CS 315 – Programming Languages

Project 2

Droner Language

Fall 2020

Osman Buğra Aydın 21704100 Section 2

İlhan Koç 21603429 Section 1

Yahya Mahmoud Ahmed Elnouby Mohamed
21801332 Section 2

Table of content

1. Complete BNF Description

1.1 Program and Statements

<program> ::= <main>|<class_declare> |<class_declare> <main>

<main> ::= <MAIN><LEFT_BRACKET><RIGHT_BRACKET>
<LEFT_CURLY_BRACKET><stmts><RIGHT_CURLY_BRACKET>

<class_declare> ::= <CLASS> <GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>

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

<stmt> ::= <loops> | <decision> | <declaration> | <input> | <output>
|<constructor_call> | <constructor_declare> | <function_declaration>
|<primitive_function_dec> |<COMMENT> | <RETURN> <parameter_arg>
|<assignment> | <precedence>

<constructor_declare> ::= <GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <LEFT_BRACKET> <RIGHT_BRACKET>
<LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>
|<GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <LEFT_BRACKET> <parameter_decs> <RIGHT_BRACKET>
<LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<constructor_call> ::= <GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <IDENTIFIER> <LEFT_BRACKET> <parameter_args>
<RIGHT_BRACKET>

1.2 Assignment

<assignment> ::= <IDENTIFIER> <ASSIGN_RIGHT_TO_LEFT_OP> <assign_expr>
|<declaration> <ASSIGN_RIGHT_TO_LEFT_OP> <assign_expr>
|<assign_expr> <ASSIGN_LEFT_TO_RIGHT_OP> <IDENTIFIER>
|<assign_expr> <ASSIGN_LEFT_TO_RIGHT_OP> <declaration>

<assign_expr> ::= <precedence> | <BOOLEAN> | <STRING> |<CHAR>

1.3 Arithmetic Expression

<precedence> ::= <precedence> <ADD_OP> < term> |<precedence>
<SUBTRACT_OP> <term> | <term>
<term> ::= <term> <MULTIPLY_OP> <factor> |<term> <DIVIDE_OP> <factor>
|<factor>

<factor> ::= <idc> <EXPONENT_OP> <factor> |<idc>

<idc> :: = <LEFT_SQUARE_BRACKET> <precedence>
<RIGHT_SQUARE_BRACKET> |<REAL> |<INTEGER> | <IDENTIFIER>
|<function_call> |<primitive_function_call>

1.4 Type and Declaration

<declaration> ::= <type_ident> <IDENTIFIER>

<type_ident> ::= <INT_TYPE> |<BOOLEAN_TYPE> |<REAL_TYPE>
|<STRING_TYPE> |<CHAR_TYPE> |<VOID_TYPE>

1.5 Function Declaration and Call

<function_declaration> ::= <FUNC> <declaration> <LEFT_BRACKET>

<parameter_decs> ::= <declaration> | <declaration> <COMMA> <parameter_decs>

<parameter_decs> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>
|<FUNC> <declaration> <LEFT_BRACKET><
RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>

<function_call> ::= <FUNC_CALL> <COLON> <IDENTIFIER> <LEFT_BRACKET>
<parameter_args> <RIGHT_BRACKET>
| <FUNC_CALL> <COLON> <IDENTIFIER>
<DOT><IDENTIFIER><LEFT_BRACKET> <parameter_args> <RIGHT_BRACKET>

<parameter_args> ::= <parameter_arg> | <parameter_arg> <COMMA>
<parameter_args> |

<parameter_arg> ::= <IDENTIFIER> |<BOOLEAN> |<INTEGER> |<STRING>
|<REAL>|<CHAR>

1.6 Primitive Function Declaration and Call

<primitive_function_dec> ::= <CLASS_IDENT> <BOOLEAN_TYPE>
<CONNECT_PC> <LEFT_BRACKET> <INT_TYPE> <IDENTIFIER>
<RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>
|<CLASS_IDENT> <REAL_TYPE> <READ_INC>
<LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>
|<CLASS_IDENT> <REAL_TYPE> <READ_ALT>
<LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>
|<CLASS_IDENT> <REAL_TYPE> <READ_TEMP>
<LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>
|<CLASS_IDENT> <REAL_TYPE> <READ_ACC>
<LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>
|<CLASS_IDENT> <VOID_TYPE> <CONTROL_CAM>
<LEFT_BRACKET> <BOOLEAN_TYPE> <IDENTIFIER> <RIGHT_BRACKET>
<LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>
|<CLASS_IDENT> <VOID_TYPE> <TAKE_PIC>
<LEFT_BRACKET> <BOOLEAN_TYPE> <IDENTIFIER> <COMMA> <INT_TYPE>
<IDENTIFIER> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>
|<CLASS_IDENT> <INT_TYPE> <READ_TIME>
<LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>

<primitive_function_call> ::= <CLASS_IDENT> <DOT> <primitive_name>
<LEFT_BRACKET> <parameter_args> <RIGHT_BRACKET>

<primitive_name> ::= <READ_INC> |<READ_ALT> |<READ_TEMP> |<READ_ACC>
|<CONTROL_CAM> |<TAKE_PIC> |<READ_TIME> |<CONNECT_PC>

1.7 Loops

<loops> ::= <while> |<for> |<do_while>

<while> ::= <WHILE> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<for> ::= <FOR> <LEFT_BRACKET> <assignment> <COLON> <expression> <COLON> <assignment> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<do_while> ::= <DO> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET> <WHILE> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET>

1.8 Expressions and Logic Precedence

<expression> ::= <assign_expr> <relational> <assign_expr>
<relational> ::= <EQUALITY_OP> | <NOT_EQUAL_OP> |<GREATER_THAN_OP> |<LESS_THAN_OP> |<GREATER_THAN_EQUAL_OP> |<LESS_THAN_EQUAL_OP>

<logic_precedence> ::= <logic_precedence> <OR> <logic_term> | <logic_term>

<logic_term> ::= <logic_term> <AND> <logic_factor> |<logic_factor>

<logic_factor> ::= <NOT> <logic_idc> | <logic_idc>

<logic_idc> ::= <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET> |<assign_expr> | <expression>

1.9 Conditional Statements

<decision> ::= <conditional>

<conditional> ::= <if_stmt> |<|if_else_stmt>

<if_stmt> ::= <IF> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET> <THEN> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<if_else_stmt> ::= <IF> <LEFT_BRACKET> <logic_precedence>
<RIGHT_BRACKET> <THEN> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET> <ELSE_DO> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>

1.10 Input and Output Statements

<input> ::= <INPUT> <LEFT_BRACKET> <IDENTIFIER> <RIGHT_BRACKET>
|<INPUT> <LEFT_BRACKET> <parameter_decs> <RIGHT_BRACKET>

<output> ::= <OUTPUT> <LEFT_BRACKET> <logic_precedence>
<RIGHT_BRACKET>

2. Explanation of Language Structure

<program> ::= <main> | <class_declare> | <class_declare><main>

Ex. class >DRONE< { statements } launch () { statements }

The program rule is the start of our language, the user can start with the main function, class declaration, or class declaration followed by a main function.

**<main> ::= <MAIN><LEFT_BRACKET><RIGHT_BRACKET>
<LEFT_CURLY_BRACKET><stmts><RIGHT_CURLY_BRACKET>**

Ex. launch () { statements }

This rule is our main function declaration, the word launch is used to start declaring our main function. We used the word launch for the user to understand that all process are executed in this function.

**<class_declare> ::= <CLASS> <GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <LEFT_CURLY_BRACKET> <stmts>
<RIGHT_CURLY_BRACKET>**

Ex. class >MYDRONE< {stmts}

One of the main ways in our language to distinguish between function declaration and class declaration is starting the declaration using the reserved word class and the name of the class must be in upper case.

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

<stmt> ::= <loops> | <decision> | <declaration> | <input> | <output>
|<constructor_call> | <constructor_declare> | <function_declaration>
|<primitive_function_dec> |<COMMENT> | <RETURN> <parameter_arg>
|<assignment> | <precedence>

Ex. #This a comment Ex2. droner.in(number a); Ex3. number a;

Statements may contain one or more statements. First line of the non-terminal states that stmts can be just one statement. Second line is a recursive way to show that <smts> can contain more than one statement. A statement has essential types to construct a solid program. These are basically the main blocks of Droner language.

<constructor_declare> ::= <GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <LEFT_BRACKET> <RIGHT_BRACKET>
<LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>
|<GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <LEFT_BRACKET> <parameter_decs> <RIGHT_BRACKET>
<LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<constructor_call> ::= <GREATER_THAN_OP> <UPPER_LETTER>
<LESS_THAN_OP> <IDENTIFIER> <LEFT_BRACKET> <parameter_args>
<RIGHT_BRACKET>

Ex. >DRONE< (boolean onFF, number time) { stmts }

These two rules are for declaring a constructor and calling it, the constructor also starts with class name and then none or more parameters.

<assignment> ::= <IDENTIFIER> <ASSIGN_RIGHT_TO_LEFT_OP>
<assign_expr>
|<declaration> <ASSIGN_RIGHT_TO_LEFT_OP> <assign_expr>
|<assign_expr> <ASSIGN_LEFT_TO_RIGHT_OP>
<IDENTIFIER>
|<assign_expr> <ASSIGN_LEFT_TO_RIGHT_OP> <declaration>

<assign_expr> ::= <precedence> | <BOOLEAN> | <STRING> |<CHAR>

Ex. 7 + 8 -> a Ex2. c <- a + 8

Assignment stands for initializing a variable, or assigning values to identifiers. We have two directions to declare a variable. Direction of the arrows always shows which value is assigned to the other. This makes it more understandable by the users to see how data is assigned either from the right hand side to the left hand side or the left hand side to the right hand side.

<precedence> ::= <precedence> <ADD_OP> < term> |<precedence> <SUBTRACT_OP> <term> | <term>
<term> ::= <term> <MULTIPLY_OP> <factor> |<term> <DIVIDE_OP> <factor> |<factor>

<factor> ::= <idc> <EXPONENT_OP> <factor> |<idc>

<idc> :: = <LEFT_SQUARE_BRACKET> <precedence> <RIGHT_SQUARE_BRACKET> |<REAL> |<INTEGER> | <IDENTIFIER>
|<function_call> |<primitive_function_call>

Ex. (a + b) * 5 - 2

The way implemented the arithmetic expression acts on maintaining both precedence and left associativity. As the rule is represented as a parse tree, the lowest expression is calculated first that's why we divided the expression into pieces in which when the rule is parsed the past by the addition-subtraction then division-multiplication then exponents and finally expressions surrounded by parentheses. This rule when represented as a parse tree it is seen that the lowest expression is the one surrounded by square brackets having the highest precedence and the on the top is addition-subtraction with the lowest precedence. The left associativity is done by having the recursive part of the left side of each expression rule.

<declaration> ::= <type_ident> <IDENTIFIER>

<type_ident> ::= <INT_TYPE> |<BOOLEAN_TYPE> |<REAL_TYPE> |<STRING_TYPE> |<CHAR_TYPE> |<VOID_TYPE>

Ex. real average

This rule is for declaring variables. Our language includes 6 different types that can be used to declare variables or as return types including integers which is represented with the reserved word number, float represented by real, boolean represented by boolean, string represented by string, char represented by char, and void represented by none.

<function_declaration> ::= <FUNC> <declaration> <LEFT_BRACKET>

<parameter_decs> ::= <declaration> | <declaration> <COMMA> <parameter_decs>

<parameter_decs> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>
|<FUNC> <declaration> <LEFT_BRACKET>< RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<function_call> ::= <FUNC_CALL> <COLON> <IDENTIFIER> <LEFT_BRACKET> <parameter_args> <RIGHT_BRACKET>
| <FUNC_CALL> <COLON> <IDENTIFIER> <DOT> <IDENTIFIER> <LEFT_BRACKET> <parameter_args> <RIGHT_BRACKET>

<parameter_args> ::= <parameter_arg> | <parameter_arg> <COMMA> <parameter_args> |

<parameter_arg> ::= <IDENTIFIER> |<BOOLEAN> |<INTEGER> |<STRING> |<REAL>|<CHAR>

Ex. func number getID() {statements} Ex2. call:getID() Ex3. call:myDrone.getID()

This is the rule for defining and calling a function. To declare a function, we start using the reserved word func to show that we are declaring a function. Besides, to call a function we start by the reserved word call: to indicate we are calling a function.

<primitive_function_dec> ::= <CLASS_IDENT> <BOOLEAN_TYPE> <CONNECT_PC> <LEFT_BRACKET> <INT_TYPE> <IDENTIFIER> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

|<CLASS_IDENT> <REAL_TYPE> <READ_INC> <LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

|<CLASS_IDENT> <REAL_TYPE> <READ_ALT> <LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

|<CLASS_IDENT> <REAL_TYPE> <READ_TEMP> <LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

|<CLASS_IDENT> <REAL_TYPE> <READ_ACC> <LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

|<CLASS_IDENT> <VOID_TYPE> <CONTROL_CAM> <LEFT_BRACKET> <BOOLEAN_TYPE> <IDENTIFIER> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

|<CLASS_IDENT> <VOID_TYPE> <TAKE_PIC> <LEFT_BRACKET> <BOOLEAN_TYPE> <IDENTIFIER> <COMMA> <INT_TYPE> <IDENTIFIER> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

|<CLASS_IDENT> <INT_TYPE> <READ_TIME> <LEFT_BRACKET> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<primitive_function_call> ::= <CLASS_IDENT> <DOT> <primitive_name> <LEFT_BRACKET> <parameter_args> <RIGHT_BRACKET>

<primitive_name> ::= <READ_INC> |<READ_ALT> |<READ_TEMP> |<READ_ACC> |<CONTROL_CAM> |<TAKE_PIC> |<READ_TIME> |<CONNECT_PC>

Ex. droner number getTimeStamp() Ex. droner.getTimeStamp()

This is the rule for declaring and calling the primitive functions. To distinguish between primitive functions and primitive functions we use the reserved word which is the name of our language, droner that is used at the beginning of declaring or calling a primitive function.

<loops> ::= <while> |<for> |<do_while>

<while> ::= <WHILE> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<for> ::= <FOR> <LEFT_BRACKET> <assignment> <COLON> <expression> <COLON> <assignment> <RIGHT_BRACKET> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<do_while> ::= <DO> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET> <WHILE> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET>

Ex. for (number a: a < 10: a <- a + 1) { statements }

These rules are the loop rules. The language includes three different ways of defining loops including the for loop using colons instead of semicolons to separate expressions, while loop, and do while loop.

<expression> ::= <assign_expr> <relational> <assign_expr>
<relational> ::= <EQUALITY_OP> | <NOT_EQUAL_OP> |<GREATER_THAN_OP> |<LESS_THAN_OP> |<GREATER_THAN_EQUAL_OP> |<LESS_THAN_EQUAL_OP>

Ex. x > 10 Ex.2 time == currentTime

This rule represents relational expressions that include greater than, greater than or equal, less than, less than or equal, equal equal and not equal relations.

<logic_precedence> ::= <logic_precedence> <OR> <logic_term> | <logic_term>

<logic_term> ::= <logic_term> <AND> <logic_factor> |<logic_factor>

<logic_factor> ::= <NOT> <logic_idc> | <logic_idc>

<logic_idc> ::= <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET> |<assign_expr> | <expression>

Ex. if ((time > x) || !(x < 10)) then { statements }
Similarly, the same was done for logical expressions in which when the parse tree is done the lowest operation in the parse tree will be the logical expressions surrounded

by parentheses (highest precedence) followed by the not operator, the middle operator is the and operator and the highest operation that is on top (lowest precedence) of the tree will be the or operator expressions.

<decision> ::= <conditional>

<conditional> ::= <if_stmt> |<|if_else_stmt>

<if_stmt> ::= <IF> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET> <THEN> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

<if_else_stmt> ::= <IF> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET> <THEN> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET> <ELSE_DO> <LEFT_CURLY_BRACKET> <stmts> <RIGHT_CURLY_BRACKET>

Ex. if (a > 1) then {statements} elseDo {statements}

This rule represents conditional statements that include the if statement and if-else statement. We defined the conditional statements using more reserved words like then and elseDo to make the conditional statements more understandable.

<input> ::= <INPUT> <LEFT_BRACKET> <IDENTIFIER> <RIGHT_BRACKET>
|<INPUT> <LEFT_BRACKET> <parameter_decs> <RIGHT_BRACKET>
|<INPUT> <LEFT_BRACKET> <IDENTIFER> <COMMA> <STRING> <RIGHT_BRACKET>
|<INPUT> <LEFT_BRACKET> <declaration> <COMMA> <STRING> <RIGHT_BRACKET>

Ex. droner.in(number a, “Desktop/input.txt”) Ex2. droner.in(number a)

The input statement is called the same way primitive functions are called using the reserved word which is the name of the programming language “droner”. In the input statements you can define the variable inside it as in example 2, declare the variable before and pass the identifier to the input statement, or pass an identifier or declaration along with a string that has the path for the input as in example 1.

<output> ::= <OUTPUT> <LEFT_BRACKET> <logic_precedence> <RIGHT_BRACKET>

Ex. droner.out (a) Ex2. droner.out(“Hello World”)

The output statement is called the same way primitive functions are called using the reserved word which is the name of the programming language “droner”. Although the rule appears as if it takes logical expressions it contains assign_expr which contains the constants and precedence rules, the user can print the identifier's value, function call that is returning a value, strings, chars, integers,floats and arithmetic expressions .

3. Non trivial Tokens

for

The reserved word “for” is used to define for loops in the droner language.

while

The reserved word “while” is used to define while loops in the droner language.

do

The reserved word “do” is used with “while” to define do while loops in the droner language.

if

The reserved word “if” is used at the beginning of if statements and if else statements to define the conditional statements in the droner language.

then

The reserved word “then” is used in if statements and if else statements to define the conditional statements in the droner language.

elseDo

The reserved word “elseDo” is used in if statements and if else statements to define the conditional statements in the droner language.

droner.in

The reserved word “droner.in” is used for taking input statement in the droner language.

droner.out

The reserved word “droner.out” is used for printing which is for output statement in the droner language.

class

The reserved word “class” is used at the start of class declaration to define a class in the droner language.

launch

The reserved word “launch” is for the main function in the droner language.

droner

The reserved word “droner” is for the primitive function call and declaration in the droner language.

func

The reserved word “func” is for the function declaration in the droner language.

call

The reserved word “call” is for the function call in the droner language.

return

The reserved word “return” is for returning values in functions (int, string, boolean, etc) in the droner language.

4. How Conflicts Were Resolved

While implementing the droner language we faced many conflicts that we successfully managed to solve all of them. Mainly the conflicts were shift reduce conflicts that were caused due to that some of our rules had similar definitions. For example, both arithmetic precedence and logical precedence ended up with identifiers or constants, so as the parser went through them both reduce at the same time causing a reduce reduce conflict or when two rules had the first few tokens similar to each other causing a shift reduce conflict. We resolved reduced conflicts mostly by adding new reserved words to

the rules that would not affect reliability and writability and for shift reduce conflicts, there were some rules that could have been combined, so we combined them to resolve shift reduce problems.

5. How Precedence and Ambiguities Were Resolved

The way implemented the arithmetic expression acts on maintaining both precedence and left associativity. As the rule is represented as a parse tree, the lowest expression is calculated first that's why we divided the expression into pieces (**as shown in the bnf rules and its descriptions**) in which when the rule is parsed then passed by the addition-subtraction then division-multiplication then exponents and finally expressions surrounded by square brackets. This rule when represented as a parse tree it is seen that the lowest expression is the one surrounded by parentheses having the highest precedence and the on the top is addition-subtraction with the lowest precedence. The left associativity is done by having the recursive part of the left side of each expression rule. Similarly, the same was done for logical expressions in which when the parse tree is done the lowest operation in the parse tree will be the logical expressions surrounded by parentheses (highest precedence) followed by the not operator, the middle operator is the and operator and the highest operation that is on top (lowest precedence) of the tree will be the or operator expressions.