Bilkent University

Department of Computer Engineering

CS315 Project 1

Icarus Programming Language

Date: 22.10.2021

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> ::= (<var> | <type><var>)<assignOperator><expr><stmt_endop> |
<var><incdecOperator><stmt_endop> | <inout><stmt_endop>| <comment> | <loop> |
<method_builtIn><stmt_endop> | <method_dec> | <method_dec_call><stmt_endop> |
<if_stmt>

<comment> ::= <inlineComment_sign><text> |
<comment_begin_sign><text><comment_end_sign>

<start_stmt> ::= <<<

<end_stmt> ::= >>>

1.2 Types

<type> ::= “bool” | “char” | “string” | “float“ | “int”

<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> ::= <intUnsigned> | <intSigned>

<intUnsigned> ::= <digit> | <intUnsigned><digit>

<intSigned> ::= <sign><intUnsigned>

<float> ::= <floatUnsigned> | <floatSigned>

<floatUnsigned> :: <intUnsigned>.<intUnsigned>

<floatSigned> ::= <intSigned>.<intUnsigned>

<var> ::= <text> | <var> <digit> | <var><text>

1.3 Operators

<assignOperator> ::= =

<operator> ::= <plusOperator> | <minusOperator> | <multOperator> | <divOperator> | <exponentOperator>

<plusOperator> ::= +

<minusOperator> ::= -

<multOperator> ::= *

<divOperator> ::= /

<exponentOperator> ::= ^

<incdecOperator> ::= <incrementOperator> | <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>

<expr_assign> ::= <var> <equal> <expr>

<expr> ::= <expr> <plusOperator> <term> |<expr> <minusOperator> <term> | <term>

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

<factor> ::= <LP><expr><RP> | <var> | <const>

<increment> ::= <var><incrementOperator>

<decrement> ::= <var><decrementOperator>

<exponent> ::= <var><exponentOperator><var> | <var><exponentOperator><const> | <const><exponentOperator><var> | <const><exponentOperator><const>

1.4 Symbols

<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> ::= <*>

<block_begin> ::= <<

<block_end> ::= >>

1.5 Functions

<parameter> ::= <type><var> | <type><var><comma><parameter> |
<var><comma><parameter> |

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

<return_type> ::= void | <type>

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

<method_direction> ::= <text><dot>direction<LSB><RSB>

<method_altitude> ::= <text><dot>altitude<LSB><RSB>

<method_temp> ::= <text><dot>temperature<LSB><RSB>

<method_move> ::= <text><dot>move<LSB><move_type><comma><intUnsigned><RSB>
| <text><dot>move<LSB><move_type><RSB>

<method_heading> ::= <text><dot>setHeading<LSB><intUnsigned><RSB>

<method_nozzle> ::= <text><dot>nozzleActive<LSB><bool><RSB>

<method_connect>::= <text><dot>connectWifi<LSB><RSB>

<method_disconnect>::= <text><dot>disconnectWifi<LSB><RSB>

<method_dec> ::= <returnType><text><LSB><parameter><RSB><block_begin>(<stmt_m>
|) <block_end>

<method_dec_call> ::= <text><LSB><parameter><RSB> |
<var><dot><text><LSB><parameter><RSB>

1.6 Constants

<const> ::= <constIdentifier><var>

<constIdentifier> : const

1.7 Conditionals

<true> ::= “true” | 1

<false> ::= “false” | 0

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

<matched> ::= if <LSB> <logicalExpr> <RSB> <block_begin><matched><block_end> else
<block_begin><matched><block_end>
<unmatched> ::= if <LSB> <logicalExpr><RSB> <block_begin><stmt><block_end> | if
<LSB><logicalExpr><RSB> <block_begin><matched><block_end> else
<block_begin><unmatched><block_end>

1.8 Loops

<initialization> ::= <type><expr> | <expr> |

<while_stmt> ::= while <LSB> <logicalExpr> <RSB> <block_begin> <stmt_m> <block_end>

<for_stmt> ::= for <LSB> <initialization> ; <logicalExpr> <RSB> <block_begin> <stmt_m> <block_end>

<doWhile_stmt> ::= do <block_begin> <stmts> <block_end> while <LSB><logicalExpr> <RSB>

1.9 Input/Output

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

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

<output> ::= icarusout<LSB><inout_param><RSB>

<input> ::= <var><assignOperator>icarusin<LSB><RSB>

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> ::= (<var> | <type><var>)<assignOperator><expr><stmt_endop> | <var><incdecOperator><stmt_endop> | <inout><stmt_endop>| <comment> | <loop> | <method_builtIn><stmt_endop> | <method_dec> | <method_dec_call><stmt_endop> | <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” | “float“ | “int”**

Icarus Language has six types. There are *bool* for boolean values, *char* for characters, *string* for texts, *float* for decimals and int for numbers. All of these types start with lowercase letters, which increases the writability of Icarus language.

- **<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.

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

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

- **<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> ::= <word> | <text><word> | <text><symbol>|<text><digit>**
- **<word> ::= <letter> | <word><letter> |<word><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> ::= <intUnsigned> | <intSigned>**
- **<intUnsigned> ::= <digit> | <int><digit>**
- **<intSigned> ::= <sign><intUnsigned>**

There is a general integer definition that consists of unsigned and signed integers. An unsigned integer is either a digit or a number. A signed digit is basically an unsigned integer that has a sign in front of it, which states whether the number is positive or negative.

- **<float> ::= <floatUnsigned> | <floatSigned>**
- **<floatUnsigned> :: <intUnsigned>.<intUnsigned>**
- **<floatSigned> ::= <intSigned>.<intUnsigned>**

There is a general float definition that consists of unsigned and signed floats. An unsigned float consists of an unsigned integer, a dot and another unsigned integer. A signed float has a sign in front of the first integer.

- **<var> ::= <text> | <var> <digit> | <var><text>**

<var> is used to define a variable that consists of a combination of text and numbers. This variable always starts with a lowercase letter and it can never start with a number or a symbol that is defined in the symbols section. This convention increases readability.

2.3 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.

- **<operator> ::= <plusOperator> | <minusOperator> | <multOperator> | <divOperator> | <exponentOperator>**
- **<plusOperator> ::= +**
- **<minusOperator> ::= -**
- **<multOperator> ::= ***
- **<divOperator> ::= /**
- **<exponentOperator> ::= ^**
- **<incdecOperator> ::= <incrementOperator> | <decrementOperator>**
- **<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.

- **<expr_assign> ::= <var> <equal> <expr>**
- **<expr> ::= <expr> <plusOperator> <term> |<expr> <minusOperator> <term> | <term>**
- **<term> ::= <term> <multOperator> <factor> | <term> <divOperator> <factor> | <factor>**
- **<factor> → <LP><expr><RP> | <var> | <const>**

These variables are used to define expressions, assignments and calculations in a precedence. Multiplication and division are prioritized against addition and subtraction.

- **<increment> ::= <var><incrementOperator>**
- **<decrement> ::= <var><decrementOperator>**

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

- **<exponent> ::= <var><exponentOperator><var> | <var><exponentOperator><const> | <const><exponentOperator><var> | <const><exponentOperator><const>**

This variable is used to take the exponent of a constant or a variable with a constant or a variable.

2.4 Symbols

- **<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 <*>.

- **<block_begin> ::= <<**
- **<block_end> ::= >>**

These symbols indicate the start and end of a method. They can be also seen in if-else statements, loops, built-in functions and custom functions.

2.5 Functions

- **<parameter> ::= <type><var> | <type><var><comma><parameter> | <var><comma><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.

- **<method_builtIn> ::= <method_direction> | <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_direction> ::= <text><dot>direction<LSB><RSB>**
- **<method_altitude> ::= <text><dot>altitude<LSB><RSB>**
- **<method_temp> ::= <text><dot>temperature<LSB><RSB>**
- **<method_move> ::= <text><dot>move<LSB><move_type><comma><intUnsigned><RSB> | <text><dot>move<LSB><move_type><RSB>**
- **<method_heading> ::= <text><dot>setHeading<LSB><intUnsigned><RSB>**
- **<method_nozzle> ::= <text><dot>nozzleActive<LSB><bool><RSB>**
- **<method_connect>::= <text><dot>connectWifi<LSB><RSB>**
- **<method_disconnect>::= <text><dot>disconnectWifi<LSB><RSB>**

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><LSB><parameter><RSB><block_begin>(<stmt_m> |) <block_end>**
- **<method_dec_call> ::= <text><LSB><parameter><RSB> | <var><dot><text><LSB><parameter><RSB>**

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.

2.6 Constants

- **<const> ::= <constIdentifier><var>**
- **<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.

2.7 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 <LSB> <logicalExpr> <RSB>**
  <block_begin><matched><block_end> else
  <block_begin><matched><block_end>
- **<unmatched> ::= if <LSB> <logicalExpr><RSB>**
  <block_begin><stmt><block_end> | if <LSB><logicalExpr><RSB>
  <block_begin><matched><block_end> else
  <block_begin><unmatched><block_end>

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.

2.8 Loops

- **<for_stmt> ::= for <LSB> <initialization> ; <logicalExpr> <RSB>**
  <block_begin> <stmt_m> <block_end>
- **<initialization> ::= <type><expr> | <expr> |**

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.

- **<while_stmt> ::= while <LSB> <logicalExpr> <RSB> <block_begin> <stmt_m>**
  <block_end>

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.

- **<doWhile_stmt> ::= do <block_begin> <stmts> <block_end> while <LSB><logicalExpr> <RSB>**

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.

2.9 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<LSB><inout_param><RSB>**
- **<input> ::= <var><assignmentOperator>icarusin<LSB><RSB>**

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

3. Non-Trivial Tokens of Icarus Language

start_stmt: Token to start program.

end_stmt: Token to end program.

return_type: Token reserved for all return types.

int_stmt: Token reserved for integer type.

string_stmt: Token reserved for string type.

char_stmt: Token reserved for character type.

bool_stmt: Token reserved for boolean type.

float_stmt: Token reserved for float type.

int_unsigned: Token reserved for unsigned integers.

int_signed: Token reserved for signed integers.

float_unsigned: Token reserved for unsigned floating point numbers.

float_signed: Token reserved for signed floating point numbers.

digit: Token reserved for a digit.

letter: Token reserved for a letter.

text: Token reserved for a text.

drone: Token reserved for drone type.

const: Token reserved for the “const” identifier.

if: Token reserved for “if” statements.

else: Token reserved for “else” statements.

for: Token reserved for “for loop” statements.

while: Token reserved for “while loop” statements.

do: Token reserved for “do loop” statements.

return: Token to return values.

icarusin: Token reserved for inputs.

icarusout: Token reserved for outputs.

line_comment: Token reserved for single line comments.

comment: Token reserved for multiple lines of comments.

heading_fcn: Token reserved for function to read heading.

altitude_fcn: Token reserved for function to read altitude.

temp_fcn: Token reserved for function to read temperature.

movement: Token reserved for specifying the movement properties of the drone.

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.

block_start: Token reserved for the beginning of a method declaration.

block_end: Token reserved for the ending of a method declaration.

4. Example Programs

4.1 Example 1

```
<<<
<**> This function activates the nozzle until the drone reaches a certain temperature.
When the drone reaches the maximum temperature, it moves down and stops. <*>
drone d;
const int maxTemperature = 80;
for [ int i = 0; i < 100; i++] <<
        if [ d.temperature < maxTemperature ] <<
                d.nozzleActive[true];
        >>
        else <<
                d.nozzleActive[false];
                while [ d.altitude > 0 ] <<
                        d.move[down];
                        i = 100;
                >>
        >>
        d.move[stop];
>>
>>>
```

4.2 Example 2

```
<<<
<**>State user,
define a drone,
give commands to it,
say goodbye!<*>
icarusout[“Hello Drone!”];
string user;
user = icarusin[];
drone d;
d.connectWifi[];
int direction = d.direction[];
d.setHeading[d + 90];
d.move[front, 5];
d.move[right, 5];
int temp = d.temperature[]
d.disconnectWifi[checkTemp[temp, 35]];
icarusout[“Goodbye!”];

bool checkTemp[int drn, int temp] <<
        if[drn == temp]<<
                return true;
        >>
        else << return false; >>
>>
>>>
```

4.3 Example 3

```
<<<
drone d;
const int riskAltitude = 1;

<> Drone should connect to WiFi after being initialized.
d.connectWifi[];

<**> This part prevents falling by accident,
the drone should rise until it goes up to 5 meters <*>
if [d.altitude <= riskAltitude] <<
        while[d.altitude < 5] <<
                d. move[up];
        >>
>>

<> This part starts the spraying by going down.
if[d.temperature < 30 && d.temperature > 10) <<
        while[d.altitude > 5] <<
                d.move[down];
        >>
        d.nozzleActive[true];
        d.setHeading[0];
>>
<> Drone disconnects from WiFi.
d.disconnectWifi[];
>>>
```

5. Evaluation of Icarus Language

5.1 Readability

Simplicity is the key feature of Icarus language. It is dedicated to a field so that it can be easily readable and understandable. With similarity to mainly used programming languages, Icarus is way simpler than them. Icarus has specific type names and keywords, so it aims to minimize the confusion on the side of the reader. The syntax consists of many common items, and words in the locations can be easily found. Syntax and relationships between words make reading seamless for the code. Some operators and functions are added to make complicated processes simpler. With these new properties and developed syntax, Icarus language is easy to read and intuitive to understand.

5.2 Writability

Features of Icarus language are easy to write and understand, because it doesn't have complicated syntax. The simple definitions and direct functionalities built in Icarus language increase its writability. Icarus language doesn't offer too many options to do the same operation. However, it doesn't make the user write a difficult expression. For example, <expr> variable under program definition section can be used for both variables or constants and can be conducted with any defined operator. In addition to that, we added <incrementOperator> and <decrementOperator> to make an increment with "++" or decrement with "--" operator. Since the main purpose of the language is controlling the drones, its functions have simple but effective built-in functionalities. In this way, users can perform any operation by writing code efficiently and without confusion.

5.3 Reliability

The grammar of Icarus language only uses left recursion, which reduces the risk of encountering ambiguity during implementation. There are also cases where the grammar uses additional non-terminals to avoid ambiguity. An example can be seen in the conditionals section where <matched> and <unmatched> resolves the ambiguous if-else statements. Although Icarus language is reliable regarding ambiguity, it doesn't have an interpreter or a compiler. This challenges its reliability in terms of type checking and handling errors. It also doesn't provide exception handling functionalities. Therefore, these issues can impact the reliability of Icarus language.