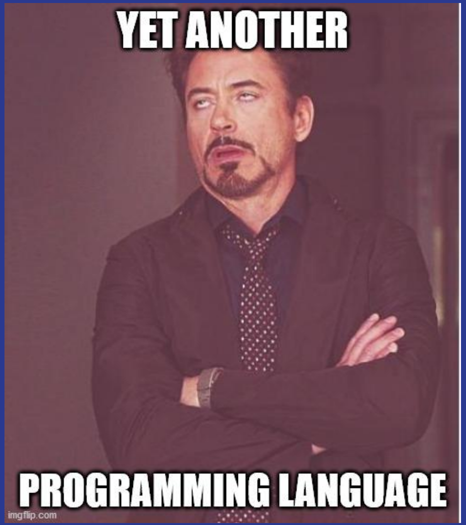
SER 502 - Spring 2019 - Team 3

YEPL
(Yet another Programming Language)

Aditya Bajaj - ASU IDs
Aihaab Shaikh
Sakshi Jain
Sukhpreet Anand



Overview

- Features of the Language
- Language Design
- Language Grammar
- Future Work

Features of YEPL

YEPL Supports

- Integer Type
- Boolean Type
- String Type
- If-else-if loop
- While and for loop
- Basic Arithmetic Operators such as +, -, *, /

YEPL Features

Statements :

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 elseif 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.

Support of other 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 ‘-‘.

YEPL Features

Statements :

i. Expression statement: Statements used for evaluating expressions.
Example: int y = 5 ;

ii. Compound statement: Statements that consist of a block with variable declarations and a list of statements.
Example: int y = 5;{y += 5;}

iii. Selection statement: Statements with conditionals using if, else and elseif statements.
Example: int x = 6; if(x==2) print(x); else print(“x is not 6”)

iv. Iteration statement: Statements using iterative constructs such as while, for and for in range.
Example: int x=3; while(x!=0){ print(x); x--; }

v. Print statement: Statements using ‘print’ keyword for printing values of identifiers, constants, expressions, etc.
Example: print(03);

YEPL Features

Support of other operators :

i. Support for assignment operator: ‘=’
Example: int y = 5 ;

ii. Support for mutable operators such as ‘+=’, ‘-=’, ‘*=’, ‘/=’.
Example: int y = 5; y+=5; print(y);

iii. Support for increment and decrement operators, ‘++’, ‘--’.
Example: int y = 5; y++; print(y);

iv. Support for logical operators such as ‘AND’, ‘OR’, ‘NOT’. It can also be used as ‘&&’, ‘||’ and ‘!’.
Example: while(3|| 5 and 9)89;

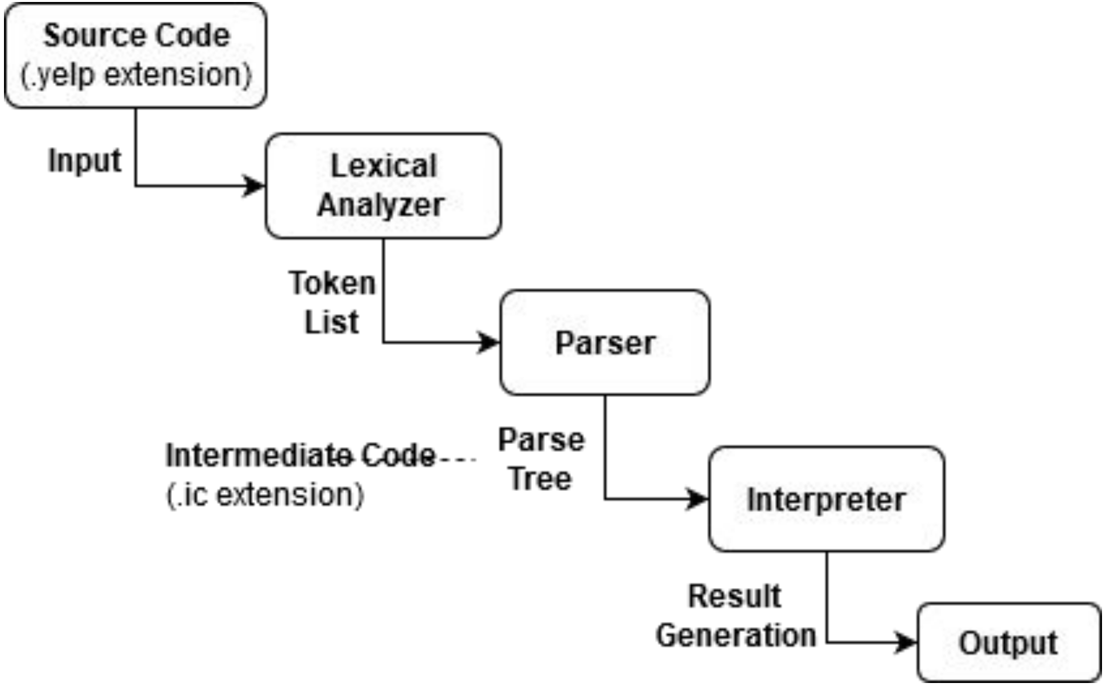
v. Support for relational operators such as ‘<’, ‘>’, ‘<=’, ‘>=’, ‘==’, ‘!=’.
Example: if (5 != 9) print(“false”) ;

vi. Support for arithmetic operators such as ‘+’, ‘-‘, ‘*’, ‘/’, ‘%’.
Example: if (5-(9*0) == 5) print(“Yes”);

vii. Support for unary operators such as ‘+’ and ‘-‘.
Example: if (-5-(9*0) != 5) print(“Yes”);

Language Design

Language Design



Components used in the design

1. Source Code
2. Lexical Analyzer
3. Parser
4. Intermediate code
5. Interpreter

Source Code

- The source code consists of a file containing the program to be executed by YEPL language and is save with a “ .yepl “ file extension.
- This source code is then read as the input by the Lexer.

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 that are recognized by the YEPL language and stores them in a list of tokens

Parser

- The parser is responsible for checking whether the source code follows the syntax rules defined by the YEPL language.
- A parse tree is generated from the list of tokens generated by the lexical analyzer.
- If all the tokens were not parsed, it means that the source code does not comply with the correct syntax of the language. In such cases an error message will be returned by the parser.
- Top-down parsing technique is used.

Intermediate Code

- The intermediate code consists of a file generated by the parser .
- The file extension is “ .ic “ .
- This file contains the parse tree for the source code.

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.
- Nodes of the parse tree are parsed in a top-down fashion.
- Evaluators in prolog are used to evaluate each node.
- At the same time we keep track of changes in the environment.

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 | string

// Keywords
STATIC                              ::= static
IF                                  ::= if
ELSIF                               ::= elsif
ELSE                                ::= else
PRINT                               ::= print

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

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

// Parantheses
BLOCKBRACESBEGIN                            ::= ‘[‘
BLOCKBRACESEND                              ::= ‘]’
SBLOCK                                      ::= ‘{‘
FBLOCK                                      ::= ‘}’
OPARANTHESIS                                ::= ‘(‘
CPARANTHESIS                                ::= ‘)’

// Loops
WHILE                                       ::= while
FOR                                         ::= for
IN                                          ::= in
RANGE                                       ::= range
```

Non-Terminal Rules

program	::= declarationList
declarationList	::= declarationList declaration | declaration
declaration	::= variableDeclaration
variableDeclaration	::= TYPESPECIFIER variableDeclarationlList SEMICOLON
variableDeclarationlList	::= variableDeclarationlList COMMA variableDeclarationInitialization | variableDeclarationInitialization
variableDeclarationInitialization	::= variableDeclarationIdentifier | variableDeclarationIdentifier ASSIGNMENT simpleExpression
variableDeclarationIdentifier	::= ID | ID BLOCKBRACESBEGIN NUMCONST BLOCKBRACESEND
statementList	::= statementList statement | ε
statement	::= expressionStatement | compoundStatement | selectionStatement | iterationStatement | printStatement

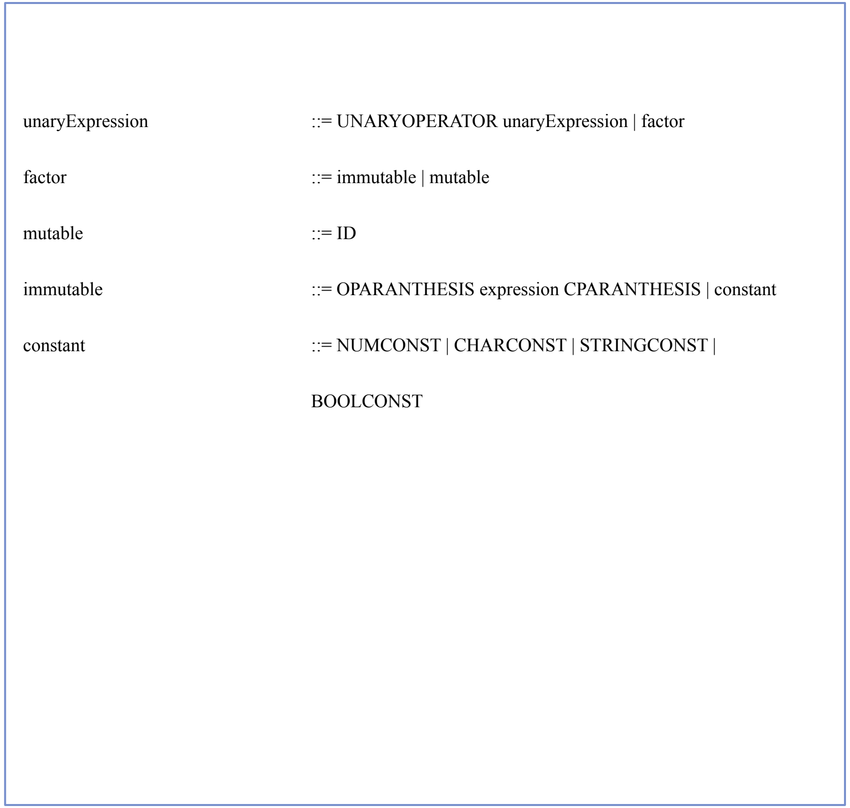
expressionStatement	::= expression SEMICOLON | SEMICOLON
iterationRange	::= OPARANTHESIS ID ASSIGNMENT simpleExpression SEMICOLON ID RELATIONALOPERATION simpleExpression SEMICOLON CPARANTHESIS | OPARANTHESIS ID ASSIGNMENT simpleExpression SEMICOLON ID RELATIONALOPERATION simpleExpression SEMICOLON expression CPARANTHESIS | ID IN RANGE OPARANTHESIS simpleExpression COMMA simpleExpression CPARANTHESIS
iterationStatement	::= WHILE OPARANTHESIS simpleExpression CPARANTHESIS statement | FOR iterationRange statement

Non-Terminal Rules

compoundStatement	::= SBLOCK localDeclarations statementList FBLOCK
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
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

Non-Terminal Rules



```
unaryExpression          ::= UNARYOPERATOR unaryExpression | factor

factor                   ::= immutable | mutable

mutable                  ::= ID

immutable                ::= OPARANTHESIS expression CPARANTHESIS | constant

constant                 ::= NUMCONST | CHARCONST | STRINGCONST |

                         BOOLCONST
```

Future Work

Future Work

- Complex data types such as Array, Lists, Sets can be added for higher order logic implementation.
- Function declaration can be implemented.
- Object oriented concepts such as inheritance, polymorphism etc can be incorporated.