SER 502 Spring'20

Team 20 - LitePiler

Abhishek Haksar
Rohit Kumar Singh
Sarvansh Prasher
Surya Chatterjee

Introduction

- As the name suggests, we call our compiler “LitePiler” because it intend to make life easier for developer.

- To develop this compiler we have made use of Prolog.

- This language is inspired by C and C++ and is statically typed.

Structure & Features

- LitePiler expects code to be written in a certain format.
  - Each code block should begin with **[enter]** keyword and end with and **[exit]** keyword.

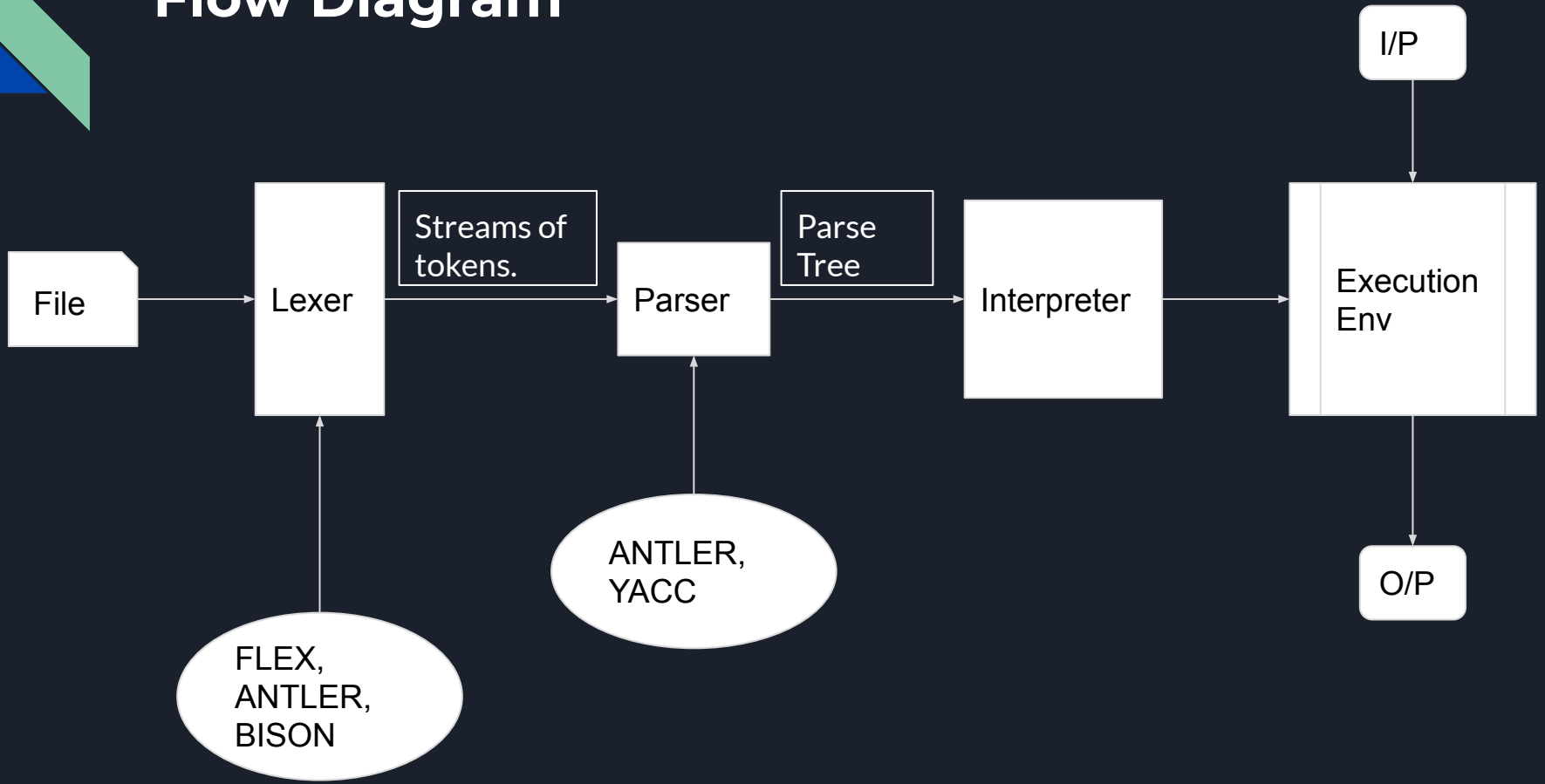
- The code must have a Declaration block where the user must declare all the variables that will be used in the code.

- Once all the variables have been declared the user can start the operation block.
  - The user can evaluate expressions using PEMDAS.
  - The user can use conditional checks using IF ELSE.
  - The user can perform loop statements using WHEN and WHILE statements.

Data types, Operators & Keywords

1. DataTypes-
   a. **int-** To declare Integer variables.
   b. **bool-** To declare boolean variables.
   c. **string-** To declare string variables.
2. Operators-
   a. **Arithmetic Operators-** +, -, *, /
   b. Logical Operators-
      i. The compiler supports **not equals operator**, **equals operator.**
      ii. It also supports **greater than, less than greater or equals and less or equals** operator.
3. Conditional Keywords- IF-ELSE
4. Iterative Keywords- WHILE, DO-WHILE and WHEN.
5. Ternary Operator

Flow Diagram



Lexer and Parser

- Parser & Lexer are written in Prolog programming language.
- For Parser of language, the rules are written in DCG(Definite Clause Grammar)
- For Lexer layer, we have constructed a table which consists of all the keywords referenced in language. As soon as the program is compiled, it generates the lexemes for programm by removing white space and new lines.
- These tokens are then passed to Parser layer which constructs the parse tree as the middle part of our language.
- We have used the Top-Down parsing technique for the parser in which our parser will construct a syntax tree from the top and then give it to the interpreter for evaluating those tokens

Parse Tree

- This will be the middle layer of our programming language.
- This will be helping us in interpreting whether our tokens have been appropriately assigned to the tree.
- Interpreter of our language will be using this parse tree.
- Format of our middle parse tree file is “**.lpy**” .

Interpreter

- This is written in Prolog programming language.
- We have implemented the reduction machine for inter Here we are going to use environment variables that will keep an account of the key and value pairs in the list.
- This layer is used for evaluating the program and giving us the output.
- Environment is looked up at each step and updated on each block.
- Reduction rules were used to evaluate the block.

```
parse --> program.

% Rule for the main function of language.
program --> structure.

% Rule for structure inside the program
structure -->[enter],declaration,
    operation,[exit].

% Rule for variable types in language.
varType --> [int].
varType --> [bool].
varType --> [string].
```

```
% Rule for declarations inside the structure
declaration --> varType,word,[;].
declaration --> varType,word,[;],declaration.

% Rule for assigning values to variable.
assignValue --> word, [=] ,exp, [;].
assignValue --> word,[is], boolExp, [;].
assignValue --> word,[=], ternary, [;].

% Rule for reading the input from system.
readValue --> [input], word, [;].
```

```
% Rule for the operations done in between structure.
operation --> assignValue, operation.
operation --> routine, operation.
operation --> print, operation.
operation  --> comment,operation.
operation --> readValue, operation.
operation --> assignValue.
operation --> routine.
operation --> print.
operation --> comment.
operation --> readValue.
```

```
% Rule for the routines done in between operations.
routine --> [if], condition, [then], operation, [else], operation, [endif].
routine -->[while],condition,[do],operation,[endwhile].
routine --> [when],['('], condition,[:],assignValue,[')'], [repeat], operation, [endrepeat].
routine-->[when],word,[between],[range],['('],number,[,],number,[')'],[repeat],operation, [endrepeat].

routine --> word,[+],[+],[;].
routine --> word,[-],[-],[;].


% Rule for evaluating ternary expressions.
ternary --> ['('],condition,[')'],[?],number,[:],number.
```

```
% Rule for conditions in routines.
condition --> boolExp, [and], boolExp.
condition --> boolExp, [or], boolExp.
condition --> [not], boolExp.
condition --> boolExp.

% Rule for determining boolean expression.
boolExp --> [true].
boolExp --> [false].
boolExp --> exp,[:=:],exp.
boolExp --> exp, [~=], exp.
boolExp --> exp, [:=:], boolExp.
boolExp --> exp, [~=], boolExp.
boolExp --> exp, [<], exp.
boolExp --> exp, [>], exp.
boolExp --> exp, [<],[=], exp.
boolExp -->  exp, [>],[=], exp.
```

```
exp --> verticalExp, [+], exp.
exp --> verticalExp, [-], exp.
exp --> verticalExp.

verticalExp --> negativeNumber,[/], verticalExp.
verticalExp --> number,[/], verticalExp.
verticalExp --> word,[/], verticalExp.
verticalExp --> negativeNumber,[*], verticalExp.
verticalExp --> number,[*], verticalExp.
verticalExp --> word, [*], verticalExp.
verticalExp --> number.
verticalExp --> negativeNumber.
verticalExp --> word.
```

```
%Rule for Identifier
word --> [Word],{atom}.

%Rules for numbers
negativeNumber --> [-],number.
number --> [NumberNode],{number}.

% Rule for comments inside block.
comment --> [!] , statement, [!].

% Rules for statements inside comment.
statement --> word,statement.
statement --> number,statement.
statement --> word.
statement --> number.
```

```
% Rule for finding length of string
wordLength --> word,[.],[length].

% Rule for string concatenation operation
wordConcat --> word,[.],[join],[.],word.

% Rule for printing values.
print --> [display],exp,[;].

print --> [display], [@],statement,[@],[;],!.
```

Thank you