CS 315

Programming Languages

Project 2 Report

I-nur!

Project Group: 29

- Kutay Tire - 22001787 - Section 2
- Atak Talay Yücel - 21901636 - Section 2
- Yiğit Yalın - 22002178 - Section 2

Supervisor: Aynur Dayanık

Table Of Contents

1. Nontrivial Tokens

<AIR_PRESSURE> ::= **airPressure**

<AIR_QUALITY> ::= **airQuality**

<AND> ::= **&&**

<ASSIGNMENT_OP> ::= **:=**

<BASIC_SYMBOL> ::= **!|$|%|&|(|)|*|+|,|-|.|/|:|;|>|=|<|?||@|[|\|]|^|_|`|{|}|~**

<GO> ::= **go**

<BOOLEAN> ::= **TRUE|FALSE**

<CONST> ::= **constant**

<COLON> ::= **:**

<COMMA> ::= **,**

<COMMENT_SYMBOL> ::= **#**

<DEFINE_CONNECTION> ::= **defineConnectionToURL**

<DIVIDE> ::= **/**

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

<DOT> ::= **.**

<ELSE> ::= **else**

<EQUAL> ::= **==**

<FOR> ::= **for**

<FUNCTION> ::= **function**

<GREATER_THAN> ::= **>**

<GREATER_EQUAL_THAN> ::= **>=**

<HUMIDITY> ::= **humidity**

<IF> = **if**

<INPUT> ::= **input**

<LB> ::= **{**

<LESS_EQUAL_THAN> ::= **<=**

<LESS_THAN> ::= **<**

<LIGHT> ::= **light**

<LOWER_ALPHABETIC> ::= **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> ::= **(**

<MINUS> ::= **-**

<MODULO> ::= **%**

<MULTIPLY> ::= *****

<NL> ::= **\n**

<NOT> ::= **!**

<NOT_EQUAL> ::= **!=**

<OR> ::= **||**

<PLUS> ::= **+**

<PRINT> ::= **print**

<RECIEVE_INTEGER> ::= **recieveIntegerValue**

<RETURN> ::= **return**

<RB> ::= **}**

<RP> ::= **)**

<SEND_INTEGER>::= **sendIntegerValue**

<SOUND_LEVEL> ::= **soundLevel**

<SWITCH_OFF> ::= **switchOff**

<SWITCH_ON> ::= **switchOn**

<TEMPERATURE> ::= **temperature**

<TIME_STAMP> ::= **timeStamp**

<UPPER_ALPHABETIC> ::=

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**

<WHILE> ::= **while**

<ALPHABETIC> ::= <LOWER_ALPHABETIC>|<UPPER_ALPHABETIC>

<ALPHANUMERIC> ::= <ALPHABETIC>|<DIGIT>

<CHARACTER> ::= <ALPHANUMERIC>|<BASIC_SYMBOL>

<COMMENT> ::= <COMMENT_SYMBOL><TEXT>

<COMMENT_SYMBOL>

<EMPTY> ::=

<FLOAT> ::= <MINUS><FLOAT_NO_SIGN>|<FLOAT_NO_SIGN>

<FLOAT_NO_SIGN> ::= <INTEGER_NO_SIGN><DOT>

<INTEGER_NO_SIGN>|<DOT><INTEGER_NO_SIGN>|

<INTEGER_NO_SIGN><DOT>

<IDENTIFIER> ::= <LOWER_ALPHABETIC><IDENTIFIER_CONTINUE>

<IDENTIFIER_CONTINUE> ::= <IDENTIFIER_CONTINUE>

<ALPHANUMERIC>|<EMPTY>

<INTEGER> ::= <MINUS><INTEGER_NO_SIGN>

|<INTEGER_NO_SIGN>

<INTEGER_NO_SIGN> ::= <DIGIT> |<DIGIT>

<INTEGER_NO_SIGN>

<STRING> ::= “<TEXT>”

<TEXT> ::= <EMPTY>|<CHARACTER>|<CHARACTER><TEXT>

2. Program Definition

<program> ::= <GO><LP><RP><LB><statement_list><RB>

Statements

<statement_list> ::= <NL>|<statement><NL>

|<statement_list><NL>|<statement_list><statement><NL>

<statement> ::= <assignment>|<comment>|<conditional>|

<function_declaration>|<expression>|<loop>|<output>|<return>

Assignment

<assignment> ::= <CONST><IDENTIFIER><ASSIGNMENT_OP>

<expression>|<IDENTIFIER><ASSIGNMENT_OP><expression>

<expression> ::= <logic_expression>|<STRING>

<logic_expression> ::= <logic_expression><OR>

<and_expression>|<and_expression>

<and_expression> ::= <and_expression><AND>

<not_expression>|<not_expression>

<not_expression> ::= <NOT><not_expression>|<BOOLEAN>|

<comparison_expression>

<comparison_expression> ::= <math_expression>

<EQUAL><comparison_expression>|<math_expression>

<NOT_EQUAL><comparison_expression>|<math_expression>

<LESS_EQUAL_THAN><comparison_expression>|<math_expression>

<GREATER_EQUAL_THAN><comparison_expression>|

<math_expression><GREATER_THAN><comparison_expression>|

<math_expression><LESS_THAN><comparison_expression>|

<math_expression>

<math_expression> ::= <math_expression><PLUS>

<multiply_divide_modulo>|<math_expression><MINUS>

<multiply_divide_modulo>|<multiply_divide_modulo>

<multiply_divide_modulo> ::= <multiply_divide_modulo>

<DIVIDE><value>|<multiply_divide_modulo>

<MULTIPLY><value>|<multiply_divide_modulo>

<MODULO><value>|<value>

<value> ::= <function_call>|<IDENTIFIER>|<INTEGER>|<FLOAT>|

<LP> <expression><RP>

Function Calls

<function_call> ::= <IDENTIFIER><LP><identifiers><RP>|

<IDENTIFIER><LP><RP>|<iot_functions>|<input>

<input> ::= <INPUT><LP><RP>

<iot_functions> ::= <iot_read_value>|<switch_on>|<switch_off>|

<define_connection>|<send_integer>|<recieve_integer>

<iot_read_value> ::= <temperature>|<humidity>|<air_pressure>|

<air_quality>|<light>|<sound_level>|<time_stamp>

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

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

<define_connection> ::= <DEFINE_CONNECTION><LP><STRING>

<RP>|<DEFINE_CONNECTION><LP><IDENTIFIER><RP>

<send_integer> ::= <SEND_INTEGER><LP><define_connection>

<COMMA><INTEGER><RP>

<recieve_integer> ::= <RECIEVE_INTEGER><LP>

<define_connection><RP>

<temperature> ::= <TEMPERATURE><LP><RP>

<humidity> ::= <HUMIDITY><LP><RP>

<air_pressure> ::= <AIR_PRESSURE ><LP><RP>

<air_quality> ::= <AIR_QUALITY><LP><RP>

<light> ::= <LIGHT><LP><RP>

<sound_level> ::= <SOUND_LEVEL><LP><RP>

<time_samp> ::= <TIME_STAMP><LP><RP>

Comment

<comment>::= <COMMENT>

Conditional

<conditional> ::= <matched>|<unmatched>

<matched> ::= <IF><LP><logic_expression><RP><LB>

<statement_list><RB><ELSE><LB><statement_list><RB>

|<IF><LP><logic_expression><RP><LB>

<matched><RB><ELSE><LB><statement_list><RB>

|<IF><LP><logic_expression><RP><LB>

<statement_list><RB><ELSE><LB><matched><RB>

|<IF><LP><logic_expression><RP><LB>

<matched><RB><ELSE><LB><matched><RB>

<unmatched> ::= <IF><LP><logic_expression><RP><LB>

<statement_list><RB>|<IF><LP><logic_expression><RP><LB>

<statement_list><RB><ELSE><LB><unmatched><RB>|<IF><LP>

<logic_expression><RP><LB><matched><RB>

<ELSE><LB><unmatched><RB>

Function Declaration

<function_declaration> ::= <FUNCTION><IDENTIFIER><LP>

<identifiers><RP><LB><statement_list><return><RB>|

<FUNCTION><IDENTIFIER><LP><RP><LB><statement_list>

<return><RB>

<identifiers> ::= <IDENTIFIER>|<IDENTIFIER><COMMA>

<identifiers>

Return

<return> ::= <RETURN><expression><NL>|<RETURN><NL>

Loops

<loop> ::= <for_loop>|<while_loop>

<for_loop> ::= <FOR><LP><IDENTIFIER><COLON><range><RP>

<LB><statement_list><RB>

<while_loop> ::= <WHILE><LP><logic_expression><RP><LB>

<statement_list><RB>

<range> ::= <LP><INTEGER> <COMMA><INTEGER><COMMA>

<INTEGER><RP>|<LP><INTEGER><COMMA><INTEGER><RP>|

<LP><INTEGER><RP>

Output

<output> ::= <PRINT><LP><output_body><RP>

<output_body> ::= <expression>|<expression><COMMA>

<output_body>

3. Nontrivial Token Explanations

3.1. Reserved Keywords

The language has some reserved keywords for:

- Program start: **go**
- Loops: **for, while**
- Conditionals: **if, else**
- Constant Declaration: **constant**
- Functions: **function, return**
- Input and Output: **input, print**
- Primitive functions for IoT operations: **airPressure, airQuality, light, temperature, soundLevel, humidity, timeStamp, switchOn, switchOff, sendIntegerValue, recieveIntegerValue, defineConnectionToURL**

3.2. Comments

The comment symbol is # which is used in the comment statements in the language. Comment statements are written between two "#" symbols. For example: # This needs an update #. We chose # as the comment symbol because it is easier to remember and to use than the other symbols in C++ or Java.

3.3. Literals

<BOOLEAN>:

It consists of **TRUE** and **FALSE**.

<INTEGER>:

Integer literals are defined with the help of <INTEGER_NO_SIGN>. By adding a "-" at the beginning, the language allows negative integers and if there is no sign in front of it, it considers the integer as positive.

<INTEGER_NO_SIGN>:

It creates the possible unsigned integer representations by using recursion and placing a <DIGIT> at each step.

<FLOAT>:

Similar to <INTEGER>, float literals are defined with the help of <FLOAT_NO_SIGN>. By adding a "-" at the beginning, the language allows negative floating numbers and if there is no sign, it considers the number as positive. Some floating number examples are "1.2" , "3." , "-.5".

<FLOAT_NO_SIGN>:

It creates the possible unsigned float representations with the help of <INTEGER_NO_SIGN>.

<STRING>:

String literals start and end by a quotation mark. In between, any <TEXT> can come. However, <TEXT> doesn't include the symbol " to avoid any complications. Therefore, the users have to use ` instead of " inside the <STRING> if they desire.

<TEXT>:

A <TEXT> can either be empty or a character and it is defined recursively. <TEXT> is used either in comment statements or in <STRING> with reserved symbols **#** and **"'"** respectively.

3.4. Basic Symbols and Operators

The language has some specific symbols and operators used in the program definition. These are:

- Basic Symbols: **!** , **$** , **%** , **&** , **(** , **)** , ***** , **+** , **,** , **-** , **.** , **/** , **:** , **;** , **>** , **=** , **<** , **?** , ,**@** , **[** , **** , ,**]** , **^** , **_** , **`** , **{** , **}** , **~** (Note that there is also a space symbol in the <BASIC_SYMBOLS> definition between the ? and @ symbols)
- Comparison Operators: **<** , **>** , **<=** , **>=** , **==** , **!=**
- And Operator: **&&**
- Or Operator: **||**
- Not Operator: **!**
- Assignment Operator: **:=**
- New Line: **\n**

- Plus Operator: **+**
- Minus Operator: **-**
- Multiply Operator: *****
- Divide Operator: **/**
- Modulo Operator: **%**

Note that since “(“, “)”, “{“, “}”, “.”, “:”, “,” are already present in <BASIC_SYMBOLS>, we didn’t explain it separately although they are also parts of nontrivial tokens on their own.

3.5. Identifiers

Identifiers are designed similarly to identifiers in other languages like C or Java. However, an identifier has to start with a lowercase letter to distinguish it from an integer. Furthermore, in our identifiers, only numeric or alphabetic characters can follow the lowercase letter. The motivation is to make it more easy to read and understand for users although it may be considered as a restriction. Still, it increases the program readability. To define an identifier, first we defined:

- Digits: **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**
- Lowercase Letters: **a**, **b, c, d**, **e**, **f**, **g**, **h**, **ı**, **j**, **k**, **l**, **m**, **n**, **o**, **p**, **q**, **r**, **s**, **t**, **u**, **v**, **w**, **x**, **y**, **z**

- Uppercase Letters: **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**

<IDENTIFIER>:

It is started by lowercase letter and followed by <IDENTIFIER_CONTINIUE>.

<IDENTIFIER_CONTINIUE>:

It is defined recursively by replacing it either by <EMPTY> or <IDENTIFIER_CONTINIUE><ALPHANUMERIC>.

<ALPHANUMERIC>:

Combination of <DIGIT> and <ALPHABETIC>.

4. BNF Nonterminal Explanations

<program>:

This nonterminal is the start state and defines the general structure of the program in the language. The program starts with the **go** keyword followed by parentheses. It lists statements inside of curly brackets.

<statement_list>:

This nonterminal represents the list of statements separated by <NL>.

<statement>:

This nonterminal represents the types of statements in the language. These are <assignment>, <comment>, <conditional>, <function_declaration>, <expressions>, <loops>, <output> and <return>. Every statement has a <NL> at the end to mark the ending point.

<assignment>:

This nonterminal defines the assignment statement in the language. There are two types of assignment statements. For constants and non-constants. For example x := 5 assigns the integer value 5 to x. Assignment statements are followed by <expression> and its components which are explained separately below.

<expression>:

This nonterminal defines the lowest precedence expression which is either a <STRING> or a <logic_expression>. Since <STRING> is defined with the lowest precedence, language does not support expressions such as “abc” <= “bcd” and “ab” + “cd”. We designed the language in this way because expressions like “ab” || “bc” or “ab” * “bc” can actually confuse the users. To make the language less complex and more understandable, <STRING> operations are not supported.

<logic_expression>:

It is created in a way that it supports the precedence adapted by other programming languages like C. Precedence of <logical_expression> is lower than the precedences of <comparison_expression> and <math_expression>. Also the logical operators have a precedence relation among them. In the language, logical operators from the lowest to highest precedence are <OR>, <AND>, <NOT>. Therefore, <OR> is used in <logic_expression> so that it is higher in the parsing tree.

<and_expression>:

This nonterminal defines how to use <AND> in the language. It also makes sure that if there is an <AND> in the expression, it comes before the <NOT> and after the <OR> in the parse tree.

<not_expression>:

This nonterminal defines how to use <NOT> in the language. It also makes sure that if there is a <NOT> in the expression, it comes after both the <AND> and the <AND>in the parse tree. Furthermore, <BOOLEAN> can replace <not_expression> to support expressions like !TRUE or TRUE || FALSE.

<comparison_expression>:

This nonterminal defines how to use operators **<**, **<=**, **>**, **>=**, **==**, **!=** in the language. It also makes sure that if one of these operators comes, it comes before the <math_expression> in the parse tree as it has lower precedence.

<math_expression>:

The precedence of mathematical operators is important. **+** and **-** have lower precedence than *****, **/** and **%** operators. Therefore, this nonterminal defines how to use **+** and **-** in the language. It also makes sure that if there is a **+** or a **-** operator, it comes before the *****, **/** and **%** operators in the parse tree.

<multiply_divide_modulo>:

This nonterminal defines how to use *****, **/** and **%** operators in the language. It also makes sure that if there is a *****, **/** and **%** operator, it comes after the **+** and **-** operators in the parse tree to preserve the precedence.

<value>:

It is the element with the highest precedence. It defines the terms in which operators come between. A <value> can either be an <INTEGER>, a <FLOAT>, an <IDENTIFIER>, a <function_call> or <LP><expression><RP> to support expressions with parentheses.

<function_call>:

It defines how to call functions in the language. There are two types of functions which are user-defined and built-in. Both functions are called in a similar way. The name of the function which is an <IDENTIFIER> is followed by arguments separated by <COMMA> between the parentheses. If there is no argument, the function is called with empty parentheses. For example, myfunc() calls a function named myfunc with no arguments.

<input>:

It is a built-in function that uses the **input** keyword in function call followed by parentheses. It defines how to get the user input in the language.

<iot_functions>:

It defines the built-in functions that support IoT operations. <iot_read_value> reads data from sensors and timer as explained. <switch_on> and <switch_off> functions set the specified switches on and off. <define_connection> defines a connection to a given URL. <send_integer> and <recieve_integer> functions send and receive integer based on the defined connections.

<iot_read_value>:

It defines how to obtain data from either the sensors or the timer. <temperature> reads the temperature, <humidity> reads humidity, <air_pressure> reads air pressure, <air_quality> reads air quality, <light> reads the light data, <sound_level> reads the sound level and finally <time_stamp> reads the time stamp from the timer.

<switch_off>:

It is a built-in function that uses the **switchOff** keyword in function call followed by parentheses. It sets the state of the switch off. It takes <INTEGER> as a parameter to indicate the index of the

switch to be set off. The value of this <INTEGER> is verified in yacc as it has to be between 0-9.

<switch_on>:

It is a built-in function that uses the **switchOn** keyword in function call followed by parentheses. It sets the state of the switch on. It takes <INTEGER> as a parameter to indicate the index of the switch to be set on. The value of this <INTEGER> is verified in yacc as it has to be between 0-9.

<define_connection>:

It is a built-in function that uses the **defineConnectionToURL** keyword in function call followed by parentheses. It takes a <STRING> or an <IDENTIFIER> as a parameter for URL. It defines a connection to a given URL. Also, we made this function embedded in <send_integer> and <recieve_integer> methods so that users have to define a connection first before sending/receiving an integer. The motivation is to ensure that a connection is established before attempting to send/recieve an integer.

<send_integer>:

It is a built-in function that uses the **sendInteger** keyword in function call followed by parentheses. It sends an integer value at a time to a connection. Similar to <recieve_integer>, it takes

<define_connection> as a parameter. However, it also takes the desired integer value to be sent as a second parameter separated by a comma from the first one.

<recieve_integer>:

It is a built-in function that uses the **receiveInteger** keyword in function call followed by parentheses. It receives an integer value at a time from a connection. It takes <define_connection> as a parameter to ensure that the connection is established.

<temperature>:

It is a built-in function that uses the **temperature** keyword in function call followed by parentheses. It reads the temperature data from the temperature sensor.

<humidity>:

It is a built-in function that uses the **humidity** keyword in function call followed by parentheses. It reads the humidity data from the humidity sensor.

<air_pressure>:

It is a built-in function that uses the **airPressure** keyword in function call followed by parentheses. It reads the air pressure data from the air pressure sensor.

<air_quality>:

It is a built-in function that uses the **airQuality** keyword in function call followed by parentheses. It reads the air quality data from the air quality sensor.

<light>:

It is a built-in function that uses the **light** keyword in function call followed by parentheses. It reads the light data from the light sensor.

<sound_level>:

It is a built-in function that uses the **soundLevel** keyword in function call followed by parentheses. It reads the sound level data from the sound level sensor.

<time_samp>:

It is a built-in function that uses the **timeStamp** keyword in function calls followed by parentheses. It reads the number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds from the timer.

<comment>:

It defines how to create comments in the language. A <TEXT> is written between two <COMMENT_SYMBOL>

<conditional>:

This nonterminal is a container for either <matched> or <unmatched> to eliminate ambiguity in conditional statements.

<matched>:

It defines how if and else statements in the language. If statement uses the **if** keyword followed by the <logic_expression> inside parentheses. Similarly, else statements use the **else** keyword. In <matched>, an if clause is always followed by an else clause.

<unmatched>:

Unlike <matched>, it defines the if statements without an associated else. <conditional> statements are separated as <matched> and <unmatched> to determine which else belongs to which if and avoid ambiguity.

<function_declaration>:

This nonterminal defines the function declaration signature. It starts with the **function** keyword and is followed by an <IDENTIFIER>. The list of parameters (if exists) is wrapped around by parentheses and the function body is around with brackets.

<identifiers>:

This nonterminal represents the list of parameters in a function declaration. It is defined recursively so that the users can insert parameters followed by <COMMA> if they please.

<return>:

This nonterminal defines the return signature of a function. The **return** keyword can be followed by an <expression> or can be left alone, in which case nothing is returned.

<loop>:

This nonterminal is a container for two loop signatures defined in the language, which are <for_loop> and <while_loop>.

<for_loop>:

This nonterminal defines the for loop signature. It starts with the reserved keyword **for**. The <IDENTIFIER> and the <range> together are wrapped around by parentheses and the body of the loop is wrapped around by brackets. The body contains <statement_list>.

<while_loop>:

This nonterminal defines the while loop signature. It starts with the reserved keyword **while**. The condition of the loop which is <logic_expression> is wrapped around by parentheses and the

body of the loop is wrapped around by brackets. The body contains <statement_list>.

<range>:

This nonterminal defines the range statement signature. <range> has a similar functionality with the one in Python. The first one of the comma-separated <INTEGER> is the start, the second one is end, and the third one is the step. If the user chooses only one <INTEGER>, it starts from 0 and counts to that <INTEGER>. 2 <INTEGER>s count from the first one to the second one. When 3 <INTEGER>s are present, it chooses the last one as the step size and the first two operate as the second case.

<output>:

This nonterminal defines the <output> statement of the language. It starts with the **print** keyword and is followed by an <output_body>.

<output_body>:

This nonterminal defines the signature for the <output> contents, which are <expressions> separated by <COMMA>.

5. Language Criteria (Writability, Readability, Reliability):

5.1 Readability

Our language provides a Python-like syntax for enhanced readability. We use common keywords such as “for,” “if,” and “return” to increase readability since users are accustomed to them from other languages. The mathematical and logical operators in the language, such as “&&” and “||,” are the same as the popular languages like C++ to speed up the learning phase. Moreover, our language is consistent in the definition of block statements. For example, the logical statements in the definition of “while” and “if” blocks are wrapped around by parentheses, and the bodies of these blocks are wrapped around by brackets. Moreover, built-in functions provide a convenient and readable way to interact with IoT nodes. Our language also allows custom function definitions to wrap the sensible and repeated code blocks. Since our language does not provide an else-if statement, some programs might have many nested loops, which can reduce the readability.

Primitive functions contribute to the readability of our language enormously. Names IoT functions match with their purposes to make it more understandable. To exemplify, temperature() gives the temperature value from the sensors and the users can understand these built-in functions without any trouble. Also, by removing primitive data types from the

language, we make it easier to read the assignment statements for the users. However, this is a tradeoff between readability and reliability.

5.2 Writability

Top-down approach is adopted in our language to speed up the production of small programs to control IoT nodes. Thus, our language does not provide class-like structures, which are suitable for large products. Our language provides built-in functions to leverage the functionality of IoT nodes. The users can also define custom functions to avoid code-repetition and to speed up the production.

The Python-like syntax in our language enhances the coding speed. There is no strict type checking in our language, which allows users to write scripts fast as they do not have to specify the primitive data types. The for loop definition provides a fast way to create simple loops. The range function is a fast and convenient way to iterate over integers with flexibility of specifying start, stop and step values as in Python.

The custom function definitions and calls take only identifiers as arguments and do not take any primitive type arguments. This forces the user to assign the primitive type arguments to a variable first, and this reduces the readability and writability of the language. We designed the language this way to make the BNF description simpler. Finally, our

statements do not have to end with semicolons which provides an easier experience for the users.

5.3 Reliability

Our language description does not allow strict type checking. Despite the speed up of the production of small programs, this reduces the reliability of the language since there is no type-checking.

6. Resolved Conflicts and Explanations

While we were creating the Yacc file, we encountered some conflicts that needed to be resolved. Our language **does not contain any conflicts** right now. One of the conflicts was about <matched> and <unmatched> nonterminals. Although there was not any ambiguity about the conditional statements, they included shift/reduce conflicts. As a result, we expanded both <matched> and <unmatched> to cover all possible combinations of if/else statements and to get rid of any conflicts. Another conflict that needed to be solved was a shift/reduce conflict in <statement_list> on the **return** token. As we wanted to increase both writability and readability in the first project, we had not included any special tokens or keywords to actually mark where a statement ends. In most of the languages like Java or C++, a semicolon is used for this. Therefore, we faced a shift/reduce conflict

and solved it by placing <NL> at the end of each statement. As we cared about writability, we did not use a semicolon for this. So, when a statement ends, a new line comes which resolves the conflict.

7. General Description of the Language

Our language satisfies all the criteria that were required and does not contain any conflicts or ambiguities. First of all, all of the precedence rules are present in the language. A logic expression can include the <OR> operator followed by an <and_expression> and a <not_expression>. This is because the logic operators have a lower precedence than comparison or mathematical operators. Then, a <comparison_expression> comes and the comparison operators have the same precedence among themselves. Finally, <math_expression> follows from a <comparison_expression>. To preserve the precedence inside a <math_expression>, *****, **/** and **%** operators are placed lower in the parse tree as they have precedence over **+** and **-**. Naturally, parentheses expressions have the highest precedence as they are at the lowest in the parsing tree.

Furthermore, conditional statements are separated as <matched> and <unmatched> nonterminals so that no ambiguity is present. In <matched>, every if clause has a corresponding else clause whereas in <unmatched>, if clauses do not have an else clause. In the language, every <statement> ends with a <NL> to mark the ending point and eliminate shift/reduce

conflicts. A function is declared by using the **function** keyword at the beginning and for IoT functions, keywords are reserved as explained in above sections.