

Functional and Concurrent Programming With



elixir



[gabrielelana](#) 

[gabrielelana](#) 

gabriele.lana@gmail.com 

In the beginning was...

Erlang

Erlang

What

Programming language born in
~1986 and in use for ~30 years

Designed by Ericsson
for a specific kind of systems
(industrial/business driven design)



Erlang

"The world is concurrent.
Things in the world don't share data.
Things communicate with messages.
Things fail."

JOE ARMSTRONG

Erlang

Goals

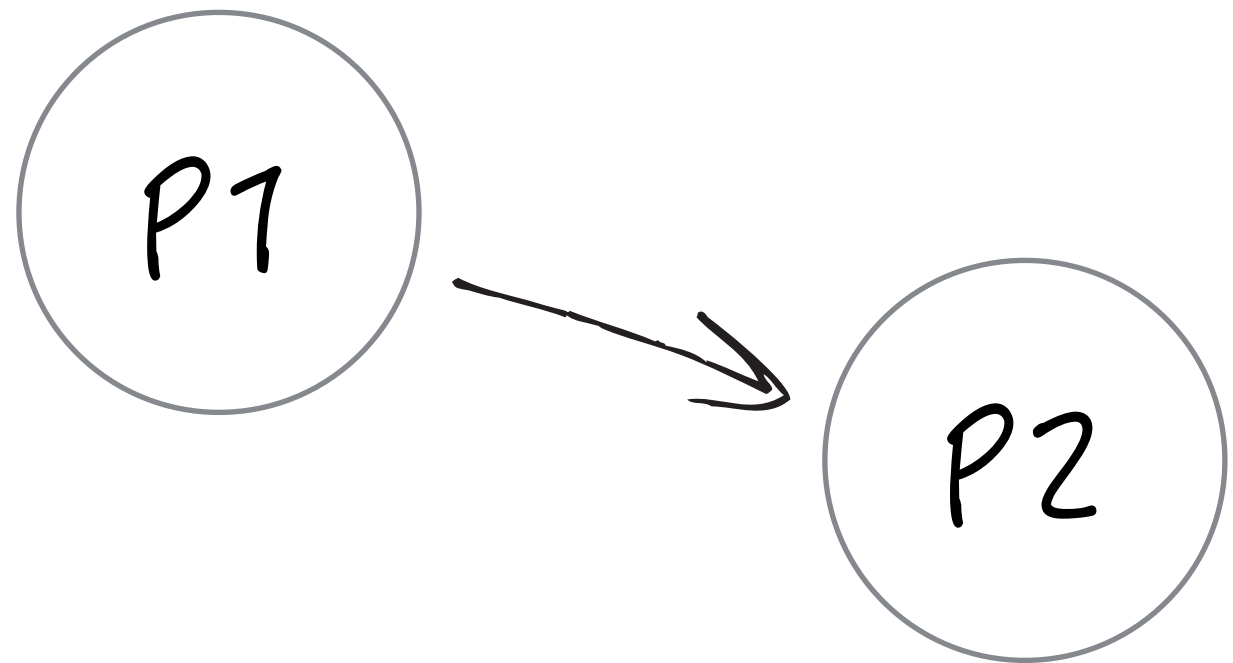
- Fault tolerant
- Massively concurrent
- Massively scalable
- Distributed
- Soft real-time



- Easier to reason about
- Easier to do it right
- More maintainable
- More concise

Erlang

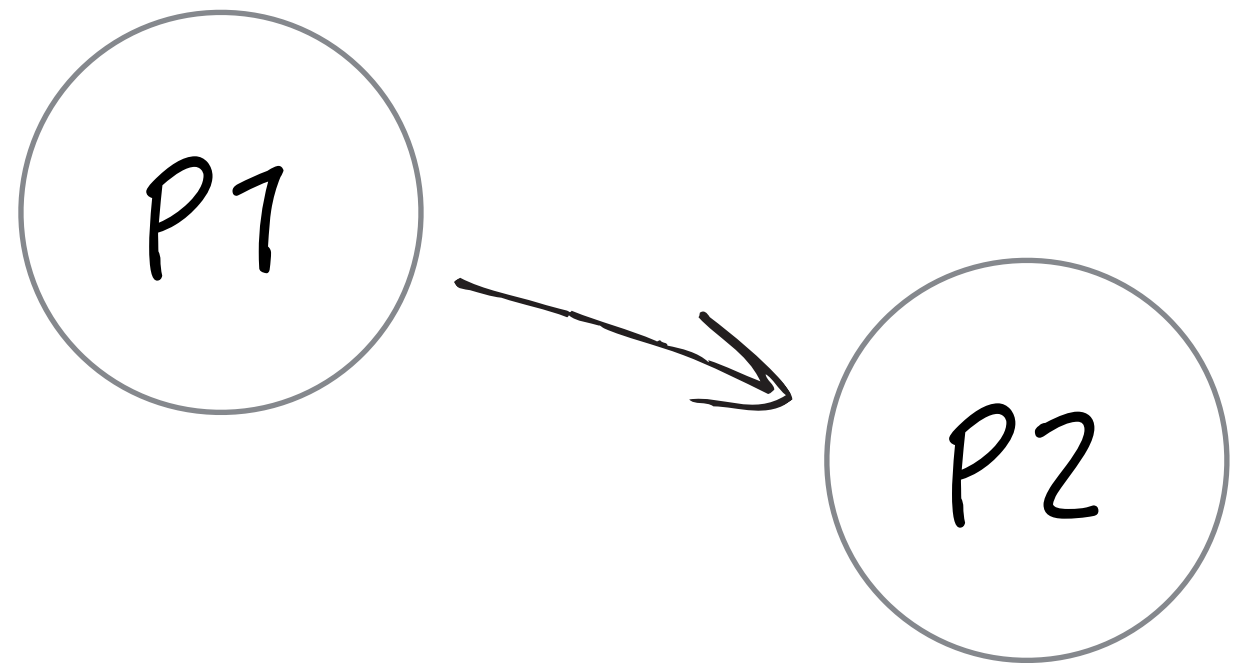
How?



- Immutable data
- Actor model concurrency
 - Really lightweight (think millions)
 - Shares nothing (no locks, no mutexes)
 - Communicate with asynchronous messages

Erlang

How?



- Hot swapping (upgrade without downtime)
- Extremely good at fault handling (let it crash)
- Incredibly robust platform (BEAM)

Erlang

Used by

- Amazon
- Yahoo!
- Facebook
- T-Mobile
- Motorola
- VMWare
- Heroku
- Call Of Duty
- League Of Legends
- AOL
- Ericsson
- WhatsApp
- CouchDB
- Github
- Basho

Erlang

Used by



- 2 Million connections on a single node
- ~450 million users handled by 32 engineers
- Acquired by Facebook for \$19 billion

Erlang

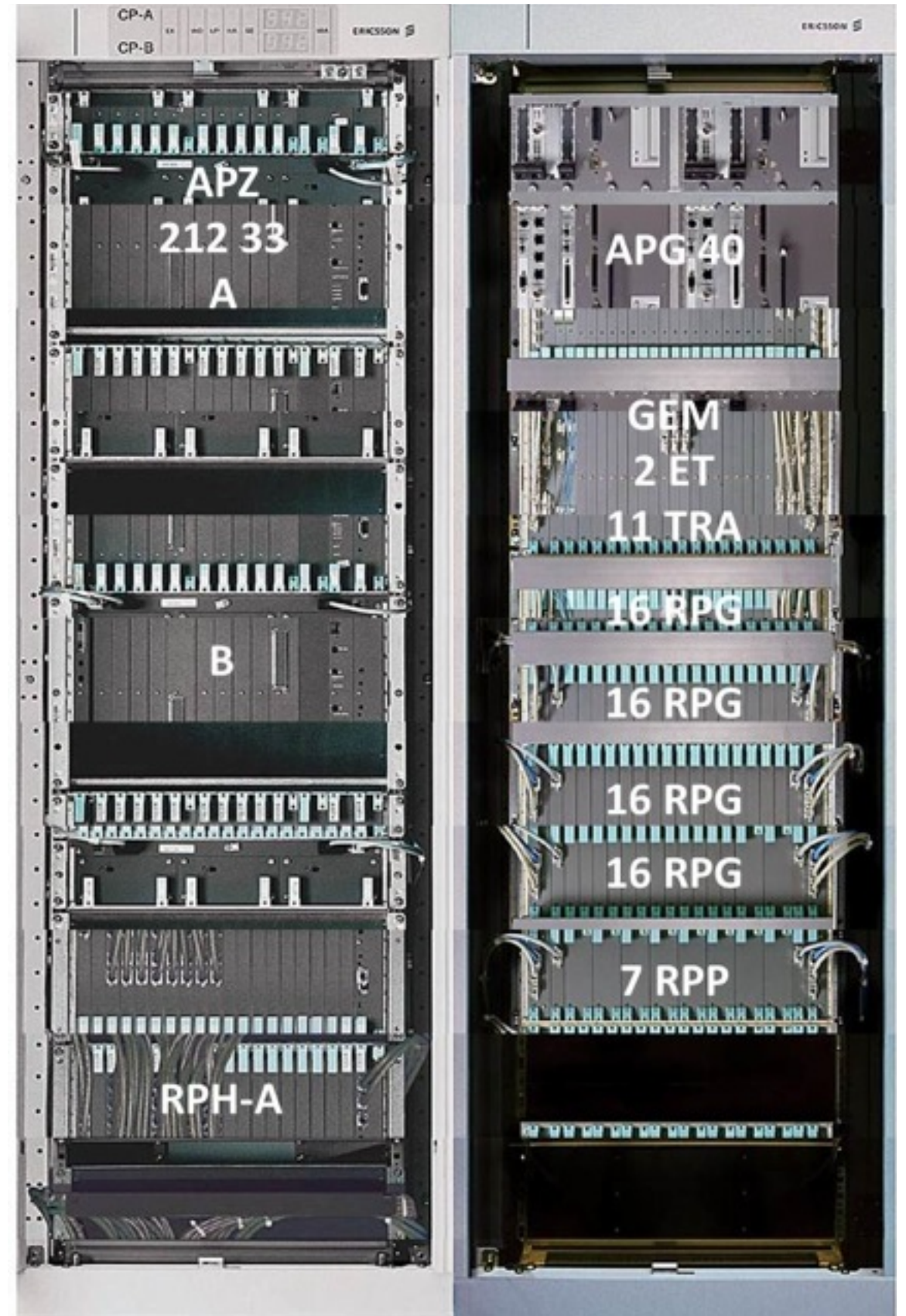
Used by

AXD-301

99.9999999999%

availability

0.6 seconds in 20 years



What's wrong with

Erlang

?... Nothing! As nothing
is wrong with C or
ASSEMBLY

What we will face?

What we will face?

- reliable
- million of users
- fault tolerant

Distributed
Systems

Erlang



What we will face?

- reliable
- million of users
- fault tolerant

Distributed
Systems

Erlang



- easy to change
- easy to prototype
- easy to evolve

Web
Applications



What we will face?

- reliable
- million of users
- fault tolerant

Distributed
Systems

Erlang



- easy to change
- easy to prototype
- easy to evolve

Web
Applications



The
Pragmatic
Programmers

Programming Elixir

Functional

|> Concurrent

|> Pragmatic

|> Fun

Dave Thomas

Foreword by
José Valim,
Creator of Elixir

edited by Lynn Beighley



elixir

"The power of Erlang,
the joy of Ruby"

"I've been looking for
this for 10 years
... finally I found it!"



Macros and
Protocols

Erlang

Semantics and
Runtime



Tooling and
Modularity



Developer Joy

Show me the code!

```
Interactive Elixir (1.0.3)
```

```
iex(1)> "Hello World"
```

```
"Hello World"
```

```
defmodule HelloWorld do
  def say_it do
    IO.puts "Hello World"
  end
end
```

Pattern Matching

Interactive Elixir (1.0.3)

```
iex(1)> a = 1
```

1

```
iex(2)> a
```

1

```
iex(3)> 1 = a
```

1

```
iex(4)> 1 = 1
```

1

```
iex(5)> 1 = 2
```

```
** (MatchError) no match of right hand side value: 2
```

```
iex(6)> a = 2
```

2

```
iex(7)> ^a = 3
```

```
** (MatchError) no match of right hand side value: 3
```

Pattern Matching

Interactive Elixir (1.0.3)

```
iex(1)> a = {1, 2, 3}  
{1, 2, 3}
```

```
iex(2)> {a, b, c} = {1, 2, 3}  
{1, 2, 3}
```

```
iex(3)> [a, b, c]  
[1, 2, 3]
```

```
iex(4)> [a, b, c | _] = [1, 2, 3, 4, 5, 6]  
[1, 2, 3, 4, 5, 6]
```

```
iex(5)> %{:a => a, :b => b, :c => c}  
%{:a => 1, :b => 2, :c => 3}
```

Pattern Matching

```
defmodule Factorial do  
  def of(0), do: 1  
  def of(n) when n > 0 do  
    n * of(n - 1)  
  end  
end
```

Interactive Elixir (1.0.3)

iex(1)> c("factorial.ex")

[Factorial]

iex(2)> Factorial.of(42)

1405006117752879898543142606244511569936384000000000

Pattern Matching

```
defmodule Bowling.Game do
  def score([]), do: 0

  def score([x, y], do: x + y
  def score([x, y, z], do: x + y + z

  def score([10, x, y | rolls]) do
    10 + x + y + score([x, y | rolls])
  end

  def score([x, y, z | rolls]) when x + y == 10 do:
    10 + z + score([z | rolls])
  end

  def score([x, y | rolls]), do: x + y + score(rolls)
end
```

Pattern Matching

```
defmodule Poker.Hand do
  # [{:ace, :diamonds},
  #  {3, :spades},
  #  {3, :hearts},
  #  {3, :diamonds},
  #  {3, :clubs}] => {:four_of_a_kind, 3}

  def identify([{:r1, s}, {:r2, s}, {:r3, s}, {:r4, s}, {:r5, s}])
    when [r2-r1, r3-r2, r4-r3, r5-r4] == [1, 1, 1, 1],
    do: {:straight_flush, r5}

  def identify([{_, s}, {_, s}, {_, s}, {_, s}, {r5, s}]),
    do: {:flush, r5}
end
```

Binaries

Interactive Elixir (1.0.3)

```
iex(1)> <<1, 2>>
```

```
<<1, 2>>
```

```
iex(2)> byte_size <<1, 2>>
```

```
2
```

```
iex(3)> byte_size <<1::size(4), 2::size(4)>>
```

```
1
```

```
iex(4)> is_binary "Hello"
```

```
true
```

```
iex(5)> <<"Hello">> == "Hello"
```

```
true
```

```
iex(6)> <<"H", tail::binary>> = "Hello"
```

```
"Hello"
```

```
iex(7)> tail
```

```
"ello"
```


Pattern Matching

Interactive Elixir (1.0.3)

```
iex(3)> mp3
```

```
<<73, 68, 51, 4, 0, 0, 0, 1, 95, ...>>
```

```
iex(4)> <<_::binary-size(mp3 - 128), id3::binary>> = mp3
```

```
<<73, 68, 51, 4, 0, 0, 0, 1, 95, ...>>
```

```
iex(5)> <<"TAG",  
        title::binary-size(30),  
        artist::binary-size(30),  
        album::binary-size(30),  
        year::binary-size(4),  
        comment::binary-size(30),  
        _rest::binary>> = id3
```

```
...
```

```
iex(6)> title
```

```
"Nothing Else Matters"
```

List Comprehension

Interactive Elixir (1.0.3)

```
iex(1)> for n <- [1, 2, 3, 4], do: n  
[1, 2, 3, 4]
```

```
iex(2)> for n <- [1, 2, 3, 4], do: n * n  
[1, 4, 9, 16]
```

```
iex(3)> for n <- 1..4, do: n * n  
for n <- 1..4, do: n * n
```

```
iex(6)> for n <- 1..100, rem(n, 3) == 0, do: n  
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51,  
54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

List Comprehensions

```
defmodule Sorting do
  def quicksort([]), do: []
  def quicksort([pivot | t]) do
    quicksort(for x <- t, x <= pivot, do: x)
    ++ [pivot] ++
    quicksort(for x <- t, x > pivot, do: x)
  end
end
```

List Comprehensions

Interactive Elixir (1.0.3)

```
iex(1)> suits = [:clubs, :diamonds, :hearts, :spades]  
[:clubs, :diamonds, :hearts, :spades]
```

```
iex(2)> ranks = [:ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, :jack, ...]  
[:ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, :jack, :queen, :king]
```

```
iex(3)> deck = for rank <- ranks, suit <- suits, do: {rank, suit}  
[{2, :clubs}, {2, :diamonds}, {2, :hearts}, {2, :spades},  
{3, :clubs}, {3, :diamonds}, {3, :hearts}, {3, :spades},  
{4, :clubs}, {4, :diamonds}, {4, :hearts}, {4, :spades}, ...]
```

```
iex(4)> Enum.shuffle(deck)  
[{8, :diamonds}, {:jack, :clubs}, {2, :hearts}, {3, :hearts},  
{:king, :hearts}, {9, :hearts}, {:king, :diamonds}, {9, :spades},  
{10, :clubs}, {10, :spades}, {:king, :spades}, {2, :diamonds}, ...]
```

Pipe Operator

```
defmodule Crunch do
  def of(something) do
    filtered = filter(something)
    sorted = sort(filtered)
    grouped = group(sorted)
    count(grouped)
  end
end
```

Pipe Operator

```
defmodule Crunch do
  def of(something) do
    count(group(sort(filter(something))))
  end
end
```

Pipe Operator

```
defmodule Crunch do
  def of(something) do
    something
    |> filter
    |> sort
    |> group
    |> count
  end
end
```

Sigils & Heredoc

Interactive Elixir (1.0.3)

```
iex(1)> ~s(this is a string with "double" and 'single' quotes)  
"this is a string with \"double\" and 'single' quotes"
```

```
iex(2)> ~w(foo bar bat)  
["foo", "bar", "bat"]
```

```
iex(3)> "foo" =~ ~r/foo|bar/  
true
```

```
iex(4)> string = """  
...(4)>     This is a  
...(4)>     multiline string  
...(4)>     properly indented  
...(4)>     """  
"This is a\nmultiline string\nproperly indented\n"
```


Sigils & Pipes

```
defmodule SExpression do
  use Paco # parser combinator library
  parser expression do
    one_of([Parser.number, expression])
    |> separated_by(~1", "w)
    |> surrounded_by(~1"("w, ~1")"w)
  end
end
```

Interactive Elixir (1.0.3)

```
iex(1)> SExpression.parse! "(1, 2, (3, 4))"
[1, 2, [3, 4]]
```

Sigils & Pipes

```
defmodule Paco do
  # ...
  def sigil_1(lexeme, [?w]) do
    lexeme |> surrounded_by(maybe(whitespaces))
  end
  # ...
end
```

Macros

```
defmodule Control do
  def unless(clause, [do: expression]) do
    if (!clause) do
      expression
    end
  end
end
```

Macros

```
defmodule Control do
  def unless(clause, [do: expression]) do
    if (!clause) do
      expression
    end
  end
end
```

```
defmodule Sandbox do
  import Control
  def try_unless(clause) do
    unless clause do
      IO.puts "Clause is false"
    end
  end
end
```

Macros

```
defmodule Control do
  def unless(clause, [do: expression]) do
    if (!clause) do
      expression
    end
  end
end
```

```
defmodule Sandbox do
  import Control
  def try_unless(clause) do
    unless clause do
      IO.puts "Clause is false"
    end
  end
end
```

```
Interactive Elixir (1.0.3)
iex(1)> Sandbox.try_unless(false)
Clause is false
:ok

iex(2)> Sandbox.try_unless(true)
Clause is false
:ok
```

Macros

```
defmodule Control do
  defmacro unless(clause, [do: expression]) do
    quote do
      if (not unquote(clause)) do
        unquote(expression)
      end
    end
  end
end
```

Macros

```
defmodule Control do
  defmacro unless(clause, [do: expression]) do
    quote do
      if (not unquote(clause)) do
        unquote(expression)
      end
    end
  end
end
```

```
defmodule Sandbox do
  import Control
  def try_unless(clause) do
    unless clause do
      IO.puts "Clause is false"
    end
  end
end
```

Macros

```
defmodule Control do
  defmacro unless(clause, [do: expression]) do
    quote do
      if (not unquote(clause)) do
        unquote(expression)
      end
    end
  end
end
```

```
defmodule Sandbox do
  import Control
  def try_unless(clause) do
    unless clause do
      IO.puts "Clause is false"
    end
  end
end
```

```
Interactive Elixir (1.0.3)
iex(1)> Sandbox.try_unless(false)
Clause is false
:ok

iex(2)> Sandbox.try_unless(true)
nil
```


Macros (AST)

Interactive Elixir (1.0.3)

```
iex(1)> quote do: 40 + 2  
{:+, [], [40, 2]}
```

```
iex(2)> quote do: IO.puts "Hello World"  
{  
  {  
    :.,  
    [],  
    [:IO, :puts]  
  },  
  [],  
  ["Hello World"]  
}
```

Macros (UTF-8)

Interactive Ruby (2.1.4)

```
irb(2.1.4) :001 > "Jalapeño".upcase  
=> "JALAPEÑO"
```

```
irb(2.1.4) :002 > "Noël".reverse  
=> "lëoN"
```

Interactive Elixir (1.0.3)

```
iex(1)> String.upcase("Jalapeño")  
"JALAPEÑO"
```

```
iex(2)> String.reverse("Noël")  
"lëoN"
```

Macros (UTF-8)

```
defmodule String.Unicode do
  # ...
  for {codepoint, upper, _, _} <- codes,
      upper && upper != codepoint do
    defp do_upcase(unquote(codepoint) <> rest) do
      unquote(upper) ++ do_upcase(rest)
    end
  end
  # ...
end
```

Macros (UTF-8)

```
defmodule String.Unicode do
```

```
# ...
```

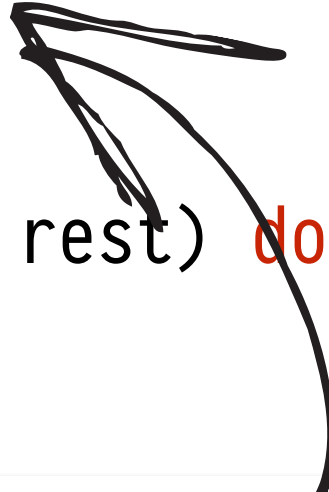
```
for {codepoint, upper, _, _} <- codes,  
    upper && upper != codepoint do
```

```
  defp do_upcase(unquote(codepoint) <> rest) do  
    unquote(upper) ++ do_upcase(rest)  
  end
```

```
end
```

```
# ...
```

```
end
```



```
01D5;LATIN CAPITAL LETTER U WITH DIAERESIS AND MACRON;Lu;0;L;00DC 0304;;;N;LATIN CAPITAL LETTER U DIAERESIS MACRON;;;01D6;  
01D6;LATIN SMALL LETTER U WITH DIAERESIS AND MACRON;Ll;0;L;00FC 0304;;;N;LATIN SMALL LETTER U DIAERESIS MACRON;;;01D5;;01D5  
01D7;LATIN CAPITAL LETTER U WITH DIAERESIS AND ACUTE;Lu;0;L;00DC 0301;;;N;LATIN CAPITAL LETTER U DIAERESIS ACUTE;;;01D8;  
01D8;LATIN SMALL LETTER U WITH DIAERESIS AND ACUTE;Ll;0;L;00FC 0301;;;N;LATIN SMALL LETTER U DIAERESIS ACUTE;;;01D7;;01D7  
01D9;LATIN CAPITAL LETTER U WITH DIAERESIS AND CARON;Lu;0;L;00DC 030C;;;N;LATIN CAPITAL LETTER U DIAERESIS HACEK;;;01DA;  
01DA;LATIN SMALL LETTER U WITH DIAERESIS AND CARON;Ll;0;L;00FC 030C;;;N;LATIN SMALL LETTER U DIAERESIS HACEK;;;01D9;;01D9  
01DB;LATIN CAPITAL LETTER U WITH DIAERESIS AND GRAVE;Lu;0;L;00DC 0300;;;N;LATIN CAPITAL LETTER U DIAERESIS GRAVE;;;01DC;  
01DC;LATIN SMALL LETTER U WITH DIAERESIS AND GRAVE;Ll;0;L;00FC 0300;;;N;LATIN SMALL LETTER U DIAERESIS GRAVE;;;01DB;;01DB  
01DD;LATIN SMALL LETTER TURNED E;Ll;0;L;;;;N;;;018E;;018E  
01DE;LATIN CAPITAL LETTER A WITH DIAERESIS AND MACRON;Lu;0;L;00C4 0304;;;N;LATIN CAPITAL LETTER A DIAERESIS MACRON;;;01DF;  
01DF;LATIN SMALL LETTER A WITH DIAERESIS AND MACRON;Ll;0;L;00E4 0304;;;N;LATIN SMALL LETTER A DIAERESIS MACRON;;;01DE;;01DE  
01E0;LATIN CAPITAL LETTER A WITH DOT ABOVE AND MACRON;Lu;0;L;0226 0304;;;N;LATIN CAPITAL LETTER A DOT MACRON;;;01E1;  
01E1;LATIN SMALL LETTER A WITH DOT ABOVE AND MACRON;Ll;0;L;0227 0304;;;N;LATIN SMALL LETTER A DOT MACRON;;;01E0;;01E0  
01E2;LATIN CAPITAL LETTER AE WITH MACRON;Lu;0;L;00C6 0304;;;N;LATIN CAPITAL LETTER A E MACRON;;;01E3;  
01E3;LATIN SMALL LETTER AE WITH MACRON;Ll;0;L;00E6 0304;;;N;LATIN SMALL LETTER A E MACRON;;;01E2;;01E2  
01E4;LATIN CAPITAL LETTER G WITH STROKE;Lu;0;L;;;;N;LATIN CAPITAL LETTER G BAR;;;01E5;  
01E5;LATIN SMALL LETTER G WITH STROKE;Ll;0;L;;;;N;LATIN SMALL LETTER G BAR;;;01E4;;01E4  
01E6;LATIN CAPITAL LETTER G WITH CARON;Lu;0;L;0047 030C;;;N;LATIN CAPITAL LETTER G HACEK;;;01E7;  
01E7;LATIN SMALL LETTER G WITH CARON;Ll;0;L;0067 030C;;;N;LATIN SMALL LETTER G HACEK;;;01E6;;01E6  
01E8;LATIN CAPITAL LETTER K WITH CARON;Lu;0;L;004B 030C;;;N;LATIN CAPITAL LETTER K HACEK;;;01E9;  
01E9;LATIN SMALL LETTER K WITH CARON;Ll;0;L;006B 030C;;;N;LATIN SMALL LETTER K HACEK;;;01E8;;01E8  
01EA;LATIN CAPITAL LETTER O WITH OGONEK;Lu;0;L;004F 0328;;;N;LATIN CAPITAL LETTER O OGONEK;;;01EB;
```

Macros (Phoenix)

```
defmodule Example.Router do
  use Phoenix.Router

  pipeline :browser do
    plug :accepts, ~w(html)
    plug :protect_from_forgery
  end

  pipeline :api do
    plug :accepts, ~w(json)
  end

  scope "/", Example do
    pipe_through :browser
    get "/", HomeController, :index
  end
end
```

Macros (Ecto)

```
defmodule Weather do
  use Ecto.Model

  schema "weather" do
    field :city, :string
    field :temp_lo, :integer
    field :temp_hi, :integer
    field :prcp, :float, default: 0.0
  end

  def cities_where_it_rains
    Repo.all(
      from w in Weather,
      where: w.prcp > 0 or is_nil(w.prcp),
      select: w
    )
  end
end
```

Macros Everywhere

```
defmodule Kernel do
```

```
  # ...
```

```
  # ...
```

```
end
```

Macros Everywhere

```
defmodule Kernel do
  # ...

  defmacro if(condition, clauses) do # ...

  # ...
end
```


Macros Everywhere

```
defmodule Kernel do
```

```
  # ...
```

```
  defmacro if(condition, clauses) do # ...
```

```
  defmacro defmodule(alias, do: block) do # ...
```

```
  # ...
```

```
end
```

Macros Everywhere

```
defmodule Kernel do
  # ...

  defmacro if(condition, clauses) do # ...

  defmacro defmodule(alias, do: block) do # ...

  defmacro def(call, expr \\ nil) do # ...

  # ...
end
```

Macros Everywhere

```
defmodule Kernel do
  # ...

  defmacro if(condition, clauses) do # ...

  defmacro defmodule(alias, do: block) do # ...

  defmacro def(call, expr \\ nil) do # ...

  defmacro defmacro(call, expr \\ nil) do # ...

  # ...
end
```

Macros

```
defmodule Control do
  defmacro unless(clause, [do: expression]) do
    quote do
      if (not unquote(clause)) do
        unquote(expression)
      end
    end
  end
end
```

Macros


```
defmodule Control do
  defmacro unless(clause, [do: expression]) do
    quote do
      if (not unquote(clause)) do
        unquote(expression)
      end
    end
  end
end
```

```
defmodule Sandbox do
  import Kernel, except: [unless: 2]
  import Control
  def try_unless(clause) do
    unless clause do
      IO.puts "Clause is false"
    end
  end
end
```

Macros

```
defmodule Control do
  defmacro unless(clause, [do: expression]) do
    quote do
      if (not unquote(clause)) do
        unquote(expression)
      end
    end
  end
end
```

Everything
is scoped



```
defmodule Sandbox do
  import Kernel, except: [unless: 2]
  import Control
  def try_unless(clause) do
    unless clause do
      IO.puts "Clause is false"
    end
  end
end
```

Macros Rules

1. Don't write macros
2. Use macros gratuitously
3. Have fun



 Included

Mix

```
$ mix new bowling_game
* creating README.md
* creating .gitignore
* creating mix.exs
* creating config
* creating config/config.exs
* creating lib
* creating lib/bowling_game.ex
* creating test
* creating test/test_helper.exs
* creating test/bowling_game_test.exs
$ cd bowling_game
$ mix compile
Compiled lib/bowling_game.ex
Generated bowling_game.app
```

Mix

```
$ mix mix help
mix          # Run the default task (current: mix run)
mix archive  # List all archives
mix archive.build  # Archive this project into a .ez file
mix archive.install  # Install an archive locally
mix archive.uninstall  # Uninstall archives
mix clean    # Delete generated application files
mix cmd      # Executes the given command
mix compile  # Compile source files
mix compile.protocols  # Consolidates all protocols in all paths
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
...
```

ExUnit

```
defmodule Bowling.Game.Test do
  import Bowling.Game
  use ExUnit.Case

  test "score is 0 for all zeros" do
    assert score(roll_many(20, 0)) == 0
  end

  test "score is 20 for all ones" do
    assert score(roll_many(20, 1)) == 20
  end

  test "a throw counts twice after a spare" do
    assert score([5, 5, 3 | roll_many(17, 0)]) == 16
    assert score([0, 10, 3 | roll_many(17, 0)]) == 16
  end

  # ...
end
```

ExUnit

```
defmodule Bowling.Game.Test do
  import Bowling.Game
  use ExUnit.Case
```

```
test "score" do
  assert score == 10
end
```

```
test "score" do
  assert score == 10
end
```

```
test "a throw" do
  assert score == 10
  assert score == 10
end
```

```
# ...
end
```

```
$ mix test
.....
```

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

```
Randomized with seed 825783
```

ExUnit

```
defmodule Bowling.Game.Test do
  import Bowling.Game
  use ExUnit.Case
```

```
test "score is 0 for all zeros" do
  assert score(roll_many(20, 0)) == 0
end
```

```
test "score is 10 for a strike" do
  assert score(roll_many(20, 10)) == 10
end
```

```
test "a throw can be a spare" do
  assert score(roll_many(20, 9)) == 18
  assert score(roll_many(20, 1)) == 1
end
```

```
# ...
end
```

```
$ mix test
```

```
1) test score is 0 for all zeros (Bowling.Game.Test)
```

```
test/bowling_game_test.exs:5
```

```
Assertion with == failed
```

```
code: score(roll_many(20, 0)) == 1
```

```
lhs: 0
```

```
rhs: 1
```

```
stacktrace:
```

```
test/bowling_game_test.exs:6
```

```
.....
```

```
Finished in 0.1 seconds (0.08s on load, 0.02s on tests)
```

```
7 tests, 1 failures
```

```
Randomized with seed 416239
```

Type Annotations

```
defmodule Bowling.Game do
  @spec score([0..10]) :: 0..300

  def score([]), do: 0

  def score([x, y]), do: x + y
  def score([x, y, z]), do: x + y + z

  # ...
end
```

Type Annotations

```
defmodule Bowling.Game do
```

```
  @spec score([0..10]) :: 0..300
```

```
  def score([], do: 0)
```

```
  def score([_], do: 0)
```

```
  def score([_], do: 0)
```

```
  # ...
```

```
end
```

```
$ mix dialyzer
Starting Dialyzer
Proceeding with analysis... done in 0m0.94s
done (passed successfully)
```

Type Annotations

```
defmodule Bowling.Game do
  def broke, do: score(:boom)
  @spec score([0..10]) :: 0..300
  def score([]), do: 0
  def score([x, y]), do: x + y
  def score([x, y, z]), do: x + y + z
  # ...
end
```


Type Annotations

```
defmodule Bowling.Game do
```

```
  def broke, do: score(:boom)
```

```
  @spec score([0..10]) :: 0..300
```

```
  def score()
```

```
  def score()
```

```
  def score()
```

```
  # ...
```

```
end
```

```
$ mix dialyzer
Starting Dialyzer
Proceeding with analysis...
bowling_game.ex:29: Function broke/0 has no local return
bowling_game.ex:30: The call 'Elixir.Bowling.Game':score('boom') will
never return since the success typing is ([number()]) -> number() and
the contract is ([0..10]) -> 0..300
done in 0m0.90s
done (warnings were emitted)
```

Doc Annotations

```
defmodule Bowling.Game do
  @moduledoc """
  Given a valid sequence of rolls for one line of American Ten-Pin Bowling ...
  * We will not check for valid rolls.
  * We will not check for correct number of rolls and frames.
  * We will not provide scores for intermediate frames.
  ...
  """

  @doc """
  Given a sequence of rolls for one line of American Ten-Pin Bowling returns the score

  ## Examples

      iex> Bowling.Game.score([10, 2, 3, 5, 5, 4, 3, 2, 1, 5, 5, 7, 2, 3, 3, 2, 1, 4, 4])
      86
  """

  @spec score([0..10]) :: 0..300
  def score([], do: 0)
  # ...
end
```

Doc Annotations

```
defmodule Bowling.Game do
```

```
  @moduledoc """
```

```
    Given a valid sequence of rolls for one line of American Ten-Pin Bowling ...
```

```
    * We will not check for valid rolls.
```

```
    * We will not check for correct number of rolls and frames.
```

```
    * We will not provide scores for intermediate frames.
```

```
    ...
  """
```

```
  @doc """
```

```
    Given a sequence of rolls for one line of American Ten-Pin Bowling, produces the total score for the game. Here are some things that the program will not do:
```

```
    ## Examples
```

```
    iex> Bowling.Game.score([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
    86
```

```
  """
```

```
  @spec score([0..9]) :: integer
```

```
  def score([], _), do: 0
```

```
  # ...
```

```
end
```

```
iex(1)> h Bowling.Game
```

Bowling.Game

Given a valid sequence of rolls for one line of American Ten-Pin Bowling, produces the total score for the game. Here are some things that the program will not do:

- We will not check for valid rolls.
- We will not check for correct number of rolls and frames.
- We will not provide scores for intermediate frames.

We can briefly summarize the scoring for this form of bowling:

...

Doc Annotations

```
defmodule Bowling.Game do
```

```
  @moduledoc """
```

```
    Given a valid sequence of rolls for one line of American Ten-Pin Bowling ...
```

```
    * We will not check for valid rolls.
```

```
    * We will not check for correct number of rolls and frames.
```

```
    * We will not check for correct number of pins.
```

```
    ...
  """
```

```
  @doc """
```

```
    Given a sequence of rolls for one line of American Ten-Pin Bowling ...
```

```
    ## Examples
```

```
    iex> Bowling.Game.score([10, 2, 3, 5, 5, 4, 3, 2, 1, 5, 5, 7, 2, 3, 8])
    86
```

```
  """
```

```
  @spec score([0..10]) :: integer
```

```
  def score([]),
```

```
    # ...
```

```
end
```

```
iex(2)> h Bowling.Game.score
```

```
def score(list)
```

```
Given a sequence of rolls for one line of American Ten-Pin Bowling
returns the total score for the game
```

```
Examples
```

```
iex> Bowling.Game.score([10, 2, 3, 5, 5, 4, 3, 2, 1, 5, 5, 7, 2, 3, 8])
86
```

Doc Annotations

```
defmodule Bowling.Game do
  @moduledoc """
  Given a valid sequence of rolls for one line of American Ten-Pin Bowling ...
  * We will not check for valid rolls.
  * We will not check for correct number of rolls and frames.
  * We will not provide scores for intermediate frames.
  ...
  """

  @doc """
  Given a sequence of rolls for one line of American Ten-Pin Bowling returns the score

  ## Examples

      iex> Bowling.Game.score([10, 2, 3, 5, 5, 4, 3, 2, 1, 5, 5, 7, 2, 3, 3, 2, 1, 4, 4])
      86
  """

  @spec score([0..10]) :: 0..300
  def score([], do: 0)
  # ...
end
```

Doc Annotations

```
defmodule Bowling.Game do
  @moduledoc """
  Given a valid sequence of rolls for one line of American Ten-Pin Bowling ...
  * We will not check for valid rolls.
  * We will not check for correct number of rolls and frames.
  * We will not
  ...
  """

  @doc """
  Given a sequen

  ## Examples

      iex> Bowli
      86
  """

  @spec score([0
  def score([]),
    # ...
end
```

```
$ mix docs
Docs successfully generated.
View them at "doc/index.html".
```

Doc Annotations

```
defmodule Bowling.Game do
  @moduledoc """
```

bowling_game v0.0.1

[Overview](#)

[Modules](#) | [Exceptions](#) | [Protocols](#)

► **BowlingGame**

[bowling_game v0.0.1](#) → [Overview](#) → [BowlingGame](#)

BowlingGame

[Summary](#)

[Functions](#)

Given a valid sequence of rolls for one line of American Ten-Pin Bowling, produces the total score for the game. Here are some things that the program will not do:

- We will not check for valid rolls.
- We will not check for correct number of rolls and frames.
- We will not provide scores for intermediate frames.

We can briefly summarize the scoring for this form of bowling:

- Each game, or "line" of bowling, includes ten turns, or "frames" for the bowler.
- In each frame, the bowler gets up to two tries to knock down all the pins.
- If in two tries, he fails to knock them all down, his score for that frame is the total number of pins knocked down in his two tries.
- If in two tries he knocks them all down, this is called a "spare" and his score for the frame is ten plus the number of pins knocked down on his next throw (in his next turn).
- If on his first try in the frame he knocks down all the pins, this is called a "strike". His turn is over, and his score for the frame is ten plus the simple total of the pins knocked down in his next two rolls.
- If he gets a spare or strike in the last (tenth) frame, the bowler gets to throw one or two more bonus balls, respectively. These bonus throws are taken as part of the same turn. If the bonus throws knock down all the pins, the process does not repeat: the bonus throws are only used to calculate the score of the final frame.
- The game score is the total of all frame scores.

Summary ↑

`score(list1)` Given a sequence of rolls for one line of American Ten-Pin Bowling returns the total score for the game

Functions

`score(list1)`

(function) # ↑

Specs:

```
score([0 .. 10]) :: 0 .. 300
```

Given a sequence of rolls for one line of American Ten-Pin Bowling returns the total score for the game

Examples

Doc Annotations

```
defmodule Bowling.Game.Test do
  import Bowling.Game
  use ExUnit.Case
```

```
doctest Bowling.Game
```

```
test "score is 0 for all zeros" do
  assert score(roll_many(20, 0)) == 0
end
```

```
test "score is 20 for all ones" do
  assert score(roll_many(20, 1)) == 20
end
```

```
# ...
end
```


Doc Annotations

```
defmodule Bowling.Game.Test do
  import Bowling.Game
  use ExUnit.Case
```

```
doctest Bowling.Game
```

```
test "score" do
  assert score == 86
end
```

```
test "score" do
  assert score == 87
end
```

```
# ...
end
```

```
$ mix test
1) test doc at Bowling.Game.score/1 (1) (Bowling.Game.Test)
   test/bowling_game_test.exs:5
   Doctest failed
   code: Bowling.Game.score([10, 2, 3, 5, 5, ..., 4, 4]) == 86
   lhs: 87
   stacktrace:
     lib/bowling_game.ex:31: Bowling.Game (module)
   .....

Finished in 0.1 seconds (0.1s on load, 0.01s on tests)
8 tests, 1 failures
```

Processes

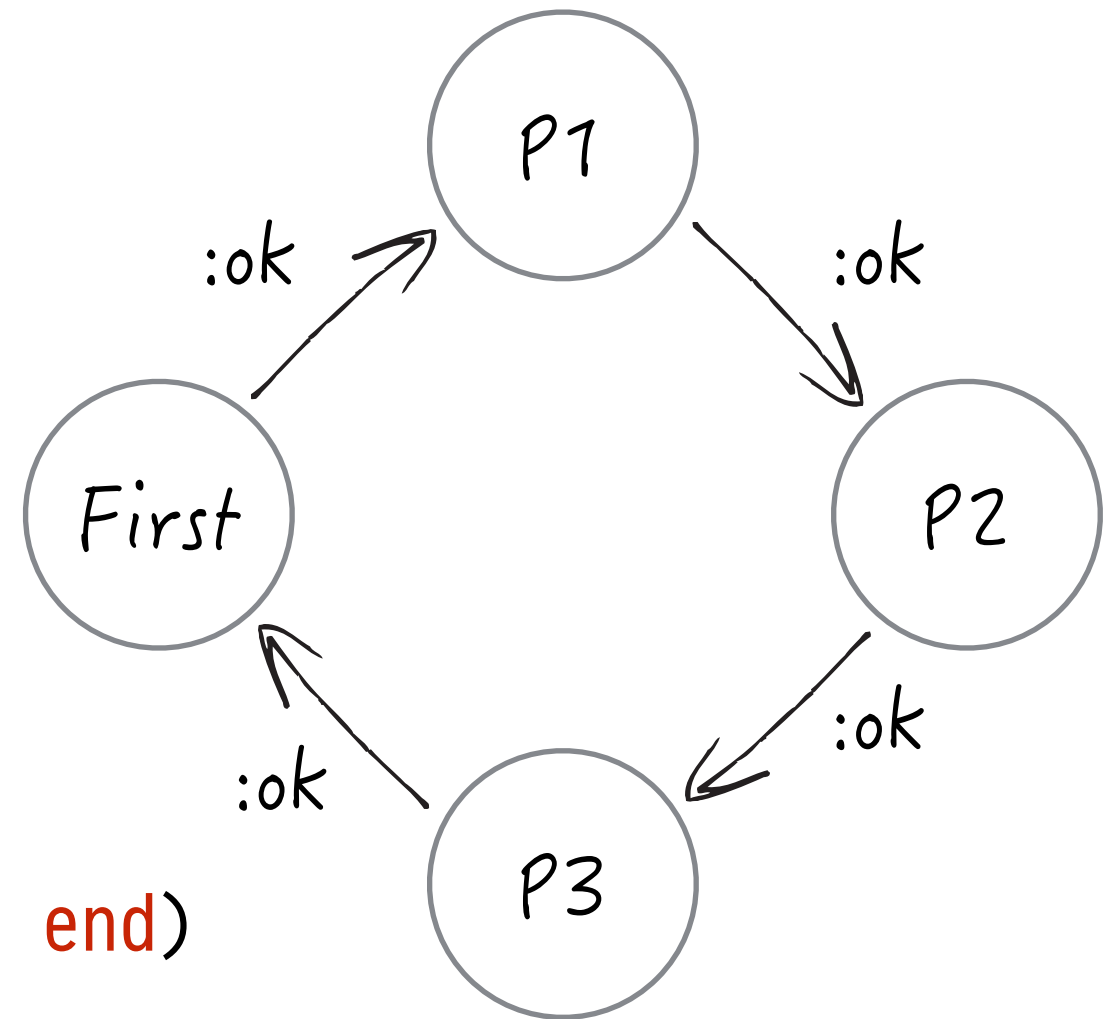


Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end

  def start(0, first) do
    send(first, :ok)
  end

  def start(n, first) do
    spawn(fn -> start(n - 1, first) end)
    |> send(:ok)
    receive do
      :ok -> :ok
    end
  end
end
```



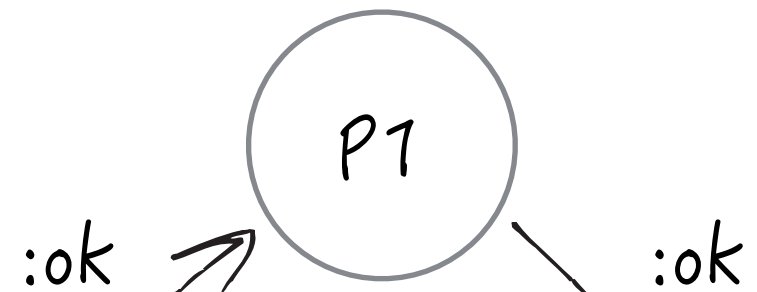
Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```

```
def start(n) do
  send(n, :ok)
end
```

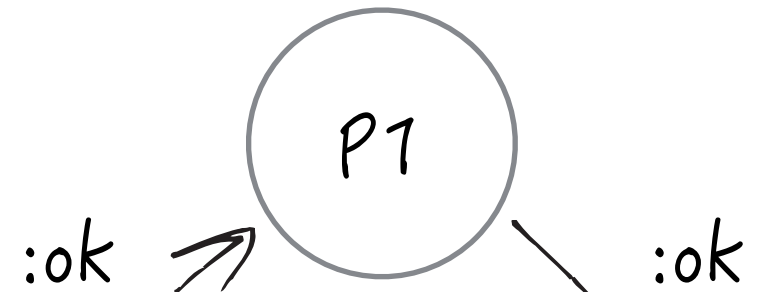
```
def start(n) do
  spawn(fn -> send(n, :ok) end)
end
```

```
end
```



Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



Interactive Elixir (1.0.3)

```
def start(
  send(fin
end
```

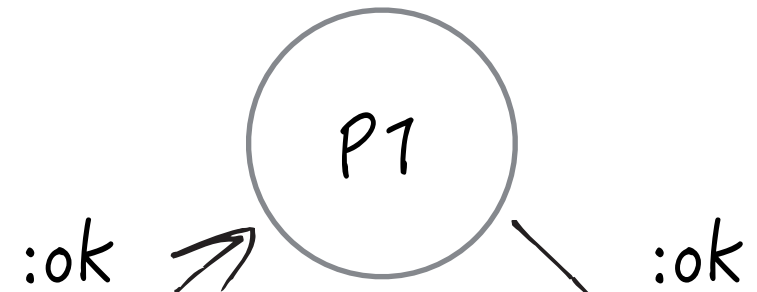
```
def start(
  spawn(fr
  |> send(
  receive
    :ok ->
  end
```

```
end
```

```
end
```

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



```
def start(n) do
  send(self(), :start)
end

def start(n) do
  spawn(fn -> send(self(), :start))
  receive do
    :ok -> ...
  end
end

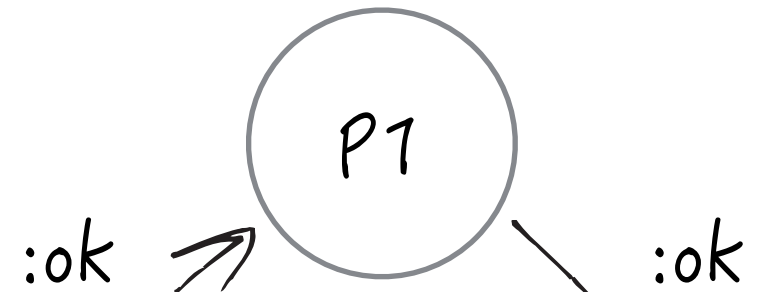
end
```

Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
```

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



```
def start(n) do
  send(self(), {n, :ok})
end

def start(n) do
  spawn(fn -> send(self(), {n, :ok}))
end
end
```

Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
{:ok, {1677, :ok}}
```

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```

A diagram showing a central circle labeled "P1". Two arrows originate from the bottom of the circle, pointing downwards and outwards. Each arrow is labeled with the text ":ok".

```
def start(iex(1)> :timer_send(fir {1677, :ok} iex(2)> :timer
```

Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
```

{1677, :ok}

```
iex(2)> :timer.tc(Ring, :start, [10_000])
```

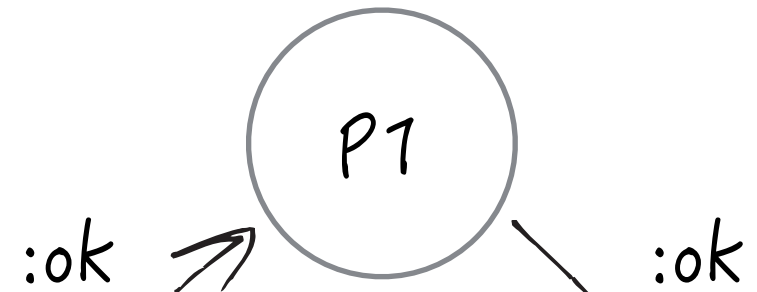
```
def start()  
  spawn(frm  
    |> send()  
  receive  
    :ok ->  
end
```

end

end

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



```
def start(n) do
  send(self(), {n, :ok})
end

def start(n) do
  spawn(fn -> send(self(), {n, :ok})
end
end
```

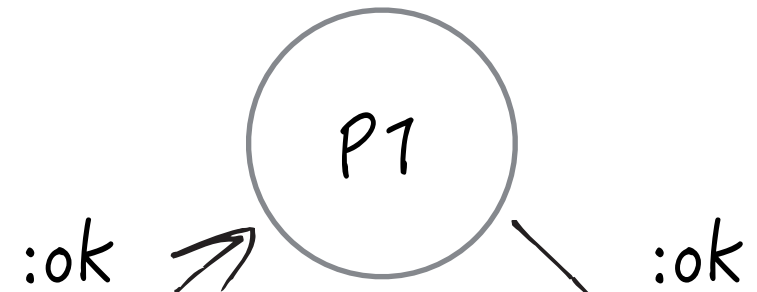
Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
{1677, :ok}

iex(2)> :timer.tc(Ring, :start, [10_000])
{12431, :ok}
```

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



```
def start(n) do
  send(self(), :ok)
end
```

```
def start(n) do
  spawn(fn -> send(self(), :ok) end)
end
```

```
end
end
```

Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
```

```
{1677, :ok}
```

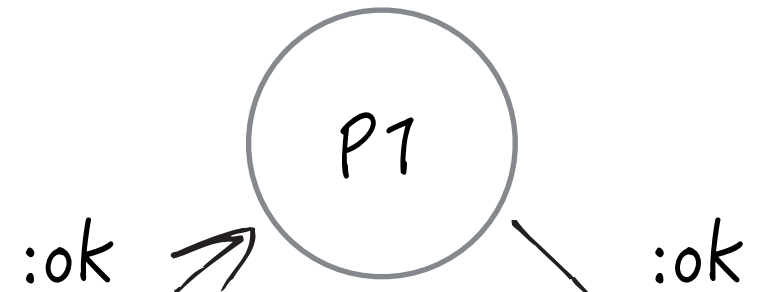
```
iex(2)> :timer.tc(Ring, :start, [10_000])
```

```
{12431, :ok}
```

```
iex(3)> :timer.tc(Ring, :start, [1_000_000])
```

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



```
def start(n) do
  send(self(), :ok)
end

def start(n) do
  spawn(fn -> send(self(), :ok) end)
end

end

end
```

Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
```

```
{1677, :ok}
```

```
iex(2)> :timer.tc(Ring, :start, [10_000])
```

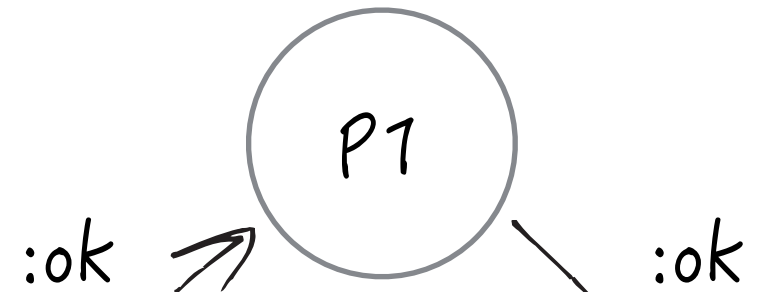
```
{12431, :ok}
```

```
iex(3)> :timer.tc(Ring, :start, [1_000_000])
```

```
{1512394, :ok}
```

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



```
def start(n) do
  send(self(), {n, :ok})
end

def start(n) do
  spawn(fn -> send(self(), {n, :ok}))
end
end
```

Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
```

```
{1677, :ok}
```

```
iex(2)> :timer.tc(Ring, :start, [10_000])
```

```
{12431, :ok}
```

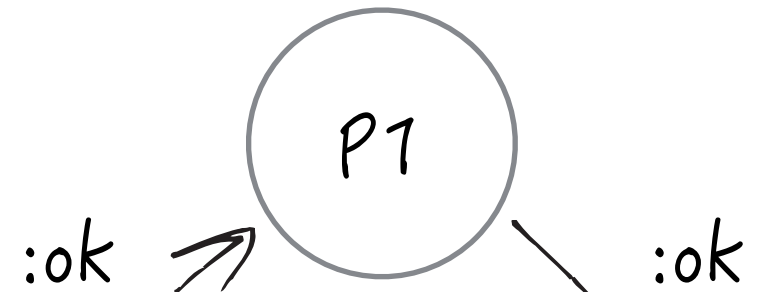
```
iex(3)> :timer.tc(Ring, :start, [1_000_000])
```

```
{1512394, :ok}
```

```
iex(4)> :timer.tc(Ring, :start, [10_000_000])
```

Processes are Cheap

```
defmodule Ring do
  def start(n) do
    start(n, self)
  end
end
```



```
def start(n) do
  send(self(), {n, :ok})
end

def start(n) do
  spawn(fn -> start(n, self))
  receive do
    {_, :ok} -> :ok
  end
end

end
```

Interactive Elixir (1.0.3)

```
iex(1)> :timer.tc(Ring, :start, [1_000])
```

```
{1677, :ok}
```

```
iex(2)> :timer.tc(Ring, :start, [10_000])
```

```
{12431, :ok}
```

```
iex(3)> :timer.tc(Ring, :start, [1_000_000])
```

```
{1512394, :ok}
```

```
iex(4)> :timer.tc(Ring, :start, [10_000_000])
```

```
{15207638, :ok}
```

Process => State

```
defmodule Counter do
  def start(n) do
    spawn(fn -> loop(n) end)
  end

  def down(pid), do: send(pid, :down)

  defp loop(n) do
    IO.puts "Counter at #{n}"
    receive do
      :down when n == 1 ->
        IO.puts "That's it, bye!"
      :down ->
        loop(n - 1)
    ->
      loop(n)
    end
  end
end
```

Process => State

```
defmodule Counter do
  def start(n) do
    spawn(fn -> loop(n) end)
  end

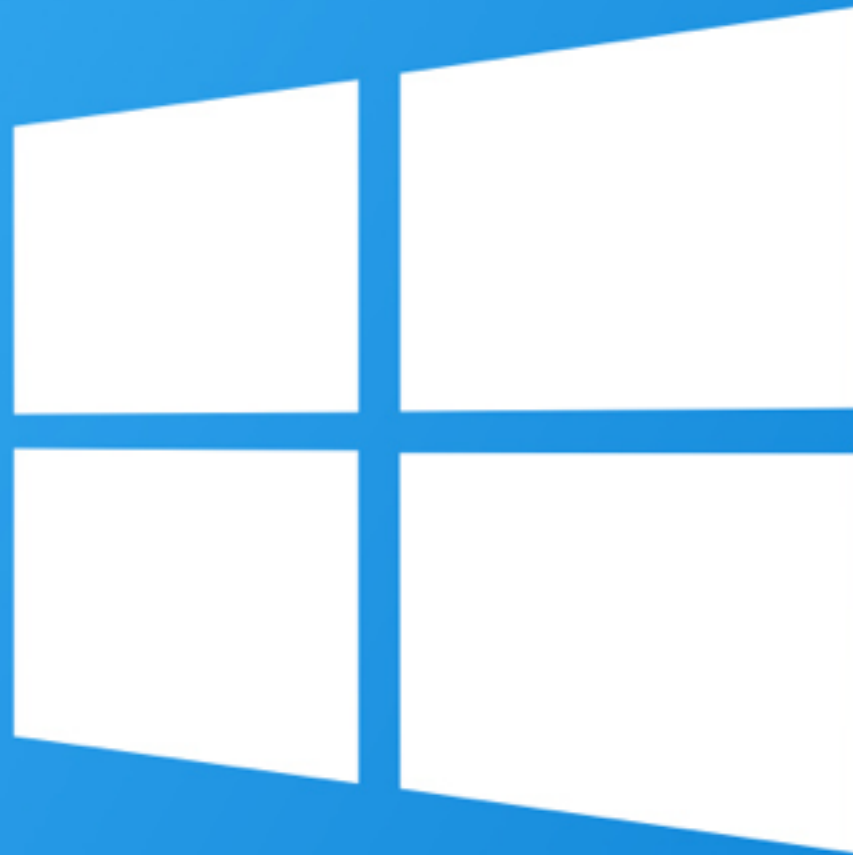
  def down(pid), do: send(pid, :down)

  defp loop(n) do
    IO.puts "Counter at #{n}"
    receive do
      :down when n == 1 ->
        IO.puts "That's it, bye!"
      :down ->
        loop(n - 1)
    ->
      loop(n)
    end
  end
end
```

```
Interactive Elixir (1.0.3)
iex(1)> s = Counter.start(3)
Counter at 3
#PID<0.113.0>
iex(2)> Process.alive? s
true
iex(3)> Counter.down
Counter at 2
:down
iex(4)> Counter.down
Counter at 1
:down
iex(5)> Counter.down
That's it, bye!
:down
iex(2)> Process.alive? s
false
```

Conclusions

Works on Windows



The
Pragmatic
Programmers

Metaprogramming Elixir

Write Less Code,
Get More Done
(and Have Fun!)

Chris McCord
(author of the Phoenix framework)
Edited by Jacquelyn Carter



The
Pragmatic
Programmers

Programming Elixir

Functional
> Concurrent
> Pragmatic
> Fun

Dave Thomas



Elixir IN ACTION

Saša Jurić

MEAP

MANNING



O'REILLY



Introducing Elixir

GETTING STARTED IN FUNCTIONAL PROGRAMMING

Simon St. Laurent & J. David Eisenberg

Erlang

It's here and it's not going away

- Don't be scared but embrace it
- Filled with years of wisdom on building distributed systems
- Don't try to reinvent the wheel



It's very young with few
production ready libraries

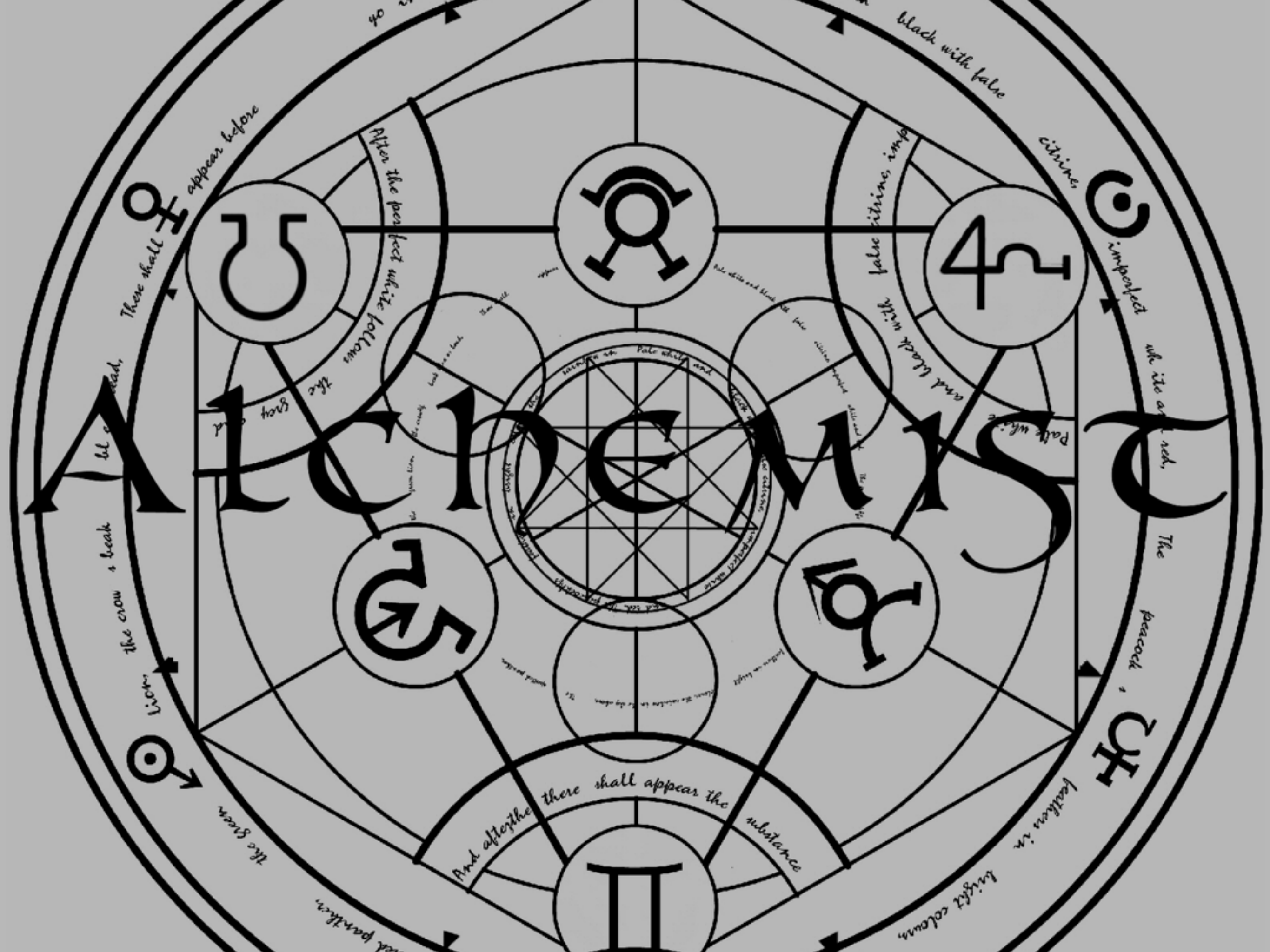
but we can use a lot
of code written in Erlang



Jump in, this is
the future, you can
make the difference,
it will be fun



How are called
the Elixir
programmers?



Questions?

