**PRACTICAL 1**

**AIM:**

Implement a lexical analyzer for a subset of C using LEX Implementation should support Error handling.

**IMPLEMENTATION:**

* lex <filename with .l extension>
* gcc <newly created .c file> -o <file name for exe file>
* <filename of exe file>

In this case, create an extra text file named abc.txt which will contain some C code to work as input for lexical analysis.

**CODE:**

%%

"#" {printf("\n %s \t Preprocessor",yytext);}

"main"|"printf"|"scanf" {printf("\n%s\tfunction",yytext);}

"if"|"else"|"int"|"unsigned"|"long"|"char"|"switch"|"case"|"struct"|"do"|"while"|"void"|"for"|"float"|"continue"|"break"|"include" { printf("\n%s\tKeyword",yytext); }

[\_a-zA-Z][\_a-zA-Z0-9]\* {printf("\n%s\tIdenifier",yytext);}

"+"|"/"|"\*"|"-" {printf("\n%s\tOperator",yytext);}

"="|"<"|">"|"!="|"=="|"<="|">=" {printf("\n%s\tRelational Operator",yytext);}

"%d"|"%s"|"%c"|"%f" {printf("\n%s\tTokenizer",yytext);}

"stdio.h"|"conio.h"|"math.h"|"string.h"|"graphics.h"|"dos.h" {printf("\n%s\tHeader File",yytext);}

";"|"," {printf("\n%s\tDelimiter",yytext);}

"("|")" {if(strcmp(yytext,"(")==0)

{

printf("\n%c\tOpening Parenthesis",yytext[0]);

}

else

{

printf("\n%c\tClosing Parenthesis",yytext[0]);

}

;}

"{" {printf("\n%s\tStart Of Function/Loop",yytext);}

"}" {printf("\n%s\tEnd of Function",yytext);}

%%

int yywrap(void)

{

return 1;

}

int main()

{

int i;

FILE \*fp;

fp=fopen("abc.txt","r");

if(fp==NULL)

{

printf("Unable To Open File");

}

else

{

yyin=fp;

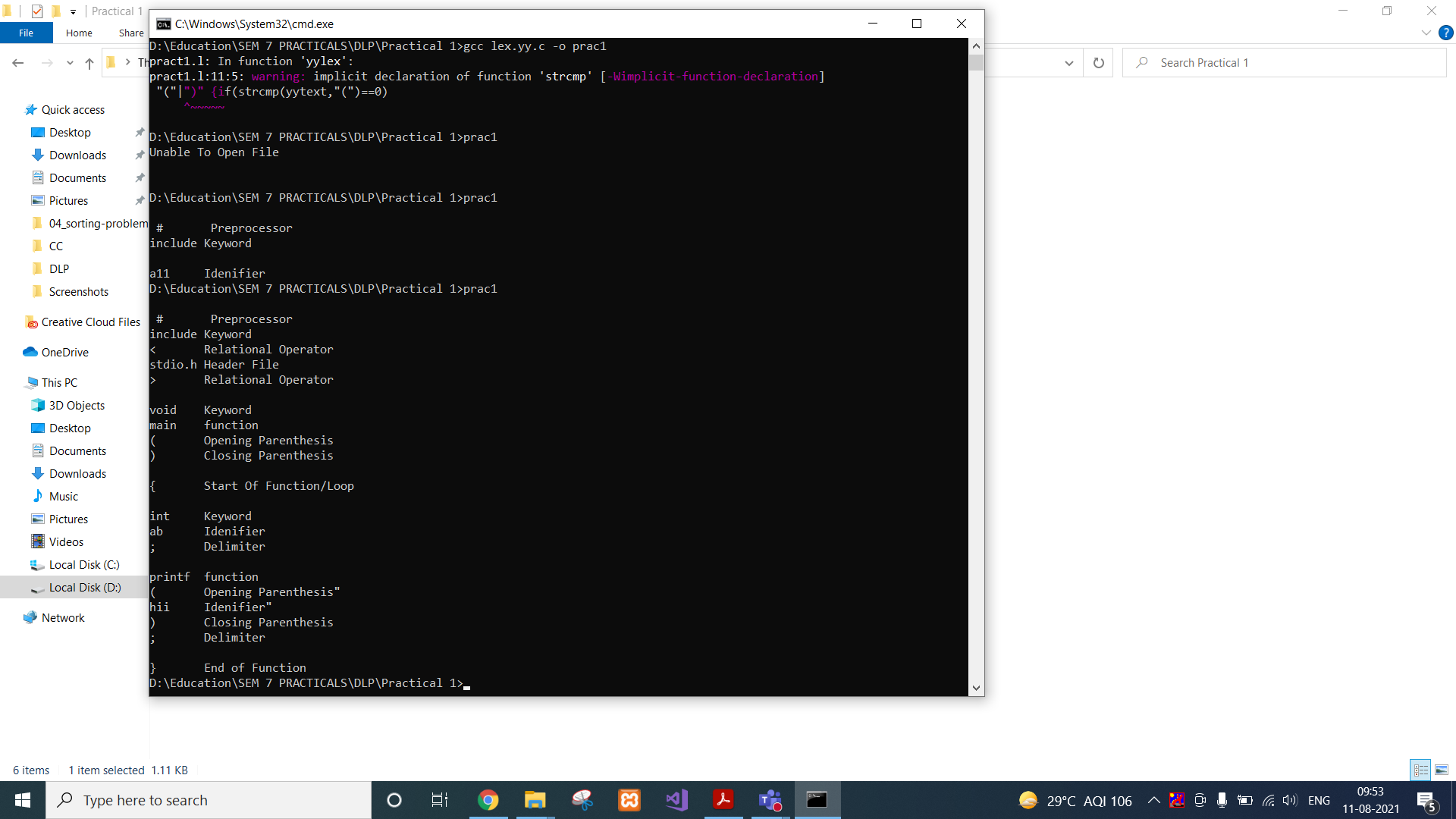
}

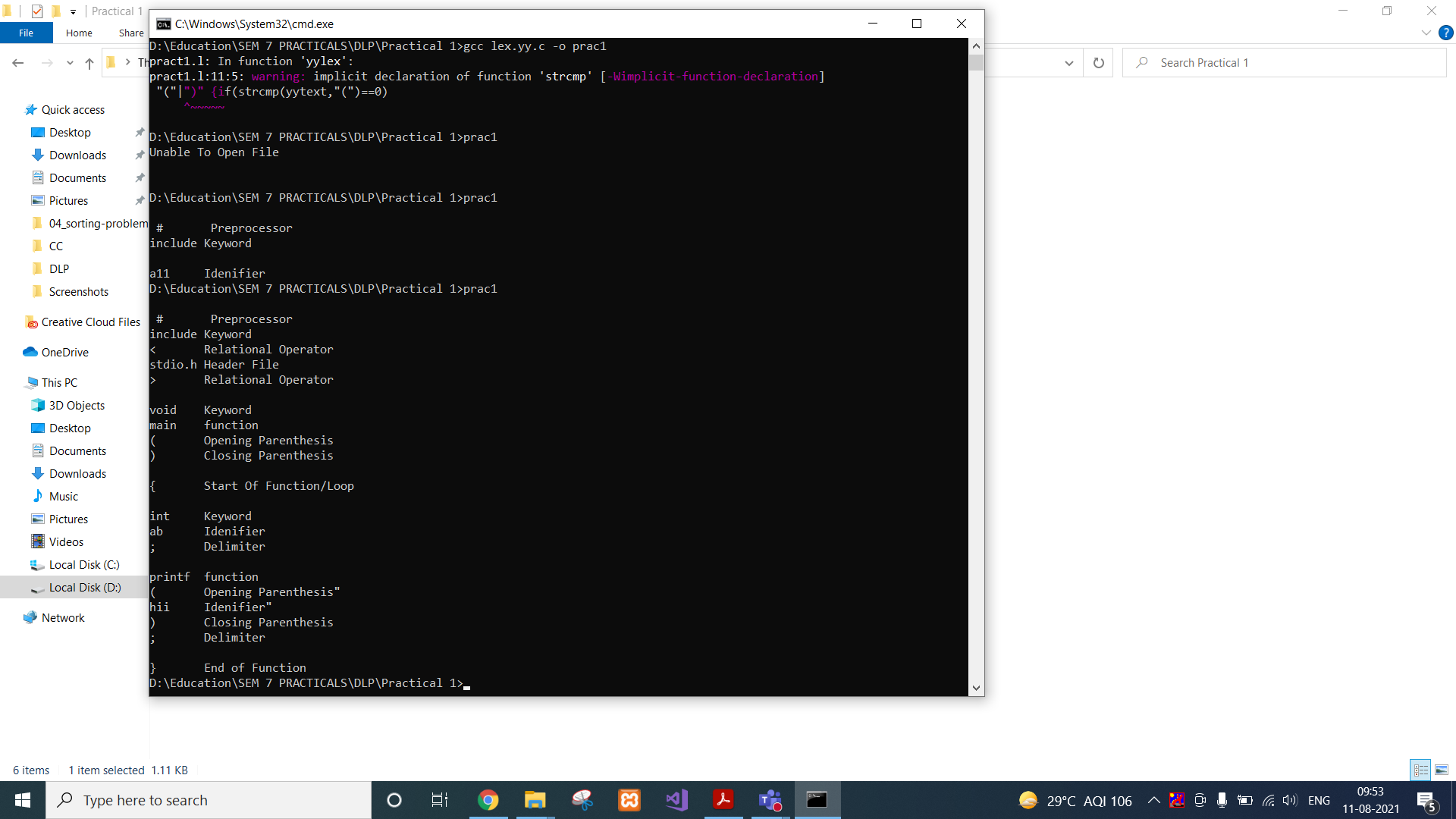
yylex();

return 0;

}

**OUTPUT:**





**CONCLUSION:**

In this practical, we learnt about lex files and implemented a program for lexical analysis.

**PRACTICAL 2**

**AIM:**

Implement a lexical analyzer for identification of numbers.

**IMPLEMENTATION:**

* lex <filename with .l extension>
* gcc <newly created .c file> -o <file name for exe file>
* <filename of exe file>

**CODE:**

bin (0|1)+

char [A-Za-z]+

digit [0-9]

oct [0-7]

dec [0-9]\*

float {digit}+("."{digit}+)

exp {digit}+("."{digit}+)?("E"("+"|"-")?{digit}+)?

hex [0-9a-fA-F]+

%%

{bin} {printf("\n %s= it is a binary number",yytext);}

{char} {printf("\n %s=it is a char",yytext);}

{oct} {printf("\n %s=it is a octal number",yytext);}

{digit} {printf("\n %s=it is a digit",yytext);}

{dec} {printf("\n %s=it is a decimal",yytext);}

{float} {printf("\n %s=it is a float",yytext);}

{exp} {printf("\n %s=it is a exp",yytext);}

{hex} {printf("\n %s=it is a hex",yytext);}

%%

int yywrap()

{

return 1;

}

int main()

{

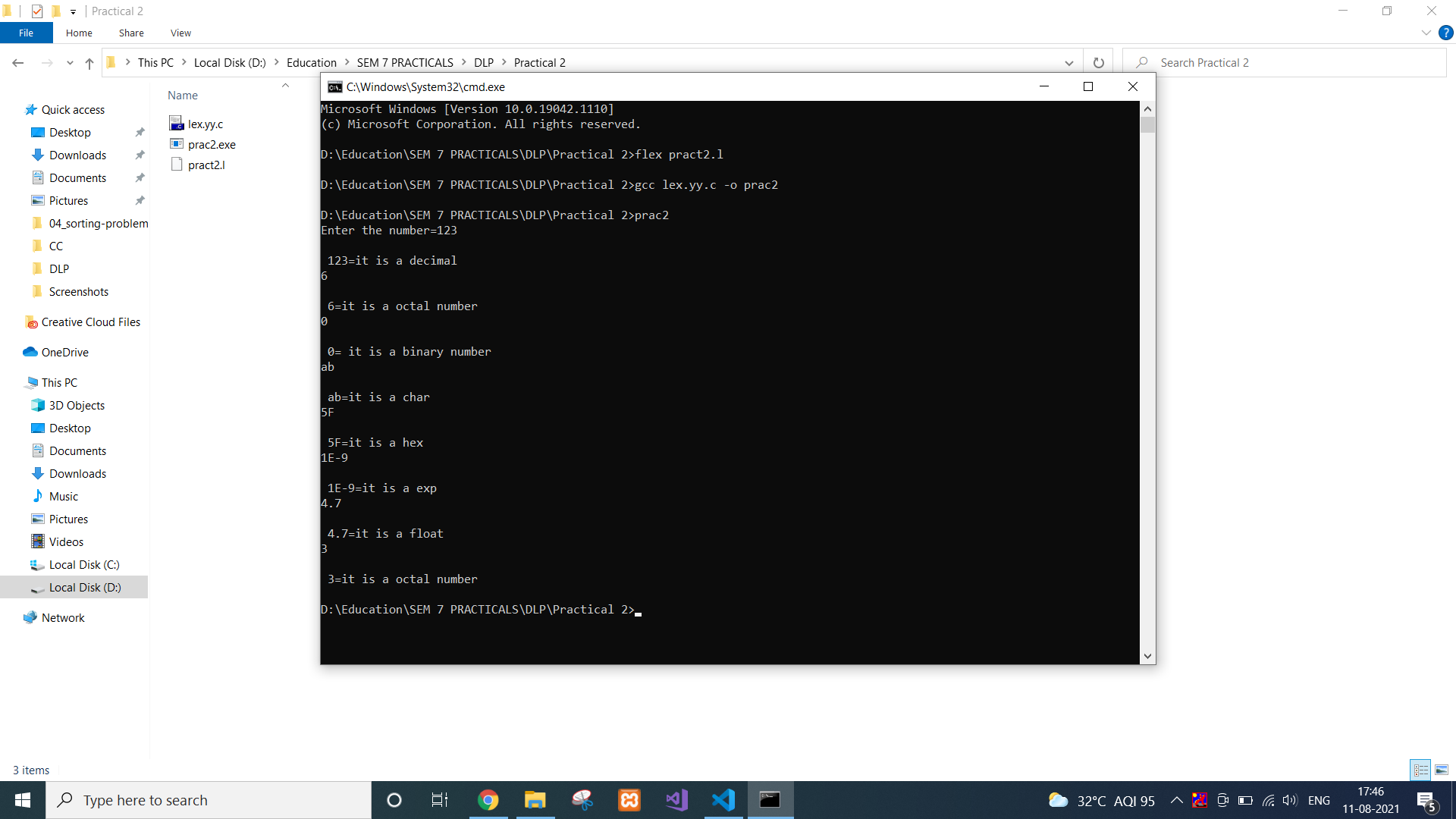
printf("Enter the number=");

yylex();

return 0;

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt about lexical analysis for numbers and characters.

**PRACTICAL 3**

**AIM:**

Write an ambiguous CFG to recognize an infix expression and implement a parser that recognizes the infix expression using YACC.

**IMPLEMENTATION:**

* yacc <filename with .y extension>
* gcc <newly created .c file> -o <file name for exe file>
* <filename of exe file>

**CODE:**

%{

/\*\*\* Auxiliary declarations section \*\*\*/

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

/\* Custom function to print an operator\*/

void print\_operator(char op);

/\* Variable to keep track of the position of the number in the input \*/

int pos=0;

char p;

%}

/\*\*\* YACC Declarations section \*\*\*/

%token NUM

%left '+'

%left '\*'

%%

/\*\*\* Rules Section \*\*\*/

start : expr '\n' {exit(1);}

;

expr: expr '+' expr {print\_operator('+');}

| expr '\*' expr {print\_operator('\*');}

| '(' expr ')'

| NUM {printf("%c ",p);}

;

%%

/\*\*\* Auxiliary functions section \*\*\*/

void print\_operator(char c){

switch(c){

case '+' : printf("+ ");

break;

case '\*' : printf("\* ");

break;

}

return;

}

yyerror(char const \*s)

{

printf("yyerror %s",s);

}

yylex(){

char c;

c = getchar();

p=c;

if(isdigit(c)){

pos++;

return NUM;

}

else if(c == ' '){

yylex(); /\*This is to ignore whitespaces in the input\*/

}

else {

return c;

}

}

main()

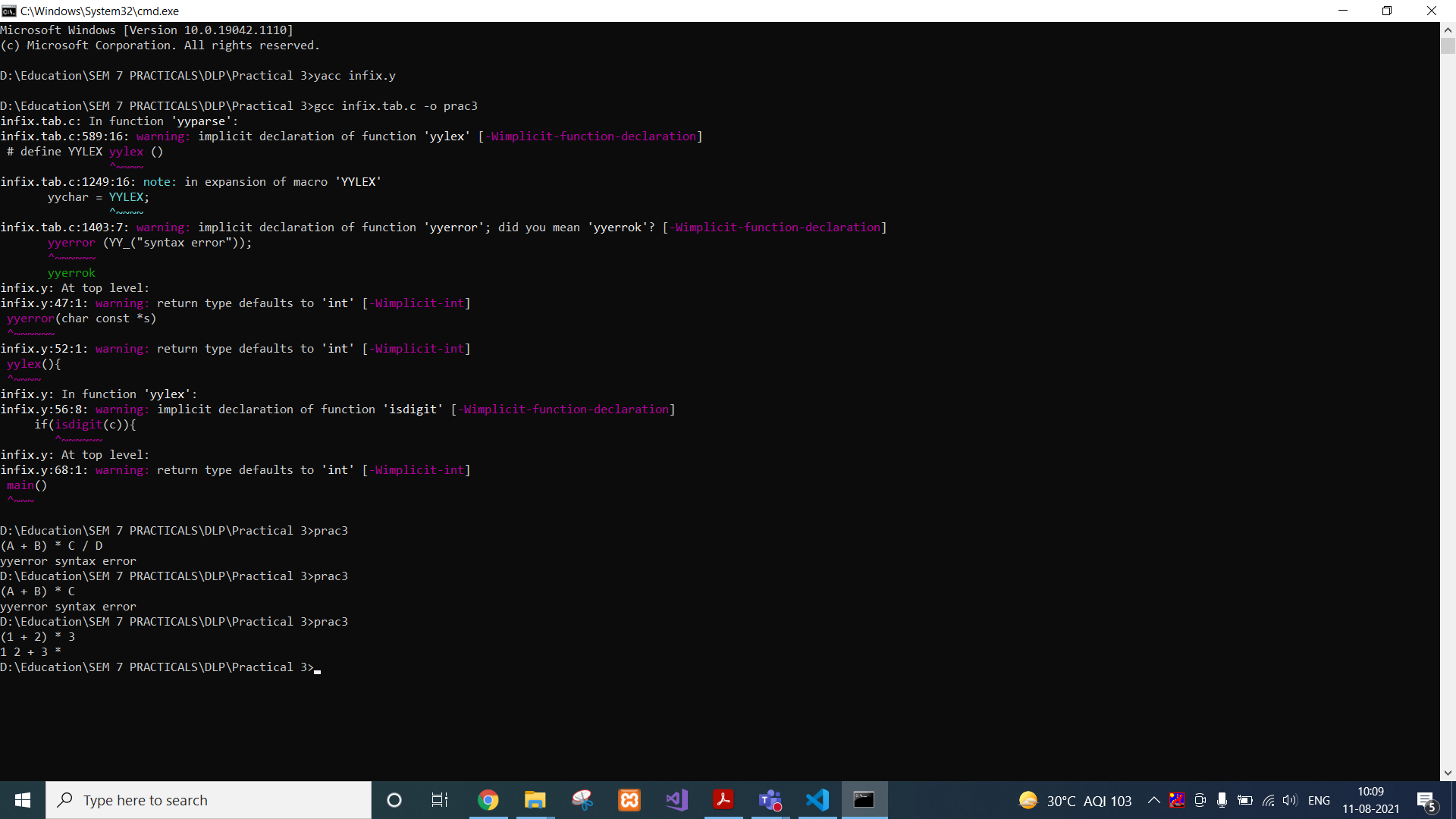
{

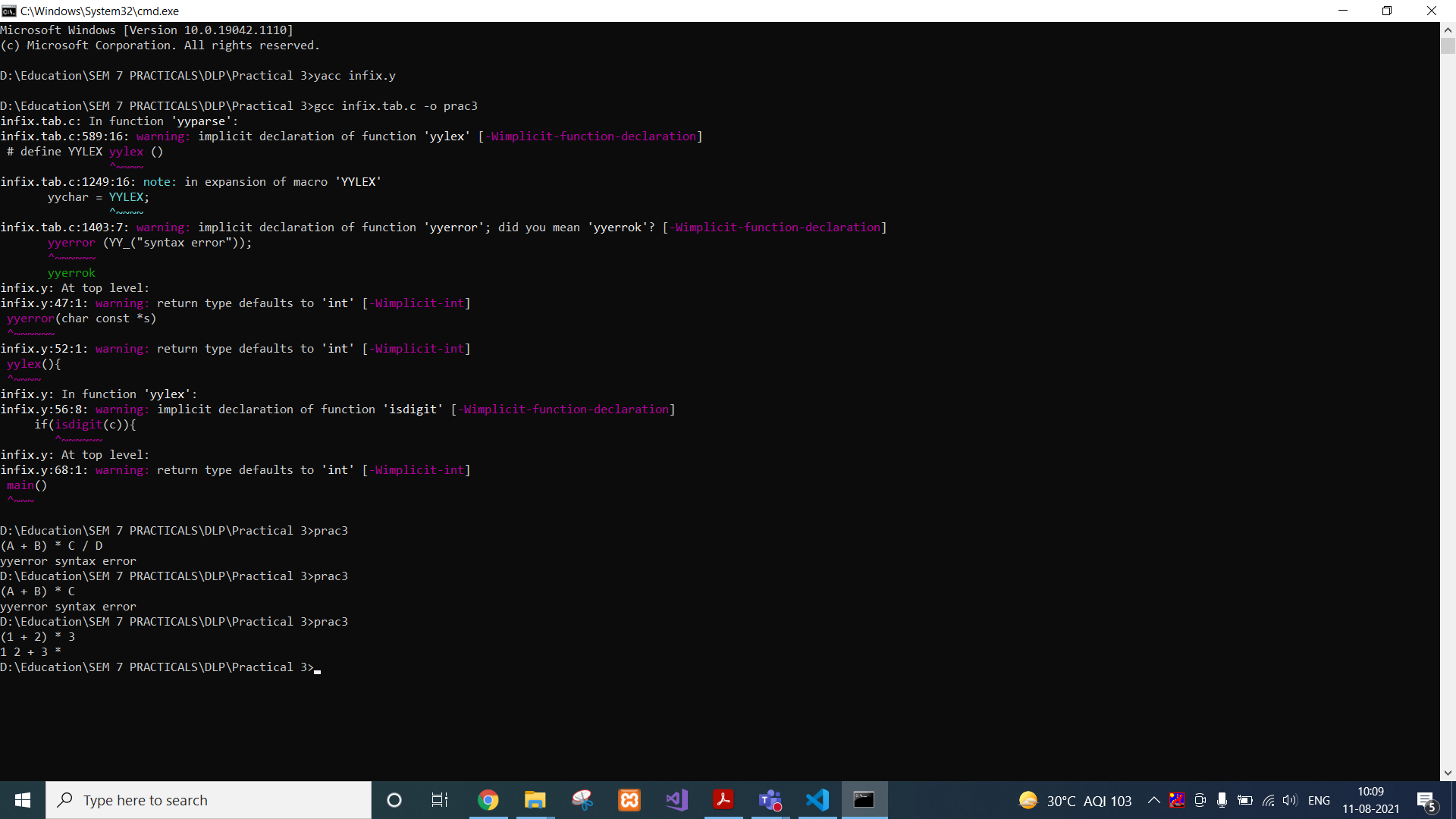
yyparse();

return 1;

}

**OUTPUT:**





**CONCLUSION:**

In this practical, we learnt about yacc and performed infix to postfix conversion.

**PRACTICAL 4**

**AIM:**

Implement a Calculator using LEX and YACC.

**IMPLEMENTATION:**

* lex <filename with .l extension>
* yacc <filename with .y extension>
* gcc <newly created .c file from yacc> -o <file name for exe file>
* <filename of exe file>

**CODE:**

*Lex File:*

DIGIT [0-9]

%option noyywrap

%%

{DIGIT} { yylval=atof(yytext); return NUM;}

\n|. {return yytext[0];}

*Yacc File:*

%{

#include<ctype.h>

#include<stdio.h>

#define YYSTYPE double

%}

%token NUM

%left '+' '-'

%left '\*' '/'

%%

S : S E '\n' { printf("Answer: %g \nEnter:\n", $2); }

| S '\n'

|

| error '\n' { yyerror("Error: Enter once more�\n" );yyerrok; }

;

E : E '+' E { $$ = $1 + $3; }

| E'-'E { $$=$1-$3;}

| E'\*'E {$$=$1\*$3;}

| E'/'E {$$=$1/$3;}

| NUM

;

%%

#include "lex.yy.c"

int main()

{

printf("Enter the expression: ");

yyparse();

}

yyerror (char \* s)

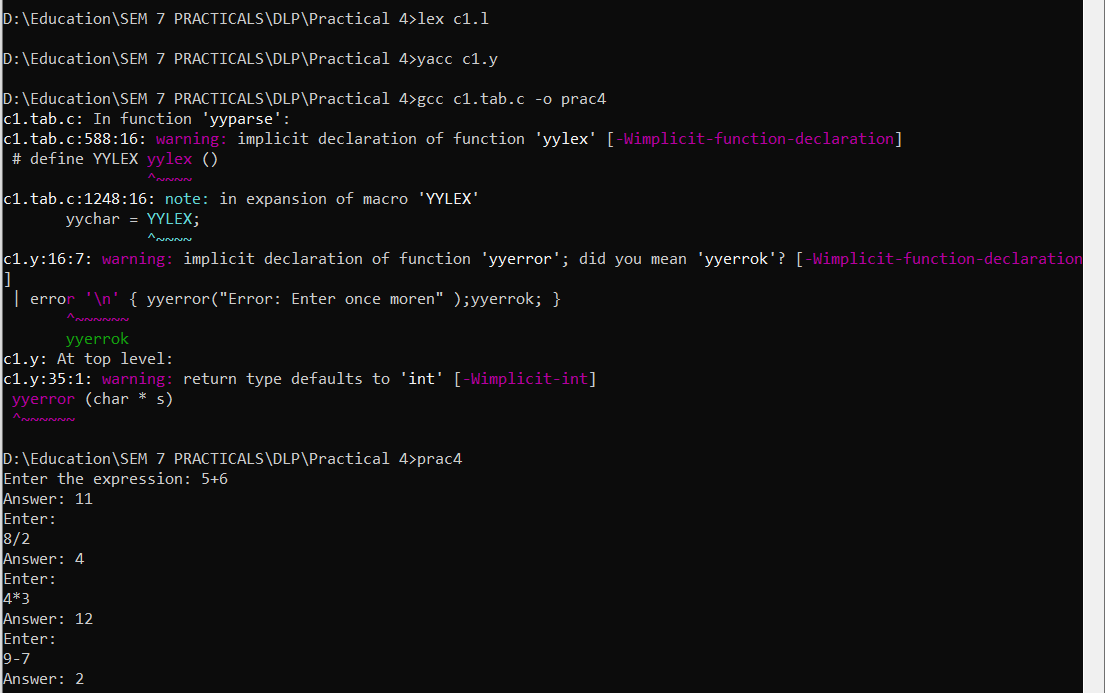
{

printf ("% s \n", s);

exit (1);

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt implemented a calculator using lex and yacc which takes expression as input and perform basic arithmetic operations.

**PRACTICAL 5**

**AIM:**

Implementation of Syntax Tree.

**IMPLEMENTATION:**

* gcc <newly created .c file> -o <file name for exe file>
* <filename of exe file>

In this case, create a syntax.txt file as input for the executable which will contain following statements.

t1=a+b

t2=c-d

t3=e+t2

t4=t1-t3

**CODE:**

#include<conio.h>

#include<stdio.h>

int main()

{

FILE \*fp;

int i=0,j=0,k,l,row,col,s,x;

char a[10][10],ch,main[50],search;

//clrscr();

fp=fopen("syntax.txt","r+");

while((ch=fgetc(fp))!=EOF)

{

if(ch=='\n')

{

row=i;

col=j;

j=0;

i++;

}

else

{

a[i][j]=ch

; j++;

}

}

printf("\n");

for(k=0;k<row+1;k++)

{

for(l=0;l<col;l++)

{

printf("%c",a[k][l]);

}

printf("\n");

}

i=0;

s=0;

for(k=0;k<row+1;k++)

{

main[i]=a[k][1];

i++;

if(a[k][3]=='t')

{

search=a[k][4];

for(l=0;l<i;l++)

{

if(main[l]==search)

{

main[i]=main[l];

i++;

break;

}

}

main[i]=a[k][5];

s=5;

i++;

}

else

{

main[i]=a[k][3];

// printf("\n%c",main[i]);

i++;

main[i]=a[k][4];

// printf(",%c\n",main[i]);

s=4;

i++;

}

s++;

if(a[k][s]=='t')

{

s++;

search=a[k][s];

for(l=0;l<i;l++)

{

if(main[l]==search)

{

main[i]=main[l];

i++;

break;

}

}

}

else

{

main[i]=a[k][s];

i++;

}

}

for(x=i-1;x>=0;x=x-4)

{

printf("\ntt%c: root->%c ",main[x-3],main[x-1]);

if(main[x-2]>48 &&main[x-2]<59)

printf("lc->t%c ",main[x-2]);

else

printf("lc->%c ",main[x-2]);

if(main[x]>48 &&main[x]<59)

printf("rc->t%c ",main[x]);

else

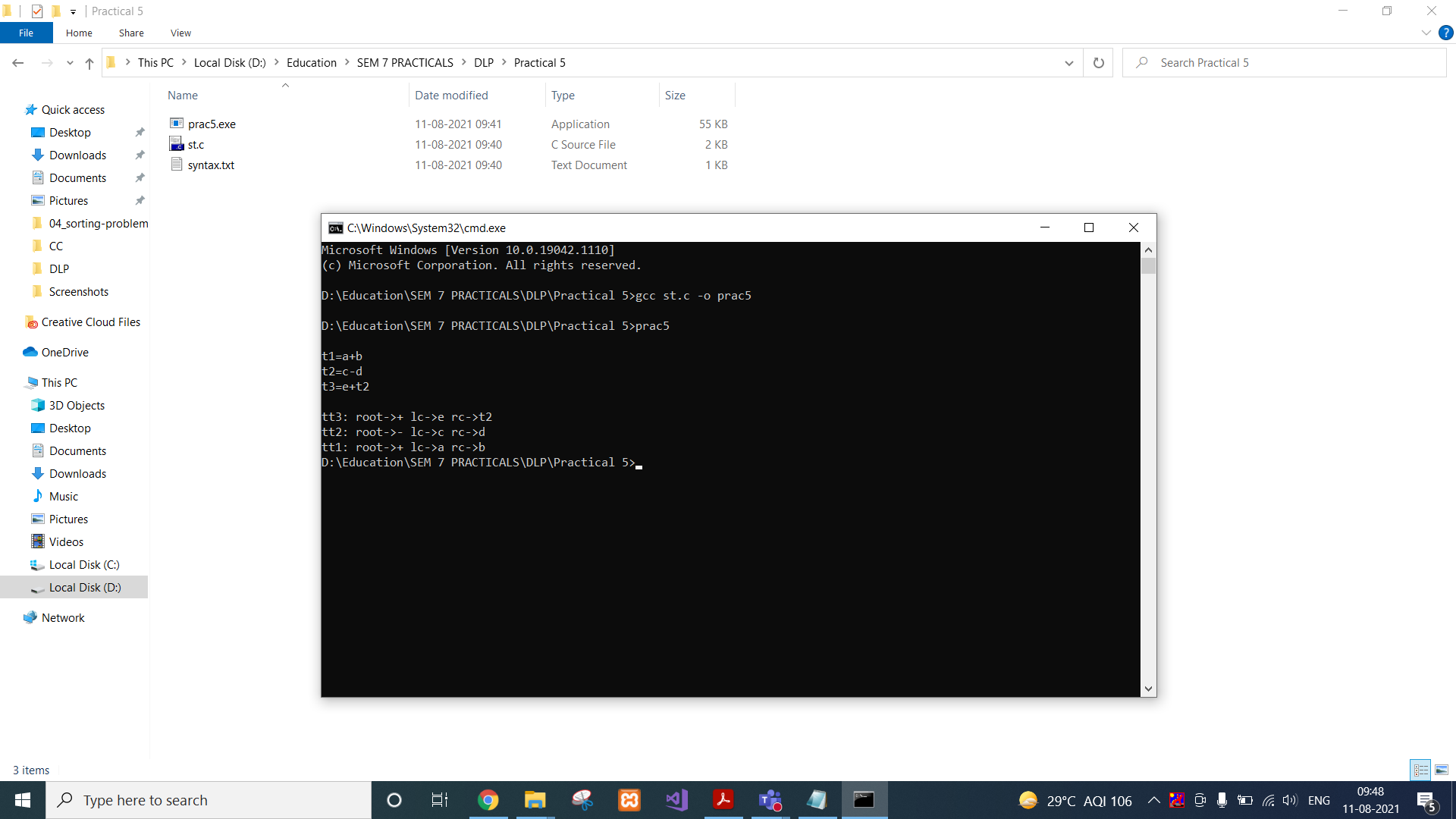
printf("rc->%c ",main[x]);

}

getch();

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt about syntax tree and implemented the concept using C.

**PRACTICAL 6**

**AIM:**

Implementation of Context Free Grammar.

**IMPLEMENTATION:**

* gcc <our .c file> -o <file name for exe file>
* <filename of exe file>

In this case, create a syntax.txt file as input for the executable which will contain following statements.

S aBaA

S AB

A Bc

B c

**CODE:**

//CFG

#include<stdio.h>

#include<string.h>

#include<conio.h>

int i,j,k,l,m,n=0,o,p,nv,z=0,t,x=0;

char str[10],temp[20],temp2[20],temp3[20];

struct prod

{

char lhs[10],rhs[10][10];

int n;

}pro[10];

void findter()

{

for(k=0;k<n;k++)

{

if(temp[i]==pro[k].lhs[0])

{

for(t=0;t<pro[k].n;t++)

{

for(l=0;l<20;l++)

temp2[l]='\0';

for(l=i+1;l<strlen(temp);l++)

temp2[l-i-1]=temp[l];

for(l=i;l<20;l++)

temp[l]='\0';

for(l=0;l<strlen(pro[k].rhs[t]);l++)

temp[i+l]=pro[k].rhs[t][l];

strcat(temp,temp2);

if(str[i]==temp[i])

return;

else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90)

break;

}

break;

}

}

if(temp[i]>=65 && temp[i]<=90)

findter();

}

int main()

{

FILE \*f;

// clrscr();

for(i=0;i<10;i++)

pro[i].n=0;

f=fopen("input.txt","r");

while(!feof(f))

{

fscanf(f,"%s",pro[n].lhs);

if(n>0)

{

if( strcmp(pro[n].lhs,pro[n-1].lhs) == 0 )

{

pro[n].lhs[0]='\0';

fscanf(f,"%s",pro[n-1].rhs[pro[n-1].n]);

pro[n-1].n++;

continue;

}

}

fscanf(f,"%s",pro[n].rhs[pro[n].n]);

pro[n].n++;

n++;

}

n--;

printf("\n\nTHE GRAMMAR IS AS FOLLOWS\n\n");

for(i=0;i<n;i++)

for(j=0;j<pro[i].n;j++)

printf("%s -> %s\n",pro[i].lhs,pro[i].rhs[j]);

while(1)

{

for(l=0;l<10;l++)

str[0]=NULL;

printf("\n\nENTER ANY STRING ( 0 for EXIT ) : ");

scanf("%s",str);

if(str[0]=='0')

printf("Exit");

// exit(1);

for(j=0;j<pro[0].n;j++)

{

for(l=0;l<20;l++)

temp[l]=NULL;

strcpy(temp,pro[0].rhs[j]);

m=0;

for(i=0;i<strlen(str);i++)

{

if(str[i]==temp[i])

m++;

else if(str[i]!=temp[i] && temp[i]>=65 && temp[i]<=90)

{

findter();

if(str[i]==temp[i])

m++;

}

else if( str[i]!=temp[i] && (temp[i]<65 || temp[i]>90) )

break;

}

if(m==strlen(str) && strlen(str)==strlen(temp))

{

printf("\n\nTHE STRING can be PARSED !!!");

break;

}

}

if(j==pro[0].n)

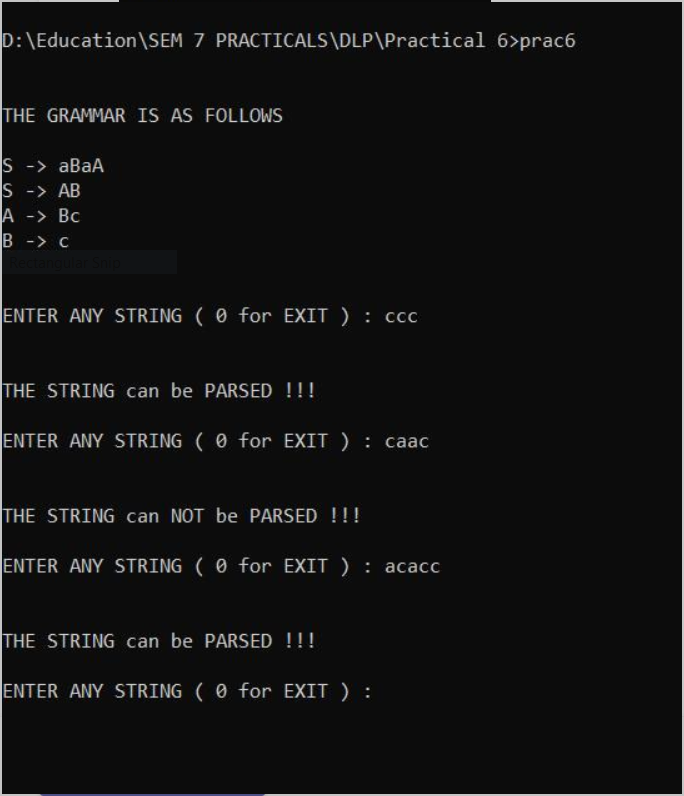
printf("\n\nTHE STRING can NOT be PARSED !!!");

}

getch();

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt about Context Free Grammar and implemented the concept using C.

**PRACTICAL 7**

**AIM:**

Design of a Predictive parser.

**IMPLEMENTATION:**

* gcc <our .c file> -o <file name for exe file>
* <filename of exe file>

**CODE:**

#include<stdio.h>

#include<conio.h>

#include<string.h>

void main()

{

char fin[10][20],st[10][20],ft[20][20],fol[20][20];

int a=0,e,i,t,b,c,n,k,l=0,j,s,m,p;

//clrscr();

printf("enter the no. of coordinates\n");

scanf("%d",&n);

printf("enter the productions in a grammar\n");

for(i=0;i<n;i++)

scanf("%s",st[i]);

for(i=0;i<n;i++)

fol[i][0]='\0';

for(s=0;s<n;s++)

{

for(i=0;i<n;i++)

{

j=3;

l=0;

a=0;

l1:if(!((st[i][j]>64)&&(st[i][j]<91)))

{

for(m=0;m<l;m++)

{

if(ft[i][m]==st[i][j])

goto s1;

}

ft[i][l]=st[i][j];

l=l+1;

s1:j=j+1;

}

else

{

if(s>0)

{

while(st[i][j]!=st[a][0])

{

a++;

}

b=0;

while(ft[a][b]!='\0')

{

for(m=0;m<l;m++)

{

if(ft[i][m]==ft[a][b])

goto s2;

}

ft[i][l]=ft[a][b];

l=l+1;

s2:b=b+1;

}

}

}

while(st[i][j]!='\0')

{

if(st[i][j]=='|')

{

j=j+1;

goto l1;

}

j=j+1;

}

ft[i][l]='\0';

}

}

printf("first pos\n");

for(i=0;i<n;i++)

printf("FIRS[%c]=%s\n",st[i][0],ft[i]);

fol[0][0]='$';

for(i=0;i<n;i++)

{

k=0;

j=3;

if(i==0)

l=1;

else

l=0;

k1:while((st[i][0]!=st[k][j])&&(k<n))

{

if(st[k][j]=='\0')

{

k++;

j=2;

}

j++;

}

j=j+1;

if(st[i][0]==st[k][j-1])

{

if((st[k][j]!='|')&&(st[k][j]!='\0'))

{

a=0;

if(!((st[k][j]>64)&&(st[k][j]<91)))

{

for(m=0;m<l;m++)

{

if(fol[i][m]==st[k][j])

goto q3;

}

fol[i][l]=st[k][j];

q3:l++;

}

else

{

while(st[k][j]!=st[a][0])

{

a++;

}

p=0;

while(ft[a][p]!='\0')

{

if(ft[a][p]!='@')

{

for(m=0;m<l;m++)

{

if(fol[i][m]==ft[a][p])

goto q2;

}

fol[i][l]=ft[a][p];

l=l+1;

}

else

e=1;

q2:p++;

}

if(e==1)

{

e=0;

goto a1;

}

}

}

else

{

a1:c=0;

a=0;

while(st[k][0]!=st[a][0])

{

a++;

}

while((fol[a][c]!='\0')&&(st[a][0]!=st[i][0]))

{

for(m=0;m<l;m++)

{

if(fol[i][m]==fol[a][c])

goto q1;

}

fol[i][l]=fol[a][c];

l++;

q1:c++;

}

}

goto k1;

}

fol[i][l]='\0';

}

printf("follow pos\n");

for(i=0;i<n;i++)

printf("FOLLOW[%c]=%s\n",st[i][0],fol[i]);

printf("\n");

s=0;

for(i=0;i<n;i++)

{

j=3;

while(st[i][j]!='\0')

{

if((st[i][j-1]=='|')||(j==3))

{

for(p=0;p<=2;p++)

{

fin[s][p]=st[i][p];

}

t=j;

for(p=3;((st[i][j]!='|')&&(st[i][j]!='\0'));p++)

{

fin[s][p]=st[i][j];

j++;

}

fin[s][p]='\0';

if(st[i][k]=='@')

{

b=0;

a=0;

while(st[a][0]!=st[i][0])

{

a++;

}

while(fol[a][b]!='\0')

{

printf("M[%c,%c]=%s\n",st[i][0],fol[a][b],fin[s]);

b++;

}

}

else if(!((st[i][t]>64)&&(st[i][t]<91)))

printf("M[%c,%c]=%s\n",st[i][0],st[i][t],fin[s]);

else

{

b=0;

a=0;

while(st[a][0]!=st[i][3])

{

a++;

}

while(ft[a][b]!='\0')

{

printf("M[%c,%c]=%s\n",st[i][0],ft[a][b],fin[s]);

b++;

}

}

s++;

}

if(st[i][j]=='|')

j++;

}

}

getch();

}

**CONCLUSION:**

In this practical, we learnt about first and follow and implemented predictive parser using C.

**PRACTICAL 8**

**AIM:**

Implementation of code generator.

**IMPLEMENTATION:**

* gcc <our .c file> -o <file name for exe file>
* <filename of exe file>

Content of Input1.txt:

a=b+c;

d=n+s;

p=q;

**CODE:**

// Pgm for Code generation by using simple code generation algorithm

#include<stdio.h>

#include<string.h>

struct table{

char op1[2];

char op2[2];

char opr[2];

char res[2];

}tbl[100];

void add(char \*res,char \*op1, char \*op2,char \*opr)

{

FILE \*ft;

char string[20];

char sym[100];

ft=fopen("result.asm","a+");

if(ft==NULL)

ft=fopen("result.asm","w");

printf("\nUpdating Assembly Code for the Input File : File : Result.asm ; Status [ok]\n");

//sleep(2);

strcpy(string,"mov r0,");

strcat(string,op1);

if(strcmp(opr,"&")==0)

{

//do nothing

}

else

{

strcat(string,"\nmov r1,");

strcat(string,op2);

}

fputs(string,ft);

if(strcmp(opr,"+")==0)

strcpy(string,"\nadd r0,r1\n");

else if(strcmp(opr,"-")==0)

strcpy(string,"\nsub r0,r1\n");

else if(strcmp(opr,"/")==0)

strcpy(string,"\ndiv r0,r1\n");

else if(strcmp(opr,"\*")==0)

strcpy(string,"\nmul r0,r1\n");

else if(strcmp(opr,"&")==0)

strcpy(string,"\n");

else

strcpy(string,"\noperation r0,r1\n");

fputs(string,ft);

strcpy(string,"mov ");

strcat(string,res);

strcat(string,", r0\n");

fputs(string,ft);

fclose(ft);

string[0]='\0';

sym[0]='\0';

}

main()

{

int res,op1,op2,i,j,opr;

FILE \*fp;

char filename[50];

char s,s1[10];

system("clear");

remove("result.asm");

remove("result.sym");

res=0;op1=0;op2=0;i=0;j=0;opr=0;

printf("\n Enter the Input Filename with no white spaces:");

scanf("%s",filename);

fp=fopen(filename,"r");

if(fp==NULL)

{

printf("\n cannot open the input file !\n");

return(0);

}

else

{

while(!feof(fp))

{

s=fgetc(fp);

if(s=='=')

{

res=1;

op1=op2=opr=0;

s1[j]='\0';

strcpy(tbl[i].res,s1);

j=0;

}

else if(s=='+'||s=='-'||s=='\*'||s=='/')

{

op1=1;

opr=1;

s1[j]='\0';

tbl[i].opr[0]=s;

tbl[i].opr[1]='\0';

strcpy(tbl[i].op1,s1);

j=0;

}

else if(s==';')

{

if(opr) // for 3 operand format ex: a=b+c;

{

op2=1;

s1[j]='\0';

strcpy(tbl[i].op2,s1);

}

else if(!opr) // for 2 operand format ex: d=a;

{

op1=1;

op2=0;

s1[j]='\0';

strcpy(tbl[i].op1,s1);

strcpy(tbl[i].op2,"&"); // simplifying the expr

strcpy(tbl[i].opr,"&"); //-------"--"----------

}

add(tbl[i].res,tbl[i].op1,tbl[i].op2,tbl[i].opr);

i++;

j=0;

opr=op1=op2=res=0;

}

else

{

s1[j]=s;

j++;

}

}

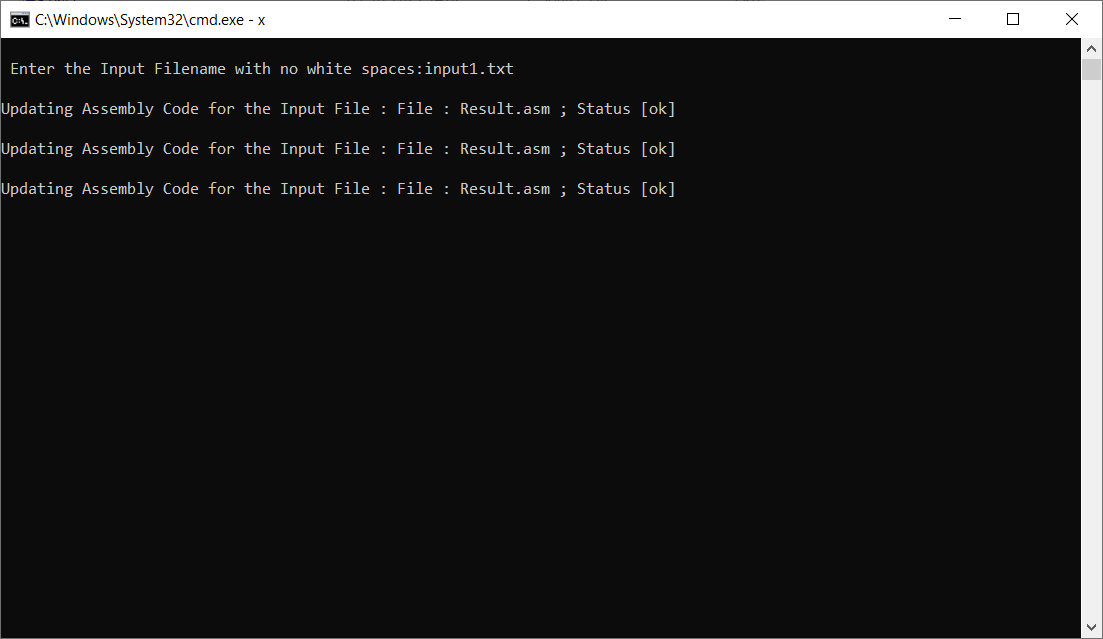
system("clear");

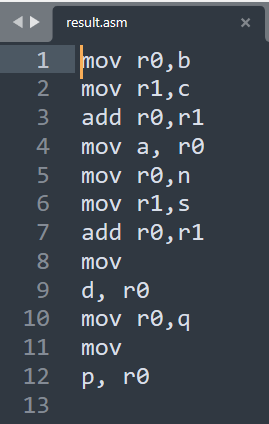
}

return 0;

}

**OUTPUT:**





**CONCLUSION:**

In this practical, we learnt about code generation and implemented the same using C.

**PRACTICAL 9**

**AIM:**

Implementation of code optimization for Common sub-expression elimination, Loop invariant code movement.

**IMPLEMENTATION:**

* gcc <our .c file> -o <file name for exe file>
* <filename of exe file>

**CODE:**

#include <stdio.h>

#include <conio.h>

#include <string.h>

struct op

{

char l;

char r[20];

}op[10], pr[10];

void main()

{

int a, i, k, j, n, z = 0, m, q;

char \*p, \*l;

char temp, t;

char \*tem;

//clrscr();

printf("enter no of values=");

scanf("%d", &n);

//n=5;

for (i = 0; i < n; i++)

{

printf("\t left: \t");

scanf(" %c", &op[i].l);

printf("\t right: \t");

scanf("%s", op[i].r);

}

/\*for (i = 0; i < n; i++)

{

printf("\n right: \t");

scanf("%s", op[i].r);

}\*/

printf(" intermediate Code\n");

for (i = 0; i < n; i++)

{

printf(" %c=", op[i].l);

printf(" %s\n", op[i].r);

}

for (i = 0; i < n - 1; i++)

{

temp = op[i].l;

for (j = 0; j< n; j++)

{

p = strchr(op[j].r,temp);

if (p)

{

pr[z].l= op[i].l;

strcpy(pr[z].r, op[i].r);

z++;

}

}

}

pr[z].l= op[n - 1].l;

strcpy(pr[z].r, op[n - 1].r);

z++;

printf("\n after dead code elimination \n");

for (k = 0; k < z; k++)

{

printf("%c = \t ",pr[k].l);

printf("%s \n",pr[k].r);

}

//sub expression elimination

for (m = 0; m < z; m++)

{ tem = pr[m].r;

for (j = m + 1; j < z; j++)

{ p = strstr(tem, pr[j].r);

if (p)

{

t = pr[j].l;

pr[j].l= pr[m].l;

for (i = 0; i < z; i++)

{

l= strchr(pr[i].r, t);

if (l){

a = l - pr[i].r;

//printf("pos: %d",a);

pr[i].r[a] = pr[m].l;

}

}

}

}

}

printf("eliminate common expression\n");

for(i=0;i<z;i++)

{

printf("%c\t =", pr[i].l);

printf("%s\n", pr[i].r);

}

// duplicate production elimination

for (i = 0; i< z; i++)

{

for (j = i + 1; j < z; j++)

{

q = strcmp(pr[i].r, pr[j].r);

if ((pr[i].l == pr[j].l) && !q)

{

pr[i].l = '\0';

//pr[i].r = "NULL";

strcpy( pr[i].r , "NULL");

}

}

}

printf("optimized code \n");

for (i = 0; i< z; i++)

{

if (pr[i].l != '\0')

{

printf("%c =", pr[i].l);

printf("%s \n", pr[i].r);

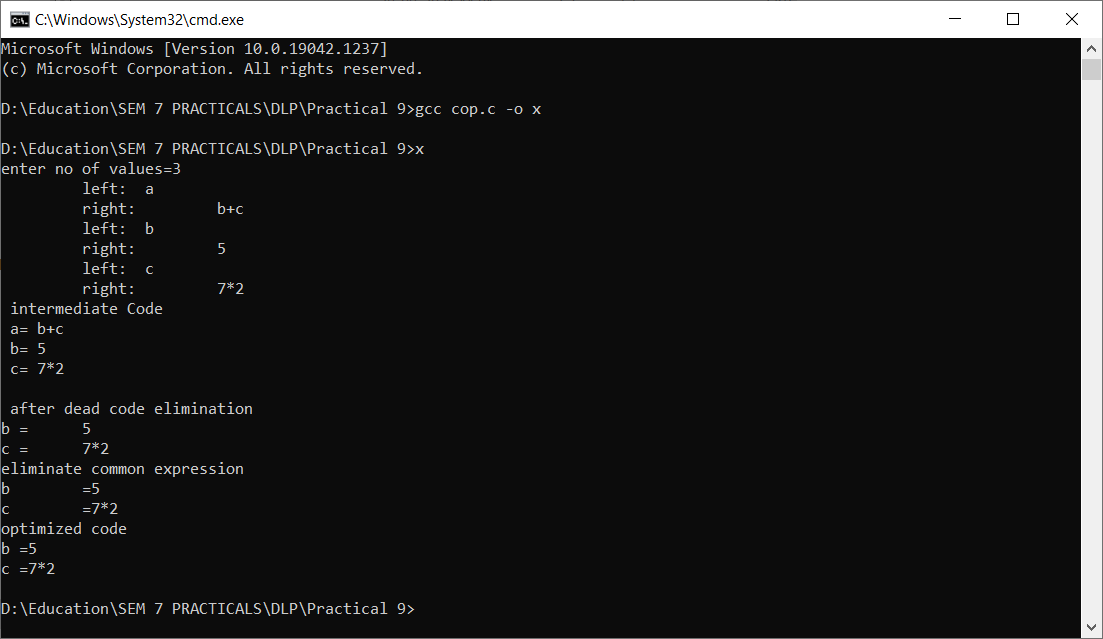
}

}

getch();

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt about code optimization and implemented the same using C.

**PRACTICAL 10**

**AIM:**

Implement Deterministic Finite Automata.

**IMPLEMENTATION:**

* gcc <our .c file> -o <file name for exe file>
* <filename of exe file>

**CODE:**

/\*

\* DFA Simulation in C

\*/

#include <stdio.h>

#include <stdlib.h>

struct node{

int id\_num;

int st\_val;

struct node \*link0;

struct node \*link1;

};

struct node \*start, \*q, \*ptr;

int vst\_arr[100], a[10];

int main(){

int count, i, posi, j;

char n[10];

//clrscr();

printf("=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=\n");

printf("Enter the number of states in the m/c:");

scanf("%d",&count);

q=(struct node \*)malloc(sizeof(struct node)\*count);

for(i=0;i<count;i++)

{

(q+i)->id\_num=i;

printf("State Machine::%d\n",i);

printf("Next State if i/p is 0:");

scanf("%d",&posi);

(q+i)->link0=(q+posi);

printf("Next State if i/p is 1:");

scanf("%d",&posi);

(q+i)->link1=(q+posi);

printf("Is the state final state(0/1)?");

scanf("%d",&(q+i)->st\_val);

}

printf("Enter the Initial State of the m/c:");

scanf("%d",&posi);

start=q+posi;

printf("=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=\n");

while(1){

printf("=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=\n");

printf("Perform String Check(0/1):");

scanf("%d",&j);

if(j){

ptr=start;

printf("Enter the string of inputs:");

scanf("%s",n);

posi=0;

while(n[posi]!='\0'){

a[posi]=(n[posi]-'0');

//printf("%c\n",n[posi]);

//printf("%d",a[posi]);

posi++;

}

i=0;

printf("The visited States of the m/c are:");

do{

vst\_arr[i]=ptr->id\_num;

if(a[i]==0){

ptr=ptr->link0;

}

else if(a[i]==1){

ptr=ptr->link1;

}

else{

printf("iNCORRECT iNPUT\n");

return;

}

printf("[%d]",vst\_arr[i]);

i++;

}while(i<posi);

printf("\n");

printf("Present State:%d\n",ptr->id\_num);

printf("String Status:: ");

if(ptr->st\_val==1)

printf("String Accepted\n");

else

printf("String Not Accepted\n");

}

else

return 0;

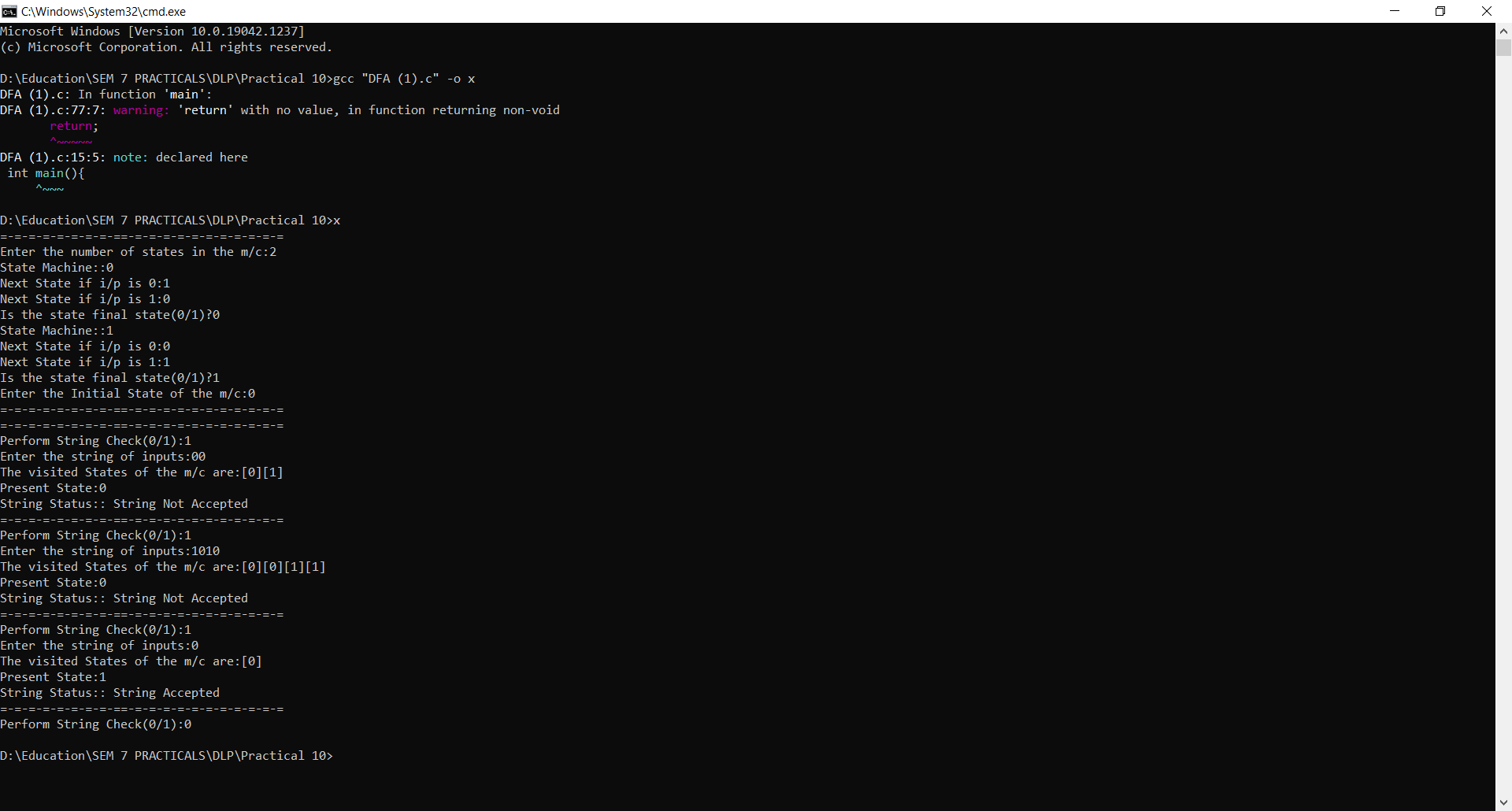
}

printf("=-=-=-=-=-=-=-=-==-=-=-=-=-=-=-=-=-=-=-=\n");

return 0;

}

**OUTPUT:**



**CONCLUSION:**

In this practical, we learnt about deterministic finite automata and implemented the same using C.