CS 315

Project 2

Muhammad Arham Khan - 21701848 - Section 3
Muhammad Bilal Bin Khalid - 21701834 - Section 3
Daniyal Khalil - 21701092 - Section 1

Language Name: DIBS

Group Name: 6

Syntax Description:

The language ‘DIBS’ has been devised while strictly keeping in mind the 3 criterias for programming languages: Readability, Writability, Reliability. The language is made as close as possible to widely used programming languages while keeping in mind the target audience and usage (IoT). To make the debugging/ development easier, the language is case insensitive unlike most imperative languages, hence it does not matter whether the Caps Lock was on or not while you were coding. In DIBS, semicolons are not required to show the end of line, just a newline is enough. Considering that there are not many data types/ classes needed in the language’s scope, DIBS does not have any data types and variable assignment/ usage is fairly straightforward. It has an infrastructure focused on IoT devices and supports only two classes, Node and Connection. Instances of Nodes and Connections are stored as variables and can be utilized throughout their scope to manipulate connections/ IoT devices. The language supports nested ifs and loops as well. Since DIBS does not employ curly brackets to mark the end of scope, we use a universal token (“end”) to identify the end of scope for For/ While/ If etc. Finally, even though the language is case insensitive the variable names are restricted to Alphanumeric characters only.

BNF:

```
<program> ::= <lines>

<lines> ::= <line>
          |<line><lines>

<line> ::= <for_loop><newline>
         |<while_loop><newline>
         |<if_st><newline>
         |<assign_st><newline>
         |<expr><newline>
         |<comments><newline>
         |<connection_st><newline>
         |<function_def><newline>
         |<function_call><newline>
         |<switch_control><newline>
         |<return_st><newline>
         |<newline>

<if_st> ::= if <logic_expr><newline><lines> end | if
           <logic_expr><newline><lines> else <newline><lines> end


<for_loop> ::= for <var> = <integer> to <integer> <newline> <lines>
             end
```

```
            | for <var> = <integer> to <var> <newline> <lines> end
            | for <var> = <var> to <integer> <newline> <lines> end
            | for <var> = <var> to <var> <newline> <lines> end
            | for <var> = <var> to <expr> <newline> <lines> end
            | for <var> = <expr> to <var> <newline> <lines> end
            | for <var> = <integer> to <expr> <newline> <lines>
              end
            | for <var> = <expr> to <integer> <newline> <lines>
              end
            | for <var> = <expr> to <expr> <newline> <lines> end

<while_loop> ::= while <logic_expr> <newline> <lines> end

<comments> ::= # <line>
            | ## <lines> ###


<output> ::= print(<var>) | print(<constant>) | print(<expr>) |
             print(<logic_expr>)

<input> ::= input() | input(<multi_params>)

<expr> ::= <operand><op><operand>
            | ( <expr> )
            | <expr><op><operand><op><operand>
            | <expr><op> ( <expr> )

<logic_expr> ::=| <operand><comparison_op><operand>
         | ( logic_expr )
         | <logic_expr><boolean_op><operand><comparison_op><operand>
         | <logic_expr><boolean_op>( <operand><comparison_op><operand )

<multi_params> ::= <multi_params> , <var>
            | <multi_params> , <const>
            | <var> | <const> | <>

<function_def> ::= function <char><string> ( <multi_params> )
                   <newline> <lines> end

<function_call> ::= <char><string> ( <multi_params> )

<return_st> ::= return <var>
            | return <expr>
            | return <const>
            | return <logic_expr>
```

```
<assign_st> ::= <var> = <const>
           | <var> = <expr>
           | <var> = <var>
           | <var> = <boolean>
           | <var> = <function_call>
           | <var> = <class_declaration>
           | <var> = <connection_receive_st>
           | <var> = <input>
           | <var> = <primitive_func>

<connection_assign_st> ::= <var>.receive()

<connection_st> ::= <var>.disconnect() <newline>
           | <var>.connect( "<string>") <newline>
           | <var>.connect( <var> ) <newline>
           | <var>.send( <integer> ) <newline>
           | <var>.send( <var> ) <newline>
           | <connection_assign_st>

<switch_control> ::= <var>.setSwitch( <digit>, <boolean>) <newline>

<primitive_func> ::= <var>.getTime()
           | <var>.getLight()
           | <var>.getHumidity()
           | <var>.getTemp()
           | <var>.getPressure()
           | <var>.getAirQuality()
           | <var>.getAirPressure()
           | <var>.getSoundLevel()

<operand> = <var> | <const> | <primitive_func> | <function_call>

<op> ::= <boolean_op> | <arithmetic_op>

<class_declaration> ::= new <class_name>

<class_name> ::= Node() | Connection("<string>") |
            Connection(<var>)

<arithmetic_op> ::= + | - | * | / | ** | %

<comparison_op> ::= > | < | >= | <= | == | !=

<boolean_op> ::= and | or
```

```
<newline>::= \n

<boolean> ::= true | false

<var> ::= <char>|<char><string>

<const> ::= "<string>" | <integer> | <float>

<string> ::= <string><char>|<string><digit>|<char>|<digit>|<>

<float> ::= <signed_integer>.<unsigned_digit> |
<unsigned_digit>.<unsigned_digit>

<unsigned_integer> ::= <integer><digit> | <digit>

<signed_integer> ::= - <unsigned_integer>

<digit> | <digit>

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

<char> ::= A| a | B | b | C | c | D | d | E | e | F | f | G | g | H |
h | I | i | J | j | K | k | L | l | M | m | N | n | O | o | P
| p |Q | q | R | r | S | s | T | t | U | u | V | v | W | w | X | x | Y
| y | Z | z
```

Description:

1) End Of Line is shown by a newline rather than a ‘;’. While in CS 101/102 using semicolons in an IDE such as drJava that did not even provide basic error definitions we had issues with adding semicolons in front of if conditions or loop. It also eases the writability of the language.

2) Data types for identifiers do not need to be declared before hand. Since the language has a very small scope, only basic data types are needed, making it more writable.

3) Case Insensitivity is something that very few present languages offer. Again to make the language more readable and writable this feature was added and due to the small catering level of the language it is better for it to be case insensitive.

4) In this language since we do not have curly braces, every loop, condition, and function ends with ‘end’. It increases the readability of the language, hence easily indicating the end of structure.

5) Single line comments in DIBS are started by ‘#’. Multiline comments start by ‘##’ and end with ‘###’. The motivation for this is that we have never seen ‘#’ been used in programming language, so we can use ‘#’ without having any interference with the code.

6) There should be a space, “ ” after all loops and selection statements to make sure that the expression is not confused with the statement/loop call.

7) There should be a space, “ ” after all loops and selection statements to make sure that the expression is not confused with the statement/loop call.

8) Function calls should not have “ ” between the function and parentheses. For example function_name()

9) Precedence is enforced by in the language

```
<program> ::= <lines>
```
Program can be multiple lines.

```
<lines> ::= <line>
         |<line><lines>
```
Lines can be multiple line attributes.

```
<line> ::= <for_loop><newline>
         |<while_loop><newline>
         |<if_st><newline>
         |<assign_st><newline>
         |<expr><newline>
         |<comments><newline>
         |<connection_st><newline>
         |<function_def><newline>
         |<function_call><newline>
         |<switch_control><newline>
         |<return_st><newline>
         |<newline>
```
A line can be anything in this language.

1) For Loop
2) While Loop
3) If statements
4) Assignment Expressions
5) Assignment Statements
6) Comments
7) Connection Statements
8) Function Definition
9) Function Calls
10) Switch Controls
11) Return Statements
12) Empty Lines

```
<comments> ::= # <line>
           | ## <lines> ###
```
BNF for comments. Single line comments starts with ‘#’ and end with the new lines. Multiples line comments are started with ‘##’ and ends with ‘###’.

```
<if_st> ::= if <logic_expr><newline><lines> end
          | if  <logic_expr><newline><lines> else <newline><lines> end
```

This is used to define an if statement with a logical expression within or without parentheses. This statement also deals with dangling ifs because in this situation every 'end' is used to define the end of statement whether 'if' is with 'else' or without 'else'.

```
<for_loop> ::= for <var> = <integer> to <integer> <newline> <lines>
                end
             | for <var> = <integer> to <var> <newline> <lines> end
             | for <var> = <var> to <integer> <newline> <lines> end
             | for <var> = <var> to <var> <newline> <lines> end
             | for <var> = <var> to <expr> <newline> <lines> end
             | for <var> = <expr> to <var> <newline> <lines> end
             | for <var> = <integer> to <expr> <newline> <lines> end
             | for <var> = <expr> to <integer> <newline> <lines> end
             | for <var> = <expr> to <expr> <newline> <lines> end
```

The for loop BNF. The following cases work in this:
For x = 1 to 3
For x = 1 to a
For x = a to 6
For x = a to b
For x = a to (1*3)
For x = 3 to (a+4)
For x = (a+3) to 10
For x = (a+4) to (b-5)

```
<while_loop> ::= while <logic_expr> <newline> <lines> end
```

The while loop which contains a logical expression and ends with an 'end'.

```
<output> ::= print(<var>)
           | print(<constant>)
           | print(<expr>)
           | print(<logic_expr>)
```

Output statement used for printing the system on the console.

```
<input> ::= input() | input("string")
```

Input statement used to enter data either from the console or from a file in the same directory.

```
<expr> ::= <expr><arithmetic_op> <var> <arithmetic_op> <const>
         | <expr><arithmetic_op> <var> <arithmetic_op> <var>
         | <expr><arithmetic_op> <primitive_func> <arithmetic_op>
           <var>
         | <constant><arithmetic_op><var>
         | <constant><arithmetic_op><constant>
         | <expr><arithmetic_op> ( <var><arithmetic_op><constant> )
         | <expr><arithmetic_op> ( <constant><arithmetic_op><constant> )
         | <expr><arithmetic_op>( <var><arithmetic_op><var> )
         | <expr><arithmetic_op>( <constant><arithmetic_op><constant> )
         | ( expr )
         | <expr><arithmetic_op> <var> <arithmetic_op> <const>
         | <expr><arithmetic_op> <var> <arithmetic_op> <var>
         | <var> <arithmetic_op> <const>
         | <var> <arithmetic_op> <var>
```

The expression statement, which can be in any form of arithmetic operators applied to constants and variables.

```
<logic_expr> ::=| <var><comparison_op><var>
           | <var><comparison_op><const>
           | <const><comparison_op><var>
           | <const><comparison_op><const>
           | ( logic_expr )
           | <logic_expr><boolean_op><var><comparison_op><var>
           | <logic_expr><boolean_op><var><comparison_op><const>
           | <logic_expr><boolean_op><const><comparison_op><var>
           | <logic_expr><boolean_op><var><comparison_op><var> )
           | <logic_expr><boolean_op><var><comparison_op><const> )
           | <logic_expr><boolean_op><const><comparison_op><var> )
```

A logic expression which can contain boolean operators, comparison operators and variable to relate them with each other. The expression can be in any form, such as (a and (b or c)).

```
<function_def> ::= function <char><string> ( <multi_params> )
                <newline> <lines> end
```

The function definition, which names the function in a string (The first letter is restricted to a character) and adds parameters in the parenthesis for the function call, it is followed by lines of code and ends with an ‘end’.

```
<function_call> ::= <char><string> ( <multi_params> )
```

Calling the function with its name, and parameters in parenthesis.

```
multi_params> ::= <multi_params> , <var>
                | <multi_params> , <const>
                | <var>
                | <const>
                | <>
```

The definition for multiple parameters to be fed into a function call. It can be a variable or a constant or nothing in case of no params.

```
<return_st> ::= return <var>
           | return <expr>
           | return <const>
           | return <logic_expr>
```

The return statement to write a variable, an expression or a constant to a function call.

```
<assign_st> ::= <var> = <const>
           | <var> = <expr>
           | <var> = <var>
           | <var> = <boolean>
           | <var> = <function_call>
           | <var> = <class_declaration>
           | <var> = <connection_assign_st>
           | <var> = <input>
           | <var> = <primitive_func>
```

The assignment state which can be any of the following:

1) Constant
2) Variable
3) Expression
4) Boolean
5) Function call
6) New class instance
7) Received data from a connection
8) Input from the User
9) Data from a primitive function

```
<connection_assign_st> ::= <var>.receive()
```

The statement to receive data from a connection and then assigning it to a separate variable.
is

```
<connection_st> ::= <var>.disconnect() <newline>
           | <var>.connect( "<string>") <newline>
           | <var>.connect( <var> ) <newline>
           | <var>.send( <integer> ) <newline>
           | <var>.send( <var> ) <newline>
           | <connection_assign_st>
```

Connection with the url to send or receive data to the url. The url in string and the data is in integer. The connection can be disconnected using 'disconnect'. '.send' is used to send a variable or integer to the Node. Connection_assign_st is used for receiving.

```
<switch_control> ::= <var>.setSwitch( <digit>, <boolean>) <newline>
```

Used to control the switches which are in the IoT nodes to control the actuators. The digits from 0-9 are used to describe the number of switches, and the boolean is true if on and false if off.

```
<primitive_func> ::= <var>.getTime()
           | <var>.getLight()
           | <var>.getHumidity()
           | <var>.getTemp()
           | <var>.getPressure()
           | <var>.getAirQuality()
           | <var>.getAirPressure()
           | <var>.getSoundLevel()
```

The variables for these functions can only be used on Nodes, in order to get data from the IoT devices.

```
<class_declaration> ::= new <class_name>
```

Making a new instance of a class, it can either be a Node or Connection.

```
<class_name> ::= Node() | Connection("<string>")
                | Connection(<var>)
```

The BNF for classes in this language. It does not allow users to make class but there are only two types which are Nodes, which do not have any parameters and Connection, which requires a string or variable of url for a connection.

```
<arithmetic_op> ::= + | - | * | / | ** | %
```

The arithmetic operators in our language.

+ : Addition

- : Subtraction

*: Multiplication

/: Division

**: Power

%: Modulus

`<comparison_op> ::=  > | < | >= | <= | == | !=`

The comparison operators that are being used in this language.
>: Greater than
<: Less than
>=: Greater than or equals to
<=: Less than or equals to
==: Equals
!=: Not equals

`<boolean_op> ::= and | or`

The only boolean operators we are using in this language are going to ‘and’ and ‘or’.

`<newline>::= \n`

BNF for newline, since we are using newline as end of line rather than semicolons.

`<boolean> ::= true | false`

A boolean is either true or false.

`<var> ::= <string>`

A variable name is always a string in this language.

`<const> ::= "<string>" | <integer>`

The BNF for constants in the language which can contain either strings or integers.

`<float> ::= <signed_integer>.<unsigned_digit> | <unsigned_digit>.<unsigned_digit>`

The BNF for float which has caters to either negative or positive decimal numbers.

`<unsigned_integer> ::= <integer><digit> | <digit>`

An integer which is not signed

`<signed_integer> ::= - <unsigned_integer> <digit> | <digit>`

An integer which is signed

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

The BNF for digits in the language.

`<char> ::= A| a | B | b | C | c | D | d | E | e | F | f | G | g | H | h | I | i | J | j | K | k | L | l | M | m | N | n | O | o | P | p |Q | q | R | r | S | s | T | t | U | u | V | v | W | w | X | x | Y | y | Z | z`

The BNF for simple characters in our language, since the language is following a case insensitive approach, the upper and lowercase letters are treated as the same.

Sample programs:

```
#Dibs Sample program

function main(input1,input2)
  ##assignment
    samples###

  temp = a

  #if statement
  if (a > b)
    iF ((a < 4) And a ==b)
      a = 1
    elSe
      a = 0
    end
  end

  #while statement
  while ( b == 1 And c < 1)
    a = 1
  end

  while a == True
    b = b + 1
  end

  a = (1 + 2) / (1+ b)
  c = b + ((a-c) * (a+d ))
  d = a + (b + 2)
  d = (a + 2)
  boolOut = a And (b Or c) Or d
  #for loop
  for a = 1 to 10
    for b = (a + 2) / 2  to b - 3
      a = 11
    end
  end
end
```

```
function main2()

  a = new ConnecTion("arhamkhan.me")
  a = new ConnecTion( a)
  b = new Node()

  a.connect( "arhamkhan.me")
  a.disconnect()
  c = a.receive()
  a.send(1)
  a.send(b)

  c = b.getTIme()
  c= b.getAirPREssure()
  c = b.getSoundLEVel()
  c = b.getLighT()

  a = (1 + 2) / (1+ a.getTime())
  b = 1 + tempFunction()

  a.setSwitch(0,true)

  return 1
  return "0"
  return 1.2
  return a
  return (1+2) + (2+ c.getTime())
end
```