

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_ASSIGNMENT_OP | DIVISION_ASSIGNMENT_OP |
MULTIPLICATION_ASSIGNMENT_OP

<var_type> ::= INT_TYPE | BOOLEAN_TYPE | DOUBLE_TYPE
| STRING_TYPE | VAR_TYPE

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> | NULL

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

$\langle \text{increment_expression} \rangle ::= \langle \text{preincrement_expr} \rangle \mid \langle \text{postincrement_expr} \rangle$
 $\langle \text{preincrement_expr} \rangle ::= \text{ADDITION_OP ADDITION_OP INTEGER}$
 $\langle \text{postincrement_expr} \rangle ::= \text{INTEGER ADDITION_OP ADDITION_OP}$
 $\langle \text{decrement_expression} \rangle ::= \langle \text{predecrement_expr} \rangle \mid \langle \text{postdecrement_expr} \rangle$
 $\langle \text{predecrement_expr} \rangle ::= \text{SUBTRACTION_OP SUBTRACTION_OP INTEGER}$
 $\langle \text{postdecrement_expr} \rangle ::= \text{INTEGER SUBTRACTION_OP SUBTRACTION_OP}$

Explanations Of Language Constructs

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

1) Types

- $\langle \text{identifier} \rangle ::= \text{IDENTIFIER} \mid \text{IDENTIFIER COMMA} \langle \text{identifier} \rangle$
 - In droneque multiple variable declarations can be made such as; var a, b, c.
- $\langle \text{arithmetic_operators} \rangle ::= \text{ADDITION_OP} \mid \text{SUBTRACTION_OP} \mid \text{MULTIPLICATION_OP} \mid \text{DIVISION_OP} \mid \text{MODULO_OP} \mid \text{EXPONENTIATION_OP}$
 - In droneque arithmetic operators are addition, subtraction, multiplication, division, modula and exponentiation
- $\langle \text{relational_operators} \rangle ::= \text{LESS_THAN_OP} \mid \text{LESS_OR_EQUAL_OP} \mid \text{GREATER_THAN_OP} \mid \text{GREATER_OR_EQUAL_OP} \mid \text{EQUAL_OP} \mid \text{NOT_EQUAL_OP}$
 - In droneque there are 6 relational operators. They are <, <=, >=, ==, !=.

2) Program Definition

- $\langle \text{program} \rangle ::= \langle \text{main} \rangle$
 - In droneque to have a valid program structure, program should be constructed in main.
- $\langle \text{main} \rangle ::= \text{LP RP LB} \langle \text{statements} \rangle \text{RB}$
 - Main has statements in it
- $\langle \text{statement} \rangle ::= \langle \text{comment} \rangle \mid \langle \text{expression} \rangle \text{END_STATEMENT} \mid \langle \text{loop_statement} \rangle \mid \langle \text{function_statement} \rangle \mid \langle \text{conditional_statement} \rangle$
 - Statement can be composed from comments, expressions, loops, functions and conditional statements.
- $\langle \text{comment} \rangle ::= \langle \text{single_line_comment} \rangle \mid \langle \text{multiple_line_comment} \rangle$

- 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 expression){<block_statements>}
- <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 statement 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 statement'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 <return_type> 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 | IDENTIFIER**
 - 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
- **<built_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 measures 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 forward 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_declaration> ASSIGN_OP INTEGER**
 - Integer expression is for assignment operation or var declaration
- **<boolean_expression> ::= IDENTIFIER ASSIGN_OP BOOLEAN | <var_declaration> ASSIGN_OP BOOLEAN**
 - Boolean expression is for assignment operation or var declaration
- **<string_expression> ::= IDENTIFIER ASSIGN_OP STRING | <var_declaration> 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 | IDENTIFIER <relational_operators> IDENTIFIER | STRING <relational_operators> STRING | BOOLEAN <relational_operators> BOOLEAN**
 - Relational expressions are for relational operations between integers, booleans and strings like equality, superiority 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 operators 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 be separated 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 (Terminals)

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 declarations 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 or 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.

DOUBLE: [+]?([0-9]*[.])?[0-9]+

Doubles can be both 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: \{

A left bracket is represented { and it is used to follow conventional naming

RB: \}

A right bracket is represented by } and it is used to follow conventional naming

LP: \((

A left parenthesis is represented by (and it is used to follow conventional naming

RP: \)

A right parenthesis is represented by) and it is used to follow conventional naming

COMMA: \,

A comma is represented by , and it is used to follow conventional naming

END_STATEMENT: \;

An end statement indicates end of the statement. It is represented by ; and it is used to follow conventional naming

ASSIGN_OP: \=

Assignment operations occur with assignment operator. It is represented by = and it is used to follow conventional naming

ADDITION_OP: \+

Addition operations occur with addition operator. It is represented by + and it is used to follow conventional naming

SUBTRACTION_OP: \-

Subtraction operations occur with subtraction operator. It is represented by - and it is used to follow conventional naming

MULTIPLICATION_OP: *

Multiplication operations occur with multiplication operator. It is represented by * and it is used to follow conventional naming

DIVISION_OP: \/

Division operations occur with division operator. It is represented by / and it is used to follow conventional naming

MODULO_OP: \%

Modulo operations occur with modulo operator. It is represented by % and it is used to follow conventional naming

EXPONENTIATION_OP: **

Exponentiation operations occur with exponentiation operator. It is represented by ** and it is used to follow conventional naming

ADDITION_ASSIGNMENT_OP: \+|=

Addition assignment operations occur with addition assignment operator. It is represented by += and it is used to follow conventional naming

SUBTRACTION_ASSIGNMENT_OP: \-|=

Subtraction assignment operations occur with subtraction assignment operator. It is represented by -= and it is used to follow conventional naming

DIVISION_ASSIGNMENT_OP: \ / |=

Division assignment operations occur with division assignment operator. It is represented by /= and it is used to follow conventional naming

MULTIPLICATION_ASSIGNMENT_OP: *|=

Multiplication assignment operations occur with multiplication assignment operator. It is represented by *= and it is used to follow conventional naming

EQUAL_OP: \|=

Equal (Boolean) operations occur with equal operator. It is represented by == and it is used to follow conventional naming

NOT_EQUAL_OP: \!|=

Not equal (Boolean) operations occur with equal not equal operator. It is represented by != and it is used to follow conventional naming

LESS_THAN_OP: \<

Less than operations occur with less than operator. It is represented by < and it is used to follow conventional naming

GREATER_THAN_OP: \>

Greater than operations occur with greater than operator. It is represented by > and it is used to follow conventional naming

GREATER_OR_EQUAL_OP: \>|=

Greater or equal operations occur with greater or equal operator. It is represented by >= and it is used to follow conventional naming

LESS_OR_EQUAL_OP: \<|=

Less or equal operations occur with less or equal operator. It is represented by <= and it is used to follow conventional naming

WHITESPACE: [\t]+

Whitespace represents both single whitespace or multiple whitespaces.

NEWLINE: (\r\n\r\n)

Newline represents both single newline or multiple newlines

LOGICAL_OR_OP: or

Logical or operations occur with logical *or* keyword. It is represented by *or* and it is different from conventional naming. The keyword *or* is used to increase readability, and writability.

LOGICAL_AND_OP: and

Logical and operations occur with logical *and* keyword. It is represented by *and* and it is different from conventional naming. The keyword *and* is used to increase readability, and writability.

IF: if

If (conditional) operations occur with *if* keyword. It is represented by *if* and it is used to follow conventional naming

ELSE: else

Else (conditional) operations occur with *else* keyword. It is represented by *else* and it is used to follow conventional naming

ELSE_IF: else if

Else if (conditional) operations occur with *else if* keyword. It is represented by *else if* and it is used to follow conventional naming

VAR_TYPE: var

var is used to declare any type of variable. It is represented by *var* and it is different from conventional naming. The keyword *var* is used to increase readability, and writability

INT_TYPE int

int is used to declare int variable. It is represented by *int* and it is used to follow conventional naming

BOOL_TYPE bool

bool is used to declare boolean variable. It is represented by *bool* and it is used to follow conventional naming

STRING_TYPE string

string is used to declare string variable. It is represented by *string* and it is used to follow conventional naming

DOUBLE_TYPE double

double is used to declare double variable. It is represented by *double* and it is used to follow conventional naming

WHILE: while

While (loop) operations occur with *while* keyword. It is represented by *while* and it is used to follow conventional naming

FOR: for

For (loop) operations occur with *for* keyword. It is represented by *while* and it is used to follow conventional naming

RETURN: return

Return is used to indicate return operation and to follow conventional naming

FUNCTION: function

Function is used to declare a function. It is different from conventional naming because there are no access modifiers in Droneque language. The keyword *function* is used to increase readability, and writability

READ_HEADING: droneque\.read_heading

droneque.read_heading is a built-in function to detect the heading of the drone (an integer value between 0 and 359)

READ_ALTITUDE: droneque\.read_altitude

droneque.read_altitude is a built-in function to measure altitude of the drone

READ_TEMPERATURE: droneque\.read_temperature

droneque.read_temperature is a built-in function to measure temperature of the drone

VERTICALLY_CLIMB_UP: droneque\.vertically_climb_up

droneque.vertically_climb_up is a built-in function to move the drone so that it climbs up with a speed of 0.1 m/s

VERTICALLY_DROP_DOWN: droneque\.vertically_drop_down

droneque.vertically_drop_down is a built-in function to move the drone so that it drops down with a speed of 0.1 m/s

VERTICALLY_STOP: droneque\.vertically_stop

droneque.vertically_stop is a built-in function to stop the drone vertically

HORIZONTALLY_MOVE_FORWARD: droneque\.horizontally_move_forward

droneque.horizontally_move_forward is a built-in function to move the drone so that it moves in the heading direction forward with a speed of 1 m/s

HORIZONTALLY_MOVE_BACKWARD: droneque\.horizontally_move_backward

droneque.horizontally_move_backward is a built-in function to move the drone so that it moves in the heading direction backward with a speed of 1 m/s

HORIZONTALLY_MOVE_STOP: droneque\.horizontally_stop

droneque.horizontally_stop is a built-in function to stop the drone horizontally

TURN_LEFT: droneque\.turn_left

TURN_RIGHT: droneque\.turn_right

SPRAY_ON: droneque\.spray_on

droneque.spray_on is a built function to open a chemical in the drone

SPRAY_OFF: droneque\.spray_off

, droneque.spray_off is a built function to close a chemical in the drone

CONNECT_WITH_COMPUTER: droneque\.connect_with_computer

CONNECT_WITH_MOBILE_DEVICE: droneque\.connect_with_mobile_device

INPUT: droneque\.input

PRINT: droneque\.print

EXIT: droneque\.exit

Non Terminals:

<identifier>:

Identifier represents one identifier or more than one identifier

<arithmetic_operators>:

Arithmetic operators include arithmetic operator terminals

<assignment_operators>:

Assignment operators include assignment operator terminals

<var_type>:

Var type represents data type terminals of droneque language

<program>:

Program starts with a main terminal

<statements>

Statements include statement, or more than one statement

<statement>

Statement includes expressions, conditional statement, loop statement and function statements

<comment>

Comment include single line comment or multiple line comment

<single_line_comment>

Single line comment includes single line comment terminal

<multiple_line_comment>

Multiple line comment includes multiple line comment terminal

<expression>

Expression non terminal includes

<relational_expression> relational expression non terminal includes data types with relational operators and their association

<bitwise_expression> bitwise expression non terminal includes logical or, logical and operators

<increment_expression> preincrement expression nonterminal is used to include both Preincrement and postincrement expressions

<preincrement_expr> pretincrement expression represents ++a

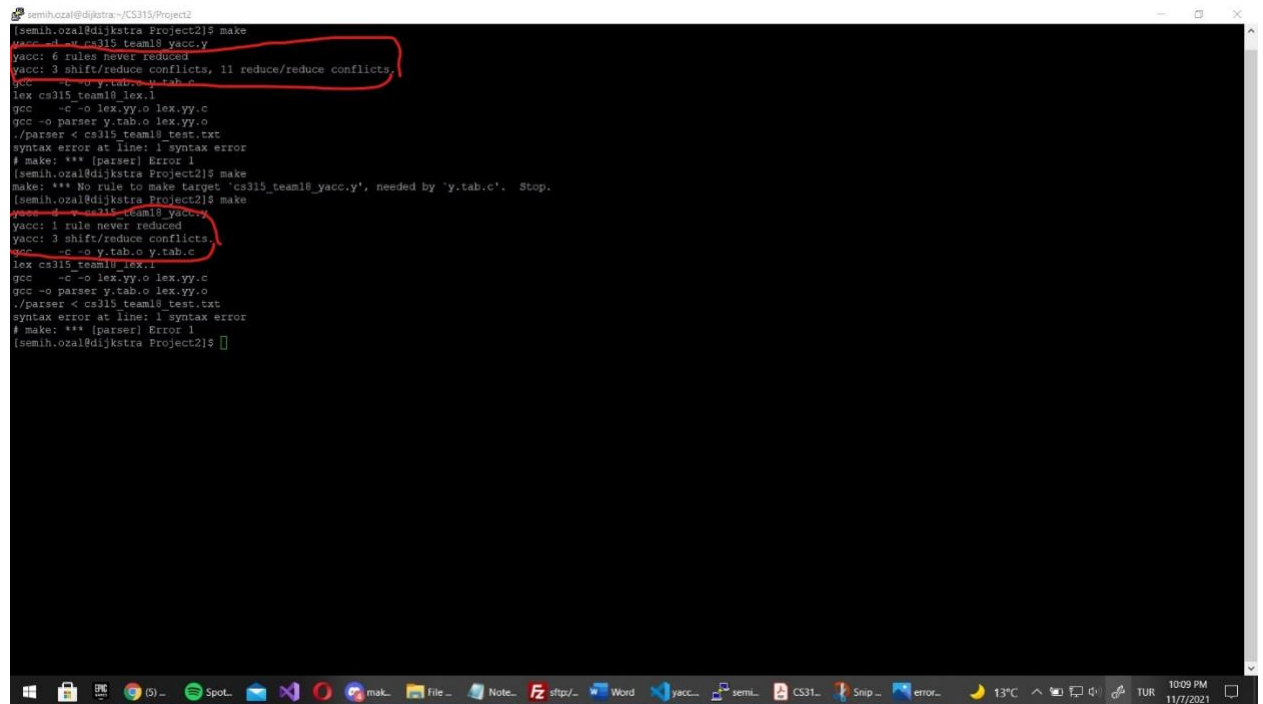
<postincrement_expr> postincrement expression represents a++

<decrement_expression> decrement expression nonterminal is used to include both predecrement and postdecrement expressions

<predecrement_expr> predecrement expression represents --a

<postdecrement_expr> postdecrement expression represents a--

How Conflicts Being Resolved



```
(semih.ozal8dijkstra Project2) make
gcc -c -o cs315_team18 yacc.y
yacc: 6 rules never reduced
yacc: 3 shift/reduce conflicts, 11 reduce/reduce conflicts.
gcc -c -o y.tab.o y.tab.c
lex cs315_team18 lex.l
gcc -c -o lex.yy.o lex.yy.c
gcc -o parser y.tab.o lex.yy.o
./parser < cs315_team18 test.txt
syntax error at line: 1 syntax error
# make: *** [parser] Error 1
(semih.ozal8dijkstra Project2) make
make: *** No rule to make target 'cs315_team18 yacc.y', needed by 'y.tab.c'. Stop.
(semih.ozal8dijkstra Project2) make
yacc: 1 rule never reduced
yacc: 3 shift/reduce conflicts.
gcc -c -o y.tab.o y.tab.c
lex cs315_team18 lex.l
gcc -c -o lex.yy.o lex.yy.c
gcc -o parser y.tab.o lex.yy.o
./parser < cs315_team18 test.txt
syntax error at line: 1 syntax error
# make: *** [parser] Error 1
(semih.ozal8dijkstra Project2) make
```

We have encountered firstly *6 rules never reduced* and *3 shift/reduce and 11 reduce/reduce conflict* problem at first. Rules never used problem primarily caused by duplication of rules in yacc file. When we have eliminated duplication problem, many

problems were resolved naturally. Then, we have only *1 rule never reduced* and *3 shift/reduce* conflicts. It was easy to solve rule never reduced problem but shift/reduce problem takes too much time to resolve. It was produced by identifier and conditional statements rules in yacc file part. To solve identifier problem, we have removed identifier non terminal and benefit from var_declaration nonterminal and IDENTIFIER terminals. Since it had a structure like identifier : IDENTIFIER | IDENTIFIER COMMA identifier, we have solved that problem removing identifier non terminal. Another problem occurred in conditional statements part and we have solved that issue by adding and removing some elements from if statement, else_if statement, and else statement. On the other hand, the problem we spend the most time on was read by parser. Parser constantly gave grammatical errors to us and we have changed some features of our bnf to solve those issues.

How Precedence and Ambiguity Handled in Droneque

In our Yacc file, the precedence rules are applied to the according to the BNF structure. The paranthesis have the higher precedence followed by relational and logical operators. The relational and logical operators take precedence over the paranthesis. Multiplication and division operator have precedence over addition and subtraction operator. If the operators have the same precedence precedence will be given to the rightmost operation. To deal with the complexity of our language, we break down complex expressions into simple expression rules in order to resolve ambiguities. To eliminate ambiguities in our Droneque language, we looked at the order of the Yacc file. To resolve ambiguities presented by our language, we added flags to detect conflicts and followed through on the mistakes generated by the parser. Furthermore, the parser's errors helped us in improving our language.