Bilkent University
Department of Computer Engineering

CS 315 Project Report

Programming Language: FanC++

Group Id: 23

Group Members:

Zeynep Selcen Öztunç 21902941 Section 1
Ayşe Kelleci 21902532 Section 2
Nisa Yılmaz 22001849 Section 1

Table Of Contents

1. BNF Description of the Language

1.1 Program Definitions

<program> ::= <begin_stmt><stmt_list><end_stmt>

<begin_stmt> ::= begin

<end_stmt> ::= end

<stmt_list> ::= <stmt>

| <stmt_list> <stmt>

<stmt> ::=<matched>

| <unmatched><SC> |<comment> | error

<matched> ::= if <LP><logic_expr><RP><LB><matched> <RB><SC> else<LB>

<matched><RB> <SC>

| <loop_stmt> <SC>

| <assign_stmt> <SC>

| <declaration_stmt><SC>

| <return_stmt> <SC>

| <call_stmt><SC>

|<ternary_stmt>

<unmatched> ::= if<LP><logic_expr><RP><LB><stmt_list><RB>

| if <LP><logic_expr><RP><LB><matched><RB><SC> else

<LB><unmatched> <SC><RB>

<comment> ::= <hashtag> <text><hashtag>

<identifier> ::= <letter>

| <identifier><letter>

|<identifier><digit>

|<identifier><UNDERSCORE>

1.2 Statements

<loop_stmt> ::= <while_stmt> | <for_stmt>

<while_stmt> ::= while <LP><logic_expr> <RP><LB><stmt_list><RB>

<for_stmt> ::=for <LP><assign_stmt><SC><logic_expr> <SC><assign_stmt>

<RP><LB> <stmt_list><RB>

|for <LP><var_declaration><SC><logic_expr> <SC><assign_stmt>

<RP><LB> <stmt_list><RB>

<assign_stmt> ::= <identifier> <assign> <expr>

|<identifier> <assign> <literal>

|<identifier> <assign> <call_stmt>

<declaration_stmt> ::= <var_declaration>

| <function_declaration>

<var_declaration> ::= <type_id> <ident_list>

|<type_id><assign_stmt>

<return_stmt> ::= return <expr>

|return <literal>

<ternary_stmt> ::= <logic_expr> <QM> <stmt> <stmt>

<ident_list> ::= <identifier>, <ident_list>

|<identifier>

1.3 Arithmetic Operations

<expr> ::= <expr> OR <and_term>

| <and_term>

<and_term> ::= <and_term> AND <add_sub_term>

| <add_sub_term>

<add_sub_term> ::= <add_sub_term> <op1> <mul_div_term>

|<mul_div_term>

<mul_div_term> ::= <mul_div_term> <op2> <factor>

| <factor>

<factor> ::= <LP><expr><RP>

| <val>

<val> ::= <identifier>

| <float>

| <int>

1.4 Logic

<logic_expr> ::= <int><comparison_op><int>

|<float><comparison_op><float>

|<digit><comparison_op><digit>

|<literal><comparison_op><literal>

|<identifier><comparison_op><val>

|<identifier> <EQ> <literal>

<comparison_op> ::= <LT> |<GT> | <EQ> | <LTE> | <GTE> | <NEQ>

<true> ::= true | 1

<false> ::= false | 0

1.5 Functions

<call_stmt> ::= <non_primitive_function>

| <primitive_function>

<primitive_function> ::= <url_connection>

|<check_connection>| <read_temp_data

|<read_humidity_data>

|<read_pressure_data>

|<read_quality_data>

|<read_light_data>

|<read_sound_data>

| <read_timer_data>

| <output>

|<input>

|<send_data_to_connection>

|<receive_data_from_connection>

|<switch_off>

|<switch_on>

<non_primitive_function> ::= <identifier><LP><function_input><RP>

<check_connection> ::= CHECK_CONNECTION<LP><string><RP>

<url_connection> ::= CONNECT_TO_URL<LP><string><RP>

<read_temp_data> ::= READ_TEMP_DATA <LP> <RP>

<read_humidity_data>::= READ_HUMIDITY_DATA <LP> <RP>

<read_pressure_data>::= READ_PRESSURE_DATA <LP> <RP>

<read_quality_data>::= READ_QUALITY_DATA <LP> <RP>

<read_light_data>::= READ_LIGHT_DATA <LP> <RP>

<read_sound_data>::= READ_SOUND_DATA <LP> <RP>

<read_timer_data> ::= READ_TIMER_DATA <LP> <RP>

<switch_id_list> ::= <digit> | <digit> <COMMA> <switch_id_list>

<send_data_to_connection> ::= SEND_DATA_TO_CONNECTION <LP> <string>
<COMMA><int> <RP>

|SEND_DATA_TO_CONNECTION <LP>
<string><COMMA><identifier> <RP>

|SEND_DATA_TO_CONNECTION <LP> <string>
<COMMA><int> <RP>

<receive_data_from_connection> ::= RECEIVE_DATA_FROM_CONNECTION <LP>
<string> <RP>

<switch_off> ::= SWITCH_OFF <LP> <switch_id_list> <RP>

<switch_on> ::= SWITCH_ON <LP> <switch_id_list> <RP>

<input> ::= IN<DOL>

<output> ::= OUT<DOL><output_body>

<output_body> ::= <expr> |<literal>

<function_declaration> ::= <function_header><LB><stmt_list><RB>

<function_header> ::= <identifier><LP><param_list><RP>

<param_list> ::= ε

| <type_id> <identifier>

| <type_id> <identifier> <COMMA> <param_list>

<function_input> ::= <identifier> <COMMA> <function_input>

|<literal> <COMMA> <function_input>

| <int> <COMMA> <function_input>

|<float> <COMMA> <function_input>

|<identifier>| <int>| <float>| <literal>

1.6 Types

<type_id> ::= int | float | char | string | boolean

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

<literal> ::= <string> | <boolean> |<char>

<char> ::= <CHAR_IDENT> <letter> <CHAR_IDENT>

|<CHAR_IDENT> <digit> <CHAR_IDENT>

|<CHAR_IDENT> <symbol> <CHAR_IDENT>

<symbol> ::= <LP> | <RP> | <LB> | <RB> | <LSB> | <RSB> | <COMMA> |

<SC> | <UNDERSCORE> | <ASSIGN> | <DOT> | <space> | <STRING_IDENT> |

<PLUS> | <MINUS> | <MUL> | <DIV> | <LT> | <GT> | <QM>

<string> ::= <STRING_IDENT><text><STRING_IDENT>

<text> ::= ε

|<text><letter>

|<text><digit>

|<text><symbol>

<int> ::= <unsigned_int> | <signed_int>

<float> ::=<unsigned_float> | <signed_float>

<unsigned_int> ::= <digit>

|<int><digit>

<signed_int> ::= <op1><int>

<unsigned_float> ::= <int><dot><digit>

| <float><digit>

| <dot><digit>

<signed_float> ::= <op1><float>

<boolean> ::= <true> | <false>

1.7 Symbols

<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

<LP> ::= (

<RP> ::=)

<LB> ::= {

<RB> ::= }

<LSB> ::= [

<RSB> ::=]

<QM> ::= ?

<COMMA> ::= ,

<SC> ::= ;

<COLON> ::= :

<UNDERSCORE> ::= _

<HASHTAG> ::= #

<AND> ::= &&

<OR> ::= |

<ASSIGN> ::= =

<NEWLINE> ::= \n

<DOT> ::= .

<STRING_IDENT> ::= “

<CHAR_IDENT> ::= ‘

<PLUS> ::= +

<MINUS> ::= -

<MUL> ::= *

<DIV> ::= /

<LT> ::= <

<GT> ::= >

<LTE> ::= <=

<GTE> ::= >=

<EQ> ::= ==

<NEQ> ::= !=

<DOL> ::= $

<space> ::= " "

<op1> ::= <PLUS>| <MINUS>

<op2> ::= <MUL> | <DIV>

2. Explanation of the Language Constructions

1. **<program> ::= <begin_stmt><stmt_list><end_stmt>**

   This expression represents the general structure of the fanC++language. The program is composed of a statement list. It starts with <begin> non-terminal and ends with an <end> non-terminal.

2. **<begin_stmt>::= begin**

The “begin” reserved word indicates the beginning of the program.

3. **<end_stmt>::= end**

   The “end” reserved word indicates the end of the program.

4. **<stmt_list> ::= <stmt> | <stmt_list><stmt>**

   The statement list of fanC++ consists of one or more statements; statements are separated with semicolons.

5. **<stmt> ::= <matched> | <unmatched> <SC> | <comment> | error**

   Statements are divided into two as matched statements and unmatched statements.

6. **<matched>::= if <LP><logic_expr><RP><LB><matched> <RB><SC>else<LB> <matched> <RB><SC>| <loop_stmt> <SC> | <assign_stmt><SC> | <declaration_stmt><SC>| <return_stmt><SC> | <call_stmt><SC> | <ternary_stmt>**

   Matched statements consist of loop statements, assignment statements, declaration statements, return statements, function call statements and some if statements. For the if statements to be considered as a matched statement, they must have a matching else and their contents must also be matched statements.

7. **<unmatched>::= if<LP><logic_expr><RP><LB><stmt_list><RB> | if <LP><logic_expr><RP><LB><matched><RB><SC> else <LB><unmatched> <SC> <RB>**

Unmatched statements are if statements without an else statement or matched if statements consisting of other unmatched statements.

8. **<comment>::= <hashtag> <string><hashtag>**

   The comment non-terminal is used to add comments to the program. Comments should start and end with the hashtag(#) symbol. It is possible to add both single-line and multi-line comments.

9. **<identifier> ::= <letter>|**

   <identifier><letter>|<identifier><digit>|<identifier><UNDERSCORE>

   Identifiers in fanC++ language are defined with the non-terminal identifier. An identifier must start with a letter and can be followed by letters, digits and underscores. Identifiers can be of any length (except zero) but if the length is one, it must be a letter.

10. **<loop_stmt> ::= <while_stmt> | <for_stmt>**

    This non-terminal is used to define types of loops in the program. A loop statement can either be a while_stmt non-terminal or a for_stmt non-terminal.

11. **<while_stmt>::= while <LP><logic_expr> <RP><LB><stmt_list><RB>**

This non-terminal is used to define the syntax of the while loop. While loops are composed of a while reserved keyword followed by a logic expression inside parentheses and a statement list inside brackets. While the result of the logical expression returns true the statements inside the brackets are executed, otherwise they are not executed.

12. **<for_stmt>::= for <LP><assign_stmt><SC><logic_expr> <SC><assign_stmt> <RP><LB> <stmt_list><RB>**

    This non-terminal is used to define the syntax of the for loop. For loops are defined with a for reserved keyword followed by a set of parentheses. Inside the parentheses there is an assignment or declaration statement, a logic expression and an assignment statement respectively separated by a semicolon. These parentheses are followed by a statement list inside brackets. When the result of the logic expression is true, the statements inside the statement list are executed.

13. **<assign_stmt>::= <identifier> <assign> <expr> |<identifier> <assign> <literal> |<identifier> <assign> <call_stmt>**

    This non-terminal is used to define the syntax of assignment statements. Assignments are always done with the assignment operator (=). The left-hand side of an assignment statement is always an identifier. This identifier can be assigned with different values such as string, boolean, char, int, float as well as an input or value that is returned by a function or the value of another identifier.

14. **<declaration_stmt> ::= <var_declaration> | <function_declaration>**

    This non-terminal is used to define different declaration statements. A declaration statement can either be a variable declaration or a function declaration.

15. **<var_declaration> ::= <type_id> <ident_list> |<type_id><assign_stmt>**

    This non-terminal is used to define the syntax of variable declaration statements. There are two types of declaration. Variables can be declared without a value or can be assigned a value upon declaration. A declaration statement always starts with a

type id to indicate the type of the variable. Identifiers can be declared as a list, but cannot be assigned any values. If the user wants to assign a value when declaring the variable, they should declare only one variable followed by an assignment statement.

16. **<return_stmt>::= return <expr> |return <literal>**

    This non-terminal defines the syntax of return statements. Return statements always start with the return reserved keyword followed by an expression, identifier of a literal. Return statements can be used to return a value from a function.

17. **<ternary_stmt>::= <logic_expr> <QM> <stmt> <stmt>**

    This non-terminal defines the syntax of ternary statements. It works like if statements, the program controls the result of the logic expression; if it is true, the program will go to the first statement, else the second one.

18. **<ident_list>::= <identifier> <ident_list> | <identifier>**

    This non-terminal defines either a single identifier or a list of identifiers.

19. **<expr>::=<expr> <OR> <and_term>| <and_term>**

    This non-terminal is used to express the logical or operation between and_terms or just a single and_term .

20. **<and_term>::=<and_term> <AND> <add_sub_term>|<add_sub_term>**

    This non-terminal is used to express the logical and operation between add_sub_terms or just a single add_sub_term.

21. **<add_sub_term>::=<add_sub_term><op1> <mul_div_term>| <mul_div_term>**

    This non-terminal is used to express addition/subtraction operations between mul_div_terms or just a single mul_div_term.

22. <mul_div_term>::=<mul_div_term><op2> <factor>| <factor>

This non-terminal is used to express multiplication/division operations between factors or just a single factor.

23. <factor>::=<LP><expr><RP>| <val>

This non-terminal is used to represent expressions or values.

24. <val>::= <identifier> | <float> | <int>

This non-terminal is used to express variables or int/float literals.

25. <logic_expr> ::= <int><comparison_op><int>| <digit><comparison_op><digit> <float><comparison_op><float>| <literal><comparison_op><literal> | <identifier><comparison_op><val>| <identifier><comparison_op><literal>

Logic expressions include the comparison of two expressions with less than, greater than, less than or equal, greater than or equal and equality operators. It also includes the comparison of two literals with each other with the equality operator.

26. <comparison_op> ::= <LT> |<GT> | <EQ> | <LTE> | <GTE> | <NEQ>

Comparison operators consist of less than, greater than, equality, less than or equal and greater than or equal or not equal operators.

27. <true> ::= true | 1

This non-terminal is used to represent the boolean true. It can either be represented by the true reserved keyword (terminal) or the integer 1.

28. <false>::= false | 0

This non-terminal is used to represent the boolean false. It can either be represented by the false reserved keyword (terminal) or the integer 0.

29. <call_stmt> ::= <non_primitive_function> | <primitive_function>

This non-terminal is used to define function call syntax. The language has both primitive functions and user defined functions. They can be called using the call statement.

30. <primitive_function>::= <input>| <output> |

|<url_connection>|<check_connection>

| <read_temp_data>

|<read_humidity_data>

|<read_pressure_data>

|<read_quality_data>

|<read_light_data>

|<read_sound_data>

|<read_timer_data| <output>| <input>

|<send_data_to_connection>

|<receive_data_from_connection>| <switch_off>

|<switch_on>

Primitive functions are given by default and do not need to be implemented by users. They are already defined with a fixed function header and body for convenience. These functions are used to get input/ print output from/to the terminal as well as do IoT related tasks such as reading sensor data from IoT nodes, connect to a url and/or send/receive integer data, read IoT timer data, turn switches on/off.

31. <non_primitive_function> ::= <identifier><LP><function_input><RP>

This non-terminal is used to define the syntax of non-primitive function call. Non-primitive functions are those which are implemented by users. These functions must have specific function name identifiers and necessary function input(s). The function identifier is followed by inputs enclosed in parentheses.

32. <url_connection> ::= CONNECT_TO_URL<LP><string><RP>

This non-terminal is used to define a connection to an URL, by passing the URL as a string to the CONNECT_TO_URL function.

33. <check_connection> ::= CHECK_CONNECTION<LP><string><RP>

This non-terminal is used to define a URL connection check by passing the URL as a string. If this method returns true, this means a connection can be established.

34. <read_temp_data>::= READ_TEMP_DATA <LP> <RP>

This non-terminal is used to read temperature data from temperature sensors, by READ_TEMP_DATA function.

35. <read_humidity_data>::= READ_HUMIDITY_DATA <LP> <RP>

This non-terminal is used to read humidity data from humidity sensors, by READ_HUMIDITY_DATA function

36. <read_pressure_data>::= READ_PRESSURE_DATA <LP> <RP>

This non-terminal is used to read air pressure data from sensors, by READ_PRESSURE_DATA function

37. <read_quality_data>::= READ_QUALITY_DATA <LP> <RP>

This non-terminal is used to read air quality data from air quality sensors, by READ_QUALITY_DATA function

38. **<read_light_data>::= READ_LIGHT_DATA <LP> <RP>**

    This non-terminal is used to read light data from light sensors, by READ_LIGHT_DATA function

39. **<read_sound_data>::= READ_SOUND_DATA <LP> <RP>**

    This non-terminal is used to read sound data from sensors, by READ_SOUND_DATA function

40. **<read_timer_data> ::= READ_TIMER_DATA <LP> <RP>**

    This non-terminal is used to define the READ_TIMER_DATA function which reads the data on the timer.

41. **<switch_id_list> ::= <digit> | <digit> <COMMA> <switch_id_list>**

    This non-terminal is used to specify the switch ids which can either be the id of a single switch or id of multiple switches.

42. **<send_data_to_connection> ::= SEND_DATA_TO_CONNECTION <LP> <string> <COMMA><int> <RP>| SEND_DATA_TO_CONNECTION <LP> <string><COMMA><identifier> <RP> | SEND_DATA_TO_CONNECTION <LP> <string> <COMMA><digit> <RP>**

    This non-terminal defines the SEND_DATA_TO_CONNECTION function syntax which takes two parameters: it either takes the URL as a string and the data to be sent as an integer or it takes the URL as a string and data to be sent as an identifier.

43. <receive_data_from_connection>::= RECEIVE_DATA_FROM_CONNECTION <LP> <string> <RP>

This non-terminal defines the RECEIVE_DATA_TO_FROM_CONNECTION function syntax, which takes a URL as input to establish a connection to it and to receive data.

44. <switch_off>::= SWITCH_OFF <LP> <switch_id_list> <RP>

This non-terminal defines the SWITCH_OFF function syntax which takes the ids of the switches that will be turned off as a parameter and turns those switches off.

45. <switch_on>::= SWITCH_ON <LP> <switch_id_list> <RP>

This non-terminal defines the SWITCH_ON function syntax which takes the ids of the switches that will be turned on as a parameter, and turns those switches on.

46. <input>::= IN<DOL>

This non-terminal takes input from the user which can be of any type.

47. <output> ::= OUT<DOL><output_body>

This non-terminal prints the output body, which can either be an expression or a literal.

48. <output_body> ::= <expr> |<literal>

The output body can be either an expression or a literal.

49. <function_declaration>::= <function_header><LB><stmt_list><RB>

The function declaration is a non-terminal expression to declare a function by using its header and statements inside the scope of that function. Function header is followed by a statement list enclosed in brackets.

50. <function_header>::= <identifier><LP><param_list><RP>

The function header is a non-terminal expression to specify the function header with a specific identifier name and parameter list required for operations

51. <param_list> ::= ε| <type_id> <identifier> | <type_id> <identifier> <COMMA> <param_list>

The parameter list is a non-terminal and includes parameters taken by the function to do operations with given values. The parameter list can include nothing, one variable, or multiple variables separated by comma.

52. <function_input> ::= <identifier> <COMMA> <function_input> |<literal> <COMMA> <function_input>| <int> <COMMA> <function_input> |<float> <COMMA> <function_input>|<identifier>| <int>| <float>| <literal>

Function parameter list can either be empty, or it can consist of one or multiple identifiers, literals, integers and/or floats.

53. <type_id>::= int | float | char |string | boolean

This non-terminal defines the terminals used to identify 6 types included in fanC++ language which are int, float, char, string and boolean.

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

Digit non-terminal defines the digit terminals.

55. <literal> ::= <string> | <boolean> |<char>

Literal non-terminal define string, boolean and char non-terminals. The reason why integer and float is not included is to avoid certain appearances of ambiguity.

56. <char>::= <CHAR_IDENT> <letter> <CHAR_IDENT>|<CHAR_IDENT> <digit> <CHAR_IDENT> |<CHAR_IDENT> <symbol> <CHAR_IDENT>

This non-terminal defines the characters in the language. Characters are single letters, digits, or symbols enclosed in single quotes.

57. <string> ::= <STRING_IDENT><text><STRING_IDENT>

This non-terminal defines strings in the language. Strings are texts enclosed in quotes.

58. <text> ::= ε|<text><letter>|<text><digit>|<text><symbol>

This non-terminal defines the syntax of a text. Text is any combination of any character.

59. <int> ::= <unsigned_int> | <signed_int>

This non-terminal defines the syntax of an integer. Integers can be either signed or unsigned.

60. <float> ::=<unsigned_float> | <signed_float>

This non-terminal defines the syntax of a float value. Floats can either be signed or unsigned.

61. <unsigned_int> ::= <digit> |<int><digit>

This non-terminal defines the syntax of an unsigned integer. Unsigned integers can be a single digit or multiple digits.

62. <signed_int> ::= <op1><int>

This non-terminal defines the syntax of a signed integer. Signed integer is a + or - followed by an unsigned integer.

63. <unsigned_float> ::= <int><dot><digit>| <float><digit>| <dot><digit>

This non-terminal defines the syntax of an unsigned float. Unsigned floats can be integers followed by a decimal point and one or more decimal values. Or they can be a decimal point followed by one or more decimal values.

64. <signed_float> ::= <op1><float>

This non-terminal defines the syntax of a signed float. Signed float is a + or - followed by an unsigned float.

65. <boolean>::= <true> | <false>

This non-terminal defines the non-terminals of boolean values true and false.

66. <symbol>::= <LP> | <RP> | <LB> | <RB> | <LSB> | <RSB> | <COMMA> | <SC> | <UNDERSCORE> | <ASSIGN> | <DOT> | <space> | <STRING_IDENT> | <PLUS> | <MINUS> | <MUL> | <DIV> | <LT> | <GT> | <QM>

This non-terminal defines all the symbols in the language.

67. <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

This non- terminal defines the letters of the English alphabet in both lower and upper case.

3. Non-Trivial Tokens of the Language

if : This token is a reserved keyword for if statements.

else : This token is a reserved keyword for else statements.

while : This token is a reserved keyword to use in while loops.

for : This token is a reserved keyword to use in for loops.

begin : This token is a reserved keyword to indicate the beginning of the programs.

end : This token is a reserved keyword to indicate the end of the programs.

return : This token is a reserved keyword to indicate return statements of the functions.

in : This token is a reserved keyword to get input from the user.

out : This token is a reserved keyword to print output to the terminal.

int : This token is a reserved keyword to indicate the type of integers.

float : This token is a reserved keyword to indicate the type of floats.

char : This token is a reserved keyword to indicate the type of characters.

string : This token is a reserved keyword to indicate the type of strings.

bool : This token is a reserved keyword to indicate the type of booleans.

#...#: This token is reserved to add comments to the code.

4. Evaluation of the Language

4.1 Motivation for Design

The FanC++ language is designed in a way that is simple to read and write. It is similar to many of the imperative languages, with a more straightforward syntax. It also offers primitive functions to program IoT devices that are easy to use.

4.2 Readability

The structure of the language and naming were designed very similarly to the most preferred programming languages. Statements are separated with semicolons, and variables are separated with commas. Also, it uses meaningful keywords. They increase the readability of the program. The language does not have multiple ways to do the same operation, therefore it is readable.

4.3 Writability

The FanC++ language has high writability because of the simple definitions that its syntax has. Variable declarations, loops, conditionals, types, statements and symbols are simple and straightforward. For example, for the loop structure there is only a for and a while loop, the operators are similar to their original mathematical definitions etc.

4.4 Reliability

Every primitive type has a type identifier in FanC++ language. These identifiers are created to establish reliability of the language. By using these identifiers type checking can be done and illegal operations can be prevented (for example integer and float cannot be added). Also comments are enclosed in hashtag signs which is also a reliability practice. If and loop statements are enclosed with brackets, it clearly determines the beginning and ending points of the statements.

5. Example Programs

5.1 Example Program 1

```
begin
string my_string_1 = "I love cs315";
#error: there should be a semicolon after assignment statements in for loops #
for (int iter = 0, iter <= 10; iter = iter + 1) {
   string my_input = IN$;
   if( my_string_1 == my_input) {
        OUT$"my_string_1 contains the given string";
```

```
  };
  else {
        x+ 5 /6 * 8
        #error: in the above line, logic expression should be assigned to a variable #
        OUT$"b doesn't contain the given string";
  };
};
end
```

5.2 Example Program 2

```
begin
float newFloat = 1.15;
int b = 3;
#error: if statement should include comparison operation, not assignment#
if( int a = 5)
{
        a = 7 + 9;
};
int a = 5*b+6/3 -4;
while(newFloat < 12){
        newFloat=newFloat+1;
};
OUT$ newFloat;
b < 3 ? b = b+1;  b = a*2;

end
```

5.3 Example Program 3

```
begin
#error: READ_HUMIDITY_DATA function does not take any parameters#
float data_of_sensor_1 = READ_HUMIDITY_DATA(4);
float data_of_sensor_5 = READ_TEMP_DATA();
string data_of_timer  = READ_TIMER_DATA();
CONNECT_TO_URL("https://www.cs.bilkent.edu.tr/~guvenir/courses/CS315/Pr1");
SEND_DATA_TO_CONNECTION("my_example_link.com", 5);

my_example_function(float a, float b) {
        float sum = a + b;
        if(sum >= 1.2) {
                SWITCH_ON(1,4,6);
                SWITCH_OFF(2);
        };
```

```
#error: Switch_off method can take integer values as a parameter but it takes float#
SWITCH_OFF( 5 ,  4.3);
        return sum;
};

#if the sum of datas from sensors 1 and 5 are greater or equal to 1.2 turn switches 1,4,6 on
and 2 off. Also return the sum#

my_example_function(data_of_sensor_1, data_of_sensor_5);

end
```

Notes:

1)There is only a single conflict in our parser, which is a shift/reduce conflict caused by if-else statements. If-else statement includes matched statements inside it. Also, stmt includes matched and unmatched statements. When the program is confronted with a matched statement, both rules want to reduce it. Therefore there is a shift/reduce conflict. However, it does not cause any problem because the following parts of the two rules are different, therefore the program can understand which rule corresponds to this statement.

2)If there is an error in while, for, if statements, the closing bracket will also cause a syntax error because the first part of those statements aren't recognized as syntactically correct so the parser cannot identify the closing brackets as well.

3)If there is more than one error on a line (for example in the logic expression inside for(), if more than one part is syntactically wrong) the parser will print more than one error warning. This is because it catches every error, and doesn't stop when it encounters an error.