CS 315 - Team 57
Programming Languages
Project 2
RIO Programming Language

Kutay Demiray	21901815	Section 3
Gökberk Keskinkılıç	21801666	Section 3
Yağız Yaşar	21902951	Section 3

08/11/2021

Complete BNF Description of RIO:

```
<start> -> <stmts>
<stmts> -> <stmt> | <stmts> <stmt>
<stmt> -> <conditional_stmt>
        | <declare> SEMICOLON
        | <assign> SEMICOLON
        | <func_call> SEMICOLON
        | <for_loop>
        | <while_loop>
        | <func_def>
        | <input_func> SEMICOLON
        | <output_func> SEMICOLON
        | <drone_func> SEMICOLON
        | <return_stmt> SEMICOLON
        | COMMENT

<stmts_block> -> CURLY_L <stmts> CURLY_R

<conditional_stmt> -> IF PAREN_L <bool_expr> PAREN_R <stmts_block>
<elses_list>

<elses_list> ->
              | ELSE <stmts_block>
              | ELSE_IF PAREN_L <bool_expr> PAREN_R <stmts_block>

<declare> -> TYPE IDENT | TYPE <assign>

<assign> -> IDENT ASSIGNMENT_OP <bool_expr>
          | IDENT ASSIGNMENT_OP CHAR
          | IDENT ASSIGNMENT_OP STRING
          | IDENT ASSIGNMENT_OP <drone_enum>

// boolean expressions
<bool_expr> -> <comparison_expr>
             | <bool_expr> EQUAL_OP <comparison_expr>
             | <bool_expr> NOT_EQUAL_OP <comparison_expr>
             | <bool_expr> EQUAL_OP <drone_enum>
             | <bool_expr> NOT_EQUAL_OP <drone_enum>

<comparison_expr> -> <comparison_first>
          | <comparison_first> SMALLER_THAN_OP <comparison_second>
          | <comparison_first> SMALLER_OR_EQ_OP
<comparison_second>
          | <comparison_first> GREATER_THAN_OP <comparison_second>
          | <comparison_first> GREATER_OR_EQ_OP
<comparison_second>

<comparison_first>  -> <logic_or>
```

```
<comparison_second> -> <logic_or>

<logic_or> -> <logic_and> | <logic_or> OR_OP <logic_and>

<logic_and> -> <logic_not> | <logic_and> AND_OP <logic_not>

<logic_not> -> <bool_term> | NOT_OP <logic_not>

<bool_term> -> BOOL | <arith_expr>

// arithmetic expressions
<arith_expr>  -> <arith_term> | <arith_expr> PLUS_OP <arith_term>
               | <arith_expr> MINUS_OP <arith_term>

<arith_term> -> <exponent> | <arith_term> MULT_OP <exponent>
              | <arith_term> DIV_OP <exponent>

<exponent> -> <factor> | <factor> EXP_OP <exponent>

<factor> -> IDENT | INT | REAL | <func_call> | <drone_func>
          | PAREN_L <bool_expr> PAREN_R

// loops
<for_loop> -> FOR PAREN_L <for_cond> PAREN_R <stmts_block>

<for_cond> -> <declare> SEMICOLON <bool_expr> SEMICOLON <assign>
            | <assign> SEMICOLON <bool_expr> SEMICOLON <assign>

<while_loop> -> WHILE PAREN_L <bool_expr> PAREN_R <stmts_block>

// i/o
<input_func>  -> INPUT PAREN_L IDENT PAREN_R
<output_func> -> OUTPUT PAREN_L <bool_expr> PAREN_R
               | OUTPUT PAREN_L STRING PAREN_R
               | OUTPUT PAREN_L CHAR PAREN_R

// general functions
<func_def> ->
 FUNC_DEF FUNC_IDENT PAREN_L <ident_list> PAREN_R TYPE
<stmts_block>
| FUNC_DEF FUNC_IDENT PAREN_L <ident_list> PAREN_R <stmts_block>
| MAIN PAREN_L PAREN_R <stmts_block>

<ident_list> -> | TYPE IDENT | TYPE IDENT COMMA <ident_list>

<return_stmt> -> RETURN | RETURN <bool_expr> | RETURN STRING
               | RETURN CHAR | RETURN <drone_enum>
```

```
<func_call> -> FUNC_IDENT PAREN_L <param_list> PAREN_R

<param_list> -> | <param> | <param> COMMA <param_list>

<param> -> <bool_expr> | CHAR | STRING | <drone_enum>

// primitive drone functions
<drone_func> -> <get_heading> | <get_altitude> | <get_temperature>
              | <get_time> | <turn> | <climb> | <move>
              | <spray_toggle> | <connect>

<drone_enum> -> <turn_dir> | <climb_dir> | <move_dir>
              | DRONE_ENUM_STOP | <spray_status> | DRONE_IP

<get_heading> ->  DRONE_FUNC_GET_HEADING PAREN_L PAREN_R

<get_altitude> -> DRONE_FUNC_GET_ALTITUDE PAREN_L PAREN_R

<get_temperature> -> DRONE_FUNC_GET_TEMPERATURE PAREN_L PAREN_R

// returns bios time in seconds as an int
<get_time> -> DRONE_FUNC_GET_TIME PAREN_L PAREN_R

<turn> -> DRONE_FUNC_TURN PAREN_L <turn_dir> PAREN_R
        | DRONE_FUNC_TURN PAREN_L IDENT PAREN_R

<turn_dir> -> DRONE_ENUM_LEFT | DRONE_ENUM_RIGHT

<climb> -> DRONE_FUNC_CLIMB PAREN_L <climb_dir> PAREN_R
         | DRONE_FUNC_CLIMB PAREN_L DRONE_ENUM_STOP PAREN_R
         | DRONE_FUNC_CLIMB PAREN_L IDENT PAREN_R

<climb_dir> -> DRONE_ENUM_UP | DRONE_ENUM_DOWN

<move> -> DRONE_FUNC_MOVE PAREN_L <move_dir> PAREN_R
        | DRONE_FUNC_MOVE PAREN_L DRONE_ENUM_STOP PAREN_R
        | DRONE_FUNC_MOVE PAREN_L IDENT PAREN_R

<move_dir> -> DRONE_ENUM_FORWARD | DRONE_ENUM_BACKWARD

<spray_toggle> ->
  DRONE_FUNC_TOGGLE_SPRAY PAREN_L <spray_status> PAREN_R
| DRONE_FUNC_TOGGLE_SPRAY PAREN_L IDENT PAREN_R

<spray_status> -> DRONE_ENUM_ON | DRONE_ENUM_OFF

<connect> -> DRONE_FUNC_CONNECT_TO PAREN_L DRONE_IP PAREN_R
           | DRONE_FUNC_CONNECT_TO PAREN_L IDENT PAREN_R
```

Explanation for Each Language Construct

Initial Definitions

<start> is the start variable, which returns a list of statements with <stmts>. <stmts> simply defines a list of <stmt> instances separated with semicolons, like C-family languages. <stmt> contains statements such as loops, function calls, assignment statements, return statements etc..

<stmts_block> contains one or more statements in it between curly brackets to make the interpretation of statement blocks easier. It is used to make the use of curly brackets mandatory in <for_loop>, <while_loop>, <conditional_stmt> and <func_def> so that the code is easier to read.

<conditional_stmt> is designed to generate any valid combination of if, elseif and else. It may derive if (<bool_expr>) <stmts_block> <elses_list>. <elses_list> derives else (<bool_expr>) <stmts> | elseif (<bool_expr>) <stmts_block> <elses_list>.

<declare> is used to declare a variable with a <type>. <type> can be int, real, bool, string, char or drone_enum. The declaration statement can just declare the variable, or declare and initialize with a given value. The details of that value are described in more detail in <assign>.

<assign> is used to define assignment statements. In RIO, a variable can be assigned either an arithmetic or boolean expression, a string, a char, a drone enum or the value of another variable.

<ident> defines an identifier as any alphanumeric string with the condition that the first character cannot be a digit.

<string> defines strings. A string in RIO is a sequence of characters (except quotation marks) that is defined by <string_chars> put between quotation marks. <comment> similarly defines comments. A comment in RIO begins with a "#" and contains a sequence of characters that lasts until the end of the line. Unlike strings, comments can contain quotation marks and even other hashtags. <alphabetic>, <digit>, <hashtag>, <quotation_mark>, and <punc> contains non-terminal symbols.

<unsigned> can contain one or more <digit>, representing unsigned numbers.

<integer> can be formed by <unsigned>, -<unsigned> or +<unsigned>. It defines unsigned and signed numbers.

<real> defines real numbers which are defined as <integer>.<unsigned>.

Boolean and Arithmetic Expressions

RIO allows conversion between boolean and int/real numbers: 0 is considered false, while all other int or real values are considered true; and false is converted to 0 and true is converted to 1 in arithmetic expressions. This means that the boolean and arithmetic expressions are defined such that arithmetic expressions are a subset of boolean expressions, so in BNF they are defined in one long chain.

Arithmetic expressions are defined according to four different degrees of priorities of operations. <exponent> has the second highest priority, and defines the exponent operation.
<arith_term> has a lower priority than <exponent> and defines multiplication and division operations. <arith_expr> has the lowest priority, and defines addition and subtraction.

<factor> defines the highest priority operations, which are evaluating expressions in parentheses, and fetching the values of variables, function calls and literal numbers. Because of boolean-int/real conversion <factor> derives <ident> | <int> | <real> | <func_call> | <drone_func> | (<bool_expr>). The last one is included in <factor> because including it in <bool_term> (highest precedence bool rule) would mean factor would need to contain (<arith_term>), which would cause a conflict. As it is now, <bool_expr> can be reduced to <arith_expr>, which means the factor can be reduced to y(<arith_expr>) as expected.

Boolean expressions are similarly evaluated in a series of rules to account for the precedence of different boolean operators. <bool_term> has the highest priority and is reduced to <bool> (true or false) or <arith_expr> (which can be further reduced into function calls and identifiers as well, as we would expect in a boolean expression). <logic_not> has the second highest priority among boolean rules, and is used for the “not” logic operator. <logic_and> comes next and implements “and”. <logic_or> comes afterwards to implement “or”. Then, <comparison_expr> is used to implement “<”, “<=”, “>=” and “>” operators. Finally, at the lowest precedence is <bool_expr>, which implements “==” and “=/=” operators.

Loops

There are 2 possible loop expressions in the grammar. First, <for_loop> is written by for (<for_cond>) <stmt>. <for_cond> derives <declare>; <bool_expr>; <assign> or <assign> ; <bool_expr> ; <assign>, the loop's length can be determined by iteration or any other rule depending on assignments. In the first one a new variable can be initialized that can be used in the for loop block. The initial assignment is the first assignment of the condition, the second assignment updates the variable in the boolean expression. Until <bool_expr> gives false output, the loop continues.

Second is <while_loop>, it can be called by while (<bool_expr>) <stmt>. If <bool_expr> is true like in the <for_loop>, the program goes into the loop until it outputs false.

Input/Output Statements

'$' sign is used to separate any input/output statements in the program. <input> derives $input(<ident>) where <ident> can be defined and assigned according to the value that is entered in the console. $input() takes the input as String, then using some other helper functions it will convert it to other types.

<output> derives $output(<bool_expr>), $output(<string>) or $output(<char>). These functions give their parameters as outputs to the console.

General Function Definitions

<func_def> derives func <func_ident>(<ident_list>) <type> {<stmt_block>} | func <func_ident>(<ident_list>) {<stmt_block>} | main <stmts_block> where func keyword aims to attract the reader and writers' attention.

main <stmts_block> is where the main program is executed in the code.

<func_ident> is different from <ident> in that it always starts with '_' to make the function identifiers and calls demonstrative in the code. <ident_list> can consist of one or more <ident> in it, and is used in function signatures.

<type> defines the functions' return type where the typeless functions do not need a return type after their execution (essentially "void" in C-family languages).

<func_block> contains <stmts_block> that can define a function. The <stmts_block> can also derive other <stmts_block>s and <return_stmt>. <return_stmt> can also be used before the <stmts_block> to end the function definition earlier than it is anticipated.

In a <return_stmt>; <bool_expr>, <string>, <char> and <drone_enum> can be returned as the value, or it can be only return;. The call of any user-defined function consists of <func_ident>(<param_list>);. <param_list> can contain one or more <param> in it. These <param> points to <bool_expr>, <string>, <char> or <drone_enum>. So, while calling any function, besides any identifier expressions and strings are also okay to calling the function depending on which type of identifiers used in the function definition.

Special Drone Functions

<drone_func> contains all functions that are related to the drone's controls, which was then added in <stmt> so that these functions can be called as statements. These functions begin with "&" to indicate they are functions related to drone controls. <drone_enum> includes parameters and categories of enums that can only be used with the functions in <drone_func>.

<get_heading>, <get_altitude>, <get_temperature> and <get_time> define functions that return values of their related hardware which are compass, barometer, thermometer, and internal clock, respectively.

<turn> defines a function that takes a <turn_dir> enum (which can be either LEFT or RIGHT) or a variable containing a member of that enum, and rotates the drone by 1° in the corresponding direction. Since there may be other considerations for determining direction of rotation, <ident> enum can be used. For the same reason, it is available for usage in &climb, &move, &toggleSpray and &connectTo functions.

<climb> expressions are created to control the drone's altitude by using UP and DOWN. In <climb> expressions consist 3 types of possible parameters for &climb() function, which are <climb_dir>, STOP and <ident>. Since climbing is a continuous move, STOP term is defined to stop an earlier climbing which can be initiated by &climb(<climb_dir>) and &climb(<ident>).

<move> expressions are created to control the drone's movements by using FORWARD and BACKWARD. In <move> expressions consist 3 types of possible parameters for &move() function, which are <move_dir>, STOP and <ident>. Since moving is a continuous move, STOP term is defined to stop an earlier movement which can be initiated by &move(<move_dir>) and &move(<ident>).

<spray_toggle> contains 2 functions in its &toggleSpray(<spray_status>) and &toggleSpray(<ident>. <spray_status> derives ON and OFF.

<connect> functions are &connectTo(<ip>) and &connectTo(<ident>). <ip> consists of a possible IPv4 address.

Non-terminal and Terminal Symbol Examples in RIO

```
<bool_term> -> BOOL | <arith_expr>
```

- BOOL is a token and thus, it is a terminal symbol whereas <bool_term> is an abstraction and non-terminal symbol.

```
<comparison_expr> -> <comparison_first>
          | <comparison_first> SMALLER_THAN_OP <comparison_second>
          | <comparison_first> SMALLER_OR_EQ_OP
<comparison_second>
          | <comparison_first> GREATER_THAN_OP <comparison_second>
          | <comparison_first> GREATER_OR_EQ_OP
<comparison_second>
```

- <comparison_first> and <comparison_second> are the non-terminal symbols for this rule, because they are abstractions for another rule, but SMALLER_THAN_OP, SMALLER_OR_EQ_OP, GREATER_THAN_OP and GREATER_OR_EQ_OP are terminal symbols for this production.

```
<func_def> ->
 FUNC_DEF FUNC_IDENT PAREN_L <ident_list> PAREN_R TYPE
<stmts_block>
| FUNC_DEF FUNC_IDENT PAREN_L <ident_list> PAREN_R <stmts_block>
| MAIN PAREN_L PAREN_R <stmts_block>

<ident_list> -> | TYPE IDENT | TYPE IDENT COMMA <ident_list>
<stmts_block> -> CURLY_L <stmts> CURLY_R
```

- In this example <ident_list> and <stmts_block> are non-terminal symbols, because they derive other non-terminal and terminal symbols where FUNC_DEF, FUNC_IDENT, PAREN_L, PAREN_R, TYPE, IDENT, COMMA, CURLY_L and CURLY_R are all tokens.

Non-Trivial and Trivial Tokens in RIO

Non-trivial tokens:

MAIN: main is a special function, defined without “func” keyword or type, so we read it as a separate token
TYPE: “int”, “real”, “bool”, “string”, “char” or “drone_enum”
FUNC_DEF: “func”, used to define new functions
FUNC_IDENT: “_” followed by an alphanumeric string (except first character can’t be a digit)
INPUT: “$input”, identifier of input function
OUTPUT: “$output”, identifier of output (print) function
DRONE_FUNC_GET_HEADING, DRONE_FUNC_GET_TEMPERATURE ... : primitive drone functions’ identifiers (which all begin with “&”)
DRONE_ENUM_LEFT, DRONE_ENUM_RIGHT ... : drone enums, used as parameters in primitive drone functions
DRONE_IP: ip, defined as 4 integers with 3 “.”s in between them
IDENT: any permutation of alphanumeric characters (except first character can’t be a digit)
CHAR: any character put between two single quotation marks (includes ‘\n’)
STRING: any permutation of characters (except double quotation mark) put between double quotation marks
INT: any integer, positive integers can have + in front or not
REAL: any real number, positive real numbers can have + in front or not
BOOL: “true” or “false”
COMMENT: any permutation of characters beginning with “#”

Trivial tokens:

RETURN: “return” (used in return statement)
FOR: “for”
WHILE: “while”
IF: “if”
ELSE_IF: “elseif”
ASSIGNMENT_OP: “=”
SEMICOLON: “;”
COMMA: “,”
PLUS_OP: “+”
MINUS_OP: “-”
MULT_OP: “*”
DIV_OP: “/”

EXP_OP: “^”
NOT_OP: “not”
AND_OP: “and”
OR_OP: “or”
EQUAL_OP: “==”
NOT_EQUAL_OP: “=/=”
SMALLER_THAN_OP: “<”
GREATER_THAN_OP: “>”
SMALLER_OR_EQ_OP: “<=”
GREATER_OR_EQ_OP: “>=”
CURLY_L: “{“
CURLY_R: “}”
PAREN_L: “(“
PAREN_R: “)”

Evaluation of the Programming Language

Evaluating a programming language has three aspects and those are in terms of readability, writability and reliability.

- **Readability of RIO**

  In the programming language RIO, different functions of this language can be distinguished by first characters of function names. Functions that are related to the drone’s controls start with “&,” input and output (print) functions start with “$,” and other functions start with “_”. This way users can easily realize different usages of functions, also using such punctuations for functions lets users differentiate them from variable identifiers. Enums are used to call drone functions rather than unrecognizable numbers. RIO also requires if-elseif-else, for, while, and function definition blocks to have their “executable” parts between curly brackets to form a statement block, even if it only consists of a single statement. Moreover, RIO is a simple programming language, operator overloading is minimal. It is aimed to have an orthogonal language, when an instruction is executed, nothing but that instruction happens. RIO contains all the necessary data types such as number (including real numbers and integers), boolean and string. There are special words for loops (`for, while`) as a syntactic design property. All these specifications make RIO a readable language.

- **Writability of RIO**

  Programming language RIO has predefined functions. At first it might look like it has some writability issues because throughout the implementation of grammar of RIO, readability has been the main concern. However, there are not many side effects on the

writability of the programming language. Adding one character in front of a function depending on its name might seem like poor writability, though it reminds the user what that function is going to be used for. Drone functions can be called by using enums rather than arbitrary number or boolean parameters, and these enums can be put into variables as well which can be later used in these functions. Both of these factors improve writability for drone functions. The fact that boolean expressions can be converted into integers (0 for false, 1 for true statements) also improves the writability a bit as the use of boolean expressions in arithmetic expressions can sometimes allow the user to write shorter code without conditional statements.

- **Reliability of RIO**

  RIO has some types which improve the reliability of the language. However, boolean expressions can be used in arithmetic expressions and arithmetic expressions can be used in boolean expressions, which reduces the reliability in some cases. Since this programming language is otherwise quite simple, there should not be any place for aliasing.