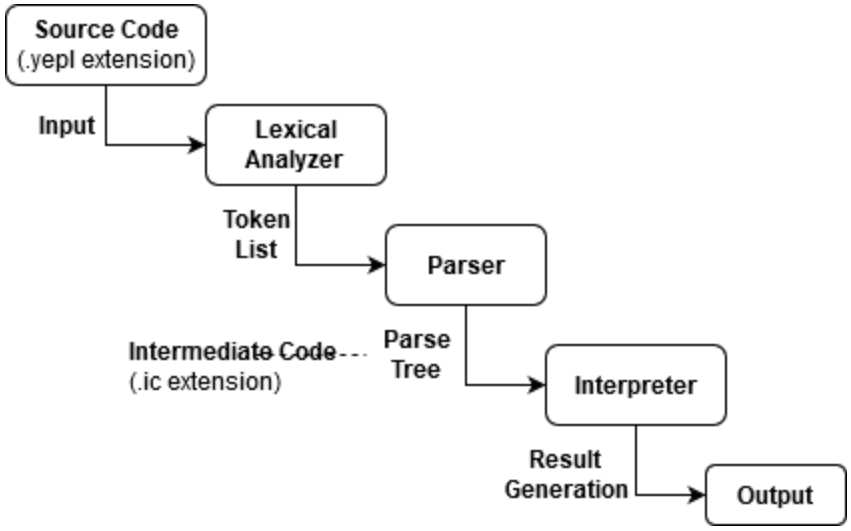
YEPL

Programming Language Name: Yepl
Language extension: .yepl

Structure

Programming paradigm: Imperative programming language

The language compilation and execution process can be broken down into the following stages:



- **Source code:**
  The source code consists of a file containing the program to be executed by Yepl language and is saved with a .yepl file extension.

- **Lexical analyzer:**
  The lexical analyzer opens the input .yepl file containing the source code and reads character by character from the file. These characters are converted into meaningful tokens recognized by the Yepl language and stores them in a token list in the sequence in which they are parsed.
  Design language – Prolog
  Data structure used – List Data Structure.

- **Parser:**
  The parser is responsible for checking whether the source code follows the syntax rules defined by the Yepl language. It reads tokens one at a time from the token list and generates a parse tree when all the tokens have been parsed. If all the tokens were not parsed, that means that the source code does not comply with the correct syntax of the language and an error message is returned by the parser.
  Design language – Prolog
  Data structure used – List data structure
  Parsing technique – Top-down parsing using unification of parse tree nodes
  Grammar rules written in – Definite Clause Grammar (DCG)

- **Intermediate code:**
  The intermediate code consists of a file generated by the parser with an extension. ic. This file contains the parse tree generated by the parser.

- **Interpreter:**
  The interpreter is responsible for reading the parse tree from the .ic file and using syntax based semantics to execute the program. We use operational and denotational semantics. We parse node by node of the parse tree in a **top down** fashion and use evaluators in Prolog to evaluate each node and at the same time, keeping a track of the changes in environment and the evaluation proceeds in the form of a Prolog list.
  Design language – Prolog
  Data structure used – List data structure

Design

1. Datatypes:

The datatypes supported by the language grammar are int, char, bool and string. The int data type can be any 32-bit integer sequence of numbers. The char datatype consists of any single character from the ASCII character set enclosed within single quotes. The bool datatype consists of the values true and false. The string datatype consists of a sequence of zero or more ASCII characters enclosed within double quotes.

2. Identifiers:

All identifier names must be composed of lowercase alphabets, uppercase alphabets, numbers, underscore and dollar characters. Identifiers cannot start with a number.

3. Variable declaration:

Variable declaration is supported both in global scope as well as within functions. A variable can be declared as a single variable or as a variable declaration list separated by commas if the variables have a common datatype. The examples given below are both valid variable declarations supported by the language:

Example 1:
```
int x = 42;
```

Example 2:
```
int x = 10, y = 20;
```

Example 3:
```
int a = 100;
int sum (int x, y) {
        int res = x + y;
return res;
}
// Here ‘a’ will be considered as global variable and ‘res’ as a member variable.
```

4. Function definition:

Function can be defined both inline as a single statement or as a block. The examples given below are valid function definitions supported by the language:

Example 1:
```
int min (int n1, n2) if (n1 < n2) return n1; else return n2;
```

Example 2:
```
int min (int n1, n2) {
        if (n1 < n2)
                return n1;
        else
                return n2;
}
```

5. Main function:

It is mandatory to implement the main () function. The flow of the code will begin from the main function.

6. Array data structure:

The Array data structure is supported by the language. The array must be declared with a predefined size. The value at any index of the array can also be assigned using the value of an expression on the right-hand side. The below examples highlight this point:

Example 1:
```
int n = 10;
int arr[n];
```

Example 2:
```
int arr[100];
arr[2] = 3 * 2;
```

7. Function parameters:

Function parameters are written within '(' and ')' braces separated as a list of zero or more <datatype, identifier> pairs separated by a semicolon ';'. The below example illustrates this point:

Example 1:
```
int sum (int a; int b)
```

Example 2:
```
int fun (int a, char b)
```

8. Initialization of variables in function parameters:

Variables can only be declared and not initialized or assigned a value inside a function parameter. The variables do not support taking of default value within a function parameter. For example, the below is NOT supported:

```
int sum (int a = 0) // Not supported
```

9. There are 7 types of statements supported by the language:

i. Expression statement: Statements used for evaluating expressions.
ii. Compound statement: Statements that consist of a block with variable declarations and a list of statements.
iii. Selection statement: Statements with conditionals using if, else and elsif statements.
iv. Iteration statement: Statements using iterative constructs such as while, for and for in range.

v. Print statement: Statements using ‘print’ keyword for printing values of identifiers, constants, expressions, etc.
vi. Break statement: Statements for breaking out of a loop using the ‘break’ keyword.
vii. Return statement: Statements for returning a value from the function using the ‘return’ keyword.

10. Selection Statements:

i. All selections statements using ‘if’ keyword must include the conditional statement within ‘(‘ and ‘)’ braces followed by a single statement of a compound statement.

Example 1: if (a < b) return a;

Example 2: if (a < b) { return a; }

ii. Support for else statement.

Example: if (a < b) return a; else return b;

iii. Support for nested if else using ‘elsif’ keyword.

Example:

```
if (a<b) {
    return a;
elsif (b > a) {
    return b;
} else {
    return 0;
}
```

11. Iterative statements:

i. Support for ‘for’ loops is available. It can be used as a traditional for loop enclosed within ‘(‘ and ‘)’ parenthesis or using ‘range in’ keyword. The initialization and conditional expression are required inside the for loop when used in the traditional format but the increment expression is optional. In the ‘range in’ format, the loop with be incremented by 1 as default. The start of the range is inclusive and the end of the range is exclusive.

Example 1:

```
for (int i=0; i<10;i++) //Valid
```

Example 2:

```
for (int i=0; i<10;) //Valid
```

Example 3:
for i in range (10, 100) // 10 is inclusive and 100 is exclusive

12. Print statement:

Print statements can be used to print any mutable and immutable value and can include identifiers, constants, function calls, expressions, etc.

Example 1:
print(“hello”); // String constant

Example 2:
print(a + b); // Expression

Example 3:
print(32); // Integer constant

Example 4:
print(a); // Variable

Example 5;
print(func(x,y)); // Prints value returned by function call

13. Support for mutability and immutability:

The concept of L-value and R-value is implemented by categorizing different constructs as being either mutable or immutable. For example, function calls, expressions and constants are immutable as they can never appear on the left side of an ‘=’ sign. On the other hand, identifiers are mutable as they can appear on the left-hand side of the ‘=’ sign.

Example:
int a = 0 //This is supported since a is mutable.
52 = 6 // This is not supported since 52 is a constant.
a + 6 = 7 // This is not supported since a + 6 is an expression.
s = func(a) //This is supported since function call is immutable and appears on RHS.
func(a) = s // This is not supported since function call is immutable.

14. Operators:

i. Support for assignment operator: ‘=’
ii. Support for mutable operators such as ‘+=’, ‘-=’, ‘*=’, ‘/=’.
iii. Support for increment and decrement operators, ‘++’, ‘--’.

iv. Support for logical operators such as ‘AND’, ‘OR’, ‘NOT’. It can also be used as ‘&&’, ‘||’ and ‘!’.
v. Support for relational operators such as ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’.
vi. Support for arithmetic operators such as ‘+’, ‘-‘, ‘*’, ‘/’, ‘%’.
vii. Support for unary operators such as ‘+’ and ‘-‘.

15. Precedence of operators:

The precedence of operators is given below in the order of high precedence to low precedence:
Assignment and Mutable operators > Increment/Decrement operators > Logical operators > Relational operators > Arithmetic operators > Unary operators.

16. Associativity or operators:

All operators except the assignment operator are left associative, whereas the assignment operator is right associative.

17. Support for function calls:

Function calls are supported by the language. The arguments used in function call is a list of zero or more expressions which form the argument list.

Example:
int a = sum (a, b + 1);

18. Function overloading is not supported. Therefore, no two functions can have the same name.

19. Function return type must be specified in the function declaration. If the function does not have a return type, we make the return type as ‘void’.

20. Code that exits a procedure without a return returns a 0 for an function returning int and false for a function returning bool, a blank for a function returning char and a null string for a function returning string.

Grammar: -

The grammar of the language given below follows the standard BNF notation for grammar rules.

Terminal Rules:

// Identifier

```
ID                                  ::= / ^[a-zA-Z_$][a-zA-Z_$0-9]*$ /

// Data constants
NUMCONST                            ::= /^[0-9]+$/
CHARCONST                           ::= /'[\x00-\x7F]'/
STRINGCONST                                 ::= / \"[\x00-\x7F]*\" /
BOOLCONST                           ::= true | false

// Data types
TYPESPECIFIER                       ::= int | bool | char | string

// Keywords
STATIC                              ::= static
IF                                  ::= if
ELSIF                               ::= elsif
ELSE                                ::= else
RETURN                              ::= return
BREAK                               ::= break
PRINT                               ::= print
SIZE                                ::= size

// Operators
ASSIGNMENT                          ::= ‘=’
MUTABLEOPERATOR                     ::= ‘+=’ | ‘-=’ | ‘*=’ | ‘/=’
INCREMENTOPERATOR                           ::= ‘++’ | ‘--'
OROPERATOR                          ::= or | ‘||’
ANDOPERATOR                         ::= and | ‘&&’
NOTOPERATOR                         ::= not | ‘!’
RELATIONALOPERATOR                  ::= ‘<=’ | ‘<’ | ‘>’ | ‘>=’ | ‘==’ | ‘! =’
ADDITIONSUBTRACTIONOPERATOR  ::= ‘+’ | ‘−‘
MULTIPLICATIONDIVISIONOPERATOR              ::= ‘∗’ | ‘/’ | ‘%’
UNARYOPERATOR                       ::= ‘+’ | ‘−‘

// Delimiters
SEMICOLON                           ::= ‘;’
COMMA                               ::= ‘,’
DOT                                 ::= '.'

// Parantheses
BLOCKBRACESBEGIN                            ::= ‘[‘
BLOCKBRACESEND                      ::= ‘]’
SBLOCK                              ::= ‘{‘
FBLOCK                              ::= ‘}’
OPARANTHESIS                        ::= ‘(‘
CPARANTHESIS                        ::= ‘)’
WHILE                               ::= while
FOR                                 ::= for
IN                                  ::= in
RANGE                               ::= range
```

Non-terminal Rules:

program ::= declarationList

declarationList ::= declarationList declaration | declaration

declaration ::= variableDeclaration | functionDeclaration

variableDeclaration ::= TYPESPECIFIER variableDeclarationlList SEMICOLON

variableDeclarationlList ::= variableDeclarationlList COMMA variableDeclarationInitialization |
variableDeclarationInitialization

variableDeclarationInitialization ::= variableDeclarationIdentifier | variableDeclarationIdentifier ASSIGNMENT
simpleExpression

variableDeclarationIdentifier ::= ID | ID BLOCKBRACESBEGIN NUMCONST BLOCKBRACESEND

functionDeclaration ::= TYPESPECIFIER ID OPARANTHESIS parameters CPARANTHESIS
statement

parameters ::= parameterList | ε

parameterList ::= parameterList , parameter | parameter

parameter ::= TYPESPECIFIER ID | TYPESPECIFIER ID BLOCKBRACESBEGIN
BLOCKBRACESEND

statementList ::= statementList statement | ε

statement ::= expressionStatement |
compoundStatement |
selectionStatement |
iterationStatement |
returnStatement |
breakStatement |
printStatement

expressionStatement ::= expression SEMICOLON | SEMICOLON

iterationRange	::= OPARANTHESIS ID ASSIGNMENT simpleExpression SEMICOLON ID RELATIONALOPERATION simpleExpression SEMICOLON CPARANTHESIS | OPARANTHESIS ID ASSIGNMENT simpleExpression SEMICOLON ID RELATIONALOPERATION simpleExpression SEMICOLON expression CPARANTHESIS | ID IN RANGE OPARANTHESIS simpleExpression COMMA simpleExpression CPARANTHESIS
iterationStatement	::= WHILE OPARANTHESIS simpleExpression CPARANTHESIS statement | FOR iterationRange statement
compoundStatement	::= SBLOCK localDeclarations statementList FBLOCK
localDeclarations	::= localDeclarations localDeclaration | ε
localDeclaration	::= TYPESPECIFIER ID SEMICOLON | TYPESPECIFIER ID BLOCKBRACESBEGIN BLOCKBRACESEND SEMICOLON
elsifList	::= elsifList ELSIF OPARANTHESIS simpleExpression CPARANTHESIS statement | ϵ
selectionStatement	::= IF OPARANTHESIS simpleExpression CPARANTHESIS statement elsifList | IF OPARANTHESIS simpleExpression CPARANTHESIS statement elsifList ELSE statement
printStatement	::= PRINT OPARANTHESIS simpleExpression CPARANTHESIS SEMICOLON
returnStatement	::= RETURN SEMICOLON | RETURN expression SEMICOLON
breakStatement	::= BREAK SEMICOLON
expression	::= mutable ASSIGNMENT expression | mutable MUTABLEOPERATOR expression | mutable INCREMENTOPERATOR | simpleExpression
simpleExpression	::= simpleExpression OROPERATOR andExpression | andExpression
andExpression	::= andExpression ANDOPERATOR unaryRelationalExpression | unaryRelationalExpression

unaryRelExpression ::= NOTOPERATOR unaryRelationalExpression | relationalExpression

relationalExpression ::= additionSubtractionExpression RELATIONALOPERATOR
additionSubtractionExpression | additionSubtractionExpression

additionSubtractionExpression ::= additionSubtractionExpression ADDITIONSUBTRACTIONOPERATOR
multiplicationDivisionExpression | multiplicationDivisionExpression

multiplicationDivisionExpression ::= multiplicationDivisionExpression
MULTIPLICATIONDIVISIONOPERATOR
unaryExpression | unaryExpression

unaryExpression ::= UNARYOPERATOR unaryExpression | factor

factor ::= immutable | mutable

mutable ::= ID | mutable BLOCKBRACESBEGIN expression BLOCKBRACESEND

immutable ::= OPARANTHESIS expression CPARANTHESIS | ID DOT SIZE | functionCall
| constant

functionCall ::= ID OPARANTHESIS arguments CPARANTHESIS

arguments ::= argumentList | ϵ

argumentList ::= argumentList COMMA expression | expression

constant ::= NUMCONST | CHARCONST | STRINGCONST | BOOLCONST