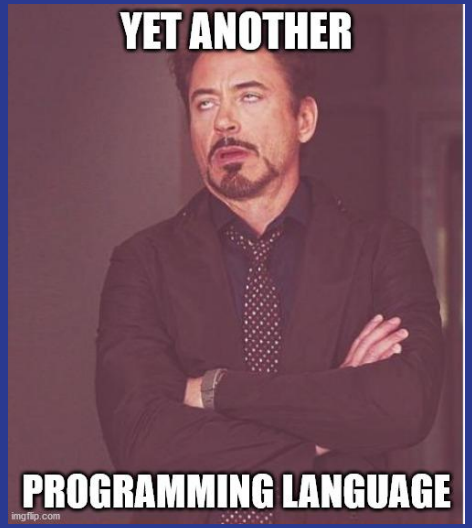
SER 502 - Spring 2019 - Team 3

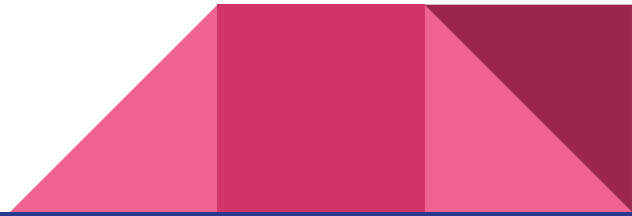
YEPL
(Yet another Programming Language)

Aditya Bajaj *(anbajaj@asu.edu)*
Aihaab Shaikh (aashaik2@asu.edu)
Sakshi Jain (smjain@asu.edu)
Sukhpreet Anand (ssanand3@asu.edu)



Overview

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



Features of YEPL

YEPL Supports

1) Implemented primitive data types - bool, int, string.

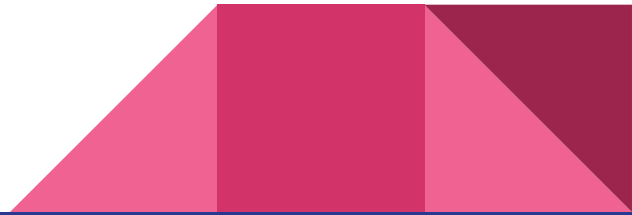
2) Implemented operations on bool and int data type.

3) Support for addition, subtraction, multiplication and division operations on “int” data type.

4) Support for and, not, or operations for bool data types.

5) Support for assignment operations and evaluation of expressions.

6) Support for 'if-else' selection statements.



YEPL Supports

7) Support for traditional 'while' iteration statements.

8) Support for traditional 'for' iteration statement.

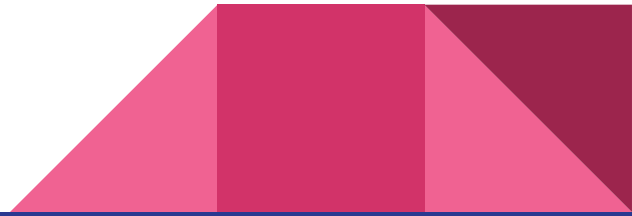
9) Support for 'for in range' iteration statement.

10) Support for 'ternary operator (?:)'.

11) Generates intermediate code (parsetree) and saves it to a .ic file.

12) Interpreter takes .ic file as input and prints the output on the Prolog runtime environment.

13) Support for 'print' statement.



YEPL Features

Statements :

i. Expression statement: Statements used for evaluating expressions.
   Example: int x, y = 5 ;
ii. Compound statement: Statements that consist of a block with a list of statements.
   Example: int y = 5; {y *= 2; y += 5;}
iii. Selection statement: Statements with conditionals using if, else and elseif statements.
   Example: int x = 6; if(x==2) print(2); elseif(x==3) print(3); else print("not 2 and 3")
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 ternary operators, '? :'.
   Example: int y = 5; y = (y == 5) ? 4 : 3;
iv. Support for increment and decrement operators, '++', '--'.
   Example: int y = 5; y++; print(y);
v. Support for logical operators such as 'AND', 'OR', 'NOT'. It can also be used as '&&', '||' and '!'.
   Example: bool i = true, j = false; if (i || j == true) print(i);
vi. Support for relational operators such as '<', '>', '<=', '>=', '==', '!='.
   Example: if (5 != 9) print("false") ;
vii. Support for arithmetic operators such as '+', '-', '*', '/', '%'.
   Example: if (5-(9*0) == 5) print("Yes");
viii. Support for unary operators such as '+' and '-'.
   Example: if (-5-(9*0) != 5) print("Yes");

Extra features

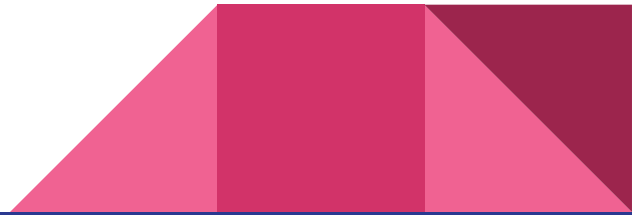
1) Implemented **type safety check** to ensure a variable holds a value permitted by the domain defined by its datatype.

2) Implemented **type casting** of data types bool and int.

3) Support for **nested if-elseif-else** statements.

4) Support for **different variants of for loops**, with or without initialization statement, with or without increment statement.

5) Support for **mutable and non-mutable expressions** to handle r-value and l-value safety checks.



Extra features

6) Support for **mutable operators**: "=", "+=", "-=", "*=", "/=", "++" and "--".

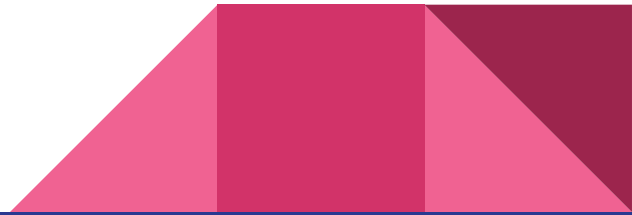
7) Support for **relational operators**: "<", "<=", ">", ">=", "==" and "!=".

8) Support for **unary operators**: "+" and "-".

9) Support for '%' **Modulus arithmetic operator**.

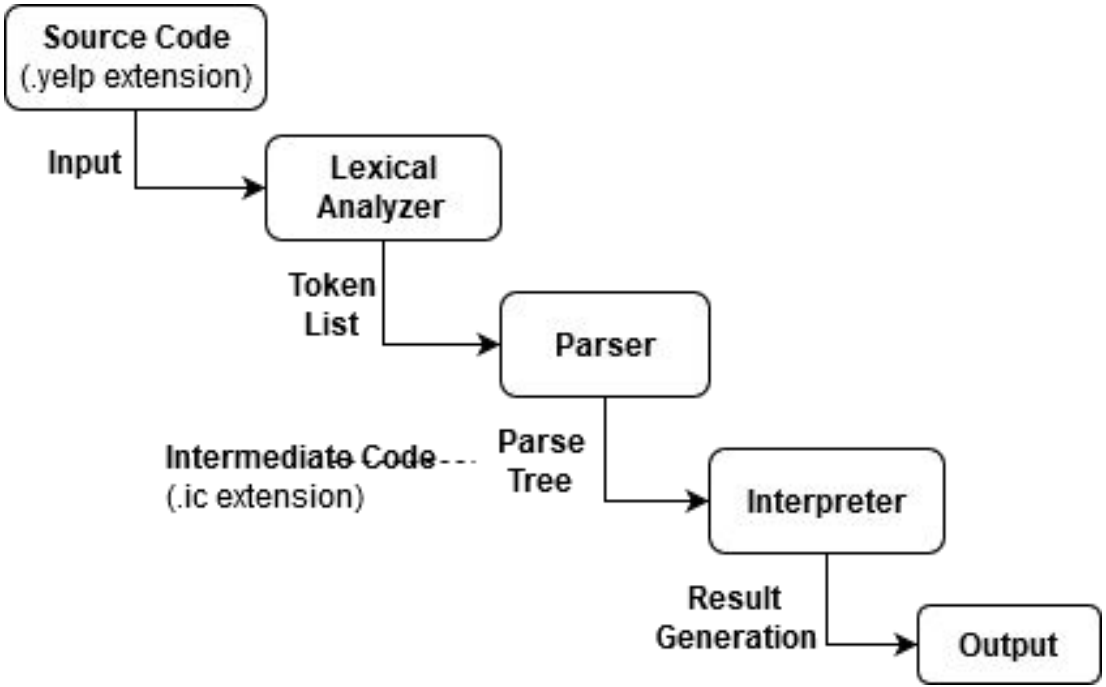
10) Support for **Parentheses** in expressions "(" and ")".

11) Handling of **precedence and associativity** for all expression evaluations based on C language.



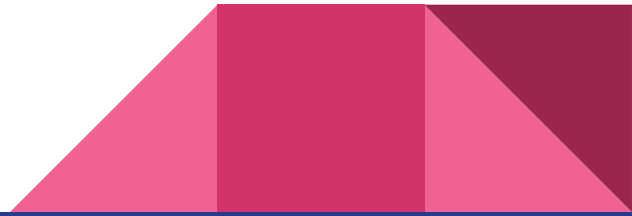
Language Design

Language Design



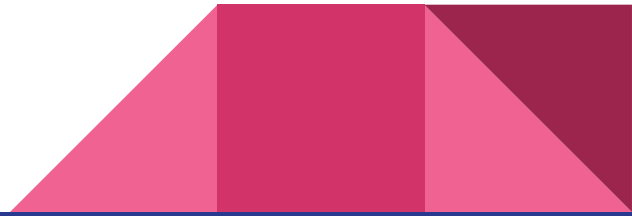
Components used in the design

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



Source Code

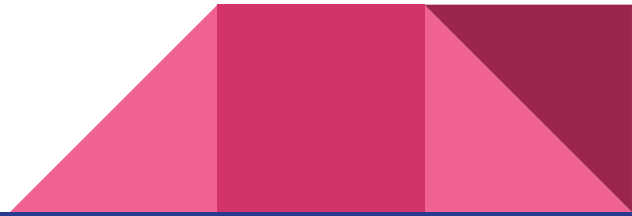
- The source code consists of a file containing the program to be executed by YEPL language and is save with a “ .yepl “ file extension.
- This source code is then read as the input by the Lexer.



Lexical Analyzer

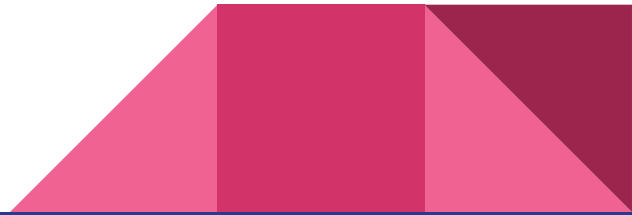
- The lexical analyzer opens the input .yepl file containing the source code and reads character by character from the file.
- These characters are converted into meaningful tokens that are recognized by the YEPL language and stores them in a list of tokens
- lexer(Cs, Tokens) :-

    phrase(tokens(Tokens), Cs)



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.

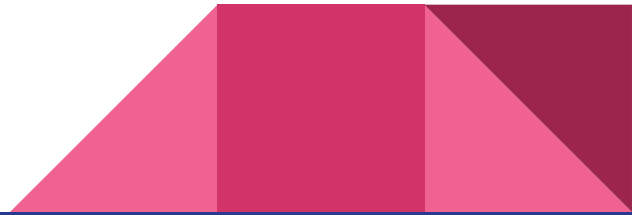


Intermediate Code

- 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]+$/
STRINGCONST                         ::= / \"[\x00-\x7F]*\" /
BOOLCONST                           ::= true | false

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

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

// Delimiters
SEMICOLON                           ::= ‘;’
COMMA                               ::= ‘,’
```

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

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

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

Non-Terminal Rules

program ::= block

block ::= declaration_list SEMICOLON statement_list | declaration_list SEMICOLON | statement_list | ϵ

declaration_list ::= declaration SEMICOLON declaration_list | declaration

declaration ::= TYPESPECIFIER variable_declaration_list.

variable_declaration_list ::= variable_declaration_list COMMA variable_declaration_initialize | variable_declaration_initialize

variable_declaration_initialize ::= variable_declaration_id | variable_declaration_id ASSIGNMENT simple_expression

variable_declaration_id ::= ID

statement ::= expression_statement | compound_statement | selection_statement | iteration_statement | print_statement

expressionStatement ::= expression SEMICOLON | SEMICOLON

compoundStatement ::= SBLOCK statementList FBLOCK | SBLOCK FBLOCK

statment_list ::= statement_list statement | statement

elsifList ::= elsifList elseif

elseif ::= ELSEIF OPARANTHESIS simpleExpression CPARANTHESIS statement

selection_statement ::= IF OPARANTHESIS simpleExpression CPARANTHESIS statement elsifList | IF OPARANTHESIS simpleExpression CPARANTHESIS statement elsifList ELSE statement | IF OPARANTHESIS simpleExpression CPARANTHESIS statement | IF OPARANTHESIS simpleExpression CPARANTHESIS statement ELSE statement

Non-Terminal Rules

iterationRange	::= OPARANTHESIS mutable ASSIGNMENT simple_expression SEMICOLON mutable RELATIONOPERATOR simple_expression SEMICOLON CPARANTHESIS | OPARANTHESIS mutable mutable RELATIONOPERATOR simple_expression SEMICOLON CPARANTHESIS | OPARANTHESIS mutable ASSIGNMENT simpleExpression SEMICOLON mutable RELATIONALOPERATION simpleExpression SEMICOLON expression CPARANTHESIS | mutable IN RANGE OPARANTHESIS simpleExpression COMMA simpleExpression CPARANTHESIS
iteration_statement	::= WHILE OPARANTHESIS simpleExpression CPARANTHESIS statement | FOR iterationRange statement
print_statement	::= PRINT OPARANTHESIS simpleExpression CPARANTHESIS SEMICOLON

expression	::= mutable ASSIGNMENT expression | mutable MUTABLEOPERATOR expression | mutable INCREMENTOPERATOR | ternary_expression
ternary_expression	::= simple_expression QUESTIONMARK expression COLON expression | simple_expression
simple_expression	::= simple_expression OROPERATOR andExpression | andExpression
andExpression	::= andExpression ANDOPERATOR unaryRelationalExpression | unaryRelationalExpression
unaryRelExpression	::= NOTOPERATOR unaryRelationalExpression | relationalExpression
relationalExpression	::= additionSubtractionExpression RELATIONALOPERATOR additionSubtractionExpression | additionSubtractionExpression

Non-Terminal Rules

additionSubtractionExpression	::= additionSubtractionExpression ADDITIONSUBTRACTIONOPERATOR multiplicationDivisionExpression |multiplicationDivisionExpression
multiplicationDivisionExpression	::= multiplicationDivisionExpression MULTIPLICATIONDIVISIONOPERATOR unaryExpression | unaryExpression
unaryExpression	::= UNARYOPERATOR unaryExpression | factor
factor	::= immutable | mutable
mutable	::= ID
immutable	::= OPARANTHESIS expression CPARANTHESIS | constant
constant	::= NUMCONST | 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.