RoboSCRIPT

Contributors:

- Group 50 -

Kuluhan Binici - 21601842

Mert Alp Taytak - 21602061

Yusuf Dalva - 21602867

Table of Contents

1. Variables **2**
- 1.1 Variable Names 2
- 1.2 Variable Types 2

2. Functions **3**
- 2.1 Function Definition 3
- 2.2 Primitive Functions 4
  - 2.2.1 move: (distance) -> ( ) 4
  - 2.2.2 turn: (angle) -> ( ) 4
  - 2.2.3 grab: ( ) -> ( ) 5
  - 2.2.4 release: ( ) -> ( ) 5
  - 2.2.5 readSensor: (id) -> (readData) 5
  - 2.2.6 sendData: (id, data) -> (statusCode) 5
  - 2.2.7 receiveData: (id, timeOut) -> (statusCode, receivedData) 6
  - 2.2.8 print: (data) -> ( ) 6
  - 2.2.9 scan: ( ) -> (string) 6
  - 2.2.10 wait: (time) -> ( ) 6
  - 2.2.11 time: () -> (currentTime) 7
- 2.3 Function Call 7
- 2.4 Grammar Summary for Functions 8

3. Operators **9**
- 3.1 Assignment Operator 9
- 3.2 Arithmetic Operators 9
- 3.3 Comparison Operators 10
- 3.4 Logical Operators 10
- 3.5 Context-Free Grammar Design 11

4. Control Structures **14**
- 4.1 Conditional Statements 14
  - 4.1.1 When-Otherwise Statement 14
- 4.2 Loop Statements 15
  - 4.2.1 While Loop 15
  - 4.2.2 For Loop 15

5. Comments **16**
- 5.1 Single Line Comment 16
- 5.2 Multi Line Comment 16

6. Summary of the RoboSCRIPT Grammar **16**
- 6.1 Variables 16
- 6.2 Functions: 17
- 6.3 Operators: 18
- 6.4 Control Structures: 19
- 6.5 Comments: 19

7. LEX Description File **20**

8. Example Program **22**

1. Variables

1.1 Variable Names

All variable names are made up of alphanumeric characters with no space in between. Also, a variable name can not start with a digit.

Grammar:

```
<var_declaration> -> var<id>
<id>      -> <letter><id>
           | <id><digit>
           | <id>_
           | <letter>
<digit>   -> [0-9]
<letter>  -> [a-zA-Z]
```

1.2 Variable Types

There is a single variable type called "var". However, each variable can be assigned different types of values such as integers, floating point numbers, lists and strings. Examples:

```
var a := 5;
var b := "Hello World!";
var c := list[5];
var d := -3.142;
```

Grammar:

```
<number> -> <digit><number>
          | <digit>

<string_char> -> [^"] | \n | \"
<string_body> -> <string_char><string_literal>
                  | <string_char>

<int_literal> -> +<number>
               | -<number>
               | <number>
<float_literal>  -> <int_literal>.<number>
<string_literal> -> "<string_body>"
<empty-list> -> list[<int_literal>]
```

2. Functions

All functions, user defined or primitive, share the same syntax for function signature. That is, from the function identifier they have the following construct:

": (parameter list) -> (parameter list)"

Following rules will be used throughout the functions section. Note that the check if the arguments for a function call matches the parameters for that function will be done later in the compilation process and is not a part of the grammar.

Grammar:

```
<empty>      ->
<param>      -> <empty>
             -> <id>
<param_list> -> <param>, <param_list>
             -> <param>
<func_map>   -> : (<param_list>) -> (<param_list>)
```

2.1 Function Definition

All functions are defined in the beginning of the code between *start_funcs* and *end_funcs* keywords. Function definitions start with the keyword *func*, followed by an identifier, followed by lists of inputs and outputs. Also, because there is only a single variable type *var* types are not included in the parameter lists. Examples:

```
</ A simple function./>
func add: (a, b) -> (c)
{
	c := a + b;
}

</ Zero outputs are allowed./>
func blink: (onTime, offTime) -> ( )
{
	<// Functions for controlling Led are for example purposes.
	turnOnLed: ( ) -> ( );
	wait: (onTime) -> ( );
	turnOffLed: ( ) -> ( );
	wait: (offTime) -> ( );
}
```

```
</ Multiple outputs are allowed./>
func intDivide: (number, divisor) -> (quotient, remainder)
{
      remainder := number % divisor;
      number := number - remainder;
      quotient := number / divisor;
}
```

An important note about our programming language is that there is no return statement and multiple outputs are possible. When a function is called, new variables for inputs and outputs are created. Then, inputs from the call are copied into corresponding new variables. After the function execution is over, the last assigned values of output variables from the function are copied into the corresponding variables from the output list of the function call.

Grammar:

```
<func_def> -> func <id> <func_map> {<stmts>}
```

2.2 Primitive Functions

2.2.1 move: (distance) -> ( )

Function used to move *distance* meters forwards or backwards in the current direction. When *distance* is positive, movement is forwards. When *distance* is negative, movement is backwards. *distance* can be an integer or a floating point number.

Grammar:

```
<prim_move> -> move <func_map>
```

2.2.2 turn: (angle) -> ( )

Function used to turn *angle* radians. When *angle* is positive, rotation is in the counterclockwise direction. When *angle* is negative, rotation is in the clockwise direction. *angle* can be an integer or a floating point number.

Grammar:

```
<prim_turn> -> turn <func_map>
```

2.2.3 grab: ( ) -> ( )

Function used to *grab* whatever is in the current location. No inputs and outputs are used.

Grammar:

```
<prim_grab> -> grab <func_map>
```

2.2.4 release: ( ) -> ( )

Function used to *release* whatever is currently being grabbed. When called while nothing is being grabbed, no exceptions occur and the function is executed normally.

Grammar:

```
<prim_release> -> release <func_map>
```

2.2.5 readSensor: (id) -> (readData)

Function used to read the value of the sensor with the given identifier *id*. Data read is then assigned to the argument given for the *readData* parameter.

Grammar:

```
<prim_rdsens> -> readSensor <func_map>
```

2.2.6 sendData: (id, data) -> (statusCode)

Function used to send the given *data* to the device with the given *id*. When called a value is assigned to the argument given for the *statusCode* parameter. This value depends on the success, failure and type of the failure to deliver the data.

Grammar:

```
<prim_sddata> -> sendData <func_map>
```

2.2.7 receiveData: (id, timeOut) -> (statusCode, receivedData)

Function used to listen to the device with given *id* for *timeOut* milliseconds. If connection was established successfully without timeout, then argument for *statusCode* is assigned the value for successful and the argument for *receivedData* is assigned the value of received data. Otherwise, the argument for *statusCode* is assigned the value for type of the failure.

Grammar:

```
<prim_rcdata> -> receiveData <func_map>
```

2.2.8 print: (data) -> ( )

Function used to print the value of a variable *data* to the console. *data* can be any type of variable number or string.

Grammar:

```
<prim_print> -> print <func_map>
```

2.2.9 scan: ( ) -> (string)

Function used to take input from the console. The value of the input is then assigned to the argument for *string* parameter.

Grammar:

```
<prim_scan> -> scan <func_map>
```

2.2.10 wait: (time) -> ( )

Function used to stop the code execution for *time* milliseconds. After the duration is over the program continues from the next instruction.

Grammar:

```
<prim_wait> -> wait <func_map>
```

2.2.11 time: () -> (currentTime)

Function used to get the internal time from the executing processor. Units are in milliseconds. The current internal time is assigned to the argument for the *currentTime* parameter.

Grammar:

```
<prim_time> -> time <func_map>
```

2.3 Function Call

Function calls are made by using the name of the function along with variables put into function input output lists. Examples:

```
var a, b, c, d;

a := 1;
b := 2;
add: (b, a) -> (c); </ c = 3 after the execution. />
intDivide: (c, b) -> (a, d); </ New values: c = 3, b = 2, a = 1, d =
1. Because c = a * b + d. />
```

Grammar:

```
<func_call> -> <user_func> | <prim_func>

<user_func> -> <id> : (<arg_list>) -> (<arg_list>)

<prim_func> -> <prim_move>    | <prim_turn>   | <prim_grab>
             | <prim_release> | <prim_rdsens> | <prim_sddata>
             | <prim_rcdata>  | <prim_print>  | <prim_scan>
             | <prim_wait>    | <prim_time>
```

2.4 Grammar Summary for Functions

```
<empty>        ->
<param>        -> <empty>
               -> <id>
<param_list>   -> <param>, <param_list>
               -> <param>

<func_map>     -> : (<param_list>) -> (<param_list>)
<func_def>     -> func <id> <func_map> {<stmts>}

<prim_move>    -> move <func_map>
<prim_turn>    -> turn <func_map>
<prim_grab>    -> grab <func_map>
<prim_release> -> release <func_map>
<prim_rdsens>  -> readSensor <func_map>
<prim_sddata>  -> sendData <func_map>
<prim_rcdata>  -> receiveData <func_map>
<prim_print>   -> print <func_map>
<prim_scan>    -> scan <func_map>
<prim_wait>    -> wait <func_map>
<prim_time>    -> time <func_map>

<func_call>    -> <user_func> | <prim_func>
<user_func>    -> <id> : (<arg_list>) -> (<arg_list>)
<prim_func>    -> <prim_move>    | <prim_turn>   | <prim_grab>
                | <prim_release> | <prim_rdsens> | <prim_sddata>
                | <prim_rcdata>  | <prim_print>  | <prim_scan>
                | <prim_wait>    | <prim_time>
```

3. Operators

As a programming language, the language that we have designed supports most of the basic mathematical operations that are familiar from mathematics. The programming language is capable of making arithmetic operations such as +,-,*,/,% and logical operations such as &,|,&&,||,^. These operations that are defined has a level of associativity with respect to each other which is explained below.

For the levels of associativity, the modern programming languages (Java, C++) are taken into consideration and the precedence level defined is taken similar to these popular and widely used programming languages. To identify this precedence with subtopics according to the contents of the operations, the precedence levels are:

NOT operation > Arithmetic operations > Comparison Operations > Logical Operations (except NOT) > Assignment operation

The precedence of these operators are explained in the subsections provided below.

3.1 Assignment Operator

As the assignment operator this operation has the lowest associativity level among all of the other operations. Unlike the classical approach (=), our programming language provides the ":=" operator as the assignment operator. A sample use of this operator is given below where y is the name of the variable.

```
y := a + b;
```

3.2 Arithmetic Operators

In the programming language that we have designed, arithmetic operators has the second level of highest associativity among all of the other structures. The level of associativity of these operations are:

(*, /, %) > (+, -)

Sample use of the syntax provided for these operations are given below, where a and b denoted are either identifiers or numbers (explained in BNF).

```
a + b [Addition]
a - b [Subtraction]
a * b [Multiplication]
a / b [Division]
a % b [Modulo]
```

3.3 Comparison Operators

The programming language provides operations for comparisons of different variables and constants. Different than most of the popular programming languages since ":=" is used for assignment, the language can use "=" as the equal to operator for comparison of two values. Also the operator "~=" is used as 'Not equal to' comparison. Other than these differences, the language follows Java constructs. The result of these comparisons returns 0 if the result is false and 1 if it is true. To give the association levels of the operations:

(=, ~=) > (<, <=, >, >=)

Use of comparison operators:
a = b [Equal to]
a ~= [Not equal to]
a < b [Smaller than]
a > b [Greater than]
a <= b [Smaller than or equal to]
a => b [Greater than or equal to]

3.4 Logical Operators

The programming language provides some of the fundamental logical operators in order to perform both logical and bitwise operations. Expect the NOT operation, all of these operations include two different values to make the computation, which makes them having a similar syntax. As stated before, NOT operation has the highest associativity in the language among all the other operations. Other than NOT operator, the other operators have the lowest associativity with respect to all the other operations (expect assignment). The associativity levels of the operators are given below:

(~) > (&) > (^) > (|) > (&&) > (||)

Sample uses of logical operators where a and b are either variables or constants:
a & b [Bitwise AND]
a | b [Bitwise OR]
~a [NOT]
a && b [Logical AND]
a || b [Logical OR]
a ^ b [Logical XOR]

3.5 Context-Free Grammar Design

The context-free grammar designed for the operations defined in the language is given below:

```
<assignment> -> <left-hand-side> := <assign_exp>
             | <left-hand-side> := <empty-list>
             | <left-hand-side>++
             | <left-hand-side>--
             | <left-hand-side> += <operand>
             | <left-hand-side> -= <operand>
             | <left-hand-side> *= <operand>
             | <left-hand-side> /= <operand>
<operand> -> <id> | <list-access> | <int_literal>
          | <float_literal> | <string_literal>
<left-hand-side> -> <id> | <list-access>
<list-access> -> <id>[<assign-exp>]
<assign-exp> -> <conditional-exp>
<conditional-exp> -> <conditional-or-exp>
<conditional-or-exp> -> <conditional-and-exp>
                     | <conditional-or-exp>||<conditional-and-exp>
<conditional_and_exp> -> <inclusive_or>
                       | <conditional-and-exp>&&<inclusive_or>
<inclusive_or> -> <xor_exp>
               | <inclusive_or> | <xor_exp>
<xor_exp> -> <inclusive-and-exp>
          | <xor_exp>^<inclusive-and-exp>
<inclusive-and-exp> -> <equal-exp>
                     | <inclusive-and-exp>&<equal-exp>
<equal-exp> -> <relational-exp>
            | <equal-exp> = <relational-exp>
            | <equal-exp> ~= <relational-exp>
<relational-exp> -> <sum-exp>
                 | <relational-exp> <  <sum-exp>
                 | <relational-exp> <= <sum-exp>
                 | <relational-exp> >  <sum-exp>
                 | <relational-exp> >= <sum-exp>
<sum-exp> -> <multiply-exp>
          | <sum-exp> + <multiply-exp>
          | <sum-exp> - <multiply-exp>
<multiply-exp> -> <negate-exp>
               | <multiply-exp> * <negate-exp>
               | <multiply-exp> / <negate-exp>
               | <multiply-exp> % <negate-exp>
<negate-exp> -> <operand> | ~<id>
```

Explanations of the non-terminals defined above are given:

<assignment>: This non-terminal denotes the general form of the assignments which are done with the operators. An assignment consists of <left-hand-side>, assignment operator(:=) and <assign-exp>

<left-hand-side>: This part of the assignment represents the value which the result of the operation is going to be assigned. This part can be either a identifier(<id>) or a list element(<list-access>)

<list-access>: This non-terminal represents an access to a predefined list. The list is identified with an identifier and the list index is represented with <assign-exp>

<assign-exp>: The operation that is going to be determining the result of the value that is going to be assigned is determined with this non-terminal. This expression derives to conditional expression

<conditional-exp>: This non-terminal represents an expression containing a conditional. It derives to <conditional-or-exp>

<conditional-or-exp>: The non-terminal represents an expression that includes the conditional or operation (||). This non-terminal derives to <conditional-and-exp>

<conditional-and-exp>: This non-terminal represents an expression containing conditional and operation (&&). This derives to <inclusive-or> non-terminal.

<inclusive-or>: The expressions containing an inclusive or (|) operation is determined with this non-terminal.This non-terminal derives to <xor-exp> non-terminal.

<xor-exp>: Derived from <inclusive-or> non-terminal, this non-terminal represents the expressions containing xor operator (^). <inclusive-and-exp> is derived from this non-terminal.

<inclusive-and-exp>: This non-terminal represents the expressions containng the inclusive and logical operator (&). This non-terminal derives to <equal-exp>.

<equal-exp>: The expressions involved with equality operators in the programming language designed are represented with this non-terminal. The equality operators defined in the language are “equal to”(=) and “not equal to”(~=). This non-terminal derives to <relational-exp> non-terminal.

<relational-exp>: The expressions including comparison operators are shown with this non-terminal. The comparison operators stated are “smaller than” (<), “greater than” (>), “smaller than or equal to” (<=), “greater than or equal to” (>=). This non-terminal derives to <sum-exp> non-terminal.

<sum-exp>: Ths non-terminal represents the expressions consisting summation operations, which are summation (+) and subtraction (-). This non-terminal derives to <multiply-exp> non-terminal.

<multiply-exp>: As the expression that has the second most associativity, this non-terminal consists of the operations “multiplication” (*) and “division” (/). This non-terminal derives to the non-terminal <negate-exp>

<negate-exp>: This non-terminal represents the king of operation that has the highest associativity. The operations that consists of NOT (~) symbol are represented with this non-terminal. This non-terminal derives to <operand> non terminal which shows an identifier.

4. Control Structures

Our programming language contains the control structures that are necessary for decision making and looping. We made some alterations on their syntaxes believing that those changes would increase our readability and would fit the purpose of this Programming Language better which is robot controlling.

4.1 Conditional Statements

4.1.1 When-Otherwise Statement

We substituted the keywords “if” & “else” with “when” & “otherwise” believing that they are more natural. The syntax pattern starts with the keyword “when” followed by a logical expression and curly braces that contain the block of statements to be executed if the condition is satisfied. The “when” block is succeeded by the “otherwise” block in case the condition is not satisfied. Example:

```
when (a <= 5) {
        a := a + 2;
}
Otherwise {
        a := a - 3;
}
```

Grammar:

```
<when_stmt> -> <matched> | <unmatched>
<matched>   -> when(<logic_expr>){<matched>}otherwise{<matched>}
             | <non_when>;
<unmatched> -> when(<logic_expr>){<stmt>}
             | when(<logic_expr>){<matched>}otherwise{<unmatched>}
<non_when> -> <while_loop> | <for_loop> | <assign> | <func_call>
```

4.2 Loop Statements

4.2.1 While Loop

For simplicity, we left the syntax of the while loop as it is in many popular programming languages. The loop pattern starts with “while” keyword followed by a logical expression and curly braces to contain the statements to be executed as long as the condition is satisfied. Example:

```
while (a < 10) {
      print(a);
      a := a + 1;
}
```

Grammar:

```
<while_loop> -> while(<logic_expr>){<stmts>}
```

4.2.2 For Loop

We modified the conventional for loop with a simpler one. The purpose of this new and simpler for loop is just to make iterations starting from an index and incrementing it in certain steps until it reaches the end index. The syntax pattern for this loop starts with the “for” keyword followed by the starting index, end index and the incrementation amount respectively delimited by colons. These values accept both <int_literal>’s and variables. Examples:

```
for (a := startInd; a : endInd : increment) {...}
for(index := 1; index:10:1) {...}
```

Grammar:

```
<for_loop> -> for(<assign>; <id>: <operand> : <operand>){<stmts>}
```

5. Comments

5.1 Single Line Comment

[Code can go here] <// [Rest of the line until new line becomes a comment]

5.2 Multi Line Comment

[Code can go here] </ [Multiline comment goes here] /> [Code can go here]

Grammar for both:
```
<comment> -> <single-comment> | <multi-comment>
<single-comment> -> <// <in-comment> | <multi-comment>
<multi-comment>  -> </ <in-comment> />
```

6. Summary of the RoboSCRIPT Grammar

6.1 Variables

```
<var_declaration> -> var<id>
<id>      -> <letter><id>
           | <id><digit>
           | <id>_
           | <letter>
<digit>   -> [0-9]
<letter>  -> [a-zA-Z]

<number> -> <digit><number>
          | <digit>

<string_char> -> [^”] | \n | \”
<string_body> -> <string_char><string_literal>
                  | <string_char>

<int_literal> -> +<number>
               | -<number>
               | <number>
<float_literal>  -> <int_literal>.<number>
<string_literal> -> “<string_body>”
<empty-list> -> list[<int_literal>]
```

6.2 Functions:

```
<empty>        ->
<param>        -> <empty>
               -> <id>
<param_list>   -> <param>, <param_list>
               -> <param>

<func_map>     -> : (<param_list>) -> (<param_list>)
<func_def>     -> func <id> <func_map> {<stmts>}

<prim_move>    -> move <func_map>
<prim_turn>    -> turn <func_map>
<prim_grab>    -> grab <func_map>
<prim_release> -> release <func_map>
<prim_rdsens>  -> readSensor <func_map>
<prim_sddata>  -> sendData <func_map>
<prim_rcdata>  -> receiveData <func_map>
<prim_print>   -> print <func_map>
<prim_scan>    -> scan <func_map>
<prim_wait>    -> wait <func_map>
<prim_time>    -> time <func_map>

<func_call>    -> <user_func> | <prim_func>
<user_func>    -> <id> : (<arg_list>) -> (<arg_list>)
<prim_func>    -> <prim_move>    | <prim_turn>   | <prim_grab>
                | <prim_release> | <prim_rdsens> | <prim_sddata>
                | <prim_rcdata>  | <prim_print>  | <prim_scan>
                | <prim_wait>    | <prim_time>
```

6.3 Operators:

```
<assignment> -> <left-hand-side> := <assign_exp>
              | <left-hand-side> := <empty-list>
              | <left-hand-side>++
              | <left-hand-side>--
              | <left-hand-side> += <operand>
              | <left-hand-side> -= <operand>
              | <left-hand-side> *= <operand>
              | <left-hand-side> /= <operand>
<operand> -> <id> | <list-access> | <int_literal>
           | <float_literal> | <string_literal>
<left-hand-side> -> <id> | <list-access>
<list-access> -> <id>[<assign-exp>]
<assign-exp> -> <conditional-exp>
<conditional-exp> -> <conditional-or-exp>
<conditional-or-exp> -> <conditional-and-exp>
                     | <conditional-or-exp>||<conditional-and-exp>
<conditional_and_exp> -> <inclusive_or>
                        | <conditional-and-exp>&&<inclusive_or>
<inclusive_or> -> <xor_exp>
                | <inclusive_or> | <xor_exp>
<xor_exp> -> <inclusive-and-exp>
           | <xor_exp>^<inclusive-and-exp>
<inclusive-and-exp> -> <equal-exp>
                     | <inclusive-and-exp>&<equal-exp>
<equal-exp> -> <relational-exp>
             | <equal-exp> = <relational-exp>
             | <equal-exp> ~= <relational-exp>
<relational-exp> -> <sum-exp>
                  | <relational-exp> <  <sum-exp>
                  | <relational-exp> <= <sum-exp>
                  | <relational-exp> >  <sum-exp>
                  | <relational-exp> >= <sum-exp>
<sum-exp> -> <multiply-exp>
           | <sum-exp> + <multiply-exp>
           | <sum-exp> - <multiply-exp>
<multiply-exp> -> <negate-exp>
                | <multiply-exp> * <negate-exp>
                | <multiply-exp> / <negate-exp>
                | <multiply-exp> % <negate-exp>
<negate-exp> -> <operand> | ~<id>
```

6.4 Control Structures:

```
<when_stmt> -> <matched> | <unmatched>
<matched>   -> when(<logic_expr>){<matched>}otherwise{<matched>}
             | <non_when>;
<unmatched> -> when(<logic_expr>){<stmt>}
             | when(<logic_expr>){<matched>}otherwise{<unmatched>}
<non_when> -> <while_loop> | <for_loop> | <assign> | <func_call>

<while_loop> -> while(<logic_expr>){<stmts>}
<for_loop> -> for(<assign>; <id>: <operand> : <operand>){<stmts>}
```

6.5 Comments:

```
<comment> -> <single-comment> | <multi-comment>
<single-comment> -> <// <in-comment> | <multi-comment>
<multi-comment>  -> </ <in-comment> />
```

7. LEX Description File (PART B)

```
%option noyywrap

DIGIT [0-9]
NUMBER [1-9][0-9]+
LETTER [a-zA-Z]
FNC_BEG start_funcs

%%
"start_funcs" printf("FNC_BEG ");
"end_funcs" printf("FNC_END ");
"var" printf("VAR ");
"for" printf("FOR ");
"while" printf("WHILE ");
"when" printf("WHEN ");
"otherwise" printf("OTHERWISE ");
"func" printf("FNC_DEF ");
"list" printf("LIST_INIT ");

"print"|"scan"|"move"|"turn"|"grab"|"release"|"readSensor"|"sendData
"|"receiveData"|"wait" printf("PRIM_FUNC ");

"</" printf("MLC_OPEN ");
"/>" printf("MLC_CLOSE ");
"<//" printf("SLC_START ");

"->" printf("MAP_OP ");
":=" printf("ASSIGN_OP ");
"~" printf("INV_OP ");
"+" printf("ADD_OP ");
"-" printf("SUB_OP ");
"*" printf("MULT_OP ");
"/" printf("DIV_OP ");
"%" printf("MOD_OP ");
"=" printf("EQ_OP ");
"~=" printf("INEQ_OP ");
"<" printf("LT_OP ");
">" printf("GT_OP ");
"<=" printf("LTE_OP ");
">=" printf("GTE_OP ");
"&" printf("BAND_OP ");
"&&" printf("LAND_OP ");
"|" printf("BOR_OP ");
"||" printf("LOR_OP ");
```

```
"^" printf("XOR_OP ");
"+=" printf("INC_BY ");
"-=" printf("DEC_BY ");
"*=" printf("MULT_BY ");
"/=" printf("DIV_BY ");
"++" printf("INC ");
"--" printf("DEC ");


[+-]?({NUMBER}|{DIGIT}) printf("INT_LITERAL ");
[+-]?({NUMBER}|{DIGIT})?"."({DIGIT}+) printf("FLOAT_LITERAL ");
(\"([^\"\\\n]|\\\"|\\n|\\t|\\\\)*\")|('([^'\\\n]|\\'|\\n|\\t|\\\\)*'
) printf("STRING_LITERAL ");
{LETTER}({DIGIT}|{LETTER}|_)* printf("ID ");

[\t\ ];
"," printf("COMMA ");
"(" printf("LP ");
")" printf("RP ");
"{" printf("LC ");
"}" printf("RC ");
"[" printf("LSQ ");
"]" printf("RSQ ");
";" printf("SEMI_COL ");
":" printf("COLON ");
%%

int main(int argc, char **argv) {
    yylex();
    return 0;
}
```

8. Example Program (PART C)

```
</
	Description: Test program
	Author: -
	Date: 28.10.2018
/>

start_funcs
func dummy: (a, b)->(c){
	var d := 10;
	c := (a + b) * d;
}
end_funcs

</// Program starts here

move: (10)->();
turn: (30)->();
grab: ()->();
readSensor: (127)->(readData);
sendData: (43, readData)->(statusCode);

when(statusCode ~= 0){
	print("success");
}
otherwise{
	print(“error”);
};


var testList = list[4];

var val1 := 30;
var val2 := 10;
var val3 := 5;
while(val1 >= val2){
	dummy: (val1, val2)->(result);
	print: (result)->();
	val1 := val1 - 2;
	val2 := val2 + 2;
	val3++;
	val3 *= 2;
	val3--;
	val1 /= 3;
	val2 += 5;

};

for(i:=0; i:20:2){
	print(i);
};
```