CS 315
PROGRAMMING LANGUAGES

PROJECT 2

DR-ONE (DoctorOne) LANGUAGE

TEAM 6

Cemre Biltekin / 21802369 / Section 2
Mustafa Yaşar / 21702808 / Section 2

I. LANGUAGE

The name of the language is DoctorOne, and its abbreviated form is DR-ONE.

II. COMPLETE BACKUS-NAUR FORM (BNF) DESCRIPTION OF DR-ONE

a. Program

<program> ::= BEGIN_PROGRAM <statements> END_PROGRAM

b. Statement Definitions General Form

<statements> ::= <statement> SEMI_COLON
| <statements> <statement> SEMI_COLON
| <statements> COMMENT_LINE | COMMENT_LINE

<statement> ::= <assignment_stmt> | <post_incr>| <post_decr>
| <func_call> | <declaration> | <conditional_stmt>
| <while_stmt> | <for_stmt> | <function_no_return>
| <function_return> | <input> | <output>
| <file_input> | <file_output>

<block_statement> ::= LBRACE <inside_statements> RBRACE
| LBRACE RBRACE

<inside_statements> ::= <inside_statement> SEMI_COLON
| <inside_statements> <inside_statement> SEMI_COLON

<inside_statement> ::= <assignment_stmt> | <post_incr> | <post_decr>
| <func_call> | <declaration> | <conditional_stmt>
| <while_stmt> | <for_stmt> | <input> | <output>
| <file_input> | <file_output>

c. Assignment Statement

<assignment_stmt> ::= IDENTIFIER ASSIGNMENT_OP <assignment_expr>
<assignment_expr> ::= <expr> | STRING | BOOLEAN
| <input> | <file_input>

d. Post Statements

<post_incr> ::= IDENTIFIER INCREMENT
<post_decr> ::= IDENTIFIER DECREMENT

e. Variable Declaration Statement

<declaration> ::= DECLARATION IDENTIFIER
| DECLARATION <assignment_stmt>

f. Constant

<constant> ::= INTEGER | STRING | BOOLEAN

g. Function Call Statement

<func_call> ::= IDENTIFIER LP RP
| IDENTIFIER LP <defined_parameters> RP
| <primitive_func_call>

h. Loop Statements (For and While)

<for_stmt> ::= FOR LP <for_initial> SEMI_COLON <expr> SEMI_COLON <for_update> RP <block_statement>
| FOR LP SEMI_COLON <expr> SEMI_COLON <for_update> RP <block_statement>

<for_initial> ::= <for_initial_helper> | <for_initial_helper> COMMA <for_initial>

<for_initial_helper> ::= DECLARATION IDENTIFIER ASSIGNMENT_OP
<arithm_term>
| IDENTIFIER ASSIGNMENT_OP <arithm_term>

<for_update> ::= <arithm_term> | <post_incr> | <post_decr>

<while_stmt> ::= WHILE LP <expr> RP <block_statement>

i. Conditional Statement (If Statement)

<conditional_stmt> ::= IF LP <expr> RP <block_statement>
| IF LP <expr> RP <block_statement>ELSE
<block_statement>

j. Input & Output Statements

<input> ::= DRIN LP RP
<output> ::= DROUT LP <assignment_expr> RP

<file_input> ::= FILE_DRIN LP STRING RP
| FILE_DRIN LP IDENTIFIER RP

<file_output> ::= FILE_DROUT LP <assignment_expr> COMMA IDENTIFIER RP
| FILE_DROUT LP <assignment_expr> COMMA STRING RP

k. Function Declaration Statements (With Return and Without Return)

<function_no_return> ::= FUNCTION_DECL IDENTIFIER LP <parameters>
RP <block_statement>
| FUNCTION_DECL IDENTIFIER LP RP <block_statement>

<function_return> ::=
FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE
<inside_statements> FUNC_RETURN <constant> SEMI_COLON
RBRACE
| FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE
FUNC_RETURN <constant> SEMI_COLON RBRACE
| FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE
<inside_statements> FUNC_RETURN IDENTIFIER SEMI_COLON
RBRACE
| FUNCTION_DECL IDENTIFIER LP RP LBRACE
<inside_statements> FUNC_RETURN <constant> SEMI_COLON
RBRACE
| FUNCTION_DECL IDENTIFIER LP RP LBRACE
FUNC_RETURN <constant> SEMI_COLON RBRACE
| FUNCTION_DECL IDENTIFIER LP RP LBRACE
<inside_statements> FUNC_RETURN IDENTIFIER SEMI_COLON
RBRACE
| FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE
FUNC_RETURN IDENTIFIER SEMI_COLON RBRACE

<parameters> ::= DECLARATION IDENTIFIER
| INTEGER
| STRING
| INTEGER COMMA <parameters>
| STRING COMMA <parameters>
| DECLARATION IDENTIFIER COMMA <parameters>

<defined_parameters> ::= IDENTIFIER
| INTEGER
| STRING
| STRING COMMA <defined_parameters>
| INTEGER COMMA <defined_parameters>
| IDENTIFIER COMMA <defined_parameters>

l. Expressions

<expr> ::= <logical_term> | <expr> LOGICAL_OR <logical_term>
<logical_term>::= <logical_prec>
| <logical_term> LOGICAL_AND <logical_prec>

<logical_prec>::= LOGICAL_NOT <logical_prec>
| <relational_expr>
| <relational_expr> LOGICAL_NOT <logical_prec>

<relational_expr> ::= <arithm_term> <relational_opr> <arithm_term>
| <arithm_term>

<arithm_term>::= <term>
| <arithm_term> PLUS <term>
| <arithm_term> MINUS <term>

<term> ::= <power>
| <term> MULTIPLICATION_OP <power>
| <term> DIVISION_OP <power>

<power> ::= <factor> POWER_OP <power>
| <factor>

<factor> ::= LP <expr> RP | INTEGER
| IDENTIFIER | <func_call>

<relational_opr> ::= SMALLER | SMALLER_OR_EQUAL
| GREATER | GREATER_OR_EQUAL
| EQUAL_CHECK | NOT_EQUAL

m. Primitive Functions

<primitive_func_call> ::= <read_incl>
| <read_altitude> | <read_temperature> | <read_acceleration>
| <read_speed> | <camera_on> | <camera_off>
| <take_picture> | <read_time> | <connect_wifi>
| <set_speed> | <set_altitude> | <set_incl> | <drone_up>
| <drone_down> | <drone_right> | <drone_left>
| <drone_forward> | <drone_backward>
| <rotate_clockwise> | <rotate_counterclockwise>
| <gotoPos>
| <lock_position> | <unlock_position> | <give_name>

<read_incl> ::= INCLINE LP RP

<read_altitude> ::= ALTITUDE LP RP

<read_temperature> ::= TEMPERATURE LP RP

<read_acceleration> ::= ACCELERATION LP RP

<read_speed> ::= SPEED LP RP

<camera_on> ::= CAMERA_ON LP RP

<camera_off> ::= CAMERA_OFF LP RP

<take_picture> ::= TAKE_PICTURE LP RP

<read_time> ::= READ_TIME LP RP

<connect_wifi> ::= CONNECT LP STRING COMMA STRING RP

<set_speed> ::= SET_SPEED LP INTEGER RP

<set_altitude> ::= SET_ALTITUDE LP INTEGER RP

<set_incl> ::= SET_INCLINE LP INTEGER RP

<drone_up> ::= MOVE_DRONE_UP LP INTEGER RP

<drone_down> ::= MOVE_DRONE_DOWN LP INTEGER RP

<drone_right> ::= MOVE_DRONE_RIGHT LP INTEGER RP

<drone_left> ::= MOVE_DRONE_LEFT LP INTEGER RP

<drone_forward> ::= MOVE_DRONE_FORWARD LP INTEGER RP

<drone_backward> ::= MOVE_DRONE_BACKWARD LP INTEGER RP

<rotate_clockwise> ::= ROTATE_CW LP INTEGER RP

<rotate_counterclockwise> ::= ROTATE_CCW LP INTEGER RP

<gotoPos> ::= GO_TO_POSITION LP INTEGER COMMA INTEGER COMMA INTEGER COMMA INTEGER RP

<lock_position> ::= LOCK_POSITION LP RP

<unlock_position> ::= UNLOCK_POSITION LP RP

<give_name> ::= GIVE_NAME LP STRING RP

III. DESCRIPTIONS OF DR-ONE CONSTRUCTS

1. **<program> ::= BEGIN_PROGRAM <statements> END_PROGRAM**

The program is enclosed with two reserved words which are start and stop which are defined in lex and have tokens BEGIN_PROGRAM and END_PROGRAM respectively. Start is located at the very beginning of DR-ONE programs (starting command), and stop is the terminating command that is located at the end of the programs. All statements, which make up the program, exist between these commands.

2. **<statements> ::= <statement> SEMI_COLON**
   | <statements> <statement> SEMI_COLON
   | <statements> COMMENT_LINE | COMMENT_LINE

Statements body of the program can be composed of one or more statements (list of statements) and comments. This is a recursive rule. All statements must end with a semicolon except comments.

3. **<statement> ::= <assignment_stmt> | <post_incr>| <post_decr>**
   | <func_call> | <declaration> | <conditional_stmt>
   | <while_stmt> | <for_stmt> | <function_no_return>
   | <function_return> | <input> | <output>
   | <file_input> | <file_output>

A statement can be one of fourteen statements which are assignment statement, post increment statement, post decrement statement, function call statement, variable declaration statement, conditional (if) statement, while statement, for statement, and function declaration statements with return and without return, input and output statements, file input and output statements. These statements make up the program.

4. **<block_statement> ::= LBRACE <inside_statements> RBRACE**
   | LBRACE RBRACE

A block statement is different from the set of statements presented in above sections. This non-terminal specifies which statements can be written inside other statements such as conditional statements, loop statements and function declarations. This was a necessity since a function declaration cannot be made inside a conditional statement or a loop statement or another function declaration in DR-ONE. Block statements are embraced with braces unlike statements. A block statement can be left empty (no statement inside braces).

5. **<inside_statements> ::= <inside_statement> SEMI_COLON**
   | <inside_statements> <inside_statement> SEMI_COLON

The inside of a block statement can be composed of one or more specific/inside statements. All inside statements end with a semicolon.

6. **<inside_statement> ::= <assignment_stmt> | <post_incr> | <post_decr>**
   | <func_call> | <declaration> | <conditional_stmt>
   | <while_stmt> | <for_stmt> | <input> | <output>
   | <file_input> | <file_output>

An inside statement can be any statement defined in the language except function declaration statements (with return and without return). Therefore, all statements can have other statements inside (e.g. if statement within a loop statement, assignment statement within function declaration with return etc.) except a function declaration (with return and without return) in DR-ONE.

7. **COMMENT_LINE**

A comment should be embraced with reserved expression that is *CM* in the beginning of the comment and in the end. Comment can be a single line and anything defined in the language (letters, digit, symbols) can be inside a comment. This non-terminal is defined in lex and its token is COMMENT_LINE.

8. **<assignment_stmt> ::= IDENTIFIER ASSIGNMENT_OP <assignment_expr>**

This non-terminal is the definition of an assignment statement. An assignment statement is composed of a left hand side (a value subject/variable for this operation), an assignment operator (=) between and the right hand side (the value assigned to subject/variable, an expression).

9. **<assignment_expr> ::= <expr> | STRING | BOOLEAN**
   | <input> | <file_input>

An assignment expression is the right hand side of the assignment statement. This non-terminal definition shows that an assignment expression can be a value of a variable, the result of an arithmetic expression, relational expression, logical expression and combination of all three expressions (these come from <expr> non-terminal which is described in detail in 28th definition), a constant value (string, boolean, integer), a function's return value, and a user input (via input statement), or a file input.

10. **<post_incr> ::= IDENTIFIER INCREMENT**

This is the definition of a post increment statement. It is the shortcut for adding 1 to a variable that is an integer (since there are no other variable types). Thus, it is an alternative to an arithmetic operation, but it is shorter and quicker. A variable is needed for this definition.

Instead of a single plus sign, there are two consecutive ones. The incrementation is done after this line is executed.

11. <post_decr> ::= IDENTIFIER DECREMENT

This is the definition of a post decrement statement. Similar to the post increment, it is a shortcut for subtracting 1 from a variable. It is again a shorter alternative to a subtraction arithmetic operation. A variable is needed for this definition. Instead of a single minus sign, there are two consecutive ones. The decrementation is done after this line is executed.

12. <declaration> ::= DECLARATION IDENTIFIER
| DECLARATION <assignment_stmt>

This non-terminal is the definition of variable declaration. There are two ways to declare a variable, first, "var" followed by the name of the variable without assignment statement (i.e. var varName;), second, "var" followed by the name of the variable with assignment statement (i.e. var varName = value;). There are no types in DR-ONE, so there are no reserved words for differentiating types of variables, so a generic reserved word is used instead. The variable can be assigned an integer, a string or a boolean value.

13. IDENTIFIER

Variable names and function names could start with a letter, or _ (underscore) sign. Underscore could only be at the beginning, digits cannot be at the beginning. For this reason, the initial character of the identifier is treated differently than the rest of the characters in the identifier. The rest of the characters of an identifier can be a letter or a digit. It is made recursive for long identifiers.

14. <constant> ::= INTEGER | STRING | BOOLEAN

The constants in the language can either be integers, strings (words) or boolean.

15. BOOLEAN

Booleans are truth values. Boolean value can either be true or false. Thus, *true* and *false* are reserved words. Its definition is made in lex.

16. STRING

A string (a word) is a series of characters (specific/valid characters) embraced by quotation marks. To construct a valid string, a string must be composed of letters, digits or symbols. Symbols are any symbols except quotation mark and an escape character. It can also be empty (nothing between the quotation marks). Its definition is made in lex.

17. INTEGER

An integer can either be signed or unsigned. An unsigned integer lacks any sign. Signs are plus or minus. The integer definition is recursive. No sign means the integer is positive. Its definition is made in lex.

18. <func_call> ::= IDENTIFIER LP RP
| IDENTIFIER LP <defined_parameters> RP
| <primitive_func_call>

A function call is made by writing the function name (identifier) first followed by parentheses. Since functions may have parameters or zero parameters, a function call may also take parameters or not depending on its function definition. A function call may also be a primitive function call, that is, the functions that exist by default in the program which need only to be called properly.

19. <for_stmt> ::= FOR LP <for_initial> SEMI_COLON <expr> SEMI_COLON <for_update> RP <block_statement>
| FOR LP SEMI_COLON <expr> SEMI_COLON <for_update> RP <block_statement>

<for_update> ::= <arithm_term> | <post_incr> | <post_decr>

In DR-ONE language, loops have similar syntax to Java for loops. The “for” reserved word should be put to the beginning followed by opening parentheses. Initialization part could be empty or many initializations could be made with commas separating them. For instance: for (var x = 0, var y = speed(); x > 5; x++) {}; is a simple for loop that does nothing. Second part of the for loop is the condition part that is composed of expressions. Lastly, the last part is increment, or decrement part that updates the for statement. For loop can take a block of statements inside. When the expression (condition) holds no longer true, the loop is left.

20. <for_initial> ::= <for_initial_helper>
| <for_initial_helper> COMMA <for_initial>

Initialization part can either have one initialization assignment or a series of them. It is a recursive definition. They are separated by commas.

21. <for_initial_helper> ::=
DECLARATION IDENTIFIER ASSIGNMENT_OP <arithm_term>
| IDENTIFIER ASSIGNMENT_OP <arithm_term>

The for initializer helper defines possible definitions of initialization. A new variable can be declared and assigned a value for the loop, an existing variable could be used, an arithmetic expression can initialize a value or an integer constant can be used.

22. <while_stmt> ::= WHILE LP <expr> RP <block_statement>

While statements start with "while" reserved word followed by opening parenthesis and the expression and closing parenthesis. It takes a block of statements inside. The expression can be composed of arithmetic expressions, relational expressions, logical expressions and combination of these three expressions as well, and when that expression no longer holds true, the loop is left. For instance, if there is a while statement like:

```
while( 3 + 5 ){};
```

This is syntactically correct in our language because after the arithmetic expression inside is evaluated, our language accepts any nonzero integer as true and zero as false.

23. <conditional_stmt> ::= IF LP <expr> RP <block_statement>
| IF LP <expr> RP <block_statement> ELSE <block_statement>

The conditional statement is an if else statement. There can only be a single if in the statement, or the if statement can be accompanied by an else statement. If the if condition holds true, the block of statements inside the if statement is executed, if not it is skipped and else is executed (if else exists). The if condition can either be a arithmetic expression, relational expression, relational expression, and combination of these 3 expressions.

Because any integer except 0 is accepted as true in DR-ONE language, statements like:

```
if ( 3 + 5 ) {};
```

are accepted as true and syntactically correct, and the statements inside the braces are executed.

24. <input> ::= DRIN LP RP
<output> ::= DROUT LP <assignment_expr> RP

In order to get input from the user drin() should be used. Whenever this method is called, the input will be taken from the user (console). Similarly, drout function can be used for the output to the console. Any assignment expression can be output to the console such as string, integer, boolean, identifier and expressions (relational, logical, and arithmetic).

25. <file_input> ::= FILE_DRIN LP STRING RP
| FILE_DRIN LP IDENTIFIER RP

<file_output> ::=
FILE_DROUT LP <assignment_expr> COMMA IDENTIFIER RP

| FILE_DROUT LP <assignment_expr> COMMA STRING RP

To get input from a file, fdrin() function should be used. The name of the file to be called should be given as input to the function as a string or a variable that holds the file name. When this method is called, the context of the file will be returned. This function can be seen as a reading file function.

To write a context to a file, fdrout() function should be used. The name of the file to be written into should be given as the second input to the function. If no file is found with the given name, a new file is created with the name of the second input, and the context given as the first input is written to the file. Assignment expressions such as arithmetic expressions, relational expressions, logical expressions, combination of these expressions, string, boolean can be given as the first input to the function.

Additionally in DR-ONE, drin() & fdrin() functions can be written as the first input to the output functions. This is syntactically correct. The program first gets an input from console or a file, then outputs to the console or a file if such a code is written:

drout(drin());

26. <function_no_return> ::=

FUNCTION_DECL IDENTIFIER LP <parameters>
RP <block_statement>
| FUNCTION_DECL IDENTIFIER LP RP <block_statement>

<function_return> ::=
FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE
<inside_statements> FUNC_RETURN <constant> SEMI_COLON RBRACE

| FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE
FUNC_RETURN <constant> SEMI_COLON RBRACE

| FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE
<inside_statements> FUNC_RETURN IDENTIFIER
SEMI_COLON RBRACE

| FUNCTION_DECL IDENTIFIER LP RP LBRACE
<inside_statements> FUNC_RETURN <constant>
SEMI_COLON RBRACE

| FUNCTION_DECL IDENTIFIER LP RP LBRACE
FUNC_RETURN <constant> SEMI_COLON RBRACE
| FUNCTION_DECL IDENTIFIER LP RP LBRACE

<inside_statements> FUNC_RETURN IDENTIFIER SEMI_COLON RBRACE

| FUNCTION_DECL IDENTIFIER LP <parameters> RP LBRACE FUNC_RETURN IDENTIFIER SEMI_COLON RBRACE

Every function declaration must begin with the keyword "drfunction" followed by the name of the function and opening and closing parenthesis and opening and closing curly braces. Inside parenthesis, parameters could be defined with a colon separating them or parameters can be left empty. For example, drfunction func1(var x, var y, z) {} is an empty function declaration. Every function name structure is identical to the IDENTIFIER structure which means every function name can start with "_" sign.The function can either have a return or not. Return is a reserved word for returning a value of a variable or a constant.

In DR-ONE language, the programmer is forced to return 1 thing only. For instance, return 3 + 5; cannot be done in DR-ONE. In order to return such a result:

```
var result = 3 + 5;

return result;
```

This can be a solution. The type of the value to be returned can be either an integer, string, boolean, or an identifier.

27. **<parameters> ::= DECLARATION IDENTIFIER**
| INTEGER
| STRING
| INTEGER COMMA <parameters>
| STRING COMMA <parameters>
| DECLARATION IDENTIFIER COMMA <parameters>

<defined_parameters> ::= IDENTIFIER
| INTEGER
| STRING
| STRING COMMA <defined_parameters>
| INTEGER COMMA <defined_parameters>
| IDENTIFIER COMMA <defined_parameters>

Inside parameters, a new variable can be declared or existing variables and constants can be used. The parameters should be separated by a comma between them. This parameter definition is used in function declaration. However, there is also a need for defined parameters for function calls. For function calls, a variable cannot be declared inside parentheses while calling a function. Therefore, there are two types of parameters with different definitions.

28. **<expr> ::= <logical_term> | <expr> LOGICAL_OR <logical_term>**

<logical_term>::= <logical_prec>
| <logical_term> LOGICAL_AND <logical_prec>

<logical_prec>::= LOGICAL_NOT <logical_prec>
| <relational_expr>
| <relational_expr> LOGICAL_NOT <logical_prec>

<relational_expr> ::= <arithm_term> <relational_opr> <arithm_term>
| <arithm_term>

<arithm_term>::= <term>
| <arithm_term> PLUS <term>
| <arithm_term> MINUS <term>

<term> ::= <power>
| <term> MULTIPLICATION_OP <power>
| <term> DIVISION_OP <power>

<power> ::= <factor> POWER_OP <power>
| <factor>

<factor> ::= LP <expr> RP | INTEGER
| IDENTIFIER | <func_call>

<relational_opr> ::= SMALLER | SMALLER_OR_EQUAL
| GREATER | GREATER_OR_EQUAL
| EQUAL_CHECK | NOT_EQUAL

An expression can either be an arithmetic expression, a relational expression, a boolean (logical) expression or a combination of these three. To avoid ambiguity and obey mathematical and logical rules like precedence and association, the BNF definition is made accordingly and the expression rules are tied to each other to achieve combinational expressions and correct evaluation of them. For precedence order, it follows the Java operator precedence table. Combinational expressions increase the writability of the language.

Arithmetic expressions can include addition, subtraction, multiplication, division, and power operations. An arithmetic expression has terms. First expression in parantheses is evaluated, then power, multiplication and division, then summation and subtraction. The terms of the expression can be an integer constant, a variable (that has an integer value) or a function call.

Relational expressions are expressions that perform less than, less than or equal to, greater than, greater than or equal to, equal to, and not equal to operations. These operations' terms could be integer variables, integer constants, or function calls.

Boolean (logical) expressions are expressions that perform AND and OR operations. There is also negation with NOT. These are reserved words for boolean expressions. There can be one or more boolean expressions. Expressions inside parentheses have more precedence. These expressions return a truth value like true or false.

For instance,

(var1 AND var2) OR (NOT var3 <= samplefunc()) AND (var4 + 5 == 10)

is a combined expression.

29. **<primitive_func_call> ::=**
<read_incl> |<read_altitude> | <read_temperature>
| <read_speed> | <camera_on> | <camera_off>
| <take_picture> | <read_time> | <connect_wifi>
| <set_speed> | <set_altitude> | <set_incl> | <drone_up>
| <drone_down> | <drone_right> | <drone_left>
| <drone_forward> | <drone_backward>
| <rotate_clockwise> | <rotate_counterclockwise>
| <gotoPos> | <unlock_position> | <give_name>
| <lock_position> | <read_acceleration>

There are primitive drone functions in DR-ONE. They all have different definitions, parameters and return values. Their details will be explained below.

Primitive Functions:

incline() : Returns the current inclination of the drone in degrees (integer value). It does not take a parameter.

altitude() : Returns the current altitude of the drone in centimeters (integer value). It does not take a parameter.

temperature() : Returns the current temperature of the medium in degree Celcius (integer value). It does not take a parameter.

acceleration() : Returns the current acceleration of the drone in cm^2/s (integer value). It does not take parameters.

speed() : Returns the current speed of the drone in cm/s (integer value). It does not take parameters.

cameraOn() : If the drone has been started, this function opens the camera. If the camera is already opened, this function does nothing. It does not take parameters and does not return anything.

cameraOff() : If the drone has been started and the camera is opened, this function turns off the camera. If the camera is already closed, the function does nothing. It does not return anything.

takePicture() : If the camera is opened, this function takes the picture, else, it does nothing. It returns nothing and takes no parameters.

readTime() : Returns how many seconds has the drone been used as integer value. It does not take any parameters.

connect(wifiName, wifiPassword) : Connects to the wifi whose name is the first parameter (string/word) and password is the second parameter (string). Returns true if connection is accomplished, false otherwise.

setSpeed(int) : Sets the speed of the drone to the parameter. The parameter must be int and between 10 and 100 cm/s. It returns nothing.

setAltitude(int) : Sets the altitude of the drone to the parameter. The parameter must be int and between 0 and 400 centimeters. It returns nothing.

setIncline(int) : Sets the incline of the drone to the parameter that is given as a degree (integer value). It does not return anything.

droneUp(x) : Ascends the drone x centimeters. The parameter must be int and between 0 and 400 centimeters. Returns false if the drone cannot go further.

droneDown(x) : Descends the drone x centimeters. The parameter must be int and between 0 and 400 centimeters. Returns false when the drone is on the ground/base level or in landing position.

droneLeft(x) : Moves the drone left x centimeters. The parameter must be int and between 0 and 400 centimeters. Returns nothing.

droneRight(x) : Moves the drone right x meters. The parameter must be int and between 0 and 400 centimeters. Returns nothing.

droneForward(x) : Moves the drone forward x meters . The parameter must be int and between 0 and 400 centimeters. Returns nothing.

droneBackward(x) : Moves the drone backward x meters. The parameter must be int and between 0 and 400 centimeters. Returns nothing.

rotateCW(x) : Rotates the drone in clockwise direction x degrees. The parameter must be int and between 0 and 360. Returns nothing.

rotateCCW(x) : Rotates the drone in counter clockwise direction x degrees. The parameter must be int and between 0 and 360. Returns nothing.

gotoPos(a, b, c, d) : Moves the drone to a,b, and c coordinates with speed d. Every parameter must be int. d must be between 10 and 100 cm/s. Returns nothing.

lockPosition() : Locks the drone so that the drone will not move up, down or right, left and it will remain in its position. It returns true if lock is successful, false if it can't stay in that position. There are no parameters.

unlockPosition(): If the drone is locked, this function unlocks the drone so that it could be moved again. If it can't do an unlock operation, it returns false. It returns true otherwise. There are no parameters.

giveName(string): It takes a string constant/variable as the parameter and names the drone. It does not return anything.

Terminals:

Semicolon is used at the end of statements and in for loop initialization. Commas are used to separate parameters from each other. Braces are used to embrace blocks of statements and parentheses are used to show precedence in arithmetic operations, make function calls and declarations and separate expressions in a combined expression. The rest of the symbols such as “+, -, ==” are operators and are used in relational, boolean and arithmetic expressions to define what operation will be done on the subjects.

IV. DESCRIPTIONS OF NONTRIVIAL TOKENS

Reserved Words: start, stop**,** if, for, while, else, var, true, false, drfunction, drin, drout, fdrin, fdrout, AND, OR, NOT, return, *CM*

Every DR-ONE program starts with the reserved word “start” and ends with the reserved word “stop”. This way it can easily be understood where the program starts and ends. Every statement including if, while, and for statements (after closing brace) ends with the symbol “;” as in Java, C++, and many languages, therefore, the program could be read more easily. Another benefit of separating statements with semicolons is that programmers are able to write more than one statements in a single line.

Functions, loops, and if-else statements all have curly braces at the beginning of them. This increases the readability of the program since this way it is easier to determine which codes belong to those statements.

While declaring functions, drfunction reserved word is used. Doing this makes the program understand easily that a function is being created.

Parentheses are commonly used in DR-ONE language and there are many benefits using them. For instance, in order to give higher precedence to an arithmetic operation, they can be used, as convention, expressions inside parentheses have higher precedence, additionally, while writing loops and conditional statements, parentheses contain the conditions, relational expressions, etc. This increases the reliability, writability, and readability of the program significantly.

Variable names and function names are composed of arbitrarily _ sign at the beginning or any letter, followed by letters and digits. They can not contain any other symbol. These words are relatively easy compared to advanced programming languages such as Java, C++, Python. Our aim was to restraint the programmers not to use unrealistic variable names and complicate readability.

As for the reserved words such as if, else, return, for, while, and such, we used these words in order to make these words similar to the other programming languages. This way programmers could get used to using DR-ONE easily which improves readability and writability.

V. GENERAL DESCRIPTION OF DR-ONE

In terms of *overall simplicity*, DR-ONE is a simple and clear language designed specifically for drones. It has a small number of basic constructs and all constructs are relevant to its mission, that is, to command drones.

In terms of *data types,* there is no type declaration in the language which makes it easier for the programmer to declare variables and read them. Instead, a reserved word “var” is used to indicate a variable declaration in the program. This saves the worry to declare types while programming. However, there are different types of constants like integer, string and boolean. A variable can be initialized to those values without having to worry about types. In DR-ONE, integer is assigned to variables generally. Since there are no types in DR-ONE, there is no *type checking.* There might be errors in the earlier stages of the program due to unmatched types, yet for its mission, type checking could be an extra effort, and it would increase the cost since drone programming is usually concerned with integers.

In terms of *expressivity* in DR-ONE, there are shortcuts for arithmetic operations which are summation and subtraction, and post increment and post decrement can be used instead respectively.

VI. SOLVING AMBIGUITIES, PRECEDENCE & ASSOCIATIVITY PROBLEMS

For precedence order, DR-ONE follows the Java operator precedence table. From higher to lower precedence it is: expression in parentheses, arithmetic expressions, relational expressions and logical expressions. Their operator precedence within types of expressions are explained above in the report. Ambiguity is eliminated in evaluating expressions in DR-ONE (in arithmetic, relational, boolean (logical) expressions), so no expression/statement yields unexpected outcomes or more than one result for the same expression, so the language

performs to its specifications. It is done by defining BNF rules accordingly to satisfy precedence and associativity. Also, the language is defined in a way to avoid meaningless statements due to general syntax rules; all definitions are written for the language to remain logical and syntactically correct. Thus, DR-ONE is also a reliable language.

We resolved all ambiguities and conflicts in our language as we implemented it. Therefore, if a program is syntactically correct, it gives the output that says “parsing is successful” after running the program. If there is an error, the line number of the error is output to the console.

We also specified associativity in yacc by defining %left and %right tokens like %right ASSIGNMENT_OP.

NOTE: Our Makefile creates an executable called *parser*. Input files should be given to the parser as:

./parser < CS315f20_team06.test