SER 502 – Project

Milestone – 2

Language Name: !XOBILE LANGUAGE

Design:

We are designing an Imperative programming language. The language includes the following features:

1. Variable Assignment Declaration - Declaring variables and assigning values to them.
2. If statement - If is a conditional statement which is executed when it's boolean expression is true.
3. If – Else statement - The else statement is a conditional statement which will execute when the boolean expression of the If statement is false.
4. if – Elif – Else statement – Elif statement is a conditional statement which will execute only if the boolean expression of the if statement is false and elif boolean expression is true, otherwise else statement will execute.
5. While statement - The group of statements inside the while block will be executed continuously until the boolean expression keeps satisfying(returns true/1).
6. Print statement - This statement is used to display any Identifier's value or constants.
7. Primitive types and operators - This includes the arithmetic operations like addition, subtraction, multiplication and division with the primitive data types like Integer, Float, String and Boolean values.

Grammar of the language:

```
/*
Mapping
P --> Program
K --> Block
D --> Declaration
C --> CommandLines (Statements)
Dt --> Data Type
Bl --> Boolean Value
B --> Boolean Expression
E --> Expression
T --> Term
F --> Form     % To assign Identifier or Number to Term
I --> Identifier
N --> Number
```

```
D --> Digit
*/


/* To check the given syntax and generate a parse tree for the same */

P ::=  K

K ::=  begin C end. /*. is used as terminal and will be provided by the user while
programming*/

D ::= boolean I = Bl;
   | Dt I = N;
   | Dt I;
   | I = I;
   | I = N;

Dt ::= int | float | string | boolean
Bl ::= true | false

/* Declaration can be done at any point of time in the program by user */
C ::= D, C;
   | I = E
   | if B { C }
   | if B { C } else { C }
   | if B { C } elif { C } Elif else { C }
   | while B { C }
   | while B { C } else { C }
   | stop
   | K
   | ε

Elif ::= elif { C } Elif | ε

B ::=  true | false | E ~ E | not B | ! B | 0 | 1

E ::=  T + E | T - E | T
T ::= ( E ) T
T ::=  F * T | F / T | F
F ::=  I | N

/* we will try to add power, interger divide and modulous for 2 Expressions in
the program, work in progress*/
```

/*grammar of identifier might be changed in next milestone, to add some constraints in assigning name to the variable*/
I(X) ::= [X]

N ::= D, N | ε

D ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Boolean expression will contain <=,>=,<,> ==, and other comparison expressions in next milestone.

Information about the Interpreter:

We chose to write the interpreter using Prolog. The specific language was chosen because the easiness of the language, as this language is declarative, everything works just by declaring values, which is relatively very easy to learn and adapt.

To handle values and addresses effectively with the usage of different data structure like List, dictionaries (hashTable), etc. We are checking the syntax of the program using prolog, and generating the parse tree, after getting the desired parse tree we are then passing the parse tree as a gesture of approval. Trace predicate is being used to test and check the flow of the syntax tree.

Parsing technique:

After the phase of tokenizing where the source code is converted into a list of tokens by the tokenizer, then the list of tokens is passed on to the phase of Parsing. We plan to parse the list of tokens by writing DCG (Definite Clause Grammar) to generate the parse tree.

After getting the parse tree, Semantic part will come into play. We will be using the same syntax grammar, which we used for generating the parse tree, and modify it to map the commandLine and create semantic meaning of the line and evaluating the commandLine.

Steps for parsing the program and executing the program

1. mainFile.py will import the Tokenization.py file, and pass the file directory along with <filename>.lol (lol: Line on Line).
2. Tokenization.py will generate the tokens and return a list of tokens to mainFile.py, mainFile.py will create a new file, <filename>.tokens.

3. Prolog file will take the .tokens file and assign all the tokens to a variable, then the variable will be passed to the parseTree program.
4. ParseTree will check the syntax and generate the parse tree, only if the syntax of the program is correct, else error will generate. After checking the syntax, tree will be generated and <filename>.tree will be generated, .tree file will be generated just to ensure that syntax is correct and file can be moved to the next stage.
5. After checking if <filename>.tree is generated, semantics of the program will be checked, if the semantic part is done in prolog then it can be reffered from the syntaxTree code and can be used directly, but if done in Python, then many changes will be required.
6. In semantic part, Index table will be generated for each environment, which will help in using the value of the variable, speed up the code.
7. A script will be used where, just by putting the filename with it's location, all the process will automatically done, and we get .tokens,.tree,.exe files in the end which will show the output.

Note: Semantic part will be done either in prolog or in python, (After some discussion with the professor, it will be decided.)