CS315 - Programming Languages

YarLang Language Specification

Team 6

Ahmet Emir Boşnak - Section 1 - 22002398

Ozan Can Gülbaz - Section 1 - 22002019

Arda İynem - Section 1 - 22002717

2 April 2023

Table of Contents

1. BNF Description

```
<start> ::= <program_list>

<program_list> ::= <program>
              | <comment_list> <program>
              | <comment_list> <program> <comment_list>
              | <program> <comment_list>

<program> ::= begin <line_list> end

<comment_list> ::= <comment>
              | <comment_list> <comment>

<line_list> ::= <line>
           | <line> <line_list>

<line> ::= <comment>
      | <stmt>

<stmt> ::= <dec_stmt>;
      | <assign_stmt>;
      | <return_stmt>;
      | <while_stmt>
      | <if_else_stmt>
      | <input_stmt>;
      | <output_stmt>;
      | <func_def>
      | <func_call>;
      | <foreach_stmt>
      | <del_expr>;

<assign_stmt> ::= <id> = <expression> | <concat_stmt>
<dec_stmt> ::= <type> <declarators>
<declarators> ::= <declarator> | <declarators>, <declarator>
<declarator> ::=  <id>
            | <id> = <expression>
            | <id> ( <param_list> ) begin <line_list> end


<input_stmt> ::= input(<id_list>)
<output_stmt> ::= print(<str_const>)
                  | print(<expression>)
```

```
<concat_stmt> ::= <list_id> += <expression>

<relational_exp> ::= <id> == <id> | <id> != <id>
<del_expr> ::= <list_id>--
<empty_expr> ::= #<list_id>

<bool_expression> ::= <func_call>
            | <relational_expr>
            | <expr>
            | <del_expr>
            | <empty_expr>

<list_expression> ::= {value_list}

<expression> ::= <list_expression> | <bool_expression>

<expr_list> ::= <expression>
            | <expr_list>, <expression>

<expr> ::= <expr> <dimp_op> <term1> | <term1>
<term1> ::= <term1> <imp_op> <term2> | <term2>
<term2> ::= <term2> <or_op> <term3> | <term3>
<term3> ::= <term3> <and_op> <term4> | <term4>
<term4> ::= <not_op> <term4> | <term5>
<term5> ::= <id> | <bool_constant> | (<expr>)

<if_stmt> ::= if ( <bool_expression> ) begin <line_list> end
            | <if_stmt> <elif_stmt>
<elif_stmt> ::= elif ( <bool_expression> ) begin <line_list> end
<else_stmt> ::= else begin <line_list> end

<if_else_stmt> ::= <if_stmt> | <if_stmt> <else_stmt>

<while_stmt> ::= while (<bool_expression>) begin <line_list> end

<foreach_stmt> ::= foreach (<id> in <foreach_input>) begin <line_list> end

<foreach_input>::= <id> | {value_list}

<func_def> ::= <func> <return_type> <id>(<param_list>) begin <line_list>
end

<param_list> ::= | <type> <id> | <param_list>, <type> <id>
<func_call> ::= <id>(<value_list>) | <id>()
```

```
<value_list> ::= <id> | <bool_constant> | <value_list>, <id> |
<value_list>, <bool_constant>

<return_stmt> ::= return <expression>
<return_type> ::= <type> | <void>

<type> ::= <bool> | <list>
<bool> ::= bool
<list> ::= bool[]
<void> ::= void
<func> ::= func

<id> ::= <letter> | <id> <digit> |  <id> <letter>
<id_list> ::= <id> | <id_list>, <id>
<list_id> ::= <id>

<not_op> ::= ! | not
<dimp_op> ::= <-> | dimp
<imp_op> ::= -> | imp
<and_op> ::= & | and
<or_op> ::= | | or
<bool_constant> ::= true | false | TRUE | FALSE

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<letter> ::= a | b | … | z | A | B | … | Z |
<punctuation> ::= : | ! | ? | , | . | - | _
                  | ( | ) | ‘ | % | & | $
                  | # | ; | “ | *
<white_spc> ::= \t | \n | \s |
<str_const> ::= “<char_array>”
<char_array> ::= <char> | <char_array> <char>
<char> ::= <letter> | <digit> | <punctuation> | <white_spc>

<comment> ::= /* <str_const> */
```

2. Explanation of Language Constructs

a. Statements

i. Declaration Statement

One or more variables of the same type can be declared simultaneously in a single statement. Types can be the boolean or the boolean data structure boolean list, denoted by “bool” and “bool[]” keywords. Any declaration made in a statement might include initialization of the

value or not. Declarations without initializations only take an identifier as the variable name. For initialization, the assignment operator is used with an expression that can be evaluated to have a meaningful result of the declaration type (boolean constant for “bool”, boolean const list for “bool[]”). Declaration statements are ended with a semicolon to indicate the end of the statement.

ii. Assignment Statement

A simple “=” sign or “is” word is used for assigning expressions to variable ids. A statement ends with a semicolon for increased reliability. Using both the “=” sign and the “is” keyword increases the writability and expressibility of our language, but the drawback is that it is slightly less readable. “expression” is divided into four subcategories: An expression can be a boolean value, a variable (designated by a variable identifier), a function call, a relational expression, a list deletion expression and a boolean operation expression. Declaration statements are ended with a semicolon to indicate the end of the statement.

1. **Concatenation Assignment**
   This is a list concatenation (adding a value to the list structure) expression. We decided to write it out as an assignment statement rather than an expression since it is constructed as an assignment statement, different from all other expression types, increasing reliability. The Syntax is here simple with a “+=” sign only, which increases readability.

iii. Conditional Statements

1. **If Statement:** If statements use “begin” and “end” keywords to indicate the body of the statement inside, like functions and loops. Using “begin” and “end” keywords instead of matching brackets is simply a stylistic choice which aligns with the simple and easy-to-read nature of our language if statements use <matched> <unmatched> constructs to ensure unambiguous if-else grammar, increasing reliability.
   .

iv. Iteration Statements

1. **While Statement:** While statement use “begin” and “end” keywords to indicate the statement body. A while statement takes a single expression as the loop condition. Since the

language only supports boolean type, a while loop can only be terminated when the value of the condition expression changes. The condition must be updated inside the loop, or the while loop loses its functionality, which decreases reliability.

2. **Foreach Statement:** Foreach statement is a powerful tool in this limited language, giving the user an easy way to iterate through the list data structure, which is impossible to do with a for loop. Foreach statements also use “begin” and “end” keywords for the body. The foreach statement uses the “in” keyword to specify the value list or list variable identifier that is iterated through. A value list may contain boolean constants or boolean variable identifiers separated by commas inside curly brackets. There must be a variable identifier before the “in” keyword, which will hold the value gathered from the value list or bool list, and a list variable identifier or value list after the “in” keyword. The “in” keyword and the foreach statement are similar to C-type languages, with high readability, writability and reliability.

v. IO Statements

I/O statements are ended with a semicolon to indicate the end of the statement.

1. Input Statement: Input statements use the keyword “input” followed by the parameter list inside parentheses. The parameter list is a list of variable identifiers. The variables in the parameter list will store the input values entered by the user. These variables must be declared before in the code, the input statement does not create temporary variables, like the foreach statement.

a. User String Input (Bool Literal): All inputs are treated as strings. Only accepted strings for Bool Literal input are “true”, “false”, “TRUE” and “FALSE”. When any other string patterns (Except the Bool List input explained below) are encountered the program will show an Illegal Input Error.

b. User String Input (Bool List): Format of the user inputs for the lists should be a list of legal constant strings mentioned above, separated by whitespace characters. Boolean constant value corresponding to each string delimited by space characters is either assigned to the boolean variables passed to the input statement or added to the boolean list variable passed to the input statement in order.

2. Output Statement: Output statements use the keyword “print” followed by the output message or value inside the parentheses. The output can be a string constant, an expression (which will be evaluated to a boolean value), and a variable identifier list. The print statement will always print a string constant or a boolean value, so an identifier will be evaluated to its value and will be printed accordingly. Output messages must be wrapped around quotation marks for them to be considered string constants; everything else will be evaluated as an expression or identifier. Boolean lists will be printed with brackets and values inside, separated by commas.

vi. Function Definitions, Function Calls and Return Statement

Function definition follows the C-type language rules, with return type (boolean, boolean list or void) identifier and parameter list structure. Again, “begin”, and “end” keywords are used for the function body. The parameter list consists of variable identifiers coupled with their respective types (boolean and boolean list). Function calls are made with a function identifier followed by a list of variables inside the parentheses. Function call lists can include variable identifiers and boolean values. The return statement is a one-line statement with the “return” keyword followed by the expression to return. Function calls are ended with a semicolon to indicate the end of the statement.

b. Types

i. **Bool:** Stores true or false values

c. Data Structures

i. **List:** Stores Bool literals, the list data structure is implemented in a queue fashion

d. Expressions

i. **Operators:** In YarLang, programmers are allowed to use both the operator symbol (&, | etc.) and their equivalent verbal versions (and, or etc.). This increases the writability criteria while decreasing the readability criteria.

1. **Unary Operators**
    a. **not:** Returns inverse of the bool value
2. **Logical Binary Operators**
    a. **and:** Returns true if both operands are true
    b. **or:** Returns true if at least one of the operators are true
    c. **imp:** Returns the result of logical implication operation with two operands
    d. **dimp:** Returns the result of logical double implication operation with two operands
3. **Relational Binary Operators**
    a. **==:** Returns true if the value of two variables is equal
    b. **!=:** Returns true if the value of two variables are equal

ii. **Delete Expression:** Deletes the last item in the list and returns the deleted item. If the list is already empty before delete expression then the program shows an Empty List Deletion Exception yet the emptiness of a list can be checked with Empty Expression (#<list_id>) to prevent facing exceptions.

iii. **Empty Expression:** An expression that returns bool value true if the list operand is not empty, otherwise returns false.

3. Description of Nontrivial Tokens

a. Identifiers

Identifiers are unique names that can be assigned to variables or functions. Identifiers are alphanumeric character strings. They must start with an alphabetic character and must differ from reserved words. Multiple identifiers can be used together as a comma-separated list to represent an “identifier list”.

b. Literals

String constant literals are used for input and output operations to communicate with the user. Output prompt can show messages to the user that are specified in the program, while input functionality can get constant boolean values from user input strings. String constant literals are always between “ “ (quotation) signs. A single and simple way of string literal representation increases readability and reliability.

c. Comments

Comments are information the programmer supplies to the readers and do not affect the functionality. The beginning and ending of the comments are indicated by “/*” and “*/” symbols, respectively, to increase reliability by ensuring nothing is written between these symbols and readability by increasing the simplicity of the code by having only one set way of writing comments.

d. Reserved Words

i. true (TRUE)
ii. false (FALSE)
iii. begin
iv. end
v. print
vi. input
vii. if
viii. else
ix. return
x. while
xi. foreach
xii. func
xiii. in
xiv. bool
xv. void
xvi. or
xvii. and
xviii. imp
xix. dimp
xx. not

4. Explanation of Language Usage

a. Scope (Enclosing Scope): The outermost program scope consists of the space between the first word (begin) and the last word (end) in the code. Each function defined in the program scope has its own scope.

Variables defined in an outer scope can be accessed from an enclosed scope, while variables defined in inner scopes are not accessible.. This system makes YarLang support nested functions and nested loops. This system does not support outer scope variable hiding operation, therefore any variable in an inner scope must have a different identifier than the variables declared in its outer scopes.
With this scoping mechanism all variables declared in the program scope is a global variable as all inner scopes have access and there can not be any name conflict between global variables and inner scope variables due to naming requirements mentioned above.

b. Parameter Passing

Parameter passing is handled by pass by value strategy between formal and actual parameters of each function. YarLang does not support any pass by reference parameter passing as any pass by reference requirements can be indirectly done by the return values of the pass by value functions and assignment statements.

c. Program Execution Start

The execution of the program starts from the statements between the outermost begin - end block. Statements are executed line by line therefore function and variable declarations must be done in an order considering that subsequent functions and statements can access priorly declared functions and variables.