Good Morning Big Elixirers!

Who am I?

Just a guy with a laptop and too much free time...

- https://github.com/bechurch
- https://twitter.com/bnchrch
- https//ben.church
- https://fission.codes

Fission

- Serverless framework
- Offline by default
- User Controlled Data
- Zero Config Deployment
- https://fission.codes

Let Business Write Business Logic!

A talk about Erlangs lexer and parser generators

A Tale of Two Libraries

1. **Lexer generation with Erlang's leex library**
2. **Parser generation with Erlangs yecc library**

Basically how to define a language by going from a sequence of characters to actionable logic.

Permission


ViralHog

1 The Long winded intro

You are here

2 The Problem

3 The Libraries

*You were hoping you were here

4 Live Code, Danger Zone

Intertubes Broken...

Be Back in a Moment...

first....

My Problems

i.e.

The Big Elixir Does Group Therapy

Appointment Surveys

1. An appointment is made
2. We determine what survey to send
3. We send the survey

Linear Complexity

- Species
- Age / Stage of Life
- Breed
- Appointment Type
- Clinic
- Vet
- Time of Day
- Season

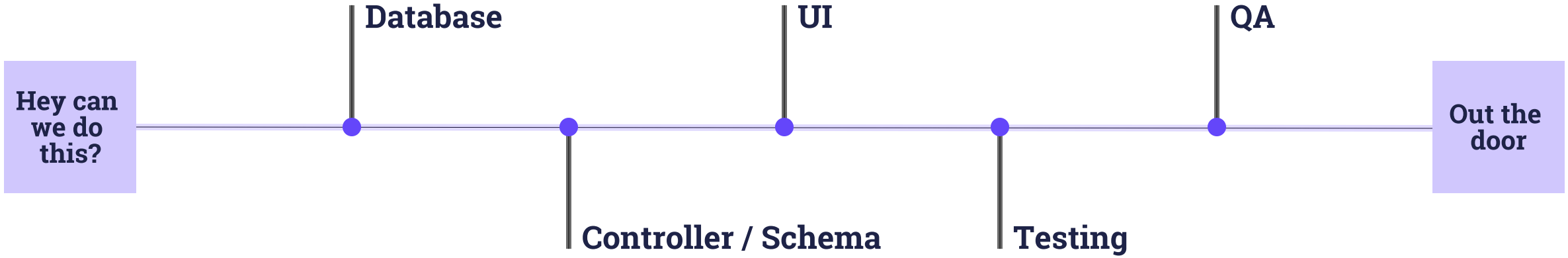
@#$%!

Why did I agree to that?!

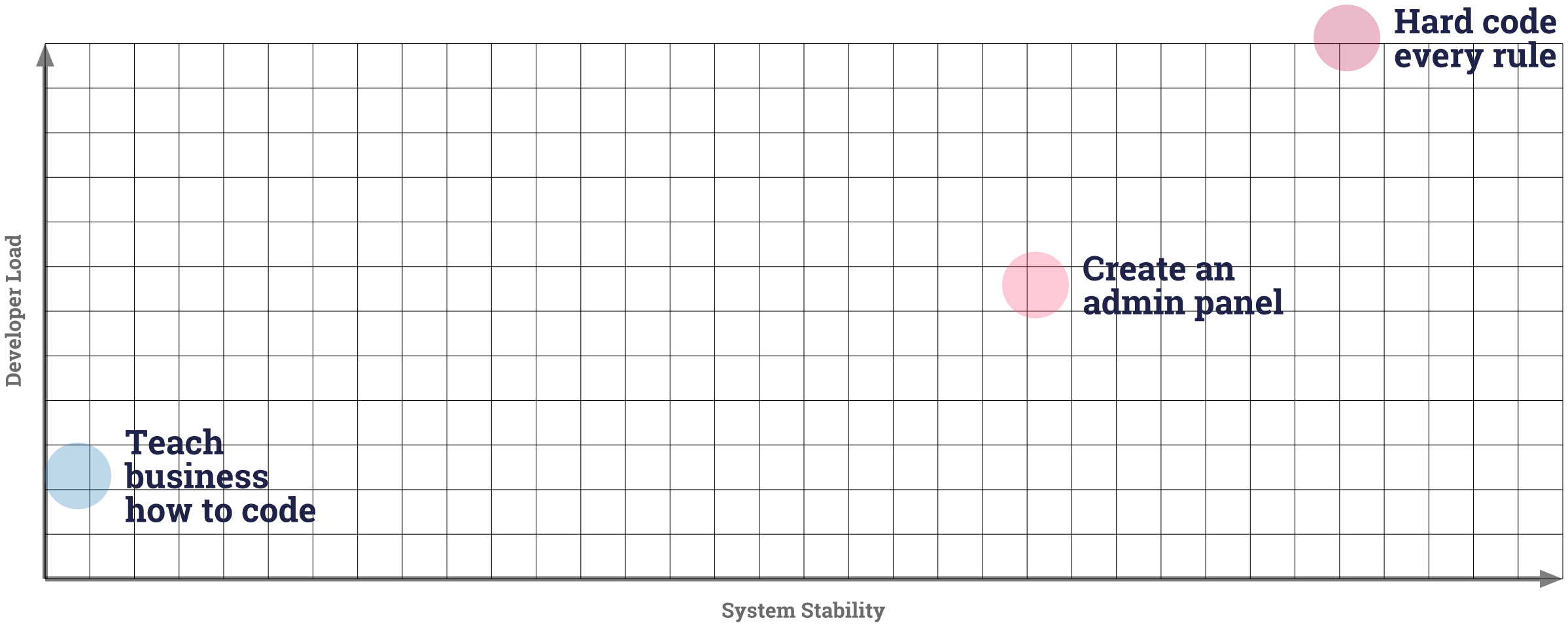
Exponential Complexity

- Species
- Age / Stage of Life
- Breed
- Appointment Type
- Clinic
- Vet
- Time of Day
- Season

- AND
- OR
- IN
- NOT
- >, <
- =, !=
- ()

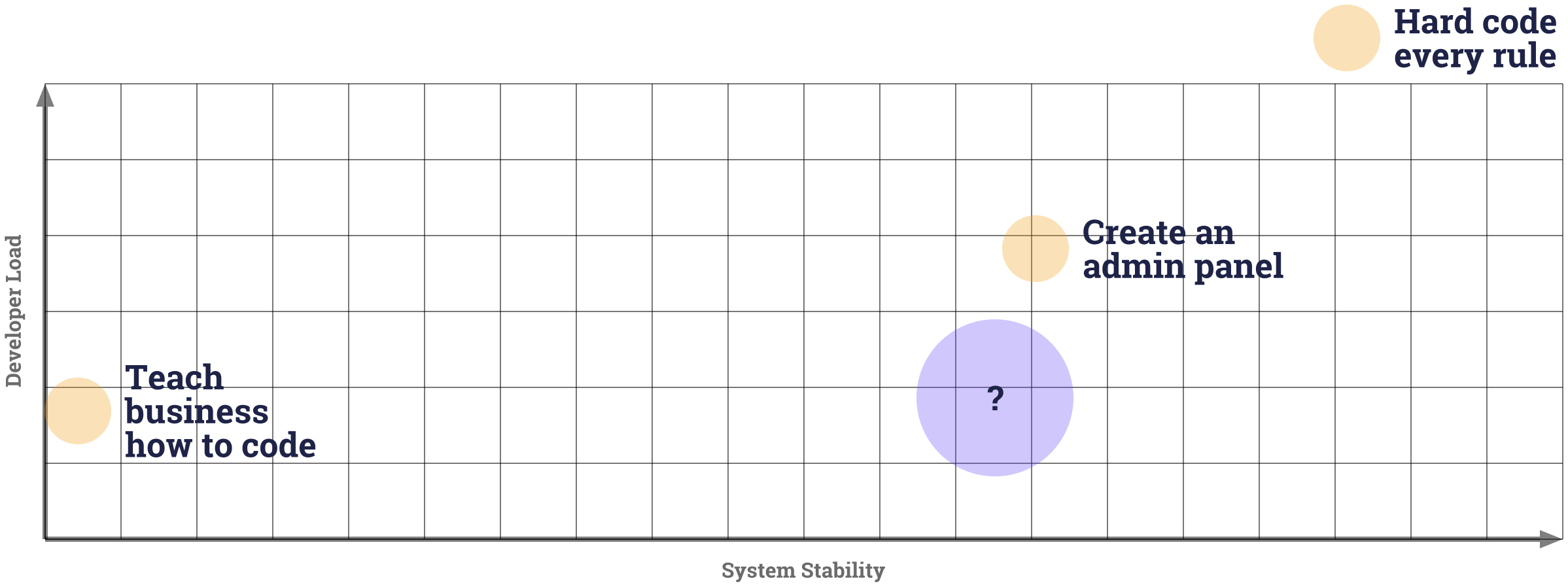

Hey can we do this?
Database
Controller / Schema
UI
Testing
QA
Out the door

How could we solve this?



Think Harder Ben....

What if there was something else?



Something that...

- Let business define arbitrary rules
- Without jeopardizing the system
- While allowing for minimal development effort and maintenance

A Unicorn

Erlang's

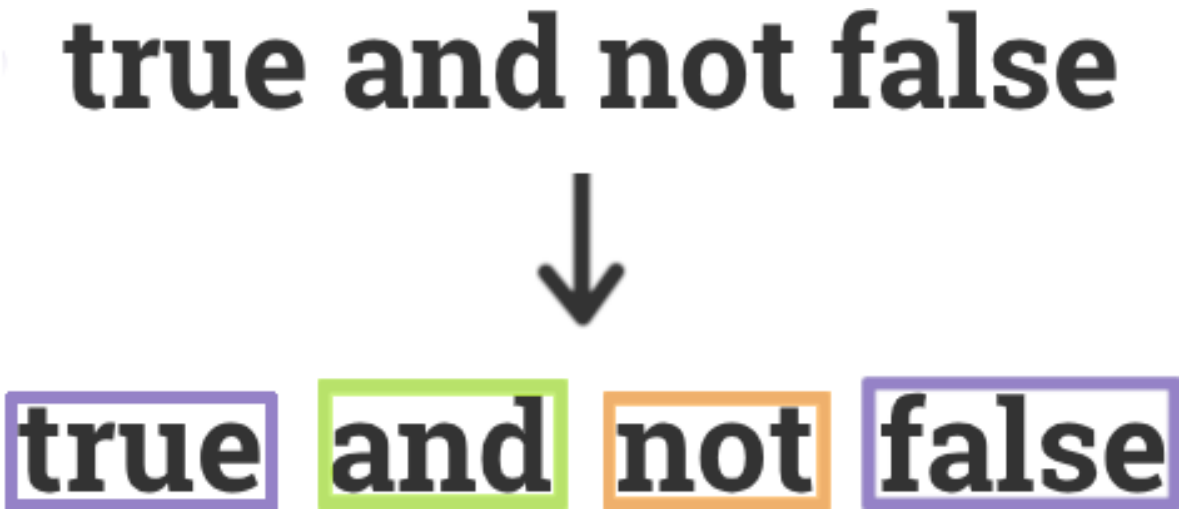
Leex and Yecc to the rescue!

```
evaluate(
  "not email_disabled and ((pet_age > 4.1 and clinic_id = 123) or (appointment_type in (1,2,3)))",
  %{"email_disabled" => false, "pet_age" => 4.2, "clinic_id" => 124, "appointment_type" => 1}
)

{:ok, true}
```

Wait....but what are they and what do they do?

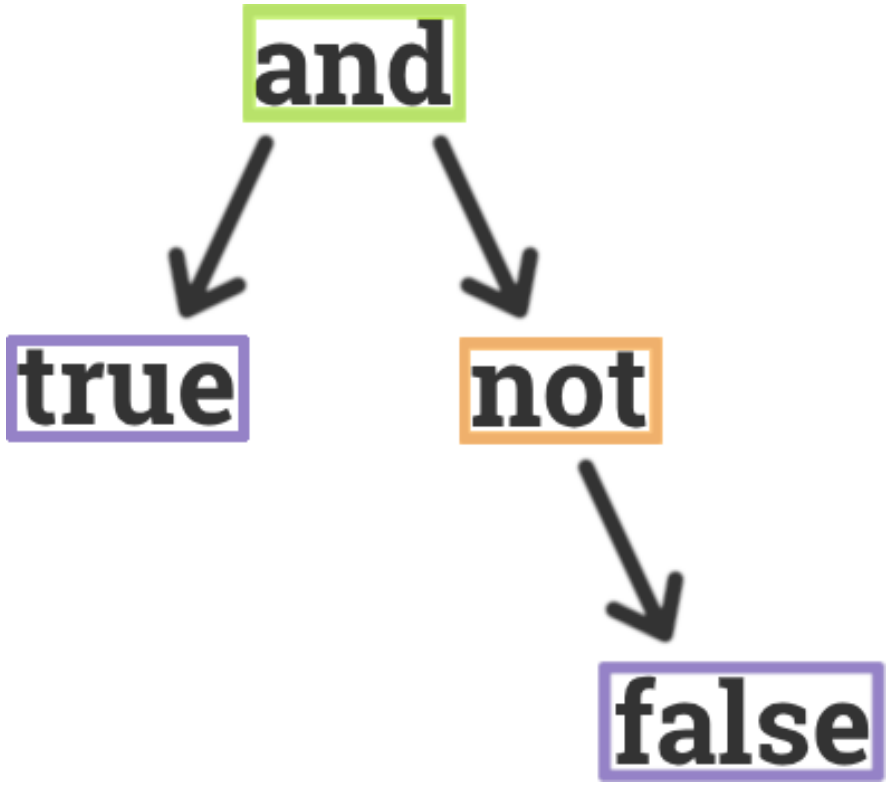
Leex

- Leex is a lexical analyzer generator for Erlang
- Included in the base OTP distribution
- Input is a String
- Outputs a format that can be grammatically parsed (BNF)



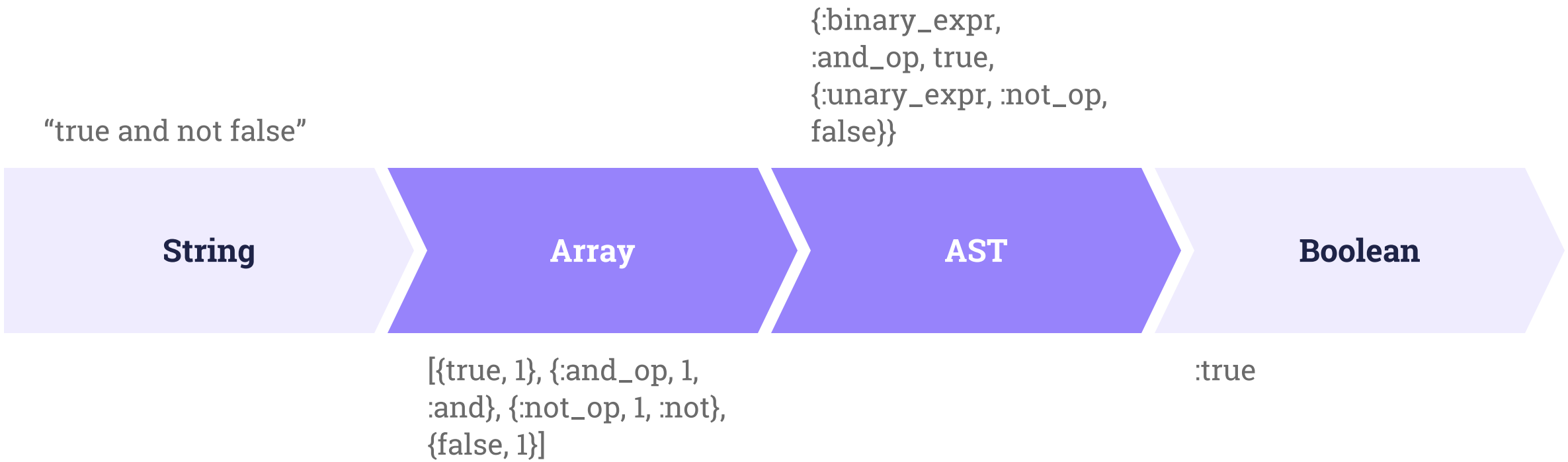
Yecc

- Yecc is a parser generator for Erlang
- Included in the base OTP distribution
- Input is a BNF grammar definition
- Outputs Erlang code for a parser.



Our system as **Stages**

Our system as **Technologies**

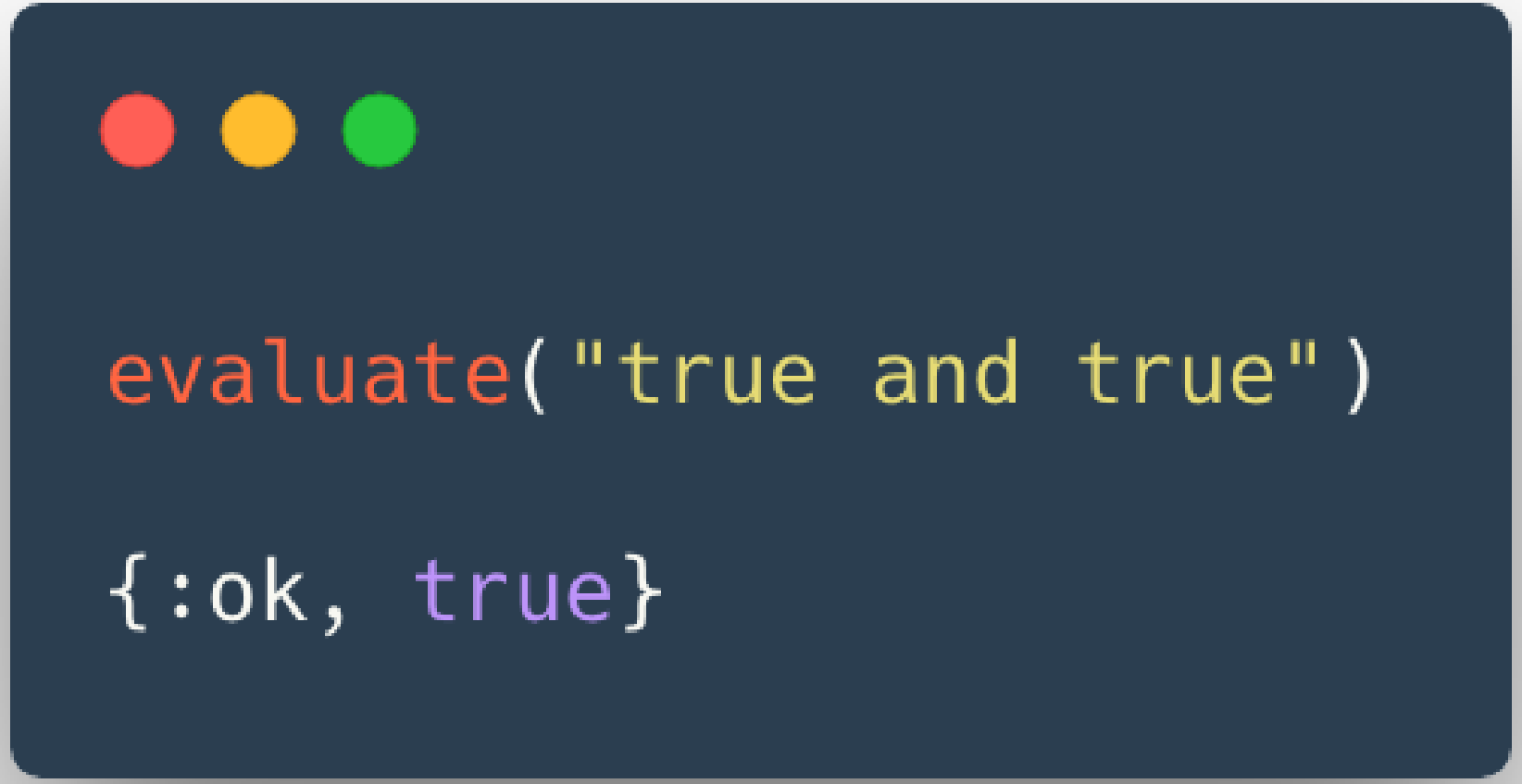


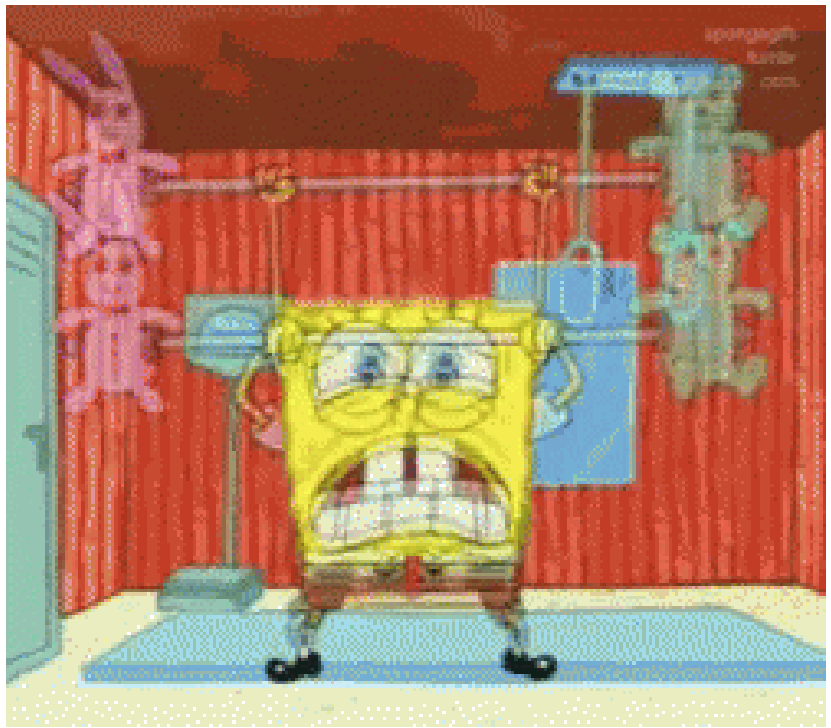
Our system as **Data**

We're headed here....

```
eval(
  "not email_disabled and ((pet_age > 4.1 and clinic_id = 123) or (appointment_type in (1,2,3)))",
  %{"email_disabled" => false, "pet_age" => 4.2, "clinic_id" => 124, "appointment_type" => 1}
)

{:ok, true}
```

...but lets begin here





1.

Keepin' it simple

```
LexerParser.evaluate('true and false')
{:ok, false}

LexerParser.evaluate('true and true')
{:ok, true}

LexerParser.evaluate('false')
{:ok, false}
```

Quick Setup

```
mix new lexer_parser
cd lexer_parser
mix deps.get
mkdir src
touch src/business_lexer.xrl
touch src/business_parser.yrl
```

Applying meaning to a sequence of characters

business_lexer.xrl

```
1   Definitions.
2   WS    = ([\000-\s]|%.*)
3
4   Rules.
5   false   : {token, {false,   TokenLine}}.
6   true    : {token, {true,    TokenLine}}.
7
8   and     : {token, {and_op,  TokenLine, list_to_atom(TokenChars)}}.
9
10  {WS}+   : skip_token.
11
```

src/business_lexer.xrl

business_lexer.xrl

```
1   Definitions.
2   WS    = ([\000-\s]|%.*)
3
4   Rules.
5   false   : {token, {false,   TokenLine}}.
6   true    : {token, {true,    TokenLine}}.
7
8   and     : {token, {and_op,  TokenLine, list_to_atom(TokenChars)}}.
9
10  {WS}+   : skip_token.
```

src/business_lexer.xrl - **Definitions**

```
1   Definitions.
2   WS    = ([\000-\s]|%.*)
3
4   Rules.
5   false   : {token, {false,   TokenLine}}.
6   true    : {token, {true,    TokenLine}}.
7
8   and     : {token, {and_op,  TokenLine, list_to_atom(TokenChars)}}.
9
10  {WS}+   : skip_token.
11
```

src/business_lexer.xrl - **Rules.**

Run it!

```
iex(1)> :business_lexer.string('true')
{:ok, [true: 1], 1}

iex(2)> :business_lexer.string('false')
{:ok, [false: 1], 1}

iex(2)> :business_lexer.string('true and false')
{:ok, [{true, 1}, {:and_op, 1, :and}, {false, 1}], 1}

iex(4)> :business_lexer.string('and')
{:ok, [{:and_op, 1, :and}], 1}

iex(5)> :business_lexer.string('doesnt exist')
{:error, {1, :business_lexer, {:illegal, 'd'}}, 1}
```

src/business_lexer.xrl - **Run it!**

Applying meaning to a sequence of tokens

business_parser.yrl

```
1   Nonterminals expression bool.
2
3   Terminals and_op true false.
4
5   Rootsymbol expression.
6
7   expression -> bool : '$1'.
8   expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
9
10  bool -> true : true.
11  bool -> false : false.
12
```

src/business_parser.yrl

business_parser.yrl

```
1   Nonterminals expression bool.
2
3   Terminals and_op true false.
4
5   Rootsymbol expression.
6
7   expression -> bool : '$1'.
8   expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
9
10  bool -> true : true.
11  bool -> false : false.
12
```

src/business_parser.yrl - **Nonterminals**

business_parser.yrl

```
1   Nonterminals expression bool.
2
3   Terminals and_op true false.
4
5   Rootsymbol expression.
6
7   expression -> bool : '$1'.
8   expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
9
10  bool -> true : true.
11  bool -> false : false.
12
```

src/business_parser.yrl - **Terminals**

business_parser.yrl

```
1   Nonterminals expression bool.
2
3   Terminals and_op true false.
4
5   Rootsymbol expression.
6
7   expression -> bool : '$1'.
8   expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
9
10  bool -> true : true.
11  bool -> false : false.
12
```

src/business_parser.yrl - **Rootsymbol**

business_parser.yrl

```
1   Nonterminals expression bool.
2
3   Terminals and_op true false.
4
5   Rootsymbol expression.
6
7   expression -> bool : '$1'.
8   expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
9
10  bool -> true : true.
11  bool -> false : false.
12
```

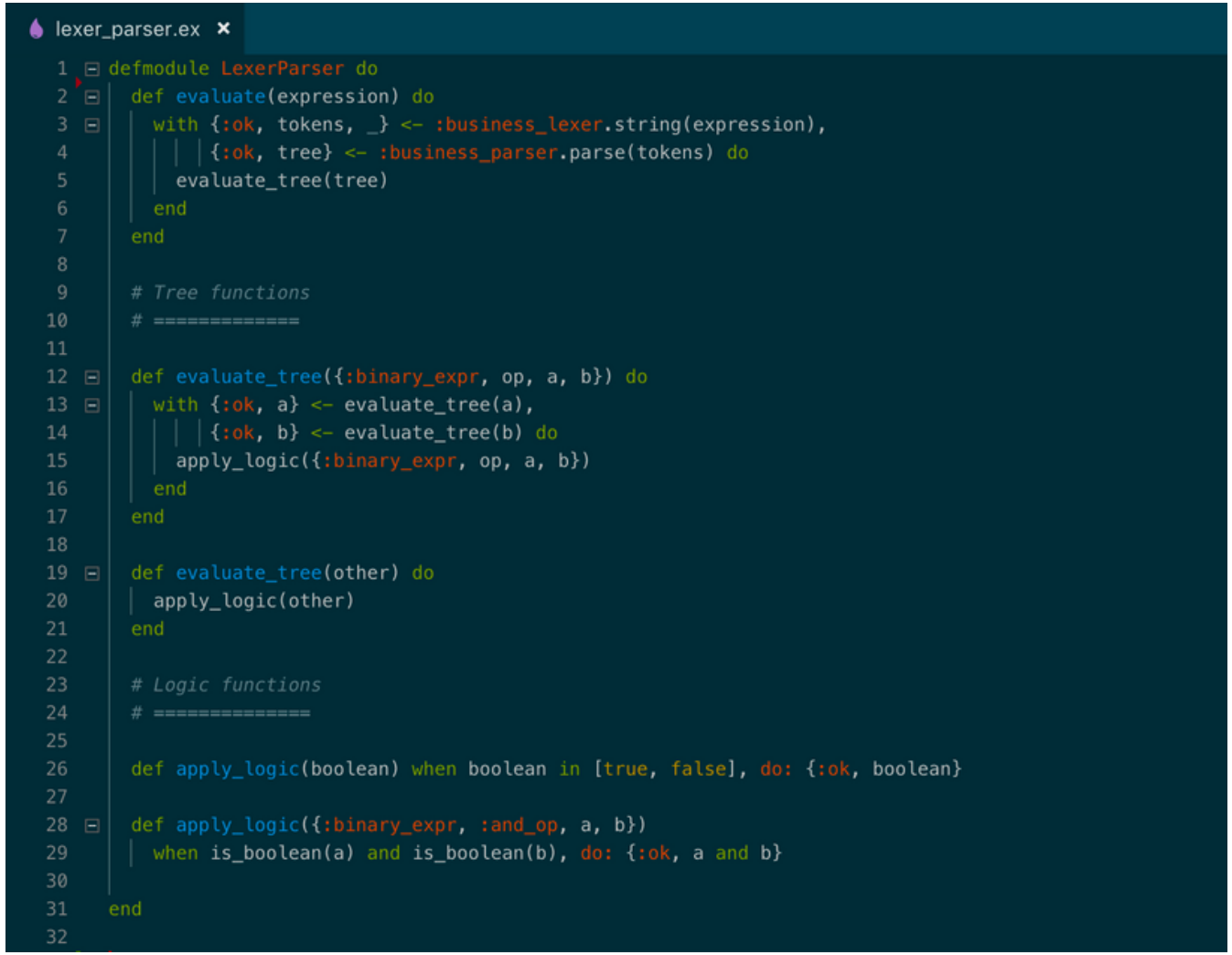
src/business_parser.yrl - **Grammar Rules**

Run it!

```
iex(1)> 'true'
    |> :business_lexer.string()
    |> fn {:ok, tokens, _} -> :business_parser.parse(tokens) end.()

{:ok, true}

iex(2)> 'false'
    |> :business_lexer.string()
    |> fn {:ok, tokens, _} -> :business_parser.parse(tokens) end.()

{:ok, false}

iex(3)> 'true and false'
    |> :business_lexer.string()
    |> fn {:ok, tokens, _} -> :business_parser.parse(tokens) end.()

{:ok, {:binary_expr, :and_op, true, false}}

iex(4)> 'true and true and true'
    |> :business_lexer.string()
    |> fn {:ok, tokens, _} -> :business_parser.parse(tokens) end.()

{:ok, {:binary_expr, :and_op, true, {:binary_expr, :and_op, true, true}}}

iex(5)> 'and'
    |> :business_lexer.string()
    |> fn {:ok, tokens, _} -> :business_parser.parse(tokens) end.()

{:error, {1, :business_parser, ['syntax error before: ', ['\'and\'']]}}
```

src/business_parser.yrl - **Run it!**

Reducing to an answer

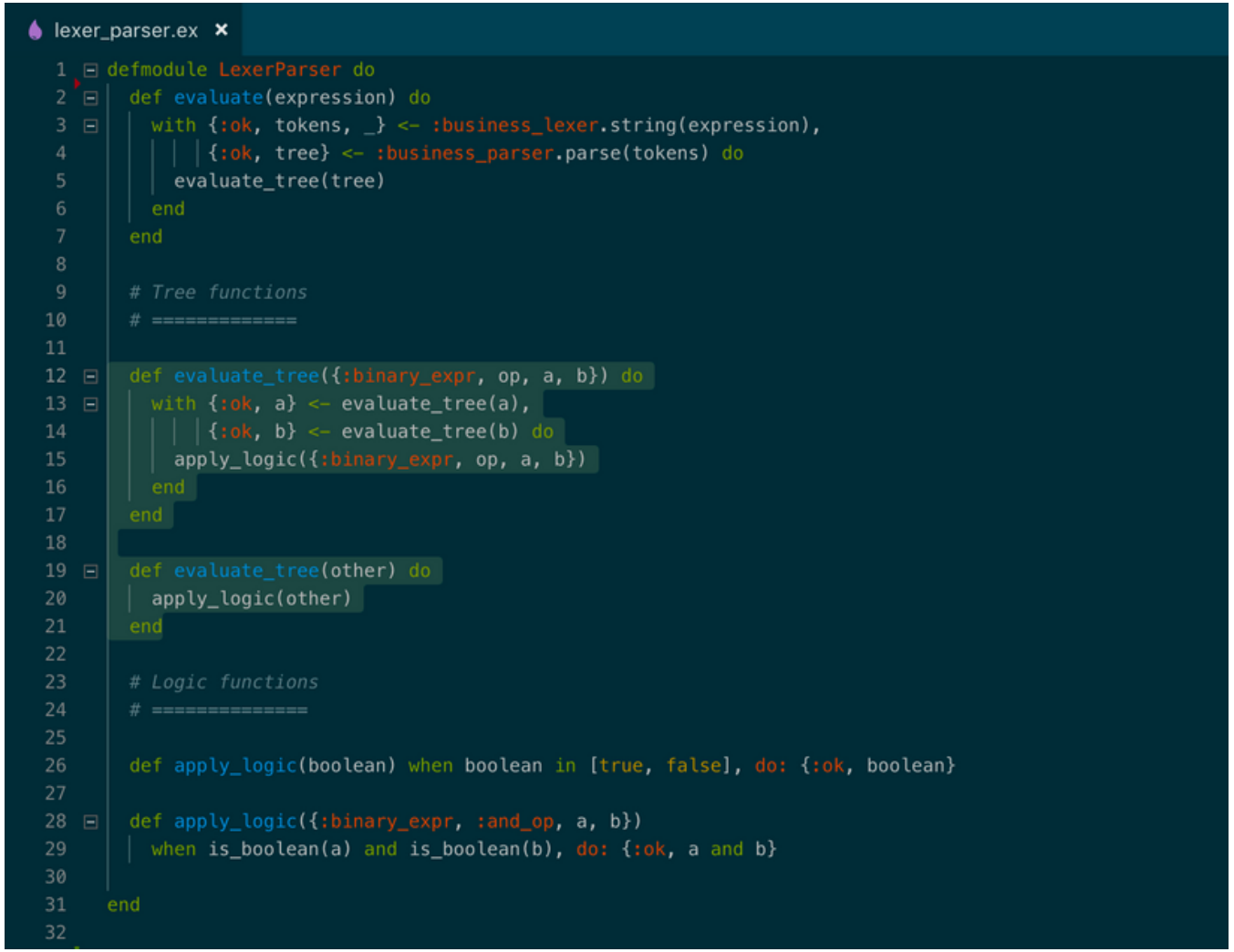


lib/lexer_parser.ex

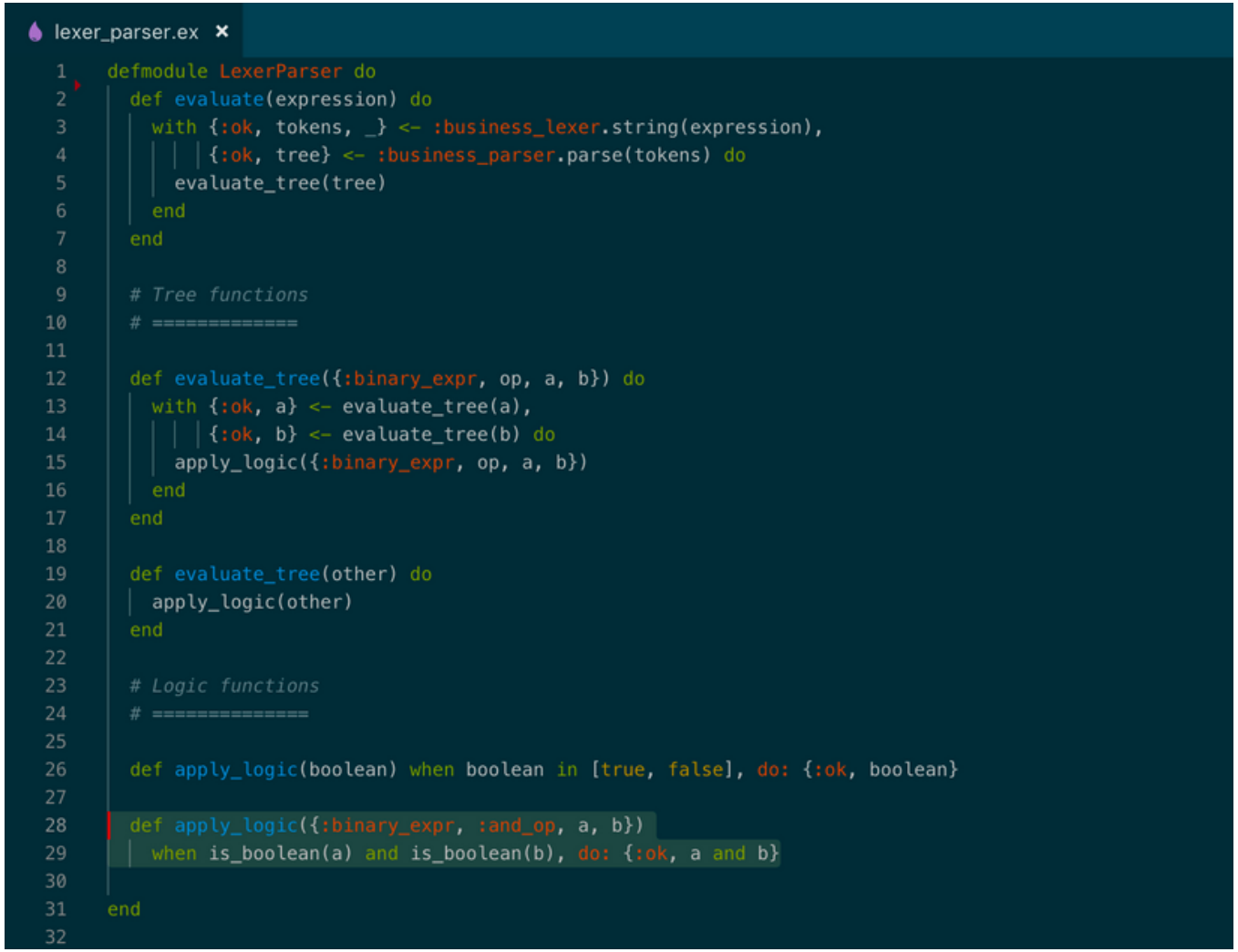
lexer_parser.ex

```elixir
defmodule LexerParser do
  def evaluate(expression) do
    with {:ok, tokens, _} <- :business_lexer.string(expression),
         {:ok, tree} <- :business_parser.parse(tokens) do
      evaluate_tree(tree)
    end
  end

  # Tree functions
  # ==============

  def evaluate_tree({:binary_expr, op, a, b}) do
    with {:ok, a} <- evaluate_tree(a),
         {:ok, b} <- evaluate_tree(b) do
      apply_logic({:binary_expr, op, a, b})
    end
  end

  def evaluate_tree(other) do
    apply_logic(other)
  end

  # Logic functions
  # ===============

  def apply_logic(boolean) when boolean in [true, false], do: {:ok, boolean}

  def apply_logic({:binary_expr, :and_op, a, b})
    when is_boolean(a) and is_boolean(b), do: {:ok, a and b}

end
```

lib/lexer_parser.ex - **evaluate/1**

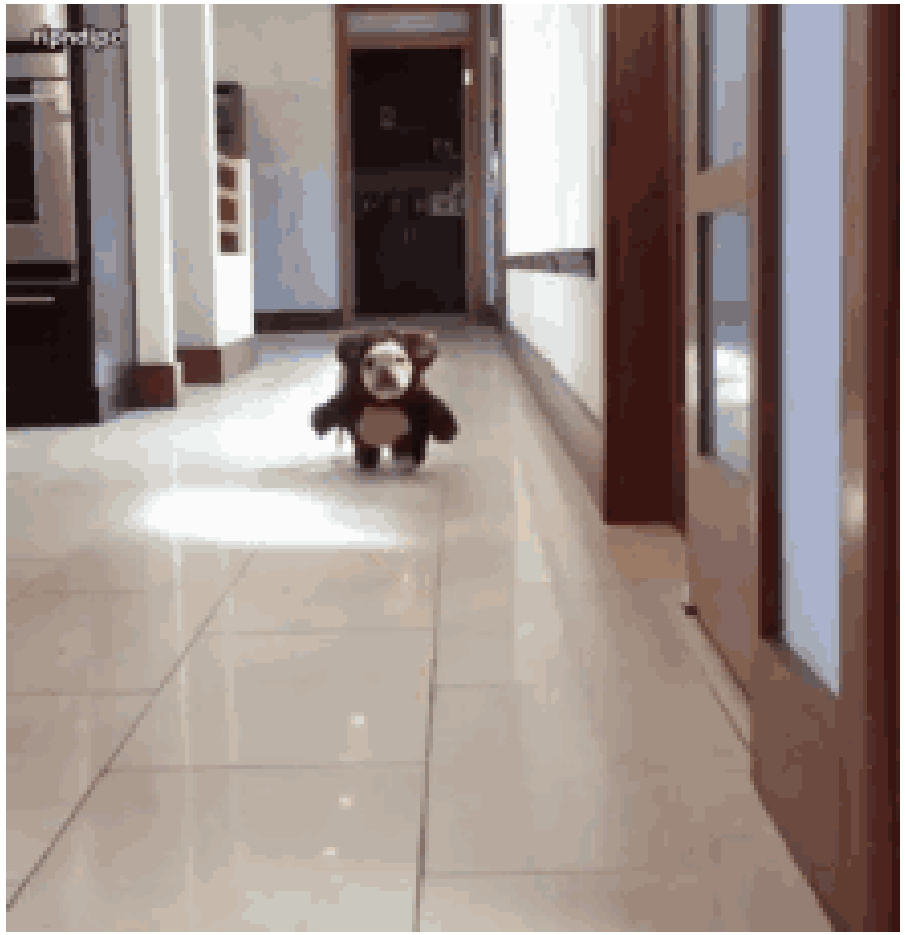


lib/lexer_parser.ex - evaluate_tree/1



lib/lexer_parser.ex - apply_logic/1

Run it!



```
iex(1)> LexerParser.evaluate('true')
{:ok, true}

iex(2)> LexerParser.evaluate('false')
{:ok, false}

iex(3)> LexerParser.evaluate('true and false')
{:ok, false}

iex(4)> LexerParser.evaluate('true and true and true')
{:ok, true}

iex(5)> LexerParser.evaluate('and')
{:error, {1, :business_parser, ['syntax error before: ', ['\'and\'']]}}

iex(6)> LexerParser.evaluate('doesnt exist')
{:error, {1, :business_lexer, {:illegal, 'd'}}, 1}
```

lib/lexer_parser.ex - **Run it!**


Global

2.

Variables

(THE SPICE OF LIFE)

```
iex(1)> LexerParser.evaluate('a and b', %{"a" => true, "b" => true})
{:ok, true}
```

business_lexer.xrl

```
1   Definitions.
2
3   VAR   = ([A-Za-z_][0-9a-zA-Z_]*)
4   WS    = ([\000-\s]|%.*)
5
6   Rules.
7
8   false   : {token, {false,   TokenLine}}.
9   true    : {token, {true,    TokenLine}}.
10
11  and     : {token, {and_op,  TokenLine, list_to_atom(TokenChars)}}.
12
13  {VAR}   : {token, {var,     TokenLine, list_to_binary(TokenChars)}}.
14  {WS}+   : skip_token.
15
```

src/business_lexer.xrl

business_parser.yrl

```
1   Nonterminals expression bool.
2
3   Terminals var and_op true false.
4
5   Rootsymbol expression.
6
7   expression -> bool : '$1'.
8   expression -> var : extract('$1').
9
10  expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
11
12  bool -> true : true.
13  bool -> false : false.
14
15  Erlang code.
16
17  extract({T,_,V}) -> {T, V}.
18
```

src/business_parser.yrl

lexer_parser.ex

```elixir
defmodule LexerParser do
  def evaluate(expression, variables \\ %{}) do
    with {:ok, tokens, _} <- :business_lexer.string(expression),
         {:ok, tree} <- :business_parser.parse(tokens) do
      evaluate_tree(tree, variables)
    end
  end

  # Tree functions
  # ==============

  def evaluate_tree({:binary_expr, op, a, b}, variables) do
    with {:ok, a} <- evaluate_tree(a, variables),
         {:ok, b} <- evaluate_tree(b, variables) do
      apply_logic({:binary_expr, op, a, b}, variables)
    end
  end

  def evaluate_tree(other, variables) do
    apply_logic(other, variables)
  end

  # Logic functions
  # ===============

  def apply_logic(boolean, _variables) when boolean in [true, false], do: {:ok, boolean}

  def apply_logic({:binary_expr, :and_op, a, b}, _variables)
    when is_boolean(a) and is_boolean(b), do: {:ok, a and b}

  def apply_logic({:var, variable}, variables) do
    case Map.get(variables, variable, nil) do
      nil -> {:error, "variable \"#{variable}\" not provided in: #{inspect variables}"}
      value -> {:ok, value}
    end
  end

end
```

lib/lexer_parser.ex - Pass in variables

lexer_parser.ex

```elixir
defmodule LexerParser do
  def evaluate(expression, variables \\ %{}) do
    with {:ok, tokens, _} <- :business_lexer.string(expression),
         {:ok, tree} <- :business_parser.parse(tokens) do
      evaluate_tree(tree, variables)
    end
  end

  # Tree functions
  # ==============

  def evaluate_tree({:binary_expr, op, a, b}, variables) do
    with {:ok, a} <- evaluate_tree(a, variables),
         {:ok, b} <- evaluate_tree(b, variables) do
      apply_logic({:binary_expr, op, a, b}, variables)
    end
  end

  def evaluate_tree(other, variables) do
    apply_logic(other, variables)
  end

  # Logic functions
  # ===============

  def apply_logic(boolean, _variables) when boolean in [true, false], do: {:ok, boolean}

  def apply_logic({:binary_expr, :and_op, a, b}, _variables)
    when is_boolean(a) and is_boolean(b), do: {:ok, a and b}

  def apply_logic({:var, variable}, variables) do
    case Map.get(variables, variable, nil) do
      nil -> {:error, "variable \"#{variable}\" not provided in: #{inspect variables}"}
      value -> {:ok, value}
    end
  end

end
```

lib/lexer_parser.ex - Apply variables

Run it!

```
iex(1)> LexerParser.evaluate('a and b', %{"a" => true, "b" => true})
{:ok, true}

iex(2)> LexerParser.evaluate('a and b', %{"a" => true, "b" => false})
{:ok, false}
```

lib/lexer_parser.ex - **Run it!**

3

NOT!

```
iex(1)> LexerParser.evaluate('not false')
{:ok, true}
```

business_lexer.xrl

```
1   Definitions.
2
3   VAR    = ([A-Za-z_][0-9a-zA-Z_]*)
4   WS     = ([\000-\s]|%.*)
5
6   Rules.
7
8   false    : {token, {false,    TokenLine}}.
9   true     : {token, {true,     TokenLine}}.
10
11  and      : {token, {and_op,   TokenLine, list_to_atom(TokenChars)}}.
12  not      : {token, {not_op,   TokenLine, list_to_atom(TokenChars)}}.
13
14  {VAR}    : {token, {var,      TokenLine, list_to_binary(TokenChars)}}.
15  {WS}+    : skip_token.
16
```

src/business_lexer.xrl - **Add 1 line**

business_parser.yrl

```
1   Nonterminals expression bool.
2
3   Terminals var and_op not_op true false.
4
5   Rootsymbol expression.
6
7   expression -> bool : '$1'.
8   expression -> var : extract('$1').
9
10  expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
11  expression -> not_op expression : {unary_expr, not_op, '$2'}.
12
13
14  bool -> true : true.
15  bool -> false : false.
16
17  Erlang code.
18
19  extract({T,_,V}) -> {T, V}.
```

src/business_parser.yrl - **Add 1 line**

```elixir
def evaluate_tree({:unary_expr, op, a}, variables) do
  with {:ok, a} <- evaluate_tree(a, variables) do
    apply_logic({:unary_expr, op, a}, variables)
  end
end
```

```elixir
def apply_logic({:unary_expr, :not_op, a}, _)
  when is_boolean(a), do: {:ok, !a}
```

lib/lexer_parser.yrl - **Add 2 functions**

```
iex(1)> LexerParser.evaluate('true and not false and a', %{"a" => true})
{:ok, true}
```

lib/lexer_parser.yrl - **Run it!**

New super power unlocked

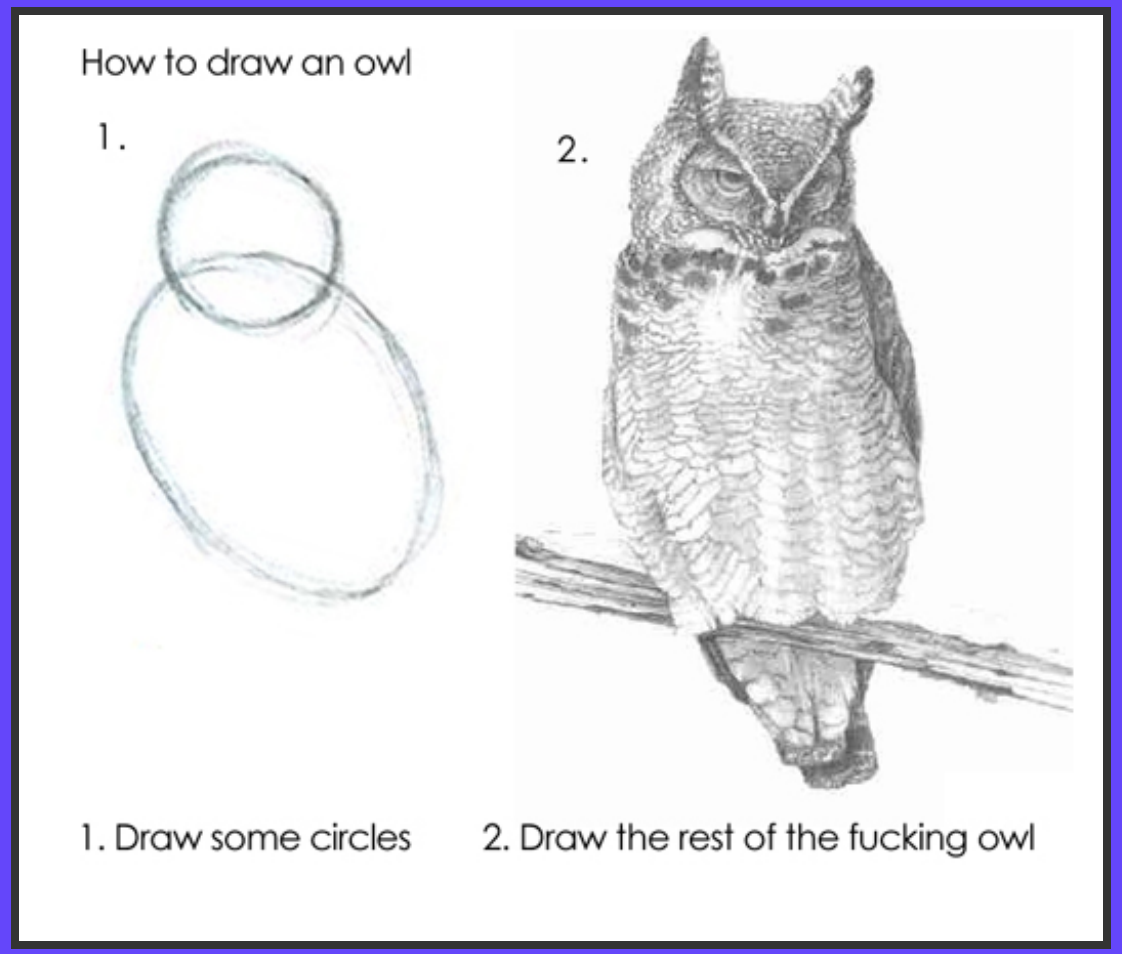
I said we're headed here....

```
eval(
  "not email_disabled and ((pet_age > 4.1 and clinic_id = 123) or (appointment_type in (1,2,3)))",
  %{"email_disabled" => false, "pet_age" => 4.2, "clinic_id" => 124, "appointment_type" => 1}
)

{:ok, true}
```

but not in the way you expected....

4.

The Rest of the fucking Owl



xpreso_lexer.xrl

```
Definitions.

VAR   = ([A-Za-z_][0-9a-zA-Z_]*)
INT   = [-+]?[0-9]+
FLOAT = [-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?
COMP  = (<|<=|>=|>)
EQ    = (=|!=)
ADD   = (\+|\-)
MULT  = (\*|/|mod)
STR   = '[^\'\n]*'
WS    = ([\000-\s]|%.*)

Rules.

false   : {token, {false,   TokenLine}}.
true    : {token, {true,    TokenLine}}.
in      : {token, {in_op,   TokenLine, list_to_atom(TokenChars)}}.
or      : {token, {or_op,   TokenLine, list_to_atom(TokenChars)}}.
and     : {token, {and_op,  TokenLine, list_to_atom(TokenChars)}}.
not     : {token, {not_op,  TokenLine, list_to_atom(TokenChars)}}.
{EQ}    : {token, {eq_op,   TokenLine, list_to_atom(TokenChars)}}.
{COMP}  : {token, {comp_op, TokenLine, list_to_atom(TokenChars)}}.
{ADD}   : {token, {add_op,  TokenLine, list_to_atom(TokenChars)}}.
{MULT}  : {token, {mult_op, TokenLine, list_to_atom(TokenChars)}}.
{VAR}   : {token, {var,     TokenLine, list_to_binary(TokenChars)}}.
{STR}   : {token, {string,  TokenLine, list_to_binary(strip(TokenChars, TokenLen))}}.
{INT}   : {token, {number,  TokenLine, list_to_integer(TokenChars)}}.
{FLOAT} : {token, {number,  TokenLine, list_to_float(TokenChars)}}.
[(),]   : {token, {list_to_atom(TokenChars), TokenLine}}.
{WS}+   : skip_token.

Erlang code.

strip(TokenChars,TokenLen) ->
    lists:sublist(TokenChars, 2, TokenLen - 2).
```

expreso_parser.yrl

```
1   Nonterminals expression predicate scalar_exp elements element bool.
2
3   Terminals atom var number string mult_op add_op and_op or_op in_op not_op eq_op comp_op '(' ')' ',' true false
4
5   Rootsymbol expression.
6   Left 100 or_op.
7   Left 200 and_op.
8   Left 300 eq_op comp_op.
9   Left 400 in_op.
10  Left 500 add_op.
11  Left 600 mult_op.
12  Nonassoc 700 not_op.
13
14  expression -> bool : '$1'.
15  expression -> predicate : '$1'.
16  expression -> var : extract('$1').
17  expression -> expression or_op expression   : {binary_expr, or_op, '$1', '$3'}.
18  expression -> expression and_op expression  : {binary_expr, and_op, '$1', '$3'}.
19  expression -> not_op expression : {unary_expr, not_op, '$2'}.
20  expression -> '(' expression ')' : '$2'.
21
22  predicate -> bool eq_op bool : {binary_expr, extract('$2'), '$1', '$3'}.
23  predicate -> scalar_exp eq_op scalar_exp : {binary_expr, extract('$2'), '$1', '$3'}.
24  predicate -> scalar_exp comp_op scalar_exp : {binary_expr, extract('$2'), '$1', '$3'}.
25  predicate -> scalar_exp in_op '(' elements ')' : {binary_expr, in_op, '$1', '$4'}.
26  predicate -> scalar_exp not_op in_op '(' elements ')' : {binary_expr, not_in_op, '$1', '$5'}.
27
28  predicate -> scalar_exp in_op var : {binary_expr, in_op, '$1', extract('$3')}.
29  predicate -> scalar_exp not_op in_op var : {binary_expr, not_in_op, '$1', extract('$4')}.
30
31  scalar_exp -> scalar_exp add_op scalar_exp : {binary_expr, extract('$2'), '$1', '$3'}.
32  scalar_exp -> scalar_exp mult_op scalar_exp: {binary_expr, extract('$2'), '$1', '$3'}.
33  scalar_exp -> element : '$1'.
34
35  elements -> element : [extract_value('$1')].
36  elements -> element ',' elements : [extract_value('$1')|'$3'].
37
38  element -> atom : '$1'.
39  element -> var : extract('$1').
40  element -> string : extract('$1').
41  element -> number : extract('$1').
42
43  bool -> true : true.
44  bool -> false : false.
45
46  Erlang code.
47
48  extract_value({_,V}) -> V.
49  extract({T,_,V}) -> {T, V}.
50
```

Expreso

A Boolean template library for elixir.

- true, false
- not, in, and, or
- <, <=, >, >=, =
- +, -, /, *

https://github.com/ympons/expreso

Yaismel Pons

```
# Evaluate if a + 2 is between 3 and 1
iex> Expreso.eval("a + 2 in (3, 1)", %{"a" => 1})
iex> {:ok, true}

# Check if the key "test" is "works"
iex> Expreso.eval("test = 'works'", %{"test" => "works"})
iex> {:ok, true}

# Check if the key "test" is "works" or "great"
iex> Expreso.eval("test = 'works' or test = 'great'", %{"test" => "great"})
iex> {:ok, true}

# Evaluate an expression with not_in_op and sum
iex> Expreso.eval("10 + 2 not in (3, 1)")
iex> {:ok, true}

# Evaluate an expression with and_logic
iex> Expreso.eval("a + 2 in (2, 3) and 1 + 1 >= 2", %{"a" => 1})
iex> {:ok, true}

# Evaluate an expression with string
iex> Expreso.eval("a = 'Hello' and b != 'World' and 1 + 1 = 2", %{"a" => "Hello",
"b" => ""})
iex> {:ok, true}

# Evaluate an expression with an array variable
iex> Expreso.eval("a in b", %{"a" => 1, "b" => [2, 1]})
iex> {:ok, true}

# Evaluate another expression with an array variable
iex> Expreso.eval("a not in b", %{"a" => 1, "b" => [2, 1]})
iex> {:ok, false}

# Evaluate an expression using not operator
iex> Expreso.eval("not 1 > 1 + a", %{"a" => 2})
```

Thanks for listening!

- https://github.com/bechurch/lexer_parser
- @bnchrch
- https//ben.church