Project 2 Report

Group Members:	Batuhan Tosyali	21702055	Section 2
	Ufuk Bombar	21703486	Section 1
	Enver Yigitler	21702285	Section 1

swBiot

1-General Description of swBiot

swBiot is imperative, domain-specific, and easy-to-use language for rapid IOT application development. We have adopted C-like loops and control structures due to the popularity of these structures. We defined special keywords, constants and operators to create IOT specific features. A swBiot program consists of statements and function definitions. There is not a main function to start the program, program starts to execute statements from start of the file sequentially. These features provide a simplistic look to the language, to improve readability and writability. Now we will look at some important language structs to get started to code in swBiot.

1.1 Declarations

Declarations can be a variable or function declaration. User can initialize variables when variable is declared.

```
INT a; %% variable declaration %%
INT b = 2; %% variable initialization %%

function factorial (INT n) INT {return 0;} %% function declaration %%
```

1.2 Reserved types, constants and operators.

We defined tree reserved type for swBiot: `URL`, switches (`SW1, SW2, ...`), and sensors (`TEMP[a], HUM[b],...`). Besides, there are two special assignment operators for these types. : input assign (<<) , output assign (>>).

Input assignment and output assignments are used for `URL` type to send or receive integer from/to connected `URL (connect()` function will be explained later). Example:

```
URL myMicroWave = 'localhost/devs.mcw.1386';
connect(myMicroWave); %% connect to URL %%

INT a;

myMicroWave >> a; %% receive integer from URL connection %%
myMicroWave << a; %% send integer to URL connection %%
```

Input assignment is used to open or close the switches.

```
SW1 << true;
SW2 << false;
```

Output assignment is used to receive integers from sensors.

```
INT a;
TEMP[1] >> a;
HUM[3] >> a;
```

1.4 Reserved functions

There are five reserved functions in swBiot: `time(), connect(URL url), disconnect(URL url), print(), error()`.

Example:

```
print("Hello World!"); %% prints Hello World! %%

print(time()); %% prints timestamp %%
```

```
URL url = ‘http://www.cs.bilkent.edu.tr/’; %% declare URL%%

connect(url); %% connect URL %%

disconnect(url); %% disconnect URL%%

if (url == ‘’){
      error(“URL is empty!”)
};  %% error outputs string and stops program %%
```

1.5 Control and Loop structures

We defined basic control structures like if, if-else and loop structures like while, for. If structure takes a logic expression inside the normal parenthesis just like any other C type language. After the parenthesis there can be a statement list inside curly brackets or there can be only one statement without curly braces. While loop is also similar to if statement. For loop takes three different parts. First part is defined as an initial statement, second is logic expression and the last one is a regular statement. For loop can also have curly braces.
Example:

```
if ( a > 4)
{
   print(“a is bigger than 4”);
} else
{
   print(“a is not bigger that 4”);
};

INT i = 10;
while( i > 1)
{
   i = i - 1;
};

for(INT i = 0; i < 10; i = i + 1)
{
      print(“i = ” + i);
};
```

1.6 Function declaration and function calls

There are two types of functions, returned or void. If function return a value a return type must be declared in the function declaration and return statement must be used in function block. Function calls are called with name and parameters inside parentheses. Additionally, parameters can be fed to a function as an empty list or multiple expressions separated by commas.
Example:

```
function foo(INT a) INT
{
   return a;
};
```

```
INT a = foo(2);

function voidfunc(STRING str)
{
      print(str);
};

voidfunc("Hello World Again!"); %% prints hello world again!%%
```

That was the general description of the language swBiot. More formal descriptions and language grammar are given in the next sections.

2- Implementation of Parser

2.1- Start of the program

Our parsing process starts with nonterminal named program, which can also be seen in the start of the file denoted with %start program.

2.2- Major Statements

Our program can only create a non-terminal stmt_list which can create a single nonterminal stmt or can create stmt_list and stmt recursively. Both ways the result will have a SEMICOLON terminal which is a character ';' at the most right.

```
stmt_list   :  stmt SEMICOLON
                   | stmt_list stmt SEMICOLON  ;
```

The statements can become distributed among five specialized statement or it can become an expr.

```
stmt    : declaration_stmt
    | assign_stmt
    | init_stmt
    | if_stmt
    | loop_stmt
    | expr;
```

Starting from declaration_stmt, the declaration_stmt can be used for variable declaration which is var_dec or it can be used for function declaration which is func_dec.

```
      declaration_stmt   : funct_declaration
          | var_declaration;
```

The assign_stmt is obviously for assignment purposes. In this language we can assign terminal IDENTIFIER, terminal SWITCH which are predefined 10 switches, or non-terminal sensor_expr, to IDENTIFIER or non-terminal expr.
The operator assignments are terminals, which are EQ , INN and OUT. EQ is simply conventional '=' but INN and OUT are new. The INN operator ('<<') is used for loading integers to IDENTIFIER (which is actually URL typed identifier) from the given URL. The OUT is ('>>') simply used for sending data to given url.

```
<assign_stmt> ::= SWITCH INN <logic_expr>
                | IDENTIFIER EQ <expr>
                | IDENTIFIER OUT IDENTIFIER
                | IDENTIFIER INN <expr>
                | <sensor_expr> OUT IDENTIFIER
```

The following statement is if_stmt. if statement is conventional like in mostly other languages. We can create if statements using terminal IF (‘if’) and terminals LP , RP which stands for left paranthesis (‘(’) and right paranthesis (‘)’). We can write a logical expression inside the given paranthesis, which is stated as non-terminal logic_expr.

```
      if_stmt : IF LP logic_expr RP LCB stmt_list RCB
             | IF LP logic_expr RP LCB stmt_list RCB ELSE LCB stmt_list
RCB;
```

Then, just like in C++ or Java we can write our statement or statements in if, using RCB and LCB which stands for (‘{‘) and (‘}’). PReferably we can support our decisions with else using ELSE token which have the similiar rules.

After if_stmt we have loop statement. Again, just like in Java and C++ the syntax is similiar. Our loop_stmt may go to while_stmt or can go to for_stmt. In while_stmt, we have to type keyword ‘while’ which is WHILE token in yacc, a left paranthesis’(‘ a right paranthesis ‘)’ and an logic_expr between paranthesis. After that we shall write statements related to while loop between it’s curly brackets ‘{‘ and ‘}’. For loop is almost same. Only difference is we should write the logic_expr and two stmt’s just like in Java or C++ in order to make it work. The only difference from Java and C++ is the the two stmt’s are obligatory rather than optional.

```
      loop_stmt   : while_stmt
         | for_stmt;
   while_stmt   : WHILE LP logic_expr RP   stmt
         |  WHILE LP logic_expr RP    LCB  stmt_list RCB  ;
   for_stmt   : FOR LP init_stmt SEMICOLON  logic_expr SEMICOLON  stmt
RP LCB stmt_list RCB
   | FOR LP init_stmt SEMICOLON  logic_expr SEMICOLON  stmt  RP stmt;
```

2.3 - Major Expressions

As we said we can move forward from stmt to expr. The expr has also a big tree under it. It distributes into two different minor expressions named arithmetic_expr and url_expr.

```
      expr   : arithmetic_expr | url_expr;
```

arithmetic_expr is the most complex non-terminal in the yacc definition. It involves recursive calls to itself and four basic operation tokens, ADD, MINUS, MULT, DIV. It can also go to other major expressions such as logic_expr which is for logic and boolean operations, can go to func_call_expr or sensor_expr. The reason behind arithmetic_expr is designed such a way is to prevent conflicts.

```
      arithmetic_expr   : arithmetic_expr PLUS arithmetic_expr
         | arithmetic_expr MINUS arithmetic_expr
         | arithmetic_expr MUL arithmetic_expr
         | arithmetic_expr DIV arithmetic_expr
         | logic_expr
         | func_call_expr
         | sensor_expr
         | LP arithmetic_expr RP;
```

The url_expr is simply reducing into URLSTRING token which is simply conventional string but instead using “ “ we use ‘ ‘.

```
url_expr : URLSTRING;
```

Following aritmetic_expr, we can move on to . The logic_expr can recursively call itself in order to use the conventional logic tokens. Which are LEQ (‘==’), LNEQ (‘!=’), LOR(‘||’), LAND(‘&&’) , LT(‘<’), GT(‘>’), LTE(‘<=’), GTE(‘>=’). Alternatively it can go to integers float and boolean literals, named INT_LITERAL FLOAT_LITERAL BOOL_LITERAL. Or it can finally go to an identifier name IDENTIFIER

```
logic_expr :  logic_expr LAND logic_expr
         | logic_expr LOR logic_expr
         | logic_expr LEQ logic_expr
         | logic_expr LNEQ logic_expr

         | logic_expr LT logic_expr
         | logic_expr LTE logic_expr
         | logic_expr GT logic_expr
         | logic_expr GTE logic_expr

         | LNOT LP logic_expr RP
         | BOOL_LITERAL
             | INT_LITERAL
      | FLOAT_LITERAL
           | STRING_LITERAL
         | IDENTIFIER;
```

func_call_expr has the special reserved function keywords and also defines how we declare a function. It can go to CONNECT_FUNC LP IDENTIFIER RP, the token other than CONNECT_FUNC are trivial so I will explain only explain CONNECT_FUNC. It connects the given url. We also have DISCONNECT_FUNC which disconnects the given url. The LOG_FUNC token is used for outputting into log and ERROR_FUNC is just like LOG_FUNC but it prints colored error statement. finally the TIME_FUNC is used for getting the time stored in computer since 1970.
The func_call can also co to IDENTIFIER LP parameter_list RP. The parameter_list goes to expr or expr COMA which is ‘,’ parameter_list which is described earlier.

```
func_call_expr    :  IDENTIFIER  LP  parameter_list  RP
                        | IDENTIFIER  LP  RP
                        | DISCONNECT_FUNC LP IDENTIFIER RP
                        | CONNECT_FUNC LP IDENTIFIER RP
                        | LOG_FUNC LP expr RP
                        | ERROR_FUNC LP expr RP
                        | TIME_FUNC LP RP;
```

sensor_expr is just a special way to define Primitive Sensor. It goes to conventional definition which is SENSOR LSB arithmetic_expr RSB. By the way sensors are predefined in SENSOR.

Lastly we shall define Literals. INT_LITERAL is basically an integer which is 32 bit, BOOL_LITERAL is true or false which is true or false. FLOAT_LITERAL is like basic float which is also 32 bit.s

Operator Precedence Rules

There are four arithmetic operators and eight logical operators.

Arithmetic operators: +, -, *, ./
Logical Operators: | (or), && (and), < , >, <=, >=, ==, !=

Precedence Ranks of Arithmetic Operators : 1. ()
2. *, ./
3. +, -

Precedence Ranks of Logical Operators: 1. <, >, &&
2. <=, >=, |
3. ==, !=

3- Example Program

```
%%
    Duty of the program:
        First Function: The calculateAverageHumidity which is declared in the global
        scope calculates the average value from all humidity sensors. If number
        of humidity sensors is less than 1 then it gives an error and stops
        execution. If not it returns the average humidity value. But we divide
        it by MAX_HUMIDITY to have humidity between 0 and 1.

        Second Function: sendValueToUrlWithDelay function sends a value to an
        URL and waits for and response, if the response is exceeds the desired
        time then it gives an error. Else, it disconnects.


        Third Function: writeGivenValueToSwitches() takes a url sends a CODE_GET_INTEGER.
        This code means to get an integer from the url. After that it turns on the switches
        by number. For example, if number 3 is received then first 3 switches are opened.
%%

FLOAT MAX_HUMIDITY = 100;
INT RESPONSE_SUCCESS = 64;
INT RESPONSE_ERROR = 128;
INT CODE_GET_INTEGER = 124;
FLOAT MAX_DELAY = 1024;

INT HUM_SENSOR_COUNT = 5;

URL localhost = 'localhost/dev.mbp.0012';

function calculateAverageHumidity(INT sensorcount) FLOAT {
    FLOAT avgHumidity = 0;
    if (sensorcount > 0) {
        FLOAT temp;
        for (INT i = 0; i < sensorcount; i = i + 1) {
            HUM[i] >> temp;
            avgHumidity = avgHumidity + temp;
        };
    }
    else {
        error("Error! No humidity sensors are on board");
    };

    return avgHumidity;
};

function sendValueToUrlWithDelay(STRING value, URL url, FLOAT maxDelay) {
    connect(url);

    FLOAT startingTime = time();
    INT response;
    BOOL flag = true;

    url << value;
    url >> response;

    while (response != RESPONSE_SUCCESS && flag) {
        FLOAT currentTime = time();

        FLOAT delay = currentTime - startingTime;

        if (delay > maxDelay || response == RESPONSE_ERROR) {
            flag = false;
        };
        url << value;
        url >> response;
    };

    if (flag) {
        error("Error! Time exceeded while waiting for response!");
    };

    disconnect(url);
};

function writeGivenValueToSwitches(URL url) {
    connect(url);

    INT response;

    url << CODE_GET_INTEGER;

    url >> response;

    SW1 << response <= 1;
    SW2 << response <= 2;
    SW3 << response <= 3;
    SW4 << response <= 4;
    SW5 << response <= 5;
    SW6 << response <= 6;
    SW7 << response <= 7;
    SW8 << response <= 8;
    SW9 << response <= 9;
    SW10 << response <= 10;

    disconnect(url);
};

%% variable decleration and assignment %%
FLOAT avg = calculateAverageHumidity(HUM_SENSOR_COUNT) ./ MAX_HUMIDITY;
print("The average humidity is " + avg + "!");

%% function call with parameters as expressions %%
sendValueToUrlWithDelay("{'average_humidity':" + avg + "}", localhost, MAX_DELAY);
print("Average humidity transmitted to mobile phone with success!");

%% function call with parameters as identifiers %%
writeGivenValueToSwitches(localhost);
print("Value from url written to switches!");
```

BNF Description

```
<program>  ::= <stmt_list>


    <stmt_list>  ::=  <stmt> SEMICOLON
                                       | <stmt_list stmt> SEMICOLON


    <stmt>   ::= <declaration_stmt>
                             | <assign_stmt>
                             | <init_stmt>
                             | <if_stmt>
                             | <loop_stmt>
    | <expr>


    <declaration_stmt>   ::= <funct_declaration>
                             | <var_declaration>
    <var_declaration_list>   ::= <var_declaration>
                             | <var_declaration_list> <var_declaration>
    <var_declaration>   ::= TYPE  IDENTIFIER

    <funct_declaration>   ::= FUNCTION IDENTIFIER LP <var_declaration_list> RP  TYPE LCB <stmt_list> RETURN <expr>
SEMICOLON RCB
                      | FUNCTION IDENTIFIER LP var_declaration_list RP LCB stmt_list RCB


     <assign_stmt> ::= SWITCH INN <logic_expr>
                 | IDENTIFIER EQ <expr>
                 | IDENTIFIER OUT IDENTIFIER
                 | IDENTIFIER INN <expr>
                 | <sensor_expr> OUT IDENTIFIER


    <init_stmt>   ::= TYPE <assign_stmt>


    <if_stmt> ::= IF LP <logic_expr> RP LCB <stmt_list> RCB
              | IF LP <logic_expr> RP LCB <stmt_list> RCB ELSE LCB <stmt_list> RCB


    <loop_stmt>   ::= <while_stmt>
                                 | <for_stmt>
    <while_stmt>   ::= WHILE LP <logic_expr> RP   <stmt.
                                 |  WHILE LP <logic_expr> RP    LCB  <stmt_list> RCB
    <for_stmt>   ::= FOR LP <init_stmt> SEMICOLON  <logic_expr> SEMICOLON  <stmt>  RP LCB <stmt_list> RCB
    | FOR LP <init_stmt> SEMICOLON <logic_expr> SEMICOLON  <stmt>  RP <stmt>


    <expr>   ::= <arithmetic_expr> | <url_expr>


    <url_expr> ::= URLSTRING

    <logic_expr> ::=  <logic_expr> LAND <logic_expr>
         | <logic_expr> LOR <logic_expr>
         | <logic_expr> LEQ <logic_expr>
         | <logic_expr> LNEQ <logic_expr>
         | <logic_expr>  LT <logic_expr>
         | <logic_expr> LTE <logic_expr>
         | <logic_expr> GT <logic_expr>
         | <logic_expr> GTE <logic_expr>
    | LNOT LP <logic_expr> RP
    | BOOL_LITERAL
    | INT_LITERAL
    | FLOAT_LITERAL
    | STRING_LITERAL
    | IDENTIFIER

    <arithmetic_expr>   ::= <arithmetic_expr> PLUS <arithmetic_expr>
    | <arithmetic_expr> MINUS <arithmetic_expr>
    | <arithmetic_expr> MUL <arithmetic_expr>
    | <arithmetic_expr> DIV <arithmetic_expr>
    | <logic_expr>
    | <func_call_expr>
    | <sensor_expr>
    | <arithmetic_expr> RP



    <sensor_expr>    ::= SENSOR LSB  <arithmetic_expr> RSB



     <func_call_expr>    ::=  IDENTIFIER  LP  <parameter_list>  RP
                        | IDENTIFIER  LP RP
                        | DISCONNECT_FUNC LP IDENTIFIER RP
                        | CONNECT_FUNC LP IDENTIFIER RP
                        | LOG_FUNC LP <expr> RP
                        | ERROR_FUNC LP <expr> RP
                        | TIME_FUNC LP RP

     <parameter_list>    :  <expr>
                              |   <expr> COMMA <parameter_list>
```