CS315 - Programming Languages

PROJECT 2 REPORT

Logismik Programming Language

Section 1 - Group 10

Alperen Erdoğan - 21703903

Hasan Alp Caferoğlu - 22203991

Mehmet Eren Balasar - 22001954

02.04.2023

BNF Description for Logismik

1. Initial Program

```
<program>          ::= begin <sc> <statement_list> end <sc>
<statement_list>   ::= <statement> | <statement><statement_list>
<statement>        ::= <declaration> | <assignment>
                     | <conditional_statement> | <loop_statement>
                     | <output_statement> | <input_statement>
                     | <func_call> <sc>
                     | <comment>
                     | <break_statement>
<break_statement> ::= break <sc>
```

2. Declaration and Assignment

```
<declaration>       ::= bool <variable_list> <sc>
                      | bool <bool_assignment>
                      | bool <lsb> <rsb> <variable_name> <sc>
                      | bool <lsb> <rsb> <array_assignment>
                      | <function_declaration>
<assignment>        ::= <bool_assignment> | <array_assignment>

<bool_assignment>   ::= <variable_name> <ao> <bool_expression> <sc>

<array_assignment>  ::= <variable_name> <ao> <array> <sc>
```

3. Expressions

```
<bool_expressions>    ::= <bool_expression>
                        | <bool_expression> <comma>
                        <bool_expressions>
<bool_expression>     ::= <bool_term> | <bool_term> <double_imp_op>
                         <bool_expression> | <bool_term> <equal_op>
```

```
                    <bool_expression> | <bool_term>
                    <not_equal_op> <bool_expression>

<bool_term>         ::= <bool_factor> | <bool_factor> <imp_op>
                        <bool_term>

<bool_factor>       ::= <bool_secondary>
                        | <bool_secondary> <or_op> <bool_factor>

<bool_secondary>    ::= <bool_primary>
                        | <bool_primary> <and_op> <bool_secondary>

<bool_primary>      ::= <bool_literal>
                        | <variable_name>
                        | <not_op> <bool_primary>
                        | <lp> <bool_expression> <rp>
                        | <func_call>

<array>             ::= <lsb> <bool_expressions> <rsb>
```

4. Conditional Statements

```
<conditional_statement> ::= <if_statement>
                          | <if_statement> <else_statement>

<if_statement>          ::= if <bool_expression> <lcb>
                            <statement_list> <rcb>
                            | <if_statement> else if
                            <bool_expression> <lcb>
                            <statement_list> <rcb>

<else_statement>        ::= else <lcb> <statement_list> <rcb>
```

5. Loops

<loop_statement> ::= <while_loop> | <foreach_loop>

<while_loop> ::= while <bool_expression> <lcb>
<statement_list> <rcb>

<foreach_loop> ::= for each <variable_name> in <variable_name>
<lcb> <statement_list> <rcb>
| for each <variable_name> in <array> <lcb>
<statement_list> <rcb>

6. Function

<function_declaration>::= <bool_function>
| <void_function>
| <array_function>

<bool_function> ::= <function_identifier> bool
<function_signature> <lcb>
<statement_list> return
<boolean_expression> <sc> <rcb>

<array_function> ::= <function_identifier> bool <lsb> <rsb>
<function_signature> <lcb>
<statement_list> return <array> <sc> <rcb>
| <function_identifier> bool <lsb> <rsb>
<function_signature> <lcb>
<statement_list> return <variable_name>
<sc> <rcb>

<void_function> ::= <function_identifier> void
<function_signature> <lcb>
<statement_list> <rcb>

<function_signature> ::= <function_name> <lp> <variable_list> <rp>

<func_call> ::= <function_name> <lp> <bool_expressions> <rp>

<variable_list> ::= <variable_name>
| <variable_name> <comma> <variable_list>

7. Input-Output Statements

<input_statement> ::= input <lp> <variable_list> <rp> <sc>

<output_statement> ::= print <lp> <print_content> <rp> <sc>

<print_content> ::= <string_constant> | <bool_expression>
|<string_constant><comma><print_content>
|<bool_expression><comma><print_content>

8. Comment

<comment> ::= <hashtag> <string_characters> <nl>

9. Symbols

<or_op> ::= or | <pipe><pipe>
<and_op> ::= and | <ampersand><ampersand>
<not_op> ::= not | <exclamation_mark>
<imp_op> ::= -> | implies
<double_imp_op> ::= <-> | double-implies
<equal_op> ::= ==
<not_equal_op> ::= !=
<bool_literal> ::= <true> | <false>
<true> ::= true

<false> ::= false

<variable_name> ::= <letter> | <variable_name> <letter> |
<variable_name> <digit>
| <variable_name> <under_score>
<variable_name>

<function_name> ::= <variable_name>

<function_identifier>::= func

<string_constant> ::= <quote> <string-characters> <quote>

<string_characters> ::= <character>
| <character><string_characters>

<character> ::= <letter> | <digit> | <symbol>

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

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

<symbol> ::= <sc> | <ao> | <lp> | <rp> | <lcb> | <rcb>
| < <lsb> | <rsb> | <quote> | <hashtag> |
<colon> | <comma> | <exclamation_mark> |
<percent> | <ampersand> | <dollor_sign> |
<star> | <plus_sign> | <hyphen> |
<under_score> | <slash> | <backslash> |
<left_bracket> | <right_bracket> |
<question_mark> | <at_sign> | <pipe> |
<caret> | <tilde> | <period>

<sc> ::= ;

<ao> ::= =

<lp> ::= (

<rp> ::=)

<nl> ::= \n

<lcb> ::= {

<rcb> ::= }

<lsb> ::= [

<rsb> ::=]

<quote> ::= "

<colon> ::= :

<hashtag> ::= #

<comma> ::= ,

<exclamation_mark> ::= !

<percent> ::= %

<ampersand> ::= &

<dollor_sign> ::= $

<star> ::= *

<plus_sign> ::= +

<hyphen> ::= -

<under_score> ::= _

<slash> ::= /

<backslash> ::= \

<left_bracket> ::= <

<right_bracket> ::= >

<question_mark> ::= ?

<at_sign> ::= @

<pipe> ::= |

<caret> ::= ^

<tilde> ::= ~

<period> ::= .

BNF Explanation

- <program>: The starting point of any Logismik program. Logismik has identifiers that denote the beginning and end of the program. The keyword "begin" with a semicolon after that indicates the starting point of the program. At the end of the program, the keyword "end" with a semicolon is required for the termination of the program. Without these keywords, the program will throw a syntax error.
- <statement-list>: A statement or list of statements that make up a program.
- <statement>: A single statement in the program. This single statement is a command which composes the program. It can be one of the following; declaration, an assignment, a conditional statement, a loop statement, an input or output statement, a function definition, a comment, or an end-statement.
- <break_statement>: This is used to end loops and functions. It should be a break statement followed by a Boolean expression and a semicolon (;).
- <declaration>: It is a statement that declares a variable with the "bool" keyword followed by a <variable-name>. Additionally, the "bool[ ]" keyword followed by <variable-name> allows programmers to declare an array in the Logismik language. This declaration
- <assignment>: A statement that assigns a value to a variable.

The Logismik language is limited to the Boolean type. In other words, if a programmer writes an integer anywhere in the program except between the double quotation mark, which would be a string constant, then the parser will print out an error message indicating the line number of the source code that contains the integer.

```
 _____________________________________________
| bool c;                                     |
| C = 234;   # parser would give an error     |
| justVarName;  # parser would give an error  |
| 2340;      #parser would give an error      |
| bool arr[];  # define array named as arr    |
| arr = [ a, b, 454] # parser would giver error |
 _____________________________________________
```

- `<bool_assignment>`: A statement that assigns a Boolean value to a variable.
- `<array_assignment>`: This is used to assign a list of Boolean values to an array of variables.The purpose of this is to store a collection of Boolean values in a variable.
- `<bool_expressions>`: This rule allows the user to write a single boolean expression or several boolean expressions separated by comma.
- `<bool_expression>`: This is a Boolean expression that can be composed of `<bool_term>` and `<bool-term>` connected by double implication (<->) to another `<bool_expression>`. The double implication has the lowest precedence. Also `<equal_op>` and `<not_equal_op>` are also used in this level.
- `<bool_term>` : A Boolean expression that can be composed of `<bool_factor>` and `<bool_factor>` connected by the implication (->) to another `<bool_term>`. The implies operation has precedence over double implies.
- `<bool_factor>` : A Boolean expression that can be composed of `<bool_secondary>` and `<bool_secondary>` connected with the or operator (||) to another `<bool_factor>`. The or operator has precedence over the implies operator.
- `<bool_secondary>` : A Boolean expression that can be composed of `<bool_primary>` and `<bool_primary>` connected with the and operator (&&) to another `<bool_secondary>` variable. The and operator has precedence over the or operator.
- `<bool_primary>` : This variable can be either a Boolean literal, a variable, a not operator (!), a Boolean expression enclosed in parentheses, or a function call. The not operator has higher precedence than other logical operators.

Operators	Operator Name	Precedence
not (!)	Not	1
and (&&)	And	2
or (||)	Or	3
->	Implication	4
<->, ==, !=	Double-implication, equal, not equal	5

Table 1: **The precedence of logical operators in Logismik.**

- `<array>`: The array construct is used in order to store a collection of Boolean values and expressions which are ultimately equal to Boolean values.
- <conditional_statement>: Conditional statement construct is a generalised rule for different conditional statements. It can be either an if statement or an if statement combined with an else statement.
- `<if_statement>`: If statement is consistent with common programming language conventions. It checks a boolean expression, which might be an expression, boolean constant, variable name or function call, then executes a block of code if the checked boolean expression is true.
- `<else_statement>`: This else statement is used with if statement, and executes a block of code if the checked boolean expression in the matched if statement is false.
- `<loop_statement>`: This non-terminal represents all the loop structures in Logismik. Namely, “while” and “foreach” loops.
- `<while_loop>`: Non-terminal that denotes a simple well known while loop. Statement list goes inside the curly braces and executes code block in it if the condition of the while loop is true.
- `<foreach_loop>`: Since Logismik does not have integers to traverse arrays and get values corresponding to indexes, a foreach loop is essential. After the “for each” lexeme a variable name must be given to store the current array element. After the “in” lexeme, the variable name for the array to be traversed or an actual array object must be given. Statement list written inside the curly braces is executed for each element of the array.
- `<function_declaration>`: All the bool, array and void function declarations are represented with this non-terminal. With `<function_declaration>` non-terminal, users can indicate and see the return type of the function that improves the reliability and readability of the Logismik language.
- `<bool_function>`: This non-terminal is used when the function that will be declared returns a boolean value. With `<bool_function>` non-terminal, we can directly understand what type of value will be returned by function. That has a positive contribution to the readability and reliability of the language.
- `<array_function>`: This non-terminal is used when the function that will be declared returns an array. With `<array_function>` non-terminal, return type of the function can be directly understood which increases the readability and reliability of the Logismik language.

- <void_function>: This non-terminal is used when the function that will be declared does not return any value.
- <function_signature>: Denotes the name and parameters of a function that will be declared.
- <func_call>: Conventional function call. By using the name of a function that has been declared before, along with boolean expressions separated by commas inside parentheses, users can call a function. Note that boolean expressions include <variable_name>, meaning that functions can be called with both variables and boolean expressions. Logismik has only two types: arrays and boolean values. Arrays are passed to functions as references. Any other parameter is passed by value.
- <variable_list>: This non-terminal represents a list of variables that are separated by comma.
- <input_statement>: Input statements are used to get boolean values from the user. The “input()” lexeme is used for this purpose. Inside the parentheses, there must be a <variable_list>. Input boolean values are registered into these variables in order. Users must type the values with commas in between. Any other data type for input will generate an error. Example (>>> means console input):

```
_____________________________________________
| input(x,y,z);                             |
| >>> true, true, false                     |
| Result: x = true, y = true, z = false)    |
_____________________________________________
```

- <output_statement>: When users want to print something onto the console, they will have to use the “print()” lexeme. Inside the parentheses, there must be a <print_content>. Automatically inserts a new line at the end of the print content.
- <print_content>: It can be composed of string constants, variables, boolean expressions and their combinations. For combination of string constants and variables, comma is used between them. By this way, users can construct output sentences easily without repeating print operation which has a positive contribution on writability of the language.
- <comment>: Comments in Logismik start with “#” and end with a new line. All of the string characters can be used in a comment. Multi-line comments are not supported.

- `<or_op>`: Logical “OR” operator. In Logismik, users can use both the “or” lexeme and the “||” lexeme to access the functionality of “or”.
- `<and_op>`: Logical “AND” operator. In Logismik, users can use both the “and” lexeme and the “&&” lexeme to access the functionality of “and”.
- `<not_op>`: Represents logical “not”. In Logismik, users can use both the “not” lexeme and the “!” lexeme to access the functionality of “not”. It has the highest precedence among all operators.
- `<imp_op>`: Represents the implication operation in logic. In Logismik, users can use both the “implies” lexeme and the “->” lexeme to access the functionality of implication.
- `<double_imp_op>`: Logical XNOR operator, mathematicians call it “if and only if”. In Logismik, users can use both the “double-implies” lexeme and the “<->” lexeme to access the functionality of double implication.
- <equal_op>: This operation is used to check whether the left-hand side of the operator and the right-hand side of the operator is equal. If the left-hand side and right-hand side of the operator are equal, then it returns true.
- <not_equal_op>: This operation is used to check whether the left-hand side of the operator and the right-hand side of the operator is equal or not. If the left-hand side and right-hand side of the operator are not equal, then it returns true.
- `<bool_literal>`: It represents both of the two boolean literal values “true” and “false”.
- `<true>`: Binary “1”, or logical “true”. It is represented as a non-terminal to increase modifiability of the BNF document.
- `<false>`: Binary “0”, or logical “false”. It is represented as a non-terminal to increase modifiability of the BNF document.
- `<variable_name>`: This non-terminal is used to represent the names given to variables that label the memory locations storing single boolean values or storing the start addresses of boolean arrays consisting of multiple boolean values. It is allowed to use letters, digits and underscore in variable names, but with conditions: Digits cannot be the first character of a variable name, and underscore cannot be the first or the last character of a variable name.
- `<function_name>`: Very similar to a `<variable_name>` this non-terminal represents names that users can give to functions.

- <function_identifier>: This non-terminal indicates that there will be a function declaration.
- <string_constant>: If you enclose a set of <string_characters>, it will become a constant string value. In Logismik, storing strings is not supported. These values are only used as constant one-time values for input and output.
- <string_characters>: All of the characters that are allowed in a string.
- <symbol>: This non-terminal is used to represent almost all of the ASCII symbols other than digits and letters.
- We have used abstractions for every symbol to increase the readability and modifiability of the BNF document.

Non-trivial Tokens and Writability - Readability

In defining Logismik, readability, writability, and reliability were considered. For instance, the use of explicit variable declaration and type specification improves program reliability by reducing the risk of type errors, even though Logismik has only one type of variable.

The use of case-sensitive identifiers improves writability by allowing for more expressive variable and function names. In addition to that, use of comments improves readability by making code easier to understand.

Carefully chosen reserved words ensure they accurately reflect their intended functionality and are consistent with common programming language conventions. Since Logismik provides both symbolic and natural language operators, it is up to the programmer to choose: more readable or more easy to write code. For example, programmers can use “and” or “&&”.

Overall, the language components in Projectlang1 were designed to promote clear, concise, and efficient code.

bool: This is the chosen token to declare a variable, even though there is only one type of variable. We decided that a token is needed to declare a variable different from languages like Python. It increases the readability of code and it is more intuitive.

bool[ ]: For a similar reason we have defined array declaration with square brackets, it is more intuitive.

return - break: Conventional ending statements for loops and functions.

if - else - else if: Conditional statements are standard. Programmers may use parentheses to enclose the boolean expression if they like.

while: Conventional while statement. Programmers may use parentheses to enclose the boolean expression if they like.

for each … in … : This loop statement is designed to be as intuitive as possible. When programmers read this statement, they will immediately know that for every loop step, the current array element will be stored inside the given variable.

func: This keyword is reserved as the function declaration token. It is shorter than, for example, “function” and more describing than “def”. So, we think it is the best keyword to use for this purpose.

void - bool: We have decided that it is better to have a return type, it increases readability but slightly decreases writability.

input: Even someone with no knowledge of Logismik would understand what this statement means. It is also structured as a function with a parameter list to maintain the coherence and simplicity of the language.

print: print name is reserved word for printing any strings, variables and their combinations. Since most of the languages use the print keyword for the same purposes, while designing Logismik same keyword is preferred.