1.Lex program for checking valid mail id.

```
%{
#include<stdio.h>
%}
%%
^[_a-z][_a-z0-9]*[@]("gmail.com"|"yahoo.in") {printf("Valid email");}
.* {printf("Invalid email");}
\n {return 0;}
%%
void main()
{
printf("Enter the email");
yylex();
}
int yywrap()
{
return 1;
}
```

2.Lex program for checking valid Identifier

```
%{
#include<stdlib.h>
#include<stdio.h>
int flag=0;
%}
%%
^[_A-Za-z][A-Za-z0-9_]* {flag=1;}
.   ;
\n {return 0;}
%%
void main()
{
printf("Enter the string");
yylex();
if(flag==1)
printf("it is an identifier");
else
printf("not an identifier");
}
int yywrap()
{
return 1;
}
```

3.Lex Program for counting total number of vowels and consonants

```
%{
#include<stdlib.h>
#include<stdio.h>
int vowels=0,consonants=0,flag=1;
%}
%%
[AEIOUaeiou]  {vowels++;}
[A-Za-z] {consonants++;}
.   {flag=0;}
%%
main()
{
yyin=fopen("sample.txt","r");
yylex();
printf("no of vowels=%d\n",vowels);
printf("no of consonants=%d\n",consonants);

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

4.Lex program to display number of lines,words and characters

```
%{
  #include<stdio.h>
  int lines=0,words=0,letters=0,digit=0,sp_ch=0;
%}
%%
\n {lines++;words++;}
[\t ' '] {words++;}
[A-za-z] {letters++;}
[0-9] {digit++;}
. {sp_ch++;}
%%
main()
{
        //char str;
        //printf("enter input");
        //scanf("%c",str);
        yyin=fopen("te.txt","r");
        yylex();
```

```
        printf("lines=%d",lines);
        printf("words=%d",words);
        printf("letters=%d",letters);
        printf("digits=%d",digit);
        printf("special_letters=%d",sp_ch);

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

5.Lex program for accepting Strings starting with vowel

```
%{
    #include<stdio.h>
    int vowel=0;
%}

%%
^[aeiouAEIOU][a-zA-Z]* {printf("%s accepted\n",yytext);}
[a-zA-Z]* {printf("%s not accpted\n",yytext);}
\n {return 0;}
%%
int main()
{
    printf("Enter the string:");
    yylex();
}
int yywrap()
{
    return(1);
}
```

6.Conversion of NFA to DFA

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int st;
struct node *link;
};
struct node1
{
int nst[20];
};
void insert(int ,char, int);
int findalpha(char);
```

```
void findfinalstate(void);
int insertdfastate(struct node1);
int compare(struct node1,struct node1);
void printnewstate(struct node1);
static int set[20],nostate,noalpha,s,notransition,nofinal,start,finalstate[20],c,r,buffer[20];
int complete=-1;
char alphabet[20];
static int eclosure[20][20]={0};
struct node1 hash[20];
struct node * transition[20][20]={NULL};
void main()
{
int i,j,k,m,t,n,l;
struct node *temp;
struct node1 newstate={0},tmpstate={0};
printf("Enter the number of alphabets?\n");
printf("NOTE:- [ use letter e as epsilon]\n");
printf("NOTE:- [e must be last character ,if it is present]\n");
printf("\nEnter No of alphabets and alphabets?\n");
scanf("%d",&noalpha);
getchar();
for(i=0;i<noalpha;i++)
{
alphabet[i]=getchar();
getchar();
}
printf("Enter the number of states?\n");
scanf("%d",&nostate);
printf("Enter the start state?\n");
scanf("%d",&start);
printf("Enter the number of final states?\n");
scanf("%d",&nofinal);
printf("Enter the final states?\n");
for(i=0;i<nofinal;i++)
scanf("%d",&finalstate[i]);
printf("Enter no of transition?\n");
scanf("%d",&notransition);
printf("NOTE:- [Transition is in the form–> qno alphabet qno]\n",notransition);
printf("NOTE:- [States number must be greater than zero]\n");
printf("\nEnter transition?\n");
for(i=0;i<notransition;i++)
{
scanf("%d %c%d",&r,&c,&s);
insert(r,c,s);
}
for(i=0;i<20;i++)
{
for(j=0;j<20;j++)
```

```
hash[i].nst[j]=0;
}
complete=-1;
i=-1;
printf("\nEquivalent DFA.....\n");
printf("Transitions of DFA\n");
newstate.nst[start]=start;
insertdfastate(newstate);
while(i!=complete)
{
i++;
newstate=hash[i];
for(k=0;k<noalpha;k++)
{
c=0;
for(j=1;j<=nostate;j++)
set[j]=0;
for(j=1;j<=nostate;j++)
{
l=newstate.nst[j];
if(l!=0)
{
temp=transition[l][k];
while(temp!=NULL)
{
if(set[temp->st]==0)
{
c++;
set[temp->st]=temp->st;
}
temp=temp->link;
}
}

}
printf("\n");
if(c!=0)
{
for(m=1;m<=nostate;m++)
tmpstate.nst[m]=set[m];
insertdfastate(tmpstate);
printnewstate(newstate);
printf("%c\t",alphabet[k]);
printnewstate(tmpstate);
printf("\n");
}
else
{
```

```
printnewstate(newstate);
printf("%c\t", alphabet[k]);
printf("NULL\n");
}
}
}
printf("\nStates of DFA:\n");
for(i=0;i<=complete;i++)
printnewstate(hash[i]);
printf("\n Alphabets:\n");
for(i=0;i<noalpha;i++)
printf("%c\t",alphabet[i]);
printf("\n Start State:\n");
printf("q%d",start);
printf("\nFinal states:\n");
findfinalstate();
}
int insertdfastate(struct node1 newstate)
{
int i;
for(i=0;i<=complete;i++)
{
if(compare(hash[i],newstate))
return 0;
}
complete++;
hash[complete]=newstate;
return 1;
}
int compare(struct node1 a,struct node1 b)
{
int i;
for(i=1;i<=nostate;i++)
{
if(a.nst[i]!=b.nst[i])
return 0;
}
return 1;
}
void insert(int r,char c,int s)
{
int j;
struct node *temp;
j=findalpha(c);
if(j==999)
{
printf("error\n");
exit(0);
```

```
}
temp=(struct node *) malloc(sizeof(struct node));
temp->st=s;
temp->link=transition[r][j];
transition[r][j]=temp;
}
int findalpha(char c)
{
int i;
for(i=0;i<noalpha;i++)
if(alphabet[i]==c)
return i;
return(999);
}
void findfinalstate()
{
int i,j,k,t;
for(i=0;i<=complete;i++)
{
for(j=1;j<=nostate;j++)
{
for(k=0;k<nofinal;k++)
{
if(hash[i].nst[j]==finalstate[k])
{
printnewstate(hash[i]);
printf("\t");
j=nostate;
break;
}
}
}
}

}
void printnewstate(struct node1 state)
{
int j;
printf("{");
for(j=1;j<=nostate;j++)
{
if(state.nst[j]!=0)
printf("q%d,",state.nst[j]);
}
printf("}\t");}
```

7.WAP to minimize any given DFA

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

```
#define STATES 99
#define SYMBOLS 20
int N_symbols; /* number of input symbols */
int N_DFA_states; /* number of DFA states */
char *DFA_finals; /* final-state string */
int DFAtab[STATES][SYMBOLS];
char StateName[STATES][STATES+1]; /* state-name table */
int N_optDFA_states; /* number of optimized DFA states */
int OptDFA[STATES][SYMBOLS];
char NEW_finals[STATES+1];
/*
Print state-transition table.
State names: 'A', 'B', 'C', ...
*/
void print_dfa_table(
int tab[][SYMBOLS], /* DFA table */
int nstates, /* number of states */
int nsymbols, /* number of input symbols */
char *finals)
{
int i, j;
puts("\nDFA: STATE TRANSITION TABLE");
/* input symbols: '0', '1', ... */
printf(" | ");
for (i = 0; i < nsymbols; i++) printf(" %c ", '0'+i);
printf("\n-----+--");
for (i = 0; i < nsymbols; i++) printf("-----");
printf("\n");
for (i = 0; i < nstates; i++) {
printf(" %c | ", 'A'+i); /* state */
for (j = 0; j < nsymbols; j++)
printf(" %c ", tab[i][j]); /* next state */
printf("\n");
}
printf("Final states = %s\n", finals);
}
/*
Initialize NFA table.
*/
void load_DFA_table()
{
DFAtab[0][0] = 'B'; DFAtab[0][1] = 'C';
DFAtab[1][0] = 'E'; DFAtab[1][1] = 'F';
DFAtab[2][0] = 'A'; DFAtab[2][1] = 'A';
DFAtab[3][0] = 'F'; DFAtab[3][1] = 'E';
DFAtab[4][0] = 'D'; DFAtab[4][1] = 'F';
DFAtab[5][0] = 'D'; DFAtab[5][1] = 'E';
DFA_finals = "EF";
```

```
N_DFA_states = 6;
N_symbols = 2;
}
/* Get next-state string for current-state string.
*/
void get_next_state(char *nextstates, char *cur_states,
int dfa[STATES][SYMBOLS], int symbol)
{
int i, ch;
for (i = 0; i < strlen(cur_states); i++)
*nextstates++ = dfa[cur_states[i]-'A'][symbol];
*nextstates = '\0';
}
/* Get index of the equivalence states for state 'ch'.

Equiv. class id's are '0', '1', '2', ...
*/
char equiv_class_ndx(char ch, char stnt[][STATES+1], int n)
{
int i;
for (i = 0; i < n; i++)
if (strchr(stnt[i], ch)) return i+'0';
return -1; /* next state is NOT defined */
}
/*
Check if all the next states belongs to same equivalence class.
Return value:
If next state is NOT unique, return 0.
If next state is unique, return next state --> 'A/B/C/...'
's' is a '0/1' string: state-id's
*/
char is_one_nextstate(char *s)
{
char equiv_class; /* first equiv. class */
while (*s == '@') s++;
equiv_class = *s++; /* index of equiv. class */
while (*s) {
if (*s != '@' && *s != equiv_class) return 0;
s++;
}
return equiv_class; /* next state: char type */
}
int state_index(char *state, char stnt[][STATES+1], int n, int *pn,
int cur) /* 'cur' is added only for 'printf()' */
{
int i;
char state_flags[STATES+1]; /* next state info. */
if (!*state) return -1; /* no next state */
```

```
for (i = 0; i < strlen(state); i++)
state_flags[i] = equiv_class_ndx(state[i], stnt, n);
state_flags[i] = '\0';
printf(" %d:[%s]\t--> [%s] (%s)\n",
cur, stnt[cur], state, state_flags);
if (i=is_one_nextstate(state_flags))
return i-'0'; /* deterministic next states */
else {
strcpy(stnt[*pn], state_flags); /* state-division info */
return (*pn)++;
}
}
/*
Divide DFA states into finals and non-finals.
*/
int init_equiv_class(char statename[][STATES+1], int n, char *finals)
{
int i, j;
if (strlen(finals) == n) { /* all states are final states */
strcpy(statename[0], finals);
return 1;
}
strcpy(statename[1], finals); /* final state group */
for (i=j=0; i < n; i++) {
if (i == *finals-'A') {
finals++;
} else statename[0][j++] = i+'A';
}
statename[0][j] = '\0';
return 2;
}
/* Get optimized DFA 'newdfa' for equiv. class 'stnt'.
*/
int get_optimized_DFA(char stnt[][STATES+1], int n,
int dfa[][SYMBOLS], int n_sym, int newdfa[][SYMBOLS])
{
int n2=n; /* 'n' + <num. of state-division info> */
int i, j;
char nextstate[STATES+1];
for (i = 0; i < n; i++) { /* for each pseudo-DFA state */
for (j = 0; j < n_sym; j++) { /* for each input symbol */
get_next_state(nextstate, stnt[i], dfa, j);
newdfa[i][j] = state_index(nextstate, stnt, n, &n2, i)+'A';
}
}

return n2;
}
```

```
/*
char 'ch' is appended at the end of 's'.
*/
void chr_append(char *s, char ch)
{
int n=strlen(s);
*(s+n) = ch;
*(s+n+1) = '\0';
}
void sort(char stnt[][STATES+1], int n)
{
int i, j;
char temp[STATES+1];
for (i = 0; i < n-1; i++)
for (j = i+1; j < n; j++)
if (stnt[i][0] > stnt[j][0]) {
strcpy(temp, stnt[i]);
strcpy(stnt[i], stnt[j]);
strcpy(stnt[j], temp);
}
}
/*
Divide first equivalent class into subclasses.
stnt[i1] : equiv. class to be segmented
stnt[i2] : equiv. vector for next state of stnt[i1]
Algorithm:
- stnt[i1] is splitted into 2 or more classes 's1/s2/...'
- old equiv. classes are NOT changed, except stnt[i1]
- stnt[i1]=s1, stnt[n]=s2, stnt[n+1]=s3, ...
Return value: number of NEW equiv. classses in 'stnt'.
*/
int split_equiv_class(char stnt[][STATES+1],
int i1, /* index of 'i1'-th equiv. class */
int i2, /* index of equiv. vector for 'i1'-th class */
int n, /* number of entries in 'stnt' */
int n_dfa) /* number of source DFA entries */
{
char *old=stnt[i1], *vec=stnt[i2];
int i, n2, flag=0;
char newstates[STATES][STATES+1]; /* max. 'n' subclasses */
for (i=0; i < STATES; i++) newstates[i][0] = '\0';
for (i=0; vec[i]; i++)
chr_append(newstates[vec[i]-'0'], old[i]);
for (i=0, n2=n; i < n_dfa; i++) {
if (newstates[i][0]) {
if (!flag) { /* stnt[i1] = s1 */
strcpy(stnt[i1], newstates[i]);
flag = 1; /* overwrite parent class */
```

```
} else /* newstate is appended in 'stnt' */
strcpy(stnt[n2++], newstates[i]);
}
}
sort(stnt, n2); /* sort equiv. classes */
return n2; /* number of NEW states(equiv. classes) */
}
/*
Equiv. classes are segmented and get NEW equiv. classes.
*/
int set_new_equiv_class(char stnt[][STATES+1], int n,
int newdfa[][SYMBOLS], int n_sym, int n_dfa)
{
int i, j, k;
for (i = 0; i < n; i++) {
for (j = 0; j < n_sym; j++) {
k = newdfa[i][j]-'A'; /* index of equiv. vector */
if (k >= n) /* equiv. class 'i' should be segmented */
return split_equiv_class(stnt, i, k, n, n_dfa);
}
}
return n;
}
void print_equiv_classes(char stnt[][STATES+1], int n)
{
int i;
printf("\nEQUIV. CLASS CANDIDATE ==>");
for (i = 0; i < n; i++)
printf(" %d:[%s]", i, stnt[i]);
printf("\n");

}
/*
State-minimization of DFA: 'dfa' --> 'newdfa'
Return value: number of DFA states.
*/
int optimize_DFA(
int dfa[][SYMBOLS], /* DFA state-transition table */
int n_dfa, /* number of DFA states */
int n_sym, /* number of input symbols */
char *finals, /* final states of DFA */
char stnt[][STATES+1], /* state name table */
int newdfa[][SYMBOLS]) /* reduced DFA table */
{
char nextstate[STATES+1];
int n; /* number of new DFA states */
int n2; /* 'n' + <num. of state-dividing info> */
n = init_equiv_class(stnt, n_dfa, finals);
```

```
while (1) {
print_equiv_classes(stnt, n);
n2 = get_optimized_DFA(stnt, n, dfa, n_sym, newdfa);
if (n != n2)
n = set_new_equiv_class(stnt, n, newdfa, n_sym, n_dfa);
else break; /* equiv. class segmentation ended!!! */
}
return n; /* number of DFA states */
}
/*
Check if 't' is a subset of 's'.
*/
int is_subset(char *s, char *t)
{
int i;
for (i = 0; *t; i++)
if (!strchr(s, *t++)) return 0;
return 1;
}
/*
New finals states of reduced DFA.
*/
void get_NEW_finals(
char *newfinals, /* new DFA finals */
char *oldfinals, /* source DFA finals */
char stnt[][STATES+1], /* state name table */
int n) /* number of states in 'stnt' */
{
int i;
for (i = 0; i < n; i++)
if (is_subset(oldfinals, stnt[i])) *newfinals++ = i+'A';
*newfinals++ = '\0';
}
void main()
{
load_DFA_table();
print_dfa_table(DFAtab, N_DFA_states, N_symbols, DFA_finals);
N_optDFA_states = optimize_DFA(DFAtab, N_DFA_states,
N_symbols, DFA_finals, StateName, OptDFA);
get_NEW_finals(NEW_finals, DFA_finals, StateName, N_optDFA_states);
print_dfa_table(OptDFA, N_optDFA_states, N_symbols, NEW_finals);
}
```

8.WAP to find epsilon closure of all states of any given NFA with epsilon transition

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

```
struct node
{
int st;
struct node *link;
};
void findclosure(int,int);
void insert_trantbl(int ,char, int);
int findalpha(char);
void print_e_closure(int);
static int set[20],nostate,noalpha,s,notransition,c,r,buffer[20];
char alphabet[20];
static int e_closure[20][20]={0};
struct node * transition[20][20]={NULL};
void main()
{
int i,j,k,m,t,n;
struct node *temp;
printf("Enter the number of alphabets?\n");

scanf("%d",&noalpha);
getchar();
printf("NOTE:- [ use letter e as epsilon]\n");
printf("NOTE:- [e must be last character ,if it is present]\n");
printf("\nEnter alphabets?\n");
for(i=0;i<noalpha;i++)
{
alphabet[i]=getchar();
getchar();
}
printf("\nEnter the number of states?\n");
scanf("%d",&nostate);
printf("\nEnter no of transition?\n");
scanf("%d",&notransition);
printf("NOTE:- [Transition is in the form–> qno alphabet qno]\n",notransition);
printf("NOTE:- [States number must be greater than zero]\n");
printf("\nEnter transition?\n");
for(i=0;i<notransition;i++)
{
scanf("%d %c%d",&r,&c,&s);
insert_trantbl(r,c,s);
}
printf("\n");
printf("e-closure of states.....\n");
printf("———————————\n");
for(i=1;i<=nostate;i++)
{
c=0;
for(j=0;j<20;j++)
```

```
{
buffer[j]=0;
e_closure[i][j]=0;
}
findclosure(i,i);
printf("\ne-closure(q%d): ",i);
print_e_closure(i);
}
}
void findclosure(int x,int sta)
{
struct node *temp;
int i;
if(buffer[x])
return;
e_closure[sta][c++]=x;
buffer[x]=1;
if(alphabet[noalpha-1]=='e' && transition[x][noalpha-1]!=NULL)
{
temp=transition[x][noalpha-1];
while(temp!=NULL)
{
findclosure(temp->st,sta);
temp=temp->link;
}
}
}
void insert_trantbl(int r,char c,int s)
{
int j;
struct node *temp;
j=findalpha(c);
if(j==999)
{
printf("error\n");
exit(0);
}
temp=(struct node *)malloc(sizeof(struct node));
temp->st=s;
temp->link=transition[r][j];
transition[r][j]=temp;
}
int findalpha(char c)
{
int i;
for(i=0;i<noalpha;i++)
if(alphabet[i]==c)
return i;
```

```
return(999);
}
void print_e_closure(int i)
{
int j;
printf("{");
for(j=0;e_closure[i][j]!=0;j++)
printf("q%d,",e_closure[i][j]);
printf("}");
}
```

9.WAP to implement lexical analyzer using c

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

#include<ctype.h>
int kwd(char buffer[]);
int main(){
char ch, buffer[15], buf[15], operators[] = "+-*/%=,;()";
FILE *fp;
int i,j=0;
int ido=0;
char ids[26] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};
fp = fopen("input.txt","r");
if(fp == NULL){
printf("error while opening the file\n");
exit(0);
}
while((ch = fgetc(fp)) != EOF){
for(i=0;i<10;i++)
{
if(ch == operators[i] && kwd(buffer) == 0)
printf("id ");
}
for(i = 0; i < 10; ++i){
if(ch == operators[i])
printf("operator");
if(operators[i] == '+')
printf("op-plus ");
else if(operators[i] == '-')
printf("op-sub ");
else if(operators[i] == '*')
printf("op-mul ");
else if(operators[i] == '/')
printf("op-div ");
else if(operators[i] == '%')
printf("op-mod ");
```

```
else if(operators[i] == '=')
printf("op-equ ");
else if(operators[i] == ';')
printf(";");
else if(operators[i] == ',')
printf(",");
else if(operators[i] == '(')
printf(".");

}
if(isalnum(ch))
{
buffer[j++] = ch;
}
else if((ch == ' ' || ch == '\n') && (j != 0))
{
buffer[j] = '\0'
j = 0;
if(kwd(buffer) == 1)
printf("kwd ");
}
}
fclose(fp);
return 0;
}
int kwd(char buffer[]){
char keywords[32][10] = {"auto","break","case","char","const","continue","default",
"do","double","else","enum","extern","float","for","goto","if","int","long","register","return","short
","signed","sizeof","static","struct","switch","typedef","union","unsign
"void","volatile","while"};
int i, flag = 0;
for(i = 0; i < 32; ++i){
if(strcmp(keywords[i], buffer) == 0){
flag = 1;
break;
}
}
return flag;
}
/*int id( char buf[]){
char ids[26] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};
int i, flag = 0;
for(i = 0; i < 62; ++i){
if(strcmp(ids[i], buf) == 0){
flag = 1;
break;
}
}
```

```
return flag;
}
*/
```

10.WAP to convert NFA with epsilon transition to NFA without epsilon transition

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int st;
struct node *link;
};
void findclosure(int,int);
void insert_trantbl(int ,char, int);
int findalpha(char);
void findfinalstate(void);
void unionclosure(int);
void print_e_closure(int);
static int set[20],nostate,noalpha,s,notransition,nofinal,start,finalstate[20],c,r,buffer[20];
char alphabet[20];
static int e_closure[20][20]={0};
struct node * transition[20][20]={NULL};
void main()
{
int i,j,k,m,t,n;
struct node *temp;
printf("enter the number of alphabets?\n");
scanf("%d",&noalpha);
getchar();
printf("NOTE:- [ use letter e as epsilon]\n");
printf("NOTE:- [e must be last character ,if it is present]\n");
printf("\nEnter alphabets?\n");
for(i=0;i<noalpha;i++)
{
alphabet[i]=getchar();
getchar();
}
printf("Enter the number of states?\n");
scanf("%d",&nostate);
printf("Enter the start state?\n");
scanf("%d",&start);
printf("Enter the number of final states?\n");
scanf("%d",&nofinal);
printf("Enter the final states?\n");

for(i=0;i<nofinal;i++)
scanf("%d",&finalstate[i]);
printf("Enter no of transition?\n");
```

```
scanf("%d",&notransition);
printf("NOTE:- [Transition is in the form--> qno alphabet qno]\n",notransition);
printf("NOTE:- [States number must be greater than zero]\n");
printf("\nEnter transition?\n");
for(i=0;i<notransition;i++)
{
scanf("%d %c%d",&r,&c,&s);
insert_trantbl(r,c,s);
}
printf("\n");
for(i=1;i<=nostate;i++)
{
c=0;
for(j=0;j<20;j++)
{
buffer[j]=0;
e_closure[i][j]=0;
}
findclosure(i,i);
}
printf("Equivalent NFA without epsilon\n");
printf("-----------------------------------\n");
printf("start state:");
print_e_closure(start);
printf("\nAlphabets:");
for(i=0;i<noalpha;i++)
printf("%c ",alphabet[i]);
printf("\nStates :" );
for(i=1;i<=nostate;i++)
print_e_closure(i);
printf("\nTransitions are...:\n");
for(i=1;i<=nostate;i++)
{
for(j=0;j<noalpha-1;j++)
{
for(m=1;m<=nostate;m++)
set[m]=0;
for(k=0;e_closure[i][k]!=0;k++)
{
t=e_closure[i][k];
temp=transition[t][j];
while(temp!=NULL)
{
unionclosure(temp->st);
temp=temp->link;
}
}
printf("\n");
```

```
print_e_closure(i);
printf("%c\t",alphabet[j] );
printf("{");
for(n=1;n<=nostate;n++)
{
if(set[n]!=0)
printf("q%d,",n);
}
printf("}");
}
}
printf("\nFinal states:");
findfinalstate();
}
void findclosure(int x,int sta)
{
struct node *temp;
int i;
if(buffer[x])
return;
e_closure[sta][c++]=x;
buffer[x]=1;
if(alphabet[noalpha-1]=='e' && transition[x][noalpha-1]!=NULL)
{
temp=transition[x][noalpha-1];
while(temp!=NULL)
{
findclosure(temp->st,sta);
temp=temp->link;
}
}
}
void insert_trantbl(int r,char c,int s)
{
int j;
struct node *temp;
j=findalpha(c);
if(j==999)
{
printf("error\n");
exit(0);
}
temp=(struct node *) malloc(sizeof(struct node));
temp->st=s;

temp->link=transition[r][j];
transition[r][j]=temp;
}
```

```
int findalpha(char c)
{
int i;
for(i=0;i<noalpha;i++)
if(alphabet[i]==c)
return i;
return(999);
}
void unionclosure(int i)
{
int j=0,k;
while(e_closure[i][j]!=0)
{
k=e_closure[i][j];
set[k]=1;
j++;
}
}
void findfinalstate()
{
int i,j,k,t;
for(i=0;i<nofinal;i++)
{
for(j=1;j<=nostate;j++)
{
for(k=0;e_closure[j][k]!=0;k++)
{
if(e_closure[j][k]==finalstate[i])
{
print_e_closure(j);
}
}
}
}
}
void print_e_closure(int i)
{
int j=0;
printf("{");
if(e_closure[i][j]!=0)
printf("q%d,",e_closure[i][0]);
printf("}\t");
}
```

11.Implement a Lexical Analyzer for a given program using lex tool

```
%{
int COMMENT=0;
%}
identifier [a-zA-Z][a-zA-Z0-9]* %%
#.* {printf("\n%s is a preprocessor directive",yytext);}
int |
float |
char |

double |
while |
for |
struct |
typedef |
do |
if |
break |
continue |
void |
switch |
return |
else |
goto {printf(" kwd");}
"/*" {COMMENT=1;}{printf("comment");}
\+ {if(!COMMENT)printf(" op-plus");}
\- {if(!COMMENT)printf(" op-sub");}
\* {if(!COMMENT)printf(" op-mul");}
\/ {if(!COMMENT)printf(" op-div");}
{identifier}\( {if(!COMMENT)printf("fun");}
\{ {if(!COMMENT)printf("block begins");}
\} {if(!COMMENT)printf("block ends");}
{identifier}(\[[0-9]*\])? {if(!COMMENT) printf(" id");}
\".*\" {if(!COMMENT)printf("str");}
[0-9]+ {if(!COMMENT) printf("num");}
\)(:)? {if(!COMMENT)printf("\n\t");ECHO;printf("\n");}
\( ECHO;
= {if(!COMMENT)printf(" op-equ");}
\<= |
\>= |
\< |
== |
\> {if(!COMMENT) printf("rel-op");}
%%
int main(int argc, char **argv)
{
FILE *file;
```

```
file=fopen("input.c","r");
if(!file)
{
printf("could not open the file");
exit(0);
}
yyin=file;
yylex();
printf("\n");
return(0);
}
int yywrap()
{
return(1);
}
```

12.Generate a YACC specification to recognize a valid string of the form a^nb^n,where n>=1

Yacc1.lex

```
%{
#include "y.tab.h"
%}
%%
[aA] {return A;}
[bB] {return B;}
[\t]  ;
[\n] {return NL;}
%%
int yywrap()
{
return 1;
}
```

Yacc1.yacc

```
%{
#include<stdio.h>
%}
%token A B NL
%%
P: S NL {printf("The String is valid");
        return 0;
        }
S: A S B | A B ;
%%
int main()
{
```

```
printf("Enter the string");
yyparse();
}
int yyerror(char *S)
{
printf("The string is invalid");
}
```

13.Generate a YACC specification to recognize an arithmetic expression that uses operators +,-,*,/ and parenthesis is valid or not.Also evaluate the expression if it is valid.

Yacc2.yacc

```
%{
   /* Definition section */
  #include<stdio.h>
  int flag=0;
%}

%token NUMBER

%left '+' '-'

%left '*' '/'

%left '(' ')'

/* Rule Section */
%%

S: E{

        printf("\nResult=%d\n", $$);

        return 0;

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

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

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

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

 |'('E')' {$$=$2;}
```

```
 | NUMBER {$$=$1;}
 ;
 %%
 //driver code
void main()
{
   printf("Enter Any Arithmetic Expression which can have operations Addition, Subtraction,
Multiplication, Division and Parenthesis:\n");

   yyparse();
   if(flag==0)
   printf("Entered arithmetic expression is Valid\n\n");
}

void yyerror()
{
   printf("Entered arithmetic expression is Invalid\n\n");
   flag=1;
}
```

Yacc2.lex

```
%{
   /* Definition section */
  #include<stdio.h>
  #include "y.tab.h"
  extern int yylval;
%}

/* Rule Section */
%%
[0-9]+ {
          yylval=atoi(yytext);
          return NUMBER;

       }
[\t] ;

[\n] return 0;

. return yytext[0];
 %%
 int yywrap()
{
return 1;
}
```

14.Generate a YACC specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.

Yacc3.lex

```
%{

   #include "y.tab.h"

%}

%%

[a-zA-Z_][a-zA-Z_0-9]* return letter;

[0-9]                   return digit;

.                   return yytext[0];

\n                   return 0;
%%
int yywrap()
{
return 1;
}
```

Yacc3.yacc

```
%{

   #include<stdio.h>

   int valid=1;

%}

%token digit letter

%%

start : letter s

s :     letter s

      | digit s

      |

      ;

%%
```

```
int yyerror()
{
    printf("\nIts not an identifier!\n");
    valid=0;
    return 0;
}
int main()
{
    printf("\nEnter the identifier ");
    yyparse();
    if(valid)
    {
        printf("\nIt is a identifier!\n");
}
}
```