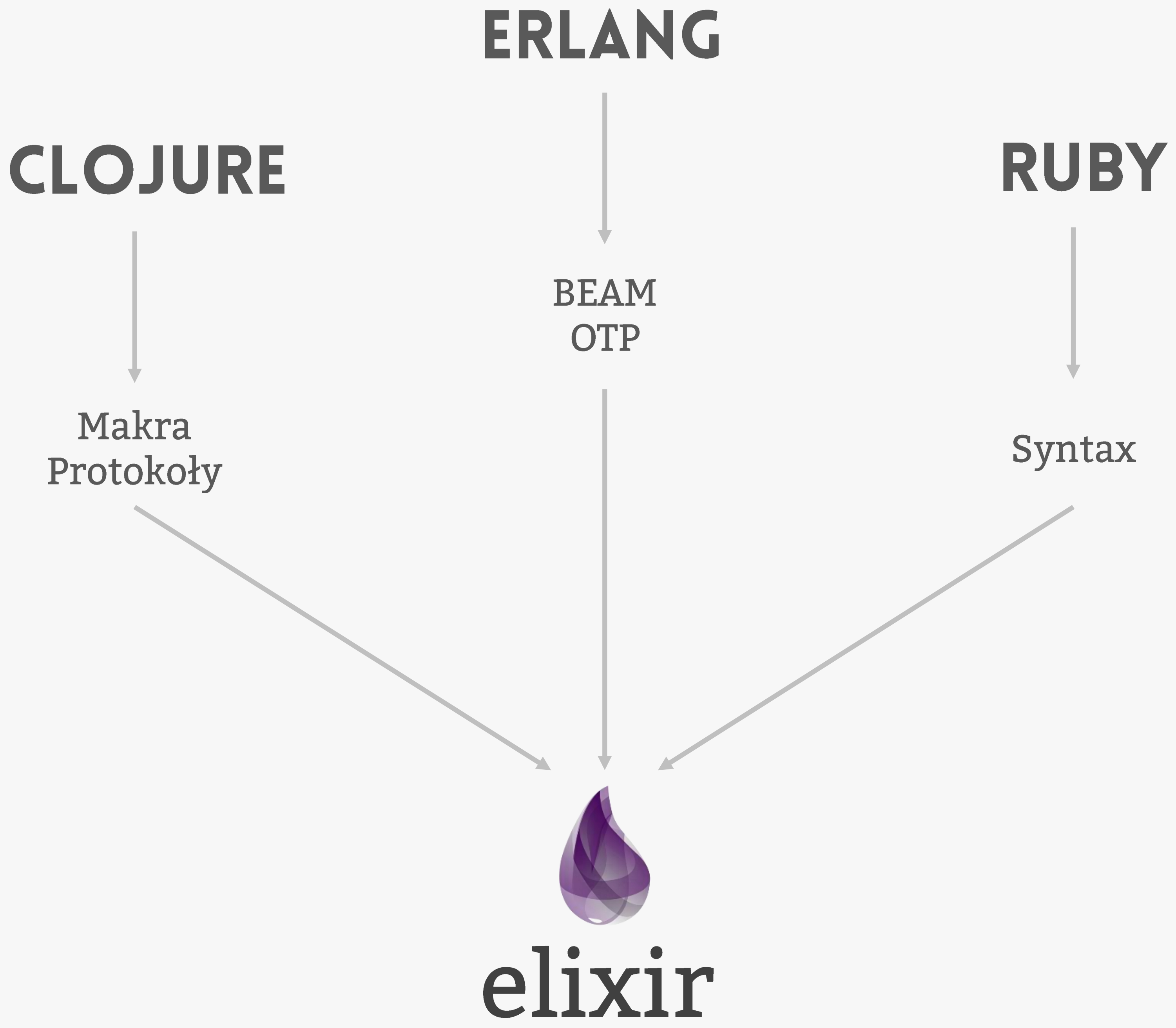
elixir

BEATA OBROK

Elixir is what would happen if Erlang, Clojure, and Ruby somehow had a baby and it wasn’t an accident.

— *Devin Torres*


ERLANG
CLOJURE
RUBY
BEAM
OTP
Makra
Protokoły
Syntax
elixir

elixir

TWÓRCA:	José Valim
LICENCJA:	APACHE License
HISTORIA:	2012: v.0.0.0 2014: v.1.0.0
DOSTĘPNOŚĆ:	Unix Windows Mac

- Erlang VM
- Model aktorów

ZAŁOŻENIA
KOMPATYBILNOŚĆ Z ERLANGIEM

We frequently say that **the Erlang VM is Elixir's strongest asset.**

— *José Valim*

```
Eshell 1> timer:tc(lists, filter,
    [ fun (X) -> X rem 3 == 0 end, lists:seq(1, 1000000) ]).
{3108780,[...]}
```

```
iex(1)> :timer.tc(:lists, :filter,
    [ fn x -> rem(x,3) == 0 end, :lists.seq(1, 1000000) ])
{2789563,[...]}

iex(2)> :timer.tc(Enum, :filter,
    [ 1..1000000, fn x -> rem(x,3) == 0 end ])
{2277837,[...]}
```

- Agent
- GenServer
- GenEvent
- Supervisor
- Application

ZAŁOŻENIA
PRODUKTYWNOŚĆ I ROZSZERZALNOŚĆ

MAKRA

- Większość Elixira jest napisana w Elixirze
  np. `if, case, unless`

```
defmacro unless(clause, expression) do
    quote do
        if(!unquote(clause), do: unquote(expression))
    end
end
```

PROTOKOŁY

- Rozszerzenie funkcjonalności do własnego typu danych

```
list = [1,2,3]
Enum.map list, fn(x) -> x * 2 end
#=> [2,4,6]

range = 1..3
Enum.map range, fn(x) -> x * 2 end
#=> [2,4,6]

set = HashSet.new [1,2,3]
Enum.map set, fn(x) -> x * 2 end
#=> [2,4,6]
```

`Enumerable protocol`

MAKRA

quote wyrażenie ⟶ reprezentacja kodu w Elixirze

```
iex> quote do: sum(1, 2, 3)
{:sum, [], [1, 2, 3]}
```

```
defmodule Unless do
    def fun(clause, expression) do
        if(!clause, do: expression)
    end

    defmacro macro(clause, expression) do
        quote do
            if(!unquote(clause), do: unquote(expression))
        end
    end
end
```

```
iex> Unless.fun true do
...> IO.puts "This should never be printed"
...> end
This should never be printed
nil

iex> Unless.macro true do
...> IO.puts "This should never be printed"
...> end
nil
```

```
defmacro match?(left, right) do
    quote do
        case unquote(right) do
            unquote(left) -> true
            _ -> false
        end
    end
end
```

```
iex> list = [{:a,1},{:b,2},{:a,3}]
[a: 1, b: 2, a: 3]
iex> Enum.filter list, fn (x) -> {:a, _} === x end
** (CompileError) iex:4: unbound variable _
iex> Enum.filter list, fn (x) -> match?({:a, _}, x) end [a: 1, a: 3]
```

ZAŁOŻENIA
METAPROGRAMOWANIE

Kod generujący kod

```
defrecord User, name: nil, age: 0
```

```
defmodule User do
    #initializer
    def new(data) do ... end

    # getters
    def name(user) do ... end
    def age(user) do ... end

    # setters
    def name(value, user) do ... end
    def age(value, user) do ... end
end
```

Kod interpretujący kod

```
ExUnit.start

defmodule MathTest do
    use ExUnit.Case, async: true

    test "adding two numbers" do
        assert 1 + 2 == 4
    end
end
```

```
1) test adding two numbers
   (MathTest)
    examples\exunit.exs:6
    Assertion with == failed
    code: 1 + 2 == 4
    lhs: 3
    rhs: 4
    stacktrace:
        examples\exunit.exs:7
```

Kod jako typ danych

```
iex> contents = quote do
...> defmodule HelloWorld do
...> def hello_world do
...> IO.puts "Hello world!"
...> end
...> end
...> end
...
iex> Code.eval_quoted contents
...
iex> HelloWorld.hello_world
Hello world!
:ok
```

DODATKOWO

- Sets, Dictionaries, Ranges
- „Lazy” Enum = Stream, Pipeline Operator

```
1..100_000 |> Stream.map(&(&1 * 3)) |> Stream.filter(odd?) |> Enum.sum
```

- UTF-8, String Binaries
- Sigils

```
regex = ~r/foo|bar/
```

- Doc Strings

WADY

- Stosunkowo nowy
- Plugin do Intellij
- Małe community
- def … end

```
iex(1)> f = fn(x) -> 2*x end
#Function<6.90072148/1 in :erl_eval.expr/5>
iex(2)> f(1)
** (RuntimeError) undefined function: f/1
iex(3)> f.(1)
2
```

ZALETY

- Szybko się rozwija
- Dobrze udokumentowany
- Tutorial: http://elixir-lang.org/
- Książki (np. Programming Elixir)
- ExUnit
- Mix