SER502, Spring20, Project Milestone 1 - Team 9

Members:
Abhinaw Sarang
Sagar Khar
Saksham Jhawar
Smit Dharmeshkumar Shah

Name of the Language:
novelC

Tools used:
Lark, Python, Prolog

Set of constraints that novelC is based on:

- The program will have a block and it ends with the “End” keyword.
- The block will contain a set of declarations followed by a list of commands.
- Declarations in the block are optional.
- All declarations and command statements will end with a semicolon.
- The language supports Boolean, Numeric and String types.
- The declaration set can contain int, string and bool declarations and initializations.
- Numeric type supports Integer (int).
- The language will support assignment operators to assign int, string, boolean value to an identifier.
- The identifier needs to be an alphabetic word.
- Identifier names need to be unique.
- String in the language needs to be in “......” double quote.

- The command can contain multiple statements including conditional statements, loops and/or assignment statements. They also can include print statements to print all supported types of values to the console.
- A new block is allowed inside the command, to declare and write commands in a nested way.
- For conditional statements, the language supports decision constructs If-elseIf-else (elseif can be 0 or more times) and ternary operator, “ boolean? expr : expr “. The output of the ternary operator needs to be assigned to an identifier.
- The language supports two types of For loop, While and Do-while loops.
- Addition, Subtraction, Multiplication, and Division will be supported by Numeric type. These shall be represented using operators such as ‘+’, ‘-’, ‘*’ and ‘/’.
- The language supports string concatenation. It shall be represented by ‘+’.
- The language supports equality comparison operations for mathematical expression and string types. These shall be represented by ‘==’.
- The language supports comparison operations for mathematical expression such as ‘!=’, ‘>’, ‘>=’, ‘<’, ‘<=’.
- The language supports update operations such as ‘++’, ‘--’.
- Supported boolean operators are ‘and’, ‘or’ and ‘not’.
- The language does not allow the user to perform a single operation on different data types. For example, adding a string to an integer variable.
- The language does not support only declarations in the program, as it does not make any sense. However, it can allow only commands in the block. For instance, a program can be as simple as ‘print “Welcome to novelC”; End’.

Extra features added:

1. *Do-while loop*
2. *Optional “elseIf” tag in decision construct statement*
3. *String concatenation*

4. *String equality comparison*
5. *Mathematical expression comparison*

Grammar in text format:

```
P ::= K End

K ::= D; C; | C;

D ::= D; D
        | int I = N | string I = S | bool I = B
        | int I | string I | bool I

C ::= C; C
        | if (B) {C} ELSEIF else {C}
        | while (B) {C}
        | do {C} while (B)
        | for (int I = N; B; UE) {C}
        | for I in range(N,N) {C}
        | I = EXP
        | I = TER
        | print VALUE
        | D

ELSEIF ::=  ELSEIF ELSEIF | elif (B) {C} | Empty

B ::= true | false
        | E == E
        | E != E
        | E < E
        | E <= E
        | E > E
        | E >= E
        | SE == SE
        | BE

UE ::= I = E | I++ | I--

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

SE ::= SE + SE | S
```

BE ::= B and B | B or B | not B | (B)

EXP ::= E | SE | BE

TER ::= B ? EXP : EXP

I ::= word
N ::= number
S ::= “ string ”
VALUE ::= I | N | S | B

Grammar in EBNF format to be used with LARK:

The lower case words here represent the rules and the upper case words represent the tokens in the below grammar:

```
program: block PEND

block: (declarations)? commands

declarations : (declaration SEMI)+

declaration: INT I ASSIGN N
         | STRING I ASSIGN S
         | BOOL I ASSIGN boolean
         | INT I
         | STRING I
         | BOOL I

commands : (command)+

command: IF OBRAC boolean CBRAC OCURL commands CCURL elseif ELSE OCURL
           commands CCURL SEMI
      | WHILE OBRAC boolean CBRAC OCURL commands CCURL SEMI
      | DO OCURL commands CCURL WHILE OBRAC boolean CBRAC SEMI
      | FOR OBRAC INT I ASSIGN N SEMI boolean SEMI updateexp CBRAC OCURL
           commands CCURL SEMI
      | FOR I IN RANGE OBRAC N COMMA N CBRAC OCURL commands CCURL SEMI
      | I ASSIGN exp SEMI
      | I ASSIGN ter SEMI
```

```
        | PRINT values SEMI
        | declarations

elseif :  (ELIF OBRAC boolean CBRAC OCURL commands CCURL)*

boolean : TRUE | FALSE
        | mathexp EQUALS mathexp
        | mathexp NOTEQUALS mathexp
        | mathexp LT mathexp
        | mathexp LTE mathexp
        | mathexp GT mathexp
        | mathexp GTE mathexp
        | stringexp EQUALS stringexp
        | boolexp

updateexp : I ASSIGN exp
        | I DPLUS
        | I DMINUS

mathexp : mathexp ADD mathexp
        | mathexp SUB mathexp
        | mathexp MUL mathexp
        | mathexp DIV mathexp
        | OBRAC mathexp CBRAC
        | I | N

stringexp : stringexp ADD stringexp | S

boolexp : boolean AND boolean
        | boolean OR boolean
        | NOT boolean
        | OBRAC boolean CBRAC

exp : mathexp | stringexp | boolexp

ter : boolean TIF exp TELSE exp

values : I | N | S | boolean

PEND : "End"
SEMI : ";"
COMMA : ","
INT     : "int"
```

```
ASSIGN  : "="
DPLUS    : "++"
DMINUS    : "--"
EQUALS    : "=="
NOTEQUALS    : "!="
LT    : "<"
LTE    : "<="
GT    : ">"
GTE    : ">="
ADD    : "+"
SUB    : "-"
MUL    : "*"
DIV    : "/"
AND    : "and"
OR    : "or"
NOT    : "not"
STRING  : "string"
BOOL    : "bool"
IF      : "if"
ELSE    : "else"
ELIF    : "elif"
WHILE    : "while"
DO        : "do"
FOR        : "for"
IN        : "in"
RANGE    : "range"
OBRAC    : "("
CBRAC    : ")"
OCURL    : "{"
CCURL    : "}"
PRINT    : "print"
TIF    : "?"
TELSE    : ":"
TRUE    : "true"
FALSE    : "false"

%import common.ESCAPED_STRING -> S
%import common.SIGNED_NUMBER -> N
%import common.WORD -> I
%import common.WS
%ignore WS
```

<u>Tools and Design choice:</u>

- **ANTLR with Java or Python**
  ANTLR takes the grammar and generates syntax trees as well as a listener to give meaning to each tree node. Here the listener is constructed in a language like Java or Python. For some part of this project, we wanted to work on Prolog, **so we decided to not use this approach**.

- **Writing own lexer, then parser and semantics in Prolog**
  We also thought of writing our own lexer, which can take a program file and give us tokens in the required format. We can use those tokens to write a parser in prolog to generate a syntax tree and then semantics in prolog to get the result. But writing lexer requires a lot of regular expression processing, and can create bugs. Moreover, using open source tools and libraries was something we wanted to do as a part of this project.

- **Using LARK to generate a syntax tree, then semantics in Prolog**
  LARK is a python library that takes grammar in EBNF format and generates tokens as well as a syntax tree for the given program. Using this we can easily generate all the tokens and in turn, use them to write the parser and semantics in prolog. However, this does not make full use of the functionalities provided in the library. To do so, we decided to generate the tokens along with the syntax tree using LARK. If not, we would have to use the grammar twice, once in LARK and again in prolog parser. From this syntax tree, we can easily write semantics in prolog. Following this approach would enable us to maximize the use of inbuilt functions in the library while working in prolog for writing semantics at the same time.

<u>How it works:</u>

- Grammar (Rules & Tokens) is defined in EBNF format.

- LARK based python program uses the above grammar.
- With a given ‘novelC’ file (Program file), LARK with python generates a syntax tree.
- The Prolog program gives semantics to the generated syntax tree.
- The syntax tree is passed to the prolog program to get the output.

Data structure used:

- List in python
- List in prolog

Sample Program:

1.

```
int x = 10;

x = false ? 1 : 2;
print x;
for i in range(1,10) {
        int z = 2;
        print z;
};
string str;
if (x/10 == 0) {
        str = “x is a ” + “multiple of 10”;
        print str;
} else {
        str = “x is not a ” + “multiple of 10”;
        print str;
};
int y = 1;
do {
      y = y * 10;
} while (not(y == 100));
print true;
End
```

2.

```
print "Hello World!";
int y = 1;
for(int i = 0; i > x + 5; i = i / 3) {
        y = y * 10;
};
int z = 5;
while(z >= 1) {
    z = z -1;
};
End
```