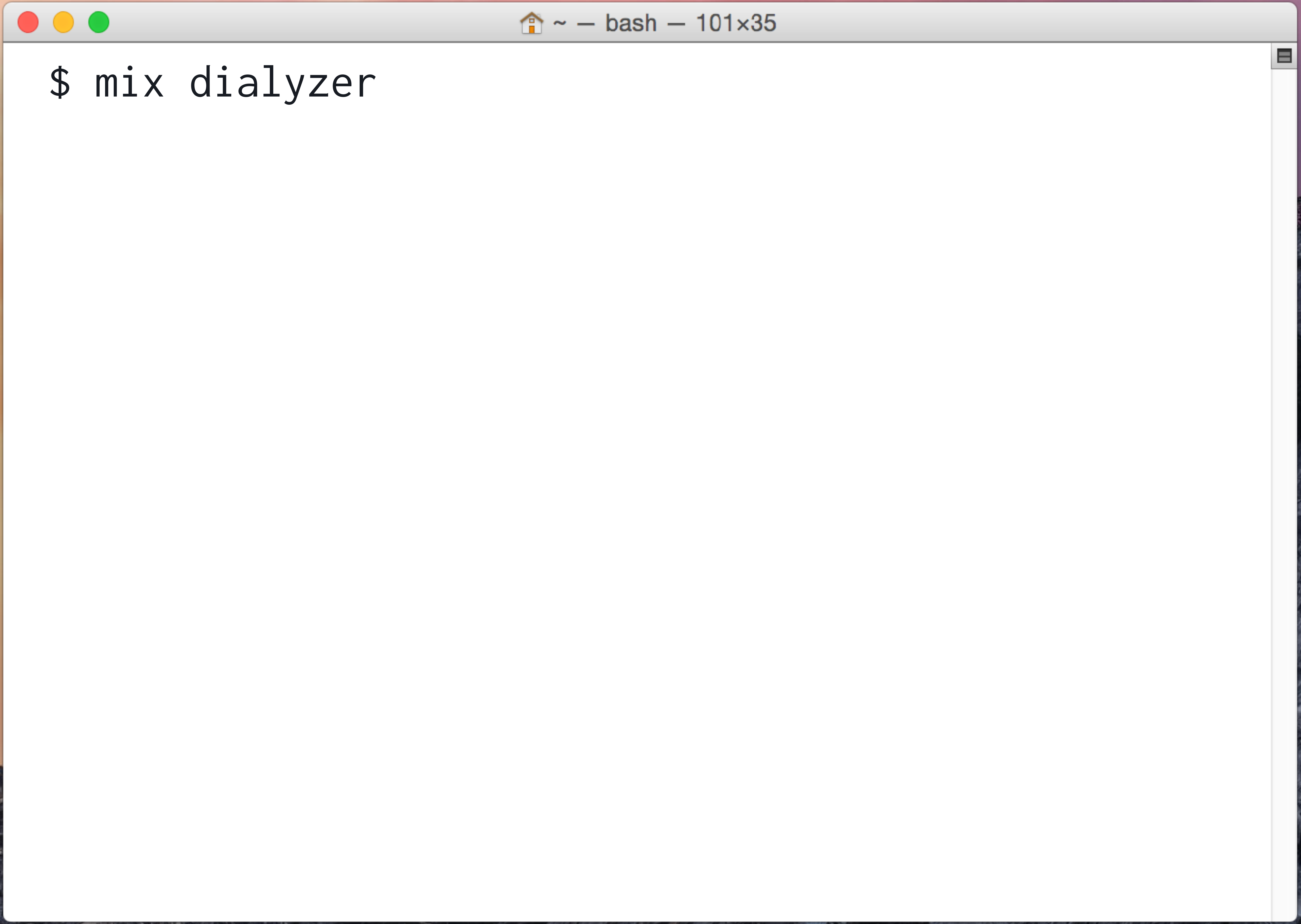
```elixir
# mix.exs
defmodule HelloWorld.MixProject do
  use Mix.Project

  # ...

  defp deps do
    [
      {:dialyxir, "~> 0.4", only: [:dev]}
    ]
  end
end
```



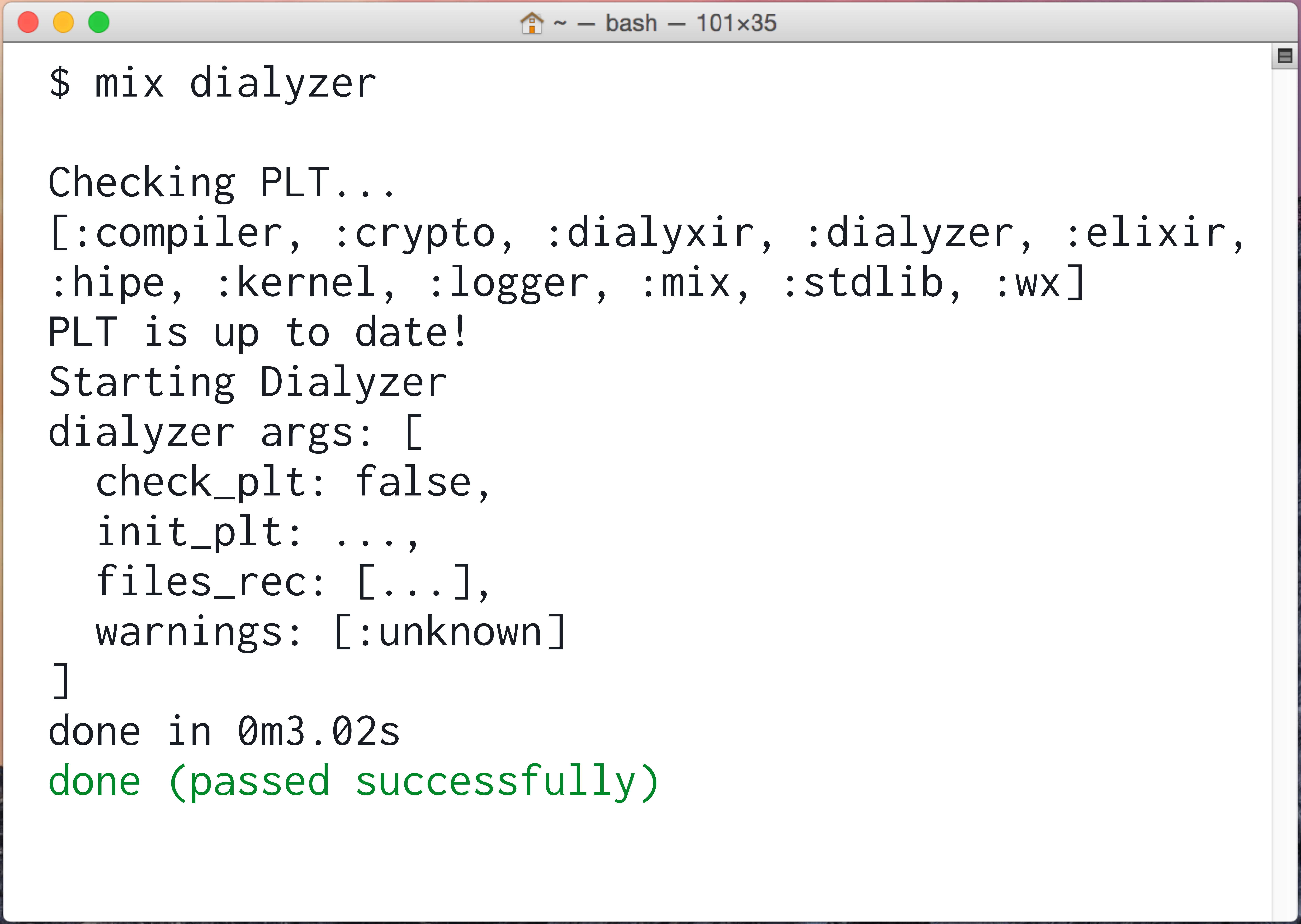
```
$ mix do deps.get, deps.compile
```



```
$ mix do deps.get, deps.compile

Resolving Hex dependencies...
Dependency resolution completed:
  dialyxir 0.5.1
All dependencies up to date
```



```
$ mix dialyzer
```



```
$ mix dialyzer

Checking PLT...
[:compiler, :crypto, :dialyxir, :dialyzer, :elixir,
:hipe, :kernel, :logger, :mix, :stdlib, :wx]
PLT is up to date!
Starting Dialyzer
dialyzer args: [
  check_plt: false,
  init_plt: ...,
  files_rec: [...],
  warnings: [:unknown]
]
done in 0m3.02s
done (passed successfully)
```

```elixir
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  @spec hello_world() :: String.t
  def hello_world() do
    greet("world")
  end


end
```

```elixir
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  @spec hello_world() :: String.t
  def hello_world() do
    greet("world")
  end


end
```

```
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  @spec hello_world() :: String.t
  def hello_world() do
    greet("world")
  end


end
```

```elixir
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  @spec hello_world() :: String.t
  def hello_world() do
    greet("world")
  end


end
```

```elixir
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  @spec hello_world() :: String.t
  def hello_world() do
    greet("world")
  end


end
```

```elixir
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  @spec hello_world() :: String.t
  def hello_world() do
    greet("world")
  end


end
```

```
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  @spec hello_world() :: String.t
  def hello_world() do
    greet("world")
  end


end
```

```elixir
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet("Martin"), do: "Thanks for ElixirCamp!"
  def greet("Martin2"), do: "Thanks for GenServers!"
  def greet("Mel"), do: "Thanks for RubyConf AU!"
  def greet("Richard"), do: "not you"
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  # ...

end
```

```
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet("Martin"), do: "Thanks for ElixirCamp!"
  def greet("Martin2"), do: "Thanks for GenServers!"
  def greet("Mel"), do: "Thanks for RubyConf AU!"
  def greet("Richard"), do: "not you"
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  # ...

end
```

```elixir
defmodule Messages do

  @spec greet(String.t) :: String.t
  def greet("Martin"), do: "Thanks for ElixirCamp!"
  def greet("Martin2"), do: "Thanks for GenServers!"
  def greet("Mel"), do: "Thanks for RubyConf AU!"
  def greet("Richard"), do: "not you"
  def greet(to_whom) do
    "Hello, #{to_whom}!"
  end

  # ...

end
```

Reuse

```elixir
defmodule FileUtils do









  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do




  @spec lines(filename: String.t) :: ...



  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do






  @spec lines(filename: String.t) :: (
      {:ok, integer()}
    # ...
  )
  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do





  @spec lines(filename: String.t) :: (
      {:ok, integer()}
    | {:error, (:not_found | :permission_denied)}
  )
  def lines(filename) do
    # ...
  end
end
```

```
defmodule FileUtils do




  @spec lines(filename: String.t) :: (
      {:ok, integer()}
    | {:error, (:not_found | :permission_denied)}
  )
  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do




  @type file_errors :: :not_found | :permission_denied
  @spec lines(filename: String.t) :: (
      {:ok, integer()}
    | {:error, file_errors}
  )
  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do




  @type file_errors :: :not_found | :permission_denied
  @spec lines(filename: String.t) :: (
      {:ok, integer()}
    | {:error, file_errors}
  )
  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do




  @type file_errors :: :not_found | :permission_denied
  @spec lines(filename: String.t) :: (
      {:ok, integer()}
    | {:error, file_errors}
  )
  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do
  @type result(good, error) :: (
      {:ok, good}
    | {:error, error}
  )
  @type file_errors :: :not_found | :permission_denied

  @spec lines(filename: String.t) :: (
    result(integer(), file_errors)
  )
  def lines(filename) do
    # ...
  end
end
```

```elixir
defmodule FileUtils do
  @type result(good, error) :: (
      {:ok, good}
    | {:error, error}
  )
  @type file_errors :: :not_found | :permission_denied

  @spec lines(filename: String.t) :: (
    result(integer(), file_errors)
  )
  def lines(filename) do
    # ...
  end
end
```

A Quick Aside

```elixir
defmodule Car do
  @type car :: ...

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end
```

```elixir
defmodule Car do
  @type car :: ...

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end

defmodule Driver do
  @spec use_car(car :: Car.car) :: any()
  # ...
end
```

```elixir
defmodule Car do
  @type car :: ...

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end

defmodule Driver do
  @spec use_car(car :: Car.car) :: any()
  # ...
end
```

```
defmodule Car do
  @type car :: {number(), %{:state => map()}, ...}

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end

defmodule Driver do
  @spec use_car(car :: Car.car) :: any()
  # ...
end
```

```elixir
defmodule Car do
  @type car :: ...

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end

defmodule Driver do
  @spec use_car(car :: Car.car) :: any()
  # ...
end
```

```elixir
defmodule Car do
  @opaque car :: ...

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end

defmodule Driver do
  @spec use_car(car :: Car.car) :: any()
  # ...
end
```

```
defmodule Car do
  @opaque t :: ...

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end

defmodule Driver do
  @spec use_car(car :: Car.t) :: any()
  # ...
end
```

```elixir
defmodule Car do
  @opaque t :: ...

  @spec drive(car) :: boolean()
  def drive(car) do
    # ...
  end

end

defmodule Driver do
  @spec use_car(car :: Car.t) :: any()
  # ...
end
```

```
defmodule String do
  @opaque t :: ...

  @spec length(t) :: integer()
  def length(string) do
    # ...
  end

end

defmodule Greet do
  @spec message(str :: String.t) :: String.t
  # ...
end
```

```
defmodule String do
  @opaque t :: ...

  @spec length(t) :: integer()
  def length(string) do
    # ...
  end

end

defmodule Greet do
  @spec message(str :: String.t) :: String.t
  # ...
end
```

MORE

Why?

Docs

contains?(string, contents)

`contains?(t(), pattern()) :: boolean()`

Checks if `string` contains any of the given `contents`.

`contents` can be either a single string or a list of strings.

Examples

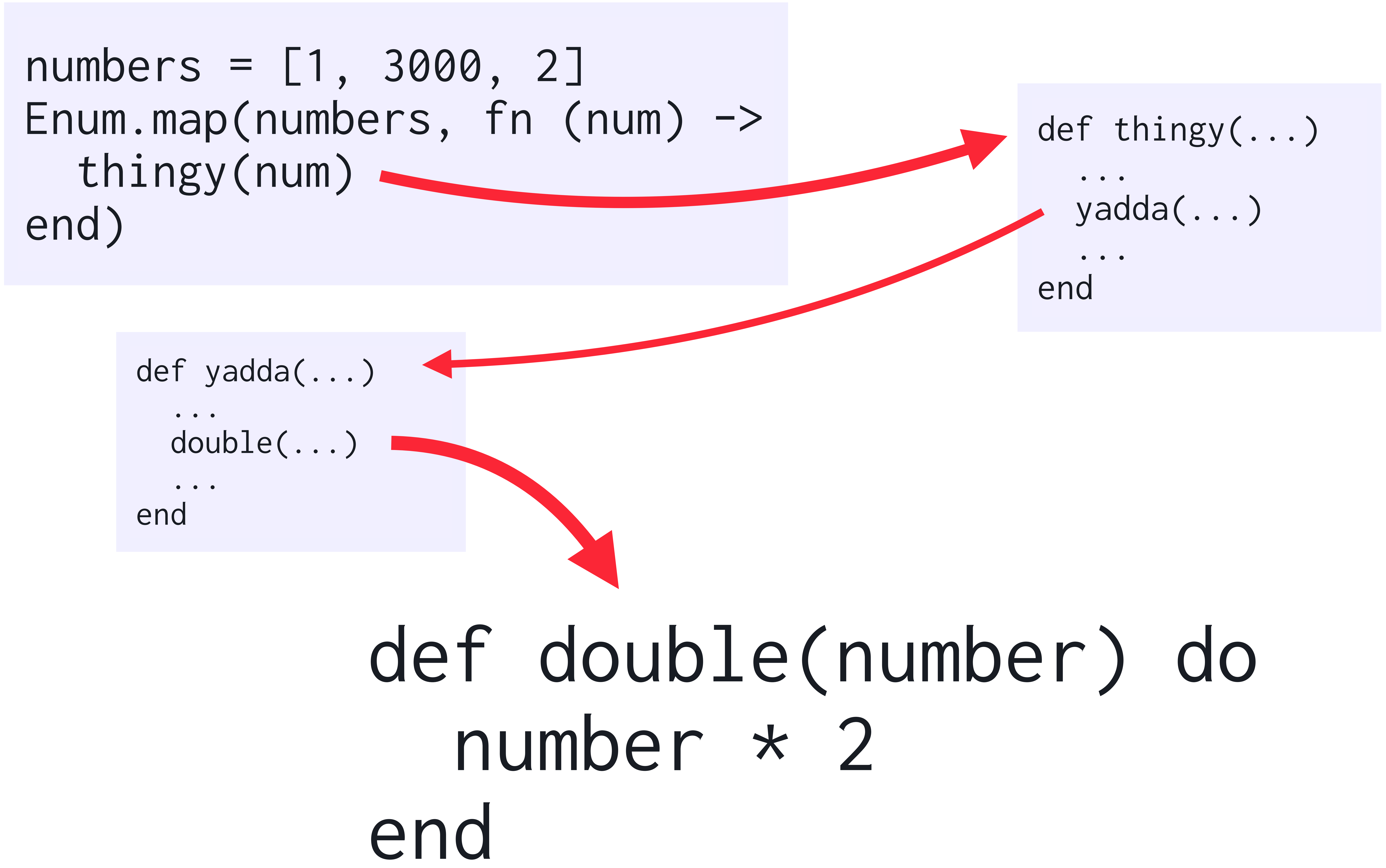
```
iex> String.contains? "elixir of life", "of"
true
iex> String.contains? "elixir of life", ["life", "death"]
true
iex> String.contains? "elixir of life", ["death", "mercury"]
false
```

An empty string will always match:

```
iex> String.contains? "elixir of life", ""
true
iex> String.contains? "elixir of life", ["", "other"]
true
```

The argument can also be a precompiled pattern:

Refactoring



```
numbers = [1, 3000, 2]
Enum.map(numbers, fn (num) ->
  thingy(num)
end)
```

```
def thingy(...)
  ...
  yadda(...)
  ...
end
```

```
def yadda(...)
  ...
  double(...)
  ...
end
```

```
def double(number) do
  number * 2
end
```

(Possibly?) Cheaper Refactoring

(Tests
vs
Type Signatures)

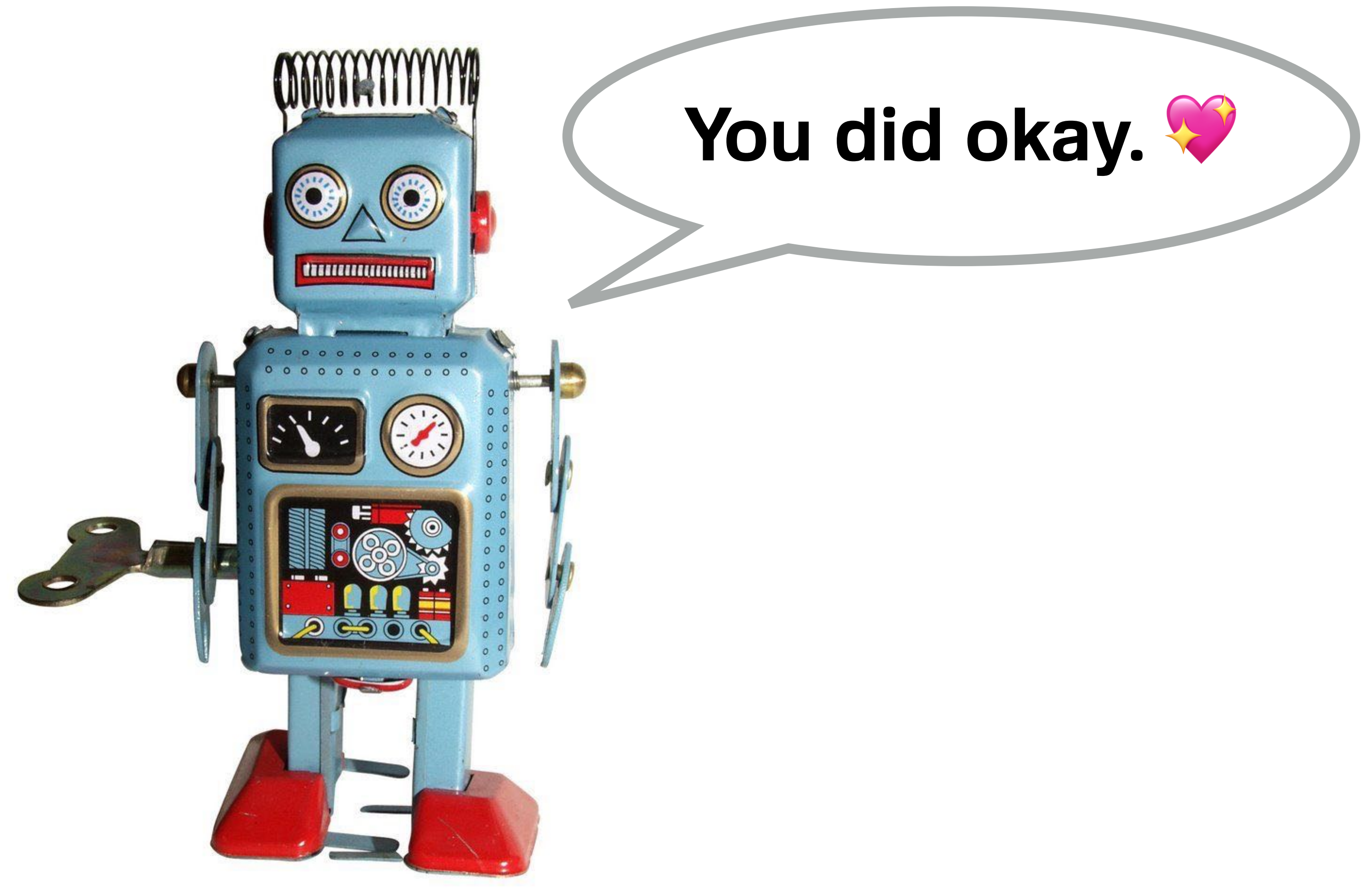
"We have pattern matching"

"Let it Crash"

it's just going to crash again if you give it bad data

try to get it right before you run it

try to get it right before you run it



Why Not?

It's Super-Optimistic

```elixir
defmodule FileUtils do
  @type result(good, error) :: (
      {:ok, good}
    | {:error, error}
 )
  @type file_errors :: :not_found | :permission_denied

  @spec lines(filename :: String.t) :: (
    result(integer(), file_errors)
  )
  def lines(filename) do
    if (:rand.uniform(10)) < 5 do
      {:ok, 123}
    else
      {:error, :nope}
    end
  end
end
```

```elixir
defmodule FileUtils do
  @type result(good, error) :: (
      {:ok, good}
    | {:error, error}
 )
  @type file_errors :: :not_found | :permission_denied

  @spec lines(filename :: String.t) :: (
    result(integer(), file_errors)
  )
  def lines(filename) do
    if (:rand.uniform(10)) < 5 do
      {:ok, 123}
    else
      {:error, :nope}
    end
  end
end
```

```elixir
defmodule FileUtils do
  # ...
  def lines(filename) do
    if (:rand.uniform(10)) < 5 do
      {:ok, 123}
    else
      {:error, :nope}
    end
  end










end
```

```elixir
defmodule FileUtils do
  # ...
  def lines(filename) do
    if (:rand.uniform(10)) < 5 do
      {:ok, 123}
    else
      {:error, :nope}
    end
  end

  def count_foo() do
    case lines("./foo.txt") do
      {:ok, n} ->
        n
      {:error, :not_found} ->
        -1
      {:error, :permission_denied} ->
        -1
  end
end
```

```
defmodule FileUtils do
  # ...
  def lines(filename) do
    if (:rand.uniform(10)) < 5 do
      {:ok, 123}
    else
      {:error, :nope}
    end
  end

  def count_foo() do
    case lines("./foo.txt") do
      {:ok, n} ->
        n
      {:error, :not_found} ->
        -1
      {:error, :permission_denied} ->
        -1
  end
end
```

An Elixir Safety Harness:

Dialyzer

Rob Howard
@damncabbage
http://robhoward.id.au

Elixir TypeSpecs Guide:

- https://hexdocs.pm/elixir/typespecs.html

Dialyzer for Elixir:

- https://github.com/jeremyjh/dialyxir

VSCode ElixirLS:

- https://github.com/JakeBecker/vscode-elixir-ls