CS 315 - Project 2

Project Final Report

Group 50 - RoboSCRIPT

Group Name: RoboSCRIPT

Group Number: 50

Kuluhan Binici - 21601842 - Section 3

Yusuf Dalva - 21602867 - Section 3

Mert Alp Taytak - 21602061 - Section 3

Table Of Contents

1. Revised Context-Free Grammar

1.1 Detailed BNF description

The detailed description of the context-free grammar is given in the BNF form is given below substituted with the suitable subsections. A complete form of the BNF context-free grammar is provided in the Appendix.

1.1.1 Program Start

```
<start>       -> <program>
<program> -> start_funcs<func_block>end_funcs<stmts>
           | start_funcs<empty>end_funcs<stmts>
           | start_funcs<empty>end_funcs<empty>
           | start_funcs<func_block>end_funcs<empty>
           | <empty>

<stmts>   -> <stmt> ; <stmts>
           | <stmt> ;
<stmt>    -> <matched>|<unmatched>

<func_block> -> <func_def> <func_block>
              | <func_def>
```

1.1.2 Variables

```
<var_declaration> -> var <id>
                   | var <id> := <assignable>
                   | var <id> := <empty_list>
<ignore> -> ignore
<id>      -> <id><id_char>
           | <id><digit>
           | <id_char>

<digit>       -> 0 | <nonzero_dig>
<number>      -> <nonzero_digit><number>
               | <digit>
<int_literal> -> +<number>
               | -<number>
               | <number>
<fraction>      -> <fraction><digit>
                 | .<digit>
<float_literal> -> <int_literal><fraction>
```

```
<string_body>    -> <string_char><string_body>
                  | <string_char>
<string_literal> -> “<string_body>”

<empty_list> -> list[<number>]

<numeric_literal> -> <int_literal> | <float_literal>
```

1.1.3 Functions

```
<empty> ->
<ne_params>   -> <id>, <ne_params>
               | <id>
<func_params> -> <ne_params> | <empty>
<in_arg>       -> <id>
                | <operand>
                | <string_literal>
<in_args>      -> <in_arg>, <in_args>
                | <in_arg>
<func_in_list> -> <in_args> | <empty>
<out_arg>       -> <ignore>
                 | <id>
                 | <list_access>
<out_args>      -> <out_arg>, <out_args>
                 | <out_arg>
<func_out_list> -> <out_args> | <empty>

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

<prim_move>    -> move : (<in_arg>) -> ()
<prim_turn>    -> turn : (<in_arg>) -> ()
<prim_grab>    -> grab : () -> ()
<prim_release> -> release : () -> ()
<prim_rdsens>  -> readSensor : (<in_arg>) -> (<out_arg>)
<prim_sddata>  -> sendData : (<in_arg>, <in_arg>) ->
                  (<out_arg>)
<prim_rcdata>  -> receiveData : (<in_arg>, <in_arg>) ->
                  (<out_arg>, <out_arg>)
<prim_print>   -> print : (<in_arg>) -> ()
<prim_scan>    -> scan : () -> (<out_arg>)
<prim_wait>    -> wait : (<in_arg>) -> ()
<prim_time>    -> time : () -> (<out_arg>)

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

```
<user_func>    -> <id> : (<func_in_list>) -> (<func_out_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>
```

1.1.4 Operators

```
<assignment> -> <left_hand_side> <assign_op> <assignable>
              | <left_hand_side> <crement_op>
              | <left_hand_side> <assign_op> <empty_list>

<assignable> -> <conditional_exp> | <string_literal>

<operand> -> <id> | <list_access> | <numeric_literal>

<left_hand_side> -> <id> | <list_access>
<list_access> -> <id>[<assignable>]
<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> -> <sum_exp> + <multiply_exp>
           | <sum_exp> - <multiply_exp>
           | <multiply_exp>
<multiply_exp> -> <negate_exp>
                | <multiply_exp> * <negate_exp>
                | <multiply_exp> / <negate_exp>
                | <multiply_exp> % <negate_exp>
<negate_exp> -> <operand> | ~<id>
```

1.1.5 Control Structures

```
<matched>   -> when(<conditional_exp>){<stmts>}
               otherwise{<stmts>}
             | <non_when>
```

```
<unmatched> -> when(<conditional_exp>){<stmts>}
             | when(<conditional_exp>){<matched>}
               otherwise{<unmatched>}
<non_when> -> <while_loop> | <for_loop> | <assignment> | <func_call>
                | <var_declaration>

<while_loop> -> while(<conditional_exp>){<stmts>}
<for_loop> -> for(<assignment>;
              <operand>: <operand> : <operand>){<stmts>}
```

1.2 Explanation of Nonterminals

In this section, the meanings of the nonterminals used in the grammar provided above are explained. Explanations can be found below:

<start>: The start symbol of the program, which is the first nonterminal that is going to be resolved

<program>: The nonterminal representing the whole program

<func_block>: This nonterminal stands for the all of the function declarations specified in the program

<stmts>: All of the statements used in the program are represented with this nonterminal

<empty>: The nonterminal stand for an empty character

<stmt>: This nonterminal stands for a single statement that is used in the program

<matched>: Every statement that includes a when condition that ends with a otherwise condition is represented by this nonterminal. As when and otherwise pair matches this grammar rule is labeled as matched. As each statement that does not include a when statement has equal number of otherwise and when labels they are also recognized by this rule.

<unmatched>: every statement that includes a when statement where number of when labels are not matching with the number of otherwise labels are considered as unmatched as this nonterminal recognizes them

<func_def>: This nonterminal recognizes the function declarations in function declaration block

<var_declaration>: Declaration of variables are recognized by this nonterminal

<id>: This nonterminals stands for the identifiers present in the program. the identifiers start with a char and then can contain chars and digits

<assignable>: This nonterminal used in assignment operation specifies the instances that can be assignment which also includes conditional expressions

<empty-list>: Empty list declarations are recognized with this nonterminal

<ignore>: As the function used may not return a value, the output arguments can be ignored. The ignore state is specified with this nonterminal

<number>. Numbers declared in the program are recognized are by this nonterminal

<digit>: All of the possible digits including zero are specified by this nonterminal

<fraction>: This nonterminal recognizes the fraction part of a float

<int_literal>: This nonterminal recognizes the integer literals that are signed or unsigned numbers

<float_literal>: Numbers that are that have the type float are considered as float-literal as nonterminals

<string_body>: The body of the string is recognized by this nonterminal

<string_literal>: The complete string is recognized by this nonterminal

<ne_params>: Function parameters that are consisting at least one character are represented by this nonterminal

<func_params>. The parameters that the function takes are specified by this nonterminal

<in_args>: Arguments that are going to be taken as inputs are considered with this nonterminal

<in_arg>: An argument that is going to be an input for a function is recognized by this nonterminal

<operand>: The nonterminal considers all of the operands

<func_in_list>: List of the inputs for the function are going to be recognized by this nonterminal

<out_args>: Arguments that are going to be the outputs are considered in this nonterminal

<out_arg>: An argument that is going to be an output of a function is considered with this nonterminal

<list_access>: Accesses to the list elements are specified with this nonterminal

<func_out_list>: List of the outputs of the function are specified with this nonterminal

<func_map>: Map showing how output arguments and input arguments are going to be defined is specified with this nonterminal

<func_def>: The nonterminal showing the structure of a function definition

<prim_move>: Nonterminal showing the structure of primitive function move

<prim_turn>: Nonterminal showing the structure of primitive function turn

<prim_grab>: Nonterminal showing the structure of primitive function grab

<prim_release>: Nonterminal showing the structure of primitive function release

<prim_rdsens>: Nonterminal showing the structure of primitive function readSensor

<prim_sddata>. Nonterminal showing the structure of primitive function sendData

<prim_rcdata>: Nonterminal showing the structure of primitive function receiveData

<prim_print>: Nonterminal showing the structure of primitive function print

<prim_scan>: Nonterminal showing the structure of primitive function scan

<prim_wait>: Nonterminal showing the structure of primitive function wait

<prim_time>: Nonterminal showing the structure of primitive function time

<func_call>: function calls that can be made are specified by this nonterminal

<user_func>: User defined function calls are defined by this nonterminal

<prim_func>: Primitive function calls are defined by this nonterminal

<assignment>: Structure of assignment operation is given with this nonterminal

<left_hand_side>: Left hand side of the assignment operation is specified with this nonterminal
<conditional-exp>: conditional expressions are generalized with this nonterminal

<conditional_or_exp>: Expressions that may include conditional or expressions are specified with this nonterminal

<conditional_and_exp>: Expressions that may include conditional and expressions are specified with this nonterminal

<inclusive_or>: Expressions that may include inclusive or operation are specified by this nonterminal

<xor_exp>: Expressions that may include exclusive or operation are specified by this nonterminal

<incluive_and_exp>: expressions that may include inclusive and operation are specified by this nonterminal

<equal_exp>: expressions which can include an equality operation are specified with this nonterminal

<relational_exp>: Expressions that may include a relational operation are classified by this nonterminal

<sum_exp>: expressions that may include a summation are classified with this nonterminal

<multiply_exp>: expressions that may include a multiplication 7 division are classified by this nonterminal

<negate_exp>: expressions that may include a negation are specified by this nonterminal

<non_when>: The nonterminal for statements that do not consists when structure on the most high-level view

<while_loop>. While loop statement is stated by this nonterminal

<for_loop>: For loop statement is stated by this nonterminal

1.3 Specifications of Important Tokens

In this section the nontrivial tokens are provided by a BNF description as a symbolic interpretation and then their role are identified for clarity:

`<id_char> -> [_a-zA-Z] :` As Identifiers are composed of chars, the chars that can be included in a identifier is declared as a token

<nonzero_dig> -> [1-9] : Since the digits are able to be used everywhere, these digits are used to define the nonterminal <digit> which specified this set of characters as a token
<string_char> -> [^”] | \n | \” : Just like the case of digits can be included in numbers and chars can be included in identifiers the characters that can be included in a string are specified as a token
<assign_op> -> := | += | -= | *= | /= : The set of assignment operators is considered as a token
<crement_op> -> ++ | -- : The operatorns that does incrementation by 1 operation or decrementation by 1 operation are considered as a token

start_funcs : This token specifies the starting place of the function declarations
end_funcs. This token specifies the endpoint of function declaration block

1.4 Rules Adopted

In the programming language that we have designed there are several features that are adopted from some other popular programming languages. In this section the features adopted and use of these features in the programming language are explained.

The programming language designed, RoboSCRIPT, as a feature being able to obtain multiple outputs from a single function is an enabled feature. The programming language makes the programmer to enter both the arguments for the inputs and outputs in the form of a list. As a result of this form of implementation the user can define functions that gives multiple outputs when executed. as an example of the structure the following function declaration is an example:

function func1 (input1, input2) -> (output1, output2, output3) {...}

As seen in the example the functions does not include any return statement unlike most of the languages around. As an alternative functions present returns a singular argument or multiple arguments. For the function calls in the program, the calls can be done as:

func1(i1, i2) -> (o1, o2, o3);

For the given function prototype above.

As another feature language provides a conditional block named when - otherwise which has the sam functional characteristics with the if - else block which is used widely in the programming languages around (ie. Java). As an example the when - otherwise block seems like.

when (x < 5) {...} otherwise {...}

RoboSCRIPT does not provide any structure like else if statement but the same functionality can be achieved with nested when - otherwise statements which is a supported feature.

In addition to when - otherwise statement, language supports for loops and while loops. the examples for these structures are provided below:

while (i < 8) {...}

for (i := 0; 0:8:2) {...}

Where for the for loop the three values separated by two ":" represents initial value, limit value and incrementation amount

As a data structure RoboSCRIPT supports the list data structure list. The list can be initialized in the form "i = list[8];" for a list composed of 8 elements , where this declaration is an empty list declaration. language also supports assigning a value to a variable at the moment of initialization, which makes "var x = 5;" a valid declaration.

For the variable types, the programming language accepts declarations with word "var" which means there is only one data type specification during declaration. However after assigning a value to a variable, the type of the variable is determined according to the value assigned which means there are multiple data types supported.

2. Lexical Analyzer Implementation

2.1 Lex File Describing the Lexical Analyzer

```
DIGIT [0-9]
NONZERO_DIG [1-9]
NUMBER [1-9][0-9]+
LETTER [a-zA-Z]
ID_CHAR [_a-zA-Z]
NL           \n

%x ML_COMMENT

%%
%{
      int nesting_level = 0;
%}

"</"              BEGIN(ML_COMMENT); ++nesting_level;
"<//".*
```

```
<ML_COMMENT>"<//"
<ML_COMMENT>[^</>\n]*
<ML_COMMENT>"\n" { extern int lineno; ++lineno; }
<ML_COMMENT>"</" ++nesting_level;
<ML_COMMENT>"/>" if (--nesting_level == 0) BEGIN(INITIAL);
<ML_COMMENT>[</>]

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

"move" return(MOVE);
"turn" return(TURN);
"grab" return(GRAB);
"release" return(RELEASE);
"readSensor" return(RDSENS);
"sendData" return(SDDATA);
"receiveData" return(RCDATA);
"print" return(PRINT);
"scan" return(SCAN);
"wait" return(WAIT);
"time" return(TIME);

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

```
"|" return(BOR_OP);
"||" return(LOR_OP);
"^" return(XOR_OP);
"+=" return(INC_BY);
"-=" return(DEC_BY);
"*=" return(MULT_BY);
"/=" return(DIV_BY);
"++" return(INC);
"--" return(DEC);

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

"," return(COMMA);
"(" return(LP);
")" return(RP);
"{" return(LC);
"}" return(RC);
"[" return(LSQ);
"]" return(RSQ);
";" return(SEMI_COL);
":" return(COLON);

{NL}         { extern int lineno; ++lineno; }
.            ;
%%

int yywrap() {return 1;}
```

3. Parser Implementation

3.1 Yacc File Specifying the Parser Implementation

```
/* parser.y */

%{
#include <stdio.h>
%}
```

```
%token FNC_BEG FNC_END VAR FOR WHILE WHEN OTHERWISE FNC_DEF
LIST_INIT IGNORE MOVE TURN GRAB RELEASE RDSENS RCDATA SDDATA PRINT
SCAN WAIT TIME MAP_OP ASSIGN_OP INV_OP ADD_OP SUB_OP MULT_OP DIV_OP
MOD_OP EQ_OP INEQ_OP LT_OP GT_OP LTE_OP GTE_OP BAND_OP LAND_OP
BOR_OP LOR_OP XOR_OP INC_BY DEC_BY MULT_BY DIV_BY INC DEC
INT_LITERAL FLOAT_LITERAL STRING_LITERAL ID COMMA LP RP LC RC LSQ
RSQ SEMI_COL COLON


%%

start : program {printf("The program is accepted.\n");}

program : FNC_BEG func_block FNC_END stmts
        | FNC_BEG empty FNC_END stmts
        | FNC_BEG empty FNC_END empty
        | FNC_BEG func_block FNC_END empty
        | empty

stmts : stmt SEMI_COL stmts
      | stmt SEMI_COL

func_block : func_block func_def
           | func_def

stmt : matched
     | unmatched

matched : WHEN LP conditional_exp RP LC stmts RC OTHERWISE LC stmts
RC
        | non_when

unmatched : WHEN LP conditional_exp RP LC stmts RC
          | WHEN LP conditional_exp RP LC matched RC OTHERWISE LC
unmatched RC

non_when : while_loop
         | for_loop
         | assignment
         | func_call
         | var_declaration

while_loop : WHILE LP conditional_exp RP LC stmts RC

for_loop : FOR LP assignment SEMI_COL operand COLON operand COLON
operand RP LC stmts RC
```

```
var_declaration : VAR ID
                | VAR ID ASSIGN_OP assignable
                | VAR ID ASSIGN_OP empty_list

empty_list : LIST_INIT LSQ INT_LITERAL RSQ

numeric_literal : INT_LITERAL
                | FLOAT_LITERAL

empty :

ne_params : ID COMMA ne_params
          | ID

func_params : ne_params
            | empty

in_arg : operand
       | STRING_LITERAL

in_args : in_arg COMMA in_args
        | in_arg

func_in_list : in_args
             | empty

out_arg : IGNORE
        | ID
        | list_access

out_args : out_arg COMMA out_args
         | out_arg

func_out_list : out_args
              | empty

func_map : COLON LP func_params RP MAP_OP LP func_params RP

func_def : FNC_DEF ID func_map LC stmts RC

prim_move : MOVE COLON LP in_arg RP MAP_OP LP RP

prim_turn : TURN COLON LP in_arg RP MAP_OP LP RP

prim_grab : GRAB COLON LP RP MAP_OP LP RP
```

```
prim_release : RELEASE COLON LP RP MAP_OP LP RP

prim_rdsens : RDSENS COLON LP in_arg RP MAP_OP LP out_arg RP

prim_sddata : SDDATA COLON LP in_arg COMMA in_arg RP MAP_OP LP
out_arg RP

prim_rcdata : RCDATA COLON LP in_arg COMMA in_arg RP MAP_OP LP
out_arg COMMA out_arg RP

prim_print : PRINT COLON LP in_arg RP MAP_OP LP RP

prim_scan : SCAN COLON LP RP MAP_OP LP out_arg RP

prim_wait : WAIT COLON LP in_arg RP MAP_OP LP RP

prim_time : TIME COLON LP RP MAP_OP LP out_arg RP

func_call : user_func
          | prim_func

user_func : ID COLON LP func_in_list RP MAP_OP LP func_out_list RP

prim_func : prim_move
          | prim_turn
          | prim_grab
          | prim_release
          | prim_rdsens
          | prim_sddata
          | prim_rcdata
          | prim_print
          | prim_scan
          | prim_wait
          | prim_time

assignment : left_hand_side assign_op assignable
           | left_hand_side crement_op
           | left_hand_side ASSIGN_OP empty_list

assign_op : ASSIGN_OP
          | INC_BY
          | DEC_BY
          | MULT_BY
          | DIV_BY
```

```
crement_op : INC
           | DEC

assignable : conditional_exp
           | STRING_LITERAL

operand : ID
        | list_access
        | numeric_literal

left_hand_side : ID
               | list_access

list_access : ID LSQ assignable RSQ

conditional_exp : conditional_or_exp

conditional_or_exp : conditional_and_exp
                   | conditional_or_exp LOR_OP conditional_and_exp

conditional_and_exp : inclusive_or
                    | conditional_and_exp LAND_OP inclusive_or

inclusive_or : xor_exp
             | inclusive_or BOR_OP xor_exp

xor_exp : inclusive_and_exp
        | xor_exp XOR_OP inclusive_and_exp

inclusive_and_exp : equal_exp
                  | inclusive_and_exp BAND_OP equal_exp

equal_exp : relational_exp
          | equal_exp EQ_OP relational_exp
          | equal_exp INEQ_OP relational_exp

relational_exp : sum_exp
               | relational_exp LT_OP sum_exp
               | relational_exp LTE_OP sum_exp
               | relational_exp GT_OP sum_exp
               | relational_exp GTE_OP sum_exp

sum_exp : sum_exp ADD_OP multiply_exp
        | sum_exp SUB_OP multiply_exp
        | multiply_exp
```

```
multiply_exp : negate_exp
             | multiply_exp MULT_OP negate_exp
             | multiply_exp DIV_OP negate_exp
             | multiply_exp MOD_OP negate_exp

negate_exp : operand
           | INV_OP ID

%%
#include "lex.yy.c"
int lineno = 1;

int main()
{
    yyparse();
    return 0;
}

int yyerror(char *s){fprintf(stderr, "Line: %d, %s \n", lineno, s);}
```

3.2 Explanations of the Conflicts Occured

After the conducted tests with the lex and yacc files provided and the example program, it resulted with no conflicts.

4. Example Program

```
start_funcs

<// Sums the two numbers.
func sum: (a, b) -> (c)
{
     c := a + b;
}

</
     Uses the integer division algorithm to calculate
     the quotient and the remainder.

     </ Surely
     <// this can handle
     nested comments. <// />
/>
```

```
func intDiv: (number, divisor) -> (quotient, remainder)
{
      quotient := 0;
      remainder := 0;

      while (number >= divisor) {
      number -= divisor;
      quotient++;
      }

      remainder := number;
}

end_funcs

var message := "Welcome to the number thing.\nHere, you enter two
integers and we print the quotient from their integer division and
the sum of the inputs.";
var num1 := 20;
var num2 := 7;
var num3 := num1 - num2;
var num4;
var numList := list[5];

print: (message) -> ();
print: ("\nEnter your number: ") -> ();
scan: () -> (num1);
print: ("\nEnter your divisor: ") -> ();
scan: () -> (num2);

intDiv: (num1, num2) -> (num3, ignore);
sum: (num1, num2) -> (num4);

print: ("\nRemainder: ") -> ();
print: (num3) -> ();
print: ("\nSum: ") -> ();
print: (num4) -> ();

print: ("\nSURPRISE ROUND!") -> ();

for (i := 0; 0:10:1) {
      when (i % 2 = 0) {
      print: ("\nEVEN!") -> ();
      } otherwise {
      print: ("\nODD!") -> ();
      }
```

```
}
```

Appendices

Appendix: Complete BNF Description

```
<start>      -> <program>
<program> -> start_funcs<func_block>end_funcs<stmts>
           | start_funcs<empty>end_funcs<stmts>
           | start_funcs<empty>end_funcs<empty>
           | start_funcs<func_block>end_funcs<empty>
           | <empty>

<stmts>   -> <stmt> ; <stmts>
           | <stmt> ;
<stmt>    -> <matched>|<unmatched>

<func_block> -> <func_def> <func_block>
              | <func_def>
<var_declaration> -> var <id>
                   | var <id> := <assignable>
                   | var <id> := <empty_list>
<ignore> -> ignore
<id>      -> <id><id_char>
           | <id><digit>
           | <id_char>

<digit>       -> 0 | <nonzero_dig>
<number>      -> <nonzero_digit><number>
               | <digit>
<int_literal> -> +<number>
               | -<number>
               | <number>
<fraction>      -> <fraction><digit>
                 | .<digit>
<float_literal> -> <int_literal><fraction>

<string_body>    -> <string_char><string_body>
                  | <string_char>
<string_literal> -> “<string_body>”

<empty_list> -> list[<number>]

<numeric_literal> -> <int_literal> | <float_literal>
```

```
<empty> ->
<ne_params>   -> <id>, <ne_params>
               | <id>
<func_params> -> <ne_params> | <empty>
<in_arg>       -> <id>
                | <operand>
                | <string_literal>
<in_args>      -> <in_arg>, <in_args>
                | <in_arg>
<func_in_list> -> <in_args> | <empty>
<out_arg>       -> <ignore>
                 | <id>
                 | <list_access>
<out_args>      -> <out_arg>, <out_args>
                 | <out_arg>
<func_out_list> -> <out_args> | <empty>

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

<prim_move>    -> move : (<in_arg>) -> ()
<prim_turn>    -> turn : (<in_arg>) -> ()
<prim_grab>    -> grab : () -> ()
<prim_release> -> release : () -> ()
<prim_rdsens>  -> readSensor : (<in_arg>) -> (<out_arg>)
<prim_sddata>  -> sendData : (<in_arg>, <in_arg>) ->
                  (<out_arg>)
<prim_rcdata>  -> receiveData : (<in_arg>, <in_arg>) ->
                  (<out_arg>, <out_arg>)
<prim_print>   -> print : (<in_arg>) -> ()
<prim_scan>    -> scan : () -> (<out_arg>)
<prim_wait>    -> wait : (<in_arg>) -> ()
<prim_time>    -> time : () -> (<out_arg>)

<func_call>    -> <user_func> | <prim_func>
<user_func>    -> <id> : (<func_in_list>) -> (<func_out_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>
<assignment> -> <left_hand_side> <assign_op> <assignable>
              | <left_hand_side> <crement_op>
              | <left_hand_side> <assign_op> <empty_list>

<assignable> -> <conditional_exp> | <string_literal>

<operand> -> <id> | <list_access> | <numeric_literal>
```

```
<left_hand_side> -> <id> | <list_access>
<list_access> -> <id>[<assignable>]
<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> -> <sum_exp> + <multiply_exp>
           | <sum_exp> - <multiply_exp>
           | <multiply_exp>
<multiply_exp> -> <negate_exp>
                | <multiply_exp> * <negate_exp>
                | <multiply_exp> / <negate_exp>
                | <multiply_exp> % <negate_exp>
<negate_exp> -> <operand> | ~<id>
<matched>   -> when(<conditional_exp>){<stmts>}
               otherwise{<stmts>}
             | <non_when>
<unmatched> -> when(<conditional_exp>){<stmts>}
             | when(<conditional_exp>){<matched>}
               otherwise{<unmatched>}
<non_when> -> <while_loop> | <for_loop> | <assignment> | <func_call>
                | <var_declaration>

<while_loop> -> while(<conditional_exp>){<stmts>}
<for_loop> -> for(<assignment>;
              <operand>: <operand> : <operand>){<stmts>}
```