Bilkent University

Department of Computer Engineering

CS 315 Project 1

Design Report

Can Kırımca 21802271 Section-2

Can Kırşallıoba 21801768 Section-2

RUNE

1) Program Definition

```
<program> ::= <statement_list>

<statement_list> ::= <statement>

                   | <statement_list> <statement>

<statement> ::= COMMENT

              | <expression> SEMICOLON

              | <loop>

              | <if>

              | <function_definition>
```

2) Types and Constants

```
<data_type> ::= INT_TYPE

              | FLOAT_TYPE

              | CHAR_TYPE

              | BOOLEAN_TYPE
```

3) Loop Definition

```
<loop> ::= <while_loop>

         | <for_loop>

<while_loop> ::= WHILE LP <logical_expression> RP LCB <statement_list> RCB

<for_loop> ::= FOR LP <for_loop_expression> RP LCB <statement_list> RCB

<for_loop_expression> ::= <expression> SEMICOLON BOOLEAN; <expression>
```

4) Conditional Definition

<if> ::= IF LP <logical_expression> RP LCB <statement_list> RCB <else>

| IF LP <logical_expression> RP LCB <statement_list> RCB

<else> ::= ELSE LCB <statement_list> RCB

5) Expressions

<expression> ::= <expression> PLUS <expression2>

| <expression> MINUS <expression2>

| <expression2>

<expression2> ::= <expression2> MULTIPLICATION <expression3>

| <expression2> DIVISION <expression3>

| <expression2> REMAINDER <expression3>

| <expression3>

<expression3> ::= <expression4> EXPONENTIATION <expression3>

| <expression4>

<expression4> ::= LP <expression> RP

| <expr>

<expr> ::= INT

| FLOAT

| <function_call>

| <logical_expression>

| IDENTIFIER

<function_call> ::= IDENTIFIER LP <argument_list> RP

```
<logical_expression> ::= BOOLEAN
                       | <comparison>
                       | <basic_equality>
                       | <function_call>
                       | <logical_expression> AND <logical_expression>
                       | <logical_expression> OR <logical_expression>
<basic_equality> ::= (INT | FLOAT | BOOLEAN | IDENTIFIER | <function_call>)
                     (EQUAL | NOT_EQUAL)
                     (INT | FLOAT | BOOLEAN | IDENTIFIER | <function_call>)
<comparison> ::=  (INT | FLOAT | IDENTIFIER | <function_call>)  <comparison_operator>
                  (INT | FLOAT | IDENTIFIER | <function_call>)
<comparison_operator> ::= GREATER_THAN
                        | SMALLER_THAN
                        | GREATER_OR_EQUAL
                        | SMALLER_OR_EQUAL
<assignment> ::= IDENTIFIER ASSIGNMENT_OPERATOR <expression>
               | <data_type> IDENTIFIER ASSIGNMENT_OPERATOR <expression>
```

6) Function Definition

```
<function_def> ::= <void_with_return>
                 | <void_without_return>
                 | <non_void_func_def>
                 | <only_return_func_def>
```

```
<void_with_return> ::= VOID_TYPE IDENTIFIER LP <argument_list> RP LCB
                                 <statement_list> <void_return_statement> RCB
<void_without_return> ::= VOID_TYPE IDENTIFIER LP <argument_list> RP LCB
                                     <statement_list> RCB
<non_void_func_def> ::= <data_type> IDENTIFIER LP <argument_list> RP LCB
                                   <statement_list> <return_statement> RCB
<only_return_func_def> ::= <data_type> IDENTIFIER LP <argument_list> RP LCB
                                      <return_statement> RCB
<void_return_statement> ::= RETURN SEMICOLON
<return_statement> ::= RETURN <expression> SEMICOLON
<argument> ::= INT
                   | FLOAT
                   | CHAR
                   | BOOLEAN
                   | IDENTIFIER
                   | <function_call>
<argument_list> ::= <argument> COMMA <argument_list>
                        | <argument>
                        | empty
```

7) Input Output Definition

```
<input_statement> ::= SCAN LP <expression> RP SEMICOLON
<output_statement> ::= PRINT LP <expression> RP SEMICOLON
```

8) Drone Functions

<read_inclination_function> ::= READ_INCLINATION LP RP SEMICOLON

<read_altitude_function> ::= READ_ALTITUDE LP RP SEMICOLON

<read_temperature_function> ::= READ_TEMPERATURE LP RP SEMICOLON

<read_acceleration_function> ::= READ_ACCELERATION LP RP SEMICOLON

<set_camera_state_function> ::= SET_CAMERA_STATE LP BOOLEAN RP SEMICOLON

<take_picture_function> ::= TAKE_PICTURE LP RP SEMICOLON

<read_timestamp_function> ::= READ_TIMESTAMP LP RP SEMICOLON

<connect_to_computer_function> ::= CONNECT_TO_COMPUTER LP RP SEMICOLON

9) Our Extra Functions

<do_flip_function> ::= DO_FLIP LP CHAR RP SEMICOLON

<takeoff_function> ::= TAKEOFF LP RP SEMICOLON

<land_function> ::= LAND LP RP SEMICOLON

<emergency_function> ::= EMERGENCY LP BOOLEAN RP SEMICOLON

<up_function> ::= UP LP (INT | FLOAT) SEMICOLON

<down_function> ::= DOWN LP (INT | FLOAT) RP SEMICOLON

<right_function> ::= RIGHT LP (INT | FLOAT) RP SEMICOLON

<left_function> ::= LEFT LP (INT | FLOAT) RP SEMICOLON

<forward_function> ::= FORWARD LP (INT | FLOAT) RP SEMICOLON

<backward_function> ::= BACKWARD LP (INT | FLOAT) RP;

<rotate_clockwise_function> ::= ROTATE_CLOCKWISE LP BOOLEAN COMMA INT RP
SEMICOLON

<set_speed_function> ::= SET_SPEED LP (INT | FLOAT) RP SEMICOLON

<get_speed_function> ::= GET_SPEED LP RP SEMICOLON

<get_battery_function> ::= GET_BATTERY LP RP SEMICOLON

Explanation of the RUNE Language Constructs

Types

In RUNE, there are 4 data types, which are the following: int, float, boolean, and char.

Symbols

Assignment Operator (<-):

In our language assignment operator is coded as <-. The reason we chose <- as assignment operator is; since the assignments are taking place from the right-hand side to the left-hand side, it is very intuitive to use it. Hence, increasing the overall readability and writability of the code. An example of its usage is as follows:

```
a <- 5;
```

In the example above, 5 is assigned to the variable a.

Variables:

The user can define and use variables in the language. This variable can take form of four different types of data as explained in the "Types" section. Example of defining and using a variable is as follows:

```
float a;
a <- 5.4;
int b <- 87;
```

Comment ($):

Comments are indicated with $ (dollar sign) in RUNE. Any text between two comment signs ($) is considered a comment. A comment can be an in-line comment, or it can have multiple lines. Since the dollar sign is not a common symbol to be found in code, it is easily identifiable in blocks of code. Thus, deciding on the dollar sign as our comment sign made the language more readable and writable. For instance;

```
$ This is an example comment. $
```

Exponentiation Operator (**):

Exponentiation operation will be implemented, just like in Python language.

An example of its usage would be as follows:

```
a <- 2**3;
```

In this example, 2 to the power of 3, which is equal to 8, is assigned to the variable a.

In RUNE, the exponentiation operator is right-associative. An example statement is as follows:

```
a <- 2**3**2
```

In this example, the exponentiation on the right-hand side is calculated first. After calculating 3 to the power of 2, which is equal to 9, the exponentiation on the left-hand side is calculated. In this case, it is 2 to the power of 9, which is equal to 512. Therefore, 512 is assigned to the variable a.

Program Definitions

A program in RUNE consists of a statement list. A statement list can be either one statement or multiple statements. Statements include comments, expressions followed by a semicolon, loops, conditionals, or a function definition. These expressions will be explained in their respective sections.

Loop Definitions

We have two loop types in RUNE: while loop and for loop. Even though the syntax is similar to the syntax of C language, the statements inside a for or while block must be enclosed between curly brackets regardless of the number of statements. Example usage of those loops is as follows:

```
while( height < 30 ){
    height <- height + 1;
}

for( int i <- 0; i < 5; i <- i + 1){
    doFlip(‘r’);
}
```

We implemented for loop as well as while loop to offer the choice of writing more functional counting loops when the user wishes to. Hence, increasing the expressivity of the language.

Conditional Definitions

In RUNE, we have two types of conditional in order to keep the language simple and friendly for beginners. The two conditionals we have are the if statement and else statement. An else could only follow an if statement. However, an if statement can be a standalone block. If the condition inside the if statement is not satisfied, statements inside the else block (if there is an else block) are executed. The syntax of if and else statements are inspired by C language. The statements after an if or else must be enclosed between curly brackets, regardless of the number of statements. Here is an example if-else block:

```
if (myNum > 6) {
        print("myNum is bigger than 6.");
} else {
        print("myNum is smaller or equal to 6.");
}
```

Expressions

We have gathered all the expressions in this section of the code. Here, we have expressions of integer types, float types, logical expressions, functions, and operators. For the associativity and the precedence of the operators, we have followed the pattern that was explained to us by our instructor and in our book. We have separated the operators that have different precedences. The operators with the lowest precedence (+, -) are higher in the hierarchy, and the operator with the highest precedence is at the lowest of the hierarchy. The following table illustrates operator precedence in RUNE:

Operator	**Precedence**	**Associativity**
(	5	**Left to Right**
)	5	**Left to Right**
**	4	**Right to Left**
/	3	**Left to Right**
%	3	**Left to Right**
*	3	**Left to Right**
-	2	**Left to Right**
+	2	**Left to Right**
&&	1	**Left to Right**
||	1	**Left to Right**

Table 1. *Demonstration of the precedence values, larger number means higher precedence.*

When we refer to an expression throughout the file, we mean that it can be an integer token, a float token, a logical expression, and a function expression.

Int Token: It includes integers.

Float Token: It includes floats.

Function Expression: It includes an identifier which is the name of the function, a pair of parentheses, and an argument list in those parentheses that will be passed to the function.

Logical Expression: Logical Expression: This is an expression that results in a Boolean value. It can be in the following forms:

- TRUE or FALSE
- A function call that returns a Boolean value.

- A comparison (<, >, <=, >=) between two numerical values (can be a function call that returns a numerical value).
- An equality (==, !=) check between two numerical or Boolean values.

Function Definitions

We were inspired by the C language for the function definitions, as well. We have a return type, which can be one of the data types we have introduced to the language. However, if the user does not want to return anything in that function, the return type can also be void. For the function's arguments, we have enabled floats, chars, booleans, identifiers, and function calls. A function may have no argument as well. Every non-void function has a return statement indicating the value returned by the function. A function can also return the value of an expression (such as “a + 5”). A void function can have an empty return statement (return;) as well. In that case, the return statement basically terminates the function call. Here are some example function definitions:

```
int foo(int num) {
        print(“Inside the function”);
        if(num == 1){
                foo2();
          }
        return num + 1;
}

int getHeight() {
        return height;
}
```

```
void foo3(int num) {
        if (num == 1) {
                return;
        }
        else {
                num <- num + 1;
        }
}
```

Input and Output Definitions

For the input and output definitions, we have selected the “scan” and “print” keywords. “scan” gets an int value from the user, and “print” outputs the given expression to the console. We chose these two reserved words because they are very intuitive to use for beginners. Also, choosing intuitive words instead of complicated ones enhances the overall readability and writability of the language. We may enhance the “scan” function in the next stage of the project to accept different data types as float, char, etc.

Drone Function Definitions

The drone functions which are described in the project document will be implemented as inbuilt functions.

- **readInclination()** function takes the data from the gyroscope and returns the value.
- **readAltitude()** function takes the data from the barometer and returns the value.
- **readTemperature()** function takes the data from the thermometer and returns the value.
- **readAcceleration()** function takes the data from the acceleration sensor and returns the value.
- **setCameraState()** function changes the state of the camera according to the boolean value passed as an argument. TRUE turns on the camera, and FALSE turns off the camera.
- **takePicture()** function uses the camera that is on the drone to take pictures.
- **readTimestamp()** function gets the time from the timer, then returns the value.
- **connectToComputer()** function forms a connection between the drone and the base computer via wifi.

Our Creative Functions

We looked at the Software Development Kit and decided to include some extra functionality of the commercial drone “Tello”. We have a variety of inbuilt functions that include but are not limited to taking off, doing flips, moving in the specified directions, etc. We will explain these functions in great detail down below.

- **doFlip()** functionality flips the drone in the specified direction. The user can specify the direction by passing the corresponding char value as an argument. The associated char values are as follows:

  ‘f’: Frontflip
  ‘b’: Backflip
  ‘r’ Right flip
  ‘l’: Left flip
- **takeOff()** function takes off the drone when the function is called.
- **land()** function lands the drone when the function is called.
- **emergency()** function stops the drone in case of an emergency. It takes a boolean value as an argument. If the value is TRUE, the drone stops. If it is false, the drone goes back to its normal state.
- **up()** function moves the drone upwards.
- **down()** function moves the drone downwards.
- **right()** function moves the drone in the right direction.
- **left()** function moves the drone in the left direction.
- **forward()** function moves the drone in forward direction.
- **backward()** function moves the drone in backward direction.
- **rotateClockwise()** function rotates the drone by looking at the passed boolean value. If the user passes TRUE, the drone rotates clockwise; if the user passes FALSE, the drone rotates counter clockwise.
- **setSpeed()** function sets the drone’s speed to the integer value passed as an argument.
- **getSpeed()** function gets the drone’s speed.
- **getBattery()** function gets the current value of the battery.

Evaluation of Our Language in Terms Of Readability, Writability, And Reliability

Readability

Our language is designed to be easy to use for users ranging from beginners to experts. The syntax is very close to C group languages. We chose this approach because the C group languages are around for a very long time, and people have grown accustomed to the syntax of them. We have paid close attention not to disregard the orthogonality of the language. We did not assign an operator to multiple operations, hence making RUNE appropriate to orthogonality. When it comes to data types, they are assigned to their respective identifiers to avoid confusion. For instance, the value of a float variable cannot be assigned to an integer variable.

Writability

Our language has an adequate number of data types and functionalities so that it does not confuse the writer of the code. Since we have a small number of primitive constructs and consistent sets of rules for combining them, we have enabled orthogonality for the language. We created different and easy forms of writing the same expression to enhance the expressivity of the language, such as using for loops instead of while loops.

Reliability

We tried to create the language so that it performs to its specifications under all sorts of conditions. For instance, in terms of type checking, we have enabled one of our functions to accept speed values for both int types and float types. At this stage, we did not implement any sort of exception handling system because we only wrote the BNF and the lex specification file. The same applies to aliasing. For readability and writability, we made sure that all the conditions for them are met, thus enhancing our language's reliability.

References

R. W. Sebesta, *Concepts of programming languages*. NY: Pearson, 2019.