# CS315


## Fall 2021

## Programming Language: Droneque

## Team Members

**Arman Engin Sucu, ID: 21801777**

**Deniz Semih Ozal, ID: 21802414**

**Instructor: Karani Kardas**

# Name of The Language: Droneque

# BNF Description of Droneque Language

## 1. Types and Constants

<identifier> ::= IDENTIFIER | IDENTIFIER COMMA <identifier>

<arithmetic_operators> ::= ADDITION_OP | SUBTRACTION_OP |
MULTIPLICATION_OP | DIVISION_OP | MODULO_OP |
EXPONENTIATION_OP

<relational_operators> ::= LESS_THAN_OP | LESS_OR_EQUAL_OP |
GREATER_THAN_OP | GREATER_OR_EQUAL_OP |
EQUAL_OP | NOT_EQUAL_OP

<assignment_operators> ::= ASSIGN_OP | ADDITION_ASSIGNMENT_OP |
SUBTRACTION_ASSIGNMNET_OP | DIVISION_ASSIGNMENT_OP
| MULTIPLICATION_ASSIGNMENT_OP

## 2. Program Definition

<program> ::= <main>

<main> ::= LP RP LB <statements> RB

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

<statement> ::= <comment> | <expression> END_STATEMENT | <loop_statement> |
<function_statement> | <conditional_statement>

<comment> ::= <single_line_comment> | <multiple_line_comment>

<sentence> ::= ANY_CHARACTER_EXCEPT_NEW_LINE

<sentences> ::= <sentence> | <sentence> NEWLINE <sentences>

<single_line_comment> ::= COMMENT_SIGN <sentence>

<multiple_line_comment> ::= COMMENT_SIGN MULTIPLICATION_OP
<sentences> MULTIPLICATION_OP COMMENT_SIGN

<expression> ::= <var_declaration> | <integer_expression> | <boolean_expression> |
<string_expression>| <arithmetic_expression> | <relational_expression>|
<bitwise_expression> |<assignment_operator> | <increment_expression>
| <decrement_expression>

## 3. Blocks and Arguments

<argument_type> ::= IDENTIFIER | BOOLEAN | INTEGER | STRING

<block_statements> ::= NULL | <statements> return <args_type> | <statements>

## 4. Loop Statements

<loop_statement>::= <while_statement> | <for_statement>

<while_statement> ::= WHILE LP (<relational_expression> | <boolean_expression>
| <bitwise_expression>) RP LB <block_statements> RB

<for_statement> :: FOR LP <integer_expression> END_STATEMENT
<relational_expression> END_STATEMENT <arithmetic_expression> |
<increment_expression> | <decrement_expression> RP
LB <block_statements> RB

## 5. Conditional Statements

<conditional_statement> ::= <if_statement> | <conditional_statement>
<else_if_statement> | <if_statement> <else_statement> |
<else_if_statement> <else_statement>

<if_statement> ::= IF LP (<relational_expression> |<boolean_expression> |
<bitwise_expression>) RP LB <block_statements> RB
(<else_if_statement> <else_statement> | <null_statement>)

<else_if_statement> ::= ELSE IF LP (<relational_expression> |<boolean_expression>
|<bitwise_expression>) RP LB <block_statements> RB
(<else_if_statement> <else_statement> | <null_statement>)

<else_statement> ::= ELSE LB <block_statements> RB

## 6. Function Statements

<var_declaration_list> ::= <var_declaration> COMMA <var_declaration_list> |
<var_declaration>

<composite_arguments> ::= <argument_type> COMMA <composite_arguments> |
<argument_type>

<function_declaration> ::= FUNCTION IDENTIFIER LP <var_declaration_list> RP
LB <block_statements> RB

<function_call> ::= IDENTIFIER LP <composite_arguments> RP |
<built_in_function> LP RP

<function_assignment> ::=  IDENTIFIER ASSIGN_OP <function_call> |
<var_declaration> ASSIGN_OP <function_call>

<build_in_function> ::= READ_HEADING |
READ_ALTITUDE |
READ_TEMPERATUR |
VERTICALLY_CLIMB_UP |
VERTICALLY_DROP_DOWN |
VERTICALLY_STOP |
HORIZONTALLY_MOVE_FORWARD |
HORIZONTALLY_MOVE_BACKWARD |
HORIZONTALLY_MOVE_STOP| TURN_LEFT |
TURN_RIGHT |
SPRAY_ON |
SPRAY_OFF |
CONNECT_WITH_COMPUTER |
CONNECT_WITH_MOBILE_DEVICE |
INPUT |
PRINT |
EXIT

## 7. Expressions

<var_declaration> ::= VAR <identifier>

<integer_expression> ::= IDENTIFIER <assignment_operators> INTEGER |
<var_decleration> ASSIGN_OP INTEGER

<boolean_expression> ::= IDENTIFIER ASSIGN_OP BOOLEAN |
<var_decleration> ASSIGN_OP BOOLEAN

<string_expression> ::= IDENTIFIER ASSIGN_OP STRING |
<var_decleration> ASSIGN_OP STRING

<arithmetic_expression>  ::= INTEGER <arithmetic_operators>
<arithmetic_expression> |  INTEGER

<relational_expression>  ::= INTEGER <relational_operators> INTEGER |
IDENTIFIER <relational_operators> IDENTIFIER | STRING
<relational_operators> STRING | BOOLEAN <relational_operators>
BOOLEAN

<bitwise_expression> ::= BOOLEAN LOGICAL_OR_OP BOOLEAN |
BOOLEAN LOGICAL_AND_OP BOOLEAN

<increment_expression>  ::= <preincrement_expr> | <postincrement_expr>

<preincrement_expr> ::= ADDITION_OP ADDITION_OP INTEGER

<postincrement_expr> ::= INTEGER ADDITION_OP ADDITION_OP

<decrement_expression> ::= <predecrement_expr> | <postdecrement_expr>

<predecrement_expr> ::= SUBTRACTION_OP SUBTRACTION_OP INTEGER

<postdecrement_expr> ::= INTEGER SUBTRACTION_OP SUBTRACTION_OP

## Explanations Of Language Constructs

- Droneque has 3 inbuilt types: Integers, Strings and Booleans.

## 1) Types

- **<identifier> ::= IDENTIFIER | IDENTIFIER COMMA <identifier>**
  - In droneque multiple variable declarations can be made such as; var a, b, c.

- **<arithmetic_operators> ::= ADDITION_OP | SUBTRACTION_OP | MULTIPLICATION_OP | DIVISION_OP | MODULO_OP | EXPONENTIATION_OP**
  - In droneque arithmetic operators are addition, subtraction, multiplication, division, modula and exponentiation

- **<relational_operators> ::= LESS_THAN_OP | LESS_OR_EQUAL_OP | GREATER_THAN_OP | GREATER_OR_EQUAL_OP | EQUAL_OP | NOT_EQUAL_OP**
  - In droneque there are 6 relational operators. They are <, <=, >=, ==, !=.

## 2) Program Definition

- **<program> ::= <main>**
  - In droneque to have a valid program structure, program should be constructed in main.

- **<main> ::= LP RP LB <statements> RB**
  - Main has statements in it

- **<statement> ::= <comment> | <expression> END_STATEMENT | <loop_statement> | <function_statement> | <conditional_statement>**
  - Statement can be composed from comments, expressions, loops, functions and conditional statements.

- **<comment> ::= <single_line_comment> | <multiple_line_comment>**
  - There are 2 types of comments: single line comment which has the structure of # and multiple line comment which has the structure of #* *#.

- **<single_line_comment> ::= COMMENT_SIGN <sentence>**

    - Single line comment starts with # and then single line sentence is written

- **<multiple_line_comment> ::= COMMENT_SIGN MULTIPLICATION_OP**
    - Multiple line comment starts with #* and then multiple line sentences is written, finally multiple line comment is terminated with *#

- **<sentence> ::= ANY_CHARACTER_EXCEPT_NEW_LINE**
    - Sentence composed of any character except new line

- **<sentences> ::= <sentence> | <sentence> NEWLINE <sentences>**
    - Sentences are multiple lines of sentence

- **<expression> ::= <var_declaration> | <integer_expression> | <boolean_expression> | <string_expression> | <arithmetic_expression> | <relational_expression> | <bitwise_expression> | <assignment_operator> | <increment_expression> | <decrement_expression>**
    - Expressions are essential parts of the statements and there are different types of expressions. Droneque has 3 different data types which are integer, boolean and strings. All three expressions that are integer, boolean and string expressions are assignment expressions.

## 3) Blocks And Arguments

- **<argument_type> ::= IDENTIFIER | BOOLEAN | INTEGER | STRING**
    - Argument type is used in parameters in the functions and arguments types are can be identifier, boolean, integer and string.

- **<block_statements> ::= NULL | <statements> return <args_type> | <statements>**
    - Block statements take place after some statements such as conditional statements, loop statements and function statements. In Droneque language, The inner part of block statements could be simply null, or it could return a value which is an argument type, or it could have statements.

## 4) Loop Statements

- **<loop_statement>::= <while_statement> | <for_statement>**
    - In droneque there are 2 loop statements: first one is while loop and other is for loop.

- **<while_statement> ::= WHILE LP (<relational_expression> | <boolean_expression> | <bitwise_expression>) RP LB <block_statements> RB**
    - While loop continues until relational, boolean or bitwise operation has become invalid. While has block statements in it. The structure of while is: while(one of the epressionse){<block_statemnts>}

- **<for_statement> :: FOR LP <integer_expression> END_STATEMENT <relational_expression> END_STATEMENT <arithmetic_expression> | <increment_expression> | <decrement_expression> RP LB <block_statements> RB**
    - For loop has 3 declaration parts first for integer expression such as, var a = 6. The other for relational expression such as, a != 8. The last one to change the var a. For example, a++ or a = a / 5. Inside of the for loop there are block statements.

## 5) Conditional Statements

- **<conditional_statement> ::= <if_statement> | <conditional_statement> <else_if_statement> | <if_statement> <else_statement> | <conditional_statement><else_if_statement> <else_statement>**
    - Conditional statement composed from 3 statements: if statement, else if statement and else statement. Conditional statements can have multiple if statements, multiple number of else if statements with the condition of 1 if statement has already set up or else expression after if or else if.

- **<if_statement> ::= IF LP (<relational_expression> |<boolean_expression> |<bitwise_expression>) RP LB <block_statements> RB (<else_if_statement> <else_statement> | <null_statement>)**
    - If statement has a condition part that composed from relational, boolean and bitwise expressions. For example if(3 < 4), if(true != false) and if(true or false) are sample conditions for if statement. If the condition is a valid then block statement will be executed.

- **<else_if_statement> ::= ELSE IF LP (<relational_expression> |<boolean_expression> |<bitwise_expression>) RP LB <block_statements> RB (<else_if_statement> <else_statement> | <null_statement>)**
    - Else if statemen has relational boolean and bitwise expression in condition part like if statements. If the condition is valid then block statement will be executed.

- **<else_statement> ::= ELSE LB <block_statements> RB**
    - Else statement does not have a conditional part. Else statment's block statement will be executed if if statement does not have a valid condition.

## 6) Function Statements

- **<var_declaration_list> ::= <var_declaration> COMMA <var_declaration_list> | <var_declaration>**
    - Var declaration list for parameters of the functions. In our language, function parameters are described in the following way (var a, var b, …). *var* includes and identifier

- **<composite_arguments> ::= <argument_type> COMMA <composite_arguments> | <argument_type>**

- Composite arguments for passed arguments into the functions and they emerge in function calls like in following way (a, b, c)

- **<function_declaration> ::= FUNCTION IDENTIFIER LP <var_declaration_list> RP LB <block_statements> RB**
  - In our language, function declaration is performed by the keyword *function* and it is followed by an identifier. Functions could be void or another return type and they have block statements after function declaration.

- **<function_call> ::= IDENTIFIER LP <composite_arguments> RP | <built_in_function> LP RP**
  - After a function call, some values like integer, boolean or string could return. Our function call structure starts with identifier, then composite arguments are aligned. Another possibility is that our language has already built in functions and these functions could be called.

- **<function_assignment> ::= IDENTIFIER ASSIGN_OP <function_call> | <var_declaration> ASSIGN_OP <function_call>**
  - Function assignment is used for assigning a return value of function to an identifier or a declared variable

- **<build_in_function> ::= READ_HEADING |**
  **READ_ALTITUDE |**
  **READ_TEMPERATURE |**
  **VERTICALLY_CLIMB_UP |**
  **VERTICALLY_DROP_DOWN |**
  **VERTICALLY_STOP |**
  **HORIZONTALLY_MOVE_FORWARD |**
  **HORIZONTALLY_MOVE_BACKWARD |**
  **HORIZONTALLY_MOVE_STOP |**
  **TURN_LEFT |**
  **TURN_RIGHT |**
  **SPRAY_ON |**
  **SPRAY_OFF |**
  **CONNECT_WITH_COMPUTER |**
  **CONNECT_WITH_MOBILE_DEVICE |**
  **INPUT |**
  **PRINT |**
  **EXIT**
  - In droneque language, there are many built in functions for different operations mainly related to features of drone movement.

  - READ_HEADING detects the heading of the drone (an integer value between 0 and 359)

- READ_ALTITUDE measures altitude of the drone
- READ_TEMPERATURE mesaures temperature of the drone

- VERTICALLY_CLIMB_UP to move the drone so that it climbs up with a speed of 0.1 m/s
- VERTICALLY_DROP_DOWN to move the drone so that it drops down with a speed of 0.1 m/s

- VERTICALLY_STOP to stop the drone vertically

- HORIZONTALLY_MOVE_FORWARD to move the drone so that it moves in the heading direction dorawrd with a speed of 1 m/s
- HORIZONTALLY_MOVE_BACKWARD to move the drone so that it moves in the heading direction backward with a speed of 1 m/s

- HORIZONTALLY_MOVE_STOP to stop the drone horizontally

- TURN_LEFT to turn the heading to left for 1 degree
- TURN_RIGHT to turn the heading to right for 1 degree

- SPRAY_ON to open the chemical
- SPRAY_OFF to close the chemical

- CONNECT_WITH_COMPUTER to connect base desktop
- CONNECT_WITH_MOBILE to connect with mobile device

- INPUT to take input from the user
- PRINT to print something in droneque

- EXIT to quit droneque

## 7) Expressions

- **<var_declaration> ::= VAR <identifier>**
  - In our language, variable declaration is performed by *var* keyword and after *var* there should be an identifier.

- **<integer_expression> ::= IDENTIFIER <assignment_operators> INTEGER | <var_decleration> ASSIGN_OP INTEGER**
  - Integer expression is for assignment operation or var declaration

- **<boolean_expression> ::= IDENTIFIER ASSIGN_OP BOOLEAN | <var_decleration> ASSIGN_OP BOOLEAN**
  - Boolean expression is for assignment operation or var declaration

- **<string_expression> ::= IDENTIFIER ASSIGN_OP STRING | <var_decleration> ASSIGN_OP STRING**
  - String expression is for assignment operation or var declaration

- **<arithmetic_expression> ::= INTEGER <arithmetic_operators> arithmetic_expression> | INTEGER**
  - Arithmetic expressions are for numerical operations between integers like addition, subtraction and so on.

- **<relational_expression> ::= INTEGER <relational_operators> INTEGER | DENTIFIER <relational_operators> IDENTIFIER | STRING <relational_operators> STRING | BOOLEAN <relational_operators> BOOLEAN**
  - Relational expressions are for relational operations between integers, booleans and strings like equaility, superority and inferiority

- **<bitwise_expression> ::= BOOLEAN LOGICAL_OR_OP BOOLEAN | BOOLEAN LOGICAL_AND_OP BOOLEAN**
  - Bitwise operations are declared for boolean literals. There are 2 bitwise operations: or, and operations.

- **<increment_expression> ::= <preincrement_expr> | <postincrement_expr>**
  - Increment expressions are 2 types: preincrement or postincrement.

- **<preincrement_expr> ::= ADDITION_OP ADDITION_OP INTEGER**
  - Preincrement expressions increment the integer by 1, before the integer used for another expression. It can be shown by adding 2 addition operatores as a prefix to the variable: ++variable.

- **<postincrement_expr> ::= INTEGER ADDITION_OP ADDITION_OP**
  - Postindrement expressions increment the integer by 1 after the integer used for the main expression. It can be shown by adding 2 additional operators as a postfix to the variable: variable++.

- **<decrement_expression> ::= <predecrement_expr> | <postdecrement_expr>**
    - There are 2 types of decrement expressions: predecrement and postdecrement expressions.

- **<predecrement_expr> ::= SUBTRACTION_OP SUBTRACTION_OP INTEGER**
    - Predecrement expressions decrement the integer by 1, before the integer used for another expression. It can be shown by adding 2 subtraction operators as a prefix to the variable: --variable.

- **<postdecrement_expr> ::= INTEGER SUBTRACTION_OP SUBTRACTION_OP**
    - Postdecrement expressions decrement the integer by 1, after the integer used for the main expression. It can be shown by adding 2 subtraction operators as a postfix to the variable: variable--.

## Conventions of Droneque

- In conditional statements every inbuilt type should be compared with themselves. For example, it is not possible to compare STRING and INTEGER, or BOOLEAN with STRING.

- Increment and decrement expressions can be used for only integer literals.

- In for loops, at the third part of the for loop integers and strings can be used, however booleans can not be used.

- The variables that are not initialized will have NULL value, since randomly initialized values might cause problems.

- Function names should consist of all lower case and words should seperated with "_". For example, droneque.read_heading() or droneque.vertically_drop_down().

- All conditional statements should be accompanied by brackets({}). Therefore, the if statements without brackets are not accessible in droneque such as
  if(condition) statement; instead it should be like this:
  if(condition){statement;}

## Descriptions Of Non-Trivial Tokens

**COMMENT_SIGN:** \#

Single line comments start with #, however, multiple line comments are constructed with MULTIPLICATION_OP: #* *#.

**IDENTIFIER:** [A-Za-z][_A-Za-z0-9]*

Identifiers are used for variable and function decelerations as well as as a function parameter names. Identifiers start with alphabet or underscore(-). Identifiers include only alphanumeric characters(a-z, A-Z, 0-9) and identifiers can not include whitespaces.

The reason behind this is about lexical analysis. If lexical analysis see a number it can not understand if it is a number of identifier. Therefore, it has to do backtracking in these situations. Namely lexical analysis should understand if the token is identifier or a number by checking the first character.

**INTEGER:** [+-]?[0-9]+

Integers can both 0 negative and positive.

**BOOLEAN:** true | false

Booleans are represented with 2 reserved words: true and false.

**STRING:** \"(\\.|[^\\"])*\"

Strings are represented in quotes so that they can be differentiated from the identifiers.

**LB:** \{

**RB:** \}

**LP:** \(

**RP:** \)

**COMMA:** \,

**END_STATEMENT:** \;

**ASSIGN_OP:** \=

**ADDITION_OP:** \+

**SUBTRACTION_OP:** \-

**MULTIPLICATION_OP:** *

**DIVISION_OP:** \ /

**MODULO_OP:** \%

**EXPONENTIATION_OP:** **

**ADDITION_ASSIGNMENT_OP:** \+\=

**SUBTRACTION_ASSIGNMNET_OP:** \-\=

**DIVISION_ASSIGNMENT_OP:** \ / \=

**MULTIPLICATION_ASSIGNMENT_OP:** *\=

**EQUAL_OP:** \=\=

**NOT_EQUAL_OP:** \!\=

**LESS_THAN_OP:** \<

**GREATER_THAN_OP:** \>

**GREATER_OR_EQUAL_OP:** \>\=

**LESS_OR_EQUAL_OP:** \<\=

**WHITESPACE:** [ \t]+

**NEWLINE:** (\r|\n|\r\n)

**ANY_CHARACTER_EXCEPT_NEW_LINE:** .*

**LOGICAL_OR_OP:** or

**LOGICAL_AND_OP:** and

**IF:** if

**ELSE:** else

**ELSE_IF:** else if

**VAR_TYPE:** var

**WHILE:** while

**FOR:** for

**RETURN:** return

**FUNCTION:** function

**NULL:** ^[A-Za-z][_A-Za-z0-9]*

**READ_HEADING:** droneque\.read_heading

**READ_ALTITUDE:** droneque\.read_altitude

**READ_TEMPERATURE:** droneque\.read_temperature

**VERTICALLY_CLIMB_UP:** droneque\.vertically_climb_up

**VERTICALLY_DROP_DOWN:** droneque\.vertically_drop_down

**VERTICALLY_STOP:** droneque\.vertically_stop

**HORIZONTALLY_MOVE_FORWARD:** droneque\.horizontally_move_forward

**HORIZONTALLY_MOVE_BACKWARD**:
droneque\.horizontally_move_backward

**HORIZONTALLY_MOVE_STOP:** droneque\.horizontally_stop

**TURN_LEFT:** droneque\.turn_left

**TURN_RIGHT:** droneque\.turn_right

**SPRAY_ON:** droneque\.spray_on

**SPRAY_OFF:** droneque\.spray_off

**CONNECT_WITH_COMPUTER:** droneque\.connect_with_computer

**CONNECT_WITH_MOBILE_DEVICE:** droneque\.connect_with_mobile_device

**INPUT:** droneque.\input

**PRINT:** dronque.\print

**EXIT:** droneque\.exit

## Evaluation of Droneque In Terms Of Readability Writability and Reliability

### 1. Readability

Droneque's bitwise operations are the word representations -and, or- of the operation instead of non-intuitive representations such as && or ||. Functions are declared with "function" prefix which suits the intuition of the user, unlike the function types of lambda in C-type languages. Moreover, comments are significantly important tools in terms of readability, since they explain the program. Furthermore, droneque's conventions and construct are very similar to the natural languages, since its features are high level which increases readability significantly.

### 2. Writability

Variables are declared with the token "var" so that the user does not have to state the inbuilt type of the variable every time. Furthermore, calling inbuilt functions with the prefix of "droneque" is easy to remember, therefore, writing inbuilt functions is easy to write. Moreover, while loops, for loops and conditional statements, separate their condition part from their block statement by with brackets({}) which increase writability, since user know that in the brackets({}) there should be block statements.

### 3. Reliability

One of the important aspects of Droneque in terms of reliability is, it is not an ambiguous language, since in the BNF rules right recursion and left recursion do not occur simultaneously. Moreover, droneque does not allow uninitialized variables to have random values, instead, it initializes them with NULL. Since having a random value would not cause an error it could be used as if the real value and this can cause many troubles in the program. Since this language is going to use for drones it is of utmost importance to have correct values for every variable.