Bilkent University

Department of Computer Engineering

CS 315 Project 2

Project Report

Can Kırımca 21802271 Section-2

Can Kırşallıoba 21801768 Section-2

BNF description of RUNE

1) Program Definition

```
<program> ::= <statement_list>
<statement_list> ::= <statement>
                   | <statement_list> <statement>

<statement> ::= COMMENT
              | <expression> SEMICOLON
              | <loop>
              | <if>
              | <function_definition>
              | <assignment> SEMICOLON
```

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 <expression> RP LCB <statement_list> RCB
<for_loop> ::= FOR LP <for_loop_expression> RP LCB <statement_list> RCB
<for_loop_expression> ::= <assignment> SEMICOLON <expression> SEMICOLON
<assignment>
```

4) Conditional Definition

```
<if> ::= IF LP <expression> RP LCB <statement_list> RCB <else>
       | IF LP <expression> RP LCB <statement_list> RCB
<else> ::= ELSE LCB <statement_list> RCB
```

5) Expressions

```
<expression> ::= <expression> OR <expression2>
               | <expression2>

<expression2> ::= <expression2> AND <expression3>
                | <expression3>

<expression3> ::= <expression3> <equality_operator> <expression4>
                | <expression4>

<expression4> ::= <expression4> <comparison_operator> <expression5>
                | <expression5>

<expression5> ::= <expression5> PLUS <expression6>
                | <expression5> MINUS <expression6>
                | <expression6>

<expression6> ::= <expression6> MULTIPLICATION <expression7>
                | <expression6> DIVISION <expression7>
                | <expression6> REMAINDER <expression7>
                | <expression7>

<expression7> ::= <expression8> EXPONENTIATION <expression7>
                | <expression8>

<expression8> ::= LP <expression> RP
                | <expr>

<expr> ::= INT
         | FLOAT
         | CHAR
         | BOOLEAN
         | <function_call>
         | IDENTIFIER
```

```
<function_call> ::= IDENTIFIER LP <function_call_argument_list> RP
                  | IDENTIFIER LP RP
                  | <primitive_function_call>
                  | <input_statement>
                  | <output_statement>

<equality_operator> ::= EQUAL
                      | NOT_EQUAL

<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_definition> ::= <void_with_return>
                        | <void_without_return>
                        | <non_void_func_def>
                        | <only_return_func_def>

<void_with_return> ::= VOID_TYPE IDENTIFIER LP <function_definition_argument_list>
                       RP LCB <statement_list> <void_return_statement> RCB
                       | VOID_TYPE IDENTIFIER LP RP LCB <statement_list>
                       <void_return_statement> RCB

<void_without_return> ::= VOID_TYPE IDENTIFIER LP
                          <function_definition_argument_list> RP LCB <statement_list>
                          RCB
                          | VOID_TYPE IDENTIFIER LP RP LCB <statement_list> RCB
```

```
<non_void_func_def> ::= <data_type> IDENTIFIER LP
                        <function_definition_argument_list> RP LCB <statement_list>
                        <return_statement> RCB
                        | <data_type> IDENTIFIER LP RP LCB <statement_list>
                        <return_statement> RCB

<only_return_func_def> ::= <data_type> IDENTIFIER LP
                           <function_definition_argument_list> RP LCB
                           <return_statement> RCB
                           | <data_type> IDENTIFIER LP RP LCB <return_statement>
                           RCB

<void_return_statement> ::= RETURN SEMICOLON

<return_statement> ::= RETURN <expression> SEMICOLON

<function_definition_argument_list> ::= <function_definition_argument> COMMA
                                         <function_definition_argument_list>
                                        | <function_definition_argument>

<function_definition_argument> ::=  INT_TYPE IDENTIFIER
                                   | FLOAT_TYPE IDENTIFIER
                                   | CHAR_TYPE IDENTIFIER
                                   | BOOLEAN_TYPE IDENTIFIER

<function_call_argument_list> ::= <function_call_argument> COMMA
                                  <function_call_argument_list>
                                  | <function_call_argument>
<function_call_argument> ::= INT
                             | FLOAT
                             | CHAR
                             | BOOLEAN
                             | IDENTIFIER
                             | <function_call>
```

7) Input Output Definition

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

8) Drone Functions

```
<primitive_function_call> ::= <read_inclination_function>
                            | <read_altitude_function>
                            | <read_temperature_function>
                            | <read_acceleration_function>
                            | <turn_on_camera_function>
                            | <turn_off_camera_function>
                            | <take_picture_function>
                            | <read_timestamp_function>
                            | <connect_to_computer_function>
                            | <takeoff_function>
                            | <land_function>
                            | <get_flight_time_function>

<read_inclination_function> ::= READ_INCLINATION LP RP
<read_altitude_function> ::= READ_ALTITUDE LP RP
<read_temperature_function> ::= READ_TEMPERATURE LP RP
<read_acceleration_function> ::= READ_ACCELERATION LP RP
<turn_on_camera_function> ::= TURN_ON_CAMERA LP RP
<turn_off_camera_function> ::= TURN_OFF_CAMERA LP RP
<take_picture_function> ::= TAKE_PICTURE LP RP
<read_timestamp_function> ::= READ_TIMESTAMP LP RP
<connect_to_computer_function> ::= CONNECT_TO_COMPUTER LP RP
<takeoff_function> ::= TAKEOFF LP RP
<land_function> ::= LAND LP RP
<get_flight_time> ::= GET_FLIGHT_TIME LP RP
```

Explanation of the RUNE Language Constructs

Types

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

Int type:

Int type consists of positive integers, negative integers and zero.

Float type:

Float type takes numbers that are in decimal form. The syntax of the float numbers can be written as .4 or 0.4. Float numbers can also range between negative and positive values.

Char type:

Char type consists of the single characters in the Unicode. The char type should be written in single quotation marks.

Boolean Type:

Boolean type consists of two values: TRUE and FALSE. They are reserved key words so they cannot be used in naming other user defined variables.

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;
```

Operators

In RUNE, six operators are defined. These operators are:

- Assignment operator

- Addition operator
- Subtraction operator
- Multiplication operator
- Division operator
- Exponentiation operator
- Remainder operator
- AND operator
- OR operator

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.

Addition Operator (+):

The addition operator is coded as “+”. An example usage is as follows:

```
a <- 5 + 3;
```

In this example, 5 + 3 = 8 is assigned to the variable a.

Subtraction Operator (-):

The subtraction operator is coded as “-”. An example usage is as follows:

```
a <- 7 - 5;
```

In this example, 7 – 5 = 2 is assigned to the variable a.

The addition and subtraction operators also indicate the positivity/negativity of a number when they are written adjacent to the number (-5 indicates negative 5). Therefore, in addition and subtraction operations, there must be a space between the operands; otherwise, the parser gives a syntax error. An example usage is as follows:

```
a <- 8 – 6 + 2;
```

Multiplication Operator (*):

The multiplication operator is coded as “*”. An example usage is as follows:

a <- 3 * 4;

In this example, 3 * 4 = 12 is assigned to the variable a.

Division Operator (/):

The division operator is coded as “/”. An example usage is as follows:

a <- 8 / 2;

In this example, 8 / 2 = 4 is assigned to the variable a.

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.

Remainder Operator (%):

The remainder operator is coded as %. It returns the remainder left over when one operand is divided by a second operand. An example usage is as follows:

a <- 8 % 5;

In this example, 8 % 5 = 3 is assigned to a.

AND Operator (&&):

If the two operands of the AND operator are both TRUE then it returns TRUE. Otherwise, it returns FALSE.

OR Operator (||):

If at least one of the operators is TRUE, the operator returns TRUE. Otherwise, it returns FALSE.

Comments

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. $
```

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 ) {
        print(height);
        height <- height + 1;
}

for( int i <- 0; i < 5; i <- i + 1) {
     takePicture();
}
```

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) {
        myNum <- 0;
} else {
        myNum <- myNum + 1;
}
```

The language allows each type of expression in an if statement. If this expression is a numerical value, all values except zero is considered TRUE and the statements in the block are executed.

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 precedence. 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
(	9	**Left to Right**
)	9	**Left to Right**
**	8	**Right to Left**
/	7	**Left to Right**
%	7	**Left to Right**
*	7	**Left to Right**
-	6	**Left to Right**
+	6	**Left to Right**
<	5	**Left to Right**
<=	5	**Left to Right**
>	5	**Left to Right**
>=	5	**Left to Right**
==	4	**Left to Right**
!=	4	**Left to Right**
&&	3	**Left to Right**
||	2	**Left to Right**
<-	1	**Not Associative**

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

Assignment: The result of any numerical expression can be assigned to a variable. An example assignment is as follows:

a <- b + 5;

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 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 integer, float, char and Boolean. 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 a 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.

Drone Function Definitions

The drone functions which are described in the project document will be implemented as primitive 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.
- **turnOnCamera** function turns on the drone’s camera.
- **turnOffCamera** function turns off the drone’s 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 wi-fi.
- **takeoff** function allows the drone to be airborne.
- **land** function lands the drone.
- **getFlightTime** function returns the time elapsed during the flight. Unit of the elapsed time is seconds.

Important Remarks about the General Structure of RUNE

In RUNE, we did not implement a structure such as main to initiate the code. Instead, we decided to let the user write the functions and the code structures they wish to write directly, just like in Python. The reason we did it this way is, because since RUNE is a drone programming language, we thought that it should be quick and simple to write code.

We do not have any conflicts in RUNE, we strived to eliminate all so that we would produce a language that works seamlessly.

References

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