CS 315 - Programming Languages

Project 2 Report

Group 7

Öykü Irmak Hatipoğlu

21802791

Section 2

Cansu Moran

21803665

Section 1

Elif Gamze Güliter

21802870

Section 2

1. Complete BNF Description of Luny

<program >::=

<stmt_list> ENDPROGRAM

<stmt_list> ::=

<stmt>

| <stmt> stmt_list>

<stmt>::=

<const_dec>

| <if_stmt>

| <loop>

| <assignment_op>

| <comments>

| <input>

|<output>

| <func_dec>

| <func_call>

<const_dec> ::=

CONST IDENTIFIER ASSIGNMENT_OP INTEGER NEWLINE

| CONST IDENTIFIER ASSIGNMENT_OP STRING NEWLINE

<if_stmt>::=

IF LP <expr> RP NEWLINE START NEWLINE <stmt_list> END NEWLINE

| IF LP <expr> RP NEWLINE START NEWLINE <stmt_list> END NEWLINE

ELSE NEWLINE START NEWLINE <stmt_list> END NEWLINE

<loop>::=

< while_loop>

| <for_loop>

<while_loop>::=

WHILE LP <expr> RP NEWLINE START NEWLINE <stmt_list> END NEWLINE

<for_loop>::=

FROM LP INTEGER RP TO LP INTEGER RP NEWLINE START NEWLINE

<stmt_list> END NEWLINE

| FROM LP INTEGER RP TO LP IDENTIFIER RP NEWLINE START NEWLINE

<stmt_list> END NEWLINE

| FROM LP IDENTIFIER RP TO LP IDENTIFIER RP NEWLINE START

NEWLINE <stmt_list> END NEWLINE

| FROM LP IDENTIFIER RP TO LP INTEGER RP NEWLINE START NEWLINE

<stmt_list> END NEWLINE

<assignment_op> ::=

IDENTIFIER ASSIGNMENT_OP <func_call>

| IDENTIFIER ASSIGNMENT_OP STRING NEWLINE

| IDENTIFIER ASSIGNMENT_OP <expr> NEWLINE

<expr>::=

<relation_expression>

| <logic>

<relation_expression>::=

<deep_term> <relation_sign> <deep_term>

<relation_sign>::=

GREATER_THAN

| LESS_THAN

| GREATER_EQUAL

| LESS_EQUAL

| IF_EQUAL

<logic>::=

<logic_low>

| <logic> OR <logic_low>

<logic_low>::=

<logic_high>

| <logic_low> AND <logic_high>

<logic_high>::=

<arithmetic_expr>

| NOT< arithmetic_expr>

<arithmetic_expr> ::=

<term>

| <term> PLUS <arithmetic_expr>

| <term> MINUS <arithmetic_expr>

<term>::=

<deep_term>

| <term> MULTIPLY <deep_term>

| <term> DIVIDE <deep_term>

<deep_term>::=

INTEGER

| IDENTIFIER

| LP <expr> RP

<comments>::=

COMMENT NEWLINE

<input> ::=

ENTER COLON LP IDENTIFIER RP NEWLINE

<output > ::=

DISPLAY COLON LP IDENTIFIER RP NEWLINE

| DISPLAY COLON LP STRING RP NEWLINE

| DISPLAY COLON LP INTEGER RP NEWLINE

<func_dec>::=

FUNCTION IDENTIFIER LP <func_args> RP NEWLINE START NEWLINE

<stmt_list> END NEWLINE

| FUNCTION IDENTIFIER LP RP NEWLINE START NEWLINE <stmt_list> END

NEWLINE

<func_args>::=

IDENTIFIER

| IDENTIFIER COMMA <func_args>

<func_call> ::=

<nonprim_call>

| <primitive_call >

<nonprim_call>::=

IDENTIFIER LP <params> RP NEWLINE

| IDENTIFIER LP RP NEWLINE

<params>::=

<param_types>

| <param_types> COMMA <params>

<param_types>::=

IDENTIFIER

| INTEGER

| STRING

<primitive_call> ::=

<getIncline>

| <getAltitude>

| <getTemperature>

| <getAcceleration>

| <setCamera>

| <takePictures>

| <getTimestamp>

| <connect>

| <turnCW>

| <turnCCW>

| <ascend>

| <descend>

| <takeOff>

| <land>

| <goTo>

<getIncline>::=

GETINCLINE LP RP NEWLINE

<getAltitude> ::=

GETALTITUDE LP RP NEWLINE

<getTemperature >::=

GETTEMPERATURE LP RP NEWLINE

<getAcceleration>::=

GETACCELERATION LP RP NEWLINE

<setCamera>::=

SETCAMERA LP IDENTIFIER RP NEWLINE

| SETCAMERA LP INTEGER RP NEWLINE

| SETCAMERA LP STRING RP NEWLINE

<takePictures>::=

TAKEPICTURES LP RP NEWLINE

<getTimestamp>::=

GETTIMESTAMP LP RP NEWLINE

<connect>::=

CONNECT LP IDENTIFIER COMMA IDENTIFIER RP NEWLINE

| CONNECT LP IDENTIFIER COMMA STRING RP NEWLINE

| CONNECT LP STRING COMMA STRING RP NEWLINE

| CONNECT LP STRING COMMA IDENTIFIER RP NEWLINE

<turnCW> ::=

TURNCW LP IDENTIFIER RP NEWLINE

| TURNCW LP INTEGER RP NEWLINE

<turnCCW>::=

TURNCCW LP IDENTIFIER RP NEWLINE

| TURNCCW LP INTEGER RP NEWLINE

<ascend>::=

ASCEND LP IDENTIFIER RP NEWLINE

| ASCEND LP INTEGER RP NEWLINE

<descend>::=

DESCEND LP IDENTIFIER RP NEWLINE

| DESCEND LP INTEGER RP NEWLINE

<takeOff>::=

TAKEOFF LP RP NEWLINE

<land>::=

LAND LP RP NEWLINE

<goTo>::=

GOTO LP IDENTIFIER COMMA IDENTIFIER RP NEWLINE

| GOTO LP IDENTIFIER COMMA INTEGER RP NEWLINE

| GOTO LP INTEGER COMMA IDENTIFIER RP NEWLINE

| GOTO LP INTEGER COMMA INTEGER RP NEWLINE

2. Description of the Structure of the Language

In our language, Luny, there are several design choices that are integral to the structure of our language.

We wanted to make sure that the users don't create an empty program. We made sure that an empty program would result in a syntax error and the users must write at least one line of code.

In our language, every statement must end with a new line(\n). We decided not to have a symbol such as ";" which is commonly used in C languages. Instead, we decided to distinguish the end of a statement with a newline. This was a conscious choice, as we decided that not having an end statement symbol would increase writability. This way, the only thing the user needs to do to finish a statement is to click enter, which is much easier than putting a symbol at the end of the line. However, because our program uses newline to distinguish the the end of a statement, the users must not enter any empty lines in the program.

We decided not to have a start statement in our program so that the users can start coding without using any initiating term. However, our language has an ending statement, which is "endprogram". At the end of each program, the user must write "endprogram" to indicate that their program has ended. This was a requirement in our language as we decided to distinguish the end of a statement with a newline. We have realized that if the users forgot to enter a newline on the last statement of the program, the program would give a syntax error. Therefore, in order

to create a more reliable language, we decided to add an ending statement, so that the users won't have to remember to add a newline on the last statement. They would just have to remember to finish the program with the word "endprogram" on the last line.

In our language, the user uses "start" and "end" reserved words to indicate the block statements written inside a function definition, if statement or while or for loops. We decided to make it obligatory for users to enter a newline before and after these words. This choice was made to increase readability of block statements. We decided that it becomes easier to distinguish block statements, if both "start"and "end" are written on their own seperated from any other statements.

As a design choice, we decided not to allow any function calls being made inside output statements, the logic statements of if statements and for and while loops, or as a parameter of another function call (ex: connect(getWifi())). We decided to do this to make our language more simplistic. There are two ways that a function call can be made. It can either be made on its own, or it can be made as a part of an assignment statement, to assign the return value of the function to a variable. If the user wishes to use the return value of a function as for example the conditional statement of an if statement, they can do it by first assigning the return value to a variable and then using the variable inside the if statement.

In our language, we have implemented precedence and associativity rules in all arithmetic, logic or relational operations. In relational operations, the users can compare identifiers and integers, and any expressions inside the parentheses. For example, they can't write 3 + 4 < 5 + a, but they can write (3 + 4) < (5 + a). This choice was made to increase readability and make sure that someone reading the code can easily understand which expressions are being compared. For the same reason, in our language the users can only compare two expressions at a time. This means that a statement such as a < b < c is invalid. We decided not to implement not equal(!=) sign in our language as it can already be made by using NOT(!) and if equal(==)

signs. For arithmetic operations, any expression inside the parentheses has higher precedence than multiplication and division, and multiplication and division have higher precedence than addition and subtraction. Similarly, for logic operations, any expression inside the parentheses has higher precedence than NOT, NOT has higher precedence than AND and AND has higher precedence than OR. As a design choice, we decided only to allow the usage of identifiers and integers as operands of any relational, logic or arithmetic operations.

In our language, we tried to keep reserved words as short and simple as possible. We decided that using simple words would not only increase readability and writability, but it would also make the language more functional, considering that the main purpose of this language is drone programming and that drone controller screens are relatively small.

2.1. Description of Non-Terminals

- <program >::= <stmt_list> ENDPROGRAM

<program> is a non-terminal that every program will have. It is the initializing rule of our language. There is no reserved word which will declare the starting of the program, however, there is a reserved word “endprogram” indicating that the program is ended.

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

<stmt_list> is a non-terminal that represents a list of statements. It includes cases where there is only one statement or there are multiple statements.

- <stmt>::= <if_stmt> | <loop> | <func_call>| <func_dec>| <assignment_op>| <comments> | <input> |<output>| <const_dec>

<stmt> that is a non-terminal that includes if statements(<if_stmt>); for and while loops (<loop>); function calls(<func_call>); function declarations(<func_dec>); assignment

operations (<assignment_op>); comments (<comments>); input statements (<input>) ; constant declarations (<const_dec>) and output statements (<output>) .

- <const_dec> ::= CONST IDENTIFIER ASSIGNMENT_OP INTEGER NEWLINE | CONST IDENTIFIER ASSIGNMENT_OP STRING NEWLINE

This non-terminal represents constant declarations in the language. Constant declarations start with the word “constant” followed by an identifier, an assignment operator(=) , and the value which will be assigned to the constant variable, which can either be a string or an integer.
Example : const x = 6 , const y = “hello” ...

- <if_stmt>::= IF LP <expr> RP NEWLINE START NEWLINE <stmt_list> END NEWLINE | IF LP <expr> RP NEWLINE START NEWLINE <stmt_list> END NEWLINE ELSE NEWLINE START NEWLINE <stmt_list> END NEWLINE

This non terminal is used for the if statements in Luny language. If statements start with the word “if” followed by an expression (<expr>) inside parenthesis. These expressions can be logical expressions, relational expressions or arithmetic expressions like in C languages. Expressions inside if statements will be placed between “start” and “end” words. If statements can also have else statements. Expressions inside else statements will be written between “start” and “end” reserved words as well.
Example: if (a < b)

```
start
if ( c > e )
        start
        a = a + 1
        end
else
```

start

a = a - 1

end

end

- <loop>::= < while_loop> | <for_loop>

This non-terminal represents the loop statements in our language. Our language has two loop statements: while loops(<while_loop>) and for loops(<for_loop>).

- <while_loop>::= WHILE LP <expr> RP NEWLINE START NEWLINE <stmt_list> END NEWLINE

This non-terminal represents while loops in our language. After the reserved word "while", the expression that controls whether the while loop will be executed should be given inside the parentheses. All the statements that will be executed inside the while loop should be written between the words "start" and "end".

Example: while (a < b)

start

a = a + 1

end

- <for_loop>::= FROM LP INTEGER RP TO LP INTEGER RP NEWLINE START NEWLINE <stmt_list> END NEWLINE | FROM LP INTEGER RP TO LP IDENTIFIER RP NEWLINE START NEWLINE <stmt_list> END NEWLINE | FROM LP IDENTIFIER RP TO LP IDENTIFIER RP NEWLINE START NEWLINE <stmt_list> END NEWLINE | FROM LP IDENTIFIER RP TO LP INTEGER RP NEWLINE START NEWLINE <stmt_list> END NEWLINE

This non-terminal represents for loops in our language. The starting point of the for loop should be written inside the parentheses after the word “from”. The starting point can either be an integer value or a value of a variable. Similarly, the end point of the loop should be written inside the parentheses after the word “to”. The end point can also be an integer value or a value of a variable. The loop increases the “from” value by 1 in each iteration, until it reaches the “to” value, in which case the loop ends. All the statements that will be executed inside the for loop should be written between the words “start” and “end”.

Example:

from (0) to (10)

start

a = a + 1

end

In this example, both “from” and “to” values are given as int values. The loop will execute 10 times and in each iteration the value of the variable “a” will be increased by 1.

- <assignment_op> ::= IDENTIFIER ASSIGNMENT_OP <func_call> | IDENTIFIER ASSIGNMENT_OP STRING NEWLINE | IDENTIFIER ASSIGNMENT_OP <expr> NEWLINE

This non-terminal represents an assignment operation in the Luny language. On the left hand side of the assignment operator(=) a variable identifier should be given. On the right hand side the value that is desired to be assigned to that identifier should be given which can be only an integer, a string or a function call.

Example : “ x = 6, x = “hello”, x = getTimestamp( )...”

- <expr>::= <relation_expression> | <logic>

<expr> non-terminal represents expressions in our language. These expressions

include logic expressions(<logic>) and relational expressions (<relation_expression>).

- <relation_expression>::= <deep_term> <relation_sign> <deep_term>

This non- terminal represents the relational expression in the Luny language. In relational expressions, comparing only two values or expressions are allowed ; and in order to prevent ambiguities and set precedence rules, the non-terminal <deep_term> is used. Further information about <deep_term> will be found below.

- <relation_sign>::= GREATER_THAN | LESS_THAN| GREATER_EQUAL | LESS_EQUAL | IF_EQUAL

This nonterminal represents the relational signs in our language such as less than(<), greater than(>), less than or equal(<=), greater than or equal(>=) and if equal(==) sign.

- <logic>::= <logic_low> | <logic> OR <logic_low>

This non-terminal represents logical expressions in our language. In order to ensure precedence and associativity of the operators, the operation OR that have the least precedence is written before other operations.

- <logic_low>::= <logic_high> | <logic_low> AND <logic_high>

This non-terminal is used to prevent ambiguities and to apply precedence rules between logical operators. The operation AND has less precedence than the operation NOT, therefore NOT operation is written inside logic_high. AND also has higher precedence than OR operator.

- <logic_high>::= <arithmetic_expr> | NOT< arithmetic_expr>

This non-terminal is also used for preventing ambiguities and applying precedence rules between logical operations. The operation NOT has the highest precedence amongst other operators therefore it is written in <logic_high> non-terminal.

- <arithmetic_expr> ::= <term> | <term> PLUS <arithmetic_expr> | <term> MINUS <arithmetic_expr>

This non-terminal represents arithmetic expressions in our language. In order to ensure precedence and associativity of the operators, the operations that have the least precedence -addition(+) and subtraction(-)- are written before other operations.

- <term>::= <deep_term> | <term> MULTIPLY <deep_term> | <term> DIVIDE <deep_term>

<term> is a non-terminal that is used to ensure precedence and associativity of the operators. It includes multiplication(*) and division(/) operations that have higher precedence than addition and subtraction but lower precedence than any arithmetic operation written inside the parentheses.

- <deep_term>::= INTEGER | IDENTIFIER | LP <expr> RP

<deep_term> is a non-terminal that represents a single term in an arithmetic, relational or logic operation or an expression inside the parentheses. It is used to ensure precedence and associativity of the operators. It can be a relational, logic or arithmetic expression written only inside the parentheses, an integer or an identifier.

- <nonprim_call>::= IDENTIFIER LP <params> RP NEWLINE | IDENTIFIER LP RP NEWLINE

This non terminal is used for the non-primitive function calls in Luny language. In order to call a non primitive function, only the name of the function and its parameters between

brackets are needed. The function that is being called can also have no parameters as well.

- <params>::= <param_types> | <param_types> COMMA <params>

This nonterminal represents the function parameters in a function call in our language which can be integer, string or identifier. In the case of multiple parameters, a comma must be used between the parameters.

- <param_types>::= IDENTIFIER | INTEGER | STRING

This non-terminal represents parameter types in the language which can be string, integer or an identifier.

- <func_dec>::= FUNCTION IDENTIFIER LP <func_args> RP NEWLINE START NEWLINE <stmt_list> END NEWLINE | FUNCTION IDENTIFIER LP RP NEWLINE START NEWLINE <stmt_list> END NEWLINE

This nonterminal is used for function declarations in our language. The word “function” is a requisite at the beginning of declaration. Following that, an identifier is used as the name of the function and inside the brackets function arguments are needed. Functions do not need to have function arguments, it can be empty as well. Statements in functions must be written between “start” and “end” reserved words.

Example :

function sayHello()

start

display: (“hello”)

end

- <func_args>:: = IDENTIFIER | IDENTIFIER COMMA <func_args>

<func_args> nonterminal represents function arguments in a function declaration which

can only be an identifier. Using multiple arguments is allowed, but in order to seperate them, a comma is needed.

- <comments>::= COMMENT NEWLINE

<comment> nonterminal represents the whole single line comment in our language. Anything written on the right hand sign of the comment sign "@" is a single line comment in Luny language.

- <input> ::= ENTER COLON LP IDENTIFIER RP NEWLINE

This non-terminal represents the input statement that can be used to get input from the user. After the reserved word "enter" a semicolon(:) should be placed. The variable(<identifier>) which the input value should be assigned to should be written inside the parentheses and only variables are allowed to be used in the input statements.
Example: enter: (name)
In this example, the input value obtained from the user will be placed inside the "name" variable.

- <output > ::= DISPLAY COLON LP IDENTIFIER RP NEWLINE | DISPLAY COLON LP STRING RP NEWLINE | DISPLAY COLON LP INTEGER RP NEWLINE

This non-terminal represents the output statement which can be used if the user wants to print anything on the console. The user can enter the name of a variable(<identifier>) to print its value or they can enter a string(<string>) or an integer(<int>) to be displayed. By design choice, the user can't make a function call directly to print it. If the user wishes to print the return value of a function, they should first assign it to a variable and then print the value of the variable. In any case, the value that is desired to

be printed on the console should be written inside the parentheses after the reserved word “display” and a semicolon.

- <func_call> ::= < primitive_call >| <nonprim_call>

This nonterminal represents function calls in our language that are either non primitive or primitive function calls.

- <primitive_call> ::= <getIncline> | <getAltitude> | <getTemperature>| <getAcceleration> | <setCamera>| <takePictures> | <getTimestamp>| <connect>| <turnCW> | <turnCCW> | <ascend> | <descend> | <takeOff> | <land> | <goTo>

This nonterminal represents all the primitive function calls in our language. In order to call primitive functions, only the names of them and their parameters between brackets are needed. However, functions can also have no parameters, in that case, inside of the parentheses should be empty.

- <getIncline>::= GETINCLINE LP RP NEWLINE

This nonterminal represents the primitive function call getIncline() which reads the incline from the drone’s gyroscope and returns the inclination of a drone.
Example : getIncline()

- <getAltitude> ::= GETALTITUDE LP RP NEWLINE

This nonterminal is used for the primitive function which reads the altitude by the drone’s barometer and returns the altitude of the drone.
Example : getAltitude()

- <getTemperature >::= GETTEMPERATURE LP RP NEWLINE

This nonterminal is used for the primitive function which reads and returns the temperature of the weather from the termometer of the drone.

Example: getTempeture()

- <getAcceleration>::= GETACCELERATION LP RP NEWLINE

This nonterminal is used for the primitive function which reads the acceleration of the drone from the acceleration sensor and returns the current acceleration of the drone.

Example: getAcceleration()

- <setCamera>::= SETCAMERA LP IDENTIFIER RP NEWLINE

| SETCAMERA LP INTEGER RP NEWLINE | SETCAMERA LP STRING RP NEWLINE

This nonterminal is the primitive function that turns the camera on the drone on and off. When the camera is on, the drone can take pictures by calling the takePictures() function.This function takes a parameter which can be a string, “on” or “off”; it can be an integer, 1 or 0 which opens and closes the camera respectively ; or it can be an identifier.

Example: setCamera(camera); camera is 0, 1, “on”, or “off”, setCamera(0), setCamera(“off”)...

- <takePictures>::= TAKEPICTURES LP RP NEWLINE

This nonterminal is used for the primitive function which performs taking pictures.

Example: takePictures()

- <getTimestamp>::= GETTIMESTAMP LP RP NEWLINE

This nonterminal is the primitive function for getting the current timestamp from the drone’s timer.

Example: getTimestamp()

- <connect>::= CONNECT LP IDENTIFIER COMMA IDENTIFIER RP NEWLINE
  | CONNECT LP IDENTIFIER COMMA STRING RP NEWLINE

| CONNECT LP STRING COMMA STRING RP NEWLINE

| CONNECT LP STRING COMMA IDENTIFIER RP NEWLINE

This nonterminal is the primitive function to connect the drone to the base computer through wi-fi. This function takes two parameters, the first parameter is the name of the wi-fi and the second parameter is the password of the wi-fi. Both Wi-fi name and password can be an identifier or a string. Using the wi-fi connection, the drone will be able to connect to the base computer. Therefore, the first instruction of the program should be connect(wifiname, password) to connect the drone to the base computer.

Example: connect(wifiname, password),connect(“wifi123”, “123”)

- <turnCW> ::= TURNCW LP IDENTIFIER RP NEWLINE

| TURNCW LP INTEGER RP NEWLINE

This nonterminal is a primitive function for turning the drone clockwise by the specified angle which will be given as a parameter through either an identifier or an integer.

Example: turnCW(angle); angle is 30, turnCW(30)

- <turnCCW>::= TURNCCW LP IDENTIFIER RP NEWLINE

| TURNCCW LP INTEGER RP NEWLINE

This nonterminal is a primitive function for turning the drone counterclockwise by the specified angle which will be given as a parameter through either an identifier or an integer.

Example: turnCCW(angle); angle is 30, turnCCW(30)

- <ascend>::= ASCEND LP IDENTIFIER RP NEWLINE

| ASCEND LP INTEGER RP NEWLINE

This nonterminal is the primitive function for making the drone ascend by the specified value on the parameter through either an identifier or an integer.

Example: ascend(up); up is 5, ascend(5)

- <descend>::= DESCEND LP IDENTIFIER RP NEWLINE | DESCEND LP INTEGER RP NEWLINE

This nonterminal is the primitive function for making the drone descend by the specified value on the parameter through either an identifier or an integer.

Example: descend(down); down is 5, descend(5)

- <takeOff>::= TAKEOFF LP RP NEWLINE

This nonterminal is the primitive function to make the drone take off when it is resting on the ground.

Example: takeOff()

- <land>::= LAND LP RP NEWLINE

This nonterminal is the primitive function to make to drone land to the ground when it is in flight.

Example: land()

- <goTo>::= GOTO LP IDENTIFIER COMMA IDENTIFIER RP NEWLINE | GOTO LP IDENTIFIER COMMA INTEGER RP NEWLINE | GOTO LP INTEGER COMMA IDENTIFIER RP NEWLINE | GOTO LP INTEGER COMMA INTEGER RP NEWLINE

This nonterminal is the primitive function to fly the drone to a desired place in terms of x and y axes which will be given in the parameters through first and second parameter respectively. These two parameters can either be identifiers or integers.

X axis is the axis which is parallel to the ground and the drone is facing to and the y axis is the axis which is also parallel to the ground but this time it is perpendicular to the x axis. The drone will fly to the vectorial sum of those two parameters.

Example: goTo(x, y)

In this example,identifier x is 3 and y is 4, therefore, the drone will turn tan (4/3) degrees clockwise

and will fly forward for 5 units.

2.2. Explanation of Non-Trivial Tokens

LP: Token for left parenthesis.

RP: Token for right parenthesis.

FUNCTION: Token for the reserved word “function” at the beginning of function declaration.

START: Token for the reserved word “start” that defines starting of an expression inside if, else or loop statements.

END: Token for the reserved word “end” that defines ending of an expression inside if , else or loop statements.

FROM: Token for the reserved word “from” that defines the starting point of a for loop.

TO: Token for the reserved word “to” that defines the ending point of a for loop.

IDENTIFIER: Token for variable names and function names, except the reserved words.

AND: Token for “and” logical operator.

OR: Token for “or” logical operator.

ENTER: Token for the reserved word for input statements.

DISPLAY: Token for the reserved word for output statements.

IF_EQUAL: Token for "==" operator that checks whether the right-hand side is equal to left-hand side.

NOT: Token for the "not" sign in logical expressions.

CONST: Token for the reserved for "const" in a constant declaration.

ENDPROGRAM: Token for the reserved word "endprogram" which defines the ending of a Luny program.

CONNECT: Token for the function "connect" which is used for the Wi-Fi connection of the drone.

TURNCW: Token for the function "turnCW" which is used for turning the drone clockwise.

TURNCCW: Token for the function "turnCCW" which is used for turning the drone counterclockwise.

GOTO: Token for the function "goTo" that takes the drone to a specific location.

NEWLINE: : Token for newlines(\n) entered in the program.

2.3 Resolved Conflicts

In our language, there are no conflicts. However, in the process of writing the parser, we faced several conflicts that we had to resolve. We faced both reduce/reduce and shift/reduce conflicts. After checking our grammar, we have realized that these conflicts were happening due to certain rules that were causing repetition. For example, there were multiple rules that were defining assignment statements such as "a = 1". After detecting such rules, we modified them so that the same statements weren't being defined by different rules at the same time. After doing these modifications, we managed to resolve all the conflicts in our language.