PAGE NO.:
DATE.: / / 20

# Practical No. 01

Aim: Write a Lex program to recognize valid strings using regular expression : (a|b)*abb

Test cases :

Valid strings : ababbb, abb, bbbabb

Invalid strings : baba, bba, bbb, ababba

Tools :

| Sr. No. | Software / Hardware | Sparification | Quantity |
|---|---|---|---|
| 1 | FLEX Compiler | Win10, 64bit, .exe | - |
| 2 | Computer System | Win10 or above, | 1 |
| | | 500GB HDD / 250GB SSD | |
| | | i3 processor | |

Theory : Regular Expression :

Every regular set can be expressed by expression called as Regular expression.

| Operator | priority | Meaning |
|---|---|---|
| ? | 1 | At most one occurences of preceding R.E |
| * | 2 | Zero or more occurences of preceding R.E |
| . | 3 | Concatenation |
| + OR \| | 4 | Union |

Rules : 1) $\epsilon$ is a regular expression & L($\epsilon$) is {$\epsilon$}, i.e language whose sole member is empty string.

2) If a is symbol in $\Sigma$, then a is regular expression & L(a) = {a}

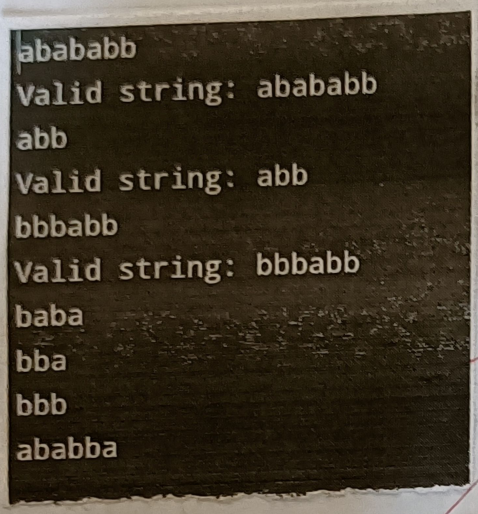
3) If r & s are two R.E then L(r) ∪ L(s), L(r)|(s) is R.E

4) (r)* is regular-expression 5) (r)(s) is R.E

Amar KRISH

Teacher's Signature _____

Results:


abababb
Valid string: abababb
abb
Valid string: abb
bbbabb
Valid string: bbbabb
baba
bba
bbb
ababba

PAGE NO.:
DATE.: / / 20

Program:

```
%{
    #include<stdio.h>
%}
%%
(a|b)*(abb)     { printf("string is valid %s", yytext); }
%%
int main()
{
    yylex();
    return 0;
}
```

Error & Remedial action:

error : could not locate lex.bak
action : reinstalled the flex software

Result :

Conclusion : LEX program to recognize given valid string implemented successfully & all test cases verified

Course Outcome attained & Mapping with Program Outcome :

| Course Outcome | Program Outcome |
|---|---|
| | 2. Problem analysis |
| CO-L: Generate scanner & parser from formal specifications | 3. Design Development of solution<br>5. Modern tool usage |

Amar KRISH

Teacher's Signature _____

PAGE NO.:
DATE.: / / 20

# Practical No. 02

**Aim:** Write a LEX program to recognize tokens 1) Identifier 2) Integer constant (Decimal, Octal, Hexadecimal) 3) Real constants (Floating point and Exponential Format)

**Test Cases:**

a. Integer Constant Ex +15, -5, 0123, 0x1A

b. Floating Point Constant Ex. +0.5, 288.0, .567, -28.89

c. Exponential format Ex. 1.6e-19, -0.5E+2, +1.7E4, 26e7

**Tools:**

| Sr. | Tools | Specification | Quantity |
|---|---|---|---|
| 1. | LEX Compiler | - Win10, 64bit | - |
| 2. | Computer System | - Windows 10+, 500/i3/4GB | - 01 |

**Theory:** a) Regular E for Decimal constant

Rules: 1. must have at least one digit

2. May be +ve or -ve 3. No spaces, comma, Period (.) are allowed.

R.E = (+|-)?(Digit)+

b) R.E for Real Const. (Floating point notation)

1. Must have at least one digit 2. may be +ve or -ve

3. Must contain decimal point 4. at least one digit after decimal point.

5. No (_) (,) (.)

R.E: (+|-)?(Digit)*.(Digit)+

c) Regular Expression for Real constant (Exponential Notation)

1. Mantissa & exponent separated by 'E' or 'e'

2. Mantissa may be real or integer. 3. +ve or -ve

4. Integer Exp 5] Exponent may +ve or -ve

6. No (-) (,) (.)

R.E: (+|-)?(Digit)*.(Digit)+(E|e)(+|-)?(Digit)+

Amar KRISH

Teacher's Signature _____

Program:

```
%{ #include<stdio.h> %}
Digit [0-9]+
%%
(+|-)?(Digit)+                          { printf("Integer/Decimal "); }
(+|-)?(Digit)*.(Digit)+                 { printf("Floating point real const "); }
(+|-)?(Digit)*.(Digit)+(E|e)(+|-)?(Digit)   { printf("Exponential real const "); }
(+|-)?(0)(0-7)+                         { printf("Octal Number "); }
(+|-)?(0X|0x)(0-9|A-F|a-f)+             { printf("Hexadecimal "); }
%%
int yywrap()
{ return 1; }
int main()
{ yylex();
  return 0;
}
```

Output / Result:

```
+15
Integer/Decimal
-5
Floating point real const
0123
Octal Number
0x1A
Exponential real constant
+0.5
Floating point real const
236.0
Floating point real const
.567
Floating point real const
-26.89
Floating point real const
1.6e-19
Exponential real constant
-0.5E+2
Exponential real constant
```

## Error & remedial action :

error : Integer underflow : occurs when input program contains -ve integer constant i.e to large in absolute value to be represented as decimal integer.

Remedial : we can check length of input string before converting it to an integer & raise an error if length for given data type

## Conclusion :

Lex program to recognize tokens 1] Identifier 2] Integer const. 3] Real const. implemented successfully

## Course Outcome Attained & mapping with Program O/c :

| Course Outcome | Program Outcome |
|---|---|
| CO-1 : Generate & parser from formal specification | PO-1, PO-2, PO3, PO4, PO5 |

03/08/23

PAGE NO.:
DATE.: / / 20

# Practical No. 03

**Aim:** Source code in a text file. Constants used in the program are strings. Write a program to read integer constants as a string and convert it to decimal Number. Integer constants are represented in Decimal, Octal and Hexadecimal. It may be +ve or -ve.

Test Cases: a. Decimal Const: +15, -5, 123.
b. octal const. +0123, 0117, -0777
c. Exponential Format Ex. 1.6e-19, -0.5E+2, +1.7E4

**Tools:**

| Sr. | Tools | Specification | Quantity |
|---|---|---|---|
| 1 | Lex Compiler (FLEX) | FLEX, Win 10, | -- |
| 2. | Computer System | 64 bit Windows 10 OS, i3+ processor, 4GB RAM | 01 |

**Theory:** In this experiment we are defining rules to match integer constants that are represented in decimal, octal & hexadecimal formats. program reads i/p program characters by character & matches it against defined rules.

- When match is found the program performs an action, which is in case is to convert matched integer constant to a decimal number and print it to the output.
- We define 3 helper fun^n to convert decimal, octal, and hexadecimal strings to decimal integers.

1] First fun^n will convert any octal no. to decimal by applying applicable method to convert.
2] Second fun^n will convert any Hexadecimal, floating into decimal.

Amar KRISH

Teacher's Signature _____

Program:

```
%{
#include<stdio.h>
#include<string.h>
int str_to_dec(const char *str){ int result=0;
    int len = strlen(str);  for(int i=0; i<len; i++){
        result = result*10 + (str[i]-'0');
    }
    return result;
}
int oct_to_dec(const char *str){ int result=0; int len=strlen(str);
    for(int i=0; i<len; i++){ result = result*8 + (str[i]-'0');
    }
return result;
}
int hex_to_dec(const char *str){ int result=0; int len=strlen(str);
    for(int i=0; i<len; i++){
    if(str[i]>='0' && str[i]<='9'){
        result = result*16 + (str[i]-'0');
    } else if(str[i]>='a' && str[i]<='f'){
        result = result*16 + (str[i]-'a'+10);
    } else if(str[i]>='A' && str[i]<='F'){
        result = result*16 + (str[i]-'A'+10);
    }}
    return result;
}
%}
%%
[-+]?[0-9]+ {printf("%d\n", str_to_dec(yytext));}
[-+]?0[0-7]+ {printf("%d\n", oct_to_dec(yytext+1));}
[-+]?0[xX][0-9a-fA-F]+ {printf("%d\n", hex_to_dec(yytext+2));}
%%
int main(){
    yylex();
    return 0;
}
```

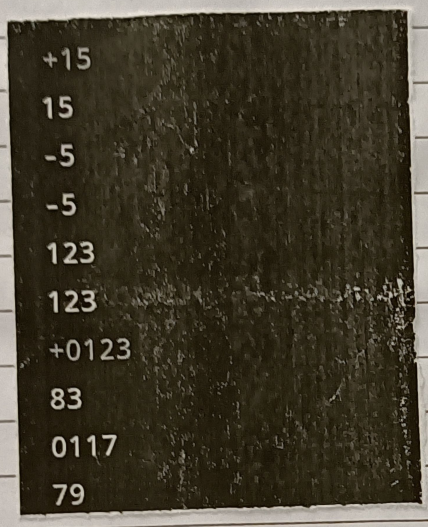
PAGE NO.:
DATE.: / / 20

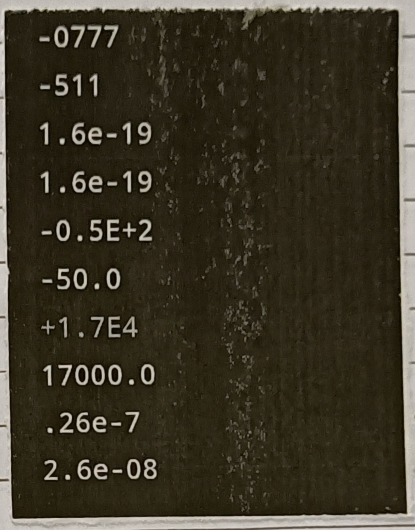
Error or remedial action:

error: Lexical error: occurs when ip program contains characters that do not match any of defined reyles. Lexer will not able to identify input as valid integer const.

remedial: add cal printf("Invalid input \n"); statement.

Result:




Conclusion: Lex program to convert input string constant that could be string, octal or hexadecimal implemented successfully

Course Outcome and mapping with program outcome:

| Course Outcome | Program Outcome |
|---|---|
| CO-1: Generate scanner and parser from formal specs. | PO1, PO2, PO3, PO5, PO4 |

Amar KRISH

Teacher's Signature _____

PAGE NO.:
DATE.: / / 20

Practical No. 04

Aim: Write a program to eliminate left recursion from the grammar. Test Cases: 1. E→E+E | E-E | id
2. S→(L) | a
L→L,S | S

Tools:

| Sr. | Tools | Specification | Quantity |
|---|---|---|---|
| 1 | – Lex compiler, C compiler | – any GDB compiler | – |
| 2 | – Computer System | – min 512GB HDD/4GB RAM | 01 |

Theory:

Steps to eliminate left recursion from gramar.

1. Identify left-recursive productions: look for prod in grammar that have form A → A alpha | β where A is nonterminal symbol & α & β are strings of terminals & non-terminals
2. Eliminate direct left recursion: to eliminate, you can replace left-recursive production A → Aα | β with a new production A → βA', where A' is a new nonterminal symbol. then add, new productions A' → αA' | ∈
3. Eliminate Indirect left recursion: where A → Bα and B → Aβ can use same approach as for direct recursion. replace B in A → Bα & its new nonterminal symbol say B'.
4. Repeat step 2 & 3 as necessary: if grammar still contains left recursion repeat process
5. Reorder production: after eliminating left recursion, you may need to reorder productions in grammar to ensure they are in correct order of parsing.

Amar KRISH

Teacher's Signature _____

Program:

```
#include <stdio.h>
#include <string.h>
#define MAX_RULES 100
#define MAX_SYMBOLS 100
char rules[MAX_RULES][MAX_SYMBOLS]
int nRules = 0;
void eliminateLR(char nonterm){
    int i, j, k;
    char newnt = nonterm + 1;
    for(int i=0; i<nRules; i++) { if rules[i][0] == nonterm)
{ char alpha[max symbols], beta[max_symbol];
    int alphalen = 0; betalen = 0;
    for(k=2; rules[i][k] != '\0'; k++){
        alpha[alphalen++] = rules[i][k];
    }
    alpha[alphalen] = '\0';
    for(k=2; rules[i][k] != '\0'; k++){
        beta[betalen++] = rules[i][k];
        beta[betalen] = '\0';
    strcpy(rules[i], "");  strcpy(rules[j], "");
    printf(rules[i]"%c -> %s%c' \n", nonterm, beta, nonterm);
    printf(rules[j], "%c' -> %s%c' | epsilon \n",
            newnonterminal, alpha, newnonterminal);
    rules++; else printf("can't be reduced\n");
    return; printf(" Is not left recursive.\n");
}
return;
}
int main() { int string; char input[100]; A[50], a[50];
    printf("Enter the productions:");
    scanf("%s -> %s", A, a);
    input[A,a] = A + "->" + a;
    eliminator(input[A,a]);
    printf("After eliminating Left Recursion are (eliminator(input));
}
```

PAGE NO.:
DATE.: / / 20

Error and remedial action :

error: Buffer overflow : program consumes max length of production rule & max no. of production rules are fixed.

Remedy : program could dynamically allocate memory for rules & symbols.

Con. Results :

```
Enter the productions: E->E+E|T
The productions after eliminating Left
    Recursion are:
E->+EE'
E'->TE'
E->ε
```

```
Enter the productions: S : ( L ) | a

 L : L , S | S
The productions after eliminating Left
    Recursion are:
S -> ( L )
   | a
 L -> ( L ) L'
   | a L'
L' -> , S L'
   | ε
```

Conclusion: C program to eliminate left recursion from gramman implemented successfully

Course outcomes attained from matching program outcomes :

| Course Outcomes | Program Outcomes |
|---|---|
| CO-1. Generate & parser from formal specification CO-2: Generate top-down and bottom-up parser using predictive parsing SLR & LR passing techniques. | PO-1, PO-2, PO-3, PO4, PO5 |

Amar KRISH

Teacher's Signature _____

Date.....

Expt. No.....Exp.03..... Page. No.....

Program :

```
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
int main(){
    char str[50];
    int base=10;
    long long int num=0;
    int i, len, exp;
    scanf("%s", str);
    len=strlen(str);
    if(str[0]=='-') return 0;
    if(len>1&&str[0]=='0'){
        if(str[1]=='x'||str[1]=='X'){
            base=16; i=2;
        }else{
            base=8;
            i=1;
        }else{ i=0; }

while (i<len){ if(str[i]>='0'&&str[i]<='9')
  { num=num*base+(str[i]-'0');
  }else if(str[i]>='a'&&str[i]<='f'){
    num=num*base+(str[i]-'a'+10);
  }else if(str[i]>='A'&&str[i]<='F')
    { num=num*base+(str[i]-'A'+10);
    }
```

Teacher's Signature.....

```
else if (str[i] == 'e' || str[i] == 'E') {
    i++;
    exp = 0;
    while (i < len) {
        exp = exp * 10 + (str[i] - '0');
        i++;
    }
    exp = -exp;
} else if (str[i] == '+') {
    exp = 0;
    i++;
    while (i < len) {
        exp = exp * 10 + (str[i] - '0');
        i++;
    }
} else { printf("Invalid constant format\n");
        return 0;
    }
num *= pow(10, exp);
break;
} else {
    printf("Invalid\n");
    return 0;
} i++;
}
printf("%d\n", num);
return 0;
}
```

Date.....

Expt. No.....4.....

Page. No.....

Program :

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX_RULES 100
#define MAX_PROD 100
#define MAX_SYMBOLS 20
typedef struct {  char lhs[MAX_SYMBOLS];
                  char rhs[MAX_PROD][MAX_SYMBOLS];
                  int num_prods;
               } Rule;
// nonterminal (uppercase letter)
int is_nt(char symbol) {
    return symbol >= 'A' && symbol <= 'Z'; }
void eliminate_lr(Rule rules[], int num_rules) {
    for (int i=0; i<num_rules; i++) {
        for (int j=0; j<rules[i].num_prods; j++)
        { char fs = rules[i].rhs[j][0];
          if ( is_nt(fs) && fs == rules[i].lhs[0])
          { char new_nt = rules.lhs[0]+1;
            Rule new_r = { num_prods = 0};
            strcpy(new_rule.lhs, &new_nt);
        for (int k=0; k<rules[i].num_prods; k++)
        { char fs = rules[i].rhs[k][0];
        if (is_nt(fs) && fs == rules[i].lhs[0])
          { new_rule.num_prods++;
          } else {
              rules[i].num_prods++;
          }
    }
}
```

Teacher's Signature.....

```
        sprintf(rules[i].rhs[rules[i].num_prods], "%c%c",
                new_nt, '\n');
        rules[i].num_prods++;
    }
}
}

int main(){
    Rule rules[];
    printf("Enter the productions:\n");
    scanf("%s
    while(1){
        Rule rule;
        memset(&rule, 0, sizeof(rule));
        if (num_rules >= MAX_RULES) {
            printf("Too many rules");
            return 1;
        }
        printf("%d: ", num_rules+1);
        if(scanf("%s", rule.lhs) == EOF) break;
        strcpy(rule.rhs[rule.num_prods], prod_str,
               MAX_LEN);
        rule.num_prods;
    }
    eliminate_lr(rules, num_rules);
    printf("\n rules after eliminating left recursion are:");
    for(int i=0; i<num_rules; i++) {
        printf("%s :", rules[i].lhs);
        for(int j=0; j<rules[i].num_prods; j++){
            printf("%s", rules[i].rhs[j]);
            if(j<rules[i].num_prods-1);
                printf(" | ");
        }
        printf("\n");
    } return 0;
}
```

Date.....

Expt. No. 05 ..... Experiment No. 05 Page. No. 01

**Aim:** Design Recursive descent parser for the grammar. $E \rightarrow +EE \mid -EE \mid a \mid b$

**Tools:**

| Sr. | Tools | Specification | Qty. |
|---|---|---|---|
| 1 | C Compiler | Turbo, GCC, or any | - |
| 2 | Computer System | Windows, 4GB RAM 1 TB HDD | 1 |

**Theory:**

A recursive descent parser is a top-down parsing technique where a parse tree is built from top & constructed down to leaves. this type of parser starts with highest level of grammar & recursively expands non-terminals until a terminal symbol is reached.

- To implement this grammar using recursive descent parser, we need to follow steps to implement parser:

1. Analyze grammar, identify terminals, non-terminals production rules
2. write function to parse each non-terminal
3. Use tokenizer to convert input string into list of tokens
4. Pass list tokens as input, function should return whether string can be derived from grammar or not.

Teacher's Signature.....

Date.....

Expt. No.....

Date.....

Page. No.....

Program :

```
#include<stdio.h>
#include <stdbool.h>
#include <string.h>
bool parse_E (char *input, int *index);
bool parse_EE (char *input, int *index);
bool parse_E (char *input, int *index) {
    if (input[*index] == 'a' || input [*index]
        == 'b') {
        (*index)++;
        return true;
    } else if (input[*index] == '*' || input [*index]
        == '_') { (*index)++;
        return parse_EE (input, index) &&
    parse_E (input, index);
                }
        else { return false; } }    //end
bool parse_EE (char *input, int *index) {
    if (input [*index] == '+' || index == '_') {
        (*index)++;
        return parse_E (input, index); }
    else { return false; }
                }

int main () {
    char input[100];
    int index = 0;
    printf ("Enter an input string: \n ");
    scanf ("%s", input);
```

Teacher's Signature.....

```
bool success = parse_E(input, &index);
if(success && index == strlen(input))
  {
    printf("The input string can be accepted
            using grammar");
  }
else {
    printf("The input string cannot be accepted
            from grammar.\n");
    }
return 0;
}
```

Output / Result :

Date.....

Expt. No.....

Date.....

Page. No.....

Error & remedial action :

1. Program assumes input string contains only symbols 'a' 'b' '+' '-', if contain any other symbol, program may produce error.

Remedial Action: program can be manipulated to check each symbol that is terminals.

Conclusion: The program to implement recursive descent parser for grammar designed successfully

Course Outcome attained & mapping with program outcomes :

| Course Outcomes | Program Outcomes |
|---|---|
| CO-1 | PO1, PO2, PO3 |
| CO-2 | PO4, PO5 |
| | |
| | |
| | |

Teacher's Signature.....

Date.....

Page. No.....

Expt. No. 06 ..... Experiment No. 06

Aim: Design a program for basic calculator using YACC or BISON.

Tools:

| Sr No. | Tools | Specs | Qty |
|---|---|---|---|
| 01 | C compiler | YACC or BISON | 1 |
| 02 | Comp. System | Windows, 4GB RAM 1TB HDD | |

Theory: YACC (Yet Another Compiler Compiler) & BISON are tools that generate parsers (programs analyze structure of text or code) based on formal desc of language grammar.
- commonly used in development of compilers, interpreters, & other language-based s/w.

Syntax:

```
%{
    // declaration section
%}
%%
% token definition
  grammar declaration
%%
int main() {
        yyparse();
        return 1;
        }
```

Teacher's Signature.....

Result:

Date.....

Expt. No.....

Page. No.....

Program:

```
%{  #include<stdio.h>
    #include<lex.yy.c> int yylex();
    int yyerror();
%}
%%
%left '+' '-';
%left '*' '/';
expression :
 |expression '*' expression { $$ = $1 * $3;
                              printf("*: %d\n", $$); };
 | expression '+' expression { $$ = $1 + $3;
                              printf("+: %d\n", $$); };
 | expression '-' expression { $$ = $1 - $3;
                              printf("-: %d\n", $$); };
 | expression '/' expression { $$ = $1 / $3;
                              printf("/: %d\n", $$); };
%%
int main(){
yyparse();
return 1;
}
```

Error & Remedial Action:

1. Parsing error for invalid input expression
   yyerror() will generate error

Remedial action: print appropriate error message.

Teacher's Signature.....

Date.....

Expt. No. 06 Page. No.....

Conclusion: The program for basic calculator using YACC or BISON designed successfully

Course Outcomes Attained & mapping with program outcomes:

| Course Outcomes | Program Outcomes |
|---|---|
| CO1 | PO1 PO2 PO3 PO4 PO5, PO7 |
| CO3 | |

Teacher's Signature.....

Date.....

Expt. No.....07..... Experiment No. 07 Page. No.....

Aim: Design a program to evaluate postfix expression.

| Tools: | Specs. | Qty. |
|---|---|---|
| 1 - C compiler | - Turbo, gcc, any | - -- |
| 2 - Computer System | - Windows, 4GB RAM | - 01 |

Theory:

- Postfix Notation: way of writing arithmetic expressions in which each operator follows its Operands. Ex. "3 + 4" => "3 4 +"

- Steps to evaluate postfix expression: Ex. "3 4 +"

1. Scan Expression from left to right:
   - Push 3 onto stack (stack: 3)
   - Push 4 onto stack (stack: 3 4)
   - Pop 3 & 4 from stack, add them together (4 + 3 = 7) & push result (7) back onto stack
2. Return expression is in top element of stack, which is 7.

Teacher's Signature.....

Date.....

Expt. No.....

Page. No.....

Program :

```
#include<stdio.h>
#include<ctype.h>
#define MAXSTACK 100
#define POSTFIXSIZE 100
int stack[MAXSTACK];
int top = -1;
void push(int item)
{ if (top >= MAXSTACK-1) {
    printf("stack overflow");
    return; }
  else { top = top + 1; stop[top] = item; } }

int pop() { int item;
            if (top < 0) {
              printf("stack under flow");
            } else { item = stack[top];
                     top = top - 1; return item;
            } }

void EvalPostfix(char postfix[])
{   int i;
    char ch;
    int val;
    int A,B;
    for(i=0; postfix[i] != '\0'; i++) {
          ch = postfix[i];
    if(isdigit(ch)) {
          push(ch-'0');
    }
```

Teacher's Signature.....

```
else if(ch== '+' || ch = '-' || ch == '*' || ch== '/')
  {
     A = pop();
     B = pop();
     switch (ch)
     { case '*' : val = B * A; break;
       case '/' : val = B / A; break;
       case '+' : val = B + A; break;
       case '-' : val = B - A; break;
     }
     push (val);
  }
  }
  printf("\n Result of expression : %d \n", pop());
}
int main() {  int i;
              char postfix[POSTFIXSIZE5];
  printf("\n Enter postfix expression, \n press right
          parenthesis ')' for end expression : ");
  for (i=0 ; i<= POSTFIXSIZE - 1 ; i++) {
     scanf("%c", &postfix[i]);
     if(postfix[i] == ')') { break; } }
     EvalPostfix(postfix);
     return 0;
  }
```

Result:

Date.....

Expt. No.....07.....

Page. No.....03.....

Error & Remedial Action :

Error : Divide by zero

Remedial : need to apply proper error message for divide by zero.

Conclusion : The program to evaluate postfix expression designed successfully.

Course Outcomes attained & mapping with Program o/c.

| CO's | PO's |
|---|---|
| CO-1 | PO-1, PO-2, PO-3, PO-4 |

Teacher's Signature.....