CS-315 Programming Languages
Project 1 Report

Team 31
Section 3
Faaiz Khan // 22001476
Maher Athar Ilyas // 22001298
Amirreza Khoshbakht // 22001198

1. BNF Description of ".FAM"

- **Initial Program**

<program> ::= STARTPROGRAM<statement_list>ENDPROGRAM
<statement_list> ::= <statement> \n | <statement> \n <statement_list>
<statement> ::= <expression> |<iteration_statement>|<comment>|
<statement_with_comment>|<if_statement>|<function_call>|
<function_declaration> |<return_statement>

The non-terminal <program> shows that programs in ".FAM" start with the non-trivial token "STARTPROGRAM", contain a list of statements, and end with the non-trivial token "ENDPROGRAM". The non-trivial tokens added here aim to increase the readability of the program. Since it will be used by people with no prior programming experience, it may be helpful to define such boundaries to them. This will also help the writability of the program as the keywords are closer to the natural language they are familiar with.
The non-terminal <statement_list> contains a list of statements, 1 or more, each ending with a newline character to give the program better definition with regards to where each statement ends. The newline character was selected as a statement terminator so that the program is more readable, with only one statement per line. It also improves the writeability since the users are non-programmers who are used to adding newline character at the end of statements rather than semicolons or some other character.
The non-terminal <statement> may contain any 1 possible statement required to run a program, including expressions, loops, conditionals, function calls, and even comments.

- **Primitive Functions**

<primitive_function_call> ::= PRIM.<primitive_function> |
prim.<primitive_function>(<argument_list>)
<primitive_function> ::= read_temp | read_hum | read_air_qual | read_air_press |
read_light | read_sound_lvl | read_timestamp | set_switch_as

The non-terminal <primitive_function_call> defines a part of the language designed to call useful primitive functions already implemented within the language. Each primitive call starts with the non-trivial token "PRIM.", indicating that the function is a primitive function not

requiring implementation, increasing readability and reliability. Some examples of primitive functions calls are PRIM.read_light() and PRIM.read_timestamp().
The non-terminal <primitive_function> provides a list of all the names of the primitive functions available in the language. These functions serve to read information from the IoT nodes that may contain: temperature, humidity, air quality, air pressure, light, sound level, and a timestamp from a timer started from midnight (UTC) of January 1. 1970. The function set_switch_as takes two arguments, a digit from 0-9 and a TRUE / FALSE value, to turn a switch on or off to control the actuators.

- **Connection**

```
<connection> ::=  <identifier>
<connect_to_url> ::=<connection>. connect_to_url(<string>)
<disconnect> ::=<connection>. disconnect()
<fetch_integer> ::=<connection>. fetch_integer()
<send_integer> ::=<connection>. send_integer(<integer>)
<connection_functions> := <connect_to_url>|<disconnect>|<fetch_integer>|
                          <send_integer>
```

The non-terminal <connection> contains the name given to the instance of a connection object that will be used to connect to a URL.
The non-terminal <connect_to_url> describes the syntax of calling the connect_to_url function for a connection object. The function contains an argument containing the URL the user would like to connect to. When this function is called, it tries to connect the user to the desired URL through the connection object, returns TRUE or FALSE and prints a status report based on the success of the connection.
The non-terminal <disconnect> further elaborates the syntax for calling the disconnect function contained in the connection object. This function disconnects the user from an already connected URL and prints an error if no URL is connected to it.
The non-terminal <fetch_integer> contains the syntax of a function call by the name of fetch_integer. This function returns an integer value fetched from the URL the user is connected to, and it prints an error and returns a NULL value if no integer can be fetched.

The non-terminal <send_integer> explains the syntax for calling the function send_integer, used to send an integer to a URL that has been connected to, returning true or false based on the success and printing a suitable status report.
The non-terminal <connection_functions> provides a list of the four functions in a connection object mentioned in the previous paragraphs.

- **Non Primitive Function Call**

<nonprim_function_call> ::= <identifier> () | <identifier> (<argument_list>)
<argument_list> ::= <data> | <argument_list> , <data>
<function_call> ::= <nonprim_function_call> | <primitive_function_call> |
<connection_functions> | <output_call> | <input_call>

The non-terminal <nonprim_function_call> contains a syntax for calling non-primitive and user-defined functions that may or may not contain arguments.
The non-terminal <argument_list> contains a way to define 1 or more arguments, each separated by comma, containing the desired type of value required for the function call.
The non-terminal <function_call> contains all the non-terminals dealing with calling a function, including primitive and non-primitive functions, connection functions, and input and output functions.

- **Function Declaration**

<function_declaration> ::= FUNC <return_type> <identifier> (<parameter_declaration>)
\n <statement_list> <return_statement> ENDFUNC
| FUNC <return_type> <identifier> ( ) \n <statement_list>
<return_statement> ENDFUNC

<return_type> ::= <data_type>| Void
<parameter_declaration> ::= <data_type> <identifier> | <paramter_declaration> ,
<data_type> <identifier>
<return_statement> ::= RETURN <data>

The non-terminal <function_declaration> explains the grammar relating to defining a function. Each function begins with the non-trivial token "FUNC" and ends with the non-trivial token "ENDFUNC". These tokens help with the readability of the function as it clearly defines boundaries, better than braces and indents, and helps a user understand exactly where a function begins and where it ends. Due to the more defined boundaries, it increases the program's reliability as well as it's writability as the tokens are easily understood and written, due to them being close to the natural language. The non-terminal <return_type> defines all the possible types a function can return, including all data types and a special void type, that returns nothing. The return type is then followed by the name of the function, after which the parameters of the functions, 0 or more, are listed in parentheses, separated by commas. The non-terminal <parameter_declaration> contains the syntax to declare the parameters of a function, if they exist. After the paramter declaration and a subsequent newline character, a list of statements follows that include the functionality of the function, finally ending with a return statement, that will break the function and return the desired value to the user. The non-terminal <return_statement> explains the syntax of a return statement, starting with the keyword "RETURN" followed by a value that the function will return. This value could just be some data or it could also be stored in a variable or even a constant.

- **Comments**

<comment> ::= #<content>#
<statement_with_comment> ::= <statement><comment>

The non-terminal <comment> is used to describe the syntax of comments in ".FAM". Comments are useful to explain code to help increase its readability and reliability. Comments also make the code easier to edit. Comments can be inline next to a statement or on their own on a separate line. Comments start and end with a # character . This way the # clearly define the boundaries of each comment, increasing readability.

- **Initialization**

<datatype> ::= String | Integer | TrueOrFalse | Real | Char | DateTime | Date | Time
<default_initialize_var> ::= <datatype> <identifier> | Connection <identifier>
<assignment_initialize_var> ::= <datatype>< assignment_expression>
<initialize_const> ::= <datatype> _< assignment_expression>

Initialization is an important part in any program, used to define variables and constants. The non-terminal <datatype> contains a list of terminals which will be used as keywords for the datatypes in the program.
The non-terminal <default_initialize_var> is used to initialize variables with their default values, starting with the datatype followed by the name of the variable.

The non-terminal <assigment_initialize_var> explains the grammar of assigning and initializing in the same statement. It includes the datatype and then the assignment statement with the new variable.
The non-terminal <initialize_const> defines the grammar of assigning and initializing constant. They follow similar grammar to the variable assignment and initialization, but include a _ before the assignment statement to differentiate constant identifiers and variable identifiers.

- **Data Types**

<string> ::= “ <content>” | <null>
<TrueOrFalse> ::= TRUE | FALSE | <null>
<char> = ‘<printable_ascii>’ <null>
<integer> ::= <number> | - <number> | <null>
<real> ::= <integer> | <integer>.<number> | <null>
<date> ::= <digit><digit>-<digit><digit>-<digit><digit><digit><digit> | <null>
<time> ::=<digit><digit>:<digit><digit>:<digit><digit> | <null>
<datetime> ::= <date> <time> | <null>

This section of the BNF defines all the datatypes that can be used in “.FAM”:

- The non-terminal <string> is a string of printable characters enclosed within double quotations, i.e: “Hello World!”.
- The non-terminal <TrueOrFalse> is a binary datatype that can only contain a TRUE or a FALSE and will normally be used for boolean expressions, flags, switches, etc.
- The non-terminal <char> is a printable ascii character enclosed within single quotations, i.e: ‘A’ or ‘\n’.
- The non-terminal <integer> contains an integer value that can be either negative or positive or zero.
- The non-terminal <real> contains any real number.
- The non-terminal <date> is a way to represent dates in InsetName. It follows the DD-MM-YYYY convention, i.e: 12-05-1999.
- The non-terminal <time> is a way to represent time in “.FAM”. It follows the HH:MM:SS convention, i.e: 22:46:31.
- The non-terminal <datetime> is a way to represent the date and time together, in that order, separated by a space.

Any of these datatypes may contain a NULL value, signifying that there is a space in the memory reserved for a certain variable or constant, but there is no value in it at present.

- **Data Types Helpers**

<null> ::= NULL

<content> = <printable_ascii> | <content><printable_ascii>

<number> ::= <digit> | <number><digit>

<identifier> ::= <letter> | <identifier><letter> | <identifier><digit> |
<identifier><underscore><identifier>

<variable> ::= <identifier>

<constant> ::= _<identifier>

<entity> ::= <variable> | <constant>

<data> ::= <string> | <TrueOrFalse> | <char> | <integer> | <real> | <date> | <time> |
<datetime> | <entity> | <function_call>

<letter> = <upper_case_letter> | <lower_case_letter>

<printable_ascii> ::= <space> | <digit> | <upper_case_letter> | <lower_case_letter> |
<non_alphanumeric_char>

This section contains helper non-terminals to make it easier to define datatypes. It outlines the grammars for numbers, identifiers, constants, variables, entities, data, letters, etc. It also outlines the naming conventions used in “.FAM” to name variables, constants, functions, etc.

- **Symbols**

<newline> ::= \n

<underscore> ::= _

<space> ::=

<upper_case_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

<lower_case_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

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

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

<conditional_op> ::= > | < | <= | >= | equals | and | or | not | xor | nand | nor

<non_alphanumeric_char> ::= ! | " | # | $ | % | & | \ | ' | (|) | * | + | , | - | . | / | : | ; | < | = |
> | ? | @ | [| \ | \ |] | ^ | _ | ` | { | \| | } | ~ | <newline>

This section of the BNF contains the most basic non-terminals used in “.FAM”. All of these non-terminals include terminals that are integral to the grammar of the language. These include frequently used symbols, amalgamations of symbols to define the boundaries of what letters, numbers, arithmetic, conditional operators, etc can be.

- **Conditional Statement**

<if_statement> ::= <if_condition> <statement_list> ENDIF
| <open_if_statement> endif |
<open_if_statement> else \n <statement_list> ENDIF
<open_if_statement> ::= <if_condition> <statement_list> | <open_if_statement>
<elif_condition> <statement_list>
<if_condition> ::= IF (<boolean_expression>) \n
<elif_condition> ::= ELSEIF(<boolean_expression>) \n

The non-terminal <if_statement> highlights the grammar of the conditional if statement in ".FAM". Every if statement starts with the keyword "IF" as outlined by the non-terminal <if_condition>. Each statement also ends with the keyword "ENDIF". This makes the language more readable and writable due to the lack of brackets and braces, etc.
<if_condition> also contains a boolean expression after the if keyword that can either be true or false, followed by a newline character. The non_terminal <open_if_statement> is a helper non-terminal designed to define the grammar for different types of if statements, with and without elseif/else statements. After every if/elseif/else condition, there is a list of statements that will only be compiled if the boolean expression pertaining to each statement results in true.

- **Loops**

<iteration_statement> :: <for_loop> | <while_loop>
<for_loop> ::= FOR (<assignment_initialize_var> , <boolean_expression> ,
<assignment_expression>) \n <statement_list> ENDFOR
<while_loop> ::= WHILE (<boolean_expression>) \n <statement_list> ENDWHILE

The non_terminal <iteration_statment> outlines the two types of loops that can be used in ".FAM": namely a for loop, and a while loop.
The non-terminal <for_loop> defines the grammar for a for loop, starting and ending with the keywords "FOR" and "ENDFOR". These keywords help to define clear boundaries of when a loop is starting and when it is ending, increasing its readability, writability, and even reliability. A for loop contains, after the starting keyword, three statements in parenthesis: an assignment and/or initialization, a boolean expression that keeps looping the program until it is false, and an expression to alter the value previously initialized. These statements are used to define the parameters of the loop so it can stop when needed

and does not run forever. After these statements is a newline character, after which follows a list of statements that will be iterated over, ending with the closing keyword.
The non-terminal <while_loop> defines the grammar for a while loop, starting and ending with the keywords “WHILE and ENDWHILE”. These keywords help to define clear boundaries of when a loop is starting and when it is ending, increasing its readability, writability, and even reliability. A while loop contains, after the starting keyword, a boolean expression enclosed within parentheses which is used as the parameter for the loop. The loop will keep iterating over the statements present inside the loop until that condition turns to false.

- **In/Out statements**

<input_call> ::= input (<identifier>)
<output_call> ::= output (<data>)

This section of the BNF outlines the grammar for two important functions that can take input from the user and output data to the user. The non-terminals <input_call> and <output_call> define the grammar for these functions. The input function is called like every other function previously defined, and contains a variable as its argument. Once the data from the user is received, it is stored in that variable, provided that the datatypes match. The output function takes some data as its argument, could be a variable or a constant, or just a literal, and prints it to the terminal.

- **Expression**

<expression> ::=<boolean_expression> |<assignment_expression>
|<arithmetic_expression>
<boolean_expression> ::= <data><conditional_op><data> | <entity>
|<boolean_expression><conditional_op><data>
|<data><conditional_op><boolean_expression> | <TrueOrFalse>
|<boolean_expression><conditional_op><boolean_expression>
<assignment_expression> ::= <identifier>= <identifier> | <identifier> = <null>
|<identifier>=<arithmetic_expression>
|<identifier>=<input_call> | <identifier> = <function_call>
|<identifier>=<boolean_expression> |<identifier>= <data>
<arithmetic_expression> ::= <addition> | <subtraction> | <multiplication> | <division>
|<power> | <modulo>

The non-terminal <expression> highlights the three expressions used in “.FAM”: boolean, assignment, and arithmetic.
The non-terminal <boolean_expression> contains the grammar for a boolean expression. This is normally two pieces of data separated by a conditional operator, but it could also simply be an entity containing a value with the TrueOrFalse datatype.
The non-terminal <assignment_expression> defines the grammar for an assignment expression. This is normally the name of a variable or constant, followed by an equals sign, and then some data in any form: variables, constants, literals, function calls, and even null values.
The non-terminal <arithmetic_expression> defines the types of different operations possible: addition, subtraction, multiplication, division, power, modulo, and any permutation of any or all of those together.

- **Arithmetic Operations**

```
<computable> ::= <number>  | <real> | <variable> | <function_call> |<constant>
<addition> ::= <computable> + <computable> | (<addition>)
             | <computable> + (<arithmetic_expression> )
             | (<arithmetic_expression>) + <computable>
             | (<arithmetic_expression>)  + (<arithmetic_expression>)
             | <string> + <string>  | <addition> + <addition>
             | <computable> + <addition>  | <addition> + <computable>
<subtraction> ::=(<subtraction>) | <computable> - <computable>
                | (<arithmetic_expression>) - <computable>
                | (<arithmetic_expression>) - (<arithmetic_expression>)
                | <computable> - (<arithmetic_expression>)
                | <subtraction> - <subtraction>  | <subtraction> - <computable>
                | <computable> - <subtraction>
<multiplication> ::= <computable> * <computable>| (<multiplication>)
                   | <computable> * (<arithmetic_expression>)
                   |(<arithmetic_expression>) * <computable>
                   |(<arithmetic_expression>)  * (<arithmetic_expression>)
                   | <multiplication> * <multiplication>
                   | <multiplication> - <computable> | <computable> - <multiplication>
<division> ::= <computable> / <computable> | <computable> / (<arithmetic_expression>)
             | (<arithmetic_expression>)   / <computable>
             |(<arithmetic_expression>)   / (<arithmetic_expression>)   | (<division>)
<power> ::= <computable> ^ <computable>  | <computable> ^ (<arithmetic_expression>)
```

```
                  | (<arithmetic_expression>)  ^ <computable>
                  |(<arithmetic_expression>)  ^ (<arithmetic_expression>)  | (<power>)
<modulo> ::= <computable> % <computable> | (<modulo>)
                  | <computable> % (<arithmetic_expression>)
                  | (<arithmetic_expression>)  % <computable>
                  |(<arithmetic_expression>)  % (<arithmetic_expression>)
```

This section of the BNF further elaborates every non-terminal present in <arithmetic_expression>. The non_terminal <computable> contains all the possible data that can be computed: an integer, a real, a variable, a constant, and even a function call. The rest of the non-terminals contain permutations of different data that can be computed, with added syntax to follow operator precedence and associativity rules. Each of these expressions can be used on a different arithmetic expression, hence the need for that. Moreover, the non-terminal <addition> also contains an option for strings to be concatenated together.

The language uses parentheses in a controlled manner so that arithmetic operations are readable, reliable, and easy to write and edit. This way, the program's precedence will be entered by the program so that every operation is explicit. For example, in .FAM, we do not allow expressions like 6/9^7/7+8, which make the operation hard to understand and is also unreliable. Instead, the user needs to use parenthesis for such cases, for example, ((6/9)^(7/7))+8. However, the user does not have to add parenthesis after every operation (to maintain writability). This means that .FAM allows expressions like 3+4+5 or 5*7*9.

2. Lexical Analysis

```
%option main
startProgram      STARTPROGRAM
endprogram                  ENDPROGRAM
prim_func_call  PRIM
prim_func
      read_temp|read_hum|read_air_qual|read_air_press|read_light|read_sound
      _lvl|read_timestamp|set_switch_as
conditional_op  EQUALS|AND|OR|NOT|XOR|NAND|NOR
fun_dec                     FUNC
return_stmt                 RETURN
end_func                    ENDFUNC
assign                      =
newline           \n
```

```
null_type		NULL
int_type		Integer
char_type		Char
string_type		String
real_type		Real
boolean_type	TrueOrFalse
date_type		Date
time_type		Time
connection_type		Connection
connection_func		connect_to_url|disconnect|fetch_integer|send_integer
datetime_type	DateTime
void_type		Void
if_stmt		IF
else_stmt		ELSE
elseif_stmt	ELSEIF
endif			ENDIF
for_stmt		FOR
endfor_stmt		ENDFOR
while_stmt		WHILE
endwhile_stmt	ENDWHILE
input			input
output			output
true		TRUE
false		FALSE
digit		[0-9]
lower_case_let	[a-z]
upper_case_let	[A-Z]
letter			{upper_case_let}|{lower_case_let}
underscore		\_
identifier	{letter}({letter}|{digit}|{underscore})*
constant	{underscore}{identifier}
func_call	{identifier}()
string_stmt	\"(.)*\"
bool_stmt		{true}|{false}
int_stmt	[-]?{digit}+
real_stmt		[-]?{digit}*(\.)?{digit}+
char_stmt		\'{letter}\'
date_stmt		{digit}{digit}-{digit}{digit}-{digit}{digit}{digit}{digit}
time_stmt		{digit}{digit}:{digit}{digit}:{digit}{digit}
datetime_stmt	{date_stmt}\ {time_stmt}
comment			#([^#])*#

%%
{null_type}				printf("NULL_VALUE ");
\!					printf("NOT ");
\.					printf("DOT ");
\,		printf("COMMA ");
\(		printf("LP ");
```

```
\)                      printf("RP ");
\{                      printf("LCB ");
\}                      printf("RCB ");
\;                      printf("SEMICOLON ");
\=                      printf("ASSIGN_OP ");
\+                              printf("ADD ");
\-                              printf("SUBTRACT ");
\*                                      printf("MULTIPLY ");
\/                                      printf("DIVIDE ");
\%                                      printf("MODULO ");
\>                                      printf("GREATER ");
\<                                      printf("LESS ");
\<=                                     printf("LESS_EQUAL ");
\>=                                     printf("GREATER_EQUAL ");
\^                                      printf("POWER");
{conditional_op}        printf(" %s ", yytext);
{startProgram}          printf("START_PROGRAM ");
{endprogram}            printf("END_PROGRAM ");
{prim_func_call}        printf("PRIMITIVE_FUNCTION_CALL ");
{prim_func}                     printf(" %s ", yytext);
{fun_dec}                       printf("FUNCTION_DECLARATION ");
{end_func}                      printf("END_FUNCTION_DECLARATION ");
{newline}                       printf("NEW_LINE \n");
{int_type}                      printf("INTEGER ");
{real_type}                     printf("REAL ");
{string_type}           printf("STRING ");
{char_type}                     printf("CHAR ");
{boolean_type}          printf("BOOLEAN ");
{date_type}             printf("DATE ");
{time_type}             printf("TIME ");
{datetime_type}         printf("DATETIME ");
{connection_type}       printf("CONNECTION ");
{connection_func}       printf(" %s ", yytext);
{if_stmt}                       printf("IF ");
{endif}                         printf("END_IF ");
{else_stmt}                     printf("ELSE ");
{elseif_stmt}           printf("ELSE_IF ");
{for_stmt}                      printf("FOR ");
{endfor_stmt}           printf("END_FOR ");
{while_stmt}            printf("WHILE ");
{endwhile_stmt}                 printf("END_WHILE ");
{true}                          printf("TRUE ");
{false}                         printf("FALSE ");
{input}                         printf("INPUT ");
{output}                printf("OUTPUT ");
{void_type}                     printf("VOID ");
{constant}                      printf("CONSTANT ");
{int_stmt}                      printf("INTEGER_VALUE ");
```

```
{real_stmt}             printf("REAL_VALUE ");
{char_stmt}                     printf("CHAR_VALUE ");
{string_stmt}           printf("STRING_VALUE ");
{comment}                       printf("COMMENT ");
{return_stmt}           printf("RETURN_TYPE");
{date_stmt}                     printf("DATE_VALUE ");
{time_stmt}                     printf("TIME_VALUE ");
{datetime_stmt}                 printf("DATETIME_VALUE ");
{identifier}            printf("IDENTIFIER ");
```

3. Example Programs

```
STARTPROGRAM

    #
    This is a
    multiple line comment
    #

    # This is a single line comment #

    output("Hello world")

    Integer age
    String prompt = "Age saved successfully"
    output("Enter your age = ")
    input(age)
    output(prompt)

    # Testing Functions and Initialization #

    Date todayDate = 14-08-1947
    Time todayTime = 12:30:07
    DateTime todayDateTime = 14-08-1947 12:30:07

    Real _maxProduct = 50
    Connection server
    Connection database
    Real temperature = PRIM.read_temp()
    Real humidity
    humidity = PRIM.read_hum()

    server.connect_to_url("www.allnighters.com")
    String databaseURL = "www.database.com"
    database.connect_to_url(databaseURL)
```

```
server.send_integer(PRIM.read_timestamp())

output(PRIM.read_timestamp())
output(todayDateTime)

Real product = calculateProduct(temperature , humidity)

IF ( product > _maxProduct )
        warnUser()
        database.send_integer(4)
ENDIF

FUNC Void warnUser()
        output("Warning Temperature and Humidity product is too high")
        PRIM.set_switch_as(4, FALSE)
        RETURN NULL
ENDFUNC

FUNC Real calculateProduct(Real x, Real y)
        Real result = x * y

        IF (x < 0)
                RETURN -1
        ENDIF

        RETURN result
ENDFUNC

server.disconnect()
database.disconnect()


# TESTING ASSIGNMENTS #

Integer age = 23
Integer minTemp = -45
Real realData = 32.123
Char char1 = 'A'
String str1 = "Hello World!"
TrueOrFalse testFalse = FALSE
TrueOrFalse testTrue = TRUE
Real currentTemp = PRIM.read_temp()


# TESTING ARITHMETIC OPERATIONS #

Integer data3 = age + minTemp
Real realData2 = realData * -34.21
```

```
Integer data4 = 12 + 23 + 21 + 2
data4 = 12 + 23 + (21 * 2)
Integer data5 = (data4 * 2) + data1
Real realData3 = realData2 ^ 0.21
realData3 = realData3 / 2.4
realData3 = ((45 / 2.4) * 9) + 6


# TESTING IF STATEMENTS #

IF(realData2 > realData)
        ELSEIF(data1 > data2)
                ELSEIF(data3 > data2 AND data1 EQUALS 23)
                        IF(data3 <= 123)
                                output("data3 is less
                        than 123")
                        ENDIF
                        output("Correct")
ELSE
        output("incorrect")
ENDIF


# TESTING LOOPS #


FOR (Integer i = 0 , i <= 100 , i = i + 1)
        Integer j = 1
        WHILE (j EQUALS 1)
                output("Valid")
                j = j - 1
        ENDWHILE
        output("Input: ")
        input(j)
        multipleOutput()
ENDFOR

FUNC Void multipleOutput()
        TrueOrFalse countinue = TRUE
        Integer counter = 0
        WHILE (countinue)
                counter = counter + 1
                output("Warning Temperature and Humidity product
            is too high")

                IF (counter > 9)
                        countinue = FALSE
                ENDIF
```

```
        ENDWHILE
        RETURN NULL
ENDFUNC

ENDPROGRAM
```

4. Output for Example Program

```
amir@Amirs-MacBook-Air Project 1 % cat mainSample.txt | ./a
START_PROGRAM NEW_LINE
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
OUTPUT LP STRING_VALUE RP NEW_LINE
NEW_LINE
INTEGER  IDENTIFIER  NEW_LINE
STRING  IDENTIFIER  ASSIGN_OP  STRING_VALUE NEW_LINE
OUTPUT LP STRING_VALUE RP NEW_LINE
INPUT LP IDENTIFIER RP NEW_LINE
OUTPUT LP IDENTIFIER RP NEW_LINE
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
DATE  IDENTIFIER  ASSIGN_OP  DATE_VALUE NEW_LINE
TIME  IDENTIFIER  ASSIGN_OP  TIME_VALUE NEW_LINE
DATETIME  IDENTIFIER  ASSIGN_OP  DATETIME_VALUE NEW_LINE
NEW_LINE
REAL  CONSTANT  ASSIGN_OP  INTEGER_VALUE NEW_LINE
CONNECTION  IDENTIFIER NEW_LINE
CONNECTION  IDENTIFIER NEW_LINE
REAL  IDENTIFIER  ASSIGN_OP  PRIMITIVE_FUNCTION_CALL DOT  read_temp LP RP NEW_LINE
REAL  IDENTIFIER  NEW_LINE
IDENTIFIER  ASSIGN_OP  PRIMITIVE_FUNCTION_CALL DOT  read_hum LP RP NEW_LINE
NEW_LINE
IDENTIFIER DOT  connect_to_url LP STRING_VALUE RP NEW_LINE
STRING  IDENTIFIER  ASSIGN_OP  STRING_VALUE NEW_LINE
IDENTIFIER DOT  connect_to_url LP IDENTIFIER RP NEW_LINE
NEW_LINE
IDENTIFIER DOT  send_integer LP PRIMITIVE_FUNCTION_CALL DOT  read_timestamp LP RP RP NEW_LINE
NEW_LINE
OUTPUT LP PRIMITIVE_FUNCTION_CALL DOT  read_timestamp LP RP RP NEW_LINE
OUTPUT LP IDENTIFIER RP NEW_LINE
NEW_LINE
REAL  IDENTIFIER  ASSIGN_OP  IDENTIFIER LP IDENTIFIER  COMMA  IDENTIFIER RP NEW_LINE
NEW_LINE
IF  LP  IDENTIFIER  GREATER  CONSTANT  RP NEW_LINE
        IDENTIFIER LP RP NEW_LINE
        IDENTIFIER DOT  send_integer LP INTEGER_VALUE RP NEW_LINE
END_IF NEW_LINE
NEW_LINE
FUNCTION_DECLARATION  VOID  IDENTIFIER LP RP NEW_LINE
        OUTPUT LP STRING_VALUE RP       NEW_LINE
        PRIMITIVE_FUNCTION_CALL DOT  set_switch_as LP INTEGER_VALUE COMMA  FALSE RP     NEW_LINE
        RETURN_TYPE NULL_VALUE NEW_LINE
END_FUNCTION_DECLARATION NEW_LINE
NEW_LINE
FUNCTION_DECLARATION  REAL  IDENTIFIER LP REAL  IDENTIFIER COMMA  REAL  IDENTIFIER RP NEW_LINE
        REAL  IDENTIFIER  ASSIGN_OP  IDENTIFIER  MULTIPLY  IDENTIFIER NEW_LINE
NEW_LINE
        IF  LP IDENTIFIER  LESS  INTEGER_VALUE RP NEW_LINE
                RETURN_TYPE INTEGER_VALUE       NEW_LINE
        END_IF          NEW_LINE
NEW_LINE
        RETURN_TYPE IDENTIFIER  NEW_LINE
END_FUNCTION_DECLARATION NEW_LINE
NEW_LINE
IDENTIFIER DOT  disconnect LP RP NEW_LINE
IDENTIFIER DOT  disconnect LP RP NEW_LINE
NEW_LINE
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE NEW_LINE
INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE NEW_LINE
REAL  IDENTIFIER  ASSIGN_OP  REAL_VALUE NEW_LINE
CHAR  IDENTIFIER  ASSIGN_OP  CHAR_VALUE NEW_LINE
```

Project 1 — -zsh — 239×70

```
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE NEW_LINE
INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE NEW_LINE
REAL  IDENTIFIER  ASSIGN_OP  REAL_VALUE NEW_LINE
CHAR  IDENTIFIER  ASSIGN_OP  CHAR_VALUE NEW_LINE
STRING  IDENTIFIER  ASSIGN_OP  STRING_VALUE NEW_LINE
BOOLEAN  IDENTIFIER  ASSIGN_OP  FALSE NEW_LINE
BOOLEAN  IDENTIFIER  ASSIGN_OP  TRUE NEW_LINE
REAL  IDENTIFIER  ASSIGN_OP  PRIMITIVE_FUNCTION_CALL DOT  read_temp LP RP NEW_LINE
NEW_LINE
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
INTEGER  IDENTIFIER  ASSIGN_OP  IDENTIFIER  ADD  IDENTIFIER  NEW_LINE
REAL  IDENTIFIER  ASSIGN_OP  IDENTIFIER  MULTIPLY  REAL_VALUE NEW_LINE
INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE  ADD  INTEGER_VALUE  ADD  INTEGER_VALUE  ADD  INTEGER_VALUE NEW_LINE
IDENTIFIER  ASSIGN_OP  INTEGER_VALUE  ADD  INTEGER_VALUE  ADD  LP INTEGER_VALUE  MULTIPLY  INTEGER_VALUE RP NEW_LINE
INTEGER  IDENTIFIER  ASSIGN_OP  LP IDENTIFIER  MULTIPLY  INTEGER_VALUE RP  ADD  IDENTIFIER NEW_LINE
REAL  IDENTIFIER  ASSIGN_OP  IDENTIFIER  POWER REAL_VALUE NEW_LINE
IDENTIFIER  ASSIGN_OP  IDENTIFIER  DIVIDE  REAL_VALUE NEW_LINE
IDENTIFIER  ASSIGN_OP  LP LP INTEGER_VALUE  DIVIDE  REAL_VALUE RP  MULTIPLY  INTEGER_VALUE RP  ADD  INTEGER_VALUE NEW_LINE
NEW_LINE
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
IF LP IDENTIFIER  GREATER  IDENTIFIER RP NEW_LINE
        ELSE_IF LP IDENTIFIER  GREATER  IDENTIFIER RP NEW_LINE
                ELSE_IF LP IDENTIFIER  GREATER  IDENTIFIER   AND  IDENTIFIER   EQUALS  INTEGER_VALUE RP NEW_LINE
                        IF LP IDENTIFIER  LESS_EQUAL  INTEGER_VALUE RP NEW_LINE
                                OUTPUT LP STRING_VALUE RP NEW_LINE
                        END_IF NEW_LINE
                        OUTPUT LP STRING_VALUE RP NEW_LINE
ELSE NEW_LINE
        OUTPUT LP STRING_VALUE RP NEW_LINE
END_IF NEW_LINE
NEW_LINE
NEW_LINE
COMMENT NEW_LINE
NEW_LINE
NEW_LINE
FOR  LP INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE  COMMA  IDENTIFIER  LESS_EQUAL  INTEGER_VALUE  COMMA  IDENTIFIER  ASSIGN_OP  IDENTIFIER  ADD  INTEGER_VALUE RP NEW_LINE
        INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE NEW_LINE
        WHILE  LP IDENTIFIER   EQUALS  INTEGER_VALUE RP NEW_LINE
                OUTPUT LP STRING_VALUE RP NEW_LINE
                IDENTIFIER  ASSIGN_OP  IDENTIFIER  SUBTRACT  INTEGER_VALUE NEW_LINE
        END_WHILE NEW_LINE
        OUTPUT LP STRING_VALUE RP NEW_LINE
        INPUT LP IDENTIFIER RP NEW_LINE
        IDENTIFIER LP RP NEW_LINE
END_FOR NEW_LINE
NEW_LINE
FUNCTION_DECLARATION  VOID  IDENTIFIER LP RP NEW_LINE
        BOOLEAN  IDENTIFIER  ASSIGN_OP  TRUE NEW_LINE
        INTEGER  IDENTIFIER  ASSIGN_OP  INTEGER_VALUE NEW_LINE
        WHILE  LP IDENTIFIER RP NEW_LINE
                IDENTIFIER  ASSIGN_OP  IDENTIFIER  ADD  INTEGER_VALUE NEW_LINE
                OUTPUT LP STRING_VALUE RP       NEW_LINE
NEW_LINE
                IF  LP IDENTIFIER  GREATER  INTEGER_VALUE RP NEW_LINE
                        IDENTIFIER  ASSIGN_OP  FALSE NEW_LINE
                END_IF NEW_LINE
                NEW_LINE
        END_WHILE NEW_LINE
        RETURN_TYPE NULL_VALUE NEW_LINE
END_FUNCTION_DECLARATION NEW_LINE
NEW_LINE
END_PROGRAM %
amir@Amirs-MacBook-Air Project 1 %
```