CS315 Project 2

Iotech

Group 40

Deniz Mert Dilaverler	22003530	Section 02
Ceren Akyar	22003158	Section 02

1. Complete BNF Description

```
<program> ::= <stmt_list> | <stmt_list> <func_list>
<func_list> ::= <func> | <func> | <func_list>
<stmt_list> ::= <stmt> | <stmt> <stmt_list>
<func> ::= func  <void_func> | func <non_void_func>

<void_func> ::= void <signature> { <stmt_list> }
<non_void_func> ::= <type> <signature> { <stmt_list> <return> }
<return> ::= return <expression>;

<signature> ::= <identifier> ( <in_args>) | <identifier> ()
<in_args> ::= <type> <identifier> | <type> <identifier>, <in_args>

<func_call> ::=         <identifier> () | <identifier> ( <args> )
                        |<non_void_primitive_func_call> | <void_primitive_func_call>
<void_primitive_func_call> ::= SetSwitch( <bool_literal >, <digit > )
                               | SetSwitch( <identifier >, <digit > )


<non_void_primitive_func_call> ::= ReadTimeStamp() | ReadTemperature() |
                                       ReadHumidity() | ReadAirPress() |
                                       ReadAirQual() | ReadLight() |
                                       ReadSoundLevel( <integer_literal> )
                                   |   ReadSoundLevel(  <identifier> )
<args> ::= <arg> | <arg>, <args>
<arg> ::= <expression>
<stmt> ::=<block_stmt> | <non_block_stmt>

<block_stmt> ::= <if_stmt> | <while_stmt> | <for_stmt> | <do_while_stmt>
<non_block_stmt> ::= <assign_stmt>; | <declaration_stmt>; | <func_call>; |
                         <one_line_comment> | <multi_line_comment> | <connect_URL>; |
                         <send>; | <terminate_connection>;


<if_stmt> ::=if  (<logic_exprs>) { <stmt_list> } |
                 if  (<logic_exprs>) { <stmt_list> } else { <stmt_list> }
```

```
                     if (<logic_exprs>) { <stmt_list> } <elseif_stmts> else { <stmt_list> } |
                     if ( <logic_exprs) { <stmt_list> } <elseif_stmts>
<elseif_stmts> ::= else if (<logic_exprs>) { <stmt_list> } |
                      else if (<logic_exprs>) { <stmt_list> } <elseif_stmts>
<while_stmt> ::=while (<logic_exprs>) { <stmt_list> }
<for_stmt> ::=for (<for_init> ;  <logic_exprs>;  <assign_stmt> ) { <stmt_list> }
<for_init> ::=<decleration_stmt> | <assign_stmt>
<do_while_stmt> ::=do { <stmt_list> } while (<logic_exprs>);

<URL> ::= <string_value>
<connect_URL> ::= connection <connect_obj>(<URL>)
<connect_obj> ::= $<identifier>
<send> ::= <connect_obj>.send(<integer_value>)
<receive> ::= <connect_obj>.receive()
<terminate_connection> ::= <connect_obj>.terminate()
<declaration_stmt> ::= <type> <identifier> | <type> <assign_stmt>
                              | const <type> <assign_stmt>
<assign_stmt> ::= <identifier> =  < expression>
<one_line_comment> ::= # .* \n
<multi_line_comment> ::= !# .* #!


<expression> ::= <expression> <low_prec_op> <term> |
                              <term>
<term> ::= <term> <high_prec_op> < factor> |
                 <factor>
<factor> ::= (<expression>) |
                     <value> |
                     <non_void_func_call> | <non_void_primitive_func_call> |
                     <logic_exprs> | <receive>


<logic_exprs>::= <logic_exprs> or <conjunction>| <conjunction>
                              <conjunction> ::= <conjunction> and <negation>    | <negation>
                              <negation>    ::= not <negation> | <logic_expr> | ( <logic_exprs> )
```

```
<logic_expr> ::=        <identifier> | <value> <comp_op> <value>
<comp_op> ::= ==| != | < | <= | > | >=

<low_prec_op> ::= + | -
<high_prec_op> ::= * | / |  %
<type> ::= float | int | bool | string | connection
<value> ::= <int_literal> | <bool_literal> | <float_literal> | <string_literal> | <identifier>
<literal> ::= <integer_literal> | <float_literal> | <bool_literal> | <string_literal>
<digit> ::= [0-9]
<digits> ::=<digit> | <digits> <digit>
<sign> ::= [+-]
<int_literal> ::=<sign> <digits> | <digits>
<bool_literal> ::=true | false
<float_literal> ::=<sign> <digits> <decimal_point> <digits>
                        <sign> <decimal_point> <digits> |
                        <digits> <decimal_point> <digits> |
                        <decimal_point> <digits>
<letter> ::=[a-z] | [A-Z]
<identifier> ::=<letter> | <identifier> <letter> | <identifier> <digit>
<string_literal> ::= \”[^\”]*\”
```

2. Description of Non-Terminal Literals

<program>

<program> is the starting literal of the program, it houses <func_list> and <stmt_list>. By separating the statement list and function list, the user can define functions only at the end of the program or could choose not to define any functions at all.

<func_list>

This function stores a list of user-defined functions. These functions can later be called from the main statements or the functions themselves.

<stmt_list>, <stmt>

<stmt_list> is used inside the block statements (e.g., loops, if then else statements, functions) and stores the statements that hold the instructions for our programs. The statements follow the C syntax rules where indentation isn't enforced, and the statements are divided by semicolons.

<func>, <void_func>, <non_void_func>

The user should use the reserved word "func" when defining a function. After the function declaration, the user must choose between the void (non-returnable) and non-void (returnable) functions. While declaring void functions, the user should use the "void" reserved word to declare. While declaring a non-void function, the user should choose a type that the function should return. The functions should be inside curly braces, and the non-void function should always return a value to avoid ambiguity.

<signature>, <in_args>

The <signature> holds the identifier for the function, and if the function requires input parameters, it uses <in_args> inside parentheses which are identifiers and their types. By convention, the user-created function should be named in the camel case.

<return>

Return statements returns the assigned expression (must be the return type) back to the function caller.

<func_call>, <arg>, <args>

Functions calls are described as such. Some functions can require arguments to pass data into their scopes. When required to pass, the <args> and <arg> literals will be used between parentheses. Identifiers and expressions will be passed as arguments. Arguments of the functions with parameters will be called by the <args> non-terminal. <args> is made up of <arg> non terminals which represents a single argument of a function call. <args> can store as many <arg> non-terminals as the function requires. All arguments are separated by comas.

<non_void_primitive_func_call>

Non-void primitive functions are built-in functions of Iotech. They enable the programmer to interact with the machine sensors and other input-output devices. All of the built-in functions return integers by design so that the data can be easily transmitted through a URL. By convention, pascal case is used for naming the primitive functions. Primitive functions are as follows:

- ReadTimeStamp()
    - Gives the number of seconds passed since January 1, 1970 (UTC)
- ReadTemperature()
    - Returns the temperature value from the input temperature sensor in Kelvin.
- ReadHumidity()
    - Returns the humidity value read from the humidity sensor.
- ReadAirPress()
    - Returns the air pressure reading from the pressure sensor in psi.
- ReadAirQual()
    - Returns the air quality read from the air quality sensor in the AQI index
- ReadLight()
    - Returns the level of the light measured by the light sensor in lux.
- ReadSoundLevel(<integer_literal >)
    - Returns the sound level in a given frequency (passed as a parameter) read by the sound sensor in decibels.

<void_primitive_func_call>

Void primitive functions are also a built-in function of Iotech, but it is separated from non-void primitive function calls because void functions do not return a value. There is currently only one void primitive function call which is SetSwitch. However, SetSwitch is derived from <void_primitive_func_call> so that the language is adaptable, and some additional void primitive functions can be added easily in the future. In SetSwitch function, the ten switches can be set as on/off to control some actuators. The first argument of the function is a boolean to define if the switch should be on or off (true for on, false for off), the second argument is the switch number which is a digit value.

<block_stmt>

Block statements are statements inside curly braces. It includes an if statement, a while loop, for loop, connect statement, and a do-while loop.

<if_stmt>

If statement is the conditional statement that performs different actions based on the result of the logic expression. In the if statement, the reserved words “if” and “else” are used. A logic expression that is written inside parentheses should follow the reserved words. The logical expression can be any logical expression or a Boolean identifier. Then, inside the curly braces, the user may write any statement.

<elseif_stmts>
<elseif_stmts> is a list of else if statements that can be the size of 1 or many more. It is called after the prior if or else if condition is false.

<while_stmt>
While statement is the while loop that starts with the reserved word “while” and continues with a logical expression inside parentheses. The logical expression can be any logical expression or a Boolean identifier. Then, inside the curly braces the user may write any statement.

<for_stmt>, <for_init>
For statement is the for loop, which starts with the reserved word “for”. After that, inside parentheses, there should be a <for_init> literal followed by a semicolon, followed by a logic expression, followed by another semicolon, followed by an assign statement. <for_init> is either a declaration or an assign statement that will determine which variable should take which value. The semicolons are to identify each statement inside parentheses. The logic expression can be either any logic expression or can be an identifier. If it is an identifier, the loop will be an infinite loop (since it will always be true or false). The statements within the blocks are enclosed in curly braces.

<do_while_stmt>
A do-while statement is a loop that will complete the statements inside the loop at least once. After the reserved word “do”, there will be statements enclosed in curly braces. Then, the reserved word “while” will be used, and the block will continue with a logical expression inside parentheses. The logical expression can be any logical expression or a boolean identifier. The do-while statement concludes with a semicolon.

<connect_URL>, <URL>

<connect_URL> is a block statement. It takes a <URL> as its parameter. The <URL> is a string value that is a URL. The connect statement establishes a connection with the URL passed and is creates a connection object.

<connect_obj>, <send>, <receive>, <terminate_connection>

When a connection is made, <connect_obj> acts as a object that has 3 fucntion calls. <send> allows user to send an integer value to the connected URL, <receive> allows user to receive an integer from the URL. And <terminate_connection> terminates the connection between the URL and the system.

<non_block_stmt>

Non-block statements are statements that are not encapsulated in curly braces. They contain assign statements, declaration statements, function calls, one line, and multi-line comments. If they are derived from <non_block_stmt> literal, there should be a semicolon after assign statements, declaration statements, and function calls.

<declaration_stmt>

The declaration statement is used for declaring a new variable to the program. The programmer must first write the data type of the variable and then the identifier of the variable. There are 2 ways to declare a variable. First is by declaring the variable and not assigning anything to it. The second is by both declaring the variable and then assigning a value by an expression. By default int values are 0, float values are 0.0, strings are ””, and bool values are true. There is also constant variable declaration where the value isn’t changeable.

<assign_stmt>

Assign statement is the operation where the users can assign a value to an identifier by using an assign operator. The left-hand side must contain an identifier, and the right-hand side can contain any expression.

<one_line_comment>, <multi_line_comment>

There are 2 types of comments: one-line and multi-line comments. One-liners start from the “#” character and go on until the end of the line. Multi-line comments will start with the “!#”

key and comment out anything until the “#!” key is matched. The user can write anything inside the comments.

<expression>

Expressions are anything that denotes a value in Iotech. Expressions are used for assignments, arithmetic operations, parameter values, and more.

<term>, <factor>

Term and factor non-terminals are used to eliminate ambiguity in arithmetic expressions. These two non-terminals both denote the same values but surround themselves with different precedenced operators. This enforces precedence in the BNF.

<high_prec_op>, <low_prec_op>

Iotech uses the same precedence rules in arithmetic operations as math. High precedence operators are “*/%” whereas low precedence operators are “+-”.

<type>

It is to declare the type of an identifier. There are four types: float, int, bool, and string. These reserved words are used to declare a function or variable’s type.

<value>

Value is the general form. It consists of float values, boolean values, string values, and float values. It is used as arguments inside functions, logical expressions, and on the assign and declaration operation’s right-hand side.

<logic_exprs>

<logic_exprs> literal is a combination of logic expressions. <logic_exprs> can be inversed by using the reserve word “not” in front of them. The reserved word “or” behaves like usual or operator, and the reserved word “and” behaves like usual and operator. Expressions are written in parentheses to avoid ambiguity.

<conjunction>, <negation>

<conjunction> and <negation> are used to make precedence rules in logical operations. Precedence order: (<logic_exprs >) ,not, and, or.

<logic_expr>

Logic expression can either be an identifier (that should hold a Boolean value) or it can be a comparison. In the comparison, on the left-hand side, there will be a value to be compared to, followed by a comparison operator. On the right-hand side, there should be another value. After the comparison is completed, the result will be either true or false.

<literal>

Literals are used for using a value directly inside expressions without assigning them to variables.

3. Terminals

> → greater than operator

>= → greater than or equal operator

< → less than operator

<= → less than or equal operator

== → equality operator

!= → not equal operator

= → assignment operator

+ → addition operator

- → subtraction operator

/ → division operator

* → multiplication operator

% → modulus operator

and→ and operator for two boolean expressions

not → not operator inverses the output of the logical expression it is assigned

or → or operator for two boolean expressions

() → parentheses are used for grouping expressions

{} → curly braces are used for denoting the area of a block of statements

# → single line comment starter

!# → multi-line comment starter

#! → multi-line comment ender

, → comas are for the separation of arguments during function calls and the declaration of function arguments

4. Non-trivial Tokens

Comments:
There are 2 types of comments in Iotech: Single-line and multi-line comments. Single-line comments are denoted by the “#” character and comment out the entire line following it. Multi-line comments, on the other hand, are started by the “!#” characters and comment out everything until the “#!” characters are matched.

Identifiers:
All identifiers consist solely of alphabetic characters to enforce uniformity. The convention for declaring identifiers is the usage of camel cases. Identifier conventions are uniform for all types of variables and user-written functions to further enforce uniformity. By this, Iotech aims for a more readable and consistent language. Connection identifiers are an exception to the convention since they require a dollar sign before their names.

A valid identifier that conforms to convention: myValue
Valid identifiers that do not conform to convention: MyValue, myvalue
Invalid identifiers: myValue1, mu_value

Connection object identifier: $myConnection

Literals:
booleans: Boolean literals can only be true and false. Iotech uses “true” and “false” reserved words to make the language more readable and writable since true and false is coomon-language terms.

strings: Strings are arrangements of characters within quotation marks. Quotation marks are necessary for string literals so that they can be separated from identifiers. The main usage and the main reason behind the addition of strings are so that they can be used for defining URLs.

integers: Integer is the core data type of Iotech since it is used in all primitive functions and connection statements. The definition of an integer literal is to simply write an integer and its sign behind it (it isn’t necessary for positive numbers). 0’s can be written on the most significant digits, and the unnecessary zeros will be omitted during run time.

floats: Float literals can be defined as decimals + period + decimals or period + decimals. Just as for integer values, “+-” can be added to denote signs (+sign isn’t necessary)

Reserved Words:

true Used for boolean literal.
false Used for boolean literal.
not Used for getting a reverse of a logic expression.
void Used for declaring function type.
int Used for declaring variable or function type.
float Used for declaring variable or function type.
bool Used for declaring variable or function type.
string Used for declaring variable or function type.
if Used for if condition statements.
else Used for if condition statements.
for Used for declaring for loop.
while Used for declaring while loop.
do Used when declaring a do-while loop.
func Used for declaring a function.
return Used when returning a value from non-void type functions.
and Used in logical expressions.
or Used in logical expressions.
const Used in decleration statements for declaring constants.
ReadTimeStamp Primitive function name to get time in seconds.
ReadTemperature Primitive function name to get temperature.
ReadHumidity Primitive function name to get humidity.
ReadAirPress Primitive function name to get air pressure.
ReadAirQuality Primitive function name to get air quality.
ReadLight Primitive function name to get light level.

ReadSoundLevel Primitive function name to get sound level.
SetSwitch Primitive function name to set switches.
.send Primitive function name to send integer to connection.
.receive Primitive function name to receive integer through a connection.
.terminate Primitive functions terminates the connection of connection object.
connect defines the connection object

Reserved words are chosen similar to the terms used in everyday and programming languages. This way, users can use a very daily-life syntax while coding, which affects writability and readability of the language.

5. Evaluation of Iotech with Language Criteria

Readability:
Iotech makes readability easier in many ways. One way is by forcing the main statements (the statements that run initially) to be written at the top of the file and the function declarations at the bottom of the file. So, when someone tries to read the code, there are no statements lingering between functions, and all functions can be easily found beneath the file. Iotech also makes it easier to recognize primitive functions by using pascal case on primitive functions, whereas using camel case on other identifiers. Another readability aspect of the syntax is that it uses everyday words as keywords like Python but also provides the concision of the C syntax. Finally, IoTech separates the connection object identifiers by enforcing dollar sign usage.

Writability:
To achieve writability, Iotech aimed for simplicity. Usually, there is only one way to perform a simple task. For instance, one way to assign a value (one assign operator), and one way to declare functions (using the term “func”), and it does not involve a complex syntax. Also, in Iotech, the usage of excessive lines while coding can be avoided by combining declaration and assignments in one line, combining multiple logic expressions in one line, returning any calculation without making and assigning the calculation to other identifiers on previous lines, directly assigning function calls to an identifier. By these combinations, the code can be more writable.

In Iotech, reserved words are chosen similar to the terms used in everyday language. This way, users can memorize and use a very daily-life syntax while coding, which affects writability. Also, the precedence rules are the same as they are in math to increase writability and avoid any confusion. The readability of the language, in general, also affects the writability of Iotech positively.

Reliability:

Reliability was a core part of the design philosophy of Iotech. For example, logical expressions are designed to eliminate precedence errors. Furthermore, the main statements which initially run when the program is running must be separated and come before the function declarations. This allows for a cleaner and more maintainable codebase. Another attempt to make the language more reliable was making all primitive functions work with integer values. This makes it so that all primitive functions can work with each other despite the language not having type conversions. Also, all if, else if, else statements' blocks were forced to be put inside curly braces (even if it is a one-liner). This is to ensure that subsequent if-else statements don't mix together. For logic expressions, there is "and", "or" and "not" precedence to avoid ambiguity. Finally, for arithmetic expressions <term> and <factor>, non-terminals are used in order to follow precedence rules in the BNF and make sure arithmetic expressions work as they do in math.