CS 315 Programming Languages

PROJECT 1

A Programming Language for IoT Devices and its Lexical Analyzer

Group 28

Mert Aslan 21702818-Section 2
Rahmiye Büşra Büyükgebiz 21702019-Section 2

Yenn Language

BNF DESCRIPTION

<program> ::= START <stmts> FINISH

<stmts> ::= <stmts><stmt> | <stmt>

<stmt> ::= <matched_stmt> | <unmatched_stmt>

<matched_stmt> ::= if (<expression>) <matched_stmt> else <matched_stmt> | <non_if_stmt> | <loop_stmt> <matched_stmt>

<unmatched_stmt> ::= if (<expression>) <stmt> | if (<expression>) <matched_stmt> else <unmatched_stmt> | <loop_stmt> <unmatched_stmt>

<non_if_stmt> ::= <assign_stmt> | <var_declaration> | <comment_string> | <func_call_stmt> | <func_declaration> | <return_stmt> | <control_switches> | <code_block>

<non_if_stmt_list> ::= <non_if_stmt_list> <stmt> | <stmt>

<assign_stmt> ::= <lside> = <rside>

<rside> ::= <expression>

<lside> ::= <identifier> | <var_declaration>

<expression> ::= <or_expression>

<or_expression> ::= <or_expression> || <and_expression> | <and_expression>

<and_expression> ::= <and_expression> && <comparison_expression> | <comparison_expression>

<comparison_expression> ::= <comparison_expression><comparator><add_sub_expression> | <add_sub_expression>

<add_sub_expression> ::= <add_sub_expression> + <mul_div_expression> | <add_sub_expression> - <mul_div_expression> | <mul_div_expression>

<mul_div_expression> ::= <mul_div_expression> * <pow_expression> | <mul_div_expression> / <pow_expression> | <pow_expression>

<**pow_expression>** ::= <pow_expression> ^ <element> | <element>

<element> ::= (<expression>) | <identifier> | <integer> | <double_value> | <array_init> | <string> | <func_call_stmt>

ARRAYS

<array_init> ::= [<array_elements>]

<array_elements> ::= <array_elements> , <array_element> | <array_element>

<array_element> ::= <integer> | <double_value> | <string> | <identifier>

DECLARATIONS

<var_declaration> ::= <type_id> <identifier> | array <identifier>

<func_declaration> ::= func <identifier> (<parameter_list>?)

LOOPS

<loop_stmt> ::= <for_stmt> | <while_stmt>

<for_stmt> ::= for (<identifier> in <identifier>) | for (<assign_stmt> where <expression> with <assign_stmt>) | for (<assign_stmt> where <expression>)

<while_stmt> ::= while (<expression>)

FUNCTIONS

<func_call_stmt> ::= <primitive_func_call> | <non_primitive_func_call>

<primitive_func_call> ::= <read_temp> | <read_hum> | <read_air_p> | <read_air_q> | <read_light> | <read_sound_lvl> | <read_timestamp_from_timer> | <send_integer_to_connection> | <read_integer_from_connection> | <connect> | <scan> | <print>

<non_primitive_func_call> ::= call <identifier> (<parameter_list_on_call>?)

<parameter_list_on_call> ::= <rside> | <parameter_list_on_call> , <rside>

<return_stmt> ::= return <identifier> | return <double_value> | return <integer> | return <string> | return <array_init>

<parameter_list> ::= <var_declaration> | <parameter_list> , <var_declaration>

<scan> ::= scan() | scan(<string>)

<print> ::= print(<identifier>) | print(<integer>) | print(<string>) | print(<double_value>) | print(<array_init>)

<read_temp> ::= read_temp**()**

<read_hum> ::= read_humidity**()**

<read_air_p> ::= read_air_pressure**()**

<read_air_q> ::= read_air_quality**()**

<read_light> ::= read_light**()**

<read_sound_lvl> ::= read_sound_level(<double_value>) | read_sound_level(<identifier>)

<read_timestamp_from_timer> ::= read_timer**()**

<send_integer_to_connection> ::= send_int(<identifier> , <identifier>) | send_int(<identifier> , <integer>)

<read_integer_from_connection> ::= read_int(<identifier>)

<connect> ::= connect(<identifier>) | connect(<string>)

SWITCHES

<control_switches> ::= <turn_on> | <turn_off>

<turn_on> ::= SwitchON1 | SwitchON2 **|** SwitchON3 | SwitchON4 | SwitchON5 | SwitchON6 | SwitchON7 | SwitchON8 | SwitchON9 | SwitchON10

<turn_off> ::= SwitchOFF1 | SwitchOFF2 **|** SwitchOFF3 | SwitchOFF4 | SwitchOFF5 | SwitchOFF6 | SwitchOFF7 | SwitchOFF8 | SwitchOFF9 | SwitchOFF10

DEFINITIONS

<code_block> ::= // <stmts>\\

<type_id> ::= int | double | string | connection

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

<double_value> ::= <integer><dot><integer>

<string> ::= “<string_characters>”

<comment_string> ::= #<string_characters>#

<string_characters> ::= <character> | <string_characters> <character>

<character> ::= <lowercase_letter> | <uppercase_letter> | <digit> | <symbols> | SPACE

<identifier> ::= <lowercase_letter> | <identifier><lowercase_letter> | <identifier> _ | <identifier><digit>

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

<lowercase_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

<uppercase_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

<symbols> ::= ! | ‘ | ^ | # | + | $ | % | & | / | (| [|) |] | = | ? | * | \ | - | “ | | | : | . | , |

<comparator> ::= > | < | >= | <= | ==

LANGUAGE EXPLANATION

- **<program> ::= START <stmts> FINISH**
  Start point of the program. Whole code is encapsulated between START and FINISH statements.

- **<stmts> ::= <stmts><stmt> | <stmt>**
  Statements can either be a single statement, or multiple statements brought together by recursion

- **<stmt> ::= <matched_stmt> | <unmatched_stmt>**
  A statement can either be a matched statement or an unmatched statement. This is done in order to prevent mismatched or un-matched if else statements in the program.

- **<matched_stmt> ::= if (<expression>) <matched_stmt> else <matched_stmt> | <non_if_stmt> | <loop_stmt> <matched_stmt>**
  This nonterminal makes sure that all nested if else statements are matched, meaning that every if has a corresponding else statement.

- **<unmatched_stmt> ::=if (<expression>) <stmt> | <if> (<expression>) <matched_stmt> else <unmatched_stmt> | <loop_stmt> <unmatched_stmt>**
  This nonterminal, with the <matched_stmt> nonterminal, makes it so that there cannot be an “elseless” if statement nested inside an if-else statement.

- **<non_if_stmt> ::= <assign_stmt> | <var_declaration> | <comment_string> | <func_call_stmt> | <func_declaration> | <return_stmt> | <control_switches> | <code_block>**
  These are the main functionality of the program. User can call any of these statements, declarations or comment in a single line.

- **<assign_stmt> ::= <lside> = <rside>**
  Assign statement has a left side and a right side.

- **<rside> ::= <expression>**
  Right side of an assignment derives to expression.

- **<lside> ::= <identifier> | <var_declaration>**
  This is the left side of an assign statement. With the <identifier> nonterminal, it enables the user to assign a value to an already declared variable. With the <var_declaration> nonterminal, it enables the user to declare and initialize a variable in the same line.

- **<expression> ::= <or_expression>**
  Expression derives into an or_expression.

- **<or_expression> ::= <or_expression> || <and_expression> | <and_expression>**
  The expression can either be an “or” expression, or continue down along the expression tree. This makes it so that “or” operator has the lowest precedence among all the operators.

- **<and_expression> ::= <and_expression> && <comparison_expression> | <comparison_expression>**
  The expression can either be an “and” expression, or continue down along the expression tree. This makes it so that “and” operator has more precedence than “or” operator.

- **<comparison_expression> ::= <comparison_expression><comparator><add_sub_expression> | <add_sub_expression>**
  The expression can either be a comparator, or continue down along the expression tree. This makes it so that a comparison operators have more precedence than the “and” operator.

- **<add_sub_expression> ::= <add_sub_expression> + <mul_div_expression> | <add_sub_expression> - <mul_div_expression> | <mul_div_expression>**
  The expression can either be an addition or subtraction, or continue down along the expression tree. This makes it so that addition and subtraction operators have more precedence than comparison operators.

- **<mul_div_expression> ::= <mul_div_expression> * <pow_expression> | <mul_div_expression> / <pow_expression> | <pow_expression>**
  The expression can either be a multiplication or division, or continue down along the expression tree. This makes it so that multiplication and division operators have more precedence than addition and subtraction operators.

- **<pow_expression> ::= <pow_expression> ^ <element> | <element>**
  The expression can either be a power operation, or continue down along the expression tree. This makes it so that power operator have more precedence than multiplication and division operators.

- **<element> ::= (<expression>) | <identifier> | <integer> | <double_value> | <array_init> | <string> | <func_call_stmt>**
  The expression can either be an arbitrary <expression> between parantheses, a variable which is expressed with <identifier>, an integer, a double value, an array initialization statement, a string initialization statement or a runction call statement. This nonterminal makes it so that an expression between parantheses has the most precedence among all other operators.

ARRAYS

- **<array_init> ::= [<array_elements>]**
  The user can initialize an array by encapsulating <array_elements> between square brackets.

- **<array_elements> ::= <array_elements> , <array_element> | <array_element>**
  Arrays can either have a single element, or multiple elements seperated by comma.

- **<array_element> ::= <integer> | <double_value> | <string> | <identifier>**
  An array element can be an integer, double, string or identifier.

DECLARATIONS

- **<var_declaration> ::= <type_id> <identifier> | array <identifier>**
  The user can declare either an array or a primitive variable. Primitive variables are constructed using a <type_id> followed by a <identifier>. Arrays are constructed using the “array” token followed by a <identifier>.

- **<func_declaration> ::= func <identifier> (<parameter_list>?)**
  This non-terminal defines the syntax of function declaration. Functions are declared with a terminal ‘func’. A function can take parameters optionally.

LOOPS

- **<loop_stmt> ::= <for_stmt> | <while_stmt>**
  Defines what a loop statement can be. It can be either for loop or while loop.

- **<for_stmt> ::= for (<identifier> in <identifier>) | for (<assign_stmt> where <expression> with <assign_stmt>)| for (<assign_stmt> where <expression>)**
  This non-terminal shows the syntax of for statement header. There are three ways to write for loop. The first way is by iterating a variable, the second way is by initializing a function, modifying its value each iteration and checking for the termination condition in each iteration, the third way is to only initializing a function and checking for the termination condition in each iteration.

- **<while_stmt> ::= while (<expression>)**
  This non-terminal shows the syntax of while statement header.

FUNCTIONS

- **<func_call_stmt> ::= <primitive_func_call> | <non_primitive_func_call>**
  This statement is used to call functions. Either primitive or non-primitive functions can be called.

- **<primitive_func_call> ::= <read_temp> | <read_hum> | <read_air_p> | <read_air_q> | <read_light> | <read_sound_lvl> | <read_timestamp_from_timer> | <send_integer_to_connection> | <read_integer_from_connection> | <connect> | <scan> | <print>**
  This statement is used to define primitive functions which are read temperature, read humidity, read air pressure, read air quality, read light, read sound level, read timestamp from timer, read integer from connection, send integer to connection, connect to the internet, scan input and print output.

- **<non_primitive_func_call> ::= <identifier> (<parameter_list_on_call>?)**
  This non-terminal shows the syntax of non-primitive function call statement. Function can be called by first using call token, then its name and parameters, though not necessarily, between left and right parenthesis. Call token is used to be able to differentiate between a primitive and non_primitive function call. If the call token did not exist, there would be an ambiguity in the language.

- **<parameter_list_on_call> ::= <rside> | <parameter_list_on_call> , <rside>**
  This non-terminal shows the syntax of parameter list when its used in function call statements.

- **<return_stmt> ::= return <identifier> | return <double_value> | return <integer> | return <string> | return <array_init>**
  This non-terminal defines the syntax of return statement. The syntax of the return statement is starting with the return token which simply is the reserved word for return. A function can return variable, double, integer, string and array.

- **<parameter_list> ::= <var_declaration> | <parameter_list> , <var_declaration>**
  This defines what parameter list can consist of. It can consist of either only variable name or more than one variable names separated by comma.

- **<scan> ::= scan() | scan(<string>)**
  This defines one of the primitive functions which is used to scan input.

- **<print> ::= print(<identifier>) | print(<integer>) | print(<string>) | print(<double_value>) | print(<array_init>)**
  This defines one of the primitive functions which is used to print output.

- **<read_temp> ::= read_temp()**
  This defines one of the primitive functions which is used to read temperature.

- **<read_hum> ::= read_humidity()**
  This defines one of the primitive functions which is used to read humidity.

- **<read_air_p> ::= read_air_pressure()**
  This defines one of the primitive functions which is used to read air pressure.

- **<read_air_q> ::= read_air_quality()**
  This defines one of the primitive functions which is used to read air quality.

- **<read_light> ::= read_light()**
  This defines one of the primitive functions which is used to read light.

- **<read_sound_lvl> ::= read_sound_level(<double_value>)| read_sound_level(<identifier>)**
  This defines one of the primitive functions which is used to read sound level. It takes either a double value or a variable name.

- **<read_timestamp_from_timer> ::= read_timer()**
  This defines one of the primitive functions which is used to read timestamp from timer.

- **<send_integer_to_connection> ::= send_int(<identifier> , <identifier>) | send_int(<identifier> , <integer>)**
  This defines one of the primitive functions which is used to send an integer at a time to a connection. It takes two parameters, a connection and an integer.

- **<read_integer_from_connection> ::= read_int(<identifier>)**
  This defines one of the primitive functions which is used to read an integer at a time from a connection. It takes connection as a parameter.

- **<connect> ::= connect(<identifier>) | connect(<string>)**
  This non-terminal defines the syntax of connection initialization.

SWITCHES

- **<control_switches> ::= <turn_on> | <turn_off>**
  User can either turn on the switches or turn them off. <control_switches> derives into both of these possibilities.

- **<turn_on> ::= SwitchON1 | SwitchON2 | SwitchON3 | SwitchON4 | SwitchON5 | SwitchON6 | SwitchON7 | SwitchON8 | SwitchON9 | SwitchON10**
  These tokens let the user turn on switches 1 to 10

- **<turn_off> ::= SwitchOFF1 | SwitchOFF2 | SwitchOFF3 | SwitchOFF4 | SwitchOFF5 | SwitchOFF6 | SwitchOFF7 | SwitchOFF8 | SwitchOFF9 | SwitchOFF10**
  These tokens let the user turn off switches 1 to 10

DEFINITIONS

- **<integer> ::= <digit> | <integer> <digit>**
  This non-terminal defines integer which can consist of one digit or more than one digits.

- **<double_value> ::= <integer><dot><integer>**
  This non-terminal defines double.

- **<string> ::= "<string_characters>"**
  This non-terminal defines string. Strings are composed of string characters written between two quotation mark tokens.

- **<comment_string> ::= #<string_characters>#**
  This non-terminal defines comment. Comments are composed of string characters written between two hashtag tokens.

- **<string_characters> ::= <character> | <string_characters> <character>**
  This non-terminal defines string characters. String characters can be either only one character or more than one character.

- **<character> ::= <lowercase_letter> | <uppercase_letter> | <digit> | <symbols> | SPACE**
  This non-terminal defines character. A character can be a lowercase letter, an uppercase letter, a digit, a symbol or space.

- **<code_block> ::= // <stmts> **
  User can define a code block by encapsulating the code between // and \\.

- **<type_id> ::= int | double | string | connection**
  A type id, which defines the type of a variable can be an int, double, string, or connection.