**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);}

%%

intyywrap(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;

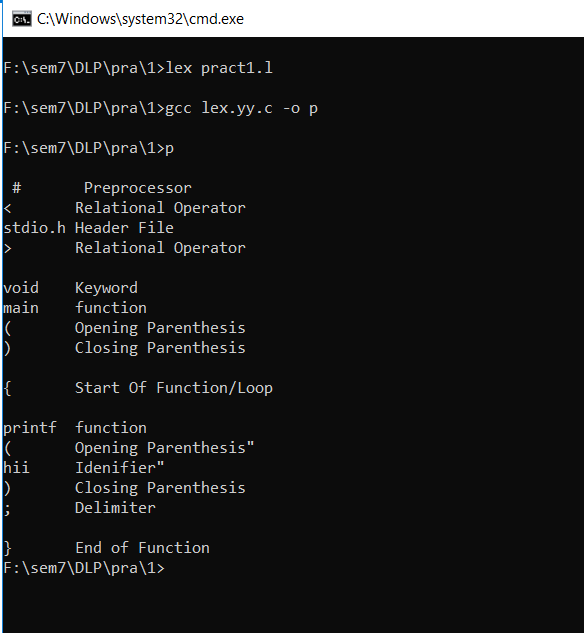
}

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);}

%%

intyywrap()

{

return 1;

}

int main()

{

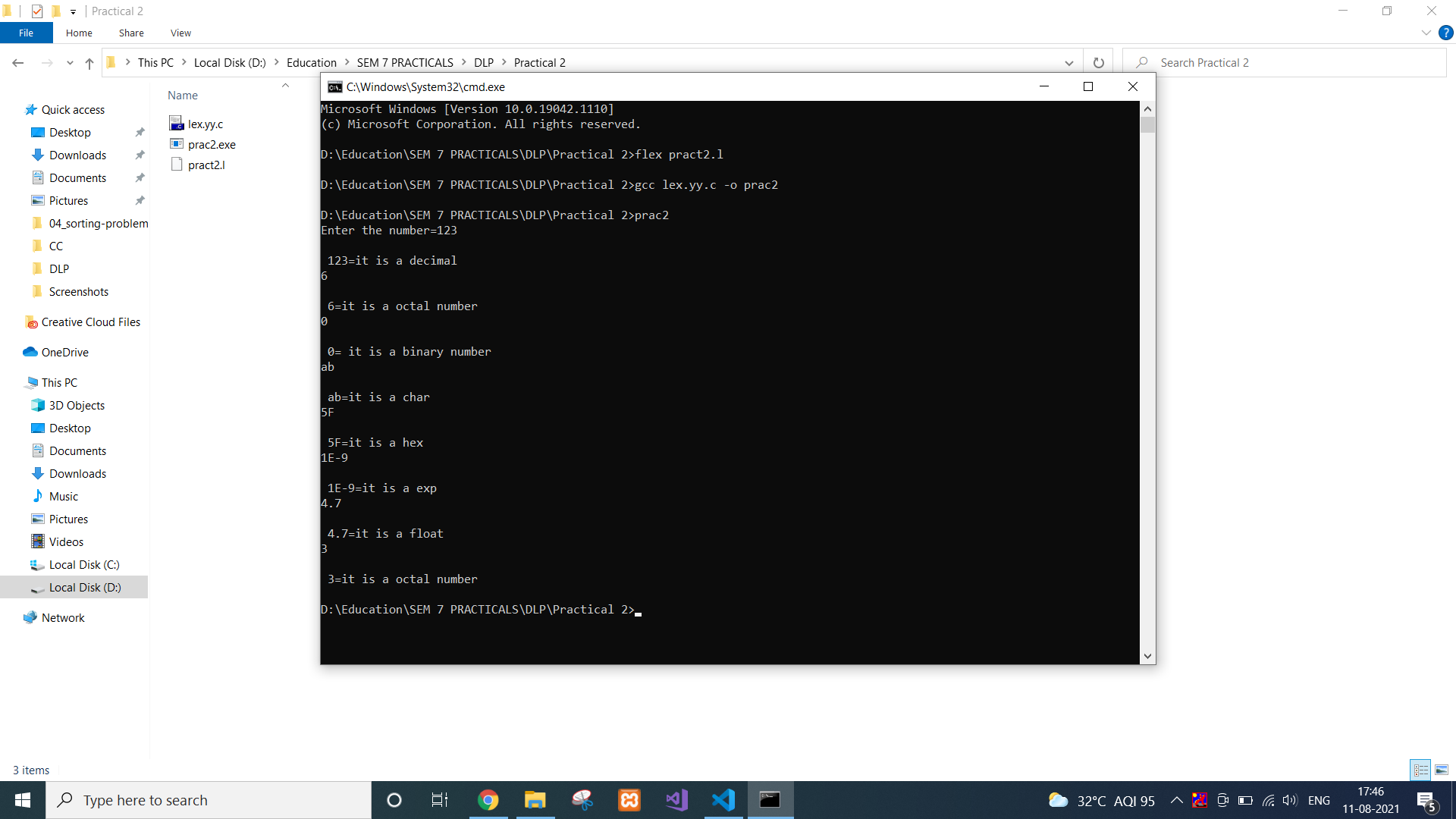
printf("Enter the number=");

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\*/

voidprint\_operator(char op);

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

intpos=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 \*\*\*/

voidprint\_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()

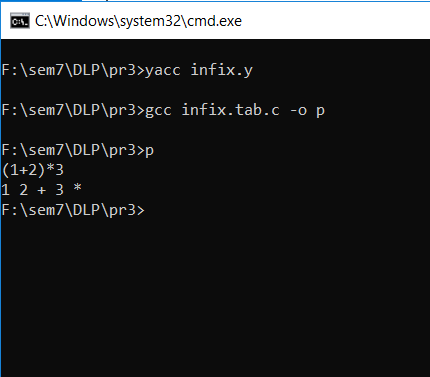
{

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.

**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)

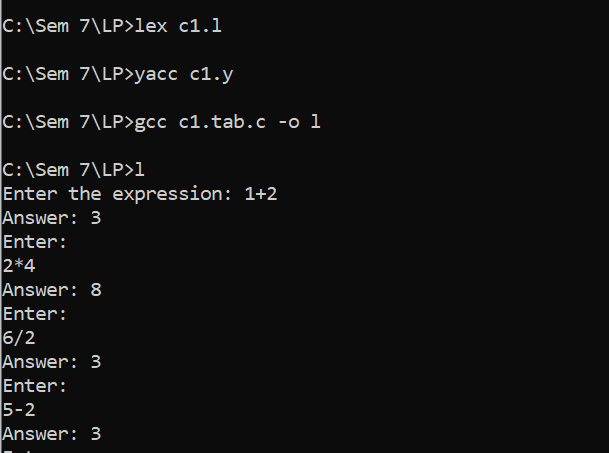
{

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

exit (1);

}

**Output:**



**Conclusion:** Here we have learned about how to implement calculator using lex and yacc.

**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

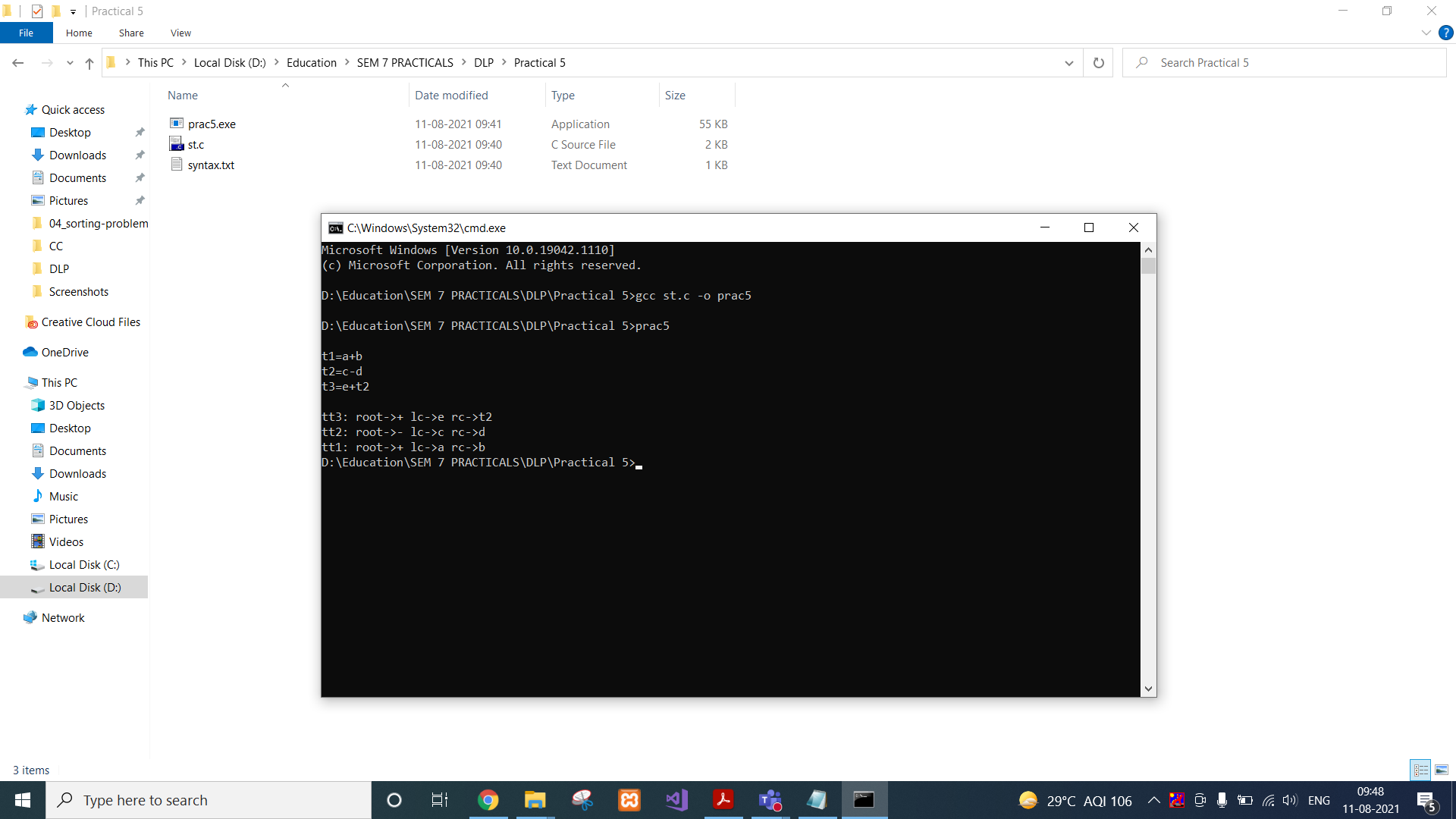
printf("rc->%c ",main[x]);

}

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.**

**CODE:**

#include<stdio.h>

#include<string.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();

}

void main()

{

FILE \*f;

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')

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;

}

}

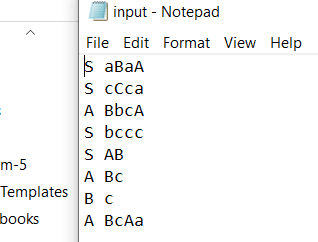
if(j==pro[0].n)

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

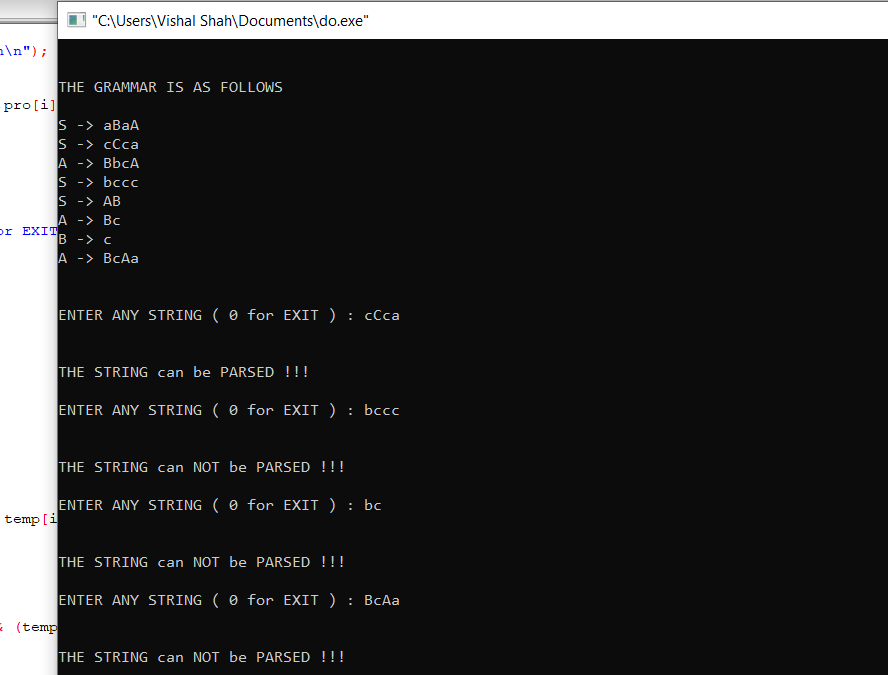
}

}

**Input.txt file:**



**OUTPUT:**



**CONCLUSION:** In this practical, we implemented Context Free Grammar.

**PRACTICAL – 7**

**AIM: Design of a Predictive parser.**

**CODE:**

#include<stdio.h>

#include<ctype.h>

#include<string.h>

#include<stdlib.h>

#define SIZE 128

#define NONE -1

#define EOS '\0'

#define NUM 257

#define KEYWORD 258

#define ID 259

#define DONE 260

#define MAX 999

char lexemes[MAX];

char buffer[SIZE];

int lastchar=-1;

int lastentry=0;

int tokenval=DONE;

int lineno=1;

int lookahead;

struct entry

{

char \*lexptr;

int token;

}

symtable[100];

struct entry

keywords[]= {"if",KEYWORD,"else",KEYWORD,"for",KEYWORD,"int",KEYWORD,"float",KEYWORD, "double",KEYWORD,"char",KEYWORD,"struct",KEYWORD,"return",KEYWORD,0,0

};

void Error\_Message(char \*m)

{

fprintf(stderr,"line %d, %s \n",lineno,m);

exit(1);

}

int look\_up(char s[ ])

{

int k;

for(k=lastentry; k>0; k--)

if(strcmp(symtable[k].lexptr,s)==0)

return k;

return 0;

}

int insert(char s[ ],int tok)

{

int len;

len=strlen(s);

if(lastentry+1>=MAX)

Error\_Message("Symbpl table is full");

if(lastchar+len+1>=MAX)

Error\_Message("Lexemes array is full");

lastentry=lastentry+1;

symtable[lastentry].token=tok;

symtable[lastentry].lexptr=&lexemes[lastchar+1];

lastchar=lastchar+len+1;

strcpy(symtable[lastentry].lexptr,s);

return lastentry;

}

/\*void Initialize()

{

struct entry \*ptr;

for(ptr=keywords;ptr->token;ptr+1)

insert(ptr->lexptr,ptr->token);

}\*/

int lexer()

{

int t;

int val,i=0;

while(1)

{

t=getchar();

if(t==' '||t=='\t');

else if(t=='\n')

lineno=lineno+1;

else if(isdigit(t))

{

ungetc(t,stdin);

scanf("%d",&tokenval);

return NUM;

}

else if(isalpha(t))

{

while(isalnum(t))

{

buffer[i]=t;

t=getchar();

i=i+1;

if(i>=SIZE)

Error\_Message("Compiler error");

}

buffer[i]=EOS;

if(t!=EOF)

ungetc(t,stdin);

val=look\_up(buffer);

if(val==0)

val=insert(buffer,ID);

tokenval=val;

return symtable[val].token;

}

else if(t==EOF)

return DONE;

else

{

tokenval=NONE;

return t;

} }}

void Match(int t)

{

if(lookahead==t)

lookahead=lexer();

else

Error\_Message("Syntax error");

}

void display(int t,inttval)

{

if(t=='+'||t=='-'||t=='\*'||t=='/')

printf("\nArithmetic Operator: %c",t);

else if(t==NUM)

printf("\n Number: %d",tval);

else if(t==ID)

printf("\n Identifier: %s",symtable[tval].lexptr);

else

printf("\n Token %d tokenval %d",t,tokenval);

}

void F()

{ //void E();

switch(lookahead)

{

case '(' :

Match('(');

E();

Match(')');

break;

case NUM :

display(NUM,tokenval);

Match(NUM);

break;

case ID :

display(ID,tokenval);

Match(ID);

break;

default :

Error\_Message("Syntax error");

}}

void T()

{ int t;

F();

while(1)

{switch(lookahead)

{

case '\*' :

t=lookahead;

Match(lookahead);

F();

display(t,NONE);

continue;

case '/' :

t=lookahead;

Match(lookahead);

display(t,NONE);

continue;

default :

return;

} }}

void E()

{ int t;

T();

while(1)

{ switch(lookahead)

{case '+' :

t=lookahead;

Match(lookahead);

T();

display(t,NONE);

continue;

case '-' :

t=lookahead;

Match(lookahead);

T();

display(t,NONE);

continue;

default :

return;

} }}

void parser()

{

lookahead=lexer();

while(lookahead!=DONE)

{ E();

Match(';');

}

}

int main()

{ char ans[10];

printf("\n Program for recursive descent parsing ");

printf("\n Enter the expression ");

printf("And place ; at the end\n");

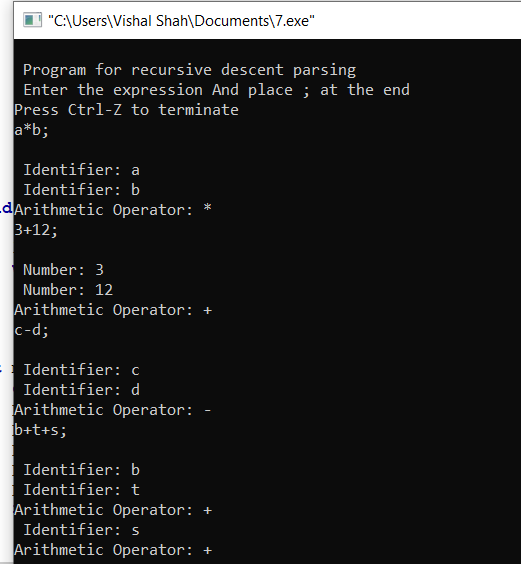
printf("Press Ctrl-Z to terminate\n");

parser();

return 0;

}

**OUTPUT:**



**CONCLUSION:** In this practical, we implemented the Predictive parser.

**PRACTICAL – 8**

**AIM: Implementation of code generator.**

**CODE:**

#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];

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

{

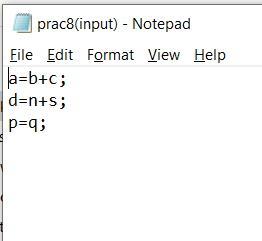
s1[j]=s;

j++;

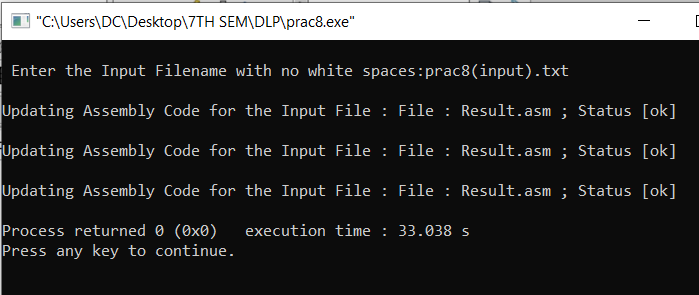
}}}

return 0;}

**Input File:**



**OUTPUT:**



**CONCLUSION:** In this practical, we implemented code generator.

**PRACTICAL – 9**

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

**CODE:**

#include<stdio.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;

printf("Enter the Number of Values:");

scanf("%d",&n);

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

{

printf("left: ");

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

printf("right: ");

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("\nAfter Dead Code Elimination\n");

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

{

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

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

}

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\n",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);

}

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';

}

}

}

printf("Optimized Code\n");

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

{

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

{

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

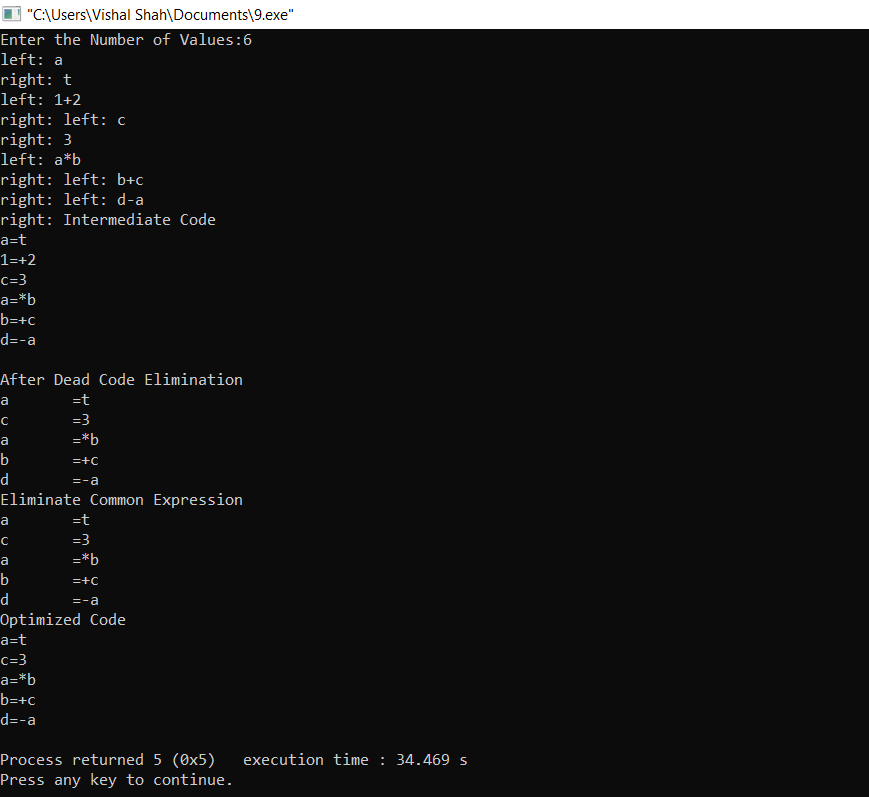
printf("%s\n",pr[i].r);

}

}

}

**OUTPUT:**



**CONCLUSION:** In this practical, we implemented code optimization for Common sub-expression elimination, Loop invariant code movement.

**PRACTICAL – 10**

**AIM: Implement Deterministic Finite Automata.**

**CODE:**

#include<stdio.h>

#include<string.h>

#define fl(i,a,b) for(i=a; i<b; i++)

#define scan(a) scanf("%d", &a)

#define nlineprintf("\n")

#define MAX 1000

int states, symbols, symdir[20], final\_states, mark[20], mat[20][20];

int main()

{

int i, j, k;

printf("Enter the number of states : ");

scan(states);

printf("Enter the number of symbols : ");

scan(symbols);

printf("Enter the symbols ");

nline;

fl(i,0,symbols)

{

printf("Enter the symbol number %d : ", i);

scan(symdir[i]);

}

printf("Enter the number of final states : ");

scan(final\_states);

printf("Enter the number of the states which are final : ");

nline;

fl(i,0,final\_states)

{

int temp;

scan(temp);

mark[temp]=1;

}

printf("Define the relations for the states and symbols : ");

nline;

fl(i,0,states)

{

fl(j,0,symbols)

{

printf("Enter the relation for Q(%d) -> %d : ", i, symdir[j]);

scan(mat[i][symdir[j]]);

}

}

//--------------------------------------------------------//

int cases;

printf("Enter the number of strings to be tested : ");

scan(cases);

fl(k,0,cases)

{

printf("Enter the string to be tested : ");

char str1[MAX];

scanf("%s", &str1);

int curr=0;

int limit=strlen(str1);

fl(i,0,limit)

{

int ele=(int)(str1[i]-'0');

curr=mat[curr][ele];

}

printf("The entered string is ");

if(mark[curr]==1)

printf("Accepted");

else

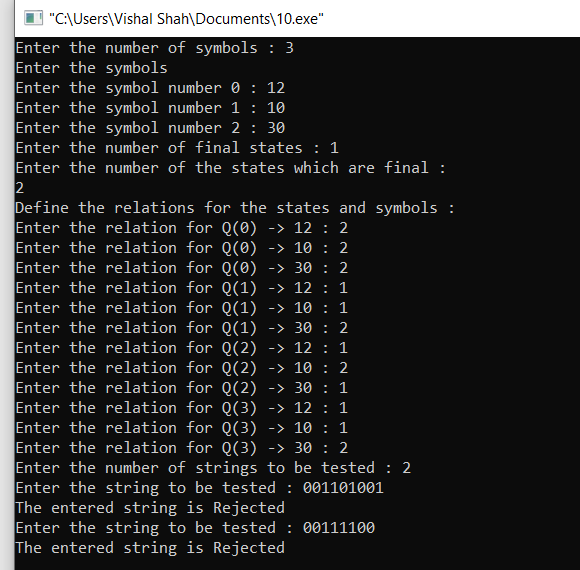
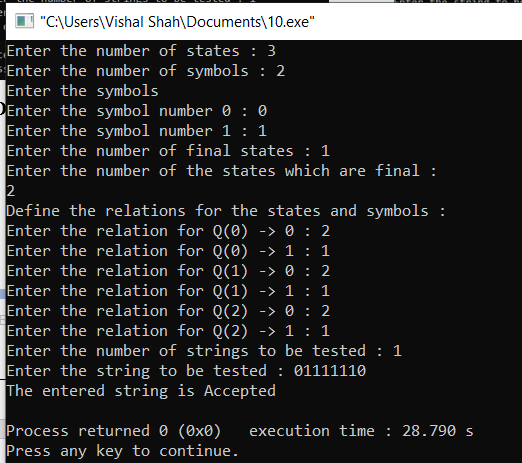
printf("Rejected");

nline;

}

return 0;}

**OUTPUT:**



**CONCLUSION:** In this practical, we implemented Deterministic Finite Automata.