CS315 PROGRAMMING LANGUAGES PROJECT-1

IsOT PROGRAMMING LANGUAGE

2022-23 Fall

Group Members

Mahmut Furkan Gön Section-3 21902582

Ahmet Arda Ceylan Section-3 22003148

Omar Hamdash Section-3 22001398

Complete BNF Description of IsOT Language

<program> ::= **start**<stmts>**end**

<stmts> ::= <stmt> | <stmt><stmts>

<stmt> ::= <if_stmt> | <non-if_stmt>

<if_stmt> ::= <matched_stmt> | <unmatched_stmt>

<non-if_stmt> ::= <assign_stmt><end_stmt>
| <input_stmt><end_stmt>
| <output_stmt><end_stmt>
| <func_define>
|<func_call><end_stmt>
| <read_from_sensor><end_stmt>
| <time_from_timer><end_stmt>
| <connect_stmt><end_stmt>
| <declare_stmt><end_stmt>
| <break_stmt><end_stmt>
| <continue_stmt><end_stmt>
| <arithmetic_op><end_stmt>
| <comment>
| <end_stmt>
| <while_stmt>
| <for_stmt>

1. Function Definition and Function Call

<func_define> ::= **func**<space><func_var><LP><parameters><RP><LB><func_body><RB>
<parameters> ::= <parameter> | <parameter><comma><parameters>
<parameter> ::= **int**<space><int_var>
| **float**<space><float_var>
| **string**<space><string_var>
| **bool**<space><bool_var>
| **char**<space><char_var>
| **status**<space><status_var>
| **protocol**<space><protocol_var>

<func_body> ::= <return_stmt> | <stmts><return_stmt>
<func_call> ::= <func_var><LP><variables><RP>
<variables> ::= <var> | <var><comma><variables>
<var_types>::=**int | float | string | char | bool | status | protocol**

2. Continue-Break, Return, and End Statements

<continue_stmt> ::= **continue**
<break_stmt> ::= **break**
<return_stmt> ::= **return**
<values> ::= <bool> | <string> | <char> | <int> | <float>
<end_stmt> ::= <semicolon>

3. Comment

<comment> ::= <hashtag><sentence><newline>

4. Arithmetic Operations

<arithmetic_op>::= <numeric_value>
| <arithmetic_op><plus_op><numeric_value>
| <arithmetic_op><minus_op><numeric_value>

<numeric_value>::= <factor> | <numeric_value><mult_op> <factor> |
<numeric_value><divide_op><factor>
<factor> ::= <int> | <float> | <int_var> | <float_var> | <LP><arithmetic_op><RP>

5. Data Types

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

<float> ::= <digits><dot><digits>
| <minus_op><digits><dot><digits>
| <dot><digits>
| <minus_op><dot><digits>

<letter>::= <uppercase_letter> | <lowercase_letter>
<uppercase_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
<lowercase_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

6. Input Output

<input_stmt> ::= **input**<LP><inbody><RP>
<input_from_connection> ::= <connect_var><dot>**input_from_connection**<LP><inbody><RP>
<output_stmt> ::= **print**<LP><outbody><RP>

<output_to_connection> ::= <connect_var><dot>**output_to_connection**<LP><outbody><RP>
<read_from_sensor> ::= **read_from_sensor**<LP><sensor_name><comma><inbody><RP>
<sensor_name>::= <identifier>
<time_from_timer> ::= **time_from_timer**<LP><int_var><RP>

<inbody> ::= <var>

<outbody> ::= <arithmetic_operation> | <values> | <var>

<URL> ::= <string>

7. Creating a Connection Object and All Related Statements

<connect_obj_creation> ::= Connection<space><connect_var><assign_op>
new<space>**Connection**<LP><RP>

<connect_stmt> ::= <connect_var><dot>**connect**<LP><URL><RP>

<get_connect_status> ::= <connect_var><dot>**getStatus**<LP><status_var><RP>

<get_connect_protocol> ::= <connect_var><dot>**getProtocol**<LP><protocol_var><RP>

<get_connect_URL> ::= <connect_var><dot>**getURL**<LP><string_var><RP>

<connect_var> ::= <identifier>

<status_var> ::= <identifier>

<protocol_var> ::= <identifier>

<status_value> ::= **connected** | **connecting** | **disconnected** | **host_not_found** |
connection_timeout

<protocol_value> ::= **http** | **https** | **tcp** | **ftp** | **tftp**

8. Conditional Statements: If-Else

<matched_stmt> ::=
if<LP><logic_expr><RP><LB><matched_stmt><RB>**else**<LB><>matched_stmt> <RB>|
<non-if_stmt>

<unmatched_stmt> ::= **if**<LP><logic_expr><RP><LB><stmts><RB>
|**if**<LP><logic_expr><RP><LB> <matched_stmt><RB>**else** <LB><unmatched_stmt><RB>

9. Loops: While and For

<while_stmt>::= **while** <LP> <logic_expr> <RP> <LB><stmts><RB>

<for_stmt>::= **for** <LP> <declare_stmt><end_stmt> <logic_expr>
<end_stmt><assign_stmt><RP> <LB><stmts><RB>

10. Logic Expressions

<logic_expr>::= <bool_value> | <logic_operation>| <comparison_operation>

<logic_operation>::= <logic_expr><logic_op><logic_expr>

<logic_op>::= <and_op> | <or_op>

<comparable>::= <num_var> | <num_value>

<comparison_operation>::= <comparable><comparison_op><comparable>

<comparison_op>::= <less_op> | <greater_op> | <equivalence_op> | <greatEq_op>
| <lessEq_op> | <not_eq_op>

11. Operators

<plus_op>::= +
<minus_op>::= -
<mult_op>::= *
<divide_op>::= /
<expo_op>::= ^

<assign_op>::= =

<and_op>::= &
<or_op>::= |

<less_op>::= <
<greater_op>::= >
<equivalence_op>::= ==
<greatEq_op>::= >=
<lessEq_op>::= <=
<not_eq_op>::= !=

12. Declaration Statements

<declare_stmt>::= <int_declare>
| <str_declare>
| <char_declare>
| <float_declare>
| <bool_declare>
| <status_declare>
|<protocol_declare>
<int_declare>::= **int**<space><int_var> | **int**<space><int_assign>
<float_declare>::= **float**<space><float_var> | **float**<space><float_assign>
<str_declare>::= **string**<space><str_var> | **string**<space><str_assign>
<char_declare>::= **char**<space><char_var> | **char**<space><char_assign>
<bool_declare>::= **bool**<space><bool_var> | **bool**<space><logic_assign>
<status_declare>::= **status**<space><status_var> | **status**<space><status_assign>
<protocol_declare>::= **protocol**<space><protocol_var> | **protocol**<space><protocol_assign>

13. Assignment Statements

<assign_stmt>::= <int_assign>
| <float_assign>
| <str_assign>
| <char_assign>
| <bool_assign>
| <status_assign>
| <protocol_assign>

<int_assign>::= <int_var><assign_op><num_var>
| <int_var><assign_op><num_value>
<float_assign>::= <float_var><assign_op><num_var>
| <float_var><assign_op><num_value>
<str_assign>::= <str_var><assign_op><string>
<char_assign>::= <char_var><assign_op><char>
<bool_assign>::= <bool_var><assign_op><bool_value>
| <bool_var><assign_op><logic_expr>
<status_assign>::= <status_var>=<status_var>
| <status_var><assign_op><status_value>
<protocol_assign>::= <protocol_var><assign_op><protocol_var>
| <protocol_var><assign_op><protocol_value>

<num_var>::= <int_var> | <float_var>
<num_value>::= <int> | <float> | <arithmetic_operation> | <alphanumeric>
<bool_value>::= **true** | **false** | **yes** | **no** | **on** | **off**

<var>::= <string_var>
| <char_var>
| <int_var>
| <float_var>
| <bool_var>
| <func_var>
| <status_var>
| <protocol_var>

<string>::= <string_id><identifier><string_id>
<char>::= <char_id><alphanumeric><char_id>
<identifier>::= <char_list>
<char_list>::= <alphanumeric> | <alphanumeric><char_list>
<alphanumeric>::= <letter> | <digit>
<char_var>::= <letter> | <letter><identifier>
<string_var>::= <letter> | <letter><identifier>
<int_var>::= <letter> | <letter><identifier>
<float_var>::= <letter> | <letter><identifier>
<bool_var>::= <letter> | <letter><identifier>
<func_var>::= <letter> | <letter><identifier>
<status_var>::= <letter> | <letter><identifier>
<protocol_var>::= <letter> | <letter><identifier>

Explanation of the Language Constructs

1. Program

<program> ::= **start**<stmts>**end**

This non-terminal is our whole program. Our program starts with the “start” command and ends with the “end” command. Other statements come between these two comments.

2. Statements

<stmts> ::= <stmt> | <stmt><stmts>

This non-terminal states that all of the statements in the program consist of either a statement or a list of statements that are described recursively.

3. Statement

<stmt> ::= <if_stmt> | <non-if_stmt>

This non-terminal states that a statement can either be an if or a non-if statement. This method is applied to get rid of ambiguity occurring in else-less if statements.

4. If Statement

<if_stmt> ::= <matched_stmt> | <unmatched_stmt>

This non-terminal says that an if statement is either a matched if statement or an unmatched if statement.

5. Matched Statement

<matched_stmt> ::=
if<LP><logic_expr><RP><LB><matched_stmt><RB>**else**<LB><>matched_stmt> <RB>| <non-if_stmt>

This non-terminal says that a matched statement can be either a matched if statement or any non-if statement. A matched if statement contains if reserved word, and logic expressions between parentheses. A matched statement between braces follows this. After them, else reserved word comes, and it is followed by a matched statement between braces.

6. Unmatched Statement

<unmatched_stmt> ::= **if**<LP><logic_expr><RP><LB><stmts><RB>
|**if**<LP><logic_expr><RP><LB> <matched_stmt><RB>**else**
<LB><unmatched_stmt><RB>

This non-terminal states that an unmatched statement is either an else-less if or an if with else which contains an unmatched statement. An else-less if contains if reserved word followed by logic expression inside the parentheses and statements between braces follow these.
An if with else which contains an unmatched statement is similar to a matched if statement. However, it contains an unmatched statement between braces after else reserved word.

7. Non-if Statement

<non-if_stmt> ::= <assign_stmt><end_stmt>

| <input_stmt><end_stmt> | <output_stmt><end_stmt>

| <func_define> |<func_call><end_stmt>

| <read_from_sensor><end_stmt>

| <time_from_timer><end_stmt>

| <connect_stmt><end_stmt> | <declare_stmt><end_stmt>

| <break_stmt><end_stmt> | <continue_stmt><end_stmt>

| <arithmetic_op><end_stmt> | <comment> | <end_stmt>

| <while_stmt> | <for_stmt>

This non-terminal states that, non-if statements consist of several statements. It is either one of the followings:

- <assign_stmt><end_stmt>
- <input_stmt><end_stmt>
- <output_stmt><end_stmt>
- <func_define>
- <func_call><end_stmt>
- <read_from_sensor><end_stmt>
- <time_from_timer><end_stmt>
- <connect_stmt><end_stmt>
- <declare_stmt><end_stmt>
- <arithmetic_op><end_stmt>
- <break_stmt><end_stmt>
- <continue_stmt><end_stmt>
- <comment>
- <end_stmt>
- <for_stmt>
- <while_stmt>

8. While Statement

<while_stmt>::= **while** <LP> <logic_expr> <RP> <LB><stmts><RB>

This non-terminal states the syntax of the while statement. It contains while reserved word followed by logic expression inside parentheses. Statements inside the braces follow this.

9. For Statement

<for_stmt>::= **for** <LP> <declare_stmt><end_stmt> <logic_expr> <end_stmt><assign_stmt><RP> <LB><stmts><RB>

This non-terminal states for loops' syntax in our language. A for loop contains for reserved word. Then, in the parentheses a declare statement, logic expression, and assignment statement which are separated by end statements follow. Statements inside the braces follow these.

10. Logic Expressions

<logic_expr>::= <bool_value> | <logic_operation>| <comparison_operation>

This non-terminal states that logic expression can be either a boolean value or a logic operation or a comparison operation.

<logic_operation>::= <logic_expr><logic_op><logic_expr>

This non-terminal states the logic operations' syntax. It contains a logic operator between two logic expressions.

<logic_op>::= <and_op> | <or_op>

This non-terminal states the logic operators. In our language there are two logic operators: and operator and or operator.

<comparison_operation>::= <comparable><comparison_op><comparable>

This non-terminal states what comparison operation is. Comparison operation is comparison of two comparables.

<comparable>::= <num_var> | <num_value>

This non terminal defines comparables. They are either number values or number variables which are float and integers.

<comparison_op>::= <less_op> | <greater_op> | <equivalence_op> | <greatEq_op> | <lessEq_op> | <not_eq_op>

This non-terminal states what the comparison operators are. In our language, comparison operators are less than, less than or equal, equivalent, greater than, greater than or equal to and not equal operators.

11. End Statement

<end_stmt> ::= <semicolon>

This non-terminal says that an end statement is a semicolon.

12. Comment

<comment> ::= <hashtag><sentence><newline>

This non-terminal states the syntax of the comment. Comments start with a hashtag symbol (#) and end with a newline character. Inside these two, it counted as sentence and isn't implemented as comments

13. Continue Statement

<continue_stmt> ::= **continue**

This terminal states that continue statement is fulfilled by the reserved word continue. It is used to break one iteration in the loops.

14. Break Statement

<break_stmt> ::= **break**

This terminal states that break statement is fulfilled by the reserved word break. It is used in loops to terminate the loop.

15. Data Types

<int> ::= <digits> | <minus_op><digits>

This non-terminal defines that integers consist either of digits or of minus operation followed by digits.

<digits> ::= <digit> | <digit><digits>

This non-terminal statements define digits as either one digit or a list of digits.

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

This terminal defines digit as numbers from 0 to 9.

<float> ::= <digits><dot><digits>

| <minus_op><digits><dot><digits>

| <dot><digits>

| <minus_op><dot><digits>

This non-terminal defines floats that they can either be digits (digits defined above) a dot followed by another digits, with an optional minus sign before it, or they can just a dot followed by digits, with an optional minus sign before that.

<letter>::= <uppercase_letter> | <lowercase_letter>

This non-terminal defines that letter is either an uppercase letter or a lowercase letter.

<uppercase_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

This terminal defines uppercase letters that are letter from A to Z.

<lowercase_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

This terminal defines lowercase letters that are letter from a to z.

16. Arithmetic Operations

<arithmetic_op>::= <numeric_value>

| <arithmetic_op><plus_op><numeric_value>

| <arithmetic_op><minus_op><space><numeric_value>

This non-terminal states that an arithmetic operation can be a numeric value or add or subtract operations(There must be a space between minus operator and the numeric value in our language. Spaces in other operations are optional). Arithmetic operations designed in a way that respects operation precedence.

<numeric_value>::= <factor> | <numeric_value><mult_op> <factor> | <numeric_value><divide_op><factor>

This non-terminal states that numeric value can be a factor or division or multiplication operation. Numeric values designed in a way that respects operation precedence.

<factor> ::= <int> | <float> | <int_var> | <float_var> | <LP><arithmetic_op><RP>

This non-terminal states that a factor can either be an integer value or an integer variable or a float value or a float variable or an arithmetic operation in parentheses.

17. Function Definition and Function Call

<func_define> ::= func<space><func_var><LP><parameters><RP><LB><func_body><RB>

This non-terminal defines the form of the function definition of the language. Function definition starts with “func” reserved word fallowed by space and function variable then, function parameters between two parentheses. Finally, followed by function body between braces.

<parameters> ::= <parameter> | <parameter><comma><parameters>

Parameters are defined as parameter or parameter followed by comma and parameters to include multiple amount of parameters.

<parameter> ::= **int**<space><int_var>

| **float**<space><float_var>

| **string**<space><string_var>

| **bool**<space><bool_var>

| **char**<space><char_var>

| **status**<space><status_var>

| **protocol**<space><protocol_var>

This non-terminal is defined as variable type followed by the variable that is specific to the defined variable type.

<func_body> ::= <return_stmt> | <stmts><return_stmt>

This non-terminal body is defined as a return statement or statements followed by a return statement. All the functions must have a return statement to terminate the function.

<func_call> ::= <func_var><LP><variables><RP>

This non-terminal is defined as a function variable followed by variables between parentheses.

<variables> ::= <var> | <var><comma><variables>

This non-terminal is defined as variable or variable followed by comma and variables to have multiple variables .

<var_types>::=**int | float | string | char | bool | status | protocol**

This terminal is defined as one of the int, float, string, char, bool, status, protocol key words.

18. Assign Statements

<assign_stmt>::= <int_assign>

| <float_assign>

| <str_assign>

| <char_assign>

| <bool_assign>

| <status_assign>

| <protocol_assign>

This non-terminal states that there are six different types of assignment. These are integer, string, character, float, boolean, status and protocol assignments.

<int_assign>::= <int_var><assign_op><num_var>

| <int_var><assign_op><num_value>

This non-terminal says that integer assignment is that integer variable name immediately followed by the assign operator, which is = . After that, there can be any numerical value (either integer or float.)

<float_assign>::= <float_var><assign_op><num_var>

| <float_var><assign_op><num_value>

This non-terminal says that float assignment is that float variable name immediately followed by the assign operator, which is = . After that, there can be any numerical value (either integer or float.)

<str_assign>::= <str_var><assign_op><string>

This non-terminal says that string assignment is that string variable name immediately followed by the assign operator, which is = . After that, there comes a string value.

<char_assign>::= <char_var><assign_op><char>

This non-terminal says that character assignment is that character variable name immediately followed by the assign operator, which is = . After that, there comes a character value.

<bool_assign>::= <bool_var><assign_op><bool_value>

| <bool_var><assign_op><logic_expr>

This non-terminal says that boolean assignment is that boolean variable name immediately followed by the assign operator, which is = . After that, there can be either a boolean value or a logic expression.

<status_assign>::= <status_var>=<status_var>

| <status_var><assign_op><status_value>

This non-terminal says that status assignment is that status variable name immediately followed by the assign operator, which is = . After that, there can be either a status value or another status variable name.

<protocol_assign>::= <protocol_var><assign_op><protocol_var>

| <protocol_var><assign_op><protocol_value>

This non-terminal says that protocol assignment is that protocol variable name immediately followed by the assign operator, which is = . After that, there can be either a protocol value or another protocol variable name.

<num_var>::= <int_var> | <float_var>

This non-terminal states that a numeric variable name is either an integer variable name or a float variable name.

<num_value>::= <int> | <float> | <arithmetic_operation> | <char>

This non-terminal states that a numeric value can be an integer value, a float value, an arithmetic operation or a character.

<bool_value>::= **true** | **false** | **yes** | **no** | **on** | **off**

This terminal states that a boolean can have values true, yes, on, false, no and off. The first three corresponds 1 and the rest corresponds 0.

<var>::= <string_var>

| <char_var>

| <int_var>

| <float_var>

| <bool_var>

| <func_var>

| <status_var>

| <protocol_var>

This non-terminal states that there are six different types of variable names. These are integer, string, character, float, boolean, status and protocol variable names.

<string>::= <string_id><identifier><string_id>

This non-terminal states that a string corresponds to an identifier consists of the same characters between double-quotes.

<char>::= <char_id><alphanumeric><char_id>

This non-terminal states that a character corresponds to an alphanumeric consists of the same characters between single-quotes.

<identifier>::= <char_list>

This non-terminal says that identifier is a charlist.

<char_list>::= <char> | <char><char_list>

This non-terminal states that a charlist can be a character or a list of characters.

<char>::= <letter> | <digit>

This non-terminal states that a character is a letter or a digit.

<char_var>::= <letter> | <letter><identifier>

This non terminal states that a character variable's name (variable name) is either a letter or a letter followed by an identifier.

<string_var>::= <letter> | <letter><identifier>

This non terminal states that a string variable's name (variable name) is either a letter or a letter followed by an identifier.

<int_var>::= <letter> | <letter><identifier>

This non terminal states that a integer variable's name (variable name) is either a letter or a letter followed by an identifier.

<float_var>::= <letter> | <letter><identifier>

This non terminal states that a float variable's name (variable name) is either a letter or a letter followed by an identifier.

<bool_var>::= <letter> | <letter><identifier>

This non terminal states that a boolean variable's name (variable name) is either a letter or a letter followed by an identifier.

<func_var>::= <letter> | <letter><identifier>

This non terminal states that a function's name (function name) is either a letter or a letter followed by an identifier.

<status_var>::= <letter> | <letter><identifier>

This non terminal states that a status variable's name (variable name) is either a letter or a letter followed by an identifier.

<protocol_var>::= <letter> | <letter><identifier>

This non terminal states that a protocol variable's name (variable name) is either a letter or a letter followed by an identifier.

19. Declaration Statement

<declare_stmt>::= <int_declare>

| <str_declare>

| <char_declare>

| <float_declare>

| <bool_declare>

| <status_declare>

|<protocol_declare>

This non-terminal states that there are six different types of declaration. These are integer, string, character, float, boolean, status and protocol declarations.

<int_declare>::= **int**<space><int_var> | **int**<space><int_assign>

This non-terminal says that integer declaration is that int reserved word immediately followed by a space. After that, there can be an integer variable name or integer assign statement.

<float_declare>::= **float**<space><float_var> | **float**<space><float_assign>

This non-terminal says that float declaration is that float reserved word immediately followed by a space. After that, there can be a float variable name or float assign statement.

<str_declare>::= **string**<space><str_var> | **string**<space><str_assign>

This non-terminal says that string declaration is that string reserved word immediately followed by a space. After that, there can be a string variable name or string assign statement.

<char_declare>::= **char**<space><char_var> | **char**<space><char_assign>

This non-terminal says that character declaration is that char reserved word immediately followed by a space. After that, there can be a character variable name or character assign statement.

<bool_declare>::= **bool**<space><bool_var> | **bool**<space><logic_assign>

This non-terminal says that boolean declaration is that bool reserved word immediately followed by a space. After that, there can be a boolean variable name or boolean assign statement.

<status_declare>::= **status**<space><status_var> | **status**<space><status_assign>

This non-terminal says that status declaration is that status reserved word immediately followed by a space. After that, there can be a status variable name or status assign statement.

<protocol_declare>::= **protocol**<space><protocol_var> | **protocol**<space><protocol_assign>

This non-terminal says that protocol declaration is that protocol reserved word immediately followed by a space. After that, there can be a protocol variable name or protocol assign statement.

20. Input Output

<input_stmt> ::= **input**<LP><inbody><RP>

This non-terminal is defined as input reserved word followed by inputbody between two parentheses. It will be used to get an input from the standard input device(i.e. keyboard). Inputbody is defined as a variable therefore input given will be assigned to the variable automatically and in this way it will be easy to keep track of data type of the input.

<input_from_connection> ::= <connect_var><dot>**input_from_connection**<LP><inbody><RP>

This non-terminal is defined as connection variable followed by a dot, input_from_conection reserved word and inputbody between two parentheses. The inbody can be any variable and the data received from the connection is stored into this variable. The connection variable is a connection object (more details about it later).

<output_stmt> ::= **print**<LP><outbody><RP>

This non-terminal is defined as “print” reserved word followed by outputbody between two parentheses. It will be used to transfer information to standard output device(i.e. screen). outbody is defined later.

<output_to_connection> ::=
<connect_var><dot>**output_to_connection**<LP><outbody><RP>

This non-terminal is defined as output_to_connection followed by left parenthesis, URL, comma, output body and right parenthesis. This allows to send data to a connection, for example a command line connection can take linux commands using this function. It is called on a connection object.

<read_from_sensor> ::=
read_from_sensor<LP><sensor_name><comma><inbody><RP>

This non-terminal is defined as read_from_sensor followed by left parenthesis then the sensor name then a comma then the input body that the we get from the sensor. This allows to read values from different sensors according to the sensor type. For example if the sensor name provided was a name of a temperature sensor then it will return in input body the temperature, but in this case <inbody> should be a float variable because temp sensor returns float values.

<sensor_name>::= <identifier>

Sensor name is defined as an identifier.

<time_from_timer> ::= **time_from_timer**<LP><int_var><RP>

This non-terminal is defined as “time_from_timer” reserved keyword followed by integer variable between parentheses. It takes the timestamp from the timer and puts it into the the integer variable <int_var>. The timestamps taken from the timer is the number of seconds elapsed since midnight Coordinated Universal Time (UTC) of January 1, 1970, not counting leap seconds.

<inbody> ::= <var>

Inputbody is defined as a variable. The programmer has to use the appropriate input body variable type depending on the situation.

<outbody> ::= <arithmetic_operation> | <values> | <var>

This non-terminal is defined as to be able to send different variables and data types. It is used when needing to give input to some functions. Again the programmer has to use the correct data type based on the situation.

<URL> ::= <string>

This non-terminal defines URL as string.

21. Creating a Connection Object and All Related Statements

<connect_obj_creation> ::=
Connection<space><connect_var><assign_op>
new<space>**Connection**<LP><RP>

This non-terminal is defined as Connection class name followed by a space followed by the connection variable name followed by the assignment operator followed by the new keyword followed by a space followed by the Connection() constructor. This allows to initialize connection objects. The a connection class has multiple useful functions that can be used to manipulate connections and do some basic operations. IsOT's approach for connection handling is by creating a connection object and data is stored in parameters. The data can be accessed using getter functions.

<connect_stmt> ::= <connect_var><dot>**connect**<LP><URL><RP>

This non-terminal is defined as connection variable followed by a dot followed by the connect keyword followed by a left parathesis followed by the URL followed by a right parathesis. This is one of the Connection class functions that allows to establish a connection by taking a URL as the address.

<get_connect_status> ::=
<connect_var><dot>**getStatus**<LP><status_var><RP>

This non-terminal is defined as a connection variable followed by a dot followed by getStatus followed by a left parenthesis followed by a status variable followed by a right parenthesis. This Connection function puts the connection status into a status variable. status is a data type in IsOT, making it easier and simpler to deal with connections as this an IoT oriented programming language. status type can have specified values predefined by the language and are reserved keywords.

<get_connect_protocol> ::=
<connect_var><dot>**getProtocol**<LP><protocol_var><RP>

This non-terminal is defined as a connection variable followed by a dot followed by getProtocol followed by a left parenthesis followed by a protocol variable followed by a right parenthesis. Exactly as status data type and variables, protocol variable has predefined values and is useful for connections used for IoT devices.

<get_connect_URL> ::=
<connect_var><dot>**getURL**<LP><string_var><RP>

This non-terminal is defined as a connection variable followed by a dot followed by getURL followed by a left parenthesis followed by a string variable followed by a right parenthesis. This function puts the URL this connection is connected to to a string variable provided between parenthesis.

<connect_var> ::= <identifier>

Connection variable is defined as an identifier

<status_var> ::= <identifier>

Status variable is defined as an identifier

<protocol_var> ::= <identifier>

Protocol variable is defined as an identifier.

<status_value> ::= **connected** | **connecting** | **disconnected** | **host_not_found** | **connection_timeout**

Status value is defined as these reserved values.

<protocol_value> ::= **http** | **https** | **tcp** | **ftp** | **tftp**

Protocol value is defined as these reserved values.

Non-Trivial Tokens

start Starts the program in our language.
end Finishes the program in our language.
if Used for the if statements in our language.
else Used for the else statements in our language.
while Used in the while statements in our language.
for Used in the for statements in our language.
break Used for break statements in our language.
return Used for return statements in our language.
continue Used for continue statements in our language.
print Used while printing outputs.
input Used while getting inputs.
input_from_connection Used as function name which is used while getting input.

output_to_connection Used as function name which is used while getting output.

int We have int datatype in our language.
bool We have bool datatype in our language.
float We have float datatype in our language.
string We have string datatype in our language.
char We have char datatype in our language.
status We have status datatype in our language.
protocol We have protocol datatype in our language.
func We have func reserved word in our language to define a function.
true It is the boolean value true in our language.
False It ıs the boolean value false in our language.
yes It is the same boolean value as true in our language.
no It is the same boolean value as false in our language.
on It is the same boolean value as true in our language.
off It is the same boolean value as falsein our language.

Readability, Writability, Reliability

IsOT language adapts common naming conventions that exist in general-purpose programming languages such as Java, C++, Python with enhancements in IoT specific functions, datatypes and other required mechanisms. It has simple and easy-to-use and easy-to-understand syntax. IsOT language has mechanisms that don't exist in common other languages. It has no ambiguity. IsOT doesn't have ambiguous definitions. Programmers can use IsOT since it is comprehensive and it makes complex programs, subprograms simple. Security is an important point in IoT development as people entrust valuable things to robots. Since IsOT makes jobs easier and simpler without any ambiguity, IsOT would make programmers job a lot easier.