CS 315 Project-1

The ETALang

Team 21

21801799 *Alp Tuğrul Ağçalı*

21903488 *Emrehan Hoşver*

21803279 *Ahmet Tuna Baykal*

Table of Contents

1. Name of the Programming Language *4*

2. BNF STATEMENTS *4*

2.1 GENERAL PROGRAM STRUCTURE **4**

2.2 STATEMENTS **5**

2.2.1 Non-functional Statements 5

2.2.2 Functional Statements 7

2.3 Token List **8**

3. Language Description *10*

3.1 TYPES **10**

3.2 Program Structure **10**

3.2.1 <program> -> { <stmts> } 10

3.2.2 <stmts> -> <stmt> | <stmt> <stmts> 10

3.2.3 <stmt> -> <function_stmts> | <nonFunc_stmts> | <func_call> 10

3.2.4 <nonFunc_stmts> -> <cond_stmts> 10

3.2.5 <function_stmts> -> <primitiveFunction> 11

3.3 Condition Statements **12**

3.3.1 <if_stmt> 12

3.3.2 <else if > 12

3.3.3 <else> 12

3.3.4 <switch > , <case> 13

3.4 Loop Statements **13**

3.4.1 <for_stmt>, For Loop 13

3.4.2 <while_stmt>, While Loop 14

3.4.3 <do_stmt>, Do While Loop 14

3.5 Operation Statements **14**

3.5.1 <declaration_stmt> -> <var> <identifier> | <var> <identifier> , ... 14

3.5.2 <assign_stmt> -> <int> <identifier> = <number> 15

3.5.3 <arOp_stmt> -> <identifier> = <identifier> <arOP> <identifier> 16

3.5.3.1 <arOp> -> + | - | % | / | * | =+ | =- | =* | =/ | =% 16

3.6 Expression **16**

3.6.1 <boolOP> -> > | < | == | >= | <= | != |&& | || 17

3.6.2 <expr> -> <identifier> <bool_OP> <identifier> 17

3.7 Functional Statements **17**

3.7.1 <parameters> -> <identifier> | 17

3.7.2 Primitive Functions 18

3.7.3 Non-primitive Functions 18

4. Descriptions Nontrivial Tokens 19

5. Examination of Language 21

5.1 Readability 21

5.2 Writability 22

5.3 Reliability 22

1. Name of the Programming Language

ETALang is designed for IoT Devices.

2. BNF STATEMENTS

2.1 GENERAL PROGRAM STRUCTURE

<program> -> { <stmts> }

<stmts> -> <stmt> | <stmt> <stmts>

<stmt> -> <function_stmts> | <nonFunc_stmts> | <func_call>

<nonFunc_stmts> -> <cond_stmts>

| <loop_stmts>

| <op_stmts>

| <return>

| <comment_stmt>

<comment_stmt> -> // <alphanumeric>

<cond_stmts> -> <if_stmts> | <swCase_stmt>

<loop_stmts> -> <for_stmt> | <while_stmt> | <do_stmt>

<op_stmts> -> <decleration_stmt>

| <assign_stmt>

| <arOp_stmt>

<if_stmts> -> <if_stmt> | <if_stmt> <elses>

<elses> -> <else_if> <elses> | <else> | <empty>

<swCase_stmt> -> <switch> <caseList>

<caseList> -> <case> | <case><caseList> | <default> | <empty>

<function_stmts> -> <primitiveFunction>

|<nonPrimitiveFunction>

<primitiveFunction> -> <readDATA>

| <timeSTAMP>

<nonPrimitiveFunction> -> <sendFUNC>

| <rcvFUNC>

| <connectFUNC>

| <func_dec>

2.2 STATEMENTS

2.2.1 Non-functional Statements

2.2.1.1 Conditional Statements

<if_stmt> -> <if> (<expr>) { <stmts> }

<else if > -> <else_if> (<epxr>) { <stmts> }

<else> -> <else_> { <stmts> }

<switch > -> <_switch> (<expr>)

<case> -> <case_> (<number>) { <stmts> }

<default> -> { <stmts> }

2.2.1.2 Loop Statements

<for_stmt> -> <for> (<assign_stmt> ; <expr> ; <arOp_stmt>) { <stmts> }

<while_stmt> -> <while> (<expr>) { <stmts> }

<do_stmt> -> <do> { <stmts> } <while> (<expr>)

2.2.1.3 Operation Statements

<declaration_stmt> -> <var> <identifier> | <var> <identifier> ,

<declaration_stmt>

| <empty>

<assign_stmt> -> <int> <identifier> = <number>

|<int> <identifier> = <number> <arOp> <number>

| <string> <identifier> = <strings>

| <string> <identifier> = <strings> + <strings>

| <double> <identifier> = <doubles>

| <double> <identifier> = <doubles> <arOp> <doubles>

| <bool> <identifier> = <bools>

| <char> <identifier> = <chars>

| <string> <identifier> = <strings> + <chars>

|<string> <identifier> = <chars> + <chars>

<arOp_stmt> -> <identifier> = <identifier> <arOP> <identifier>

| <identifier> = <identifier> <arOP> <const>

| <identifier> = <const> <arOP> <const>

<arOp> -> + | - | % | / | * | =+ | =- | =* | =/ | =%

2.2.1.4 Expression

<boolOP> -> > | < | == | >= | <= | != |&& | ||

<expr> -> <identifier> <bool_OP> <identifier>

| <identifier>

|<identifier> <bool_OP> <identifier> <bool_OP> <expr>

2.2.2 Functional Statements

<parameters> -> <identifier> |

<bools> |

<identifier> , <parameters> |

<bools> , <parameters> |

<empty> |

<sensor>| <url>

2.2.2.1 Primitive Functions

<readDATA> -> <readData> (<parameters>)

<timeSTAMP> -> <timeStamp> (<parameters>)

2.2.2.2 Non-primitive Functions

<func_dec> -> <var> <identifier> (<decleration_stmt>) { <stmts> <return>}

<func_call> -> <identifier> (<parameters>)

<sendFUNC> -> <sendFunc> (<parameters>)

<rcvFUNC> -> <rcvFunc> (<parameters>)

<connectFUNC> -> <connectFunc> (<parameters>)

2.3 Token List

<var> -> <int> | <string> | <bool> | <char> | <double>

<if> -> IF

<else_if> -> ELSE_IF

<else> -> ELSE

<_switch> -> SWITCH

<case_> -> CASE

<for> -> FOR

<while> -> WHILE

<do> -> DO

<int> -> INT

<string> -> STRING

<double> -> DOUBLE

<bool> -> BOOL

<char> -> CHAR

<sensor> -> SENSOR

<switch_> -> SWITCH_

<url> -> URL

<**return> ->** RETURN

<readData> -> READDATA

<timeStamp> -> TIMESTAMP

<sendFunc> -> SENDFUNC

<rcvFunc> -> RECEIVEFUNC

<connectFunc> -> CONNECTFUNC

<strings> -> STRINGS

<doubles> -> DOUBLES

<number> -> NUMBER

<bools> -> BOOLS

<chars> -> CHARS

<number> ->[0-9]+

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

<alphabet> -> 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

<alphanumeric> -> <alphabet>{number|alphabet}

<identifier> -> IDENTIFIER

3. Language Description

3.1 TYPES

Program consists of 5 different data types. These are int, string, bool, char and double.

3.2 Program Structure

3.2.1 <program> -> { <stmts> }

Program is a list of statements between the first left bracket, which comes after the program, and the last right bracket.

3.2.2 <stmts> -> <stmt> | <stmt> <stmts>

Statements can be only one statement or may consist of a list of statements.

3.2.3 <stmt> -> <function_stmts> | <nonFunc_stmts> | <func_call>

One statement can be non function statements, function statements and function call statements.

3.2.4 <nonFunc_stmts> -> <cond_stmts>

| <loop_stmts>

| <op_stmts>

| <return>

| <comment_stmt>

Non-function statements include conditional statements, loop statements, operation statements and lastly comment statements.

3.2.4.1 <comment_stmt> -> // <alphanumeric>

Comment statements are used for writing comments. using "//" makes the used line a comment line.

3.2.4.2 <cond_stmts> -> <if_stmts> | <swCase_stmt>

Conditional statements include if-else statements and switch-case statements.

3.2.4.3 <loop_stmts> -> <for_stmt> | <while_stmt> | <do_stmt>

Loop statements consist of for loops, while loops and do-while loops.

3.2.4.4<op_stmts> -> <declaration_stmt>

| <assign_stmt>

| <arOp_stmt>

Operation statements include declaration statements, assign statements and arithmetic operation statements.

3.2.5 <function_stmts> -> <primitiveFunction>

|<nonPrimitiveFunction>

Function statements consist of two parts. One of them is primitive Functions and other one is non-primitive functions.

3.2.5.1 <primitiveFunction> -> <readDATA>

| <timeSTAMP>

Primitive functions include read Data and timeStamp functions.

3.2.5.2 <nonPrimitiveFunction> -> <sendFUNC>

| <rcvFUNC>

| <connectFUNC>

| <func_dec>

Non-Primitive functions include sendFunction, receiveFunction, connectFunction and function declaration.

3.3 Condition Statements

3.3.1 <if_stmt>

If statement is a condition statement that structurally consists of if keyword , expression and any kind of statement inside of it. Logically, if a statement accomplishes the statement that is written inside of its body, whether the expression of the if statement is valid.

3.3.2 <else if >

Else if statement is also part of condition statement that structurally consist of else if keyword, expression and followed by statement inside of its body. Else if statement is an alternative option to if statement. If if_statement's expression is not valid and else_if statement's expression is valid, then the statement which is inside of the else_if statement is done.

3.3.3 <else>

Else statement is part of the condition statement as well. Structurally consists of other keywords and statements. Logically, if if_statement's expression or else_if statement's expression is both invalid, then the else statement is done.

3.3.4 <_switch > , <case>

Switch and case is a condition statement that , if inside of the switch expression's pair is encountered in case statement's number, the statement of the <case> is done.

3.3.4.1 <default>

This is a default statement case for switch case statements , if a disagreement happens between the <_switch> and <case> statements, this default statement will be done.

3.4 Loop Statements

Basic structure of loop statements of this programming language consist of well-known loops. In BNF, loops are gathered under <loop_stmts> as below.

<loop_stmts> -> <for_stmt> | <while_stmt> | <do_stmt>

<for_stmt> represents for loop, <while_stmt> represents while loop and <do_stmt> represents do while loop.

3.4.1 <for_stmt>, For Loop

In the BNF below, we can see that the for loop consists of 5 different elements.

<for_stmt> -> <for> (<assign_stmt> ; <expr> ; <arOp_stmt>) { <stmts> }

<for> denotes that this statement is for loop. In the parentheses, we have <assign_stmt>,<expr>,<ar_op_smt> to control the for loop.

<assign_stmt> stands for creating a local temporal variable to control the for loop via <expr> and <arOp_stmt>.

<expr> is the control statement of the for loop. If the loop dissatisfies the condition, the loop will break automatically. Thus, choosing a reasonable control statement might be useful.

<arOp_stmt> is the arithmetic operation which is going to be used in each iteration of the for loop. After each iteration this operation will be executed.

<stmts> can consist of one statement or statements which will occur in each iteration of the for loop.

3.4.2 <while_stmt>, While Loop

While loop of this programming is very similar to the well known while loop across other languages and it is represented as <while_stmt> in the BNF. BNF of the while loop is given below.

<while_stmt> -> <while> (<expr>) { <stmts> }

<while>, demonstrates that this is while to the user and lexer. In the parentheses, we give a condition to the while loop, which will control the lifetime of the while loop. At the body of the while loop, which is in between left bracket and right bracket, we give a functionality to the loop to execute during its lifetime.

3.4.3 <do_stmt>, Do While Loop

Do while is very similar to the While Loop which is described above. Only difference between them is <do> at the beginning of the Do While loop. This statement makes the loop run regardless of what in the first iteration. BNF is given below.

<do_stmt> -> <do> { <stmts> } <while> (<expr>)

3.5 Operation Statements

3.5.1 <declaration_stmt> -> <var> <identifier> | <var> <identifier> ,

<declaration_stmt>

| <empty>

Declaration statement allows the user to declare a new variable or variables at the same time. By using commas different types of variables can be defined.

3.5.2 <assign_stmt> -> <int> <identifier> = <number>

|<int> <identifier> = <number> <arOp> <number>

| <string> <identifier> = <strings>

| <string> <identifier> = <strings> + <strings>

| <double> <identifier> = <doubles>

| <double> <identifier> = <doubles> <arOp> <doubles>

| <bool> <identifier> = <bools>

| <char> <identifier> = <chars>

| <string> <identifier> = <strings> + <chars>

|<string> <identifier> = <chars> + <chars>

Assignment statements consist of 10 different sub-statements. First one is assigning a number to integer variable. Another one is assigning a mathematical outcome to int variable. Same as the first one it can be assigned a string to a string variable. Another feature is it can add strings and assign it to a new string. Same as int variable, double value can be assigned to a double variable or outcome of two double can be assigned to a double variable. Also char can be assigned to a new char and addition of a char to a string can be assigned to a string. Lastly, addition of two char can be assigned to a string as well.

3.5.3 <arOp_stmt> -> <identifier> = <identifier> <arOP> <identifier>

| <identifier> = <identifier> <arOP> <const>

| <identifier> = <const> <arOP> <const>

arOp is an arithmetic operation for short. An identifier can be assigned if some operations done between two identifiers can be assigned to the identifier. In addition to this, arithmetic outcomes of two constants can be assigned to identifiers as well. Lastly arithmetic operation outcomes of identifiers and constants can be assigned to the identifier.

3.5.4 <arOp> -> + | - | % | / | * | =+ | =- | =* | =/ | =%

+ operation adds, - operation subtracts, % operation gets mod ,/ operation divides , * operation multiplies , =+ operation is self increments , =- operation is self subtracts , =* operation is self multiplies, =/ operation is self divides , =% operation is self modules. These operations are part of <arOp>.

3.6 Expression

Expression of the programming language consists of the sub header which are <boolOp> and <expr>. While <boolOp> gives the power of basic logical operators to the language, <expr> takes it and combines it with identifies. Moreover, with the recursion ability of <expr>, we can create more complex and powerful expressions for developed programs. <boolOp> is given as,

3.6.1 <boolOP> -> > | < | == | >= | <= | != |&& | ||

in the BNF. <boolOp> can refer to the functionalities such as greater, smaller, equal , greater or equal, smaller or equal, not equal, and or. Thus, we can create expressions with boolean inequalities.

BNF of <expr> is given below,

3.6.2 <expr> -> <identifier> <bool_OP> <identifier>

| <identifier>

|<identifier> <bool_OP> <identifier> <bool_OP> <expr>

3.7 Functional Statements

3.7.1 <parameters> -> <identifier> |

<bools> |

<identifier> , <parameters> |

<bools> , <parameters> |

<empty> |

<sensor>| <url> | <switch_>

Parameters are used for calling functions. They can be identifiers, booleans, sensors and urls. Additionally, one function can have more than one parameter.

3.7.2 Primitive Functions

3.7.2.1 <readDATA> -> <readData> (<parameters>)

This function is used for reading datas from sensors. This data could be temperature, humidity, air pressure, air quality, light, sound level. Function takes sensor as a parameter.

3.7.2.2 <timeSTAMP> -> <timeStamp> (<parameters>)

This function returns the time stamp from the timer. Timer keeps time since January 1, 1970.

3.7.3 Non-primitive Functions

3.7.3.1 <func_dec> -> <var> <identifier> (<decleration_stmt>) { <stmts> }

This is a function definition statement that takes a variable which is going to be returned at the end of the function and takes an identifier that is going to be the name of the function. Inside of the parentheses a declaration statement is desired if the function needs a parameter, and lastly inside of the body there is a statement to be done.

3.7.3.2 <func_call> -> <identifier> (<parameters>)

This is a function call that takes the name of the existing function call as identifier and if it has any parameter it takes the parameters as well in the parentheses.

3.7.3.3 <sendFUNC> -> <sendFunc> (<parameters>)

This function sends an integer value at a time. It may take zero or infinite number of desired parameter.

3.7.3.4 <rcvFUNC> -> <rcvFunc> (<parameters>)

This function receives an integer value at a time. It may take zero or infinite number of desired parameter.

3.7.3.5 <connectFUNC> -> <connectFunc> (<parameters>)

Connect Function does connect to a URL and also takes the URL as a parameter.

4. Descriptions Nontrivial Tokens

<var> -> <int> | <string> | <bool> | <char> | <double>

Var means variable type. int represents integers, string represent strings, bool represents booleans, char represent characters and double represent doubles.

<if> -> IF

it's syntax if(expression) { codeblock }

<else_if> -> ELSE_IF

it's syntax else if (expression) { codeblock }

<else> -> ELSE

it’s syntax else { codeblock }

<_switch> -> SWITCH

it’s syntax switch (expression)

<case_> -> CASE

it’s syntax case x: ,where x is an integer.

<for> -> FOR

its syntax is for (int i = 0; i < x; i++) { codeblock } where x is an integer.

<while> -> WHILE

its syntax is while (i < x) { codeblock } where x is an integer.

<do> -> DO

its syntax is do { codeblock } while(i < x) where x is an integer.

<sensor> -> SENSOR

Sensor is the parameter of readData function. The number of sensors is indicated using its name. For example SENSOR1 to SENSOR6

<switch_> -> SWITCH_

Switch is the parameter of readData function. The number of switches is indicated using its name. For example SWITCH_0 to SWITCH_9

<url> -> URL

URL is the parameter of the connect function. URL can include both digits and letters.

<**return> ->** RETURN

Return statement is used for returning the value from function.

<strings> -> STRINGS

Strings must be in quotes. For example, “exampleString”.

<doubles> -> DOUBLES

Doubles must be divided using dots. For example, 3.15.

<number> -> NUMBER

Numbers are representations of integers.

<bools> -> BOOLS

Bools can be either true or false.

<chars> -> CHARS

Chars must be in apostrophe. For example, ‘exampleChar’.

<identifier> -> IDENTIFIER

Identifiers are names of the variables.

5. Examination of Language

5.1 Readability

Readability of this language may seem better for people when it is compared with other programming languages due its combined nature. Basically, new lines control each line as Python style. However, in the loops and function blocks it may be confusing for reading code by line separation. Thus, we added block statements for such cases. Where things get complicated you can define a block via functions or loops or condition statements and continue under it line by line through the block. Which can create clearer vision for the reader and help his/her to understand the code faster. Also, a user must define the type of each variable that he/she created. Accordingly, readers will grasp the concept better. Also, we have SENSOR data type since the programming language designed for the IoT department, it will be easier to understand and distinguish that data type from others.

5.2 Writability

This programming language is designed to be easily written. First of all, any of the lines does not require to be end with a semicolon or any other symbol. However, Left and Right brackets are used to ease the understanding of the indentation while programmers are writing code. Also, primitive functions are easy to understand and memorizable so that while coding those functions it is hard to mis-write. Also programmers are allowed to use any name for their variables or function names except for starting with number and symbol, other than these they are fully motivated to write as they will.

5.3 Reliability

This programming language is reliable in most of the cases. Due to the style of variables, the user must define the type of each variable that is created and accordingly, the user will be aware of the types and can create statements which will not throw any type related error during the program. Moreover, as stated above the programming language has a block statement for function declaration and loop statements. Hence, anything written in these blocks will be directly related to loop or function itself. As a result, there will be no problem if the user writes code under the line and blocks obligations and defines the variables in a sense. However, we encourage the users to not write long lines of code. It is better practice to divide the code in to small chunks and write them into different lines in a reasonable order.