CS 315 - Programming Languages
Project 1

ProLogistic

Arda Kıray - 21302072 - Section 2

Table of Contents

1. The Grammar of The Language

ProLogistic uses the ASCII character set as its alphabet. The possible Tokens and Complete BNF description of the language are provided below.

1.1 Tokens:

```
DIGIT [0-9]

DOT  \.

ASSIGN \=
LEQUAL \<\=
GEQUAL \>\=
NOT_EQUAL \!\=
EQUAL \=\=
NOT \!
OR \|
AND \&
IN in
OUT out
STRING \"([^\\\"]|\\\"|\\\n|\\\\)*\"
TRUE (true|TRUE|True|{POSITIVE})
FALSE (false|False|FALSE|{NEGATIVE})
IMPLY \-\>
LESS \<
GREATER \>
IMPLY_IMPLY ({LESS}{IMPLY})
XOR \^
IF if
ELSE else
WHILE while
LEFT_PARANTEZ \(
RIGHT_PARANTEZ \)
NOR ({NOT}{OR}|{OR}{NOT})
NAND ({NOT}{AND})
NEG_SIGN \-
SUM \+
POSITIVE ({SUM}?{DIGIT}+)
NEGATIVE ({NEG_SIGN}{DIGIT}+)
LETTER [A-Za-z]
ALPHANUMERIC ({DIGIT}|{LETTER})
ID ({LETTER}{ALPHANUMERIC}*)
BOOLEAN bool
STATEMENT statement
```

VOID void
RETURN return
LCURLY \{
RCURLY \}
RBRACET \[
LBRACET \]
SEMICOLON \;
CONSTANT const
NULL null
COMMENT \#
BEG_COM \#*
END_COM *\#

1.2 BNF Description:

We used extended version of BNF and extension symbols are defined below.

[] : indicates optional argument

{} : indicates repetition of the argument

For the sake of simplicity, we used left and right parantheses, semicolons curlybraces and dot with their symbol "[]", "{}", ";", "." rather than their token names.

```
<program> ::= {<declaration> ";" | <function>}

<declaration> ::= <type> <var_decl> {"," <var_decl>}

<var_decl> ::= ID [ "[" POSITIVE "]" ]

<type> ::= BOOLEAN | STATEMENT

<truthvalue> ::= TRUE | FALSE

<function> ::= <type> ID "(" <param_dec> ")" "{" { <type> <var_decl>
               { "," <var_decl> } ";" } { <stmt> } "}"
               | VOID ID "(" <param_dec> ")" "{" { <type> <var_decl>
               { "," <var_decl> } ";" } { <stmt> } "}"

<param_decl> ::= <type> ID [ "[" "]" ] { "," <type> ID [ "[" "]" ] } | ε

<stmt> ::= <cond_stmt> | <assgn> ";" | RETURN [<expr>] ";"
           | "{" <stmt> "}" | ";"
```

```
<cond_stmt> ::= IF “(“ <expr> “)” <stmt> [ELSE <stmt>]
                  | FOR “(“ [<assgn>] “;” [<expr>] “;” [<assgn>] “)” <stmt>
                  | WHILE “(“ <expr> “)” <stmt>
<assgn> ::= ID [“[“ <expr> “]”] ASSIGN <expr> “.” <truthvalue>
            | ID ASSIGN <truthvalue>
<expr> ::= STRING | NOT <expr> | “(“ <expr> “)” | ID
           | <expr> <connectives> <expr> | <expr> <rel_op> <expr>
<connectives> ::= AND | OR | IMPLY | IMPLY_IMPLY | XOR | NOR | NAND
<rel_op> ::= LESS | GREATER |LEQUAL | GEQUAL | EQUAL | NOTEQUAL
```

1.3 Detailed Description, Conventions and Constraints:

The starting symbol of our language is the <program>. Every program starts with that symbol. A program can consists of method declarations and variable declarations which is provided by <program> ::= {<declaration> “;” | <function>} this BNF expression. Variables are declared by the rules implied by <declaration> non-terminal. <declaration> first requires a <type> non-terminal which defines the type of the declared variable. In Prologistic there are two types: Boolean and Statement. Boolean type can be declared by using bool keyword and it takes true or false values. Statement type can be declared by Statement keyword and it takes string expressions like “Eiffel Tower is taller than Mount Everest” follows with a “.”<truthvalue> which indicates the truth value of the given statement. <truthvalue> expand into either TRUE or FALSE token which is true and false value of the given expression. <declaration> lets programmer to define more than one variable with the same <type> by using <var_decl> rule which enclosed in separation curly braces along with “,” separation of each <var_decl>. <var_decl> rule specifies how a variable is declared by using its unique ID. <var_decl> ::= ID [“[“ POSITIVE “]”] has optional [POSITIVE] extension which allow program to define arrays. POSITIVE token indicates a positive integer. It is used to make sure that the given array’s size is valid(not a negative number or unrelated expression). Furthermore, this declaration lets the user to define a variable without instantiation.

Other than defining types <program> can expand into defining methods which is ruled by <function> non-terminal:

<function> ::= <type> ID "(" <param_dec> ")" "{" { <type> <var_decl>

{ "," <var_decl> } ";" } { <stmt> } "}"

| VOID ID "(" <param_dec> ")" "{" { <type> <var_decl>

{ "," <var_decl> } ";" } { <stmt> } "}"

Like a variable a function can have a <type> otherwise it must be void type which cannot return any value. However, like C, our program lets a void type to return without any value like "return;". Void type is presented by the VOID token. The types other than void must return with a value but this cannot be ensured by using BNF notation. <function> have the same function signature declaration as C like languages. Within the enclosed parentheses, <function> lets user to write parameters for the function which is <param_decl>. <param_decl> can either be a variable or an array. It is ensured by using [] optional brackets as:

<param_decl> ::= <type> ID ["[" "]"] { "," <type> ID ["[" "]"] }

After defining its signature {} represents the beginning and the end of function's body. With its body a function takes various variable declarations and statements. Statements are defined by using <stmt> non-terminal which is:

<stmt> ::= <cond_stmt> | <assgn> ";" | RETURN [<expr>] ";"

| "{" <stmt> "}" | ";"

A statement in Prologist can be a conditional statement like if-else, looping statements like while or for, an assignment or a return statement. Conditional statements declared by <cond_statement> non-terminal which indicates rules for writing a If-else statement, For statement and While statement as:

<cond_stmt> ::= IF "(" <expr> ")" <stmt> [ELSE <stmt>]

| FOR "(" [<assgn>] ";" [<expr>] ";" [<assgn>] ")" <stmt>

| WHILE "(" <expr> ")" <stmt>

The syntax of those statements are the same with C-like languages. If statement have an optional else clause which is enclosed in [] optionality indicator brackets. For statement have the snytax:

for(assingment; expression; assignment)

Howevever, each of those <assgn> and <expr> units are optional like C language for giving programmer to flexibility when writing code. A <stmt> in Prologistic can be an assignment statement which is denoted by <assgn> non-terminal. <assgn> is used for assigning a value to variables. It is :

<assgn> ::= ID [“[“ <expr> “]”] ASSIGN <expr> “.” <truthvalue>

| ID ASSIGN <truthvalue>

ASSIGN indicates the assignment operator as a token. Every assignment need to have a both right hand side and a left hand side. In the left hand side, there must be the name of of the identifier of the variable. There is two types of a left hand side. If the variable type is Statement or if it is an array, it assign by using <expr> which is:

<expr> ::= STRING | NOT <expr> | “(“ <expr> “)” | ID

| <expr> <connectives> <expr> | <expr> <rel_op> <expr>

An expression can be a string expression which is denoted by STRING token. It is basically a sequence of alphanumeric characters enclosed in quotation marks (“”). NOT <expr> lets user to assign an existing <expr> to the variable with the opposite <truthvalue>. “(“ <expr> “)” is used for ensuring the precedence among various expressions. Prologistic can also let programmer to assign a variable to another variable which is denoted by ID token. The <expr>. <expr><connectives><expr> and <expr><rel_op><expr> lets programmer to calculate the outcome of various expressions bonded with relational and connectional operator like:

<connectives> ::= AND | OR | IMPLY | IMPLY_IMPLY | XOR | NOR | NAND

<rel_op> ::= LESS | GREATER | LEQUAL | GEQUAL | EQUAL | NOTEQUAL

They are simply the tokens of the and, or, imply, if and only if, xor, nor and nand which are defined in Propositional Calculus. <rel_op> lets programmer to compare numbers and gives the result as a <truthvalue>. An assignment with an <expr> must define a truth value for that expression which is ensure by adding “.”<truthvalue> to an <expr> assignment. Assignment without <expr> can basically assign a <truthvalue> for given variable.

Associativities and Precedences:

Precedence is indicated as: from above to ground precedence decreases

Operator:	Associativity:
! (unary)	right to left
<, >, ->, <->, ==, !=, >=, =>	left to right
&, |, ^, !|, !&	left to right

Some Rules and Conventions:

- A variable's name must start with a letter and it cannot contain any special character. It is ensured by the ID token which is defined by the regular expression ID ({LETTER}{ALPHANUMERIC}*).

- A function can be defined at most once.

- Functions that does not have any return value must be defined as void.

- By convention, function names should be all lowercase.

- Language supports both single line and multi-line comments. Single line comments starts with a symbol "#" and continues at the end of the line. Multi-line comments starts with "#*" symbol and ends with "*#" symbol. Lexical analyser simply ignores all whitespace characters and comments. However, for the sake of readability those options exists as another programming languages.

- Reserved words are defined as tokens which are: bool, statement, in, out, true, false, True, False, TRUE, FALSE, if, else, while, return, const and null.

- Indentation is not required but for readability we strongly suggest the use of indentation as much as possible.

- Type of the right hand side of an assignment must be compatible with the type of the left hand side of the same assignment.

- The returned type of a method must be compatible with the type of the method which it was declared

- The conditional expression of if, while and for conditionals must be a bool type