CS 315
PROGRAMMING LANGUAGES

PROJECT 1

DR-ONE (DoctorOne) LANGUAGE

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> ::= start <statements> stop
```

b. Statement Definitions General Form

```
<statements> ::= <statement> <semicolon> | <statements> <statement> <semicolon>
                 | <statements> <comment_block> | <comment_block>
<statement> ::= <assignment_stmt> | <post_incr> | <post_decr>
                 | <function_call> | <declaration> | <conditional_stmt>
                 | <while_stmt> | <for_stmt> | <function_no_return>
                 | <function_return> | <input> | <output>

<block_statement> ::=  <LBRACE>  <inside_statements> <RBRACE>
                         |   <LBRACE>  <RBRACE>
<inside_statements> ::= <inside_statement> <semicolon>
                          | <inside_statements> <inside_statement> <semicolon>
<inside_statement> ::= <assignment_stmt> | <post_incr> | <post_decr>
                         | <function_call> | <declaration> | <conditional_stmt>
                         | <while_stmt> | <for_stmt> | <input> | <output>
```

c. Comment Statement

```
<comment_block> ::= *CM* <anything_in_lng> *CM*
<anything_in_lng> ::= <any_symbol> | <letter_digit_symbol> <anything_in_lng>
<any_symbol> ::= <symbol> | " | \
```

d. Assignment Statement

```
<assignment_stmt> ::= <var_name> <assignment_op> <assignment_expr>
<assignment_expr> ::= <var_name> | <arithm_expr> | <constant>
                        | <function_call> | <input_stmt>
```

e. Post Statements

```
<post_incr> ::= <var_name> ++
<post_decr> ::= <var_name> --
```

f. Variable Declaration Statement

```
<declaration> ::=  var <var_name> | var <var_name> = <assignment_expr>
```

g. Variable Identifier

```
<var_name> ::= <initial> | <initial> <rest>
```

<initial> ::= <str_letter> | _
<rest> ::= <str_letter>| <digit> | <str_letter> <rest> | <digit><rest>
<str_letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x
                 | y | z | A | B | C | D | E| F | G | H | I | J | K | L | M | N | O | P | Q | R | S
                 | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<constant> ::= <int> | <string> | <boolean>

<boolean> ::= true | false

<string> ::= "<valid_str>"
             | ""
<valid_str> ::= <letter_digit_symbol>
                 | <letter_digit_symbol> <valid_str>
<letter_digit_symbol> ::= <str_letter> | <digit> | <symbol>
                 | <assignment_op> | <sign> | <semicolon>

<sign> ::= <plus> | <minus>
<int> ::= <sign_int> | <nosign_int>
<sign_int> ::= <sign> <nosign_int> | <nosign_int>
<nosign_int> ::= <digit> <nosign_int> | <digit>

h. Function Call Statement

<function_call> ::= <func_name> <LP> <RP>
                 | <func_name> <LP> <defined_parameters> <RP>
                 | <primitive_func_call>

i. Loop Statements (For and While)

<for_stmt> ::= for <LP> <for_initial> <semicolon> <relational_expression><semicolon> <for_update> <RP> <block_statement>
        | for <LP> <semicolon> <relational_expression><semicolon> <for_update> <RP> <block_statement>

<for_initial> ::= <for_initial_helper> | <for_initial_helper> <comma> <for_initial>

<for_initial_helper> ::= var <var_name> = <var_name>
                 | var <var_name> = <arithm_expr>
                 | var <var_name> = <int>
                 | <var_name> = <var_name>
                 | <var_name> = <arithm_expr>
                 | <var_name> = <int>

<while_stmt> ::= while <LP> <condition> <RP> <block_statement>

j. Conditional Statement (If Statement)

<conditional_stmt> ::= if <LP> <condition> <RP> <block_statement>
                                | if <LP> <condition> <RP> <block_statement>else
                                  <block_statement>

<condition> ::= <relational_expr> | <boolean_expr>

k. Input & Output Statements

<input> ::= drin <LP> <RP>
<output> ::= drout <LP> <var_name> <RP> | drout <LP> <constant> <RP>

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

<function_no_return> ::= drfunction <func_name> <LP> <parameters>
                                <RP> <block_statement>
        | drfunction <func_name> <LP><RP> <block_statement>

<function_return> ::= drfunction <func_name> <LP> <parameters> <RP>
<LBRACE> <inside_statements> return <constant> <semicolon> <RBRACE>
                | drfunction <func_name> <LP> <parameters><RP> <LBRACE>
                return <constant><semicolon> <RBRACE>
                | drfunction <func_name> <LP> <parameters> <RP> <LBRACE>
                <inside_statements> return <var_name> <semicolon> <RBRACE>
                | drfunction <func_name> <LP> <parameters> <RP> <LBRACE>
                return <var_name> <semicolon> <RBRACE>
                | drfunction <func_name> <LP> <RP> <LBRACE>
                <inside_statements> return <constant> <semicolon> <RBRACE>
                | drfunction <func_name> <LP> <RP> <LBRACE>
                return <constant><semicolon> <RBRACE>
                | drfunction <func_name> <LP> <RP> <LBRACE>
                <inside_statements> return <var_name> <semicolon> <RBRACE>
                | drfunction <func_name> <LP> <parameters> <RP> <LBRACE>
                return <var_name> <semicolon> <RBRACE>

<parameters> ::= var <var_name>
                | <int>
                | <string>
                | <int> <comma> <parameters>
                | <string> <comma> <parameters>
                | var <var_name> <comma> <parameters>

```
<defined_parameters> ::= <var_name>
                       | <int>
                       | <string>
                       | <string> <comma> <defined_parameters>
                       | <int> <comma> <defined_parameters>
                       | <var_name>  <comma> <defined_parameters>

<func_name> ::= <str_letter> | <str_letter> <rest>
```

m. Expressions

```
<expression> ::= <arithm_expr> | <relational_expr>
               | <boolean_expr> | <comb_exp>

<arithm_expr> ::= <term>
                | <arithm_expr> <plus> <term>
                | <arithm_expr> <minus> <term>
<term> ::= <power> | <term> <multiplication_op> <power>
         | <term> <division_op> <power>
<power> ::= <factor> <power_op> <power> | <factor>
<factor> ::=  <LP>  <arithm_expr>  <RP> | <int> | <var_name>


<relational_expr> ::= <var_name> <relational_opr> <var_name>
                    | <var_name> <relational_opr> <int>
                    | <int> <relational_opr> <var_name>

<relational_opr> ::=  <smaller_than> | <smaller_or_equals> | <larger_than>
                    | <larger_or_equals> | <equals> | <not_equals>

<boolean_expr> ::= <var_name> <boolean_operator> <var_name>
                 | <LP> <boolean_expr> <RP> <boolean_expr>
                 | <boolean_expr> <boolean_operator> <boolean_expr>
                 | <boolean_expr> NOT <boolean_expr>
                 | <LP> <boolean_expr> <RP>
                 | <var_name>
                 | NOT <var_name>
                 | NOT

<boolean_operator> ::= AND | OR

<comb_expr> ::= <relational_expr> <boolean_operator> <boolean_expr>
              | <comb_expr> <boolean_operator> <LP> <relational_expr> <RP>
              | <comb_expr> <boolean_operator> <boolean_expr>
              | <LP> <comb_expr> <RP> <boolean_operator> <boolean_expr>
```

```
                        | <relational_expr> | <boolean_expr> | <var_name>
```

n. 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> | <goto>
                        | <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> ::= cameraOn <LP>  <RP>

<camera_off> ::= cameraOff <LP>  <RP>

<take_picture> ::= takePicture <LP>  <RP>

<read_time> ::= readTime <LP>  <RP>

<connect_wifi> ::= connect <LP>  <string>  <comma> <string>  <RP>

<set_speed> ::= setSpeed <LP>  <int>  <RP>

<set_altitude> ::= setAltitude <LP>  <int>  <RP>

<set_incl> ::= setIncline <LP>  <int>  <RP>

<drone_up> ::= droneUp <LP>  <int>  <RP>

<drone_down> ::= droneDown <LP>  <int>  <RP>

<drone_right> ::= droneRight <LP>  <int>  <RP>

<drone_left> ::= droneLeft <LP>  <int>  <RP>

<drone_forward> ::= droneForward <LP>  <int>  <RP>
```

<drone_backward> ::= droneBackward <LP> <int> <RP>

<rotate_clockwise> ::= rotateCW <LP> <int> <RP>

<rotate_counterclockwise> ::= rotateCCW <LP> <int> <RP>

<goto> ::= goto <LP> <int> <comma> <int> <comma> <int> <comma> <int> <RP>

<lock_position> ::= lockPosition<LP><RP>

<unlock_position> ::= unlockPosition<LP><RP>

<give_name> ::= giveName<LP><string><RP>

o. Symbols

<semicolon> ::= ;

<comma> ::= ,

<assignment_op> ::= =

<LP> ::= (

<RP> ::=)

<LBRACE> ::= {

<RBRACE> ::= }

<multiplication_op> ::= *

<division_op> ::= /

<power_op> ::= ^

<plus>::= +

<minus>::= -

<smaller_than> ::= <

<smaller_or_equals>::= <=

<larger_than>::= >

<larger_or_equals>::= >=

<equals>::= ==

<not_equals>::= !=

<symbol> ::= ! | % | & | £ | # | $ | ½ | [|] | _ | \| | : | .

III. DESCRIPTIONS OF DR-ONE CONSTRUCTS

1. <program> ::= start <statements> stop

The program is enclosed with two reserved words which are start and stop. 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> <semicolon>
| <statements> <statement> <semicolon>
| <statements> <comment_block> | <comment_block>

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>
| <function_call> | <declaration> | <conditional_stmt>
| <while_stmt> | <for_stmt> | <function_no_return>
| <function_return> | <input> | <output>

A statement can be one of ten 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. 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> <semicolon>
| <inside_statements> <inside_statement> <semicolon>

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>
| <function_call> | <declaration> | <conditional_stmt>
| <while_stmt> | <for_stmt> | <input> | <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_block> ::= *CM* <anything_in_lng> *CM*

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.

8. <anything_in_lng> ::= <any_symbol> | <letter_digit_symbol> <anything_in_lng>

This non-terminal specifies what can be inside a comment. Any symbol, any letter and digit can be a part of it. It is built recursive to construct a long comment (more than a character).

**9. <any_symbol> ::= <symbol> | “ | **

Any symbol is a symbol plus the quotation mark and the escape character. The reason why any symbol and symbol is defined separately is that symbols that can be in a string (word) and symbols that can be inside a comment differs. Any symbol is used for comment definition and symbol is used for string definition since strings cannot have quotation marks inside but the string is embraced with quotation marks.

10. <assignment_stmt> ::= <var_name> <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 (=) and the right hand side (the value assigned to subject/variable, an expression).

11. <assignment_expr> ::= <var_name> | <arithm_expr> | <constant>
| <function_call> | <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, a constant value, a primitive function's return value, and a user input (via input statement).

12. <post_incr> ::= <var_name>++

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.

13. <post_decr> ::= <var_name>--

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.

14. <declaration> ::= var <var_name> | var <var_name> = <assignment_expr>

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 expression (i.e. var varName;), second, "var" followed by the name of the variable with assignment expression (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.

15. <var_name> ::= <initial> | <initial><rest>

Variable names could start with a letter, or _ (underscore) sign. Underscore could only be at the beginning, digits can not be at the beginning of a variable name. For this reason, the initial character of the variable name is treated differently than the rest of the characters in the identifier. It is a recursive definition.

16. <initial> ::= <str_letter> | _

The initial character of the variable identifier can either be a letter or an underscore.

17. <rest> ::= <str_letter>| <digit> | <str_letter> <rest> | <digit><rest>

The rest of the characters of a variable identifier can be a letter or a digit. Therefore, it is made recursive for long identifiers.

18. <str_letter> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w
| x | y | z | A | B | C | D | E| F | G | H | I | J | K | L | M | N | O | P
| Q | R | S | T | U | V | W | X | Y | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

These are the definitions for a letter and a digit. Letters are all lower case and upper case letters in alphabet and all digits.

19. <constant> ::= <int> | <string> | <boolean>

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

20. <boolean> ::= true | false

Booleans are truth values. Boolean value can either be true or false. Thus, true and false are reserved words.

21. <string> ::= "<valid_str>"
| ""

A string (a word) is a series of characters (specific/valid characters) embraced by quotation marks. It can also be empty (nothing between the quotation marks).

22. <valid_str> ::= <letter_digit_symbol>
| <letter_digit_symbol> <valid_str>
<letter_digit_symbol> ::= <str_letter> | <digit> | <symbol>
| <assignment_op> | <sign> | <semicolon>

To construct a valid string, a string must be composed of letters, digits or symbols. Symbols are specific set of symbols as explained in previous section which does not contain a quotation mark and an escape character. Additionally to defined set of symbols, a string may contain an assignment operator (=), semicolon and signs (+ and -).

23. <sign> ::= <plus> | <minus>
<int> ::= <sign_int> | <nosign_int>
<sign_int> ::= <sign> <nosign_int> | <nosign_int>
<nosign_int> ::= <digit> <nosign_int> | <digit>

The first definition is the sign definition which includes a minus sign and a plus sign. This definition is used to define signed and unsigned integers since an integer can either be signed or unsigned. A signed integer is an unsigned integer with a sign in the beginning. An unsigned integer lacks any sign. They are recursive. No sign means the integer is positive.

24. <function_call> ::= <func_name> <LP> <RP>
| <func_name> <LP> <defined_parameters> <RP>
| <primitive_func_call>

A function call is made by writing the function name first followed by parantheses. 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.

25. <for_stmt> ::= for <LP> <for_initial> <semicolon><relational_expression>
<semicolon> <for_update> <RP> <block_statement>
| for <LP> <semicolon> <relational_expression><semicolon>
<for_update> <RP> <block_statement>

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 = 1; 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 relational 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 relational expression holds no longer true, the loop is left.

26. <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 seperated by commas.

27. <for_initial_helper> ::= var <var_name> = <var_name>
| var <var_name> = <arithm_expr>
| var <var_name> = <int>
| <var_name> = <var_name>
| <var_name> = <arithm_expr>
| <var_name> = <int>

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.

28. <while_stmt> ::= while <LP> <condition> <RP> <block_statement>

While statements start with "while" reserved word followed by opening parenthesis and the condition and closing parenthesis. It takes a block of statements inside. The condition is a relational expression or a boolean expression, and when that expression no longer holds true, loop is left.

29. <conditional_stmt> ::= if <LP> <condition> <RP> <block_statement>
| if <LP> <condition> <RP> <block_statement>else <block_statement>
<condition> ::= <relational_expr> | <boolean_expr>

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 relational expression or a boolean expression.

30. <input> ::= drin <LP> <RP>
<output> ::= drout <LP> <var_name> <RP> | drout <LP> <constant> <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. Similarly, drout function can be used for the output. To output string, integer, boolean, or a variable they should be written inside parenthesis following drout. For instance, drout("hello world") will output the string.

31. <function_no_return> ::= drfunction <func_name> <LP> <parameters> <RP> <block_statement>
| drfunction <func_name> <LP> <RP><block_statement>

<function_return> ::= drfunction <func_name> <LP> <parameters> <RP> <LBRACE> <inside_statements> return <constant> <semicolon> <RBRACE>
| drfunction <func_name> <LP> <parameters><RP> <LBRACE> return <constant><semicolon> <RBRACE>
| drfunction <func_name> <LP> <parameters> <RP><LBRACE> <inside_statements> return <var_name> <semicolon> <RBRACE>
| drfunction <func_name> <LP> <parameters> <RP> <LBRACE> return <var_name> <semicolon> <RBRACE>
| drfunction <func_name> <LP> <RP> <LBRACE> <inside_statements> return <constant> <semicolon> <RBRACE>

| drfunction <func_name> <LP> <RP> <LBRACE> return <constant><semicolon> <RBRACE>
| drfunction <func_name> <LP> <RP> <LBRACE> <inside_statements> return <var_name> <semicolon> <RBRACE>
| drfunction <func_name> <LP> <parameters> <RP> <LBRACE> return <var_name> <semicolon> <RBRACE>
<func_name> ::= <str_letter> | <str_letter> <rest>

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 could only be composed of digits and letters. Initial character must be a letter for the function name. Rest can be either letter or digit. The function can either have a return or not. Return is a reserved word for returning a value of a variable or a constant.

32. <parameters> ::= var <var_name>
| <int>
| <string>
| <int> <comma> <parameters>
| <string> <comma> <parameters>
| var <var_name> <comma> <parameters>

<defined_parameters> ::= <var_name>
| <int>
| <string>
| <string> <comma> <defined_parameters>
| <int> <comma> <defined_parameters>
| <var_name> <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.

33. <expression> ::= <arithm_expr> | <relational_expr>
| <boolean_expr> | <comb_exp>

An expression can either be an arithmetic expression, a relational expression, a boolean expression or a combination of them (combination of boolean and relational).

```
34.   <arithm_expr> ::= <term>
                    | <arithm_expr> <plus> <term>
                    | <arithm_expr> <minus> <term>
      <term> ::= <power> | <term> <multiplication_op> <power>
               | <term> <division_op> <power>
      <power> ::= <factor> <power_op> <power> | <factor>
      <factor> ::=  <LP>  <arithm_expr>  <RP> | <int> | <var_name>
```

Arithmetic expressions can include addition, subtraction, multiplication, division, and power operations. These operations could be combined. An arithmetic expression has terms. To avoid ambiguity and obey mathematical rules like precedence and association, the BNF definition is made accordingly. 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 or a variable (that has an integer value).

```
35.   <relational_expr> ::= <var_name> <relational_opr> <var_name>
                          | <var_name> <relational_opr> <int>
                          | <int> <relational_opr> <var_name>

      <relational_opr> ::=  <smaller_than> | <smaller_or_equals> | <larger_than>
                          | <larger_or_equals> | <equals> | <not_equals>
```

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 could be between two integer variables or one integer variable and an integer constant.

```
36.   <boolean_expr> ::=
                  | <var_name> <boolean_operator> <var_name>
                  | <var_name>
                  | <LP> <boolean_expr>  <RP> <boolean_operator> <var_name>
                  | NOT <var_name>
                  | NOT <LP> <boolean_expr>  <RP> <boolean_operator>
                    <var_name>
                  | NOT <LP> <boolean_expr>  <RP> <boolean_operator>
                    <var_name>
                  | <LP> <boolean_expr>  <RP> <boolean_operator> NOT
                    <var_name>
                  | <boolean_expr> <boolean_operator> <var_name>
                  | NOT <boolean_expr> <boolean_operator> <var_name>
                  | <boolean_expr> <boolean_operator> NOT <var_name>
                  | NOT <boolean_expr> <boolean_operator> NOT <var_name>
```

<boolean_operator> ::= AND | OR

Boolean 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.

37. <comb_expr> ::= <relational_expr> <boolean_operator> <boolean_expr>
| <comb_expr> <boolean_operator> <LP> <relational_expr><RP>
| <comb_expr> <boolean_operator> <boolean_expr>
| <LP> <comb_expr> <RP> <boolean_operator> <boolean_expr>
| <relational_expr> | <boolean_expr> | <var_name>

For instance, (var1 AND var2) OR (var3 <= var4) is a combined expression. This definition is made so that relational and boolean expressions can be combined. This expression yields a truth/boolean value.

38. <primitive_func_call> ::= <drone_start> | <drone_stop> | <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> | <goto>
| <lock_position> | <unlock_position> | <give_name>

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

<u>Primitive Functions:</u>

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.

goto(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:

Terminals are the symbols in section 2 of the report. 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 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, 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 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 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. EVALUATION OF DR-ONE

a. Readability

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. It is also easy to learn since its constructs resemble other programming languages like Java and Python, and its small number of constructs also fastens the learning process of the language. Although it has feature multiplicity (the post increment and decrement feature as shortcuts for arithmetic expressions), there are not many instances of feature multiplicity unlike Java to complicate the readability. DR-ONE only has post increment and decrement, just two instances of alternatives, which does not affect simplicity and readability poorly. There are not any operator overloading creations in DR-ONE, therefore, its readability is not reduced. This is a functional language which makes it readable again.

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.

In terms of *syntax design,* there are special words of which its list is given in the previous section. The reserved words indicate the meaning of the context the reserved word is used like “if” word to indicate a conditional statement. The statements’ form matches/indicates their meaning, too. For example, drfunction followed by function name and parameters indicate a function definition, and when drfunction word is not present but the structure is similar, it is easily understood that it is a function call. For these reasons, DR-ONE serves exactly its purpose in a clear and readable way.

b. Writability

In terms of *simplicity and orthogonality* again, there are sufficient amounts of constructs and combinations of them to avoid having to learn all uses of constructs which might lead to misuse.

There is a decent example of *expressivity* in DR-ONE, which is the shortcuts for some arithmetic operations like summation and subtraction. One can write the following:

```
dummyVar = dummyVar + 1;                    dummyVar = dummyVar - 1;

dummyVar++;                                 dummyVar--;
```

The grouped two expressions mean the same thing. However, instead of writing it in full (repeating the variable name again), a much shorter post expression is written. Thus, the writability of the language is increased and DR-ONE applies to writability criteria successfully.

c. Reliability

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.

Since DR-ONE’s writability is good, it is easy to write the program, and therefore, it is likely to run correctly and smoothly. Since its readability is also good, it is hard to misuse the constructs, and therefore, results are likely to be expected outcomes (no complications). This also applies to good reliability.

Additionally, ambiguity is eliminated in evaluating expressions in DR-ONE (especially in arithmetical operations), so no expression/statement yields unexpected outcomes or more than one result for the same expression, so the language performs to its specifications. 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.

NOTES ABOUT LEX FILE:

In order to run our program, we have prepared 3 different input.txt example files. Our sample program which can be generated by the codes **lex drone.l** and **gcc -o output lex.yy.c** runs these 3 input example files and prints the output of all example programs at once to the console for the sake of efficiency. Thus, our lex file is modified to generate tokens of all example files at once. All outputs are clearly separated by ***END OF INPUT FILE*** sentences.