COM2041 Lab Project 1 Report

Name, Surname: Elvin Huseynli **Date: 08.11.2021**

Student ID: 20290122 **Signature:** 

1. Description of Rules

There are some keywords that are reserved and have some functionalities:

- Keywords:
    1. "let" - It works as a function to start the program. If any variable is declared or any operation done before it is initiated, the program will give a syntax error. You cannot call this function more than once, otherwise it will give a syntax error.
    2. "done" - Its functionality consists of ending the program. If you call it anywhere in the function, it ends the program and gives an "OK" message.
    3. "whether" - It is a conditional statement that checks the truth of the expression and passes a Boolean value (or just 1 or 0) to the statement and if the value is 1, it operates the statement, else it passes out.
    4. "loop" - It is a control flow statement that checks the truth of the expression and passes a Boolean value (or just 1 or 0) to decide the next operation – to continue the flow or not.
    5. "show" - It is a function that displays the result of an expression or just the value of variables. But because the pdf specified that "do not deal with any semantic rules", it doesn't display any result, just saves the value in memory.

There are arithmetic, logical, relational, unary, postfix & assignment operators:

Let X and Y be values, variables or expressions, and VAR variable.

- Operations:
    1. **Arithmetic Operators**:
        1. "X+Y" - It operates addition operation
        2. "X-Y" - It operates subtraction operation
        3. "X/Y" - It operates division operation
        4. "X*Y" - It operates multiplication operation

    2. **Logical Operators**:
        1. "X&&Y" - It is "and" operation that returns 1 if both values are different than 0
        2. "X||Y" - It is "or" operation that returns 1 if either of values is different than 0

    3. **Relational Operators**:

1. "X>Y" - It is "greater than" operator and returns 1 if the expression is true
2. "X<Y" - It is "less than" operator and returns 1 if the expression is true
3. "X==Y" - It is "equal to" operator and returns 1 if the expression is true
4. "X>=Y"- It is "greater equal than" operator and returns 1 if the expression is true
5. "X<=Y"- It is "less equal than" operator and returns 1 if the expression is true
6. "X!=Y" - It is "not equal" operator and returns 1 if the expression is true

4. **Unary Operator:**
    1. "X = -Y" - It changes the value of X to the negative value of Y
    2. "!X" - It is "not" operation that returns 1 if the value is 0, and vice versa

5. **Postfix Operator:**
    1. "X = (Y)" - It creates a precedence of the Y

6. **Assignment Operator:**
    1. "VAR = X" - It assigns X to the variable VAR. If X contains any character or operation ruining the rule descriptions, then it will give a syntax error

Variables contain two type of data – integer and float.

Here is the description of how to declare variables and to name variables

- Data types are specified like **dec** – integer type & **rat** – float type. They are called before declaring variables. To declare variables, using assignment operator is enough.
- Variable names can contain underscore character or letter (capital or small) as first character of name, and optionally, remaining characters can contain letters (capital or small) or digits.
- Even if variable is not declared before, when it is used in an expression, it is assigned to zero as default.
- If a variable name contains any character other than specified above, it gives a syntax error.

Operators Precedence:

Note: In the table, as operators decrease in order, the precedence also decreases.

Category	Operators	Associativity
Postfix	( )	Left to right
Unary	- !	Right to left
Factor	* /	Left to right
Additive	- +	Left to right
Relational	== != > >= <=<	Left to right
Logical	&& ||	Left to right

2. Details of Grammar in BNF Notation

line ::= <**begin**> ';' <**statements**> ';'
| <**exit_command**> ';'
| <**line**><**exit_command**> ';'

statements ::= <**statement**>
| <**statements**> ';' <**statement**>

statement ::= <**exp**> | <**TYPE**><**variable**> '=' <**exp**>
| <**print**><**exp**>
| <**IF**> '(' <**exp**> ')' <**statement**>
| <**WHILE**> '(' <**exp**> ')' <**statement**>

TYPE ::= <**dec**>
| <**rat**>

exp ::= <**decimal**>
| <**variable**>

| <rational>

| '-' <exp> %prec <NEG>

| <exp> '+' <exp>

| <exp> '-' <exp>

| <exp> '*' <exp>

| <exp> '/' <exp>

| <exp> <AND> <exp>

| <exp> <OR> <exp>

| <NOT> <exp>

| <exp> '>' <exp>

| <exp> '<' <exp>

| <exp> <GE> <exp>

| <exp> <LE> <exp>

| <exp> <NE> <exp>

| <exp> <EQ> <exp>

| '(' <exp> ')'

<IF> ::= "whether"

<WHILE> ::= "loop"

<begin> ::= "let"

<exit_command> ::= "done"

<print> ::= "show"

<AND> ::= "&&"

<OR> ::= "||"

<NOT> ::= "!"

<GE> ::= ">="

<LE> ::= "<="

<NE> ::= "!="

<EQ> ::= "=="

<letter> ::= "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" | "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"

<digit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | "0"

<dec> ::= {digit}

<variable> ::= {<letter> | "_"} | {<letter> | <digit>}

<rat> ::= {digit}"."{digit}

3. Code Description

There are some functions in Yacc file that do some operations on symbol table.
SymbolVal – If it gets any character, it goes and checks the value of it in symbol table.

UpdateSymbolVal – It gets a symbol and an integer value and it makes a respective entry that passes the value of symbol to the Symbol Table.

ComputeSymbolIndex – It helps to UpdateSymbolVal function to determine the character value and returns the value for it.

4. Explanation of my own effort

After learning some materials provided by Professor, I have done some research to absorb the idea behind the lex & yacc and studied some other reports and tutorials to get a better understanding of programming languages. After this, I designed a beta version of my own Programming Language "PGM", an abbreviation of "Programming Got Me". After all, I worked much more on this, and redesigned a final version of PGM. All the explanations above regarding rules and BNF notation can be a proof that I haven't copied any source code to design a Programming Language.