

Elixir

Simple Object Orientation and
charming syntax on top of

Elixir

Simple Object Orientation and
charming syntax on top of

ID

José Valim

blog

blog.plataformatec.com

twitter

[@josevalim](https://twitter.com/josevalim)

I am José Valim

@josevalim

I work at 

blog.plataformatec.co

Why Erlang?

Erlang VM

- + Concurrent Processes
- + Message Based
- + Hot Code Swapping
- + Runs with low memory

Erlang Language

- + Small and quick to learn
- + Functional programming
- Harsh syntax
- No object orientation

Elixir

Simple Object Orientation and
charming syntax on top of

1. Why?

2. How?

Why?

The Syntax



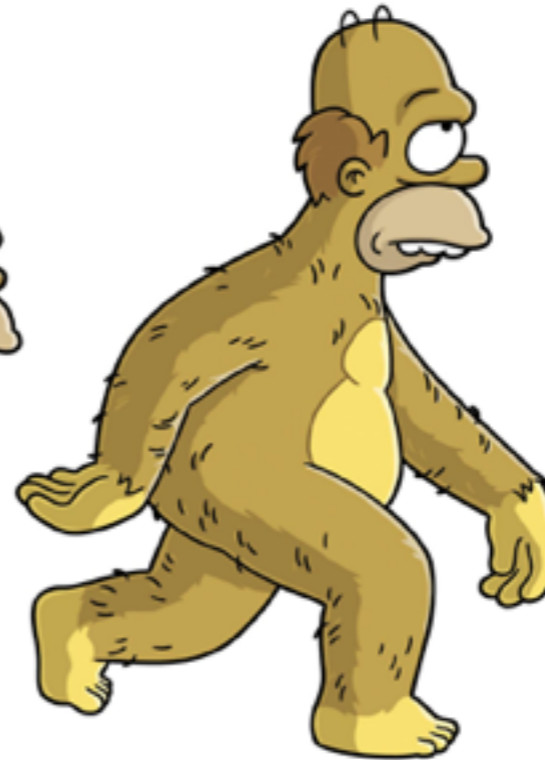
MONKIUS EATALOTIS



CHIMPUS IMBECILUS



APEIS STUPIDIUS



NEANDERSLOB



HOMERSAPIEN

HOMERSAPIEN

MAT 5/20/06

The Syntax

+ Cognitive load

+ Line noise

+ Hidden messages

+ Succinctness

48 | 1532 | 942

48 | 15 | 32 | 942

Line noise

```
-module(dog).  
-export([talk/0]).
```

```
talk() ->  
    io:format("Woof").
```

```
-module(cat).  
-export([talk/0]).
```

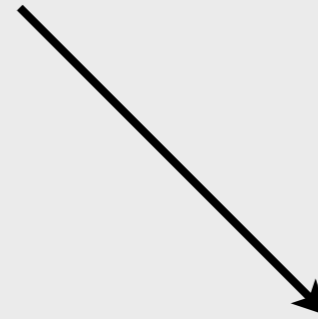
```
talk() ->  
    io:format("Meow").
```

```
[Animal:talk() || Animal <- [dog, cat]].
```

Good line noise



`-module(dog).`



Bad line noise

```
[Animal:talk() || Animal <- [dog, cat]].
```

Bad line noise

Bad line noise

```
[Animal:talk() for Animal in [dog,cat]].
```

`talk(loud) ->`
`io:format("WOOF");`

`talk(_) ->`
`io:format("woof").`

Good line noise

Good line noise

```

animal.cc      (83): #<>{:()=;};;{:()};;{:()}{::<<"\";}:{:()};;
                ::(){:::<<"\";}{()*={,};(=;<;++)[]->()};;
Animal.java   (67): {{()};}{()}{.("");}}{()}{.("");}}([]){[]=[]
                {(),()};(=;<.;++)[].(.);}}
animal.erl    (55): -().-([/]).()->:("").-().-([/]).()->:("").
                [:()||<-[,]].
animal.pl     (41): ;{{};}{"\";};{{};}{"\";};$(->,->){$->()};
animal.py     (23): :():"":():""[(),()]:.()
animal.rb     (10): """"[.,.].

```

<http://onestepback.org/index.cgi/Tech/Ruby/LineNoise.rdoc>

Hidden

```
lists:member?(1, [1,2,3])
```

```
ets:delete!(some_table)
```

Succinctness

- + Succinctness is power
- + The quicker we program, the more we can program
- + You write less code, you see less bugs

“Ulf Wiger of Ericsson did a [study](#) that concluded that Erlang was 4-10x more succinct than C++, and proportionately faster to develop software in.

The study also [...] reveals that programs written in more powerful languages tend to have fewer bugs.”

<http://paulgraham.com/power.html>

Succinct without
sacrificing

```
object Article < ORM
  self.has_many('comments', {'foreign_key': 'article'});
  self.validates_presence_of('title');
end;
```

```
object Article < ORM
  self.has_many('comments', {'foreign_key': 'article'})
  self.validates_presence_of('title')
end
```

```
object Article < ORM
  has_many('comments, {'foreign_key: 'article'})
  validates_presence_of('title')
end
```

```
object Article < ORM
  has_many 'comments, {'foreign_key: 'article}'
  validates_presence_of 'title'
end
```

```
object Article < ORM
  has_many 'comments', 'foreign_key: 'article
  validates_presence_of 'title'
end
```

```
object Article < ORM
  self.has_many('comments', {'foreign_key': 'article'});
  self.validates_presence_of('title');
end;
```

The Object

What does it
mean to be
object

Message passing

```
lists:delete(1, [1,2,3])
```

[1, 2, 3].delete(1)

Succinctness

How?

Examples

% Numbers

3 = 1 + 2

2.0 = 4 / 2

2 = 4 div 2

% + is actually a method call

3 = 1.+(2)

% _ for readability

1_048_576 = 1024 * 1024

```
% Atoms  
'elixir'
```

```
% Strings (are binaries)  
"Elixir"  
"E11x1r" = "E1#{1}x#{1}r"
```

```
% Char list (Erlang strings)  
$"Elixir"
```

```
% Bit strings  
<<1,2,3:16>>
```

% Tuples

{ 1, 2, 3 }

% (Ordered) Dictionaries

{ 'a: 1, 'b: 2 }

% Lists

[1, 2, 3]

[1, 2 | [3]]

```
% Functions
```

```
do_something = -> (x, y)
```

```
    x + y
```

```
end
```

```
% Inline functions
```

```
do_something = -> (x, y) x + y
```

```
% You can also use do
```

```
do_something = do (x, y)
```

```
    x + y
```

```
end
```

```
list = [1,2,3]
```

```
% Method call
```

```
list.delete(1)
```

```
% No need for parens
```

```
list.delete 1
```

```
% Map
```

```
list.map(-> (x) x * 2)
```

```
list.map do (x)
```

```
  x * 2
```

```
end
```

% Variables

x = 1

x = 2

% Boom

~x = 3

% Lists

[h|t] = [1, 2, 3]

[~h|t] = [1, 4, 5]

% Boom

[~h|t] = [3, 4, 5]

```
% hello.ex
module Hello
  def world
    IO.puts "Hello World"
  end
end
```

```
% elixirc hello.ex
% elixir -e "Hello.world"
```

```
% hello.exs
```

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

```
% elixir hello.exs
```

Compilation time
depends on

Meta-

```
module Numbers
```

```
  dict = { 'one: 1, 'two: 2, 'three: 3 }
```

```
  dict.each do (key, value)
```

```
    module_eval "def #{key}; #{value}; end"
```

```
  end
```

```
end
```

```
1 = Numbers.one
```

```
2 = Numbers.two
```

```
3 = Numbers.three
```

```
module Numbers
```

```
  def one
```

```
    1
```

```
  end
```

```
  def two
```

```
    2
```

```
  end
```

```
  def three
```

```
    3
```

```
  end
```

```
end
```

Erlang Abstract Format

`-module(foo).` \rightarrow `{attribute, LINE, module, foo}`

`10` \rightarrow `{integer, LINE, 10}`

`foo` \rightarrow `{atom, LINE, foo}`

`"bar"` \rightarrow `{string, LINE, "bar"}`

`[a, b]` \rightarrow `{cons, LINE, {atom, LINE, a},
 {cons, LINE, {atom, LINE, b},
 {nil, LINE}}}`

A module is defined
by a list of forms
represented in the

Forms are tree-like
tuples that represent
top-level expressions
as attributes and

```
-module(foo).  
-export([bar/0]).
```

```
bar() -> 1 + 2.
```

```
[
  {attribute, 1, module, foo},
  {attribute, 2, export, [{bar, 0}]},
  {function, 4, bar, 0, [
    {clause, 4, [
      {op, 4, '+'},
      {integer, 4, 1}, {integer, 4, 2}
    ]}
  ]}
]
```

Tools

puts

puts

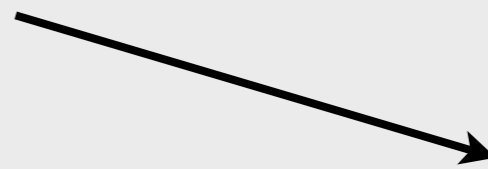


Lexer

puts



Lexer



[{identifier,
1, "puts"},

puts



Lexer



{identifier,
1, "puts"},

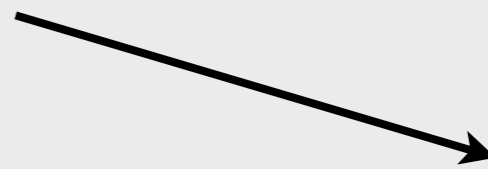


Parse

puts



Lexer



[{identifier,
1, "puts"},



Parse

{call, 1,
{atom, 1, puts},
[{string, 1, "hi"}]
}

puts

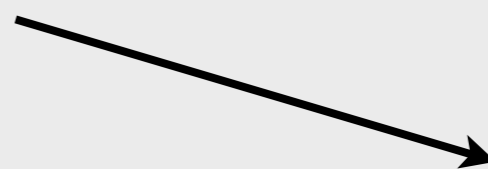
Lexer

[{identifier,
1, "puts"},

Parse

{call, 1,
{atom, 1, puts},
[{string, 1, "hi"}]
}

Transform



puts

Lexer

[{identifier,
1, "puts"},

Parse

{call, 1,
{atom, 1, puts},
[{string, 1, "hi"}]
}

Transform

{call, 1,
{atom,
1, puts},
[{string,

puts

Lexer

[{identifier,
1, "puts"},

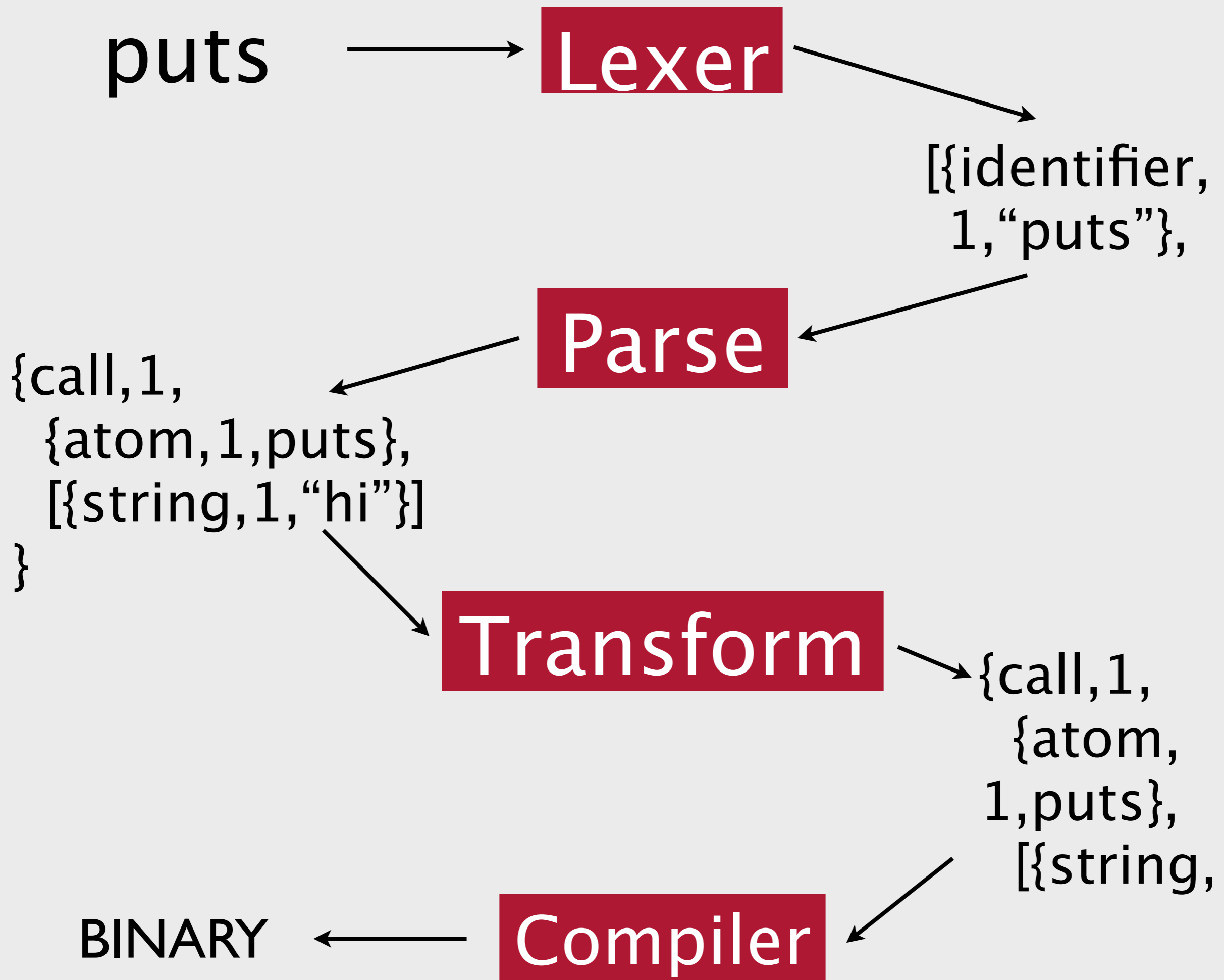
Parse

{call, 1,
{atom, 1, puts},
[{string, 1, "hi"}]
}

Transform

{call, 1,
{atom,
1, puts},
[{string,

Compiler



Lexer



leex

Parse



yecc

Transform



erlang code

Compiler



compile:forms

Transformation

Ambiguity in variables

```
def foo
  1
end
```

```
def bar
  foo
end
```

```
def baz
  foo = 2
  foo
end
```

```
def baz
  foo = 2
  foo
end
```

```
Tokens = [  
  { identifier, 1, foo },  
  { '=', 1 },  
  { integer, 1, 2 },  
  { eol, 1 },  
  { identifier, 2, foo }  
]
```

```
Tree = [  
  {match, 1,  
    {var, 1, foo},  
    {integer, 1, 2}  
  },  
  {identifier, 2, foo}  
]
```

```
transform_tree(Tree) ->  
  lists:mapfoldl(fun transform/2, [], Tree).
```

```
Tree = [  
  {match, 1,  
    {var, 1, foo},  
    {integer, 1, 2}  
  },  
  {identifier, 2, foo}  
]
```

```
transform({match, Line, Left, Right}, Scope) ->  
  { TLeft, LS } = transform(Left, Scope),  
  { TRight, RS } = transform(Right, Scope),  
  { { match, Line, TLeft, TRight }, merge(LS, RS) }.
```

`transform({match, Line, Left, Right}, Scope) ->`
`{ TLeft, LS } = transform(Left, Scope),`
`{ TRight, RS } = transform(Right, Scope),`
`{ { match, Line, TLeft, TRight }, merge(LS, RS) }.`

`transform({var, _Line, Name} = Var, Scope) ->`
`{ Var, [Name | Scope] }.`

`transform({integer, _Line, _Name} = Integer, Scope) ->`
`{ Integer, Scope }.`

{match, 1, {var, 1, foo}, {integer, 1, 2}}, []



transform({match, Line, Left, Right}, Scope) ->
{ TLeft, LS } = transform(Left, Scope),
{ TRight, RS } = transform(Right, Scope),
{ { match, Line, TLeft, TRight }, merge(LS, RS) }.



{match, 1, {var, 1, foo}, {integer, 1, 2}}, [foo]

```
Tree = [  
    {match, 1,  
        {var, 1, foo},  
        {integer, 1, 2}  
    },  
    {identifier, 2, foo}  
]
```

```
transform({identifier, Line, Name}, Scope) ->
  case lists:member(Name, Scope) of
    true -> { {var, Line, Name}, Scope };
    false -> {
      {call, Line, {atom, Line, Name}, []},
      Scope
    }
  end.
```

{identifier, 2, foo}, [foo]



```
transform({identifier, Line, Name}, Scope) ->  
case lists:member(Name, Scope) of  
  true -> { {var, Line, Name}, Scope };  
  false -> {  
    {call, Line, {atom, Line, Name}, []},  
    Scope  
  }  
end.
```



{var, 2, foo}, [foo]

Multi-assignment variables

x = 0

x

y = 'a

y

x = 1

x

y = 'b

y

x = 0

x

y = 'a'

y

x = 1

x

y = 'b'

y



x = 0

x

y = 'a'

y

x = 1

x

y = 'b'

y



x = 0

x

y = 'a'

y

V0 = 1

V0

V1 = 'b'

V1

Meta-

```
module Foo
```

```
  1 + 2
```

```
  def bar
```

```
    1
```

```
  end
```

```
end
```

```
module Foo
```

```
  1 + 2
```

```
  def bar
```

```
    1
```

```
  end
```

```
end
```



```
module Foo
  1 + 2

  def bar
    1
  end
end
```



```
-module(exFoo).

baz(Self) ->
  1.
```

```
module Foo
  1 + 2

  def bar
    1
  end
end
```

?



```
-module(exFoo).
baz(Self) ->
  1.
```

```
module Foo
  1 + 2

  def bar
    1
  end
end
```

```
module Foo
  1 + 2

  def bar
    1
  end
end
```

```
elixir:object('Foo', fun(Self) ->
  1 + 2,
  elixir:method('Foo', bar, 0, [{integer, 5, 1}])
end)
```

```
elixir:object('Foo', fun(Self) ->
  1 + 2,
  elixir:method('Foo', bar, 0, [{integer, 5, 1}])
end)
```

% Methods

```
elixir:object(Name, Function) ->
  ets:new(Name),
  Object = #elixir_object{name=Name},
  Function(Object),
  compile(Object).
```

```
elixir:method(Name, Method, Arity, Clause) ->
  ets:insert(Name, { Method, Arity, Clause }).
```

```
module Foo
  1 + 2

  def bar
    1
  end
end
```

```
module Foo
```

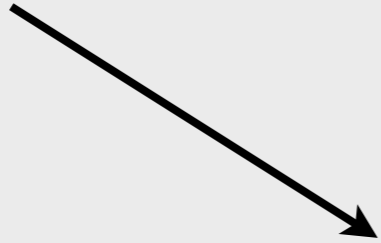
```
  1 + 2
```

```
  def bar
```

```
    1
```

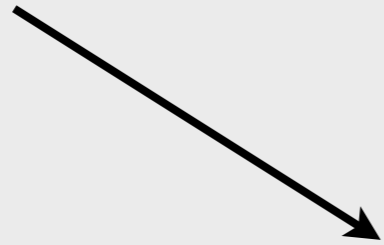
```
  end
```

```
end
```



```
module Foo
  1 + 2

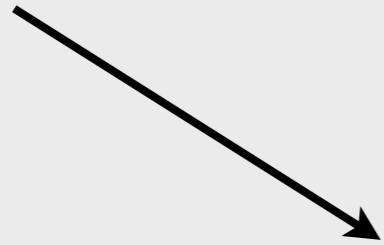
  def bar
    1
  end
end
```



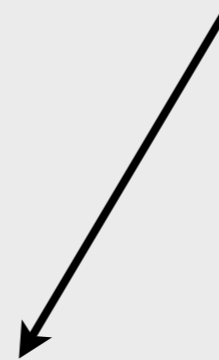
```
elixir:object('Foo', fun(Self) ->
  1 + 2,
  elixir:method('Foo',
    bar, 0, [{integer, 5, 1}])
end)
```

```
module Foo
  1 + 2

  def bar
    1
  end
end
```



```
elixir:object('Foo', fun(Self) ->
  1 + 2,
  elixir:method('Foo',
    bar, 0, [{integer, 5, 1}])
end)
```

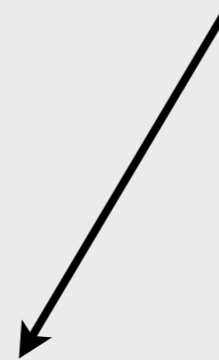


```
module Foo
  1 + 2

  def bar
    1
  end
end
```



```
elixir:object('Foo', fun(Self) ->
  1 + 2,
  elixir:method('Foo',
    bar, 0, [{integer, 5, 1}])
end)
```



```
-module(exFoo).
bar(Self) -> 1.
```

What if?

Simpler Generators

[X * 2 || X <- [1,2,3]]

[X * 2 || <<X>> <=< [1,2,3]>>]



$[X * 2 \mid \mid X <- [1, 2, 3]]$



$[X * 2 \mid \mid \langle \langle X \rangle \rangle \langle = \langle \langle 1, 2, 3 \rangle \rangle]$

Why not make it
implicit?

public/private

```
-module(foo).  
-export([bar/0]).
```

```
bar() -> 1.
```

```
% Internal methods
```

```
baz() -> 2.
```

-module(foo).

bar() -> 1.

-private().

baz() -> 2.

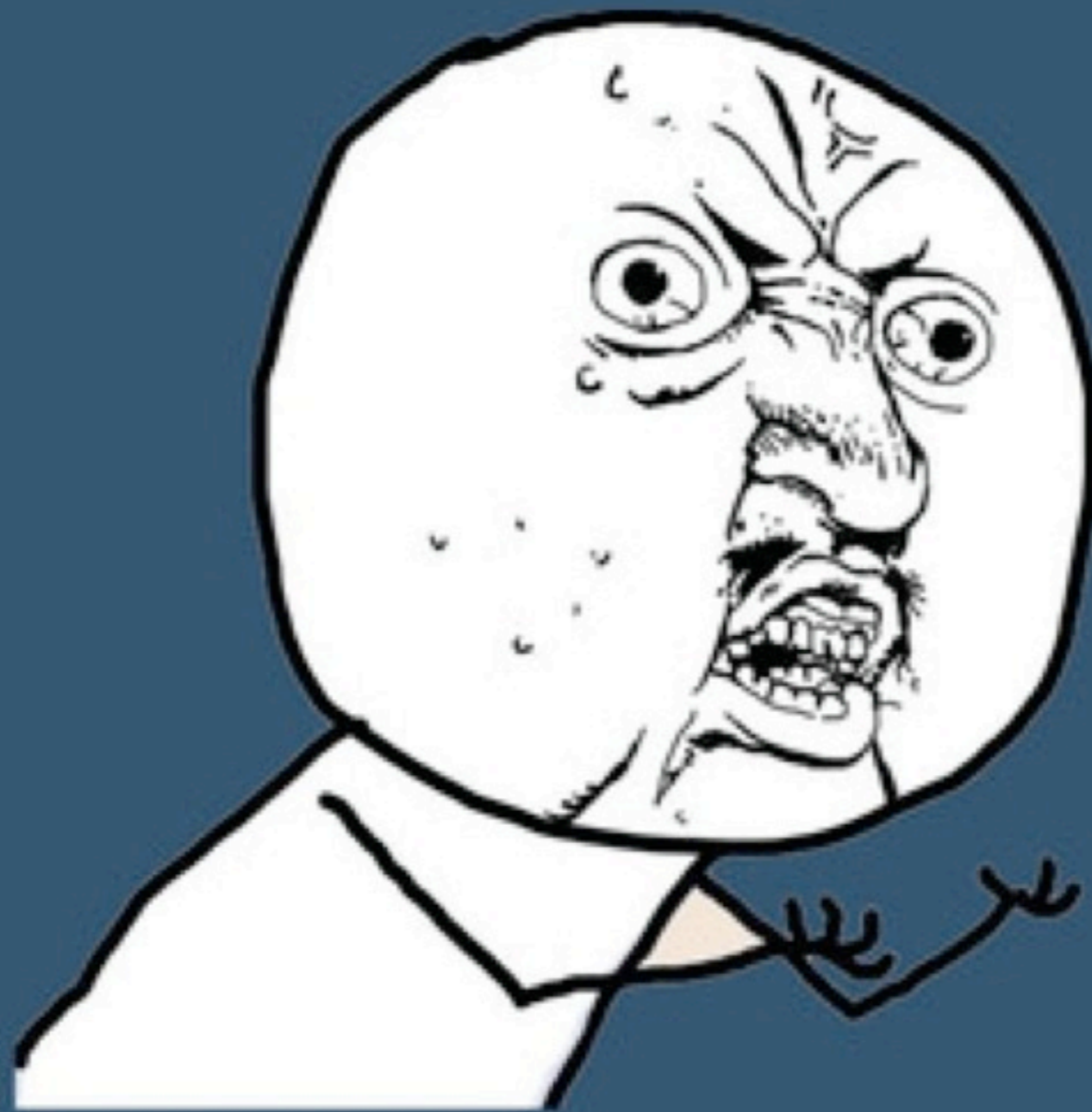
Default

reverse(List) ->
reverse(List, []);

reverse([H|T], Acc) ->
reverse(T, [H|Acc]).

reverse([H|T], Acc := []) ->
reverse(T, [H|Acc]).

ERLANG VM



Y U NO LINES IN BACKTRACE?

Questions?

github.com/josevalim/

Questions?

github.com/josevalim/

ID

José Valim

blog

blog.plataformatec.com

twitter

[@josevalim](https://twitter.com/josevalim)