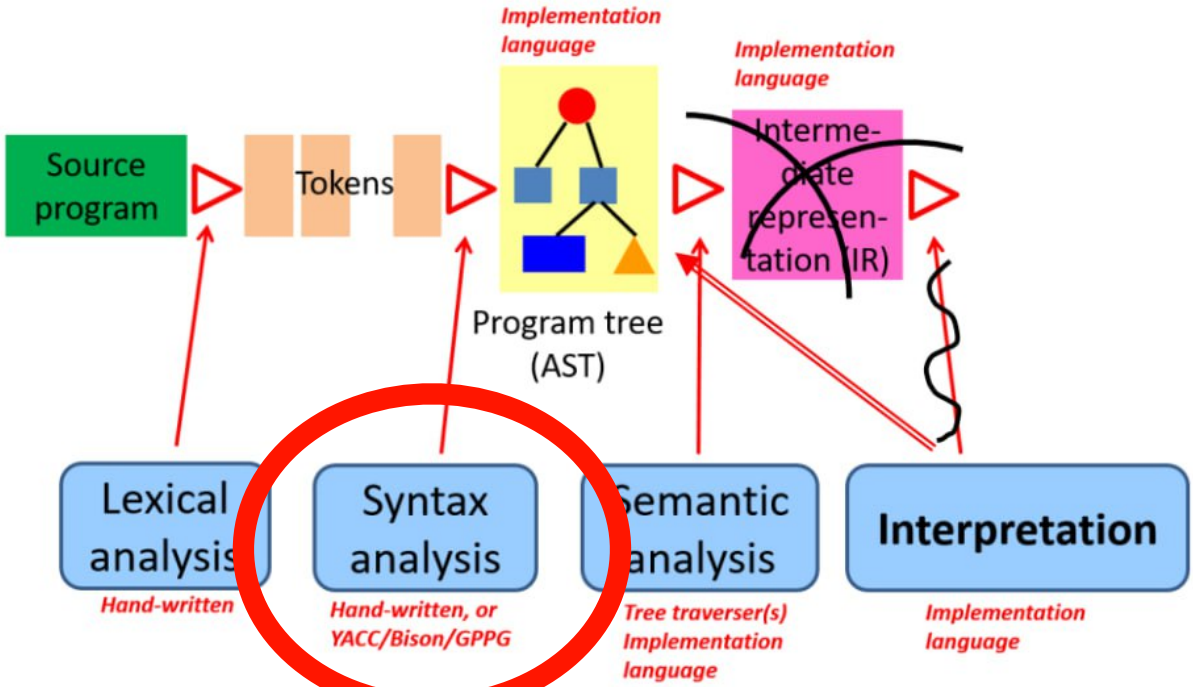
Pechenen team

Parser progress

Project F: Functional



Code example of pechenen LISP

GCD

```
1 (func Gcd (a b) (
2   cond (equal a b)
3     (return a) (
4     cond (greater a b)
5       (return (Gcd b (minus a b)))
6       (return (Gcd a (minus b a)))
7     )
8   )
9 )
```

Parser output Gcd

```
func Gcd(a b)(
  cond (equal a b)
    (return a) (
…
```

```
{%{column: 2, line: 1, value: :func},
   [
     %{column: 7, line: 1, value: :Gcd},
     {%{column: 12, line: 1, value: :a}, [%{column: 14, line: 1, value: :b}]},
     {%{column: 2, line: 2, value: :cond},
      [
        {%{column: 8, line: 2, value: :equal},
         [%{column: 14, line: 2, value: :a}, %{column: 16, line: 2, value: :b}]},
        {%{column: 5, line: 3, value: :return},
         [%{column: 12, line: 3, value: :a}]},
…
```

Parser output Gcd

```
cond (greater a b)
  (Return (gcd b (minus a b)))
…
```

```
{%{column: 4, line: 4, value: :cond},
        [
         {%{column: 10, line: 4, value: :greater},
          [
            %{column: 18, line: 4, value: :a},
            %{column: 20, line: 4, value: :b}
          ]},
         {%{column: 7, line: 5, value: :return},
          [
            {%{column: 15, line: 5, value: :Gcd},
             [
               %{column: 19, line: 5, value: :b},
               {%{column: 22, line: 5, value: :minus},
                [
                  %{column: 28, line: 5, value: :a},
                  %{column: 30, line: 5, value: :b}
                ]}
             ]}
          ]},
…
```

Parser output Gcd

```
(Return (gcd a (minus b a)))
```

```
{%{column: 7, line: 6, value: :return},
          [
            {%{column: 15, line: 6, value: :Gcd},
             [
               %{column: 19, line: 6, value: :a},
               {%{column: 22, line: 6, value: :minus},
                [
                  %{column: 28, line: 6, value: :b},
                  %{column: 30, line: 6, value: :a}
                ]}
             ]}
          ]}
```

Another code example of pechenen LISP

GCD

```
(func Fibonacci (n) (
  cond (less n 2)
    (return n)
    (return (plus (Fibonacci (minus n 1)) (Fibonacci (minus n 2)))))
)
```

Parser output Fibonacci

```
(Func fibonacci (n) (
  cond (less n 2)
    (return n)
```

```
{%{column: 2, line: 1, value: :func},
 [
   %{column: 7, line: 1, value: :Fibonacci},
   {%{column: 18, line: 1, value: :n}, []},
   {%{column: 2, line: 2, value: :cond},
    [
      {%{column: 8, line: 2, value: :less},
       [%{column: 13, line: 2, value: :n}, %{column: 15, line: 2, value: 2}]},
      {%{column: 5, line: 3, value: :return},
       [%{column: 12, line: 3, value: :n}]},
```

Parser output Fibonacci

```
(return (plus (Fibonacci (minus n 1) )
(Fibonacci (minus n 2)))))
```

```
{%{column: 5, line: 4, value: :return},
        [
          {%{column: 13, line: 4, value: :plus},
           [
             {%{column: 19, line: 4, value: :Fibonacci},
              [
                {%{column: 30, line: 4, value: :minus},
                 [
                   %{column: 36, line: 4, value: :n},
                   %{column: 38, line: 4, value: 1}
                 ]}
              ]},
             {%{column: 43, line: 4, value: :Fibonacci},
              [
                {%{column: 54, line: 4, value: :minus},
                 [
                   %{column: 60, line: 4, value: :n},
                   %{column: 62, line: 4, value: 2}
                 ]}
              ]}
           ]}
        ]}
```

Node structure

Title

```elixir
defmodule Parser.Node do
  alias Parser.NodeValue
  @type t :: {NodeValue.t(), [NodeValue.t() | Node.t()]}
end
defmodule Parser.NodeValue do
  @type t :: %__MODULE__{
          value: String.t() | boolean() | integer() | float() | atom() | list(),
          line: non_neg_integer(),
          column: non_neg_integer()
        }

  @enforce_keys [:value, :column, :line]
  defstruct [:value, :column, :line]
end
```

Parser implementation

Title

```elixir
defmodule Parser do
  alias Parser.Node

  @spec parse(binary) :: {:error, any} | {:ok, Node.t()}
  def parse(string) do
    {:ok, tokens} = Lexer.scan(string)

    :parser.parse(Enum.map(tokens, fn token -> {token.type, token} end))
  end
end
```

Parser implementation

parser.yrl

```
Nonterminals
   list elements element
    .

Terminals
    atom liter '(' ')' '\''
    .

Rootsymbol elements.

list -> '(' ')' : make_empty_list('$1').
list -> '(' elements ')' : make_list_node('$2').
list -> '\'' element : make_list_node([quote | ['$2']]).

elements ->
  element : ['$1'].

elements ->
  element elements : ['$1' | '$2'].

element -> liter : extract_token('$1').
element -> atom : extract_token('$1').
element -> list : '$1'.

Erlang code.

extract_token({_Type, #{value := Value, line := Line, column := Column}}) -> #{value => Value, line => Line, column => Column}.
make_empty_list({_Type, #{line := Line, column := Column}}) -> #{value => [], line => Line, column => Column}.

make_list_node([Element | T]) -> {Element, T};
make_list_node(#{value := [], line := Line, column := Column}) -> return_error([{line, Line}, {column, Column}], "Unexpected ().
If you want to define an empty list use `quote` function").

```

Team Contribution

Nikita Pozdniakov

Parser module wrap implementation

Maxim Filonov

Integration of written Lexer <-> Parser

Pavel Bakharuev

Token extraction, testing

Andrey Sandimirov

Paser node structure implementation