Contents

Experiment 1: Transition Diagram to Transition Table

Aim:

Write a program in C/C++ to show the transition table from a given transition diagram.

Algorithm:

1. Start
2. Enter the number of states.
3. Enter the number of input variables.
4. Enter the state and its information.
5. Enter the input variables.
6. Enter the transition function information i.e. transition value from a state with a input variable.
7. Show the Transition Table.
8. Stop

Program (tt.c):

```
// write a program to display transition table on the
screen

#include<stdio.h>
#include<stdlib.h>
struct setStates
{
     int state;
     int final; // 0 - NO    1 - YES
     int start; // 0 - NO    1 - YES
};
typedef struct setStates sstate;

void main()
{
     int s,v,i,j;
     int **sv,*var;
     sstate *states;

     printf("\nInput the number of finite set of states :
");
     scanf("%d",&s);
     printf("\nInput the number of finite set of input
veriables : ");
     scanf("%d",&v);

     // creating transition table
     sv = (int **)malloc(v*sizeof(int));
     //printf("\n1 sucess\n");
     for(i=0;i<s;i++)
     {
          sv[i]=(int *)malloc(sizeof(int));
```

```
     }
     /*printf("\n2 sucess\n");
     printf("\nThe Array : \n");
     for(i=0;i<s;i++)
     {
          for(j=0;j<v;j++)
          {
               printf("%d\t",sv[i][j]);
          }
          printf("\n");
     }*/

     // storing state information
     states = (sstate *)malloc(s*sizeof(sstate));
     printf("\nInput the states and its info (state start
final): \n");
     for(i=0;i<s;i++)
     {


scanf("%d%d%d",&states[i].state,&states[i].start,&states[i]
.final);
     }

     // storing input veribale
     var = (int *)malloc(v*sizeof(int));
     printf("\nInput the veriables : \n");
     for(i=0;i<v;i++)
     {
          scanf("%d",&var[i]);
     }

     // storing inputs of transition function
     for(i=0;i<s;i++)
     {
          for(j=0;j<v;j++)
          {
               printf("\nThe sates %c with input veribale
%c move to state : ",states[i].state,var[j]);
               scanf("%d",&sv[i][j]);
          }
     }

     // display transition table on screen
     printf("\nThe Transition Table : \n");
     printf("\t");
     for(i=0;i<v;i++)
     {
          printf("%c\t",var[i]);
     }
```

```
     printf("\n-----------------------------------------
----");
     for(i=0;i<s;i++)
     {
          printf("\n%c  %c
%c\t",states[i].state,(states[i].start==0)?'
':'$',(states[i].final==0)?' ':'*');
          for(j=0;j<v;j++)
          {
               printf("%c\t",sv[i][j]);
          }
          printf("\n");
     }
}
```

Output:

```
Input the number of finite set of states : 4
Input the number of finite set of input veriables : 2

Input the states and its info (state start final):
97 1 1
98 0 0
99 0 0
100 0 0

Input the veriables :
48
49

The sates a with input veribale 0 move to state : 98
The sates a with input veribale 1 move to state : 99
The sates b with input veribale 0 move to state : 100
The sates b with input veribale 1 move to state : 97
The sates c with input veribale 0 move to state : 97
The sates c with input veribale 1 move to state : 100
The sates d with input veribale 0 move to state : 100
The sates d with input veribale 1 move to state : 98

The Transition Table :
        0       1
------------------------------------------------
a  $  * b       c

b       d       a

c       a       d

d       d       b
```

Experiment 2: Lexical Analyzer

Aim:

Write a program in C/C++ to implement a lexical analyzer.

Algorithm:

1. Start
2. Get the input expression from the user.
3. Store the keywords and operators.
4. Perform analysis of the tokens based on the ASCII values.
5.

ASCII Range	TOKEN TYPE
97-122	Keyword else identifier
48-57	Constant else operator
Greater than 12	Symbol

6. Print the token types.
7. Stop

Program (lexi.c):

```
/* Lexical Analyzer */
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<string.h>
void main()
{
     char
key[11][10]={"for","while","do","then","else","break","swit
ch","case","if","continue"};
     char oper[13]={'+','-
','*','/','%','&','<','>','=',';',':','!'};
     char a[20],b[20],c[20];
     int i,j,l,m,k,flag;
     clrscr();
     printf("\n Enter the expression: ");
     gets(a);
     i=0;
     while(a[i])
     {
          flag=0;
          j=0;
          l=0;
          b[0]='\0';
          if((toascii(a[i]>=97))&&(toascii(a[i]<=122)))
          {

     if((toascii(a[i+1]>=97))&&(toascii(a[i+1]<=122)))
```

```
          {

while((toascii(a[i]>=97))&&(toascii(a[i]<=122)))
               {
                    b[j]=a[i];
                    j++; i++;
               }
               b[j]='\0';
          }
          else
          {
               b[j]=a[i];
               i++;
               b[j+1]='\0';
          }
          for(k=0;k<=9;k++)
          {
               if(strcmpi(b,key[k])==0)
               {
                    flag=1;
                    break;
               }
          }

          if(flag==1)
               printf("\n %s is the keyword",b);
          else
               printf("\n %s is the identifier",b);
     }
     else if((toascii(a[i]>=48))&&(toascii(a[i]<=57)))
     {

if((toascii(a[i+1]>=48))&&(toascii(a[i+1]<=57)))
          {

while((toascii(a[i]>=48))&&(toascii(a[i]<=57)))
               {
                    c[l]=a[i];
                    l++;  i++;
               }
          }
          else
          {
               c[l]=a[i];
               i++;l++;
          }
          c[l]='\0';
          printf("\n %s is the constant",c);
     }//second ifelse
     else
     {
```

```
                for(m=0;m<13;m++)
                {
                     if(a[i]==oper[m])
                     {
                          printf("\n %c is the
operator",a[i]);
                          break;
                     }
                }
                if(m>=13)
                     printf("\n %c is the symbol",a[i]);
                i++;
           }//last else
     } //while
     getch();
}
```

Output:

```
Enter the expression: while(i<5)break

 while is the keyword
 ( is the symbol
 i is the identifier
 < is the operator
 5 is the constant
 ) is the symbol
 break is the keyword



 Enter the expression: if(b>20)continue

 if is the keyword
 ( is the symbol
 b is the identifier
 > is the operator
 20 is the constant
 ) is the symbol
 continue is the keyword
```

Experiment 3: Implementation of PASS 1 Assembler

Aim:

To write a C program to translate assembly language to intermediate code.

Algorithm:

1. open the file (assembly language program.)
2. Separate the mnemonic instructions into label , opcode and operand.
3. Generate the symbol table
4. to generate Literal table check whether operand[0]is equal to '=' then copy the operand to literal table.
5. If opcode 'END' is encountered then check whether literal are assigned address.
6. Check whether the literal address is zero
7. if true then store the pc to the value of the literal table for the first literal .
8. Increment pc by 3.
9. steps 7 & 8 are repeated until all the literals are assigned addresses.
10. Print the pc , label , opcode & operand and store the intermediate code in file for later use by Pass 2.
11. Steps 2 to 10 are executed until EOF is encountered.

Program:

```
#include<stdio.h>
#include<conio.h>
struct sym
{
char lab[10];
int val;
};
struct li
{
char oprn[10];
int addr;
};
main ()
{
FILE *f1;
char la[10],op[10],opr[10],a[1000],c;
int i,j,n,k=0,lc=0,m=0,p=0;
struct sym s[10];
struct li l[10];
clrscr();
f1=fopen("pass1inp.txt","r");
c=fgetc(f1);
i=0;
printf ("\n SOURCE PROGRAM \n");
printf("%c",c);
while (c !=EOF)
{
```

```
a[i]=c;
c=fgetc(f1);
i++;
printf("%c",c);
}
i=0;
printf("\n INTERMEDIATE FILE \n");
while(strcmp(op,"end")!=0)
{
if(a[i]=='\t')
{
strcpy(la," ");
i++;
}
else
{
j=0;
while(a[i]!='\t')
{
la[j]=a[i];
i++;
j++;
}
la[j]='\0';
i++;
}
if(a[i]=='\t')
{
strcpy(op," ");
i++;
}
else
{
j=0;
while (a[i]!='\t')
{
op[j]=a[i];
i++;

j++;
}
op[j]='\0';
i++;
}
if(a[i]=='\n')
{
strcpy(opr," ");
i++;
}
else
 {
```

```
 j=0;
while (a[i] !='\n')
 {
opr [j]=a [i];
 i++;
j++;
}
opr[j]='\0';
i++;
}
j=0;
if (strcmp (la," ") !=0)
{
strcpy(s[m].lab,la);
if (strcmp(op, "start") ==0)
{
lc=atoi(opr);
s [m] .val=lc,
m++;
continue;
}
else if (strcmp (op, "equ") ==0)
{
printf("\n%d\t",lc);
s[m] .val=atoi(opr);
m++;
}
else if (strcmp (op, "resw") ==0)
{
printf("\n%d\t",lc);
s[m] .val=lc;
lc=lc+atoi(opr) *3;
m++;
}
else if (strcmp (op, "resb") ==0)
 {
printf("\n%d\t",lc);
s[m] .val=lc;
lc=lc+atoi(opr);
m++;
}
else
{
printf("\n%d\t",lc);
strcpy(s[m].lab,la);
s[m] .val=lc;
lc=lc+3;
m++;
}
}
else
```

```
{
printf("\n%d\t",lc);
lc=lc+3;
}
if(opr[0] =='=')
{
strcpy(l[k].oprn,opr);
k++;
}
printf("%s\t%s\n",op,opr);
}
if(strcmp(op,"end")==0)
for(n=p;n<k;n++)
{
l[n].addr=lc-3;
printf("\n%d\t%s\n",l[n].addr,l[n].oprn);
lc=lc+3;
p++;
}
printf("\n symbol table \n");
for(i=0;i<m;i++)
printf("\n%s\t%d\n",s[i].lab,s[i].val);
printf("\n Literal table \n");
for(i=0;i<k;i++)
printf("\n%s\t%d\n",l[i].oprn,l[i].addr);
getch();
}
```

Output:

```
 SOURCE PROGRAM
add     start   1000
           lda     ='02'
           add     ='05'
           sta     two
two     resw    1
           end     add


 INTERMEDIATE FILE

1000    lda     ='02'

1003    add     ='05'

1006    sta     two

1009    resw    1

1012    end     add
```

```
1012    ='02'

1015    ='05'

 symbol table

add     1000

two     1009

 Literal table

='02'   1012

='05'   1015
```

Experiment 4: Implementation of PASS 2 Assembler

Aim:

To write a C program to translate intermedite code to machine language.

Algorithm:

1. Open the symbol table, Literal table and intermediate file.
2. Check whether opcode is not equal to end. If false goto step 7.
3. search optable for opcode
4. when condition is true display the corresponding machine value from optable
5. Check whether the operand is literal , if true get the literals address from the literal table .Find the displacement address and display it.
6. otherwise check the operand for symbol in symbol table
7. If the instruction is format 4 store the symbol value as operand address. Otherwise Find the displacement address and display it.
8. If opcode is BYTE or WORD then convert constant to object code.
9. goto step 2.
10. stop the process.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct
{
char sym[10];
int val;
}s[10];
struct
{
char opt[10];
int val;
}o[10];
struct
{
char lit[10];
int addr;
}l[10];
struct
{
char op[10],opr[10];
int lc;
}inter[10];
main()
{
FILE *f1,*f2,*f3,*f4;
int i=0,j=0,k=0,m=0,n=0,r;
```

```
char c,a[1000];
clrscr();
f1=fopen("symbb.txt","r");
f2=fopen("op.txt","r");
f3=fopen("li.txt","r");
f4=fopen("inter.txt","r");
printf("symbol table\n\n");
while(!feof(f1))
{
fscanf(f1,"%s%d",s[j].sym,&s[j].val);
printf("%s\t%d\n",s[j].sym,s[j].val);
j++;
}
printf("optable\n\n");
while(!feof(f2))
{
fscanf(f2,"%s%d",o[k].opt,&o[k].val);
printf("%s\t%d\n",o[k].opt,o[k].val);
k++;
}
printf("Literal table\n\n");
while(!feof(f3))
{
fscanf(f3,"%s%d",l[m].lit,&l[m].addr);
printf("%s\t%d\n",l[m].lit,l[m].addr);
m++;
}
printf("Intermediate file\n");
while(!feof(f4))
{
fscanf(f4,"%d%s%s",&inter[n].lc,inter[n].op,inter[n].opr);
printf("%d\t%s\t%s\n",inter[n].lc,inter[n].op,inter[n].opr)
;
}
rewind(f4);
printf("\nmachine instruction \n\n");
while(!feof(f4))
{
fscanf(f4,"%d%s%s",&inter[n].lc,inter[n].op,inter[n].opr);
if((strcmp(inter[n].op,"equ")==0)||(strcmp(inter[n].op,"wor
d")==0))
continue;
if((strcmp(inter[n].op,"resw")==0)||(strcmp(inter[n].op,"re
sb")==0))
continue;
printf("%d\t",inter[n].lc);
for(i=0;i<k;i++)
{
if(strcmp(inter[n].op,o[i].opt)==0)
printf("%d\t",o[i].val);
}
```

```
for(i=0;i<m;i++)
{
if(inter[n].op[0]!='*')
if(strcmp(inter[n].opr,l[i].lit)==0)
printf("%d\n",l[i].addr-inter[n].lc-3);
}

for(i=0;i<j;i++)
{
if(strcmp(inter[n].opr,s[i].sym)==0)
{
if(inter[n].op[0]=='+')
printf("%d\n",s[i].val);
else
printf("%d\n",s[i].val-inter[n].lc-3);
}
}
if(inter[n].op[0]=='*')
{
printf("%s\n",inter[n].opr);
}
}
getch();
return 0;
}
```

Output:

```
symbol table
addpgm  1000
two     1009
optable
lda     1
add     2
sta     3
+sta    3
Literal table
='02'   1012
='05'   1015
Intermediate file
1000    lda     ='02'
1003    add     ='05'
1006    +sta    two
1010    resw    1
1013    *       ='02'
1016    *       ='05'
machine instruction
1000    1       9
1003    2       9
1006    3       1009
```

```
1013    ='02'
1016    ='05'
```

Experiment 5: Implement Macro Preprocessor

Aim:

To write a C program to implement macro preprocessor.

Algorithm:

MACROPROCESSOR

- EXPANDING=FALSE.
- Read each line and call GETLINE() and PROCESSLINE() until END encounters.

PROCESSLINE ( )

- If OPCODE is a macroname then EXPAND ( ).
- Else if OPCODE is MACRO ,then DEFINE ( ).
- Else write line to expanded file as such.

DEFINE( )

- Enter Macro name into NMATAB.
- Enter macro prototype into DEFTAB.
- Set LEVEL=1.
- Substitute parameters with positional notations and enter to DEFTAB.
- If OPCODE=MACRO, LEVEL++;
- If OPCODE=MEND, LEVEL--;
- Continue this until LEVEL=0
- Store beginning and end of definition as pointers within NAMTAB

EXPAND ( )

- EXPANDING = TRUE
- Set up arguments from macro invocation in ARGTAB.
- Write macro invocation statement to expanded file as a comment line.
- Call GETLINE() and PROCESSLINE() till macro definition ends.
- Set EXPANDING=FALSE.

GETLINE ( )

- If EXPANDING is TRUE, read from DEFTAB (data structure where macro body is stored) and substitute arguments for positional notations.
- If EXPANDING is FALSE , read next line from input file.

Program:

```
#include<stdio.h>
#include<string.h>

void GETLINE();
void PROCESSLINE();
void DEFINE();
void EXPAND();


 FILE *expanded;
 FILE *input;

 char label[10],opcode[10],operand[25];
 char line[20];
 int namcount=0, defcount=0;
 int EXPANDING;
 int curr;



 struct namtab
 {
 char name[10];
 int start,end;
}mynamtab[15];

 struct deftab
 {
        char macroline[25];
  }mydeftab[25];

struct argtab
{
       char arg[3][9];
 }myargtab;
///MACRO MAIN
int main()

{
EXPANDING=0;

input =fopen("input.txt","r");
expanded=fopen("expanded.txt","w");
GETLINE();
```

```
while(strcmp(opcode,"END")!=0)
{

     PROCESSLINE();
     GETLINE();
 }
fprintf(expanded,"%s",line);
getch();
return 1;
}

//   GETLINE
void GETLINE()
{    char word1[10],word2[10],word3[10],buff[10];
int count=0,i,j=0;
if(EXPANDING)strcpy(line,mydeftab[curr++].macroline);
 else fgets(line,20,input);
opcode[0]='\0';label[0]='\0';operand[0]='\0';word1[0]='\0';
word2[0]='\0';word3[0]='\0';

                      for(i=0;line[i]!='\0';i++)
                      {

                          if(line[i]!=' ')
                          buff[j++]=line[i];
                          else
                             {

                             buff[j]='\0';
                              strcpy(word3,word2);
                              strcpy(word2,word1);
                             strcpy(word1,buff);
                             j=0;count++;
                              }
                    }

                       buff[j-1]='\0';
                      strcpy(word3,word2);
                      strcpy(word2,word1);
                       strcpy(word1,buff);


switch(count)
{
         case 0:strcpy(opcode,word1);break;
         case 1:{strcpy(opcode,word2);strcpy(operand,word1);}break;
```

```
          case 2:{strcpy(label,word3);
strcpy(opcode,word2);
strcpy(operand,word1);}break;
                        }
}
//     PROCESSLINE
                        void PROCESSLINE()
                        {
                             int i;
                         for(i=0;i<namcount;i++)

                                  if(!strcmp(opcode,mynamtab[i].name))
                                  {
EXPAND();return;
     }
{

if(!strcmp(opcode,"MACRO"))
DEFINE();
else fprintf(expanded,"%s",line);
}
}
     void DEFINE()
     {
          int LEVEL,i=0,j=0,k=0;
       char param[5][9];
       char s[3];
      strcpy(s,"123");

          strcpy(mynamtab[namcount].name,label);
          mynamtab[namcount].start=defcount;
          strcpy(mydeftab[defcount].macroline,line);
          while(operand[i]!='\0')
          {
                                  if(operand[i]!=',')
                                  param[j][k++]=operand[i];
                                  else
                                  {
                                      param[j++][k]='\0';

                                      k=0;
                                      }

                                 i++;
                                  }
                                  param[j][k]='\0';
```

```
      LEVEL=1;

    while(LEVEL>0)
      {

        GETLINE();

          if(operand[0]!='\0')
         {
        for(i=0;i<3;i++)
        {
        if(!strcmp(operand,param[i]))
        {

        operand[0]='?';
        operand[1]=s[i];
        operand[2]='\0';
         }
         }
         }
       if(!strcmp(opcode,"MACRO"))
        LEVEL++;
       else if(!strcmp(opcode,"MEND"))
        LEVEL--;

        strcpy(mydeftab[defcount].macroline,opcode);
        if(operand[0]!='\0')
        {
        strcat(mydeftab[defcount].macroline," ");
        strcat(mydeftab[defcount].macroline,operand);
        strcat(mydeftab[defcount].macroline,"\n");
        }
        strcat(mydeftab[defcount++].macroline,"\n");

                    }
                    mynamtab[namcount++].end=defcount;
                    }
void EXPAND()
                    {
                         int i,end=0,j=0,k=0;
                         EXPANDING=1;
                         int arg=0;

                     fprintf(expanded,"//%s",line);
```

```
for(i=0;i<namcount;i++)
{
if(!strcmp(opcode,mynamtab[i].name))
{

curr=mynamtab[i].start;
end=mynamtab[i].end;


      while(operand[i]!='\0')
   {
   if(operand[i]!=',')
   myargtab.arg[j][k++]=operand[i];
   else
   {
       myargtab.arg[j++][k]='\n';

       k=0;
       }

  i++;
   }
    myargtab.arg[j][k]='\n';


}
}

while(curr<(end-1))
{
                  GETLINE();
 if(operand[0]=='?')

strcpy(operand,myargtab.arg[operand[1]-
'0'-1]);

 fprintf(expanded,"%s %s
%s",label,opcode,operand);

}
EXPANDING=0;


}
```

Output:

Input.txt

```
COPY START 1000

RDBUFF MACRO P,Q,R
CLEAR A
CLEAR S
CLEAR X
+LDT #4096
TD P
JEQ *-3
RD P
STCH Q
JLT *-19
LDA R
COMP #0
STX R
MEND

//main program

FIRST STL RETADR
RDBUFF F1,BUFF1,L1
CLEAR X
RDBUFF F2,BUFF2,L2
RDBUFF F3,BUFF3,L3
JLT *-19
STA
END
```

Expanded.txt

```
COPY START 1000
//main program
FIRST STL RETADR
//RDBUFF F1,BUFF1,L1
 CLEAR A
 CLEAR S
 CLEAR X
 +LDT #4096
 TD F1
 JEQ *-3
 RD F1
 STCH BUFF1
 JLT *-19
 LDA L1
 COMP #0
```

```
 STX L1
CLEAR X
//RDBUFF F2,BUFF2,L2
 CLEAR A
 CLEAR S
 CLEAR X
 +LDT #4096
 TD F2
 JEQ *-3
 RD F2
 STCH BUFF2
 JLT *-19
 LDA L2
 COMP #0
 STX L2
//RDBUFF F3,BUFF3,L3
 CLEAR A
 CLEAR S
 CLEAR X
 +LDT #4096
 TD F3
 JEQ *-3
 RD F3
 STCH BUFF3
 JLT *-19
 LDA L3
 COMP #0
 STX L3
JLT *-19
STA
END
```

Experiment 6: Regular Expression to NFA

Aim:

Write a program in C/C++ to generate the NFA from a given Regular Expression.

Algorithm:

Given a regular expression r. To construct a DFA D that recognizes L(r).

1. Construct a syntax tree for the augmented regular expression(r)#, where # is unique endmarker appended to (r).
2. Construct the functions ***nullable, firstpos, lastpos,*** and ***followpos*** by making depth-first traversals of T.
3. Construct ***Dstates***, the set of states of D, and ***Dtran***, the transition table for D by the procedure. The states in ***Dstates*** are sets of positions; initially, each state is "unmarked" and a state becomes "marked", just before we consider its out-transitions. The start state q of D is ***firstpos(root)***, and the accepting states are all those containing the position associated with the endmarker#.

intitally, the only unmarked state in ***dstates*** is ***firstpos(root),***
where ***root*** is the root of the syntax tree for(r)#;
while there is an unmarked state T is ***Dstates*** **do begin**
mark T;
for each input sysmbol a **do begin**
let U be the set of positions that are in ***followpos(p)***
for some position that are in ***followpos(p)***
such that the symbol at postion ***p*** is a;
if U is not empty and is not in ***Dstates*** **then**
add U as an unmarked state to ***Dstates;***
Dtran[T,a] := U
end
end

Program (regnfa.cpp):

```
/* Regular Expression To NFA */

#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<process.h>
#include<alloc.h>

int num=1;
struct state
{
     char label;
     state* next1;
     state* next2;
```

```
     int final,num;
     void assign(char,int,state*,state*,int);
};

void state::assign(char c=0,int n=0,state *s2=NULL,state
*s3=NULL,int f=0)
{
     label=c;
     next1=s2;
     next2=s3;
     final=f;
     num=n;
}

int accept(char *c,state *s);

void error()
{
     cout<<"Error";
     getch();
     exit(0);
}

char* findmatch(char *s)
{
     while(*(++s)!='\0')
     {
          if(*s==')')
               return s;
          if((*s=='(')&&((s=findmatch(s))==NULL))
               return NULL;
     }
     return NULL;
}

char* findplus(char *s)
{
     while(*s!='\0')
     {
          if((*s=='(')&&((s=findmatch(s))==NULL))
               return NULL;
          if(*s=='|')
               return s;
          s++;
     }
     return NULL;
}

state* ConstructNFA(char *rexp)
{
     char *temp1,*temp2;
```

```
int len=strlen(rexp);
if(len==1)
{
     state* s1=new state;
     state* s2=new state;
     state* s3=new state;
     s1->assign(*rexp,num++,s2,NULL);
     s2->assign(0,num++,NULL,NULL,1);
     s3->assign(0,0,s1,s2);
     return s3;
}
temp2=findplus(rexp);
if(temp2!=NULL)
{
     temp1=(char*)malloc(len);
     strncpy(temp1,rexp,temp2-rexp);
     temp1[temp2++-rexp]='\0';
     if(*temp2=='\0')
          error();
     state* s1=ConstructNFA(temp1);
     state* s2=ConstructNFA(temp2);
     state* s3=new state;
     state* s4=new state;
     s3->assign(0xEE,num++,s1->next1,s2->next1);
     s1->next2->assign(0xEE,s1->next2->num,s4);
     s2->next2->assign(0xEE,s2->next2->num,s4);
     s4->assign(0,num++,NULL,NULL,1);
     state* s5=new state;
     s5->assign(0,0,s3,s4);
     return s5;
}
if(rexp[0]=='(')
{
     temp2=findmatch(rexp);
     if(temp2==NULL)
          error();
     if((rexp[len-1]==')')&&(rexp+len-1==temp2))
     {
          if(rexp==NULL);
               rexp[len-1]='\0';
          return ConstructNFA(rexp+1);
     }
     temp2=findmatch(rexp);
     temp1=(char *)malloc(temp2-rexp+20);
     strncpy(temp1,rexp,temp2-rexp+1);
     temp1[temp2++-rexp+1]='\0';
}
else
{
     temp1=(char *)malloc(10);
     temp1[0]=rexp[0];
```

```
            temp1[1]='\0';
            temp2=rexp+1;
      }
      if(strcmp(temp2,"*")==0)
      {
            state* s1=ConstructNFA(temp1);
            state* s3=new state;
            state* s4=new state;
            s3->assign(0xEE,num++,s1->next1,s4);
            s1->next2->assign(0xEE,s1->next2->num,s1-
>next1,s4);
            s4->assign(0,num++,NULL,NULL,1);
            state *n=new state;
            n->assign(0,0,s3,s4);
            return n;
      }
      if(temp2[0]=='*')
      {
            strcat(temp1,"*");
            temp2++;
      }
      state* s1=ConstructNFA(temp1);
      state* s2=ConstructNFA(temp2);
      state *x=new state;
      s1->next2->assign(0xEE,s1->next2->num,s2->next1);
      x->assign(0,0,s1->next1,s2->next2);
      return x;
}

int accept(char *c,state *s)
{
      if((*c=='\0')&&(s->final==1))
            return 1;
      if(s->label=='î')
            return accept(c,s->next1)|(s-
>next2==NULL?0:accept(c,s->next2));
      if(s->label==*c)
            return accept(++c,s->next1);
      return 0;
}


void reinitialize(state *s)
{
      if((s==NULL)||(s->final==0))
            return;
      if(s->final==2)
            s->final=0;
      reinitialize(s->next1);
      reinitialize(s->next2);
}
```

```
void display(state *s)
{
     if((s==NULL)||(s->next1==NULL)||(s->final==2))
          return;
     cout<<s->num<<"\t"<<s->label<<"\t["<<s->next1->num;
     if(s->next1->final==1)
          cout<<"*";
     if(s->next2!=NULL)
     {
          cout<<","<<s->next2->num;
          if(s->next2->final==1)
          cout<<"*";
     }
     cout<<"]\n";
     s->final=2;
     display(s->next1);
     display(s->next2);
}

void DisplayNFA(state *s)
{
     display(s);
     reinitialize(s);
}

void main()
{
     clrscr();
     state *s1;
     char s[300];
     cout<<"Enter regular expression:";
     cin>>s;
     int n;
     s1=ConstructNFA(s);
     while(1)
     {
          clrscr();
          cout<<"\n\t1.View Transition Table i.e.
NFA\n\t2.Simulate DFA\n\t3.Exit\n";
          cout<<"\nEnter Your Choice:";
          cin>>n;
          switch(n)
          {
          case 1:
               DisplayNFA(s1->next1);
               break;
          case 2:
               cout<<"Enter String:";
               cin>>s;
               if(accept(s,s1->next1))
```

```
                cout<<"The string is in the language\n";
                else
                cout<<"The string is not in the language\n";
                break;
           case 3:
                exit(0);
           }
           getch();
     }
}
```

Output:

```
Enter regular expression:(a|b)*abb


        1.View Transition Table i.e. NFA
        2.Simulate DFA
        3.Exit

Enter Your Choice:1
7       ε       [5,8]
5       ε       [1,3]
1       a       [2]
2       ε       [6]
6       ε       [5,8]
8       ε       [9]
9       a       [10]
10      ε       [11]
11      b       [12]
12      ε       [13]
13      b       [14*]
3       b       [4]
4       ε       [6]


        1.View Transition Table i.e. NFA
        2.Simulate DFA
        3.Exit

Enter Your Choice:2
Enter String:abaabbabb
The string is in the language

        1.View Transition Table i.e. NFA
        2.Simulate DFA
        3.Exit

Enter Your Choice:3
```

Experiment 7: NFA to DFA

Aim:

To write a C program to construct a DFA from the given NFA.

Algorithm:

1. Start the program.
2. Accept the number of state A and B.
3. Find the E-closure for node and name if as A.
4. Find v(a,a) and (a,b) and find a state.
5. Check whether a number new state is obtained.
6. Display all the state corresponding A and B.
7. Stop the program.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#include<process.h>
typedef struct
{
        int num[10],top;
}
stack;
stack s;
int mark[16][31],e_close[16][31],n,st=0;
char data[15][15];
void push(int a)
{
        s.num[s.top]=a;
        s.top=s.top+1;
}
int pop()
{
        int a;
        if(s.top==0)
                return(-1);
        s.top=s.top-1;
        a=s.num[s.top];
        return(a);
}
void epi_close(int s1,int s2,int c)
{
        int i,k,f;
```

```
	for(i=1;i<=n;i++)
	{
		if(data[s2][i]=='e')
		{
			f=0;
			for(k=1;k<=c;k++)
			if(e_close[s1][k]==i)
			f=1;
			if(f==0)
			{
				c++;
				e_close[s1][c]=i;
				push(i);
			}
		}
	}
	while(s.top!=0) epi_close(s1,pop(),c);
}
int move(int sta,char c)
{
	int i;
	for(i=1;i<=n;i++)
	{
		if(data[sta][i]==c)
		return(i);
	}
	return(0);
}
void e_union(int m,int n)
{
	int i=0,j,t;
	for(j=1;mark[m][i]!=-1;j++)
	{
		while((mark[m][i]!=e_close[n][j])&&(mark[m][i]!=-1))
		i++;
		if(mark[m][i]==-1)mark[m][i]=e_close[n][j];
	}
}
void main()
{
	int i,j,k,Lo,m,p,q,t,f;
	clrscr();
	printf("\n enter the NFA state table entries:");
	scanf("%d",&n);
	printf("\n");
	for(i=0;i<=n;i++)
	printf("%d",i);
	printf("\n");
	for(i=0;i<=n;i++)
	printf("------");
```

```
printf("\n");
for(i=1;i<=n;i++)
{
        printf("%d|",i);
        fflush(stdin);
        for(j=1;j<=n;j++)
        scanf("%c",&data[i][j]);
}
for(i=1;i<=15;i++)
for(j=1;j<=30;j++)
{
        e_close[i][j]=-1;
        mark[i][j]=-1;
}
for(i=1;i<=n;i++)
{
        e_close[i][1]=i;
        s.top=0;
        epi_close(i,i,1);
}
for(i=1;i<=n;i++)
{
        for(j=1;e_close[i][j]!=-1;j++)
        for(k=2;e_close[i][k]!=-1;k++)
        if(e_close[i][k-1]>e_close[i][k])
        {
        t=e_close[i][k-1];
        e_close[i][k-1]=e_close[i][k];
        e_close[i][k]=t;
        }
}
printf("\n the epsilon closures are:");
for(i=1;i<=n;i++)
{
        printf("\n E(%d)={",i);
        for(j=1;e_close[i][j]!=-1;j++)
        printf("%d",e_close[i][j]);
        printf("}");
}
j=1;
while(e_close[1][j]!=-1)
{
        mark[1][j]=e_close[1][j];
        j++;
}
st=1;
printf("\n DFA Table is:");
printf("\n a b ");
printf("\n--------------------------------------");
for(i=1;i<=st;i++)
```

```
{
        printf("\n{");
        for(j=1;mark[i][j]!=-1;j++)
        printf("%d",mark[i][j]);
        printf("}");
        while(j<7)
        {
                printf(" ");
                j++;
        }
        for(Lo=1;Lo<=2;Lo++)
        {
                for(j=1;mark[i][j]!=-1;j++)
                {
                        if(Lo==1)
                        t=move(mark[i][j],'a');
                        if(Lo==2)
                        t=move(mark[i][j],'b');
                        if(t!=0)
                        e_union(st+1,t);
                }
                for(p=1;mark[st+1][p]!=-1;p++)
                for(q=2;mark[st+1][q]!=-1;q++)
                {
                        if(mark[st+1][q-1]>mark[st+1][q])
                        {
                                t=mark[st+1][q];
                                mark[st+1][q]=mark[st+1][q-1];
                                mark[st+1][q-1]=t;
                        }
                }
                f=1;
                for(p=1;p<=st;p++)
                {
                        j=1;
                        while((mark[st+1][j]==mark[p][j])&&(mark[st+1][j]!=-1))
                        j++;
                        if(mark[st+1][j]==-1 && mark[p][j]==-1)
                        f=0;
                }
                if(mark[st+1][1]==-1)
                f=0;
                printf("\t{");
                for(j=1;mark[st+1][j]!=-1;j++)
                {
                        printf("%d",mark[st+1][j]);
                }
                printf("}\t");
                if(Lo==1)
                printf(" ");
```

```
                        if(f==1)
                        st++;
                        if(f==0)
                        {
                                for(p=1;p<=30;p++)
                                mark[st+1][p]=-1;
                        }
                }
}
getch();
}
```

Output:
Enter the NFA state table entries: 11
(**Note:** *Instead of* '-' *symbol use blank spaces in the output window*)
0 1 2 3 4 5 6 7 8 9 10 11
---
1 - e - - - - - e - - -
2 - - e - e - - - - - -
3 - - - a - - - - - - -
4 - - - - - - e - - - -
5 - - - - - b - - - - -
6 - - - - - - e - - - -
7 - e - - - - - e - - -
8 - - - - - - - - e - -
9 - - - - - - - - - e -
10 - - - - - - - - - - e
11 - - - - - - - - - - - The Epsilon Closures
Are:
E(1)={12358}
E(2)={235}
E(3)={3}
E(4)={234578}
E(5)={5}
E(6)={235678}
E(7)={23578}
E(8)={8}
E(9)={9}
E(10)={10}
E(11)={11}
DFA Table is:

a	b	
{12358}	{2345789}	{235678}
{2345789}	{2345789}	{23567810}
{235678}	{2345789}	{235678}
{23567810}	{2345789}	{23567811}
{23567811}	{2345789}	{235678}

Experiment 8: Computation of Leading Sets

Aim:

Write a program in C/C++ to detect the leading edges of the given set of productions of a grammar.

Algorithm:

1. Start the program.
2. Get the Set of Productions for the grammar from the user. No redundant & cyclic productions must be given.
3. The conditions to be checked are:

Conditions	Inclusions in result
S->Sa	add a
S->Aa	add a, production of A
S->ab	add a
S->AB	Production of A
S->SA	none
S->a	take a
S->SA*	none taken
S->*a	take * leave a

4. Print the Leading edges.
5. Stop the program.

Program (leading.cpp):

```
/* Leading Edges */
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>

char av[100],av1[100];
int v=0,j=0,v1=0;

void disp(int);

struct pro
{
     char h,t,t1;
}p[100];

int search(char x)
{
     for(int i=0;i<v;i++)
          if(av[i]==x) return 1;
     return 0;
}

int search1(char x)
{
```

```
     for(int i=0;i<v1;i++)
          if(av1[i]==x) return 1;
     return 0;
}

void disp1(char x)
{
     for(int i=0;i<j;i++)
          if(p[i].h==x) disp(i);
}

void disp(int px)
{
     if(int(p[px].t)>=65 && int(p[px].t)<=90)
     {
          if(p[px].t1!='\0' && search1(p[px].t1)==0)
          {
               if(p[px].t1!='\n')
                    cout<<p[px].t1;
               av1[v1]=p[px].t1;
               v1++;
          }
          disp1(p[px].t);
     }
     else if(p[px].t!='#')
     {
          if(search1(p[px].t)==0)
          {
               cout<<"\t"<<p[px].t;
               av1[v1]=p[px].t;
               v1++;
          }
     }
}

void main()
{
     clrscr();
     cout<<"Enter the production: end with ~"<<endl<<endl;
     char a1[100];
     for(int i=0;(a1[i]=getc(stdin))!='~';i++);

     a1[i]='\0';
     clrscr();
     cout<<a1;

     for(int k=0;k<i;k++)
     {
          if(a1[k]=='-' && a1[k+1]=='>')
          {
               p[j].h=a1[k-1];
```

```
                p[j].t=a1[k+2];
                p[j].t1='\0';

                if(p[j].h==p[j].t)
                {
                     p[j].t=a1[k+3];
                     if(int(p[j].t)>=65 && int(p[j].t)<=90)
                          p[j].t='#';
                     p[j].t1='\0';
                }
                else if(int(p[j].t)>=65 && int(p[j].t)<=90)
                {
                     p[j].t1=a1[k+3];
                     if((int(p[j].t1)>=65) &&
(int(p[j].t1)<=90))
                          p[j].t1='\0';
                }
                j++;
           }
     }
     cout<<endl<<"The Leading edges r as follows: "<<endl;
     for(i=0;i<j;i++)
     {
          if(search(p[j].h)==0)
          {
                av[v]=p[i].h;
                cout<<endl<<av[v]<<": {";
                disp1(av[v]);
                cout<<"   }"<<endl<<endl;

                for(k=0;k<v1;k++)
                     av1[k]='\0';
                v1=0;
                v++;
          }
     }
     getch();
}
```

Output:

Enter the production: end with ~

S->(L)
S->a
L->L,S
L->S
~

S->(L)

S->a
L->L,S
L->S

The Leading edges r as follows:

S: { (a }

S: { (a }

L: { , (a }

L: { , (a }

Experiment 9: Computation of Trailing Sets

Aim:

Write a program in C/C++ to detect the trailing edges of the given set of productions of a grammar.

Algorithm:

1. Start the program.
2. Get the Set of Productions for the grammar from the user. No redundant & cyclic productions must be given.
3. Reverse each input productions and print it.
4. The conditions to be checked according to the reversed inputs are:

Conditions	Inclusions in result
S->Sa	add a
S->Aa	add a, production of A
S->ab	add a
S->AB	Production of A
S->SA	none
S->a	take a
S->SA*	none taken
S->*a	take * leave a

5. Print the Trailing edges.
6. Stop the program.

Program (trailing.cpp):

```
/* Trailing Edges */
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>

char b1[100];
char a1[100];
char av[100],av1[100];
int v=0,j=0,v1=0;

void disp(int);

struct pro
{
     char h,t,t1;
}p[100];

void revpro(int l)
{
     int k1,k2,j;
     for(int i=0;i<=l;i++)
     {
          a1[i]=b1[i];
```

```
          if(b1[i]=='>')
          {
               for(j=i+1;;j++)
               {
                    if(int(b1[j])==10)
                    {
                         a1[j]=b1[j];
                         break;
                    }
               }
               for(k1=i+1,k2=j-1;k1<j;k1++,k2--)
                    a1[k1]=b1[k2];
               i=j;
          }
     }
}


int search(char x)
{
     for(int i=0;i<v;i++)
          if(av[i]==x) return 1;
     return 0;
}

int search1(char x)
{
     for(int i=0;i<v1;i++)
          if(av1[i]==x) return 1;
     return 0;
}

void disp1(char x)
{
          for(int i=0;i<j;i++)
          if(p[i].h==x) disp(i);
}

void disp(int px)
{
     if(int(p[px].t)>=65 && int(p[px].t)<=90)
     {
          if(p[px].t1!='\0' && search1(p[px].t1)==0)
          {
               if(p[px].t1!='\n')
                    cout<<p[px].t1;
               av1[v1]=p[px].t1;
               v1++;
          }
          disp1(p[px].t);
     }
```

```
     else if(p[px].t!='#')
     {
          if(search1(p[px].t)==0)
          {
               cout<<"\t"<<p[px].t;
               av1[v1]=p[px].t;
               v1++;
          }
     }
}
void main()
{
     clrscr();
     cout<<"Enter the production: end with ~"<<endl<<endl;
     for(int i=0;(b1[i]=getc(stdin))!='~';i++);

     b1[i]='\0';
     revpro(i);
     clrscr();
     cout<<a1;

     for(int k=0;k<i;k++)
     {
          if(a1[k]=='-' && a1[k+1]=='>')
          {
               p[j].h=a1[k-1];
               p[j].t=a1[k+2];
               p[j].t1='\0';

               if(p[j].h==p[j].t)
               {
                    p[j].t=a1[k+3];
                    if(int(p[j].t)>=65 && int(p[j].t)<=90)
                         p[j].t='#';
                    p[j].t1='\0';
               }
               else if(int(p[j].t)>=65 && int(p[j].t)<=90)
               {
                    p[j].t1=a1[k+3];
                    if((int(p[j].t1)>=65) &&
(int(p[j].t1)<=90))
                         p[j].t1='\0';
               }
               j++;
          }
     }
     cout<<endl<<"The Trailing edges r as follows: "<<endl;
     for(i=0;i<j;i++)
     {
          if(search(p[j].h)==0)
          {
```

```
                av[v]=p[i].h;
                cout<<endl<<av[v]<<": {";
                disp1(av[v]);
                cout<<"   }"<<endl<<endl;

                for(k=0;k<v1;k++)
                     av1[k]='\0';
                v1=0;
                v++;
           }
      }
      getch();
}
```

Output:

Enter the production: end with ~

S->(L)
S->a
L->L,S
L->S
~

S->)L(
S->a
L->S,L
L->S

The Trailing edges r as follows:
S: {) a }

S: {) a }

L: {,) a }

L: {,) a }

Experiment 10: Implementation of Shift Reduce Parsing

Aim:

Write a program in C/C++ to implement the shift reduce parsing.

Algorithm:

1. Start the Process.
2. Symbols from the input are shifted onto stack until a handle appears on top of the stack.
3. The Symbols that are the handle on top of the stack are then replaces by the left hand side of the production (reduced).
4. If this result in another handle on top of the stack, then another reduction is done, otherwise we go back to shifting.
5. This combination of shifting input symbols onto the stack and reducing productions when handles appear on the top of the stack continues until all of the input is consumed and the goal symbol is the only thing on the stack - the input is then accepted.
6. If we reach the end of the input and cannot reduce the stack to the goal symbol, the input is rejected.
7. Stop the process.

Program (srp.cpp):

```
/* Shift Reduce Parsing */
#include<stdio.h>
#include<conio.h>
#include<string.h>

void check();
void check1();
void copy();
void print(int val);


char stack[20];
char temp[10];
char result[10];
int i,j;

void main()
{
   clrscr();
   printf("Enter Your Expression:");
   scanf("%s",&stack);
   check();
   getch();

}
void check()
{
```

```
    for(;i<strlen(stack)+1;i++)
    {
       if(stack[i]=='+' || stack[i]=='-' || stack[i]=='*'
|| stack[i]=='/'|| stack[i]=='\0')
        {
      temp[j]='E';
      j++;
      temp[j]=stack[i];
      j++;
       }
    }
    check1();
}
void check1()
{
    printf("\n     STACK VALUES\tINPUT    \n");

l: for(j=0,i=0;i<strlen(temp);)
   {
    if(temp[i]=='+' || temp[i]=='-' || temp[i]=='*' ||
temp[i]=='/')
    {
     printf("\n\t %c",temp[i]);
     i++;
     print(i);
     printf("\n\t %c",temp[i]);
     i++;
     print(i);
     i--;
     copy();
     goto l;
    }
    else
    {
     printf("\n\t %c",temp[i]);
     i++;
     print(i);
    }
   }
    printf("\n\n\t Expressions Output:%s",temp);
}
void copy()
{
  j=0;
  while(temp[i]!='\0')
  {
   temp[j]=temp[i];
   j++;
   i++;
  }
  temp[j]='\0';
```

```
}
void print(int val)
{
   printf("\t\t");
   for(;val<strlen(temp);val++)
    printf("%c",temp[val]);

}
```

Output:

```
Enter Your Expression:E+E*E-E

     STACK VALUES              INPUT

         E                     +E*E-E
         +                      E*E-E
         E                     *E-E
         E                     *E-E
         *                     E-E
         E                     -E
         E                     -E
         -                     E
         E
         E

       Expressions Output: E
```

Experiment 11: Intermediate Code Generation

Aim:

Write a program in C/C++ to generate intermediate code from a given syntax tree statement.

Algorithm:

1. Start the process.
2. Input an expression EXP from user.
3. Process the expression from right hand side to left hand side.
4. FLAG:=0; TOP = -1;
5. IF EXP = '=' then
   - i. IF EXP(index – 1) = 0 then
     1. PRINT EXP element from index to (index – 1) and POP STACK[TOP]. Terminate

   Else
   - i. PRINT Wrong Expression

   [EndIF]
   IF an operator is found and FLAG = 0 then
   - i. TOP:= TOP + 1
   - ii. add to STACK[TOP].
   - iii. FLAG:=1

   Else
   - i. pop twice the STACK and result add to the newID(identifier) and PRINT.
   - ii. TOP:=TOP-2. Save newID to STACK[TOP]
   - iii. FLAG:=0

   [EndIF]
6. IF an operand is found then
   - i. TOP:=TOP+1
   - ii. move to STACK [TOP]
   - iii. IF TOP > 1 then
     1. pop twice the STACK and result add to the newID(identifier) and PRINT.
     2. TOP:=TOP-2. Save newID to STACK[TOP]
     3. FLAG:=0

   [End]
7. End the process

Program (icgen.cpp):

```
/* Intermediate Code Generator */
// Here consideration is any input expression
// only contain digits at the end
#include<iostream.h>
#include<stdio.h>
#include<conio.h>
#include<string.h>
```

```
#include<ctype.h>

void main()
{
     char g,exp[20],stack[20];
     int m=0,i,top=-1,flag=0,len,j;
     cout<<"\nInput an expression : ";
     gets(exp);
     cout<<"\nIntermediate code generator\n";
     len=strlen(exp);

     //If expression contain digits
     if(isdigit(exp[len-1]))
     {
          cout<<"T = inttoreal(";
          i=len-1;
          while(isdigit(exp[i]))
          {
               i--;
          }
          for(j=i+1;j<len;j++)
          {
               cout<<exp[j];
          }
          cout<<".0)\n";
          exp[i+1]='T';len=i+2;
     }
     else         //If expression having no digit
     {
          cout<<"T = "<<exp[len-1]<<"\n";
          exp[len-1]='T';
     }
     for(i=len-1;i>=0;i--)
     {
          if(exp[i]=='=')
          {
               if((i-1)==0)
               {
                    // If expression contains unary
operator in RHS near = operator
                    if(isalpha(stack[top]))
                    {
                    cout<<exp[i-1]<<" "<<exp[i]<<"
"<<stack[top];
                    }
                    else
                    {
                    cout<<exp[i-1]<<" "<<exp[i]<<"
"<<stack[top]<<stack[top-1];
                    }
                    break;
```

```
                }
                else
                {
                     cout<<"\nWrong Expression !!!";
                     break;
                }
           }

     if(exp[i]=='+'||exp[i]=='/'||exp[i]=='*'||exp[i]=='-
'||exp[i]=='%')
           {
                if(flag==0)
                {
                     flag=1;top=top+1;
                     stack[top]=exp[i];
                }
                else
                {
                     g=char('A' + m);m++;
                     cout<<g<<" = "<<stack[top]<<stack[top-
1]<<"\n";
                     stack[top-1]=g;
                     stack[top]=exp[i];
                     flag=0;
                }
           }
           else
           {
                top=top+1;
                stack[top]=exp[i];
                if(top>1)
                {
                     g=char('A' + m);m++;
                     cout<<g<<" = "<<stack[top]<<stack[top-
1]<<stack[top-2]<<"\n";
                     top=top-2;
                     stack[top]=g;flag=0;
                }
           }
     }
}
```

Output:

Input an expression : a=b+c-6

Intermediate code generator
T = 6
A = c-T
B = b+A

a = B

Input an expression : d=e+f*-c%-a+k

Intermediate code generator
T = k
A = a+T
B = -A
C = c%B
D = -C
E = f*D
F = e+E
d = F

Experiment 12: Computation of FIRST Sets

Aim:

Write a program in C/C++ to find the FIRST set for a given set of production rule of a grammar.

Algorithm:

Procedure First

1. Input the number of production N.
2. Input all the production rule *PArray*
3. Repeat steps a, b, c until process all input production rule i.e. *PArray*[N]
    a. If $X_i \neq X_{i+1}$ then
        i. Print Result array of X_i which contain FIRST(X_i)
    b. If first element of X_i of *PArray* is Terminal or ε Then
        i. Add Result = Result U first element
    c. If first element of X_i of *PArray* is Non-Terminal Then
        i. searchFirst(i, *PArray*, N)
4. End Loop
5. If N (last production) then
    a. Print Result array of X_i which contain FIRST(X_i)
6. End

Procedure searchFirst(i, *PArray*, N)

1. Repeat steps Loop j=i+1 to N
    a. If first element of X_j of *PArray* is Non-Terminal Then
        i. searchFirst(j, of *PArray*, N)
    b. If first element of X_j of *PArray* is Terminal or ε Then
        i. Add Result = Result U first element
        ii. Flag=0
2. End Loop
3. If Flag = 0 Then
    a. Print Result array of X_j which contain FIRST(X_j)
4. End

Program:

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>

void searchFirst(int n, int i, char pl[], char r[], char
result[], int k)
{
     int j,flag;
     for(j=i+1;j<n;j++)
     {
          if(r[i]==pl[j])
          {
```

```
                if(isupper(r[j]))
                {
                     searchFirst(n,j,pl,r,result,k);
                }
                if(islower(r[j]) || r[j]== '+' || r[j]=='*'
|| r[j]==')' || r[j]=='(')
                {
                     result[k++]=r[j];
                     result[k++]=','; flag=0;
                }
           }
     }
     if(flag==0)
     {
          for(j=0;j<k-1;j++)cout<<result[j];
     }
}
void main()
{
     clrscr();
     char pr[10][10],pl[10],r[10],prev,result[10];
     int i,n,k,j;
     cout<<"\nHow many production rule : ";
     cin>>n;
     if(n==0) exit(0);
     for(i=0;i<n;i++)
     {
          cout<<"\nInput left part of production rules : ";
          cin>>pl[i];
          cout<<"\nInput right part of production rules :
";
          gets(pr[i]);
          r[i]=pr[i][0];
     }
     cout<<"\nProduction Rules are : \n";
     for(i=0;i<n;i++)
     {
          cout<<pl[i]<<"-
>"<<pr[i]<<"\n";//<<";"<<r[i]<<"\n";
     }
     cout<<"\n----O U T P U T---\n\n";
     prev=pl[0];k=0;
     for(i=0;i<n;i++)
     {
          if(prev!=pl[i])
          {
               cout<<"\nFIRST("<<prev<<")={";
               for(j=0;j<k-1;j++)cout<<result[j];
               cout<<"}";
               k=0;prev=pl[i];
               //cout<<"\n3";
```

```
            }
            if(prev==pl[i])
            {
                  if(islower(r[i]) || r[i]== '+' || r[i]=='*'
|| r[i]==')' || r[i]=='(')
                  {
                        result[k++]=r[i];
                        result[k++]=',';
                  }
                  if(isupper(r[i]))
                  {
                        cout<<"\nFIRST("<<prev<<")={";
                        searchFirst(n,i,pl,r,result,k);
                        cout<<"}";
                        k=0;prev=pl[i+1];
                  }
            }
      }
      if(i==n)
      {
            cout<<"\nFIRST("<<prev<<")={";
            for(j=0;j<k-1;j++)cout<<result[j];
            cout<<"}";
            k=0;prev=pl[i];
      }
      getch();
}
```

Output:

How many production rule : 8

Input left part of production rules : E
Input right part of production rules : TX
Input left part of production rules : X
Input right part of production rules : +TX
Input left part of production rules : X
Input right part of production rules : e
Input left part of production rules : T
Input right part of production rules : FY
Input left part of production rules : Y
Input right part of production rules : *FY
Input left part of production rules : Y
Input right part of production rules : e
Input left part of production rules : F
Input right part of production rules : (E)
Input left part of production rules : F
Input right part of production rules : i

```
Production Rules are :
E->TX
X->+TX
X->e
T->FY
Y->*FY
Y->e
F->(E)
F->i

----O U T P U T---

FIRST(E)={(,i}
FIRST(X)={+,e}
FIRST(T)={(,i}
FIRST(Y)={*,e}
FIRST(F)={(,i}




How many production rule : 5

Input left part of production rules : E
Input right part of production rules : aTX
Input left part of production rules : E
Input right part of production rules : TX
Input left part of production rules : T
Input right part of production rules : FY
Input left part of production rules : F
Input right part of production rules : (E)
Input left part of production rules : F
Input right part of production rules : i

Production Rules are :
E->aTX
E->TX
T->FY
F->(E)
F->i

----O U T P U T---


FIRST(E)={a,(,i}
FIRST(T)={(,i}
FIRST(F)={(,i}
```

Experiment 13: Computation of FOLLOW Sets

Aim:

Write a program in C/C++ to find a FOLLOW set from a given set of production rule.

Algorithm:

1. Declare the variables.
2. Enter the production rules for the grammar.
3. Calculate the FOLLOW set for each element call the user defined function follow().
4. If x->aBb
    a. If x is start symbol then FOLLOW(x)={$}.
    b. If b is NULL then FOLLOW(B)=FOLLOW(x).
    c. If b is not NULL then FOLLOW(B)=FIRST(b).

   END.

Program:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
int n,m=0,p,i=0,j=0;
char a[10][10],f[10];
void follow(char c);
void first(char c);
void main()
{clrscr();
int i,z;
char c,ch;
14
printf("Enter the no.of productions:");
scanf("%d",&n);
printf("Enter the productions(epsilon=$):\n");
for(i=0;i<n;i++)
scanf("%s%c",a[i],&ch);
do
{
m=0;
printf("Enter the element whose FOLLOW is to be found:");
scanf("%c",&c);
follow(c);
printf("FOLLOW(%c) = { ",c);
for(i=0;i<m;i++)
printf("%c ",f[i]);
printf(" }\n");
printf("Do you want to continue(0/1)?");
scanf("%d%c",&z,&ch);
}
while(z==1);
}
void follow(char c)
```

```
{
if(a[0][0]==c)f[m++]='$';
for(i=0;i<n;i++)
15
{
for(j=2;j<strlen(a[i]);j++)
{
if(a[i][j]==c)
{
if(a[i][j+1]!='\0')first(a[i][j+1]);
if(a[i][j+1]=='\0'&&c!=a[i][0])
follow(a[i][0]);
}
}
}
}
void first(char c)
{
int k;
if(!(isupper(c)))f[m++]=c;
for(k=0;k<n;k++)
{
if(a[k][0]==c)
{
if(a[k][2]=='$') follow(a[i][0]);
else if(islower(a[k][2]))f[m++]=a[k][2];
else first(a[k][2]);
}
}
}
```

Output:

```
Enter the no of productions :3
Enter the production (epsion =$):
E=E+T
T=T*F
F=a
Enter the element whose FOLLOW is to be found ;E
FOLLOW(E)={$ +}
Do you want to continue (0/1)?1
Enter the element whose FOLLOW is to be found ;T
FOLLOW(T)={$ +}
Do you want to continue (0/1)?1
Enter the element whose FOLLOW is to be found ;F
FOLLOW(F)={$ +}
Do you want to continue (0/1)?0
```

Experiment 14: Implement Loader

Aim:

Write a program in c/c++ to implement a loader .

Program:

```
#iinclude<stdio.h>
#include<conio.h>
#include<stdlib.h>
Void main ()
{
FILE *f1,*f2;
unsigned int str;
char a[10],b[10];
printf ("enter the starting address ");
scanf("%u",&str);
f1=fopen("ref.txt","r");
f2=fopen("output.txt","w");
while(!feof(f1))
{
fscanf(f1,"%s%s",a,b);
fprintf(f2,"\t%u\t%s\n",str,b);
str++;
}
fclose(f2);
fclose(f1);
```

Output:

```
input
//ref.txt
LDA 00
LDCH 50
LDX 04
JLT 38
ADD 18
OUTPUT
Enter the starting address :4000
//4000 00
4001 50
4002 04
4003 38
4004 18
```

Experiment 15: Demonstrate Operator Precedence Parsing

Aim:

Write a program in C/C++ to demonstrate operator precedence parsing.

Algorithm:

1. Declare the variables: Tokens, Action and Parse table.
2. The rules that are used in the operator precedence parsing are:
    a. If an operator "opp1" has higher precedence than an operator "opp2", i.e. if * has higher precedence than + make * > + & + < *. In an expression of the form E+E*E+E the central E*E is the handle & will be computed first.
    b. If "opp1" & "opp2" have equal precedence that is opp1=opp2 then in an expression of the form E(exp)E(exp)E, the last part E(exp)E will act as a handle & will be computed first.
    c. The precedence order of operators in the increasing order is given as: + , - < * , / < (exp)
3. Blanks are denoted as error entries.
4. End.

Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef enum { false , true } bool;

/* actions */
typedef enum {
    S,                  /* shift */
    R,                  /* reduce */
    A,                  /* accept */
    E1,                 /* error: missing right parenthesis */
    E2,                 /* error: missing operator */
    E3,                 /* error: unbalanced right parenthesis */
    E4                  /* error: invalid function argument */
} actEnum;

/* tokens */
typedef enum {
    /* operators */
    tAdd,                /* + */
    tSub,                /* - */
    tMul,                /* * */
    tDiv,                /* / */
    tPow,                /* ^ (power) */
    tUmi,                /* - (unary minus) */
    tFact,               /* f(x): factorial */
    tPerm,               /* p(n,r): permutations, n objects, r at
a time */
    tComb,               /* c(n,r): combinations, n objects, r at
a time */
```

```
    tComa,              /* comma */
    tLpr,               /* ( */
    tRpr,               /* ) */
    tEof,               /* end of string */
    tMaxOp,             /* maximum number of operators */
    /* non-operators */
    tVal                /* value */
} tokEnum;

tokEnum tok;            /* token */
double tokval;          /* token value */

#define MAX_OPR         50
#define MAX_VAL         50
char opr[MAX_OPR];      /* operator stack */
double val[MAX_VAL];    /* value stack */
int oprTop, valTop;     /* top of operator, value stack */
bool firsttok;          /* true if first token */

char parseTbl[tMaxOp][tMaxOp] = {
    /* stk      ------------------ input ------------------------
*/
    /*         +   -   *   /   ^   M   f   p   c   ,   (   )   $
*/
    /*         --  --  --  --  --  --  --  --  --  --  --  --  --
*/
    /* + */  { R,  R,  S,  S,  S,  S,  S,  S,  S,  R,  S,  R,  R
},
    /* - */  { R,  R,  S,  S,  S,  S,  S,  S,  S,  R,  S,  R,  R
},
    /* * */  { R,  R,  R,  R,  S,  S,  S,  S,  S,  R,  S,  R,  R
},
    /* / */  { R,  R,  R,  R,  S,  S,  S,  S,  S,  R,  S,  R,  R
},
    /* ^ */  { R,  R,  R,  R,  S,  S,  S,  S,  S,  R,  S,  R,  R
},
    /* M */  { R,  R,  R,  R,  R,  S,  S,  S,  S,  R,  S,  R,  R
},
    /* f */  { R,  R,  R,  R,  R,  R,  R,  R,  R,  R,  S,  R,  R
},
    /* p */  { R,  R,  R,  R,  R,  R,  R,  R,  R,  R,  S,  R,  R
},
    /* c */  { R,  R,  R,  R,  R,  R,  R,  R,  R,  R,  S,  R,  R
},
    /* , */  { R,  R,  R,  R,  R,  R,  R,  R,  R,  R,  R,  R,
E4},
    /* ( */  { S,  S,  S,  S,  S,  S,  S,  S,  S,  S,  S,  S,
E1},
    /* ) */  { R,  R,  R,  R,  R,  R,  E3, E3, E3, R,  E2, R,  R
},
    /* $ */  { S,  S,  S,  S,  S,  S,  S,  S,  S,  E4, S,  E3, A
}
};

int error(char *msg) {
```

```
    printf("error: %s\n", msg);
    return 1;
}

int gettok(void) {
    static char str[82];
    static tokEnum prevtok;
    char *s;

    /* scan for next symbol */
    if (firsttok) {
        firsttok = false;
        prevtok = tEof;
        gets(str);
        if (*str == 'q') exit(0);
        s = strtok(str, " ");
    } else {
        s = strtok(NULL, " ");
    }

    /* convert symbol to token */
    if (s) {
        switch(*s) {
        case '+': tok = tAdd; break;
        case '-': tok = tSub; break;
        case '*': tok = tMul; break;
        case '/': tok = tDiv; break;
        case '^': tok = tPow; break;
        case '(': tok = tLpr; break;
        case ')': tok = tRpr; break;
        case ',': tok = tComa; break;
        case 'f': tok = tFact; break;
        case 'p': tok = tPerm; break;
        case 'c': tok = tComb; break;
        default:
            tokval = atof(s);
            tok = tVal;
            break;
        }
    } else {
        tok = tEof;
    }

    /* check for unary minus */
    if (tok == tSub) {
        if (prevtok != tVal && prevtok != tRpr) {
            tok = tUmi;
        }
    }

    prevtok = tok;
    return 0;
}

int shift(void) {
```

```
    if (tok == tVal) {
        if (++valTop >= MAX_VAL)
            return error("val stack exhausted");
        val[valTop] = tokval;
    } else {
        if (++oprTop >= MAX_OPR)
            return error("opr stack exhausted");
        opr[oprTop] = (char)tok;
    }
    if (gettok()) return 1;
    return 0;
}

double fact(double n) {
    double i, t;
    for (t = 1, i = 1; i <= n; i++)
        t *= i;
    return t;
}

int reduce(void) {
    switch(opr[oprTop]) {
    case tAdd:
        /* apply E := E + E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] + val[valTop];
        valTop--;
        break;
    case tSub:
        /* apply E := E - E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] - val[valTop];
        valTop--;
        break;
    case tMul:
        /* apply E := E * E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] * val[valTop];
        valTop--;
        break;
    case tDiv:
        /* apply E := E / E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = val[valTop-1] / val[valTop];
        valTop--;
        break;
    case tUmi:
        /* apply E := -E */
        if (valTop < 0) return error("syntax error");
        val[valTop] = -val[valTop];
        break;
    case tPow:
        /* apply E := E ^ E */
        if (valTop < 1) return error("syntax error");
        val[valTop-1] = pow(val[valTop-1], val[valTop]);
```

```
            valTop--;
            break;
        case tFact:
            /* apply E := f(E) */
            if (valTop < 0) return error("syntax error");
            val[valTop] = fact(val[valTop]);
            break;
        case tPerm:
            /* apply E := p(N,R) */
            if (valTop < 1) return error("syntax error");
            val[valTop-1] = fact(val[valTop-1])/fact(val[valTop-1]-
val[valTop]);
            valTop--;
            break;
        case tComb:
            /* apply E := c(N,R) */
            if (valTop < 1) return error("syntax error");
            val[valTop-1] = fact(val[valTop-1])/
                (fact(val[valTop]) * fact(val[valTop-1]-
val[valTop]));
            valTop--;
            break;
        case tRpr:
            /* pop () off stack */
            oprTop--;
            break;
        }
        oprTop--;
        return 0;
}

int parse(void) {

    printf("\nenter expression (q to quit):\n");

    /* initialize for next expression */
    oprTop = 0; valTop = -1;
    opr[oprTop] = tEof;
    firsttok = true;
    if (gettok()) return 1;

    while(1) {

        /* input is value */
        if (tok == tVal) {
            /* shift token to value stack */
            if (shift()) return 1;
            continue;
        }

        /* input is operator */
        switch(parseTbl[opr[oprTop]][tok]) {
        case R:
            if (reduce()) return 1;
            break;
```

```
        case S:
            if (shift()) return 1;
            break;
        case A:
            /* accept */
            if (valTop != 0) return error("syntax error");
            printf("value = %f\n", val[valTop]);
            return 0;
        case E1:
            return error("missing right parenthesis");
        case E2:
            return error("missing operator");
        case E3:
            return error("unbalanced right parenthesis");
        case E4:
            return error("invalid function argument");
        }
    }
}

void main() {
    while(1) parse();
}
```

Output:

```
enter expression (q to quit):
4 + 5 ^ 2
value = 29.000000

enter expression (q to quit):
q
```

Experiment 16: Construct Predictive Parser Table

Aim:

Write a program in C/C++ to construct Predictive Parser Table.

Algorithm:

INPUT: Grammar G.
OUTPUT: Parsing table M.
METHOD:

1. For each production $A \rightarrow \alpha$ of the grammar, do the following:
   a. For each terminal a in FIRST(A), add $A \rightarrow \alpha$ to M[A, a] .
   b. If $\in$ is in FIRST(α) , then for each terminal b in FOLLOW(A) , add $A \rightarrow \alpha$ to M [A, b] . If $\in$ is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $] as well.
2. If, after performing the above, there is no production at all in M[A, a], then set M[A, a] to error (which we normally represent by an empty entry in the table).
3. End.

Program:

```
#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];
l++;
q3:
}
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();
}
```

Output:

```
Enter the no. of co-ordinates
2
Enter the productions in a grammar
S->CC
C->eC | d
First pos
FIRS[S] = ed
FIRS[C] = ed

Follow pos
```

```
FOLLOW[S] =$
FOLLOW[C] =ed$

M [S , e] =S->CC
M [S , d] =S->CC
M [C , e] =C->eC
M [C , d] =C->d
```

Experiment 1: Using Lex accept only digit

Aim:

Write a program in Lex to accept only digit.

Program:

```
%{
#include <stdio.h>
%}

%%
[0-9]+  { printf("%s\n", yytext); }
.|\n {}
%%

int main( )
{
yylex( );
}

int yywrap( )
{
return 1;
}
```

Output:

```
C:\Dev-Cpp>flex digitLex.l
C:\Dev-Cpp>gcc lex.yy.c -o digitLex.exe
C:\Dev-Cpp>digitLex
1234
1234
qwe
123df
123
asd34
34

C:\Dev-Cpp>
```

Experiment 2: Using Lex, count character, word and newline

Aim:

Write a program in Lex to count character, word and newline in an input sentence.

Program:

```
%{
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%}

delim     [ \t]+

%%
\n        { ch++; wd++; nl++; }
^{delim}  { ch+=yyleng; }
{delim}   { ch+=yyleng; wd++; }
.         { ch++; }
%%

int main()
{
yylex();
printf("%8d%8d%8d\n", nl, wd, ch);
}

int yywrap()
{
return 1;
}
```

Output:

```
C:\Dev-Cpp>flex wordCount.l
C:\Dev-Cpp>gcc lex.yy.c -o wordCount.exe
C:\Dev-Cpp>wordCount
unified process is not a fixed
series of steps for constructing a
software product. Its an adaptable methodology
process has to be modified for specific
```

```
software product to be developed.
188   30   5

C:\Dev-Cpp>
```

Experiment 3: Using Lex, identify identifier, number and other character

Aim:

Write a program in Lex to identify identifier, number and other character.

Program:

```
%{
#include <stdio.h>
%}

digit     [0-9]
letter    [A-Za-z]
id        {letter}({letter}|{digit})*

%%
{digit}+  { printf("number: %s\n", yytext); }
{id}      { printf("ident: %s\n", yytext); }
.         { printf("other: %s\n", yytext); }
%%

int main()
{
yylex();
}

int yywrap()
{ return 1;
}
```

Output:

```
C:\Dev-Cpp>flex otherChar.l

C:\Dev-Cpp>gcc lex.yy.c -o otherChar.exe

C:\Dev-Cpp>otherChar
abcd123
identifier: abcd123

123
number: 123
```

```
123d
number: 123
identifier: d

%#
other: %
other: #


C:\Dev-Cpp>
```

Experiment 4: Using Lex, Convert vowel to Uppercase

Aim:

Write a program in Lex to convert lowercase vowel character to uppercase character in a sentence.

Program:

```
%{
#include<stdio.h>
int cc=0, vc=0;
%}
vowel    [aeiou]
%%
{vowel}  {
     vc++;
     printf("%c",toupper(yytext[0]));
     }
     {
     printf("%c",yytext[0]);
     }
%%
int yywrap()
{
printf("End of file reached");
return;
}

int main()
{
     yylex();
     printf("No. of vowels = %d \n", vc);
     return 1;
}
```

Output:

```
C:\>cd Dev-Cpp

C:\Dev-Cpp>flex upperCase.l

C:\Dev-Cpp>gcc lex.yy.c -o upperCase.exe

C:\Dev-Cpp>upperCase
the quick brown fox jump over the lazy dog.
```

```
thE qUIck brOwn fOx jUmp OvEr thE lAzy dOg.


this is a text to see the conversetion.
thIs Is A tExt tO sEE thE cOnvErsEtIOn.
End of file reachedNo. of vowels = 24


C:\Dev-Cpp>
```

Experiment 5: Using Lex, demonstrate Lexical Analyzer

Aim:

Write a program in Lex to demonstrate lexical analyzer.

Program:

```
%{
%}

%%
"="{printf("\n%s\tOperator is
ASSIGNMENT",yytext);}

"++"{printf("\n%s\tOperator is
INCREMENT",yytext);}

"--"{printf("\n%s\tOperator is
DECREMENT",yytext);}

"+="{printf("\n%s\tOperator is INCREMENT and
ASSIGN",yytext);}

"-="{printf("\n%s\tOperator is DECREMENT and
ASSIGN",yytext);}

"+"{printf("\n%s\tOperator is
PLUS",yytext);}

"*"{printf("\n%s\tOperator is
MULTIPLICATION",yytext);}

"-"{printf("\n%s\tOperator is
MINUS",yytext);}

"=="{printf("\n%s\tOperator is  EQUAL
TO",yytext);}
"/"{printf("\n%s\tOperator is
DIVISION",yytext);}
```

```
"<"{printf("\n%s\tOperator is LESS
THEN",yytext);}

">"{printf("\n%s\tOperator is GREATER
THEN",yytext);}

">="{printf("\n%s\tOperator is GREATER THEN
EQUAL TO",yytext);}

"<="{printf("\n%s\tOperator is LESS THEN
EQUAL TO",yytext);}

"!="{printf("\n%s\tOperator is NOT EQUAL
TO",yytext);}

"," {printf("\n%s\tComma",yytext);}
";"{printf("\n%s\tSemi Colon",yytext);}
"("     {printf("\n%s\tBraces",yytext);}
")"{printf("\n%s\tBraces",yytext);}
"if"{printf("\n%s\tKEYWORD",yytext);}
"while"{printf("\n%s\tKEYWORD",yytext);}
"for"{printf("\n%s\tKEYWORD",yytext);}
"float"{printf("\n%s\tKEYWORD",yytext);}
"char"{printf("\n%s\tKEYWORD",yytext);}
"int"{printf("\n%s\tKEYWORD",yytext);}
[-]*[0-9]*{printf("\n%s\tConstant",yytext);}
[a-zA-Z]*[\ ]*
{printf("\n%s\tIdentifier",yytext);}
.return yytext[0];
\n return 0;
%%
int main()
{
yylex();
}

int yywrap()
```

```
{
return 1;
}
```

Output:

```
C:\Dev-Cpp>flex operator.l

C:\Dev-Cpp>gcc lex.yy.c -o operator.exe

C:\Dev-Cpp>operator
for(i=10;i>=1;i--)

for     KEYWORD
(       Braces
i       Identifier
=       Operator is ASSIGNMENT
10      Constant
;       Semi Colon
i       Identifier
>=      Operator is GREATER THEN EQUAL TO
1       Constant
;       Semi Colon
i       Identifier
--      Operator is DECREMENT
)       Braces
C:\Dev-Cpp>
```