2021-2022 FALL SEMESTER

CS 315

PROJECT 1 - HUDSON

INSTRUCTOR: H. ALTAY GÜVENİR

TEAM MEMBERS:

1) METEHAN SAÇAKÇI, 21802788, SECTION: 01
2) ARDA İÇÖZ, 21901443, SECTION: 01
3) NURETTİN ONUR VURAL, 21902330, SECTION: 01

CONTENTS

BNF of HUDSON

1) Program Definition

<initialState> ::= <activate>

<activate> ::= ACTIVATE_DRONE <statements> STOP_DRONE

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

<statement> ::= <comment> | <expression> | <loops> | <if_else_statement>

<comment> ::= COMMENT <content> END_COMMENT

<content> ::= STRING <content> | STRING | <empty>

<expression> ::= <declarations> | <assignments> | <input> | <output>

<loops> ::= <for_loop> | <while_loop>

<declarations> ::= <declare_function>
| <call_function> | <call_primitive_function>
| <declare_integer> | <declare_constant_integer>
| <declare_float> | <declare_constant_float>
| <declare_string> | <declare_constant_string>
| <declare_boolean>

<assignments> ::= <assign_integer>
| <assign_float>
| <assign_string>
| <assign_boolean>

2) Declarations

<declare_integer> ::= INT identifier
| INT <assign_integer>

<declare_constant_integer> ::= CONSTANT_INT <assign_constant_integer>

<declare_float> ::= FLOAT identifier

```
                    | FLOAT <assign_float>
```

<declare_constant_float> ::= CONSTANT_FLOAT <assign_constant_float>

<declare_string> ::= STRING identifier

```
                      | STRING <assign_string>
```

<declare_constant_string> ::= CONSTANT_STRING <assign_constant_string>

<declare_boolean> ::= BOOLEAN identifier

```
                          | BOOLEAN <assign_boolean>
```

3) Assignments

<assign_integer> ::= identifier <- <integer_operation>

```
                       | identifier <increment>
                       | identifier <decrement>
```

<assign_float> ::= identifier <- <float_operation>

```
                     | identifier <increment>
                     | identifier <decrement>
```

<assign_string> ::= identifier <- string;

<assign_boolean> ::= identifier <- <boolean_operation>

4) Loops

<for_loop> ::= FOR(<declarations>; <boolean_expression>; <assignments>) { <statements> }

<while_loop> ::= WHILE(<boolean_expression>) { <statements> }

5) Conditionals

<if_else_statement> ::= IF(<boolean_expression>) {<statements>}

| IF(<boolean_expression>) {<statements>} ELSE {<statements>}

6) Functions

<declare_function> ::= <types> FUNCTION identifier(<parameters>) {<function_body>}

| void FUNCTION identifier(<parameters>){<void_function_body}

<function_body> ::= <statements> RETURN identifier

| <empty>

<void_function_body> ::= <statements>

| <empty>

<parameters> ::= <empty>

| <parameter>

| <parameter>, <parameters>

<parameter> ::= INT identifier

| FLOAT identifier

| STRING identifier

| BOOLEAN identifier

<call_function> ::= identifier(identifier)

| identifier()

```
<call_primitive_function> ::= CONNECT_DRONE_TO_WIFI(<string_factor>, <string_factor>)
                            | DISCONNECT_DRONE_FROM_WIFI()
                            | READ_HEADING() | GET_HEADING()
                            | READ_ALTITUDE() | GET_ALTITUDE()
                            | READ_SPRAY_STATUS() | GET_SPRAY_STATUS()
                            | READ_TEMPERATURE()
                            | READ_WIFI_CONNECTION_STATUS()
                            | READ_WIFI_CONNECTION_INFORMATION()
                            | GET_VERTICAL_VELOCITY()
                            | GET_HORIZONTAL_VELOCITY()
                            | CLIMP_UP(<float_number>)
                            | DROP_DOWN(<float_number>)
                            | STOP_VERTICALLY()
                            | MOVE_FORWARD(<float_number>)
                            | MOVE_BACKWARD(<float_number>)
                            | STOP_HORIZONTALLY()
                            | TURN_LEFT(<integer_number>)
                            | TURN_RIGHT(<integer_number>)
                            | TURN_NORTH()
                            | TURN_EAST()
                            | TURN_WEST()
                            | TURN_SOUTH()
                            | TURN_NORTH_EAST()
                            | TURN_NORTH_WEST()
                            | TURN_SOUTH_EAST()
                            | TURN_SOUTH_WEST()
                            | TURN_ON_SPRAY()
                            | TURN_OFF_SPRAY()
```

<types> ::= INT

| FLOAT

| STRING

| BOOLEAN

7) Operations

<integer_operation> ::= <integer_operation> <add_subtract_operator> <integer_term>

| <integer_term>

<integer_term> ::= <integer_term> <multiplication_division_operator> <integer_factor>

| <integer_factor>

<integer_factor> ::= (<integer_operation>) | <integer_number>

<integer_number> ::= INTEGER | identifier

<float_operation> ::= <float_operation> <add_subtract_operator> <float_term>

| <float_term>

<float_term> ::= <float_term> <multiplication_division_operator> <float_factor>

| <float_factor>

<float_factor> ::= (<float_operation>) | <float_number>

<float_number> ::= FLOAT | identifier

<boolean_operation> ::= <boolean_operation> <boolean_operator> <boolean_term>

| <boolean_term>

<boolean_term> ::= (<boolean_expression>)

| <boolean_factor>

| NOT <boolean_factor>

<boolean_factor> ::= TRUE | FALSE | identifier

<boolean_operator> ::= AND | NAND | OR | NOR | XOR | NOT

<comparison_operator> ::= < | <= | > | >= | == | NOT=

<incerement> ::= \+\+

<decrement> ::= \-\-

<multiplication_division_operator> ::= * | /

<add_subtract_operator> ::= + | -

<boolean_expression> ::= <boolean_number_expression>

| <boolean_operation>

<boolean_number_expression> ::= <integer_operation> <comparison_operator> <integer_operation>

| <float_operation> <comparison_operator> <float_operation>

<string_factor> ::= STRING | identifier

8) Input & Output

<input> ::= INPUT(identifier);

<printable> ::= <integer_operation>
| <float_operation>
| <boolean_operation>
| <string_factor>

<printables> ::= <empty>
| <printable>
| <printable> + <printables>

<print> ::= PRINT(<printables>);
| PRINTLINE(<printables>);

<empty> ::= NULL

Language Constructs

<initialState> : According to our language, every program will start with initial state which directs to the “activate” part of the language.

<activate> : Activate is the main structure of the program which will be run when execution occurred. That is why commands and other codes(statements) which under the activation will be executed one by one. Please see the implementation:

EXAMPLE:

```
ACTIVATE_DRONE
        <statements>
STOP_DRONE
```

<statements> : Statements can be seen as collection fo the single states where all will be end with “;”. Please see the example:

EXAMPLE:

statement1;

statement2;

statement3;

.

.

.

<statement> : In this language, statement can have with four different approahces: comments, loops, if else statements and expressions.

<comment> : In this each comment content must be put between “COMMENT” “END_COMMENT”

EXAMPLE:

COMMENT This is explanation for drone implementation END_COMMENT

<content> : It should be seen as the message of the comment.

<expressions> : Expression can be three different structures: input, output, assignments and declarations.

<input> : In this language, input will be taken with INPUT(identifier) function.

Examples:

INT numberInput;

INPUT(numberInput); COMMENT input will be expected as integer END_COMMENT

FLOAT numberFloatAsInput;

INPUT(numberFloatAsInput); COMMENT input will be expected as float END_COMMENT

STRING stringInput;

STRING(stringInput); COMMENT input will be expected as string END_COMMENT

<output> : In this language, output can be given with two different print functions: PRINT(<printables>) and PRINTLINE(<printables>) where PRINTLINE automatically will contain “\n” newline at the end.

Examples:

CONSTANT_STRING printableString <- “Print me!”;

PRINT(printableString);

INT printableNumber <- 20;

PRINT(printableNumber);

<declarations> : Declarations is collection of function declarations, function calling, primitive function calling, (constant) integer declarations, (constant) float declarations, (constant) string declarations, and boolean declarations.

<assignments> : In our langauge, assignments is consisted of the following operations: integer/float/string/boolean assigning.

<declare_integer> : In our language, integers are declared like the following:

INT costOfSingleTicket;

INT numberOfTickets <- 20;

INT totalProfit <- costOfSingleTicket * numberOfTickets;

COMMENT Assume costOfSingleTicket has a value END_COMMENT

<declare_constant_integer> : Constant integer declaration example can be found below:

CONSTANT_INT SPEEDOFLIGHT <- 5000;

<declare_float> : In our language, floats are declared like the following:

FLOAT speedOfFirstCar;

FLOAT speedOfSecondCar <- 100.0;

FLOAT speedOfThirdCar <- 105.5

FLOAT differencesOfSpeed <- speedOfThirdCar - speedOfSecondCar;

<declare_constant_float> : Constant float declaration example can be found below:

```
CONSTANT_FLOAT SPEEDOFLIGHT <- 5000;
```

<declare_string> : In our language, strings are declared like the following:

```
STRING firstString;
STRING secondString <- “I am the second string!”;
STRING thirdString <- secondString;
```

<declare_constant_string> : Constant string declaration example can be found below:

```
CONSTANT_STRING CONSSTRING <- “You cannot change me”;
```

<declare_boolean> : In our language, booleans are declared like the following:

```
BOOLEAN firstBoolean;
BOOLEAN secondBoolean <- true;
BOOLEAN thirdBoolean <- firstBoolean AND secondBoolean;
COMMENT Assume firstBoolean has a value END_COMMENT
```

<assign_integer> : In our language integer assigning can be done with three ways: operations, increment, decrement.

```
firstNumber <- 10;
secondNumber <- firstNumber;
thirdNumber <- firstNumber – secondNumber;
thirdNumber++;
thirdNumber--;
```

<assign_float> : In our language float assigning can be done with three ways: operations, incerement, decrement. Similar to the assigning integer.

```
firstNumber <- 10.25;
secondNumber <- firstNumber;
thirdNumber <- firstNumber – secondNumber;
```

```
thirdNumber++;
thirdNumber--;
```

<assign_string> : String identifier can be assigned with the following ways:

```
firstString <- “String is incoming”;
secondString <- firstString;
```

<assign_boolean> : Assigning boolean can be done with following ways.

```
firstBoolean <- TRUE;
secondBoolean <- firstBoolean AND false;
```

<for_loop> : Implementation of the for loop is similar with the both C languages and the Java. See the example below:

```
FOR( INT index <- 0; index < 10; index++)
{
        COMMENT for codes END_COMMENT
}
```

<while_loop> : Implementation of the while loop also similar to the both C languages and the Java. Example:

```
 INT numberFirst <- 5;
INT numberSecond <- numberFirst * number First
WHILE( numberFirst < numberSecond)
{
        COMMENT while codes END_COMMENT
        numberFirst++;
}
```

<if_statment> : Usage of if (else) is similar to the C family and Java. Parentheses are essential.

```
BOOLEAN trueBoolean <- TRUE;
```

```
IF(trueBoolean)
{
        COMMENT if codes END_COMMENT
}
ELSE
{
        COMMENT else codes END_COMMENT
}
```

<declare_function> : The declaration of function is similar to traditional Java declarations however the keyword "FUNCTION" should be used before function_identifier. If function's return type is not void there should be a return command at the end.

The prototype of the function is:

```
return_type  FUNCTION function_name(parameters)
{
        <function body> or <void_function_body>
}
```

Example:

```
INT FUNCTION integerIncrement(INT number)
{
        number++;
        RETURN number;
}
```

<function_body> : This function body contains return command for the non-void functions.

<void_function_body> : This function body is just for the void function which does not contains return type.

<parameters> : List of parameter(s).

<parameter> : Parameter can be for INT, FLOAT, STRING, BOOLEAN with their identifier.

<call_function> : It will be used to invoke the functions. Like the following:

INT result <- integerIncrementFunction(number);

<call_primitive_function> : Calling primitive functions of our language is pretty similar to the our call function rule. All of the primitive functions and their usage can be found below:

Examples:

CONNECT_DRONE_TO_WIFI(“WifiName”, “WifiPassword”);

DISCONNECT_DRONE_FROM_WIFI();

READ_HEADING(); COMMENT prints and returns the heading of the drone to the console END_COMMENT

GET_HEADING(); COMMENT only returns heading attribute of the drone as integer END_COMMENT

INT currentHeading <- GET_HEADING();

READ_ALTITUDE(); COMMENT prints and returns the altitude of the drone to the console END_COMMENT

GET_ALTITUDE(); COMMENT only returns altitude attribute of the drone as float END_COMMENT

FLOAT currentAltitude <- GET_ALTITUDE();

READ_SPRAY_STATUS(); COMMENT prints and returns the spray is on or off END_COMMENT

GET_ SPRAY_STATUS(); COMMENT only returns boolean result to show spray is on or off. If on result will be TRUE, else it is off result will be FALSE END_COMMENT

BOOLEAN sprayStatus <- GET_SPRAY_STATUS();

READ_TEMPERATURE(); COMMENT prints the temperature value of current place where drone is located via height calculations as float(see that in each 100 meter, temperature reduce 0.5° C) END_COMMENT

READ_WIFI_CONNECTION_STATUS(); COMMENT Prints and returns boolean that shows does drone connect to wifi or not END_COMMENT

READ_WIFI_CONNECTION_INFORMATION(); COMMENT prints and returns the information that “wifi name” and “wifi password” of currently connected wifi. END_COMMENT

GET_VERTICAL_VELOCITY(); COMMENT returns and float which represents the vertical current velocity of the drone as float END_COMMENT

FLOAT verticalVelocity <- GET_VERTICAL_VELOCITY();

GET_HORIZONTAL_VELOCITY(); COMMENT returns and float which represents the horizontal current velocity of the drone as float END_COMMENT

FLOAT horizontalVelocity <- GET_HORIZONTAL_VELOCITY();

CLIMP_UP(5.5); COMMENT Orders drone to get 5.5 m altitude END_COMMENT

DROP_DOWN(4.1); COMMENT Orders drone to lose 4.1 m alitude END_COMMENT

STOP_VERTICALLY(); COMMENT Orders drone to stop vertically(vertical velocity will be zero) END_COMMENT

MOVE_FORWARD(10.1); COMMENT Orders drone to move 10.1 m forward END_COMMENT

MOVE_BACKWARD(5.2); COMMENT Orders drone to move 5.2 m backward END_COMMENT

STOP_HORIZONTALLY(6.5); COMMENT Orders drone to stop horizontally(horizontal velocity will be zero) END_COMMENT

TURN_LEFT(50); COMMENT Orders drone to turn 50 degree to left END_COMMENT

TURN_RIGHT(60); COMMENT Orders drone to turn 60 degree to right END_COMMENT

TURN_NORTH(); COMMENT Orders drone to make its heading 0 degree END_COMMENT

TURN_NORTH_EAST(); COMMENT Orders drone to make its heading 45 degree END_COMMENT

TURN_EAST(); COMMENT Orders drone to make its heading 90 degree END_COMMENT

TURN_SOUTH_EAST(); COMMENT Orders drone to make its heading 135 degree END_COMMENT

TURN_SOUTH(); COMMENT Orders drone to make its heading 180 degree END_COMMENT

TURN_SOUTH_WEST(); COMMENT Orders drone to make its heading 225 degree END_COMMENT

TURN_WEST(); COMMENT Orders drone to make its heading 270 degree END_COMMENT

TURN_NORTH_WEST(); COMMENT Orders drone to make its heading 315 degree END_COMMENT

TURN_ON_SPRAY(); COMMENT Orders drone to turn on its spray END_COMMENT

TURN_OFF_SPRAY(); COMMENT Orders drone to turn off its spray END_COMMENT

<integer_operation>, <float_operation> : It is leftmost recursion process for making number operations that are true for presedence rules.

<integer_term>, <float_term> : This term is required to give multiplication and division operations precedence over adding and subtraction operations. It still will be used for numerical operations.

<integer_factor>, <float_factor> : This factor is used to give presendence to parentheses.

<integer_number>, <float_number> : It is used to represent integer or float numbers or integer number identifiers.

Examples for integer and float operations:

```
INT firstNumber <- 100;

INT secondNumber <- 200;

INT result <- 0;

result <- result - firstNumber - (firstNumber + secondNumber) * 25 / 5;


FLOAT firstNumber <- 100.2;

FLOAT secondNumber <- 200.1;

FLOAT result <- 10;

result <- result / /(firstNumber -  1000.25) + secondNumber) * 25 – 5 / 100;
```

<boolean_operation>, <Boolean_term> : It should be seen that it is similar to the numeric operations but this time logical expressions are evaluated with considering presence rules.

<boolean_factor> : Boolean factor can be used to represent two boolean result TRUE, FALSE or a Boolean identifier.

<boolean_operator> : For boolean operations the following operators are used in our language: AND, NAND, OR, NOR, XOR, NOT.

<**comparison_operator**> : While comparing two identifiers with identical FLOAT or INT type, following comparison operators are used in the language:

< (LESS THAN)

<= (LESS THAN OR EQUALS)

> (GREATER THAN)

>= (GREATER THAN OR EQUALS)

== (EQUALS)

NOT= (NOT EQUALS)

Boolean Expression Examples:

BOOLEAN firstBoolean <- TRUE;

BOOLEAN secondBoolean <- FALSE;

BOOLEAN thirdBoolean <- TRUE AND FALSE;

BOOLEAN result <- firstBoolean OR (secondBoolean NAND (2 NOT= 5)) AND TRUE OR thirdBoolean;

<increment> : To increment and add one to an integer or a float identifier following syntax can be used:

INT number <- 1;

number++;

FLOAT numberSecond <- 11.5;

numberSecond++;

<decrement> : To decrement and subtract one from an integer or a float identifier following syntax can be used:

INT number <- 1;

number--;

FLOAT numberSecond <- 11.5;

numberSecond--;

<multiplication_division_operator> : In this language, standard multiplication and division operators are used:

* for multiplication

/ for division

<add_subtract_operator> : In this language, standard addition and subtraction operators are used:

+ for addition

- for subtraction

<boolean_expression> : Boolean expression can be completed with two ways: Boolean number expressions or Boolean operations.

<boolean_number_expression> : To compare identifiers with identical integer or float type comparison operators are used. See the example:

Example:

```
2 <= 5

2.5 NOT= 25

INT firstNumber <- 9;

INT secondNumber <- 5;

firstNumber > secondNumber;


FLOAT firstNumber <- 9.256;

FLOAT secondNumber <- 5.111;

firstNumber >= secondNumber;
```

<string_factor> : To represent a string or an identifier which's type is string.

<printable> : Printables are basically collection of integer/float/boolean operations or basically a string factor.

<printables> : Collection of printables.

<types> : Types can be INT, FLOAT, STRING, or BOOLEAN.

<empty> : Represents null.

NONTRIVIAL TOKENS

Comments

Since the language is aimed to be as human-readible as possible, the comment structure follows this policy as well. Accordingly, a comment must start with COMMENT and end with END_COMMENT where the comment statements are placed in between. Besides, it must also be adressed that comments are given in a single line.

Identifiers

When it comes to identifiers, the language takes a similar approach with other programming languages for consistancy. In this respect, all identifiers must be at least length of one, start with a letter and the rest can be any character. The motivation behind using the same formula with other remarkable programming languages and not having any additional rules was for the language being easy to write and easy to adopt.

Literals

There are three kinds of literals in the language, namely string literals, numeric literals and boolean literals. Just like the identifers, the literals follow a similar procedure and focus on writability, readability and consistancy. Accordingly, the strings are expressed inside double quotation marks. The main reason behind this is to both show that strings are immutable and to follow the traditional approach. Numeric literals are divided into two groups as integers and floats. Similar to other programming languages, integers can only be composed of numbers and floats can take a decimal point for more complex calculations. Accordignly, two different types of numeric literals are useful to switch between simple operations and more delicate ones. Booleans can take only two values, namely TRUE or FALSE.

Reserved Words

Reserved words define the special words that can only be used for selected purposes. While selecting the types and formats of reserved words, the motivation was to protect the core functionalities of a programming language while also aiming to make it much simpler in terms of readiblity at the same time. In this perspective, the exact words defining functionalities are used in general. Using tokens named PRINT, INPUT, COMMENT, AND comes out examples of this approach. At this point, it must be addressed that known names are preferred for core expressions such as if, else, for. To differenciate constants more easily, it is arranged so that they are separately declared with <constant_> in the beginning. A detailed list that shows the reserved words and demonstrates their corresponding function is included below.

Reserved Word	**Function**
IF	Defines the beginning of if statement
ELSE	Defines the beginning of else option in if statements
FOR	Defines the start of a for loop structure
WHILE	Defines the start of a while loop structure
RETURN	Defines returning value
FUNCTION	Defines function declaration
PRINT	Defines outputting value
PRINTLINE	Defines outputting value and passing to next line
INPUT	Defines receiving value
INT	Defines integer
CONSTANT_INT	Defines constant integer
FLOAT	Defines float
CONSTANT_FLOAT	Defines constant float
STRING	Defines string
CONSTANT_STRING	Defines constant string
OR	Defines logical or operation
NOR	Defines logical nor operation
AND	Defines logical and operation
NAND	Defines logical nand operation
XOR	Defines logical xor operation
NOT	Defines logical not operation
TRUE	Defines true logic value (1)
FALSE	Defines false logic value (0)
NULL	Defines empty/non-allocated value
COMMENT	Defines start of comment
END_COMMENT	Defines end of comment
ACTIVATE_DRONE	Defines the start of main function
STOP_DRONE	Defines the end of main function
CONNECT_DRONE_TO_WIFI	Defines the name of connecting drone function
DISCONNECT_DRONE_FROM_WIFI	Defines the name of disconnecting drone function
READ_HEADING	Defines the name of reading heading function
GET_HEADING	Defines the name of returning heading function
READ_ALTITUDE	Defines the name of reading altitude function
GET_ALTITUDE	Defines the name of getting altitude function
READ_SPRAY_STATUS	Defines the name of reading spray status function
GET_SPRAY_STATUS	Defines the name of getting spray status function
READ_TEMPERATURE	Defines the name of reading temperature function

READ_WIFI_CONNECTION_STATUS	Defines the name of reading wifi connection status function
READ_WIFI_CONNECTION_INFORMATION	Defines the name of reading wifi connection information function
CLIMP_UP	Defines the name of climb up function
DROP_DOWN	Defines the name of drop down function
STOP_VERTICALLY	Defines the name of stop vertically function
MOVE_FORWARD	Defines the name of move forward function
MOVE_BACKWARD	Defines the name of move backward function
STOP_HORIZONTALLY	Defines the name of stop horizontally function
TURN_LEFT	Defines the name of turn left function
TURN_RIGHT	Defines the name of turn right function
TURN_NORTH	Defines the name of turn north function
TURN_EAST	Defines the name of turn east function
TURN_WEST	Defines the name of turn west function
TURN_SOUTH	Defines the name of turn south function
TURN_NORTH_EAST	Defines the name of turn north east function
TURN_NORTH_WEST	Defines the name of turn north west function
TURN_SOUTH_EAST	Defines the name of turn south east function
TURN_SOUTH_WEST	Defines the name of turn south west function
TURN_ON_SPRAY	Defines the name of turn on spray function
TURN_OFF_SPRAY	Defines the name of turn off spray function
GET_VERTICAL_VELOCITY	Defines the name of function returning vertical velocity component
GET_HORIZONTAL_VELOCITY	Defines the name of function returning horizontal velocity component

Figure 1: Reserved words and their meanings

Evaluation of HUDSON

1) Readability

The readability of a language can be seen as how long does it take to understand the code which used by that language. The understanding language process is directly related with the language's grammar rules' accuracy, logic, memorability. In order to make HUDSON readable, it is tried to protect general grammar rules of C family languages that are popular in the programming world, however, as a team we embraced one idea about our language to make it more readable. The idea is making reserved words and every kind of special words, functions, attributes that are coming from our language itself uppercase. That is why when the programmer reads code that is created via HUDSON, he/she can realize any special words of our language easily. As example, instead of "if", we used "IF"; instead of "int", we used "INT"; instead of "while", we used "WHILE", etc. Also, we change some of the logical expression symbols such as "&&", "||", "!=". Instead of using these symbols, we just determined new upper cased reserve words that represents exactly their meanings. As an example, instead of using "&&" for "and", we just used "AND" for "and"; instead of using "||" for "or", we just used "OR" word for "or"; and finally instead of using "!=" for "not equal", we just used "NOT=" for "not equal". Furthermore, we paid extra attention to not use short forms of the reserved words. As an example to implement constant float we used "CONSTANT_FLOAT". Another example can be instead of "bool" we used "BOOLEAN". We pick this approach because it makes the meaning of reserved words clearer for the user. Our only exception about this rule is we still considering integer type as INT.

With these changes, our aim was make reader to understand to code's function more easily and make it more readable.

2) Writability

The writability of a language can be seen as how it is easy to write and does expressing the coder's idea requires any kind of obstacle that comes from the language's grammar, reserved words, or selected operators. Also, it is important that how many alternative syntax approach there are to express one operation. In terms of operator choice, we tried to determine operators' purposes according to their common role in the world. For example "=" sign represents equality in the mathematical perspective, however, it is common that in many programming languages, "=" symbol can be used to assign variable purposes. We tried to not use this common approach but instead of we aimed to achieve use every symbol in related operations. As an example we define a left arrow symbol (<-) to show variable assign processes. At this part, it also can be added that in our language all reserved words are designed to be fully upper case which makes it more possible users to understand which parts are coming from the HUDSON and which parts belongs to the user. Furthermore, we claim that our approach to use meaningful reserved words for logical expressions such as instead of "&&" using "AND" operator to express "logical and" makes it easier to express coders themselves in the HUDSON. However, we accept that our language grammar rules limit user's freedom to express an operation with different ways. For example, we do not allow to use operation of two variables with two different types in order to increase the reliability. Also, user needs to know how to describe float and integer number in our language. In our language float description should include decimal dot symbol. For example, writing “0.0” is used describing float number, however, “0” is used to describe integer. All of these sacrifices are made to make it possible to increase readability and reliability of the HUDSON.

3) Reliability

In HUDSON, numeric and logical presence rules are critically adjusted. It should be seen that to control and manage a drone every numeric calculations should be calculated without any potential error which might come from the language in some certain cases. We fix this issue by designing our BNF according to the precedence rules. As an example, multiplication and division operators have higher precedencies than addition and subtraction operators. Parentheses can be used to give precedence to specific operations as well such as 3 * (2+5). In order to increase our language’s reliability we considered the logical operations’ precedence rules as well that includes associativity as well. Also, in order to increase our language’s reliability we decided that any variable type can be used in operation with another variable which must have the same type with the first variable. We claim that all of these considerations increase the reliability of our language.