Bilkent University

Department of Computer Engineering

CS315 Project 2

Icarus Programming Language

Date: 08.11.2021

Ata Seren 21901575 Section 1

Ayda Yurtoğlu 21903153 Section 3

Can Avşar 21902111 Section 1

Table of Contents

1. BNF Description of Icarus Language

1.1 Program Definition

<program> ::= <main>

<main> ::= <start_stmt><stmt_m><end_stmt>

<stmt_m> ::= <stmt_s> | <stmt_m><stmt_s>

<stmt_s> ::= <comment> | <expr><semicolon>

| <loop> | <method_dec> | <if_stmt>

<comment> ::= <inlineComment_sign><text>

| <comment_begin_sign><text><comment_end_sign>

<start_stmt> ::= <<<

<end_stmt> ::= >>>

1.2 Types

<type> ::= “bool” | “char” | “string” | “real”

<bool> ::= <true> | <false>

<drone_type> ::= “drone”<text>

<char> ::= <char_identifier><letter><char_identifier>

| <char_identifier><digit><char_identifier>

<string> ::= <string_identifier><text><string_identifier>

<text> ::= <letter> | <digit> | <symbol> | <text><letter>

| <text><symbol> | <text><digit> |

<int> ::= <digit> | <int><digit>

<real> ::= (<sign> |)<int>.<int>

1.3 Expressions

<expr> ::= <assignment_expr>

| <identifier><dot><builtIn_method>

| <method_dec_call> | <inout_expr>

<identifier> ::= <lowerCase_letter> <text>

<assignment_expr> ::= <var_declaration> | <general_expr>

| <real_expr> | <bool_expr> | <string_expr>

| <char_expr> | <input_expr> | <method_expr>

<var_declaration> ::= <type><identifier>

<general_expr> ::= <identifier><assignOperator><identifier>

| <identifier> <assignOperator>

<string_identifier><text> <string_identifier>

| <identifier><assignOperator>

<char_identifier><text><char_identifier>

| <identifier> <assignOperator><assignOperator>

| <identifier><assignOperator><real>

<real_expr> ::= <type><identifier><assignOperator><operation>

<bool_expr> ::= <type><identifier><assignOperator><bool>

<string_expr> ::= <type><identifier><assignOperator>

<string_identifier><text><string_identifier>

<char_expr> ::= <type><identifier><assignOperator><char>

<input_expr> ::= <identifier><assignOperator><input>

<method_expr> ::= <type><identifier><assignOperator>

<identifier><dot><builtIn_method> | <identifier>

<assignOperator><identifier><dot><builtIn_method>

1.4 Operators

<assignOperator> ::= =

<plusOperator> ::= +

<minusOperator> ::= -

<multOperator> ::= *

<divOperator> ::= /

<addition> ::= <plusOperator> | <minusOperator>

<multiplication> ::= <multOperator> | <divOperator>

<incDec> ::= <identifier><incrementOperator>

| <identifier><decrementOperator>

<incrementOperator> ::= ++

<decrementOperator> ::= --

<logicalOperator> ::= <and> | <or> | <not> | <equalCheck>

| <notEqualCheck> | <less> | <greater>

| <lessEqual> | <greaterEqual>

<and> ::= &&

<or> ::= ||

<not> ::= !

<less> ::= <

<greater> ::= >

<lessEqual> ::= <=

<greaterEqual> ::= >=

<equalCheck> ::= ==

<notEqualCheck> ::= !=

<logicalParam> ::= <method_dec_call> | <expr> | <bool>

| <not><method_dec_call> | <not><expr>

| <not><bool>

<logicalExpr> ::= <logicalParam><logicalOperator><logicalParam>

<operation> ::=<operation> <addition> <term> | <term>

<term> ::= <term> <multOperator> <factor> |<factor>

<factor> ::= <operation> | <identifier> | <real>

1.5 Symbols

<lowerCase_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'

<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

<symbol> ::= <LP> | <RP> | <LSB> | <RSB> | <LB> | <RB> | <dot>
| <comma>| <semicolon> | <underscore> | <equal>
| <space> | <char_identifier> | <string_identifier>
| <hashtag> | <sign> | <endline>

<LP> ::= (

<RP> ::=)

<LSB> ::= [

<RSB> ::=]

<LB> ::= {

<RB> :: = }

<dot> ::= .

<comma> ::= ,

<semicolon> ::= ;

<equal> ::= =

<underscore> ::= _

<space> ::= “ ”

<char_identifier> ::= ‘

<string_identifier> ::= “

<hashtag> ::= #

<sign> ::= + | -

<endline> ::= \n

<inlineComment_sign> ::= <>

<comment_begin_sign> ::= <**>

<comment_end_sign> ::= <*>

1.6 Functions

<parameter> ::= <type><identifier> | <type><identifier>,<parameter>
| <identifier>,<parameter> |

<move_type> ::= “up” | “down” | “left” | “right” | “front” | “back” | “stop”

<return_type> ::= void | <type>

<builtIn_method> ::= <method_heading> | <method_altitude>
| <method_temp> | <method_move> | <method_heading>
| <method_nozzle> | <method_connect>
| <method_disconnect>

<method_heading> ::= <text>.readHeading[ ]

<method_altitude> ::= <text>.readAltitude[ ]

<method_temp> ::= <text>.readTemperature[ ]

<method_move> ::= <text>.setMove[<move_type> , <real>]
| <text>.setMove[<move_type>]

<method_heading> ::= <text>.setHeading[<real>]

<method_nozzle> ::= <text>.setNozzle[<bool>]

<method_connect> ::= <text>.connectWifi[ ]

<method_disconnect> ::= <text>.disconnectWifi[ ]

<method_dec> ::= <returnType><text>[<parameter>] { (<stmt_m> |) }

<method_dec_call> ::= <text>[<parameter>] | <identifier>.<text>[<parameter>]

1.7 Constants

<const> ::= <constIdentifier><identifier>

<constIdentifier> ::= const

1.8 Conditionals

<true> ::= “true” | 1

<false> ::= “false” | 0

<if_stmt> ::= <matched> | <unmatched>

<matched> ::= if [<logicalExpr>] {<matched>} else {<matched>}

| if [<logicalExpr>] {<matched>} <elseif_stmt>

<unmatched> ::= if [<logicalExpr>] {<stmt>}

| if [<logicalExpr>] {<matched>} else {<unmatched>}

| if [<logicalExpr>] {<matched>} <elseif_stmt>

<elseif_stmt> ::= else if [<logicalExpr>] {<stmt>}

| else if [<logicalExpr>] {<matched>} <elseif_stmt>

| else if [<logicalExpr>] {<matched>} else {<unmatched>}

1.9 Loops

<while_stmt> ::= while [<logicalExpr>] {<stmt_m>}

<for_stmt> ::= for [<initialization> ; <logicalExpr>; <incDec>] {<stmt_m>}

<initialization> ::= <type><identifier> | <identifier> |

<doWhile_stmt> ::= do {<stmts>} while [<logicalExpr>]

1.10 Input/Output

<inout_param> ::= <var> | <text> | <string_identifier><text><string_identifier>

| <char_identifier><text><char_identifier>

<inout> ::= <input> | <output>

<output> ::= icarusout[<inout_param>]

<input> ::= <assignOperator>icarusin[ ]

2. Explanation of Icarus Language Constructions

2.1 Program Definition

<program> **::= <main>**

A valid Icarus program uses components of <main>. This rule gives a start to the program.

<main> **::= <start_stmt><stmt_m><end_stmt>**

This variable includes statements between double curly brackets.

<stmt_m> **::= <stmt_s> | <stmt_m><stmt_s>**

This variable is used to hold multiple statements when needed.

<stmt_s> **::= <comment> | <expr><semicolon>**

| <loop> | <method_dec> | <if_stmt>

This statement variable is used to determine the type of statement such as expressions, input-output statements, comment lines, loops, method declarations, calling built-in or custom methods and conditional statements.

<comment> **::= <inlineComment_sign><text>**

| <comment_begin_sign><text><comment_end_sign>

Comment is one of the components of <stmt_s> and is used to create single line or multiple line comments to make any explanation about the code anywhere on the code. For single line, “<>” symbol is used and for start and end of multiple line comments, “<**>” and “<*>” are used, respectively.

<start_stmt> **::= <<<**

<end_stmt> **::= >>>**

These statements indicate the start and end point of the program. The execution starts after “<<<” and ends with “>>>”.

2.2 Types

<type> **::= “bool” | “char” | “string” | “real”**

Icarus Language has six types. There are *bool* for boolean values, *char* for characters, *string* for texts, *real* for all numbers including integers and floating point numbers. All of these types start with lowercase letters, which increases the writability of Icarus language.

<bool> **::= <true> | <false>**

Bool represents truth values. The language has two truth values: true or false.

<drone_type> **::= “drone”<text>**

This variable will be used to define drones in the program and built-in functions can be used on this drone variable.

<char> **::= <char_identifier><letter><char_identifier>**

| <char_identifier><digit><char_identifier>

This defines characters. A character is either a *letter* or a *digit*.

<string> **::= <string_identifier><text><string_identifier>**

<text> **::= <letter> | <digit> | <symbol> | <text><letter>**

| <text><symbol> | <text><digit> |

These three terminals used to represent the texts. A *string* is a text between string identifiers (“). A *text* consists of words. The *letters* or *digits* can be used to build a word.

<int> **::= <digit> | <int><digit>**

<real> **::= (<sign> |)<int>.<int>**

Icarus language uses “real” type for all numbers, which includes integers and floating point numbers. This eases the writability and readability of the language. This type can be signed or unsigned depending on its use in the program. A signed real number states whether the number is positive or negative. An unsigned real number is assumed to be positive.

2.3 Expressions

<expr> **::= <assignment_expr> | <identifier><dot><builtIn_method>**

| <method_dec_call> | <inout_expr>

An <expr> can be either an expression for assignments, built-in methods, method declarations or icarusin[ ] and icarus [ ] statemenst.

<identifier> **::= <lowerCase_letter> <text>**

Identifier means variables declared by the programmer. For example “d” in “Drone d;” is an identifier. The convention to declare variables in Icarus is to always start with a lowercase letter.

<assignment_expr> **::= <var_declaration> | <general_expr>**

| <real_expr> | <bool_expr> | <string_expr>

| <char_expr> | <input_expr> | <method_expr>

Assignment expressions are used to assign a value to an identifier. The assigned values can be of type real, bool, string and char. The programmer can also declare a variable such as “drone d;” and call methods.

<var_declaration> **::= <type><identifier>**

A variable is declared by first stating its type and then its name.

<general_expr> **::= <identifier><assignOperator><identifier>**

| <identifier> <assignOperator>

<string_identifier><text> <string_identifier>

| <identifier><assignOperator>

<char_identifier><text><char_identifier>

| <identifier> <assignOperator><assignOperator>

| <identifier><assignOperator><real>

The general expression means all the possible assignment operations for all types (real, bool, string, char) which are not declared at the same time. For example, a statement such as “a = 3.2” is a general expression, but “real a = 3.2” is not. We classify the second example where the type of the identifier is specified as a “real expression”.

<real_expr> **::= <type><identifier><assignOperator><operation>**

<bool_expr> **::= <type><identifier><assignOperator><bool>**

<string_expr> **::= <type><identifier><assignOperator>**

<string_identifier><text><string_identifier>

<char_expr> **::= <type><identifier><assignOperator><char>**

The above four expressions are used when a variable is both declared and assigned a value. This increases the readability and writability of the language by adding flexibility when declaring variables. So both statements are valid: string str = “text”; and string str; str = “text”;

<input_expr> **::= <identifier><assignOperator><input>**

This is used with input and output expressions.

<method_expr> **::= <type><identifier><assignOperator>**

<identifier><dot><builtIn_method> | <identifier>

<assignOperator><identifier><dot><builtIn_method>

Method declarations include two types: one where the type of identifier is specified and one where the type is not specified, because the identifier has been declared before.

2.4 Operators

<assignOperator> **::= =**

The assignment operator in Icarus is “=”. It is a special type of operator because it assigns the value on the right hand side of the equation to the left hand side, instead of stating an arithmetic or logical expression.

<plusOperator> **::= +**

<minusOperator> **::= -**

<multOperator> **::= ***

<divOperator> **::= /**

<addition> **::= <plusOperator> | <minusOperator>**

<multiplication> **::= <multOperator> | <divOperator>**

These variables are used for addition, subtraction, multiplication and division operations. They are under 2 different variables to prevent ambiguity of operations defined below.

<incDec> **::= <identifier><incrementOperator>**

| <identifier><decrementOperator>

These variables are used to perform increment and decrement which can be seen in other languages in “<var>++” and “<var>--” forms.

<incrementOperator> ::= ++

<decrementOperator> ::= --

The above operators provide arithmetic operations to Icarus. The programmer can sum, subtract, multiply, divide two int or float types. Exponentiation is also possible by using “^”. We also included increment and decrement operators to simplify adding and subtracting 1 to a variable.

<logicalOperator> ::= <and> | <or> | <not> | <equalCheck>

| <notEqualCheck> | <less> | <greater>

| <lessEqual> | <greaterEqual>

<and> ::= &&

<or> ::= ||

<not> ::= !

<less> ::= <

<greater> ::= >

<lessEqual> ::= <=

<greaterEqual> ::= >=

<equalCheck> ::= ==

<notEqualCheck> ::= !=

The logical operators provide logical operations. When they are used, they return a truth value. This value can be inverted with the not operator. The basic logical functions “and” and “or” are also provided. The programmer can also make a logical expression using less than and greater than symbols. Additionally, two expressions can be checked to see whether they are logically equal or not with “==” and “!=” signs.

<logicalParam> ::= <method_dec_call> | <expr> | <bool>

| <not><method_dec_call> | <not><expr>

| <not><bool>

<logicalExpr> ::= <logicalParam><logicalOperator><logicalParam>

The logical expression non-terminal is used to declare conditions that the conditionals check to see if they will execute a certain statement. A logical expression returns a truth value by using various combinations of expressions. These expressions can include functions that return a truth value and can be created with the use of logical operators.

<operation> **::=<operation> <addition> <term> | <term>**

<term> **::= <term> <multOperator> <factor> |<factor>**

<factor> **::= <operation> | <identifier> | <real>**

Operation variable is used to make any calculation and use this calculation in variable assignments. To support the operation variable, two other variables, term and factor are created. Operation variable provides addition and subtraction, term variable provides multiplication and division and factor variable provides elements that can be used in calculations and opportunity to write these operations in parentheses. In this way, ambiguity is prevented on calculations.

2.5 Symbols

<lowerCase_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'

<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'

The letter terminal consists of 26 lowercase and 26 uppercase English letters.

<digit> **::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

Digit terminal can be used to determine digits, 0-9.

<symbol> **::= <LP> | <RP> | <LSB> | <RSB> | <LB> | <RB>**

| <dot> | <comma>| <semicolon> | <underscore>

| <equal> | <space> | <char_identifier>

| <string_identifier> | <hashtag> | <sign> | <endline>

Symbols can be left parenthesis, right parenthesis, left square brackets, right square brackets, left braces, right braces, dot, comma, semicolon, underscore, equal sign, space, single quote, double quotes, hashtag, plus/minus sign, and an endline symbol \n, accordingly.

<inlineComment_sign> ::= <>

<comment_begin_sign> ::= <>**

<comment_end_sign> ::= <*>

A single line comment is denoted by <> symbol. The multi line comments begin with <**> and end with <*>.

2.6 Functions

<parameter> ::= <type><identifier>

| <type><identifier>,<parameter>

| <identifier>,<parameter> |

Parameters are used in built-in functions and custom-declared functions. Components of it such as a variable declaration and multiple variable declarations separated by commas can be used in function declarations. Also, one of its components of declared variables can be used in function calls.

<move_type> ::= “up” | “down” | “left” | “right” | “front”

| “back” | “stop”

These predefined keywords are used to define the movement of the drone. In addition to the direction keywords, there is also a “stop” keyword to indicate the end of the movement. All these reserved words start with lowercase letters and this enhances the writability of the language.

<return_type> ::= void | <type>

This variable is used to determine if a method returns a value or not.

<builtIn_method> ::= <method_heading> | <method_altitude>

| <method_temp> | <method_move>

| <method_heading>

| <method_nozzle> | <method_connect>

| <method_disconnect>

Built-in function variable is a component of statements and its components are built-in functions which can be called for a specific object.

<method_heading> ::= <text>.readHeading[ ]

<method_altitude> ::= <text>.readAltitude[ ]

<method_temp> ::= <text>.readTemperature[ ]

<method_move> ::= <text>.setMove[<move_type> , <real>]

| <text>.setMove[<move_type>]

<method_heading> ::= <text>.setHeading[<real>]

<method_nozzle> ::= <text>.setNozzle[<bool>]

<method_connect> ::= <text>.connectWifi[ ]

<method_disconnect> ::= <text>.disconnectWifi[ ]

These are built-in functions that can be used on a specific object to retrieve, change or insert the data of an object. For example, retrieving altitude and direction of a drone, temperature of the environment, commanding a drone to turn to a desired heading or move in a desired way of movement directions, turning on or off of the nozzle of the drone or connecting to and disconnecting from a Wi-Fi can be performed by these built-in functions. The convention when calling these functions is to first state which drone will be used for the function, use a dot and then declare the function with its appropriate parameter. Using a dot increases the writability of the language.

<method_dec> ::= <returnType><text>[<parameter>] { (<stmt_m> |) }

<method_dec_call> ::= <text>[<parameter>]

| <identifier>.<text>[<parameter>]

These variables are used to allow users to declare and call custom methods. The custom methods can include parameters if desired. The convention when declaring a method is to start with a lowercase letter, which enhances the writability of the language.

2.7 Constants

<const> ::= <constIdentifier><identifier>

<constIdentifier> ::= const

The user can declare a constant by using the “const” identifier. The constants cannot be changed in the program once the programmer declares its value. The convention for constant declaration is to first write the “const” identifier, then state the type of the constant and give a name to it.

2.8 Conditionals

<true> **::= “true” | 1**

<false> **::= “false” | 0**

There are two boolean values in Icarus language: true and false, which can also be indicated with 1 and 0.

<if_stmt> **::= <matched> | <unmatched>**

<matched> **::= if [<logicalExpr>] {<matched>} else {<matched>}**

| if [<logicalExpr>] {<matched>} <elseif_stmt>

<unmatched> **::= if [<logicalExpr>] {<stmt>}**

| if [<logicalExpr>] {<matched>} else {<unmatched>}

| if [<logicalExpr>] {<matched>} <elseif_stmt>

<elseif_stmt> **::= else if [<logicalExpr>] {<stmt>}**

| else if [<logicalExpr>] {<matched>} <elseif_stmt>

| else if [<logicalExpr>] {<matched>} else {<unmatched>}

Conditionals are used to define if and else statements. If a “true” logical expression is seen in a conditional, the following statement will be executed. Else, another statement will be executed. The non-terminals <matched> and <unmatched> are used to avoid ambiguity. The convention for if-else statements is to first state the condition and then state what will happen if this condition is met.

Else if variable is used to allow users to write better conditional statements. Since we can consider it as an if statement that will be considered after the previous if statement, its grammar is written with <matched> and <unmatched> non-terminals, which are also used in if-else statements.

2.9 Loops

<while_stmt> **::= while [<logicalExpr>] {<stmt_m>}**

While loops start with a while keyword and a logical expression. They repeat by the condition of the logical expression. After that there are statements to execute. The convention for a while loop is to state the boundary with a logical expression for a variable.

<for_stmt> **::= for [<initialization> ; <logicalExpr>; <incDec>]**

{<stmt_m>}

<initialization> **::= <type><identifier> | <identifier> |**

For loops denote an initialization of a counting variable. The loop also consists of logical expressions for the boundary and incrementation/decrementation. The convention for a for loop is to initialize the variable, state the boundary and then incrementation or decrementation, all separated with a semicolon.

<doWhile_stmt> **::= do {<stmts>} while [<logicalExpr>]**

Do-while loops starts with *do* keyword, ends with *while* keyword and a logical expression that determines repetition. Statements go between *do* and *while* keywords. The convention for a do-while loop is to state the boundary with a logical expression for a variable after the while keyword.

2.10 Input/Output

<inout_param> **::= <var> | <text>**

| <string_identifier><text><string_identifier>

| <char_identifier><text><char_identifier>

This variable is used to put any input/output parameter to input/output functions.

<inout> **::= <input> | <output>**

This variable is used to determine the operation whether it's input or output.

<output> **::= icarusout[<inout_param>]**

<input> **::= <assignOperator>icarusin[ ]**

These variables declare input and output functions as "icarusin[ ]" and "icarusout[ ]", respectively.

3. Non-Trivial Tokens of Icarus Language

start_stmt	Token to start the program.
end_stmt	Token to end program.
void	Token reserved for methods that don't return anything.
real_type	Token reserved for real number type.
string_type	Token reserved for string type.
char_type	Token reserved for character type.
bool_type	Token reserved for boolean type.
const_identifier	Token reserved for the “const” identifier.
drone_type	Token reserved for drone type.
type	Token reserved for all types.
if_stmt	Token reserved for “if” statements.
else_stmt	Token reserved for “else” statements.
elseif_stmt	Token reserved for “else if” statements.
for_stmt	Token reserved for “for loop” statements.
while_stmt	Token reserved for “while loop” statements.
do_stmt	Token reserved for “do loop” statements.
return_stmt	Token reserved for all return types.
icarusin	Token reserved for inputs.
icarusout	Token reserved for outputs.
true	Token for “true” boolean value.
false	Token for “true” boolean value.
line_comment_sym	Token reserved for inline comment symbols.
line_comment	Token reserved for inline comments.
doc_comment_start	Token reserved for multiple line comment start symbol.
doc_comment_end	Token reserved for multiple line comment end symbol.
doc_comment	Token reserved for multiple line comments.

not	Token reserved for "!" symbol.
lp	Token reserved for "(" symbol.
rp	Token reserved for ")" symbol.
lb	Token reserved for "{" symbol.
rb	Token reserved for "}" symbol.
lsb	Token reserved for "[" symbol.
rsb	Token reserved for "]" symbol.
dot	Token reserved for "." symbol.
comma	Token reserved for "," symbol.
semicolon	Token reserved for ";" symbol.
assign_op	Token reserved for "=" symbol.
equal_check	Token reserved for "==" symbol.
not_equal_check	Token reserved for "!=" symbol.
string_identifier	Token reserved for " “ " symbol.
char_identifier	Token reserved for " ‘ " symbol.
plus	Token reserved for "+" symbol.
minus	Token reserved for "-" symbol.
multiply	Token reserved for "*" symbol.
divide	Token reserved for "/" symbol.
exp	Token reserved for "^" symbol.
gt	Token reserved for ">" symbol.
lt	Token reserved for "<" symbol.
lte	Token reserved for "<=" symbol.
gte	Token reserved for ">=" symbol.
and	Token reserved for "&&" symbol.
or	Token reserved for "||" symbol.
newline	Token reserved for "\n" symbol.

underscore	Token reserved for "_" symbol.
read_heading_fcn	Token reserved for function to read heading.
read_altitude_fcn	Token reserved for function to read altitude.
read_temp_fcn	Token reserved for function to read temperature.
movement	Token reserved for specifying the movement properties of the drone.
set_move_fcn	Token reserved for function to move the drone in various directions.
set_heading_fcn	Token reserved for function to set heading of the drone.
set_nozzle_fcn	Token reserved for function to turn on or off the nozzle of the drone.
connect_wifi_fcn	Token reserved for function to connect to WiFi.
disconnect_wifi_fcn	Token reserved for function to connect to WiFi.
parameter	Token reserved for parameters used in methods.
digit	Token reserved for digits.
capital_letter	Token reserved for uppercase letters.
lower_letter	Token reserved for uppercase letters.
letter	Token reserved for a letter.
text	Token reserved for a text.
identifier	Token reserved for identifiers of variables.
string_stmt	Token reserved for string statements.
bool_stmt	Token reserved for boolean statements.
char_stmt	Token reserved for character statements.
real	Token reserved for all real numbers.
digit	Token reserved for a digit.
increment_stmt	Token reserved for incrementation and “++” symbols.
decrement_stmt	Token reserved for decrementation and “--” symbols.

4. Resolution of Conflicts

When we wrote all rules and tokens in our yacc and parsed it on Dijkstra machine, we got more than 200 shift/reduce and reduce/reduce conflicts. When we investigated the yacc file, we saw that some of the rules don't end with tokens. Instead, they shift to a rule that hasn’t been defined or some irrelevant rule. Therefore, as the first step of resolution of conflicts, we ensured that all rules can reach to and end at tokens. After completing this step, our conflicts decreased to 11 shift/reduce conflicts.

Since there are less conflicts now, we need to be more precise about solving these conflicts. Therefore, as the second step of resolution of conflicts, we inspected the y.output file on a text editor after downloading it from FileZilla. When we inspected the conflicts, we saw that there are 3 sources of conflicts: calculations, types and token rule definitions with the same functionality. To fix conflicts in calculations, we defined a new token in the .lex file for all types and used it in the rules of calculations. In this way, we fixed both conflicts caused by types and grammar used against ambiguity in calculation rules. This new token is also the key for fixing the same functionality rule definitions. Instead of using different type tokens, we used a single token to detect all types used in the test program and decreased the number of rules for the same functionality.

In general, we ensured that all rules contain tokens or relevant rules, merged rules and rule definitions for the same functionality and defined new tokens to prevent shift/reduce conflicts that happened on rules of type definitions to solve the all conflicts.