VirtualoT

CS 315 PROJECT 1

Maryam Azimli 22101528 Sec: 1
Özgür İkidağ 22102315 Sec: 1
Mennatallah Abouelenin 22101539 Sec: 1

TEAM: 12

Part A: BNF for Programming Language VirtuaIoT

<program> ::= <stmt_list>

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

Since there are several types of statements, the statement list consists of a single statement or several statements next to each other. Technically every program is a statement list.

<expr> ::= <arithmetic_expr> | <boolean_expr> | <sensor_expr> | <timer_expr>

Expressions are used to initialize variables mostly. Boolean expressions are used for if statements.

<sensor_expr> ::= sensor.<sensor_type>();

<sensor_type> ::= temperature | humidity | airPressure | airQuality | light
| soundLevel

<timer_expr> ::= timer.getYear(); | timer.getMonth(); | timer.getDay(); | timer.getHour(); | timer.getMinute(); |timer.getSecond(); | timer.getFull();

The sensor and timer expressions are enablers for the specific use of the IoT device. Sensor and timer keywords are default for our programming language.

<arithmetic_expr> ::= <arithmetic_expr> + <mult_expr> | <arithmetic_expr> - <mult_expr> | <mult_expr>

Arithmetic expressions include 6 operations: 4 main arithmetic operations plus the modulus calculation(%) and exponent calculation. We, of course, take into consideration the operation precedence (pemdas).

<mult_expr> ::= <mult_expr> * < expo_term> | <mult_expr> / < expo_term >

| <mult_expr> % <expo_term > | <expo_term>

Since mult_expressions come before addition and subtraction, we do multiplication, division, modulus, and exponent operations here. Exponential calculations always come first; that's why it has its own abstraction.

<expo_term> ::= <expo_ term> ^ <term> | <term>

<term> ::= (<arithmetic_expr>) | <var_id> | <const> |<conn_read_stmt>

Term is the smallest piece of expression. It can be a constant, a variable, or a member of an array. From this hierarchy, one can do all arithmetic calculations.

<boolean_expr> ::= <boolean_expr> || <bool_expo> | <bool_expo>

Boolean expressions are used for if conditions. Since NOT comes before AND, and AND comes

before OR, it has its own abstraction. It can also be a comparison expression itself.

<bool_expo> ::= <bool_expo> && <bool_term> | <bool_not>

<bool_not>::= ! <bool_term>|<bool_term>

<bool_term> ::= (<boolean_expr>) | <comparison_expr>

Bool expo consists of an AND operation or a bool term. Bool term is the smallest piece of a boolean expression. Comparison expression is also a bool term.

Bool not has higher precedence than && and ||.

<comp_operator> ::= < | > | <= | >= | =:= | !=:=

<comparison_expr> ::= <arithmetic_expr> <comp_operator> <arithmetic_expr>

Comparison expression is the same as any C-type language. Two arithmetic expressions are compared by one of the comparison operators.

stmt can be an if, else_if, else or non_if. non_if feautrues all the other statements. having k rule helps us to have one or more else ifs.

<stmt>::=stmt: if (<boolean_expr>) [<stmt_list>] <k> else [<stmt_list>]
| if (<boolean_expr>) [<stmt_list>] else [<stmt_list>]
| if (<boolean_expr>) [<stmt_list>]| <non_if>
;

k::= else_if (<boolean_expr>) [<stmt_list>] | else_if (<boolean_expr>) [<stmt_list>] <k>;

loop_stmt::= five_stmt | white_stmt
;

<non_if>::=<declaration_stmt> |<assignment_stmt> | <loop_stmt> | <conditional_stmt> | <function_def_stmt> | <function_call_stmt> | <input_stmt> | <output_stmt> | <return_stmt> | <comment> | <conn_declaration_stmt>|<conn_send_stmt> | <switch_set_stmt>;

There are many different statements in our language. Each is explained below.

A separate statement type for non-if statements. It allows us to define if statements more clearly.

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

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

<digit_list> ::= <digit> | <digit> <digit_list>

We wanted to make sure that numbers cannot start with zero, so we have two separate digit types.

<const> ::= 0 | <positive_const> | <negative_const>

<positive_const> ::= <nonzero_digit> | <nonzero_digit> <digit_list> | +<nonzero_digit> | +<nonzero_digit> <digit_list>

<negative_const> ::= -<nonzero_digit> | -<nonzero_digit> <digit_list>

Consts are integer literals. They can also be used to construct variable names.

<const_list> ::= <const> | <const> , <const_list>

This is a list of multiple integers separated by commas. Mainly used for array initializations.

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

<ident_list>::= <var_id> | <var_id> , <ident_list>

Ident list is for declaration statements of several variables.

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

Char list is for naming variables. Note that it doesn't contain special characters except for "_".

<var_id> ::= <char_list>

<out_char> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | r | s | t | u | v | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | R | S | T | U | V | Y | Z | _ | ^ | + | % | & | | | ! | " | ' | (|) | * | , | - | . | / | : | ; | < | = | > | ? | [|] | { | } | \ | ` | ~ | |

<url> ::= https://<out_list> | http://<out_list>

Url format is used in connection functions. It is basically a set of characters with specific starter string.

<out_list> ::= <out_list> | <out_char> <out_list>

Our list contains possible combinations of chars of outputs or comments.

<declaration_stmt> ::= var <ident_list> ;

Declaring several variables at once.

<assignment_stmt> ::= var <var_id> = <expr> ; | <var_id> = <expr> ;

Users can assign values to predefined variables or define and initialize them at the same time.

<conn_declaration_stmt> ::= connection = connectionCreate(“<url>”) ;

Connection is a built-in default keyword in our programming language. It allows the IoT device to define a connection with a given url.

<conn_read_stmt> ::= var <var_id> = connection.Read() ;

Connection.read enables us to receive the value of the existing integer in the connection that has been defined earlier.

<conn_send_stmt> ::= connection.Send(<var_id>) ; |connection.Send(<const>) ;

Connection.send enables us to post the value of an integer variable or a constant to the connection that has been defined earlier.

<switch_set_stmt> ::= <switch_type>.on(); | <switch_type>.off();

<switch_type> ::= lights | heater | airConditioner | switch4 | switch5 | switch6 | switch7 | switch8 | switch9 | switch10

<loop_stmt> ::= <five_stmt> | <white_stmt>

We have two loop statements, five and white.

<five_stmt> ::= five (<assignment_stmt> <boolean_expr> ; <var_id> = <arithmetic_expr>) [<stmt_list>]

Inside five statements, there must be one assignment statement, one boolean expression, and another assignment statement at the end. After that, another statement list comes between square brackets, just like a for loop.

<white_stmt> ::= white (<boolean_expr>) [<stmt_list>]

Our white loop acts like a while loop. It takes a boolean expression and runs the statement list inside it until the boolean is false.

<conditional_stmt> ::= <paired> | <unpaired>

For our conditional statements, we used the algorithm in the book to make sure there is no room for ambiguity. It uses “paired” system to pair every “else” statement with an “if” statement, which has two possibilities for paired and unpaired situations:

<paired> - meaningful statement(s) inside “if” is paired & meaningful statement(s) inside “else” is paired OR it is not a conditional stmt.

<unpaired> - there is only one meaningful statement inside “if” OR meaningful statement(s) inside “if” is paired & meaningful statement(s) inside “else” is unpaired.

<function_def_stmt> ::= void <var_id>(var<ident_list>) [stmt_list] | int <var_id>(var<ident_list>) [stmt_list] | void <var_id>() [stmt_list] | int <var_id>() [stmt_list]

In order to define a function in VirtuaIot, the user has to indicate the return value first, or void. Then they have to write the function name. Lastly, they have to specify the parameters. If there

is no parameter, the parenthesis is simply left empty.

<function_call_stmt> ::= <var_id>(<ident_list>) ; | <var_id>() ;

Calling a function is the same as in any C-like language.

<input_stmt> ::= scan -> <var_id> ;

Taking an input is as shown.

<output_stmt> ::= out (“<out_list>”) ; | out (“”) |out(<var_id>) ; | outln(“<out_list>”); | outln(“”) ; | outln(<var_id>) ;

To print a statement, writing “out” and putting the string inside is enough.

<return_stmt> ::= turn <var_id> ; | turn <const> ;

A function returns the desired value by writing a turn before the integer value.

<comment> ::= **<out_list>**

*Comments are written inside ** ** asterisks.*

In test 3, there are errors and comments on how to fix those errors. Please follow them.

As well as in program2, named 5.txt, has errors with comments on how to fix them.