Pechenen team

Final presentation

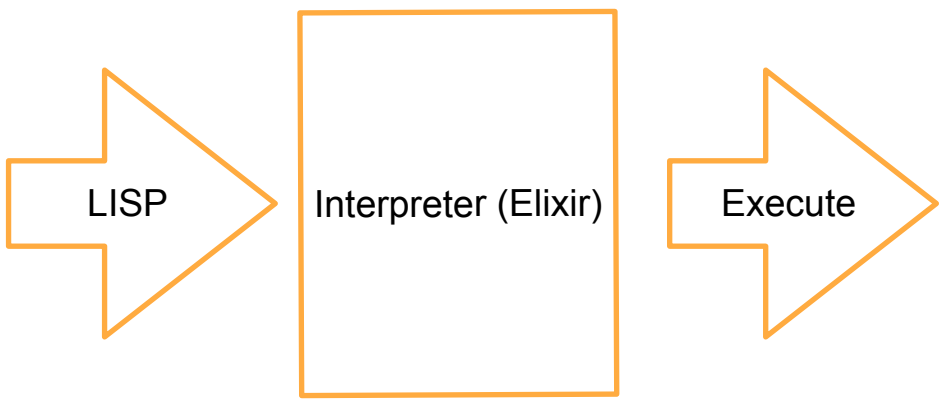
Project: People

Task Description

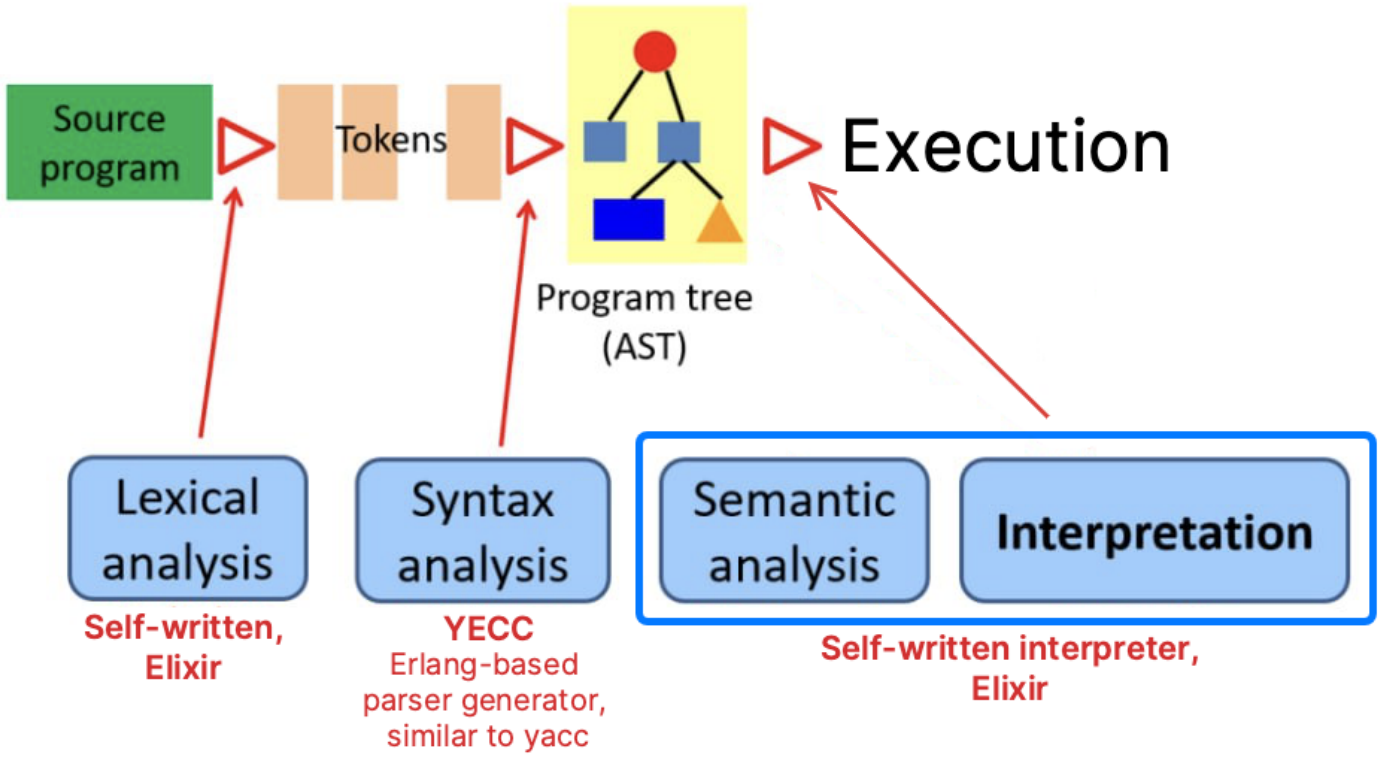
Implement a interpreter for LISP programming language using Elixir programming language. The interpreter should be able to construe LISP source code to execute.

Supported features:

1. Data types
2. Arithmetic operations
3. Functions
4. Control flow statements
5. Variable assignments
6. Recursion



Compiler Architecture



Lexical Analyzer

Data Structures

TOKEN:
- Value
- Type (operator/atom)
- Column
- Line

Fib.out

```
{:ok,
  [
    %Lexer.Token{value: "(", type: :operator, column: 1, line: 1},
    %Lexer.Token{value: :func, type: :atom, column: 2, line: 1},
    %Lexer.Token{value: :Fibonacci, type: :atom, column: 7, line: 1},
    %Lexer.Token{value: "(", type: :operator, column: 17, line: 1},
    %Lexer.Token{value: :n, type: :atom, column: 18, line: 1},
    %Lexer.Token{value: ")", type: :operator, column: 19, line: 1},
    %Lexer.Token{value: "(", type: :operator, column: 21, line: 1},
    %Lexer.Token{value: :cond, type: :atom, column: 2, line: 2},
    %Lexer.Token{value: "(", type: :operator, column: 7, line: 2},
    %Lexer.Token{value: :less, type: :atom, column: 8, line: 2},
    %Lexer.Token{value: :n, type: :atom, column: 13, line: 2},
    ...
  ]
}
```

Token Example

Lexical Analyzer

Technologies

Hand-written

token.ex

```elixir
defmodule Lexer.Token do
  @types ~w(atom liter operator)a

  @type t :: %__MODULE__{
          value: String.t() | boolean() | integer() | float() | atom(),
          type: type()
        }

  @typep type ::
           unquote(
             @types
             |> Enum.map(&inspect/1)
             |> Enum.join(" | ")
             |> Code.string_to_quoted!()
           )
  @enforce_keys [:value, :type, :column, :line]
  defstruct [:value, :type, :column, :line]
end
```

Token implementation (key code)

Syntax Analyzer (Parser)

Data Structures

Node structure:

- Column
- Line
- Value

```
(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}]},
```

Node Example

Syntax Analyzer (Parser)

Implementation

Erlang based parser
generator, similar to YACC

Key code

Interpreter

Technologies

Hand-written

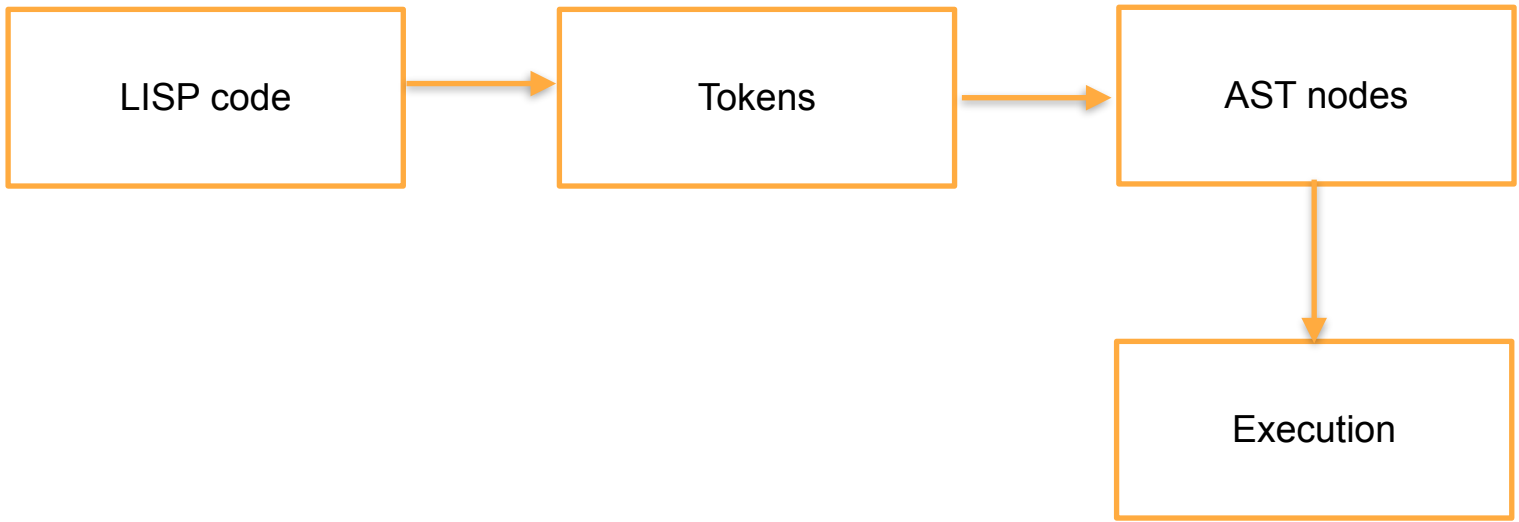
```elixir
@spec interpret([Node.t()], [String.t() | integer()]) :: any()
def interpret(ast, args) do
  {_value, state} =
    Enum.reduce(ast, {:null, initial_state()}, fn node, {_value, acc} ->
      interpret_node(node, acc)
    end)

  {:ok,
   if prog = state.service[:prog] do
     with {func_args, function} <- prog,
          {:number_of_args, true} <- {:number_of_args, length(args) == length(func_args)} do
       args
       |> Enum.zip(func_args)
       |> Enum.reduce(state, fn {val, arg_name}, state ->
         put_in(state, [:scope, arg_name], val)
       end)
       |> put_in([:service, :line], -1)
       |> put_in([:service, :column], -1)
       |> function.()
     else
       _ ->
         raise "prog definition error"
     end
   end}
end
```

Standard library

Written in lisp

len/1	len(list) -> list
reverse/1	reverse(list) -> list
push/2	push(item, list) -> list // appends an element
repeat/2	repeat(item, times) -> list // makes a list
get/2	get(index, list) -> item // get by index
map/2	map(func, list) -> list
filter/2	filter(predicate, list) -> list
reduce/3	reduce(func, init, list) -> value
zip/2	zip(list, list) -> list
unzip/1	unzip(list) -> (list, list)
apply/3	apply(func, times, init) -> value // applies a function to itself

Datastructures (recap)



Results

We have implemented:

- ☑ Lexical analyzer
- ☑ Parser
- ☑ Interpreter

To support LISP:

- ☑ Arithmetic operations
- ☑ Functions
- ☑ Control flow statements
- ☑ Variable assignments
- ☑ Recursion
- ☑ STD lib

Team Contribution

Nikita Pozdniakov

Lexer: code extraction
Parser: module wrap imp.
Interpreter: base, loops

Maxim Filonov

Lexer: constructions (code)
Parser: Integration Lexer <-> Parser
Interpreter: arithmetics

Pavel Bakharuev

Lexer: token design, presentation
Parser: token extraction imp., presentation
Interpreter: lists imp., testing, presentation

Andrey Sandimirov

Lexer: research & testing
Parser: node structure impl.
Interpreter: code STD lib, testing & fixes