BILKENT UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

CS315-PROGRAMMING LANGUAGES

FALL 2021

GROUP 14 - LEXRONE

İdil YILMAZ - 21703556 - Section 02
Doğukan Ertunga Kurnaz - 21702331 - Section 02
Turgut Alp Edis - 21702587 - Section 02

BNF

1. ***Variable identifiers & Assignment Operators & Precedence, Associativity of the Operators & Expressions (Arithmetic, Relational, Boolean, Their Combination)***

<math_op_add> ::= +

<math_op_sub> ::= -

<math_op_mult> ::= *

<math_op_mod> ::= %

<math_op_div> ::= /

<math_op> ::= <math_op_add> | <math_op_mult> | <math_op_sub> | <math_op_div>
|<math_op_mod>

<assign> ::= =

<relation> ::= > | < | >= | <=

<dot> ::= .

<and> ::= &&

<or> ::= ||

<not_eq> ::= !=

<eq> ::= ==

<end_line> ::= ;

<new_line> ::= \n

<sign> ::= + | -

<underscore_ch> ::= _

<space_char> ::=

<tab> ::= “\t”

<bool> ::= true | false

<comment_sign> ::= ***

<left_brace> ::= {

<right_brace> ::= }

<colon> ::= :

<left_parenthesis> ::= (

<right_parenthesis> ::=)

<char> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|R|S|T|U|V|Y|Z|Q|X
| a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|r|s|t|u|v|y|z|x|q

<non_digit_char> ::= <char> | <underscore_ch>

<digit> ::= 0|1|2|3|4|5|6|7|8|9

<int> ::= <sign>?(<digit> | <int><digit>)

<float> ::= <int><dot><int> | <dot><int>

<assign_op> ::= <math_op><assign> | <assign>

<alpha_num> ::= (<non_digit_char> | <digit>) | <alpha_num> (<non_digit_char>
| <digit>)

<identifier> ::= <char>(<non_digit_char> | <digit>)

<comment> ::= <comment_sign><identifier><comment_sign>

<string> ::= '(<identifier>|<dot>|<math_op>|<space_char>)'

<string_expr> ::= <identifier> <assign> <string>

<int_expr> ::= <identifier> <assign> <int>

<bool_expr> ::= <identifier> <assign> <bool>

<float_expr> ::= <identifier> <assign> <float>

<math_expr> ::= <math_expr> <math_op_add> <math_term> | <math_term>

<math_term> ::= <math_term><math_op_mulp><math_factor> | <math_factor>

<math_factor> ::= (<math_expr>) | <int>

<relational_expr> ::= <identifier> <relation> <identifier> | <int> <relation> <int>
| <float> <relation> <float> | <identifier> <eq> <identifier>
| <identifier> <not_eq> <identifier> | <int> <eq> <int>
| <int> <not_eq> <int> | <float> <eq> <float>
| <float> <not_eq> <float>

<expr> ::= <math_expr> | <string_expr> | <int_expr> | <bool_expr> | <float_expr>

<types> ::= <identifier> | <int> | <float> | <string> | <bool>

<parameter> ::= <types> <identifier> | <types> <identifier> <parameter> |

2. *Statements & Conditions & Functions*

<program> ::= begin<stmts>end
<stmts> ::= <stmt> | <stmts><stmt>
<stmt> ::= <expr><end_line> | <comment> | <loop_types> | <func_call> | <condt_stmt>
<block_stmt> ::= <stmts> return <types> | <stmts>
<if_stmt> ::= if <left_parenthesis> <condition> <right_parenthesis> <colon>
<block_stmt> (else <colon> <block_stmt> endif | endif)

<condition> ::= <relational_expr> | <func_call>
<func_call> ::= <identifier> <left_parenthesis> <parameter> <right_parenthesis>
<end_line>

<funct_def> ::= func <types> <identifier> <left_parenthesis> <parameter>
<right_parenthesis> <colon> <block_stmts> endfunc

3. *Loops*

<while> ::= while <left_parenthesis> <condition> <right_parenthesis> <colon>
<block_stmt> endwhile
<loop_types> ::= <while> | <for>
<for> ::= for <left_parenthesis> <condition> <right_parenthesis> <colon> <block_stmt>
endfor

4. *Statements for Input / Output*

<input> ::= in

<output> ::= print

5. *Primitive Functions*

<turn_heading> ::= turnLeft() | turnRight()
<altitude> ::= readAltitude()
<heading> ::= readHeading()

<velocity> ::= readVelocity()

<go_vertical> ::= up() | down() | stop()

<go_horizontal> ::= forward() | backward() | stop()

<temp> ::= readTemp()

<nozzle_stat> ::= on() | off()

<wifi_conn>::= connect() | disconnect() | status()

BNF EXPLANATIONS

Types

LexRone has 5 types in order to be used: *int*, *float*, *bool, string* and *char*.

<int>: Defined as any integer that can be used in calculation or identifier's name.

<float>: This type is used for fractional numbers when int type is insufficient.

<bool>: Checks the true/false status of a condition or variable.

<string>: This type is used for defining string values, it can be used as any variable name.

<char> : This type is used for creating identifiers. From the identifiers, we can derive strings.

Explanation of LexRone Language Constructs

<program> ::= begin<stmts>end

This non-terminal is used to represent our program constructed from statements. The statements part is where our code actually goes. This functionality is crucial to represent where our code starts and finishes for the compiler. In our language it starts with "begin" command and stops with "end" command.

<stmts> ::= <stmt> | <stmts><stmt>

This non-terminal is used to represent statements that can be constructed from statement or statements and statement.

<stmt> ::= <expr><end_line> | <comment> | <loop_types> | <func_call> | <condt_stmt>

It is used to create the basic structure of the program by using expressions, comments, loops, conditional statements and function calls.

<block_stmt> ::= <stmts> return <types> | <stmts>

This non-terminal is used to represent the statement block can be constructed from statements with return types or it can be constructed from statements.

<if_stmt> ::= if <left_parenthesis> <condition> <right_parenthesis> <colon> <block_stmt> (else <colon> <block_stmt> endif | endif)

This non-terminal is used to represent the usage of "if statement" in LexRone. In LexRone, we defined if statement without curly braces, we used colon for simplicity. To not get an exception of else without the if condition, we defined the else part as well in here. It is optional though, but for the reliability we advise to use else part of the block as well. For the represent "if statement" block is finished we used the "endif" tag.

<func_call> ::= <identifier> <left_parenthesis> <parameter> <right_parenthesis> <end_line>

This non-terminal is used to call the function which is defined earlier in the code. For the function name, all combinations of the identifier are possible and the defined parameter for the function can be used as well. After the function is called, it returns the type of the function.

<funct_def> ::= func <types> <identifier> <left_parenthesis> <parameter> <right_parenthesis> <colon> <block_stmts> endfunc

This non-terminal is used to represent how the functions should be defined in LexRone. The function definition should start with the “func” keyword then it takes which type it will be. Then, between left and right parenthesis it takes the required parameters to be used in the statement block in the function body. After right parenthesis, we use colon to represent the statements for the function starts. After the block of statements, we define the “endfunc” keyword to represent function definition block’s ends there.

<condition> ::= <relational_expr> | <func_call>

This non-terminal is used to represent how conditions are constructed in LexRone. A condition can be constructed from a relational expression or it can be constructed from a function call. This condition terminal can be used in if statements, for loops and while loops.

<loop_types> ::= <while> | <for>

This non-terminal is used to represent which loop types we have in LexRone. A loop can be a while loop or for loop.

<while> ::= while <left_parenthesis> <condition><right_parenthesis>
<colon><block_stmt> endwhile

This non-terminal is used to represent how while loops are constructed in LexRone. A while loop can be constructed by starting with the “while" keyword then condition inside the parenthesis, after that colon is used to show that while loop block statements are started. After a block of statements we defined an “endwhile” token to represent the while loop is finished here.

<for> ::= for <left_parenthesis> <condition> <right_parenthesis> <colon>
<block_stmt>endfor

This non-terminal is used to represent how “for” loops are constructed in LexRone. A for loop can be constructed by starting with the “for” keyword then condition inside the parenthesis, after that colon is used to show that for loop block statements are started. After a block of statements we defined an “endfor” token to represent the for loop is finished here.

<turn_heading> ::= turnLeft() | turnRight()

This non-terminal states the primary function to adjust the head of the drone. This non-terminal can call 2 functions, turnLeft() reduces the head degree by 1 degree while turnRight() increases the head degree by 1 degree.

<altitude> ::= readAltitude()

This non-terminal states the primary function for reading the current altitude of the drone. It returns the float value of the altitude of the drone. This non-terminal can only call one function for this implementation but we can add more functions in the next iterations.

<heading> ::= readHeading()

This non-terminal states the primary function for reading the current heading position of the drone. It returns the integer value of the heading of the drone. This non-terminal can only call one function for this implementation but we can add more functions in the next iterations.

<velocity> ::= readVelocity()

This non-terminal states the primary function for reading the current velocity of the drone. It returns the float value of the velocity of the drone. This non-terminal can only call one function for this implementation but we can add more functions in the next iterations.

<go_vertical> ::= up() | down() | stop()

This non-terminal states the primary function for giving vertical movement commands to the drone. This non-terminal can call 3 function, it can be up(), down() or stop(). These functions returns the float value of the current speed of the vertical movement. For convenience, we implemented stop() function as well.

<go_horizontal> ::= forward() | backward() | stop()

This non-terminal shows the primary function to adjust the horizontal movement of the drone. It includes forward, backward movement and stop. The forward() and backward() functions return the float value of the current speed of the drone while the stop function terminates the current speed of the drone.

<temp> ::= readTemp()

This non-terminal states the primary function for reading the current outside temperature. It returns the integer value of the current temperature of the outside environment. This non-terminal can only call one function for this implementation but we can add more functions in the next iterations.

<nozzle_stat> ::= on() | off()

This non-terminal states the primary functions to control the nozzle spray system of the drone. The spray can be turned on or off with these two functions, which are on() is used to turn on the spray and off() is used to turn off the spray.

<wifi_conn>::= connect() | disconnect() | status()

This non-terminal states the primary function for connectivity of the drone. It returns the boolean value of the called function to give the status update about the function. This non-terminal can call 3 functions. These are, connect(), disconnect() and status(). The connect() function makes the connection between the wifi and the drone with the credentials already written inside of the drone. The disconnect() function disconnects the established wifi connection. The status() function gives the current status of the connection.

<non_digit_char> ::= <char> | <underscore_ch>

This non_digit_char can can take the char values or underscore char which is “_”

<int> ::= <sign>?(<digit> | <int><digit>)

This integer(<int>) can take digit with its sign(it can be + or -) or if we want to take input as a multiple digit number we should do in recursively as it can be seen <int><digit> part

<float> ::= <int><dot><int> | <dot><int>

This float can take integer following dot with another integer or it can take only dot following an integer(as an example of .5 which demonstrates 0.5)

<assign_op> ::= <math_op><assign> | <assign>

Assignment operator can be used with mathematical operations(addition subtraction multiplication and division) following assign operator(=).

<alpha_num> ::= (<non_digit_char> | <digit>) | <alpha_num> (<non_digit_char> | <digit>)

Alpha_num states that any combination of digits and non-digit characters can be possible.

<identifier> ::= <char>(<non_digit_char> | <digit>)

The identifier is like a variable name in our PL. For example, an identifier has to include a char following a non-digit char which contains an underscore or just one of the digits.

<comment> ::= <comment_sign><identifier><comment_sign>

In order to make comments in our programming language, there have to be 2 comment signs and an identifier which is between these comment signs.(*** …. ***)

<string> ::= '(<identifier>|<dot>|<math_op>|<space_char>)'

The string is one of the types in our PL. it contains an identifier or a dot(.) or one of the math operations or space char(empty)

<string_expr> ::= <identifier> <assign> <string>

String expression includes an identifier following assign and then following a string. The equation a = ‘b’ is an example of string expression in our PL.

<int_expr> ::= <identifier> <assign> <int>

Integer expression contains an identifier following assign and then a new integer which is int. a = 2 as an example of integer expression.

<bool_expr> ::= <identifier> <assign> <bool>

Boolean expression checks if the identifier is true or false according to boolean value.

<float_expr> ::= <identifier> <assign> <float>

Float expression can be used if an identifier is equal to any float number. c = .5 is an example of float expression.

<math_op> ::= <math_op_add> | <math_op_mult> | <math_op_sub>
| <math_op_div> |<math_op_mod>

This non-terminal is defined for the mathematical operations. This non-terminal can call 5 mathematical operation terminal. These operations are; addition, subtraction, multiplication, division and mod operation.

<math_expr> ::= <math_expr> (<math_op_add> | <math_op_sub>) <math_term>
| <math_term>

Math expression can only involve add, subtract operations and the non-terminal math_term because it is used to determine the precedence of the mathematical operations.

<math_term> ::= <math_term>(<math_op_mulp> | <math_op_div>
| <math_op_mod>)<math_factor> | <math_factor>

This non-terminal states the precedence of the math operations with only involving the multiplication, division, modulus operations and the non-terminal math_factor.

<math_factor> ::= (<math_expr>) | <int>

This non-terminal states the highest precedence mathematical calculation which is the brackets.

<relational_expr> ::= <identifier> <relation> <identifier> | <int> <relation> <int>
| <float> <relation> <float> | <identifier> <eq> <identifier>
| <identifier> <not_eq> <identifier> | <int> <eq> <int>
| <int> <not_eq> <int> | <float> <eq> <float>
| <float> <not_eq> <float>

This non-terminal is used to state that any type can be involved in a relation with the same type. It always returns a boolean that indicates if the relation is true or not and it is used with the conditional statements and loops.

<expr> ::= <math_expr> | <string_expr> | <int_expr> | <bool_expr> | <float_expr>

The expression defines all possible fundamental expressions for this language to indicate that all of these expressions can define the program by themselves.

<types> ::= <int> | <float> | <string> | <bool> | <char>

This non-terminal defines all types of variables which can be used in this language. Also, it can be used in the functions to determine the type of the return value and parameter.

<parameter> ::= <types> <identifier> | <types> <identifier> <parameter>

The parameter is used to define the input values of the functions. It is created with a type followed by an identifier to indicate the type and name of the input that can be used inside the function.

Explanation of LexRone Language Tokens

<math_op_add> ::= +

This allows the user to perform a mathematical addition.

<math_op_sub> ::= -

This allows the user to perform a mathematical subtraction.

**<math_op_mult> ::= *

This allows the user to perform a mathematical multiplication.

<math_op_mod> ::= %

This allows the user to perform modular arithmetic operations.

<math_op_div> ::= /

This allows the user to perform a mathematical division.

<assign> ::= =

This token allows the user to perform an equality.

<relation> ::= > | < | >= | <=

This token allows the user to make a relationship between two or more variables.

<dot> ::= .

This “dot” token is used for both float (for fractional numbers) and string variables.(to the end of sentences or other tasks.)

<and> ::= &&

This “and” token is used in relation, conditional statements and loops (such as if, while or for).

<or> ::= ||

This “or” token is used inside the relation, conditional statements and loops (such as if, while or for).

<not_eq> ::= !=

This token demonstrates that there is no equality between variables.

<eq> ::= ==

This token demonstrates that there is an equality between variables.

<end_line> ::= ;

This end line is used for a better understanding of compilers after variables and functions.

<new_line> ::= \n

This new line token is used to switch a new line.

<sign> ::= + | -

This sign token is used for an integer or float variable that can be a negative or positive number.

<underscore_ch> ::= _

This underscore token is used to create the identifier and string.

<space_char> ::=

This space char is used as space to avoid confusion.

<tab> ::= “\t”

This tab is used for cleaner code.

It is used immediately after conditions such as if, while, for, so that there is no confusion inside these conditions.

<bool> ::= true | false

This bool token demonstrates whether a condition, variable or identifier is true or false.

<comment_sign> ::= *

To comment in the code, a comment is written between two comment signs.

<left_brace> ::= {

Left brace token is a reserved token for next version of our PL.

<right_brace> ::= }

Right brace token is a reserved token for next version of our PL.

<colon> ::= :

It is used immediately after conditions such as if, while, for, so that there is no confusion inside these conditions.

<left_parenthesis> ::=

Used in loop conditions, if-else statements, and functions

<right_parenthesis> ::=)

Used in loop conditions, if-else statements, and functions

<input> ::= in

This token is used to receive input from the user.

<output> ::= print

If the user wants to print output on the console the user can use output token

<char> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|R|S|T|U|V|Y|Z|Q|X
| a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|r|s|t|u|v|y|z|x|q

Char is the type that defines all alphabetic characters for this language.

<digit> ::= 0|1|2|3|4|5|6|7|8|9

Digit defines the all digit numbers for this language.

Readability

LexRone is easy to understand in terms of readability. Because we construct each component from the basics, there is no collision between any language component. In the design part of LexRone we considered the mostly used cases they may occur in a drone controlling program. With the experience of our team in integrated software development, LexRone's components are specially selected from the common languages used in the world. LexRone developed with ease of use and readability in mind. It was developed by analyzing the best aspects of the languages that are used frequently in the world. We paid attention to visual intensity in code and try to minimize complexity for easy understanding of code for target users. That's how we developed LexRone as a best fit for our project topic.

Writability

LexRone is easy to write as well. The detailed BNF of the language is written in this report. Developers can use this BNF description as a source. LexRone has sufficient grammar for almost every scope. When we designed LexRone, we aimed to design a language that is easy to remember when writing. That's how developers' warm-up time with our language is less than the competitor languages. In order not to confuse the user with too many constructors or the combination of these structures, we try not to give the user a lot of free space. In addition, because of the lack of unnecessary complex values(simplicity) and since we try to prevent misuse of some features, the LexRone is very similar to writing in English.

Reliability

While designing LexRone, another crucial point we discussed about it is reliability. Because we design a special language for drones, it should be precise and reliable. With the consideration that the programs written from our language will be used by enterprise users, we designed LexRone. In software languages, the error probability is important but the behavior when the program throws an error is more important. LexRone is designed to handle this situation flawlessly. When an error happens LexRone handles this situation to minimize the effect of error on the program.