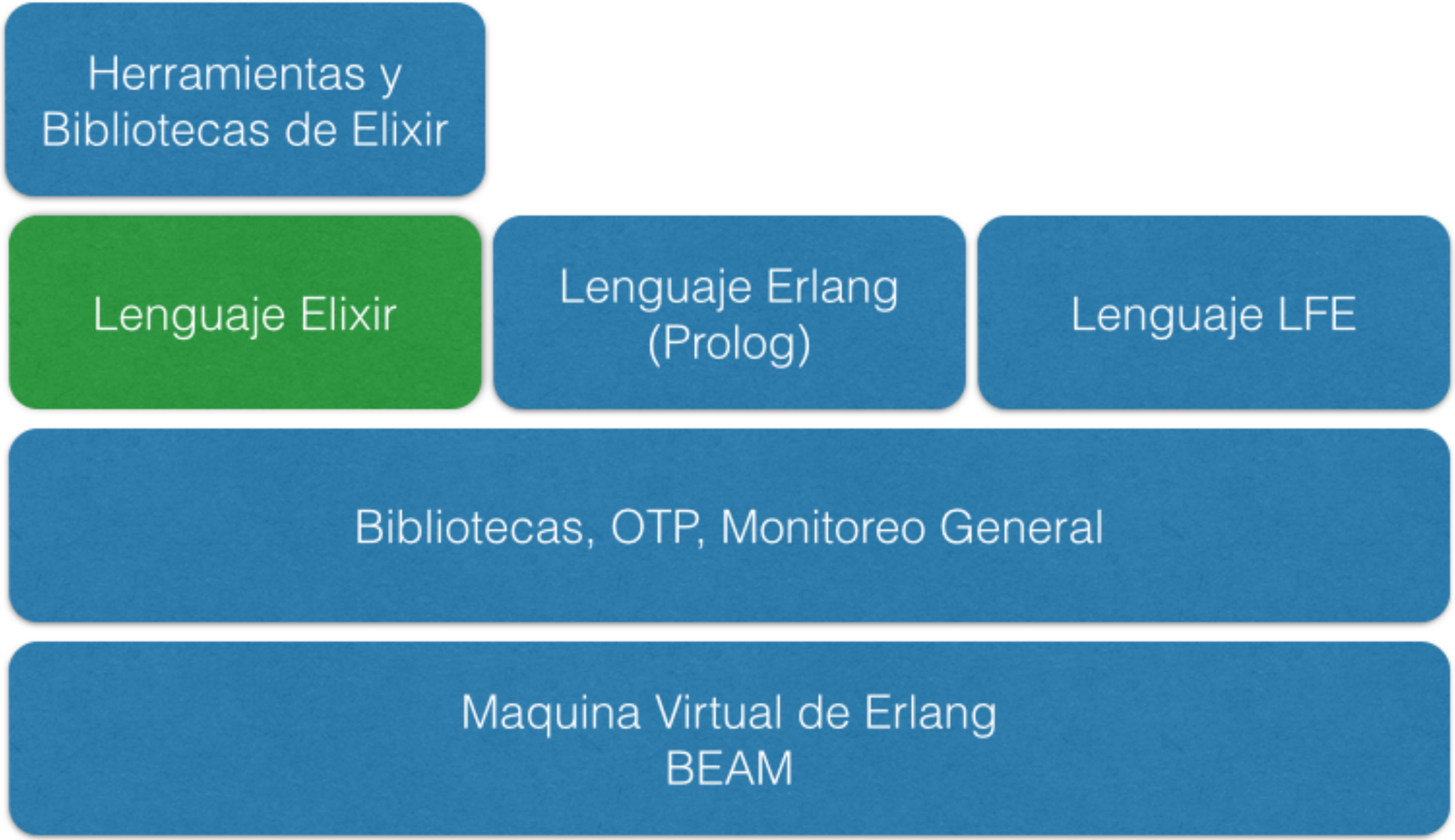
Elixir

Agenda

- History
- Special characteristics
- Language - Part I



Herramientas y Bibliotecas de Elixir

Lenguaje Elixir

Lenguaje Erlang (Prolog)

Lenguaje LFE

Bibliotecas, OTP, Monitoreo General

Maquina Virtual de Erlang BEAM

History

Chronology

- Created by José Valim
- 1st version: January 2011
- Erlang compatible version: August 2012
- Version 1.0: September 2014
- Current version: 1.9.1

Characteristics - 1

- Runs on the Erlang vitual machine
- Dynamic typing with type annotations
- Functional programming
- Supports Unicode (UTF-8)
- Values are immutable

Functional ?!

But… I already know Object Oriented Programming…

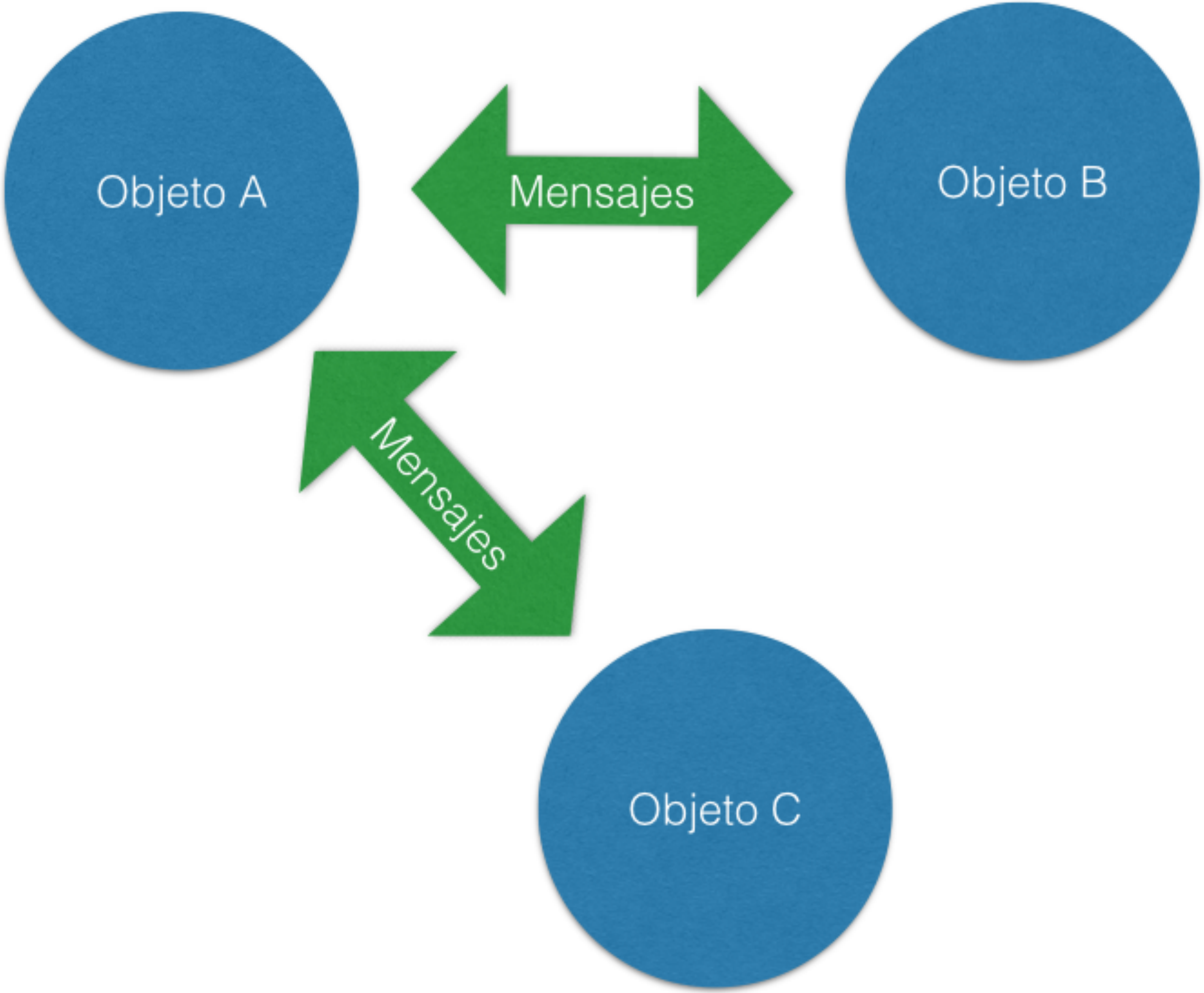
What is Object Oriented Programming?

"OOP to me means only **messaging**, local retention and protection and **hiding of state-process**, and **extreme late-binding** of all things."

— Alan Kay on the Meaning of “Object-Oriented Programming”

“I thought of objects being like biological cells and/or individual computers on a network, only able to communicate with messages.”

— Alan Kay on the Meaning of “Object-Oriented Programming”


Objeto A
Mensajes
Objeto B
Mensajes
Objeto C

What is functional programming?

It means programmin using mathematical functions

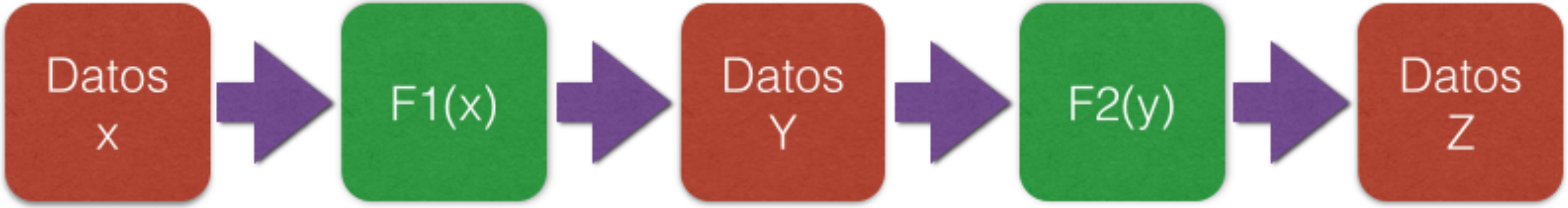
The concept is really simple

- Functions receive and return values

```
f1(int) -> Bool
f1(4) -> true

f2(string) -> string
f2("hola") -> "adios"
```

A mismos valores de entrada, misma salida

Las funciones no alteran los valores de entrada



Promotes a declarative style (what) instead of an imperative one (how)

```
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-2) + fib(N-1).
```

- Using expressions instead of statements
- SQL is another example of declarative vs imperative styles

Functional

- Functions are also a data type
- They can be passed as parameters
- They can be returned by other functions
- They avoid to have secondary effects
- The style of programming is more like a sequence (pipeline?) of transformations on data

First program

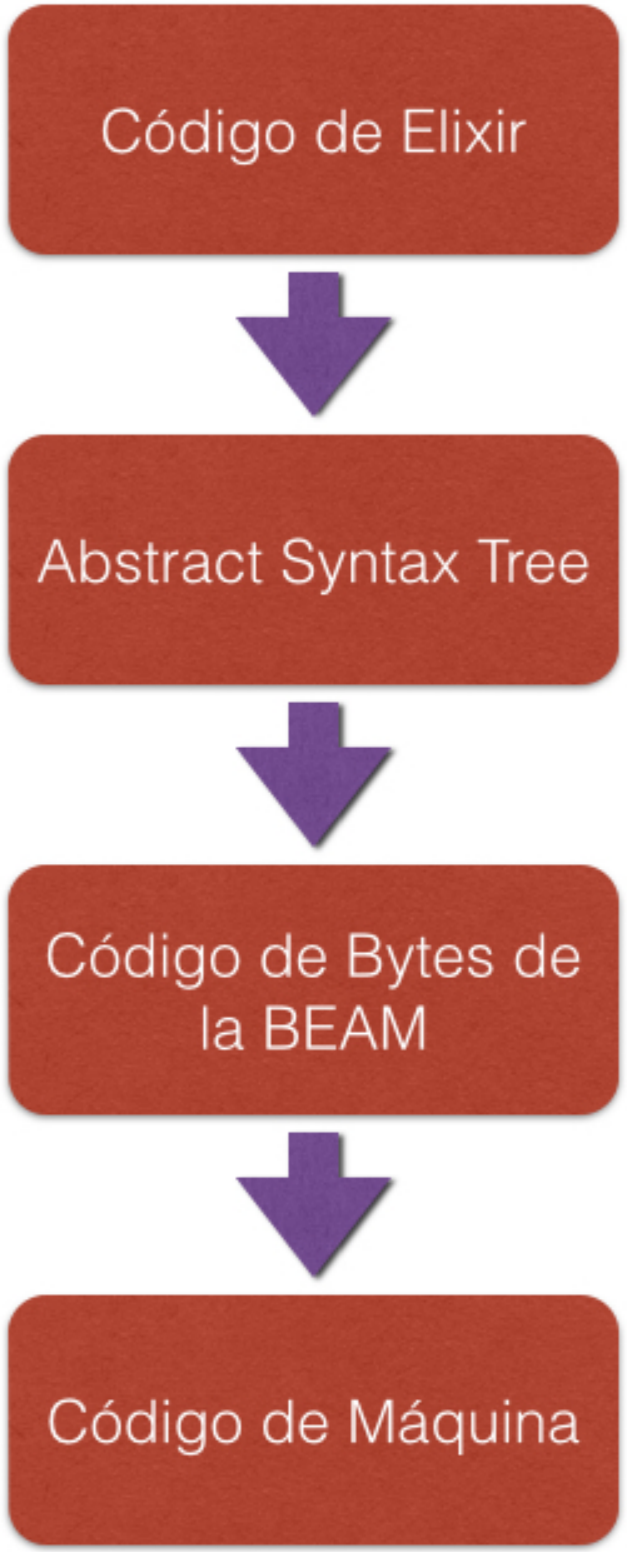
```
$ vim hello_world.exs
```

```
IO.puts "Hello World!"
```

```
$ elixir hello_world.exs
```

Characteristics - 2

- Pattern matching
- Keyword arguments
- Regular expressions
- Hygienic macro system

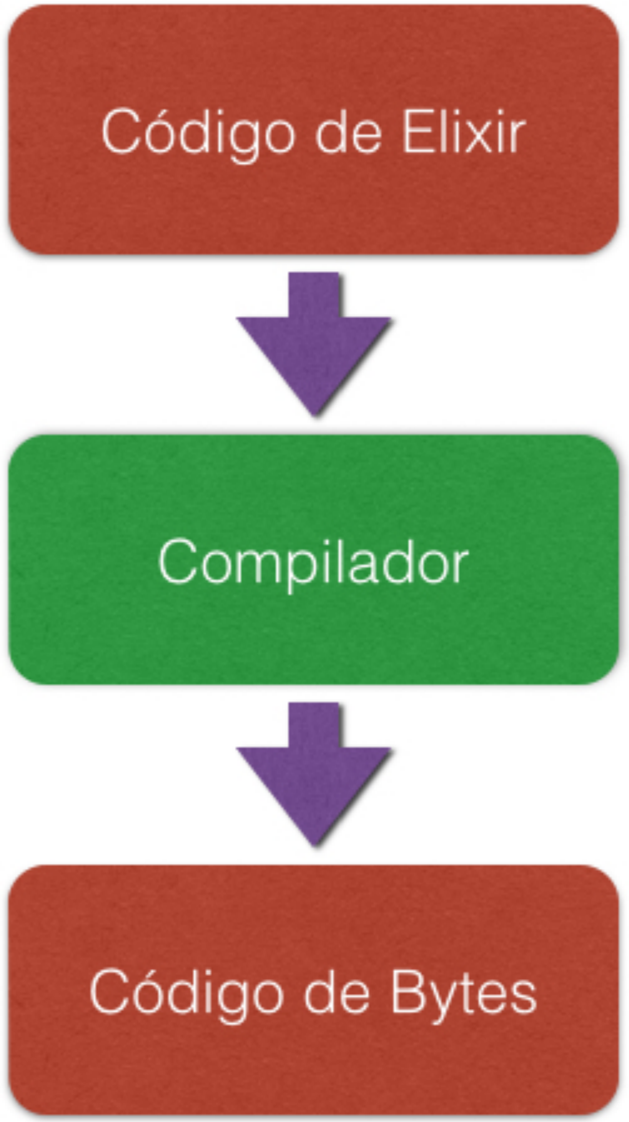


Código de Elixir

Abstract Syntax Tree

Código de Bytes de la BEAM

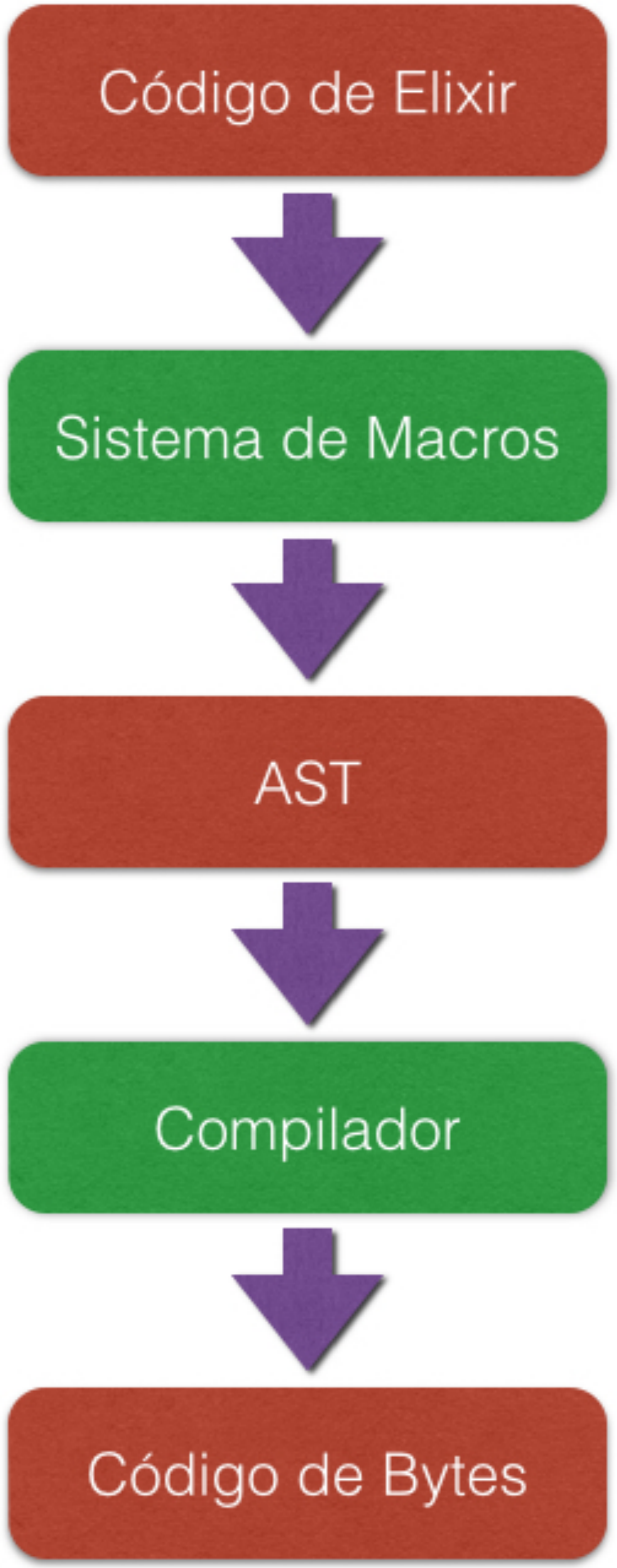
Código de Máquina



Código de Elixir

Compilador

Código de Bytes



Código de Elixir

Sistema de Macros

AST

Compilador

Código de Bytes

Second program

```
defmodule Hello do
  IO.puts "Defining the function world"

  def world do
    IO.puts "Hello World"
  end

  IO.puts "Function world defined"
end
```

```
$ elixir hello_world.exs
```

Goals in language design

- Easy to learn
- Compatibility
- Extensibility
- Productivity

Easy to learn

- More familiar syntax
- Better documentation
- Better tools (mix)

Erlang compatibility

```
:crypto.md5("Using crypto from Erlang OTP")
<<192,223,75,115,...>>
```

Extensibility

"Now we need to go meta. We should now think of a language design as being a pattern for language designs.
A tool for making more tools of the same kind."

— Guy Steele - "Growing a language" at ACM OOPSLA 1998

Macros

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

unless(is_number(x), do: raise("oops"))
```

Testing DSL

```
defmodule MathTest do
  use ExUnit.Case

  test "basic operations" do
    assert 1 + 1 == 2
  end
end
```

Querying DSL

```
from p in Post,
where: p.published_at < now and
       p.author == "hiphoox",
order: p.created_at
```

Productivity

- Documentation (Markdown) -> *Python*
- Tooling (ExUnit, IEx, Mix) -> *Go, Javascript*
- Package management (Hex) -> *Ruby*
- REPL
- Releases (exrm)

Demos

Elixir Syntax

Multilines

```
iex(1)> 2 * (
      3+ 1
      ) / 4
2.0
```

Writing two or more expresions in the same line

```
iex(1)> 1+2; 1+3
```

Variables

```
iex(1)> monthly_salary = 10000
10000

iex(2)> monthly_salary
10000

iex(3)> monthly_salary = 11000
11000

iex(4)> monthly_salary * 12
120000
```

Modules

```
iex(1)> IO.puts("Hello World!")
Hello World!
:ok

iex(2)> defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end
end

iex(3)> Geometry.rectangle_area(6, 7)
42
```

Nested Modules

```
defmodule Geometry do
  defmodule Rectangle do
    ...
  end
  ...
end

defmodule Geometry.Rectangle do
  ...
end
```

Functions

```
iex(1)> defmodule Geometry do
  def rectangle_area(a, b) do
    a * b
  end
end

iex(2)> Geometry.rectangle_area(3, 2)
6

iex(3)> Geometry.rectangle_area 3, 2
6
```

Functions in a single line

```
defmodule Geometry do
  def rectangle_area(a, b), do: a * b
end
```

Functions without parameters

```
defmodule Program do
  def run do
    ...
  end
end
```

Pipe operator

```
iex(1)> -5 |> abs |> Integer.to_string |> IO.puts
5

iex(2)> IO.puts(Integer.to_string(abs(-5)))
5

# In a file
-5
|> abs
|> Integer.to_string
|> IO.puts
```

Arity

```
defmodule Rectangle do
  def area(a), do: area(a, a)
  def area(a, b), do: a * b
end
```

Default values

```
defmodule MyModule do
  def fun(a, b \\ 1, c, d \\ 2) do
    a + b + c + d
  end
end
```

Private functions

```
defmodule TestPrivate do
  def double(a) do
    sum(a, a)
  end

  defp sum(a, b) do
    a + b
  end
end
```

import directive

Includes functions and macros defined in another module

```
import Module [, only:|except: ]
```

Similar to @include in Ruby

Example

```
defmodule Example do
  import List

  def func1 do
     List.flatten [1,[2,3],4]
  end

  def func2 do
    flatten [5,[6,7],8]
  end
end
```

Be careful with this directive, you can make the code difficult to understand!

Example

```
defmodule MyModule do
  def func1 do
     List.flatten [1,[2,3],4]
  end

  def func2 do
    import List, only: [flatten: 1]
    flatten [5,[6,7],8]
  end
end
```

alias directive

```
# Complete Sintaxis
alias Mix.Tasks.Doctest, as: Doctest

# or
alias Mix.Tasks.Doctest
```

Example

```
defmodule MyModule do
  def func do
    alias Mix.Tasks.Doctest, as: Doctest
    doc = Doctest.setup
    doc.run(Doctest.defaults)
  end
end
```

Module Attributes

```
iex(1)> defmodule Circle do
          @pi 3.14159
          def area(r), do: r*r*@pi
          def circumference(r), do: 2*r*@pi
        end

iex(2)> Circle.area(1)
3.14159

iex(3)> Circle.circumference(1)
6.28318
```

Documentation and metadata

```
defmodule Circle do
  @moduledoc "Implements basic circle functions"
  @pi 3.14159

  @doc "Computes the area of a circle"
  def area(r), do: r*r*@pi

  @doc "Computes the circumference of a circle"
  def circumference(r), do: 2*r*@pi
end
```

Type annotations

```
defmodule Circle do
  @pi 3.14159

  @spec area(number) :: number
  def area(r), do: r*r*@pi

  @spec circumference(number) :: number
  def circumference(r), do: 2*r*@pi
end
```

Used by dialyzer tool

Comments

```
# This is a comment
a = 3.14      # Also this
```

Data Types

Numbers I

```
iex(1)> 3
3
iex(2)> 0xFF
255
iex(3)> 3.14
3.14
iex(4)> 1.0e-2
0.01
```

Numbers II

```
iex(1)> 4/2
2.0
iex(2)> 3/2
1.5

iex(3)> div(5,2)
2
iex(4)> rem(5,2)
1

iex(5)> 1_000_000
1000000
```

Atoms I

```
:an_atom
:another_atom

:"an atom with spaces"
```

Atoms II

```
iex(1) AnAtom
AnAtom

iex(2) :"Elixir.AnAtom"
AnAtom

iex(3)> AnAtom == :"Elixir.AnAtom"
true

iex(4)> AnAtom == Elixir.AnAtom
true
```

Aliases and Modules

```
iex(3)> alias IO, as: MyIO
iex(4)> MyIO.puts("Hello!")
Hello!

iex(5)> MyIO == Elixir.IO
true
```

Atoms and Booleans

```
iex(1)> :true == true
true
iex(2)> :false == false
true

iex(3)> true and false
false
iex(4)> false or true
true
iex(5)> not false
true
```

Nil

```
iex(1)> nil == :nil
true

iex(2)> nil || false || 5 || true
5

iex(3)> true && 5
5
iex(4)> false && 5
false

iex(5)> nil && 5
nil

iex(6)> read_cached || read_from_disk || read_from_database
```

Implementing ternary operator

```
my_string = condition && "value 1" || "value 2"
```

Tuples

```
iex(1)> person = {"Bob", 25}
{"Bob", 25}

iex(2)> age = elem(person, 1)
25

iex(3)> put_elem(person, 1, 26)
{"Bob", 26}
```

Inmutables

```
iex(4)> person
{"Bob", 25}

iex(5)> older_person = put_elem(person, 1, 26)
{"Bob", 26}

iex(6)> older_person
{"Bob", 26}

iex(7)> person = put_elem(person, 1, 26)
{"Bob", 26}
```

Lists

```
iex(1)> prime_numbers = [1, 2, 3, 5, 7]
[1, 2, 3, 5, 7]

iex(2)> length(prime_numbers)
5

iex(3)> Enum.at(prime_numbers, 4)
7
```

Operator in

```
iex(4)> 5 in prime_numbers
true
iex(5)> 4 in prime_numbers
false
```

Changing values

```
iex(6)> List.replace_at(prime_numbers, 0, 11)
[11, 2, 3, 5, 7]

iex(9)> List.insert_at(prime_numbers, 4, 1)
[11, 2, 3, 5, 1, 7]

iex(10)> List.insert_at(prime_numbers, -1, 1)
[11, 2, 3, 5, 7, 1]
```

Concatenation

```
iex(11)> [1,2,3] ++ [4,5]
[1, 2, 3, 4, 5]
```

Head & Tail

```
# a_list = [head | tail]

iex(1)> [1 | []]
[1]

iex(2)> [1 | [2 | []]]
[1, 2]

iex(3)> [1 | [2]]
[1, 2]

iex(4)> [1 | [2, 3, 4]]
[1, 2, 3, 4]
```

Extraction

```
iex(1)> hd([1, 2, 3, 4])
1

iex(2)> tl([1, 2, 3, 4])
[2, 3, 4]
```

Efficient insertion

```
iex(1)> a_list = [5, :value, true]
[5, :value, true]

iex(2)> new_list = [:new_element | a_list]
[:new_element, 5, :value, true]
```

Maps

```
iex(1)> bob = %{:name => "Bob", :age => 25, :works_at => "Initech"}
%{age: 25, name: "Bob", works_at: "Initech"}

iex(2)> bob = %{name: "Bob", age: 25, works_at: "Initech"}
%{age: 25, name: "Bob", works_at: "Initech"}
```

Retriving values

```
iex(3)> bob[:works_at]
"Initech"

iex(4)> bob[:non_existent_field]
nil

iex(5)> bob.age
25

iex(6)> bob.non_existent_field
** (KeyError) key :non_existent_field not found
```

Changing Values

```
iex(7)> next_years_bob = %{bob | age: 26}
%{age: 26, name: "Bob", works_at: "Initech"}

iex(8)> %{bob | age: 26, works_at: "Initrode"}
%{age: 26, name: "Bob", works_at: "Initrode"}

iex(9)> Map.put(bob, :salary, 50000)
%{age: 25, name: "Bob", salary: 50000, works_at: "Initech"}

iex(10)> Dict.put(bob, :salary, 50000)
%{age: 25, name: "Bob", salary: 50000, works_at: "Initech"}
```

Binary Strings

```
iex(1)> "This is a string"
"This is a string"
```

String Interpolation

```
iex(1)> "Embedded expression: #{3 + 0.14}"
"Embedded expression: 3.14"
```

Concatenation

```
iex(3)> "String" <> " " <> "concatenation"
"String concatenation"
```

Keyword Lists

If you have a list of tuples AND the tuples has just two elements AND the first element of the tumple is an atom.

```
iex(1)> days = [{:monday, 1}, {:tuesday, 2}, {:wednesday, 3}]
```

then you can use this syntax

```
iex(2)> days = [monday: 1, tuesday: 2, wednesday: 3]
```

Getting values

```
iex(3)> Keyword.get(days, :monday)
1

iex(4)> Keyword.get(days, :noday)
nil

iex(5)> days[:tuesday]
2
```

Keyword lists and variadic parameters

```
iex(6)> Float.to_string(1/3)
"3.33333333333333314830e-01"

iex(7)> Float.to_string(1/3, [decimals: 2])
"0.33"

iex(8)> Float.to_string(5.2, decimals: 2, compact: true)
"5.2"

def my_fun(arg1, arg2, opts \\ []) do
  ...
end
```

Sigils

```
iex(1)> ~s(This is a string \x26 I love Elixir)
"This is a string & I love Elixir"

iex(2)> ~s("Do... or do not. There is no try." -Master Yoda)
"\"Do... or do not. There is no try.\" -Master Yoda"

iex(3)> ~S(Not interpolated #{3 + 0.14})
"Not interpolated \#{3 + 0.14}"

iex(4)> ~S(Not escaped \n)
"Not escaped \\n"
```

Regular Expressions

```
iex(1)> Regex.run ~r/[aeiou]/, "caterpillar"
["a"]

iex(2)> Regex.scan ~r/[aeiou]/, "caterpillar"
[["a"], ["e"], ["i"], ["a"]]

iex(3)> Regex.split ~r/[aeiou]/, "caterpillar"
["c", "t", "rp", "ll", "r"]

iex(4)> Regex.replace ~r{[aeiou]}, "caterpillar", "*"
"c*t*rp*ll*r"
```

Docs

```
iex(1)> """
        Heredoc must end on its own line """
        """
"Heredoc must end on its own line \"\"\"\n"
```

Lists of words

```
iex(1)> ~w[Sigil de ca#{'d'}ena]
["Sigil", "de", "cadena"]

iex(2)> ~W[Unescaped ca#{'d'}ena]
["Unescaped", "ca#{'d'}ena"]
```

Functions as types

In Elixir functions are first class citizens. They can be assigned to variables.

```
iex(1)> square = fn(x) -> x * x
end

iex(2)> square.(5)
25
```

Functions as parameters

```
iex(1)> print_element = fn(x) -> IO.puts(x) end

iex(2)> Enum.each(
          [1, 2, 3],
          print_element
        )
1
2
3
:ok
```

Without a variable

```
iex(1)> Enum.each(
          [1, 2, 3],
          fn(x) -> IO.puts(x) end
        )
1
2
3
:ok
```

Simplification with capture syntax (&)

```
iex(1)> Enum.each(
          [1, 2, 3],
          &IO.puts/1
        )
1
2
3
:ok
```

Parameter capture

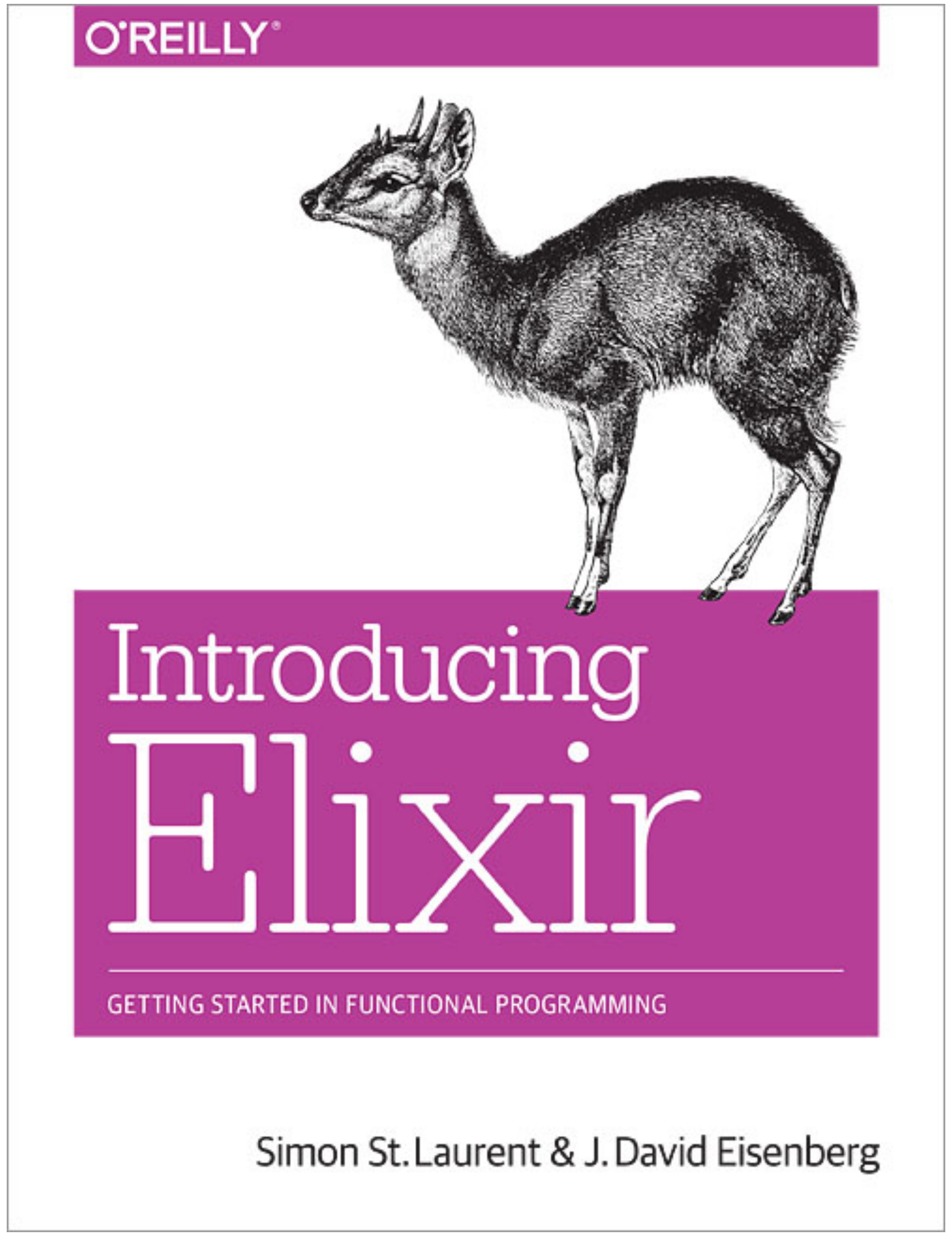
```
iex(7)> lambda = fn(x, y, z) -> x * y + z end

iex(8)> lambda = &(&1 * &2 + &3)

iex(9)> lambda.(2, 3, 4)
10
```

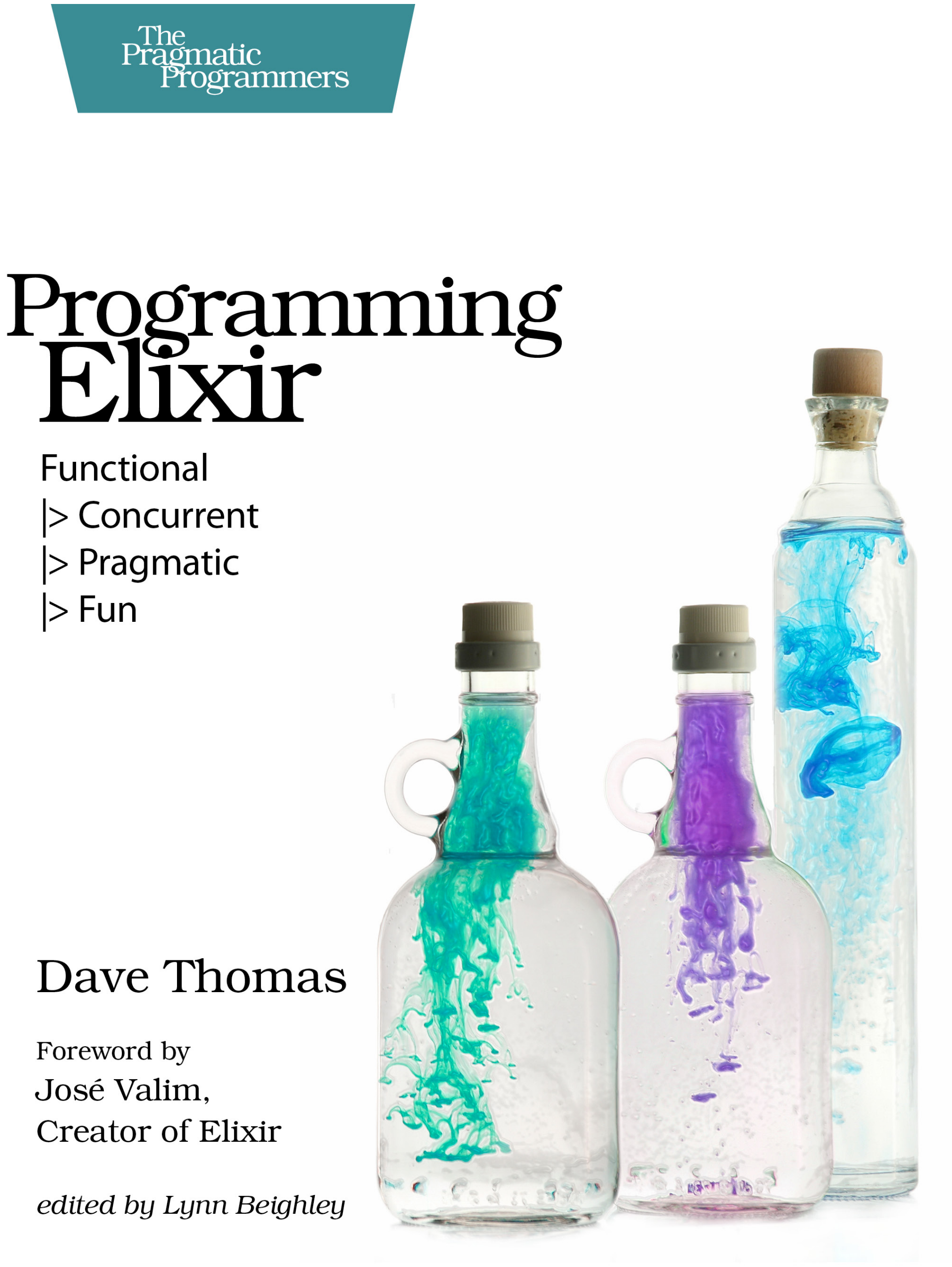
References

Elixir Official Site

Getting Started

Books


O'REILLY
Introducing Elixir
GETTING STARTED IN FUNCTIONAL PROGRAMMING
Simon St.Laurent & J. David Eisenberg


The Pragmatic Programmers
Programming Elixir
Functional
|> Concurrent
|> Pragmatic
|> Fun
Dave Thomas
Foreword by
José Valim,
Creator of Elixir
edited by Lynn Beighley

TWENTY QUESTIONS
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
DICK
HERB
VAN
FLORENCE
ALDO RAY