SER502-Spring2023-Team15
PDF DOCUMENT

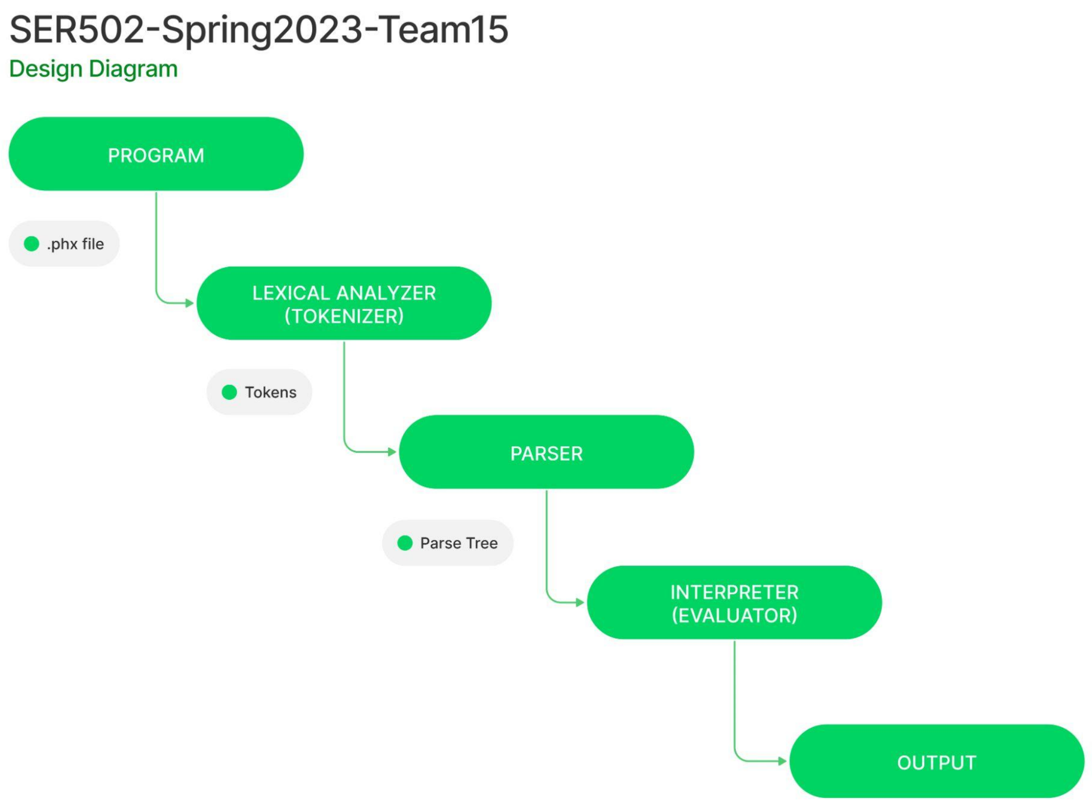
Milestone-1

Team Members:
Janki Padiya
Omkar Pisal
Heet Punjawat
Shivanjay Wagh
Manan Kohli

Github Repository:
https://github.com/heetpunjawat/SER502-Spring2023-Team15

About Language
Name: Phoenix
Extension: .phx

Design:



Program
A program written in Phoenix (.phx) language will be processed by computer to produce the desired output. The processing of a program involves several stages, including lexical analysis, parsing, interpretation or evaluation, and output generation.

Lexical Analyzer (Tokenizer)
Lexical Analyzer will accept the source code (.phx) file as an input and produce a list of tokens. We will use Python for lexical analysis. Each token represents a specific type of lexical element such as keywords, identifiers, operators, constants, and punctuation marks. A stream of tokens will be produced by lexical analyzer that can be used as input for the next stage of the compilation process, which is parsing.

Parser
The parser checks if syntax and format of the written program is correct. It makes sure that the program follows the rules of the programming language. It generates the parse tree using the tokens provided by the lexical analyzer. A parser builds a parse tree using a set of rules called a grammar. A parser uses grammar to check whether a program conforms to the syntax of the language.

Interpreter (Evaluator)
The evaluation process will start as soon as the evaluator approves and uses the parse tree that the parser produced. The evaluator uses the parse tree essentially as a blueprint to carry out the relevant set of instructions. To put it another way, the evaluator is in charge of analyzing the parse tree and carrying out the required calculations according to its layout. It is a crucial element of the compiler or interpreter that helps with the transformation of code from human-readable to machine-executable. The parse tree, which acts as a link between the parser and the execution stage, is a crucial element that directs the evaluator's decision-making process.

Output
The final output will be generated based on the all steps and intermediate outputs generated from all the previous steps which have been passed to later stages in order to generate end results.

Parsing Technique
We are using Recursive Descent Parsing which is one of the most used top down parsing techniques. The right hand side of the grammar rules are used to create a group of mutually recursive procedures based on all the non-terminals. The parser's rules are expressed in DCG.

Data Structures
The Python lexer will use a list as a data structure. It will save this as a token file. This token file can then be read by the Prolog parser and converted into another list. The parser will generate a parse tree list as output, which will be accepted by the evaluator component and executed directly on the machine. All variables used during program execution will be stored as a Prolog list.

Programming Languages Used

Component	Language used
Lexer	Python
Parser	Prolog
Evaluator	Prolog

Data Types:

- Phoenix supports integer (int), float, string (string), boolean(bool) data types.

  Integer:
  int k;
  K = 10;

  String:
  String s;
  S = “hello”;

  Boolean:
  bool k;
  k = True;

Operators:

- Assignment(=), addition(+), subtraction(-), multiplication(*) and division(/) operators are supported by Phoenix.
- Phoenix supports Boolean identifiers such as and, or, not as well as the ternary operator ‘?:’
- It supports <, >, <=, >=, !=, == comparison operators too and increment (++), decrement(--) operators too.

Control Structures:

- Control structure **if-then-else** is supported by Phoenix
- Phoenix supports other additional control structures named for loop, while loop and a for in range(number1, number2) loop

Miscellaneous:

- It has a print statement to print the output on screen. It can print string, int etc on screen.
- expects a semicolon at the end of declaration
- Identifier should be an alphabetic word starting with a small letter.
- String in Phoenix should be in “ ” double quoted.

Grammar:

% TERMINALS %
----

bool_val --> ['True'].
bool_val --> ['False'].

var_type --> ['int'] | ['float'] | ['bool'] | ['string'].

and_operator --> ['and'].
or_operator --> ['or'].
not_operator --> ['not'].

assignment_operator --> [‘=’].
end_of_statement --> [‘;’].

inc_operator --> ['++'].
dec_operator --> ['--'].

comp_operators --> ['<'], ['>'], ['<='], ['>='], ['=='], ['!='].

ternary_operator --> ['?']

lower_case --> ['a'] | ['b'] | ['c'] | ['d'] | ['e'] | ['f'] | ['g'] | ['h'] | ['i'] | ['j'] | ['k'] | ['l'] | ['m'] | ['n'] | ['o'] | ['p'] | ['q'] | ['r'] | ['s'] | ['t'] | ['u'] | ['v'] | ['w'] | ['x'] | ['y'] | ['z'].

```
upper_case --> ['A'] | ['B'] | ['C'] | ['D'] | ['E'] | ['F'] | ['G'] | ['H'] | ['I'] | ['J'] | ['K'] | ['L'] | ['M'] |
['N'] | ['O'] | ['P'] | ['Q'] | ['R'] | ['S'] | ['T'] | ['U'] | ['V'] | ['W'] | ['X'] | ['Y'] | ['Z'].

symbol --> [' '] | ['~'] | ['!'] | ['@'] | ['#'] | ['$'] | ['%'] | ['^'] | ['&'] | ['+'] | ['-'] | ['*'] | ['/'] | [','] | ['.'] |
[':'] | [';'] | ['<'] | ['='] | ['>'] | ['?'] | ['\\'] | ['\"'] | ['_'] | ['`'] | ['('] | [')'] | ['['] | [']'] | ['{'] | ['}'] | ['|'].

digit --> ['0'] | ['1'] | ['2'] | ['3'] | ['4'] | ['5'] | ['6'] | ['7'] | ['8'] | ['9'].

single_quote --> ['\''].
double_quote --> ['\"'].

---------------------
/* NON-TERMINALS */
---------------------

program --> statement_list.

block --> ['{'], statement_list, ['}'].

statement_list --> statement.
statement_list --> statement, statement_list.
statement_list --> statement_without_block.
statement_list --> statement_without_block, statement_list.


% statements without block (simple statements)
statement_without_block --> print_statement.
statement_without_block --> assign_statement.
statement_without_block --> var_decl_statement.

% Multi Line statements (complex statements)
statement --> while_loop_statement.
statement --> for_loop_statement.
statement --> for_enhanced_statement.
statement --> if_statement.
statement --> if_elif_else_statement.
statement --> if_else_statement.

if_clause --> ['if'], ['('], condition, [')'], block.
else_clause --> ['else'], block.
```

```
elif_clause --> ['elif'], ['('], condition, [')'], block.
elif_clause --> ['elif'], ['('], condition, [')'], block, elif_clause.

if_statement --> if_clause.
if_elif_else_statement --> if_clause, elif_clause, else_clause.
if_else_statement --> if_clause, else_clause.

while_loop_statement --> ['while'], ['('], condition, [')'], block.

for_enhanced_statement --> ['for'], var_name, ['in'], ['range'], ['('], range_val, [';'],
range_val, [')'], block.

range_val --> var_name | integer.

for_loop_statement --> ['for'], ['('], assign_statement, [';'], condition, [';'],
var_change_part, [')'], block.

var_change_part --> inc_expression.
var_change_part --> dec_expression.
var_change_part --> var_name, assignment_operator, expression.

condition --> expression, comp_operators, expression.

inc_expression --> var_name, inc_operator.
inc_expression --> inc_operator, var_name.
dec_expression --> var_name, dec_operator.
dec_expression --> dec_operator, var_name.

print_statement --> [print_str], ['('], string_val, [')'], end_of_statement.
print_statement --> [print_str], ['('], var_name, [')'], end_of_statement.
print_statement --> [print_expr], ['('], expression, [')'], end_of_statement.

expression --> value.
expression --> value, operator, expression.
expression --> ternary_expression.
expression --> ['('], expression, [')'], operator, expression.

ternary_expression --> ['('], condition, [')'], ['?'], expression, [':'], expression.

value --> float | integer | bool_val | string_val | var_name.
```

```
boolean_operators --> and_operator | or_operator | not_operator.

operators --> ['+'] | ['-'] | ['*'] | ['/'] | boolean_operators.

assign_statement --> var_name, assignment_operator, expression, end_of_statement.

var_decl_statement --> var_type, var_name, end_of_statement.
var_decl_statement --> var_type, var_name, assignment_operator, expression,
end_of_statement.

var_name --> lower_case, var_name.
var_name --> var_name, upper_case.
var_name --> var_name, upper_case, var_name.
var_name --> var_name, ['_'], var_name.
var_name --> lower_case.

string_val --> single_quote, char_phrase, single_quote.
string_val --> double_quote, char_phrase, double_quote.

char_phrase --> char, char_phrase.
char_phrase --> char.

char --> lower_case | upper_case | digit | symbol.

float --> integer, ['.'], integer.
float --> integer.

integer --> digit, integer.
integer --> digit.
```