Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
Bacharelado em Ciência da Computação
Compiladores

BASIC LANGUAGE

André Winston Arruda Skeete
Gabriel Araújo de Souza
Victor Agnez Lima

Natal
2019.2

Contents

1 Introduction

This is a compiler for Dartmouth BASIC language (`https://www.dartmouth.edu/basicfifty/commands.html`). Some commands were modified to allow a better use of the language.

The source code and installation instructions can be found at `https://github.com/GabrielArSouza/CompilerBasicLanguage`.

2 Lexical Analysis

2.1 Statement Symbols

Our grammar has the following statement symbols:

Statement Symbol	Description
`LET`	Introduces the assignment statement, and is required.
`PRINT`	Provides free-form output.
`END`	Is required.
`READ`	Assigns values to variables from internal data.
`DATA`	Introduces internal data.
`INPUT`	Reads data from standard input.
`GOTO`	Does just that, transfers to another line-numbered statement.
`IF`	Gives a conditional GOTO.
`FOR`	Introduces the looping construct.
`NEXT`	Terminates the looping construct.
`GOSUB`	Does a GOTO to a subroutine.
`RETURN`	Returns from the end of the subroutine.
`DEF`	Introduces programmer-defined functions.
`DIM`	Allows dimensioning arrays.
`REM`	Provides comments.
`STOP`	Same as reaching the END statement.

Table 2: Statements symbols to BASIC language

2.2 Expression Symbols

The expression symbols are the following:

Expression Symbol	Description
`ABS`	The absolute value.
`ATN`	The arctangent.
`COS`	The cosine.
`EXP`	The exponential, i.e., e^x.
`INT`	The integer part (truncating toward 0).
`LOG`	The natural logarithm.
`RND`	The next random number.
`SIN`	The sine.
`SQR`	The square root.
`TAN`	The tangent.
`OR`	The OR logic operator
`AND`	The AND logic operator
`NOT`	The NOT logic operator
`>`	The greater than operator
`<`	The less than operator
`>=`	The greater than or equals operator
`<=`	The less than or equals operator
`<>`	The difference operator
`+`	The plus symbol.
`-`	The minus symbol.
`*`	The multiplication symbol.
`/`	The division symbol.
`^`	The exponential symbol.
`=`	The equals symbol.
`%`	The module symbol.
`( , )`	The parentheses symbol.

Table 4: Expressions symbols to BASIC language

2.3 Variables

The variables start with a single letter and may also have a digit after it. Examples of variables are A, B. C, X, A1, A2.

2.4 Functions

It's also possible to define functions, whose symbols are in the format FNX, where X is any letter. Therefore, there are 26 possible user-defined functions, from FNA to FNZ.

2.5 Literals

This language has integer, float, boolean, string and character literals. Integer literals are finite sequences of digits. Float is a finite sequence of digits, followed by a dot, followed by other sequence of digits - those sequences represent the integer and fractional parts respectively. Boolean literals are either TRUE or FALSE. Strings literals start and end with double quotes (") contain a sequence (possibly empty) of characters. Characters literals are similar to strings, but they use single quotes and contain exactly one character between them. As usual, one character might be represented by more than one symbol, if it uses the escape symbol (\), e.g. the end-of-line character: '\n'.

2.6 Comments

The word REM is used to introduce a comment, and anything at the same line written on the right of that is ignored.

2.7 BNF Grammar

The syntax for this language is described by the grammar below

```
program         : stmts
                ;

end             : INTEGER END
                ;

stmts           : end
                | stmt_decl stmts
                ;

stmt_decl       : INTEGER stmt ENDL
                | ENDL
                | INTEGER ENDL

stmt            : LET variable EQUALS expr
                | PRINT expr_list
                | READ  variable_list
                | DATA  num_list
                | INPUT variable_list
                | GOTO INTEGER
                | IF expr THEN INTEGER
                | GOSUB INTEGER
                | RETURN
                | DEF FUNCTION LPAREN VARIABLE RPAREN EQUALS expr
                | DIM variable
                | STOP
                | FOR VARIABLE EQUALS expr TO expr STEP expr
                | FOR VARIABLE EQUALS expr TO expr
                | NEXT VARIABLE
                ;

num_list        : number
                | num_list COMMA number
                ;

number          : INTEGER
                | FLOAT
                | PLUS INTEGER
                | PLUS FLOAT
                | MINUS INTEGER
                | MINUS FLOAT
```

```
expr_list       : expr
                | expr_list COMMA expr
                | expr_list SEMICOLON expr
                ;

variable_list   : variable
                | variable_list COMMA variable
                ;

expr            : or_exp
                ;

or_exp          : or_exp OR and_exp
                | and_exp
                ;

and_exp         : and_exp AND rel_exp
                | rel_exp
                ;

rel_exp         : sum_exp rel_op sum_exp
                | sum_exp
                ;

sum_exp         : sum_exp PLUS prod_exp
                | sum_exp MINUS prod_exp
                | prod_exp
                ;

prod_exp        : prod_exp TIMES expo_exp
                | prod_exp DIVIDE expo_exp
                | prod_exp MOD expo_exp
                | expo_exp
                ;

expo_exp        : unary_exp EXPONENTIAL expo_exp
                | unary_exp
                ;

unary_exp       : INTEGER
                | FLOAT
                | variable
                | STRING
                | CHAR
                | BOOLEAN
```

```
                | unary_op unary_exp
                | LPAREN expr RPAREN
                | FUNCTION LPAREN expr RPAREN
                | native_function LPAREN expr RPAREN
                ;

variable        : VARIABLE
                | VARIABLE LPAREN expr RPAREN
                | VARIABLE LPAREN expr COMMA expr RPAREN
                ;

native_function : ABS
                | ATN
                | COS
                | EXP
                | INT
                | LOG
                | RND
                | SIN
                | SQR
                | TAN
                ;

unary_op        : MINUS
                | PLUS
                | NOT
                ;

rel_op          : DIFF
                | EQUALS
                | LT
                | GT
                | LTE
                | GTE
                ;
```

2.8 Lexical Analyzer

The lexical analyzer is responsible for translating BASIC code (strings) into tokens to be processed by the syntactical analyzer. Each token has a label. If a code, for instance, is formed by 120 times the number 12345, the lexical analyzer would return 120 times the token INTEGER, what would result in a syntactical error when doing the syntactical analysis. Strings that are not part of BASIC are tagged as LEXERROR (lexical error). Our lexical analyzer was developed in C++ language, using the tool Lex, and its code can be found at token.l.

3 Syntax

3.1 Commands

The syntactical structure of each command was presented in the BNF grammar (section 2.7). Also, each command was introduced in Table 2.

This section presents a more detailed description of each command and provides practical examples. Each command starts with an integer to identify the line number, and ends with a end-of-line character. The commands are sorted in increasing order by the line numbers.

3.1.1 LET

The LET command is the only assignment command in this language. It has the following structure:

```
LET <variable> = <expr>
```

Where <*variable*> is the target variable (possibly an array element) in which the result of the expression <*expr*> will be stored. Examples:

```
10 LET X = 1 + (5 + 7) / 2
20 LET Y = X - X / 2
25 LET A = "These are the values stored by the LET command: "
30 PRINT A ; X ; Y
40 END
```

3.1.2 PRINT

The PRINT command is used to output values on the screen. It starts with the word PRINT, followed by a list of at least one expression. The elements in this list may be separated by commas (,) or semicolons (;). If by commas, then it moves the start to the next zone (zones are 15 characters in width), else, by semicolons, it starts the next item at the next space.

An end-of-line is printed at the end of each PRINT statement. Also, if the width of the line printed exceeds 75 characters, an end-of-line is automatically printed.

Numeric and boolean values are printed with a space separating them. below you may find examples of usage:

```
1 PRINT "2 + 3 =" , 2 + 3
10 PRINT "Long print statement? " ; TRUE ; " Will all theses characters be
    printed on the same line? What is the answer to life? " ; FALSE ; 42
30 END
```

3.1.3 END and STOP

The END command must be the last command in the program, and appear exactly once. Should you desire to interrupt the program before reaching this command, you may use the STOP command, which can be used at any part of the code and has a similar behavior.

3.1.4 FOR and NEXT

The FOR command is used to execute a loop. Its structure can be found below:

```
FOR VARIABLE = <expr> TO <expr> [STEP <expr>]
```

The first and second expressions say the initial and final values of the variable, respectively. The STEP part is optional and it is used to inform the value to increment. Its default value is 1. All three expressions are evaluated only once.

The NEXT command indicates when the variable should be incremented and it contains the name of this variable. When the NEXT command is reached, the variable is incremented and returns to the loop condition.

The program will run the loop iteration if the variable stores a value less than or equals to the final value - if the step is non-negative - or if it stores a value greater than or equals to the final value - if the step is negative.

If the loop condition is evaluated to false, the program exits the loop by continuing running after the last NEXT command reached. If there is no such command (in case the condition fails in the first iteration), it will raise the undefined control flow error.

The example below creates an array with values from 0 to 190.

```
10 DATA 20
20 READ N
30 DIM V(N)
40 FOR I = 0 TO N-1
50 LET V(I) = I * 10
60 NEXT I
70 END
```

The next example creates an array with values from 0 to 20 and then multiplies each even position by 100.

```
10 DATA 20
20 READ N
30 DIM V(N)
40 FOR I = 0 TO N-1
50 LET V(I) = I
60 NEXT I
70 FOR I = 0 TO N-1 STEP 2
80 LET V(I) = V(I) * 100
90 NEXT I
100 END
```

3.1.5 DATA, READ and INPUT

The DATA command is followed by a sequence of numeric values, each of them being either INTEGER or FLOAT, possibly with a sign on the left. These values are then held in order to be consumed afterwards by a READ command, which receives a list of variables. The INPUT command is similar to READ, but reads from the standard input. If the value in the standard input is not numeric, then it will be read as a string.

The example below creates an array with values from -1 to 5:

```
10 DATA -1, 0, 1, +2, 3, 4, 5
20 DIM V(7)
30 FOR I = 0 TO 6
40 READ V(I)
50 NEXT I
60 END
```

And this other example asks for the first name of the user, reads it from the standard input and then prints it on the screen.

```
10 PRINT "What's your first name?"
20 INPUT X
30 PRINT "Hello, " ; X ; "!!!"
40 END
```

3.1.6 GOTO

This command's structure contains simply GOTO INTEGER, and it moves the program flow to the line specified by the INTEGER. The example below shows a program that prints the numbers from 1 to 10:

```
10 LET I = 1
20 IF I > 10 THEN 50
25 PRINT I
30 LET I = I + 1
35 GOTO 20
50 END
```

3.1.7 IF

The IF command is used to implement conditional branching in BASIC. The syntax for the command is as follows:

```
IF <expr> THEN INTEGER
```

The meaning for this can be seen as "IF the expression <*expr*> evaluates to true, THEN go to the line specified by the INTEGER". Example:

```
10 LET I = 42
20 IF I%2 > 0 THEN 40
30 PRINT "Divisible by 2"
40 IF I%3 > 0 THEN 60
50 PRINT "Divisible by 3"
60 IF I%5 > 0 THEN 80
70 PRINT "Divisible by 5"
80 END
```

3.1.8 GOSUB and RETURN

The GOSUB command, much like the GOTO command, is used to move the execution flow to another line: it contains only the keyword GOSUB and the integer which represents that line. However, there exists another command, RETURN, which is composed simply by this word, which moves the execution flow by the line after the last GOSUB command executed.

These commands implement subroutines in BASIC, making it possible to change the execution flow, run some sequence of commands, and finally return to the line that would be executed in sequence, below the GOSUB statement.

The program below implements the Euclidean algorithm to find the greatest common divisor between two integers recursively.

```
10 LET A = 28
20 LET B = 20
25 PRINT "A = " ; A ; ", B = " ; B
30 GOSUB 50
35 PRINT "GCD(A, B) = " ; C
40 STOP
50 IF B = 0 THEN 80
55 LET T = A % B
60 LET A = B
65 LET B = T
70 GOSUB 50
75 RETURN
80 LET C = A
85 RETURN
100 END
```

3.1.9 DEF

The DEF command is used to define functions. Each function has exactly one parameter and abbreviates an expression. The structure of this command is the following:

```
DEF FUNCTION ( VARIABLE ) = <expr>
```

Where VARIABLE is the parameter, that might be used or not in the expression. This parameter differs from the variable with the same name.

The example below illustrates how to define a function that returns the second power of its argument:

```
10 DEF FNN(V) = V * V
15 LET A = 5
20 LET V = 10
25 PRINT A ; " TO THE POWER OF 2 IS " ; FNN(A)
30 PRINT V ; " TO THE POWER OF 2 IS " ; FNN(V)
35 PRINT (A + V - 3) / 2 ; " TO THE POWER OF 2 IS " ; FNN((A + V - 3) / 2)
50 END
```

3.1.10 DIM

The DIM command is used to redimension a variable. It can be used to turn a variable (possibly an array) into a zero, one or two-dimensional array. The values inside the parentheses, when they exist, indicate the range of each dimension. An example with the syntax follows:

```
10 LET A = 3
20 DIM A(5,5)
30 LET A(0,0) = 3
40 END
```

The example above initializes the value 3 to a variable A, then it is redimensioned in order to transform it into a five-by-five two-dimensional array. In the last step, the element at index (0, 0) of A is set to 3.

As a side note, it is also possible to redimension an array into a single variable, by not using parentheses in the variable after the DIM command.

3.2 Look-Ahead LR Parser

Look-Ahead LR (LALR) parser is a simplified version of a LR parser which is more efficient regarding memory than a LR(1) parser, though its expressive power is not the same.

The tool YACC was used to generate the LALR parser, which is responsible for doing the syntax analysis of the code. The grammar could be simplified to the one presented in 6.1 since YACC is powerful enough to solve associativity and precedence of the operators. The actual grammar can be found at grammar.y and also contains error recovery rules and semantic actions.

Along with the syntax rules, YACC allows us to define semantic actions. It is even possible to generate code by executing these actions, though for better modularity and readability it is not done in this project. Instead, the semantic actions are used to build the abstract parse tree, defined in the following subsection.

3.3 Abstract Parse Tree

Derivations in context-free grammars might be seen as trees whose internal nodes are the non-terminal symbols, the leafs are terminal symbols, and each non-terminal symbol is the parent node of the symbols derived from it.

Such trees containing all symbols of the derivation are called concrete parse trees. Those trees are not built in practice since they contain many useless or redundant nodes, that might useful, for instance, as auxiliary syntactic symbols (e.g. the ENDL token in BASIC), or to make the grammar unambiguous. A simplified and cleaner version of this tree, which discards that unnecessary information is the abstract parse tree.

The file tree_nodes.hpp shows the structure of each node class in the abstract parse tree, which is built during the syntactical analysis in the grammar semantic actions. Each node has a friend method accept, which receives a visitor. The reason for that is that the visitor design pattern was used to improve modularity, making it possible to create a separate object responsible traversing and processing each node of the tree, which is part of the semantic analysis.

3.4 Error recovery

Each line that contains a syntax or lexical error generates an error message. The error contains the unexpected token found and the position of the statement in the file. If the line starts with an integer, which represents the line number defined by the user, this number is also displayed, in order to improve debugging. If the statement type can be identified (by the first word following the line number), this piece of information is displayed as well. Finally, another possible syntax error is the lack of the END command, so the compiler warns about this error in case of reaching the end of file unexpectedly.

4 Semantic Analysis

4.1 Static Analysis

The semantic analysis ensures that the commands written by the programmer indeed make sense. It may be divided into two parts: static and dynamic analysis. The former is done in compile time, using information from the source code, without actually running the program, while the latter is done in runtime.

Just as it is common to write context-free grammars in order to define the syntax rules of a language, the static semantic rules may be formally defined through a syntax-directed definition. This structure consists of a context-free grammar, attributes and rules. Those attributes are associated to symbols of the grammar, and are useful, for instance, to determine the type of an expression. Attributes may be constructed from the semantic rule associated with the production of that symbol (in this case, it is called synthesized attribute) or may be defined from attributes at this node's parent and siblings (called inherited attribute).

Another useful tools that the semantic rules may interact with are the table symbols. Table symbols are responsible for storing all useful information about the symbols of the code. By saying this, it is included the types of the variables, the scope of their declarations, the methods that have been defined and their signatures, among much more information.

Since block scope is not allowed in BASIC, the variables are not declared, the type of each variable is dynamic, and each function might be defined and re-defined several times in runtime, it was not necessary to use a table symbol. The file `visitor.cpp` contains the piece of code responsible for compiling the abstract parse tree into C language code. The pieces of information stored in compile time while doing the tree traversal, global variables in the referred file will be explained below.

First, we store a vector with the labels of the command lines (and their subdivisions), used create a switch structure, which is used to move the control flow to a specific label, and to verifies if the programmer asks to move to an undefined line. In this case, we opted to warn the user that this line does not exist, but generate the C code anyway. For this, we have an extra vector with the target labels of all GOTO, GOSUB and IF statements, storing also their lines to refer in the warning message. This data is also used to produce a warning in case of the same label (line number) being used in multiple lines.

Also, we store some information available only while compiling an expression. For each function call we store the expression to be evaluated as a parameter, and the name of the function that will be called. We do this in order to first evaluate the parameter, then move the control flow to the desired function, evaluate this function, and finally come back to the expression and use the result of the function there.

It is stored, for each definition of a function, its label, parameter name and code, in order to move this piece of code to other place in the compiled code. The reason for this is that we do not want to evaluate a function at the moment we define it; we want to simply store (in the activation records, during runtime) that function at that point. So, the actual implementation of the function is moved to a different part of the code, which is accessed whenever the function is called. This data structure has a key (which is the label of each user-defined function) and stores information related to what is being defined by this label (the expression of that function), just like a table symbol, however, it is important to notice that no look-up nor update operation is performed in this data structure: actually, it works more like a buffer, which just performs insertion operations and at the end reads all of the stored data. When compiling the code of an expression, we also store the name of the parameter, so we can avoid using the variable with the same name. Finally, we store a global buffer with the code being generated.

4.2 Dynamic Analysis

All the memory access operations, type checks, allocation of memory for new variables, among many others, are performed by library methods. Those methods were written in C++ and may be found in the file `activation_records.cpp`.

The contents of each variable are represented by a structure that has a label to identify its type, and the actual content as an union.

Loops and subroutines need to store information somewhere in order to be able to return to the beginning of the loop or when reaching the RETURN statement, for example. Parameters and temporary variables must be stored when functions are evaluated, since it changes the control flow, and need to be recovered after returning. All these data is stored in a stack, as it is done for most programming languages, and it is called activation records.

Also, the memory for the functions store simply a label that points to the beginning of the function implementation, and this label is updated whenever a DEF command is reached. Variables have as default value the integer 0, but they are allocated only in the first access in order to save memory.

Finally, the built-in functions are also implemented in this file, as well as all the BASIC language operators.

5 Sample Programs

5.1 Binary Search

```
1   DATA 10, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100
2
3   READ N
4   DIM V(N)
5   FOR I = 0 TO N-1
6   READ V(I)
7   NEXT I
8
9   LET L = 0
10  LET R = N - 1
11  DATA 60
12  READ X
13
14
15  IF (L <= R) THEN 17
16  GOTO 26
17  LET M = L + INT((R - L)/2)
18  IF (V(M) = X) THEN 28
19  IF (V(M) < X) THEN 21
20  IF (V(M) > X) THEN 23
21  LET L = M + 1
22  GOTO 15
23  LET R = M - 1
24  GOTO 15
25
26  PRINT -1
27  GOTO 29
28  PRINT M
29  END
```

5.2 Quick Sort

```
1 DATA 10, 6, 22, 41, 14.2, 14.3, +7, -88, -9, 14.35, 8
2 READ N
3 DIM V(N)
4 FOR I = 0 TO N-1
5 READ V(I)
6 NEXT I
7 LET L = 0
8 LET H = N-1
9 GOSUB 60 REM QUICKSORT
10 GOTO 80

19 REM : SWAP SUBPROGRAM; SWAPS POSITIONS S1 AND S2 FROM ARRAY V
20 LET T1 = V(S1)
21 LET V(S1) = V(S2)
22 LET V(S2) = T1
23 RETURN

38 REM : PARTITION SUBPROGRAM; PARTITIONS ARRAY V FROM L TO H,
39 REM   SETS PIVOT OF PARTITION TO P
40 LET X = V(H)
41 LET I = L-1
42 FOR J = L TO H-1
43 IF V(J) > X THEN 48
44 LET I = I + 1
45 LET S1 = I
46 LET S2 = J
47 GOSUB 20 REM SWAP
48 NEXT J
49 LET S1 = I+1
50 LET S2 = H
51 GOSUB 20 REM SWAP
52 LET P = I+1
53 RETURN

59 REM : QUICKSORT SUBPROGRAM; SORTS ARRAY V FROM POSITION L TO POSITION H
60 DIM S(H-L+1)
61 LET T = 0
62 LET S(T) = L
63 LET T = T+1
64 LET S(T) = H
65 IF T < 0 THEN 79
66 LET H = S(T)
67 LET L = S(T-1)
68 LET T = T-2
69 GOSUB 40 REM PARTITION
70 IF P-1 <= L THEN 74
71 LET S(T+1) = L
72 LET S(T+2) = P-1
73 LET T = T+2
74 IF P+1 >= H THEN 78
75 LET S(T+1) = P+1
76 LET S(T+2) = H
77 LET T = T+2
78 GOTO 65
79 RETURN

80 FOR I = 0 TO N - 1
81 PRINT V(I)
82 NEXT I
83 END
```

5.3 Merge Sort

```
1  INPUT N
2  DIM V(N)
3  FOR I = 0 TO N-1
4  INPUT V(I)
5  NEXT I
6
7  LET C = 1
8  IF C >= N THEN 52
9  FOR L = 0 TO N-2 STEP 2 * C
10 LET M = L + C - 1
11 IF M < N - 1 THEN 14
12 LET M = N - 1
13
14 LET R = L + 2 * C - 1
15 IF R < N - 1 THEN 19
16 LET R = N - 1
17 IF R = M THEN 20
18
19 GOSUB 24
20 NEXT L
21 LET C = 2 * C
22 GOTO 8
23
24 LET N1 = M - L + 1
25 LET N2 = R - M
26 DIM A(N1)
27 DIM B(N2)
28 FOR I = 0 TO N1 - 1
29 LET A(I) = V(L + I)
30 NEXT I
31 FOR I = 0 TO N2 - 1
32 LET B(I) = V(M + 1 + I)
33 NEXT I
34
35 LET I = 0
36 LET J = 0
37 LET K = L
38
39 IF I >= N1 AND J >= N2 THEN 51
40 IF I >= N1 THEN 47
41 IF J >= N2 THEN 43
42 IF A(I) > B(J) THEN 47
43 LET V(K) = A(I)
44 LET I = I + 1
45 LET K = K + 1
46 GOTO 39
47 LET V(K) = B(J)
48 LET J = J + 1
49 LET K = K + 1
50 GOTO 39
51 RETURN
52 FOR I = 0 TO N - 1
53 PRINT V(I)
54 NEXT I
55 END
```

6 Attachments

6.1 Yacc Grammar

```
%start program

// Operator's associativity and precedence
%left OR
%left AND
%nonassoc DIFF EQUALS LT GT LTE GTE
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right EXPONENTIAL NOT

%%
program         : stmts
                ;

end             : INTEGER END
                ;

stmts           : end
                | stmt_decl stmts
                ;

stmt_decl       : INTEGER stmt ENDL
                | ENDL
                | INTEGER ENDL
                ;

stmt            : LET variable EQUALS expr
                | PRINT expr_list
                | READ  variable_list
                | DATA  num_list
                | INPUT variable_list
                | GOTO INTEGER
                | IF expr THEN INTEGER
                | GOSUB INTEGER
                | RETURN
                | DEF FUNCTION LPAREN VARIABLE RPAREN EQUALS expr
                | DIM variable
                | NEXT VARIABLE
                | FOR VARIABLE EQUALS expr TO expr STEP expr
                | FOR VARIABLE EQUALS expr TO expr
```

```
                | STOP
                ;

num_list        : number
                | num_list COMMA number
                ;

number          : INTEGER
                | FLOAT
                | PLUS INTEGER
                | PLUS FLOAT
                | MINUS INTEGER
                | MINUS FLOAT
                ;

expr_list       : expr
                | expr_list COMMA expr
                | expr_list SEMICOLON expr
                ;

variable_list   : variable
                | variable_list COMMA variable
                ;

expr            : expr OR expr
                | expr AND expr
                | expr DIFF expr
                | expr EQUALS expr
                | expr LT expr
                | expr GT expr
                | expr LTE expr
                | expr GTE expr
                | expr PLUS expr
                | expr MINUS expr
                | expr TIMES expr
                | expr DIVIDE expr
                | expr MOD expr
                | expr EXPONENTIAL expr
                | LPAREN expr RPAREN
                | native_function LPAREN expr RPAREN
                | FUNCTION LPAREN expr RPAREN
                | NOT expr
                | MINUS expr %prec EXPONENTIAL
                | PLUS expr %prec EXPONENTIAL
                | variable
                | INTEGER
```

```
                | FLOAT
                | STRING
                | CHAR
                | BOOLEAN
                ;

variable        : VARIABLE
                | VARIABLE LPAREN expr RPAREN
                | VARIABLE LPAREN expr COMMA expr RPAREN
                ;

native_function : ABS
                | ATN
                | COS
                | EXP
                | INT
                | LOG
                | RND
                | SIN
                | SQR
                | TAN
                ;
```