CS315 - Project 2

ASA

Group 19

İsmail Sergen Göçmen (21903707) - Section 1
Ahmed Salih Cezayir (21802918) - Section 2
Abdulkadir Erol (21703049) - Section 2

LEX TOKENS

In ASA, the tokens that are used in BNF are represented as below.

VOID_TYPE void
INT_TYPE int
BOOLEAN_TYPE boolean
STRING_TYPE string
CHAR_TYPE char
FLOAT_TYPE float
CONST const
GLOBAL global
RETURN return
IF if
ELSE else
ELSEIF elseif
FOR for
WHILE while
DO do
CONTINUE continue
BREAK break
TRY try
CATCH catch
PLUS \+
MINUS \-
MUL *
DIV \/
SIN_COM \/\/.*
MULT_COM \/*([^*]|*+[^*/])**\/
SIN_Q \'
DBL_Q \"
TAB \t
LP \(
RP \)
LSB \[
RSB \]
LCB \{
RCB \}
LT \<
GT \>
LTE \<\=
GTE \>\=
ASSIGN \=
EQ \=\=

```
NEQ \!\=
OR \|\|
AND \&\&
NOT \!
S_COLON \;
COMMA \,
DOLLAR \$
DIGIT [0-9]
INT [+-]?[0-9]+
FLOAT [+-]?{DIGIT}*(\.)?{DIGIT}+
BOOLEAN true|false
LETTER [a-zA-Z_]
SIGN [+|-]
ALPHANUMERIC ({LETTER}|{DIGIT})
IDENTIFIER {LETTER}{ALPHANUMERIC}*
STRING \"(\\.|[^"\\])*\"
CHAR \'{LETTER}\'
ARRAY \<\<[ ]*{IDENTIFIER}[ ]*\>\>
MAIN \$main\$
READ_HEADER \$readHeader\$
READ_ALTITUDE \$readAltitude\$
READ_TEMPERATURE \$readTemperature\$
MOVE_VERTICALLY \$moveVertically\$
MOVE_HORIZONTALLY \$moveHorizontally\$
TURN_HEADING \$turnHeading\$
TURN_SPRAY \$turnSpray\$
CONNECT \$connect\$
GET \$get\$
PRINT \$print\$
FUNC_NAME \${IDENTIFIER}\$
NEWLINE \n
```

BNF DESCRIPTION

In the following part, BNF for our language can be seen. The words that are written in uppercase letters represent the tokens that we used in our lex description file (<CHAR>, <READ_HEADER>, <RETURN>, <GT>, etc.).

The order of our BNF description is done for our YACC file in order to increase readability.

INITIAL & MAIN

- ASA starts with an initial token that includes the main function. Global variable declaration, function definition, and comments are allowed before main. So, everything else should be added in the main function.

  <before_main> ::= <comment_line>
  | <function_definition> <before_main>
  | <global_var_dec> <before_main>
  |

  <MAIN> ::= <LP> <RP> <LCB> <statements> <RCB>

STATEMENTS

- Statements are defined as like C++ or Java. A statement can be a loop statement, function call, function definition, var dec, assign, comment line, input, output, error statement, or if statement. Statements can be a statement, combination of statements, or an empty. Return statements are taken separately due to their usage being different from other statements. Our primitive functions are considered as functions, so they are not added one more but they can be statements as well.

  <statements> ::= <statement> <statements> |

  <statement> ::= <loop_statement> | <function_call> | <function_definition>
  | <var_dec> | <assign> | <comment_line> | <input>
  | <output> | <error_statement> | <if_statement>

  <return_statement> ::= <RETURN> <val> <S_COLON>

  | <RETURN> <var_name> <S_COLON>

  | <RETURN> <S_COLON>

CONDITIONAL STATEMENTS

- The if statements are checked whether they are only if or include else in ASA. If statements execute their code block if the condition is true. If the condition is not true, the else block will be executed. Moreover, else if statements are added. The usage of else if is as using if statements, using else if statement and else statement at the end.

  <if_statement> ::= <only_if> | <if_else> | <if_elseif_else>

<only_if> ::= <IF> <LP> <l_expression> <RP> <LCB> <statements> <RCB>

<if_else> ::= <IF> <LP> <l_expression> <RP> <LCB> <statements> <RCB> <ELSE>
<LCB> <statements> <RCB>

<if_elseif_else> ::= <IF> <LP> <l_expression> <RP> <LCB> <statements> <RCB>
<ELSEIF> <LP> <l_expression> <RP> <LCB> <statements> <RCB>
<ELSE> <LCB> <statements> <RCB>

LOOPS

- There are three different loop types in ASA: While (<WHILE>), do while (<DO>), and for (<FOR>) loop. These loops are similar to the Java language.

  <loop_statement> ::= <while_loop> | <for_loop> | <do_while>

- While (<WHILE>) takes a relational expression as a condition and loops through a block of codes until the condition becomes false.

  <while_loop> ::= <WHILE> <LP> <l_expression> <RP> <LCB> <statements> <RCB>

- For loop (<FOR>) takes variable declaration, logical expression, and an arithmetic expression. The statement sets a variable before the loop starts and is executed only one time. Relational expression is checked for every iteration. When it becomes false, the iteration ends. An arithmetic expression is completed before the next iteration starts (Note that ASA does not support postfix increment variable aka. i++).

  <for_loop> ::= <FOR> <LP> <var_dec> <l_expression> <S_COLON> <assign> <RP>
  <LCB> <statements> <RCB>

- In the do while (<do_while_statement>) statement, the condition is checked after the code block is executed. This process means code blocks will be performed at least one time.

  <do_while_loop> ::= <DO> <LCB> <statements> <RCB> <WHILE> <LP>
  <l_expression> <RP> <S_COLON>

ARRAYS

- Array (<array>) is a variable that has a series of memory locations that starts with << and ends with >>. In between, a variable name (<var_name>) is located. After the array,

the square brackets, the size of the array is defined. It can be an integer (<INT>) or a variable name (<var_name>) that is also an integer.

<array> ::= << <var_name> >> <LSB> <INT> <RSB>
| << <var_name> >> <LSB> <var_name> <RSB>

PRECEDENCE & EXPRESSIONS

- In ASA, expressions can be defined as variable declaration, or logical expression. Logical expressions include relational, boolean, and arithmetic expressions and their combinations. Variable declarations (<var_dec>) and logical expressions (<l_expression>) can be seen below.

  <expression> ::= <var_dec> | <l_expression>

- Var declaration can be either a global variable declaration (<global_var_dec>) or non global variable declaration (<non_global_var_dec>). Global and non global variable declarations can be either (<GLOBAL>) constant variable declaration (<const_var_dec>) or non const variable declaration (<non_const_var_dec>).

  Const and non const variable declarations are defined like C++. On the left hand side, variable identifier type, variable name and assign operator (=), on the right hand side, it consists of expressions (Ex: int x = y + 7) or function calls. At the end, a semicolon is added. Moreover, an array can be declared as, on the left hand side, array and assign operator (=), on the right hand side, list between curly brackets. In const variable declarations, in addition to the non const variable declarations, the const token will be added. List consists of vals and in between the comma is added like C++.

  <var_dec> ::= <global_var_dec> | <non_global_var_dec>

  <global_var_dec> ::= <GLOBAL> <const_var_dec>
  | <GLOBAL> <non_const_var_dec>

  <non_global_var_dec> ::= <const_var_dec>
  | <non_const_var_dec>

  <non_const_var_dec> ::= <var_id_type> <var_name> <ASSIGN> <l_expression>
  <S_COLON>
  | <var_id_type> <var_name> <ASSIGN> <function_call>
  | <var_id_type> <var_name> <S_COLON>
  | <array> <ASSIGN> <LCB> <list> <RCB> <S_COLON>

  <const_var_dec> ::= <CONST> <var_id_type> <var_name> <ASSIGN> <l_expression>
  <S_COLON>

| <CONST> <var_id_type> <var_name> <ASSIGN> <function_call>
| <CONST> <var_id_type> <var_name> <S_COLON>
| <CONST> <array> <ASSIGN> <LCB> <list> <RCB> <S_COLON>

<list> ::= <val> <COMMA> <list> |

- Logical expressions consist of arithmetic, relational, and boolean expressions. So, all of their combinations are provided. Moreover, logical expressions are ordered in precedence.

  From highest to lowest:
  1- Parentheses (())
  2- Not operator (!)
  3- Less than or equal & Less than & Greater than or equal & Greater than (<=, <, >=, >)
  4- Not equal & Equal (!=, ==)
  5- AND operator (&&)
  6- OR operator (||)
  7- Multiplication & Division (*, /)
  8- Addition & Subtraction (+, -)

  <l_expression> ::= <l_factor_0> | <l_expression> <PLUS> <l_factor_0>

  <l_expression> <MINUS> <l_factor_0>

  <l_factor_0> ::= <l_factor_1> | <l_factor_0> <MUL> <l_factor_1>

  | <l_factor_0> <DIV> <l_factor_1>

  <l_factor_1> ::= <l_factor_2> | <l_factor_1> <OR> <l_factor_2>

  <l_factor_2> ::= <l_factor_3> | <l_factor_2> <AND> <l_factor_3>

  <l_factor_3> ::= <l_factor_4> | <l_factor_3> <EQ> <l_factor_4>

  | <l_factor_3> <NEQ> <l_factor_4>

  <l_factor_4> ::= <l_factor_5> | <l_factor_4> <GT> <l_factor_5>

  | <l_factor_4> <GTE> <l_factor_5>

  | <l_factor_4> <LT> <l_factor_5>

  | <l_factor_4> <LTE> <l_factor_5>

  <l_factor_5> ::= <l_factor_6> | <l_factor_5> <NOT> <l_factor_6>

<l_factor_6> ::= <INT> | <FLOAT> | <BOOLEAN> | <CHAR> | <var_name >
| <LP> <l_expression> <RP>

INPUT & OUTPUT STATEMENTS

- Input statements are responses that are asked from the user. Output statements are the responses that the computer does to the user.

- Input (<input>) function can be taken with primitive function get (<GET>), and between the parentheses, string (<STRING>) value can be taken. If entered, this string is printed to the console before getting the user input.

  <input> ::= <GET> <LP> <STRING> <RP> <S_COLON>
  | <GET> <LP> <RP> <S_COLON>

- Output (<output>) function can be taken with primitive function print (<PRINT>), and between the parentheses, a variable name is located.

  <output> ::= <PRINT> <LP> <var_name> <RP> <S_COLON>
  | <PRINT> <LP> <val> <RP> <S_COLON>

VARIABLES & VARIABLE IDENTIFIERS

- In our language ASA, there are many variables according to their specifications. Char (<CHAR>) consists of uppercase letters, lowercase letters, and underscore character (_) since underscore character (_) is used frequently in modern languages. Int (<INT>) is the integer that involves both negative and non-negative integer values. Float (<FLOAT>) consists of the numbers that are not an integer. String (<STRING>) is an array consisting of chars and digits in between double quotes (“”). Char (<CHAR>) is a single character in between single quotes(‘’). Boolean (<BOOLEAN>) is a variable that can be true or false.

- Variable identifier type (<var_id_type>) is the variable identifier. It can be int, char, boolean, string, or float.

  <var_id_type> ::= <INT_TYPE> | <CHAR_TYPE> | <BOOLEAN_TYPE> |
  <STRING_TYPE> | <FLOAT_TYPE>

- Var (<var_name>) is an array of chars and digits. However, it should start with a char like other programming languages.

<var_name> ::= <LETTER> | <var_name> <DIGIT> | <var_name> <LETTER>

- Val (<val>) represents the values, holding the value of an int, float, string, or char.

  <val> ::= <INT> | <FLOAT> | <STRING> | <CHAR>

ASSIGNMENT & ASSIGNMENT OPERATOR

- Assignments can be expressed with an equal (=) operator which assigns the right-hand side of the equal (=) operator to the left-hand side. A variable can be assigned to an expression or a function's return value.

  <assign> ::= <var_name> <assign_op> <l_expression> <S_COLON>

  | <var_name> <assign_op> <function_call>

  <assign_op> ::= <ASSIGN>

ERROR HANDLING

- For error handling, ASA has a basic try & catch block. When there is a problem in the try block, the program does not terminate. Instead, it goes to the catch block. In this block, the user can do anything, such as print an error message.

  <error_statement> ::= <TRY> <LCB> <statement> <RCB> <CATCH> <LCB> <statement> <RCB>

FUNCTION DEFINITIONS AND CALLS

- Users can define functions to use later in their program in ASA. The function definition is like C++. Firstly, the function type is declared, and between the parentheses, parameters are put. After that, it consists of statements and a return statement in between curly brackets.

  <function_definition> ::= <function_type> <function_identifier> <LP> <parameters> <RP> <LCB> <statements> <return_statement> <RCB>

- Function identifier <function_identifier> is used to represent function name. Also, $ should be used at the start and end of the function name.

<function_identifier> ::= $<var_name>$

- Function return type <function_type> represents the data type of function return value. The function type can be string, int, float, char, and void.

  <function_type> ::= <INT_TYPE> | <VOID_TYPE> | <STRING_TYPE>

  | <BOOLEAN_TYPE> | <CHAR_TYPE>

- Parameters <parameters> are used to define argument values in function definition which function takes. They can be string, char, int, boolean, float, or empty.

  <parameters> ::= <var_id_type><var_name>

  | <var_id_type><var_name> <COMMA> <parameters>

  |

- The function call can be done with function identifier, parentheses, and arguments in between parentheses like Java or C++.

  <function_call> ::= <function_identifier> <LP> <arguments> <RP> <S_COLON>
  | <primitive_function_call>

- Arguments <arguments> are the values that function takes to use in their code block.

  <arguments> ::= <var_name>
  | <val>
  | <var_name> <COMMA> <arguments>
  | <val> <COMMA> <arguments>
  |

PRIMITIVE FUNCTIONS

- There are some primitive functions in our language ASA, readHeader (<read_header_function>), readAltitude(<read_altitude_function>) and readTemperature(<read_temperature_function>) read the current heading,altitude and temperature of the drone, respectively. They do not need to have any parameters.

  <primitive_function_call> ::= <read_header_function>
  | <read_altitude_function>
  | <read_altitude_function>
  | <read_temperature_function>

| <move_vertically_function>
| <move_horizontally_function>
| <turn_heading_function>
| <turn_spray_function>
| <connect_function>

- reading the heading

<read_header_function> ::= <INT> <READ_HEADER> <LP> <RP> <S_COLON>
| <INT> <READ_HEADER> <LP> <STRING> <RP> <S_COLON>

- reading the altitude

<read_altitude_function> ::= <INT> <READ_ALTITUDE> <LP> <RP> <S_COLON>
| <INT> <READ_ALTITUDE> <LP> <STRING> <RP> <S_COLON>

- reading the temperature

<read_temperature_function> ::= <INT> <READ_TEMPERATURE> <LP> <RP>
<S_COLON>
| <INT> <READ_TEMPERATURE> <LP> <STRING>
<RP> <S_COLON>

- moveVertically (<move_vertically_function>) and moveHorizontally (<move_horizontally_function>) moves drone vertically and horizontally. They have 3 function signatures. The parameter is for time. It can be int, float or variable. Drone speed is assumed as 0.1 m/s for moveVertically and 1 m/s for moveHorizontally. The sign values of speed indicate movement direction.
  - vertically, climb up, drop down, or stop

<move_vertically_function> ::= <MOVE_VERTICALLY> <LP> <INT> <RP> <S_COLON>
| <MOVE_VERTICALLY> <LP> <FLOAT> <RP>
<S_COLON>
| <MOVE_VERTICALLY> <LP> <var_name> <RP>
<S_COLON>

  - horizontally, move forward, backward, or stop

<move_horizontally_function> ::= <MOVE_HORIZONTALLY> <LP> <INT> <RP>
<S_COLON>
| <MOVE_HORIZONTALLY> <LP> <FLOAT> <RP>
<S_COLON>
| <MOVE_HORIZONTALLY> <LP> <var_name> <RP>
<S_COLON>

- turnHeading (<turn_heading_function>) function takes an integer that reflects total angle change of drone's turn. Logical expression is used to calculate negative variables fastly.
  - turn the heading to left or right

  <turn_heading_function> ::= <TURN_HEADING> <LP> <l_expression> <RP>

- turnSpray (<turn_spray_function>) takes a boolean parameter which indicates whether drone spray as it moves or not.
  - turning on or off spray nozzle

  <turn_spray_function> ::= <TURN_SPRAY> <LP> <BOOLEAN> <RP> <S_COLON>

- Connect (<connect_function>) takes a string as a parameter that indicates the drone's serial number so that the computer only gives commands to the connected drone.
  - connecting to the base computer through wi-fi

  <connect_function> ::= <CONNECT> <LP> <STRING> <RP> <S_COLON>
  | <CONNECT> <LP> <IDENTIFIER> <RP> <S_COLON>

COMMENTS

- In ASA, users can comment with one line or multiple lines. The symbol // is used to represent a single line. The symbol /* represents the beginning of multiple line comments, and this symbol */ represents the end of the multiple line comments. The comment mechanism is the same as the java language.

  <comment_line> ::= <SIN_COM> | <MULT_COM>