

Name: Prathamesh Prabhakar Thakare  
Class: TY CS D  
Batch: 2  
Roll number: 58  
PRN: 12220016

Assignment number 5: Write a Yacc Program to generate a parse tree from given RE.

### Theory:

#### **Yacc (Yet Another Compiler Compiler):**

Yacc is a tool used for generating parsers or syntax analyzers. It takes a formal grammar as input and generates C code for a parser that can parse input according to the specified grammar rules. Yacc is often used in combination with Lex to create complete compilers or interpreters.

#### **Structure of Yacc Code:**

**1. Parser Rules Section (%% ... %%):**

This section defines the context-free grammar rules that specify the syntax of the language you are parsing. Each rule consists of a non-terminal symbol, a colon, and a sequence of symbols (terminals and non-terminals) representing the rule's production.

**2. Yacc Actions:**

Actions are C code snippets associated with grammar rules. They are executed when a production is reduced during parsing. Actions perform actions like creating abstract syntax trees, evaluating expressions, or generating intermediate code.

**3. Token Definitions:**

You define tokens using %token or %left, %right, %nonassoc declarations. These declarations specify the tokens used in the grammar and their associativity and precedence.

**4. User Code Section:**

This section typically includes the main() function or other code to initialize the parser and start the parsing process. It sets up input sources, invokes the parser, and handles parsing results.

**5. Error Handling (yyerror):**

You can define a custom error handling function yyerror to handle syntax errors during parsing. It is called when a parsing error occurs.

### **Algorithm:**

1. Define constants and global variables:
  - MAX: Maximum size for various arrays.
  - productions[MAX][MAX]: Array to store productions.
  - count: Counter for the number of productions.
  - i, j: Loop counters.
  - temp[MAX + MAX], temp2[MAX + MAX]: Temporary strings.
2. Define tokens for the lexer:
  - ALPHABET: Represents alphabetic characters.
3. Define operator precedence and associativity rules:
  - %left '|': Left-associative OR operator.

- %left '!': Left-associative concatenation operator.
- %nonassoc '\*': Non-associative Kleene star operator.
- %nonassoc '+': Non-associative positive closure operator.

4. Define grammar rules using Yacc/Bison syntax:

- S: Starting symbol for the grammar, represents regular expressions.
  - On reduction, print the rightmost derivation of the input regular expression.
- re: Represents regular expressions.
  - Handle various cases like single alphabet, parentheses, Kleene star, positive closure, concatenation, and alternation.
  - Store productions in the 'productions' array and increment the 'count' variable.

5. Implement the main function:

- Call yyparse to start parsing.

6. Implement the lexer function yylex:

- Read characters from the input and set yylval to the read character.
- If the character is alphabetic, return the ALPHABET token.
- Otherwise, return the character itself.

7. Implement the error reporting function yyerror:

- Print an error message to stderr and exit.

8. Implement the getREindex function:

- Find the index of the substring "re" in a given string and return the index.

9. Inside the 'S' rule of the grammar, construct and print the rightmost derivation of the regular expression using a loop. Start from the last production and work backward, modifying 'temp' and 'temp2' strings as necessary.

10. In the 're' rule of the grammar, create productions for different regular expression cases and store them in the 'productions' array.

11. The lexer (yylex) reads characters one by one and classifies them as tokens, while the grammar (Yacc/Bison rules) define the structure of regular expressions.

12. The main function initiates the parsing process by calling yyparse.

13. Error handling is provided through the yyerror function, which prints an error message if a syntax error is encountered during parsing.

14. The getREindex function helps in constructing rightmost derivations by finding the index of "re" in a given string.

15. The provided code is designed to parse and derive regular expressions, printing the rightmost derivation as the output.

## Code

```
%{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define MAX 100

int getREindex ( const char* );

signed char productions[MAX][MAX];
int count = 0 , i , j;
char temp[MAX + MAX] , temp2[MAX + MAX];
%}

%token ALPHABET

%left '|'
%left '.'
%nonassoc '*' '+'

%%
S : re '\n' {
    printf ( "This is the rightmost derivation--\n" );
    for ( i = count - 1 ; i >= 0 ; --i ) {
        if ( i == count - 1 ) {
            printf ( "\nre => " );
            strcpy ( temp , productions[i] );
            printf ( "%s" , productions[i] );
        }
        else {
            printf ( "\n  => " );
            j = getREindex ( temp );
            temp[j] = '\0';
            sprintf ( temp2 , "%s%s%s" , temp ,
productions[i] , (temp + j + 2) );
            printf ( "%s" , temp2 );
            strcpy ( temp , temp2 );
        }
    }
    printf ( "\n" );
    exit ( 0 );
}
re : ALPHABET {
    temp[0] = yylval; temp[1] = '\0';
    strcpy ( productions[count++] , temp );
}
| '(' re ')'
{ strcpy ( productions[count++] , "(re)" ); }
| re '*'
{ strcpy ( productions[count++] , "re*" ); }
| re '+'
```

```

        { strcpy ( productions[count++] , "re+" ); }
| re '|' re
        {strcpy ( productions[count++] , "re | re" );}
| re '.' re
        {strcpy ( productions[count++] , "re . re" );}
| re '*' re
        {strcpy ( productions[count++] , "re * re" );}
| re '+' re
        {strcpy ( productions[count++] , "re + re" );}
;
%%
int main ( int argc , char **argv )
{

    yyparse();

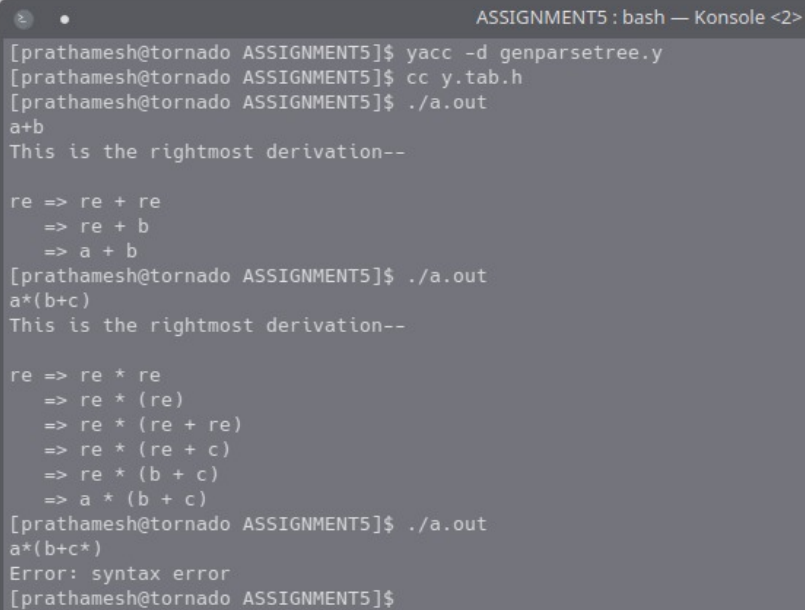
    return 0;
}

yylex()
{
    signed char ch = getchar();
    yylval = ch;
    if ( isalpha ( ch ) )
        return ALPHABET;
    return ch;
}
yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
    exit(1);
}

int getREindex ( const char *str )
{
    int i = strlen ( str ) - 1;
    for ( ; i >= 0 ; --i ) {
        if ( str[i] == 'e' && str[i-1] == 'r' )
            return i-1;
    }
}

```

## Output



```
ASSIGNMENT5 : bash — Konsole <2>
[prathamesh@tornado ASSIGNMENT5]$ yacc -d genparsetree.y
[prathamesh@tornado ASSIGNMENT5]$ cc y.tab.h
[prathamesh@tornado ASSIGNMENT5]$ ./a.out
a+b
This is the rightmost derivation--

re => re + re
   => re + b
   => a + b
[prathamesh@tornado ASSIGNMENT5]$ ./a.out
a*(b+c)
This is the rightmost derivation--

re => re * re
   => re * (re)
   => re * (re + re)
   => re * (re + c)
   => re * (b + c)
   => a * (b + c)
[prathamesh@tornado ASSIGNMENT5]$ ./a.out
a*(b+c*)
Error: syntax error
[prathamesh@tornado ASSIGNMENT5]$
```

## Conclusion

In this assignment, we created a program using Lex and Yacc (Flex and Bison) to analyze and process regular expressions. The program's main components included a lexer (yylex) to tokenize input characters, a parser (yyparse) to generate rightmost derivations of regular expressions, and an error handling mechanism (yyerror) for reporting syntax errors.