Program: Lexical Analyzer

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
int main()
{
FILE *input, *output;
int l=1;
int t=0;
int j=0;
int i,flag;
char ch,str[20];
input = fopen("input.txt","r");
output = fopen("output.txt","w");
char keyword[30][30] =
{"int","main","if","else","do","while"};
fprintf(output,"Line no. \t Token no. \t Token
\t Lexeme\n\n");
while(!feof(input))
{
i=0;
flag=0;
ch=fgetc(input);
if( ch=='+' || ch== '-' || ch=='*' || ch=='/' )
{
fprintf(output,"%7d\t\t %7d\t\t Operator\t
%7c\n",l,t,ch);
t++;
}
else if( ch==';' || ch=='{' || ch=='}' || ch=='(' ||
ch==')' || ch=='?' ||
ch=='@' || ch=='!' ||
ch=='%')
{
fprintf(output,"%7d\t\t %7d\t\t Special
symbol\t %7c\n",l,t,ch);
t++;
}

else if(isdigit(ch))
{
fprintf(output,"%7d\t\t %7d\t\t Digit\t\t
%7c\n",l,t,ch);
t++;
}
else if(isalpha(ch))
{
str[i]=ch;
i++;
ch=fgetc(input);
while(isalnum(ch) && ch!=' ')
{
str[i]=ch;
i++;
ch=fgetc(input);
}
str[i]='\0';
for(j=0;j<=30;j++)
{
if(strcmp(str,keyword[j])==0)
{
flag=1;
break;
}
}
if(flag==1)
{
fprintf(output,"%7d\t\t %7d\t\t Keyword\t
%7s\n",l,t,str);
t++;
}
else
{
fprintf(output,"%7d\t\t %7d\t\t Identifier\t
%7s\n",l,t,str);
t++;
}
}
else if(ch=='\n')
{l++;
}
}
fclose(input);
fclose(output);
return 0;
}
```

Count no.of characters

```
%{
int c=0;
%}
%%
[A-Za-z] c++;
. ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count is %d\n",c);
}
int yywrap()
{
return 1;
}
```

B.c

This is a lex program

Output:

Count the digits

```
%{
int c=0;
%}
digit [0-9]
%%
{digit} c++;
. ;
%%
int main()
{
yyin=fopen("a.c","r");
yylex();
printf("count is %d\n",c);
}
int yywrap()
{
return 1;
}
```

A.c

```
2 3 4 1 0
```

Output:

Count the positive and negative numbers

```
%{
int c=0,d=0;
%}
pdigit [0-9]
ndigit [-][0-9]
%%
{pdigit} c++;
{ndigit} d++;
. ;
%%
int main()
{
yyin=fopen("d.txt","r");
yylex();
printf("Positive count is %d\nNegative count is %d\n",c,d);
}
int yywrap()
{
return 1;
}
```

D.txt

```
2 -3 4 -1 0
```

Output:

Calculator

Calc.y

```
 %{

    #include<stdio.h>

    int flag=0;



%}

%token NUMBER



%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression: 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'%'E {$$=$1%$3;}

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

 | NUMBER {$$=$1;}

;

%%



void main()

{

   printf("\nEnter Any Arithmetic
Expression:\n");

   yyparse();

  if(flag==0)

   printf("\n\n");



}

void yyerror()

{

   printf("\nInvalid\n\n");

   flag=1;

}
```

Calc.l

```
%{
/* 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;
}
```

Output:

First and Follow

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

// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);

// Function to calculate First
void findfirst(char, int, int);
int count, n = 0;

// Stores the final result
// of the First Sets
char calc_first[10][100];

// Stores the final result
// of the Follow Sets
char calc_follow[10][100];
int m = 0;

// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;

int main(int argc, char **argv)
{
int jm = 0;
int km = 0;
int i, choice;
char c, ch;
count = 8;

// The Input grammar
strcpy(production[0], "E=TR");
strcpy(production[1], "R=+TR");
strcpy(production[2], "R=#");
strcpy(production[3], "T=FY");
strcpy(production[4], "Y=*FY");
strcpy(production[5], "Y=#");
strcpy(production[6], "F=(E)");
strcpy(production[7], "F=i");

int kay;
char done[count];
int ptr = -1;

// Initializing the calc_first array
for(k = 0; k < count; k++) {
for(kay = 0; kay < 100; kay++) {
calc_first[k][kay] = '!';
}
}
int point1 = 0, point2, xxx;

for(k = 0; k < count; k++)
{
c = production[k][0];
point2 = 0;
xxx = 0;

// Checking if First of c has
// already been calculated
for(kay = 0; kay <= ptr; kay++)
if(c == done[kay])
xxx = 1;

if (xxx == 1)
continue;
// Function call
findfirst(c, 0, 0);
ptr += 1;

// Adding c to the calculated list
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;

// Printing the First Sets of the grammar
```

```
for(i = 0 + jm; i < n; i++) {
int lark = 0, chk = 0;

for(lark = 0; lark < point2; lark++) {

if (first[i] == calc_first[point1][lark])
{
chk = 1;
break;
}
}
if(chk == 0)
{
printf("%c, ", first[i]);
calc_first[point1][point2++] = first[i];
}
}
printf("}\n");
jm = n;
point1++;
}
printf("\n");
printf("----------------------------------------------\n
\n");
char donee[count];
ptr = -1;

// Initializing the calc_follow array
for(k = 0; k < count; k++) {
for(kay = 0; kay < 100; kay++) {
calc_follow[k][kay] = '!';
}
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{
ck = production[e][0];
point2 = 0;
xxx = 0;

// Checking if Follow of ck
// has alredy been calculated
for(kay = 0; kay <= ptr; kay++)
if(ck == donee[kay])
xxx = 1;

if (xxx == 1)
continue;
land += 1;

// Function call
follow(ck);
ptr += 1;

// Adding ck to the calculated list
donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;

// Printing the Follow Sets of the grammar
for(i = 0 + km; i < m; i++) {
int lark = 0, chk = 0;
for(lark = 0; lark < point2; lark++)
{
if (f[i] == calc_follow[point1][lark])
{
chk = 1;
break;
}
}
if(chk == 0)
{
printf("%c, ", f[i]);
calc_follow[point1][point2++] = f[i];
}
}
printf(" }\n\n");
km = m;
point1++;
}
}
void follow(char c)
{
int i, j;

// Adding "$" to the follow
// set of the start symbol
```

```
if(production[0][0] == c) {
f[m++] = '$';
}
for(i = 0; i < 10; i++)
{
for(j = 2;j < 10; j++)
{
if(production[i][j] == c)
{
if(production[i][j+1] != '\0')
{
// Calculate the first of the next
// Non-Terminal in the production
followfirst(production[i][j+1], i, (j+2));
}

if(production[i][j+1]=='\0' &&
c!=production[i][0])
{
// Calculate the follow of the Non-Terminal
// in the L.H.S. of the production
follow(production[i][0]);
}
}
}
}

}

void findfirst(char c, int q1, int q2)
{
int j;

// The case where we
// encounter a Terminal
if(!(isupper(c))) {
first[n++] = c;
}
for(j = 0; j < count; j++)
{
if(production[j][0] == c)
{
if(production[j][2] == '#')
{
if(production[q1][q2] == '\0')
first[n++] = '#';
else if(production[q1][q2] != '\0'
&& (q1 != 0 || q2 != 0))

{
// Recursion to calculate First of New
// Non-Terminal we encounter after epsilon
findfirst(production[q1][q2], q1, (q2+1));
}
else
first[n++] = '#';

}
else if(!isupper(production[j][2]))
{
first[n++] = production[j][2];
}
else
{
// Recursion to calculate First of
// New Non-Terminal we encounter
// at the beginning
findfirst(production[j][2], j, 3);
}
}
}
}

void followfirst(char c, int c1, int c2)
{
int k;

// The case where we encounter
// a Terminal
if(!(isupper(c)))
f[m++] = c;
else
{
int i = 0, j = 1;
for(i = 0; i < count; i++)
{
if(calc_first[i][0] == c)
break;
```

```
}

//Including the First set of the

// Non-Terminal in the Follow of
// the original query
while(calc_first[i][j] != '!')
{
if(calc_first[i][j] != '#')
{
f[m++] = calc_first[i][j];
}
else
{
if(production[c1][c2] == '\0')
{
// Case where we reach the
// end of a production
follow(production[c1][0]);
}
else
{
// Recursion to the next symbol
// in case we encounter a "#"
followfirst(production[c1][c2], c1, c2+1);
}
}
j++;
}
}
}
```

Output:

Check the validity of an arithmetic expression

Arithmetic.y

```
%{
#include<stdio.h>
%}
%token ID NUMBER
%left '+' '-'
%left '*' '/'
%%
stmt:expr {printf("valid Expression\n");}
;
expr: expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '(' expr ')'
| NUMBER
| ID
;
%%
int main ()
{
do
{printf("Enter the expression\n");
yyparse();
}while(1);
return 1;
}
int yyerror (char *msg)
{
printf("Invalid Expression\n");
}
yywrap()
{
return(1);
}
```

Arithmetic.l

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[a-zA-Z] {return ID;}
[0-9] {return NUMBER;}
\n { printf ("reached end of line\n");
return 0;
}
. { printf ("found other data \"%s\"\n", yytext);
return yytext[0];
/* so yacc can see things like '+', '-', and '='
*/
}
```

Output:

Check the validity of a variable followed by letters or digits

Valid.y

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token DIGIT LETTER
%start S
%%
S : variable { printf("Valid Variable\n"); }
;
variable : LETTER alphanumeric
;
alphanumeric :LETTER alphanumeric
|DIGIT alphanumeric
|LETTER
|DIGIT
;
%%
int main ()
{
do
{printf("Enter the expression\n");
yyparse();
}while(1);
return 1;
}
int yyerror (char *msg)
{
printf("Invalid Expression\n");
}
yywrap()
{
return(1);
}
```

Valid.l

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[a-zA-Z] {return LETTER;}
[0-9] {return DIGIT;}
\n { printf ("reached end of line\n");
return 0;
}
. { printf ("found other data \"%s\"\n", yytext);
return yytext[0];
/* so yacc can see things like '+', '-', and '='
*/
}
```

Output:

Operator Precedence

```
#include<stdio.h>
#include<string.h>
void main(){
char stack[20],ip[20],opt[10][10][1],ter[10];
int i,j,k,n,top=0,col,row;
for(i=0;i<10;i++)
{
stack[i]=NULL;ip[i]=NULL;
for(j=0;j<10;j++)
{
opt[i][j][1]=NULL;
}
}
printf("Enter the no.of terminals :\n");
scanf("%d",&n);
printf("\nEnter the terminals :\n");
scanf("%s",&ter);
printf("\nEnter the table values :\n");
for(i=0;i<n;i++)
{
for(j=0;j<n;j++)
{
printf("Enter the value for %c
%c:",ter[i],ter[j]);
scanf("%s",opt[i][j]);
}
}
printf("\n**** OPERATOR PRECEDENCE
TABLE ****\n");
for(i=0;i<n;i++)
{
printf("\t%c",ter[i]);
}
printf("\n");
for(i=0;i<n;i++){printf("\n%c",ter[i]);
for(j=0;j<n;j++){printf("\t%c",opt[i][j][0]);}}
stack[top]='$';
printf("\nEnter the input string:");
scanf("%s",ip);i=0;
printf("\nSTACK\t\t\tINPUT
STRING\t\t\tACTION\n");
printf("\n%s\t\t\t%s\t\t\t",stack,ip);
while(i<=strlen(ip))
{
for(k=0;k<n;k++)
{
if(stack[top]==ter[k])
col=k;
if(ip[i]==ter[k])
row=k;
}
if((stack[top]=='$')&&(ip[i]=='$')){
printf("String is accepted\n");
break;}
else if((opt[col][row][0]=='<')
||(opt[col][row][0]=='='))
{ stack[++top]=opt[col][row][0];
stack[++top]=ip[i];
printf("Shift %c",ip[i]);
i++;
}
else{
if(opt[col][row][0]=='>')
{
while(stack[top]!='<'){--top;}
top=top-1;
printf("Reduce");
}
else
{
printf("\nString is not accepted");
break;
}
}
printf("\n");
for(k=0;k<=top;k++)
{
printf("%c",stack[k]);
}
printf("\t\t\t");
for(k=i;k<strlen(ip);k++){
printf("%c",ip[k]);
}
printf("\t\t\t");
}
```

Loop Unrolling

```
#include<stdio.h>
void main()
{
unsigned int n;
int x;
char ch;
printf("\nEnter N\n");
scanf("%u",&n);
printf("\n1. Loop Roll\n2. Loop UnRoll\n");
printf("\nEnter ur choice\n");
scanf(" %c",&ch);
switch(ch)
{
case '1':
x=countbit1(n);
printf("\nLoop Roll: Count of 1's : %d" ,x);
break;
case '2':
x=countbit2(n);
printf("\nLoop UnRoll: Count of 1's : %d" ,x);
break;
default:
printf("\n Wrong Choice\n");
}
}
int countbit1(unsigned int n)
{
int bits = 0,i=0;
while (n != 0)
{
if (n & 1) bits++;
n >>= 1;
i++;
}
printf("\n no of iterations %d",i);
return bits;
}
int countbit2(unsigned int n)
{
int bits = 0,i=0;
while (n != 0)
{
if (n & 1) bits++;
if (n & 2) bits++;
if (n & 4) bits++;
if (n & 8) bits++;
n >>= 4;
i++;
}
printf("\n no of iterations %d",i);
return bits;
}
```

Output:

Constant Propagation

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
void input();
void output();
void change(int p,char *res);
void constant();
struct expr
{
char op[2],op1[5],op2[5],res[5];
int flag;
}arr[10];
int n;
void main()
{
input();
constant();
output();
}
void input()
{
int i;
printf("\n\nEnter the maximum number of
expressions : ");
scanf("%d",&n);
printf("\nEnter the input : \n");
for(i=0;i<n;i++)
{
scanf("%s",arr[i].op);
scanf("%s",arr[i].op1);
scanf("%s",arr[i].op2);
scanf("%s",arr[i].res);
arr[i].flag=0;
}
}
void constant()
{
int i;
int op1,op2,res;
char op,res1[5];
for(i=0;i<n;i++)
{
if(isdigit(arr[i].op1[0]) && isdigit(arr[i].op2[0])
|| strcmp(arr[i].op,"=")==0) /*if both digits,
store them in variables*/
{
op1=atoi(arr[i].op1);
op2=atoi(arr[i].op2);
op=arr[i].op[0];
switch(op)
{
case '+':
res=op1+op2;
break;
case '-':
res=op1-op2;
break;
case '*':
res=op1*op2;
break;
case '/':
res=op1/op2;
break;
case '=':
res=op1;
break;
}
sprintf(res1,"%d",res);
arr[i].flag=1; /*eliminate expr and replace
any operand below that uses result of this
expr */
change(i,res1);
}
}
}
void output()
{
int i=0;
printf("\nOptimized code is : ");
for(i=0;i<n;i++)
{
if(!arr[i].flag)
{
```

```
printf("\n%s %s %s
%s",arr[i].op,arr[i].op1,arr[i].op2,arr[i].res);
}
}
}
void change(int p,char *res)
{
int i;
for(i=p+1;i<n;i++)
{
if(strcmp(arr[p].res,arr[i].op1)==0)
strcpy(arr[i].op1,res);
else if(strcmp(arr[p].res,arr[i].op2)==0)
strcpy(arr[i].op2,res);
}
}
```

Output:

recursive descent parser

```
#include"stdio.h"
#include"string.h"
#include"stdlib.h"
#include"ctype.h"

char ip_sym[15],ip_ptr=0,op[50],tmp[50];
void e_prime();
void e();
void t_prime();
void t();
void f();
void advance();
int n=0;
void e()
{
 strcpy(op,"TE'");
 printf("E=%-25s",op);
 printf("E->TE'\n");
 t();
 e_prime();
}

void e_prime()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
          if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='E';n++);
if(ip_sym[ip_ptr]=='+')
 {
          i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i<=l);
  op[n++]='+';
  op[n++]='T';
  op[n++]='E';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("E'->+TE'\n");
  advance();
  t();
  e_prime();
 }
 else
 {
          op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("E'->e");
 }
}
void t()
{
 int i,n=0,l;
 for(i=0;i<=strlen(op);i++)
  if(op[i]!='e')
   tmp[n++]=op[i];
 strcpy(op,tmp);
 l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);

 i=n+1;
 do
 {
  op[i+2]=op[i];
  i++;
 }while(i < l);
 op[n++]='F';
 op[n++]='T';
 op[n++]=39;
 printf("E=%-25s",op);
 printf("T->FT'\n");
 f();
 t_prime();
}

void t_prime()
{
int i,n=0,l;
```

```
for(i=0;i<=strlen(op);i++)
        if(op[i]!='e')
 tmp[n++]=op[i];
strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='T';n++);
if(ip_sym[ip_ptr]=='*')
 {
        i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i < l);
  op[n++]='*';
  op[n++]='F';
  op[n++]='T';
  op[n++]=39;
  printf("E=%-25s",op);
  printf("T'->*FT'\n");
  advance();
  f();
  t_prime();
 }
 else
 {
   op[n]='e';
  for(i=n+1;i<=strlen(op);i++)
 op[i]=op[i+1];
 printf("E=%-25s",op);
 printf("T'->e\n");
 }
}

void f()
{
int i,n=0,l;
for(i=0;i<=strlen(op);i++)
        if(op[i]!='e')
 tmp[n++]=op[i];
 strcpy(op,tmp);
l=strlen(op);
for(n=0;n < l && op[n]!='F';n++);
 if((ip_sym[ip_ptr]=='i')||(ip_sym[ip_ptr]=='I'))
 {
 op[n]='i';
 printf("E=%-25s",op);
 printf("F->i\n");
 advance();
 }
 else
 {
  if(ip_sym[ip_ptr]=='(')
  {
   advance();
   e();
   if(ip_sym[ip_ptr]==')')
   {
        advance();
        i=n+2;
 do
 {
 op[i+2]=op[i];
 i++;
 }while(i<=l);
  op[n++]='(';
  op[n++]='E';
  op[n++]=')';
  printf("E=%-25s",op);
  printf("F->(E)\n");
   }
  }
  else
  {
   printf("\n\t syntax error");

   exit(1);
  }
 }
}

void advance()
{
 ip_ptr++;
}

void main()
{
```

```
 int i;

 printf("\nGrammar without left recursion");
 printf("\n\t\t E->TE' \n\t\t E'->+TE'|e \n\t\t
T->FT' ");
 printf("\n\t\t T'->*FT'|e \n\t\t F->(E)|i");
 printf("\n Enter the input expression:");
 scanf("%s",ip_sym);
 printf("Expressions");
 printf("\t Sequence of production rules\n");
  e();
  for(i=0;i < strlen(ip_sym);i++)
 {

if(ip_sym[i]!='+'&&ip_sym[i]!='*'&&ip_sym[i]!
='('&&
ip_sym[i]!=')'&&ip_sym[i]!='i'&&ip_sym[i]!='l')
  {
   printf("\nSyntax error");
   break;
  }
  for(i=0;i<=strlen(op);i++)
          if(op[i]!='e')
 tmp[n++]=op[i];
          strcpy(op,tmp);
          printf("\nE=%-25s",op);
 }

}
```

Shift Reduce Parser

```
#include<stdio.h>
//#include<conio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
void main()
{
//clrscr();
puts("GRAMMAR is E->E+E \n E->E*E \n
E->(E) \n E->id");
puts("enter input string ");
gets(a);
c=strlen(a);
strcpy(act,"SHIFT->");
puts("stack \t input \t action");
for(k=0,i=0; j<c; k++,i++,j++)
{
if(a[j]=='i' && a[j+1]=='d')
{
stk[i]=a[j];
stk[i+1]=a[j+1];
stk[i+2]='\0';
a[j]=' ';
a[j+1]=' ';
printf("\n$%s\t%s$\t%sid",stk,a,act);
check();
}
else
{
stk[i]=a[j];
stk[i+1]='\0';
a[j]=' ';
printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
check();
}
}
//getch();
}
void check()
{
strcpy(ac,"REDUCE TO E");
for(z=0; z<c; z++)
if(stk[z]=='i' && stk[z+1]=='d')
{
stk[z]='E';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
j++;
}
for(z=0; z<c; z++)
if(stk[z]=='E' && stk[z+1]=='+' &&
stk[z+2]=='E')
{
stk[z]='E';
stk[z+1]='\0';
stk[z+2]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}
for(z=0; z<c; z++)
if(stk[z]=='E' && stk[z+1]=='*' &&
stk[z+2]=='E')
{
stk[z]='E';
stk[z+1]='\0';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}
for(z=0; z<c; z++)
if(stk[z]=='(' && stk[z+1]=='E' &&
stk[z+2]==')')
{
stk[z]='E';
stk[z+1]='\0';
stk[z+1]='\0';
printf("\n$%s\t%s$\t%s",stk,a,ac);
i=i-2;
}
}
```

Find ε – closure of all states of any given NFA with ε transition.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN 100
char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int zz = 0;
// Structure to store DFA states and their
// status ( i.e new entry or already present)
struct DFA {
char *states;
int count;
} dfa;
int last_index = 0;
FILE *fp;
int symbols;
/* reset the hash map*/
void reset(int ar[], int size) {
int i;
// reset all the values of
// the mapping array to zero
for (i = 0; i < size; i++) {
ar[i] = 0;
}
}
// Check which States are present in the
e-closure
/* map the states of NFA to a hash set*/
void check(int ar[], char S[]) {
int i, j;
// To parse the individual states of NFA
int len = strlen(S);
for (i = 0; i < len; i++) {
// Set hash map for the position
// of the states which is found
j = ((int)(S[i]) - 65);
ar[j]++;
}
}
// To find new Closure States
void state(int ar[], int size, char S[]) {
int j, k = 0;
// Combine multiple states of NFA
// to create new states of DFA
for (j = 0; j < size; j++) {
if (ar[j] != 0)
S[k++] = (char)(65 + j);
}
// mark the end of the state
S[k] = '\0';
}
// To pick the next closure from closure set
int closure(int ar[], int size) {
int i;
// check new closure is present or not
for (i = 0; i < size; i++) {
if (ar[i] == 1)
return i;
}
return (100);
}
// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
int i;
for (i = 0; i < last_index; i++) {
if (dfa[i].count == 0)
return 1;
}
return -1;
}
/* To Display epsilon closure*/
void Display_closure(int states, int
closure_ar[],
char *closure_table[],
char *NFA_TABLE[][symbols + 1],
char *DFA_TABLE[][symbols]) {
int i;
for (i = 0; i < states; i++) {
reset(closure_ar, states);
closure_ar[i] = 2;
// to neglect blank entry
if (strcmp(&NFA_TABLE[i][symbols], "-") !=
0) {
```

```
// copy the NFA transition state to buffer
strcpy(buffer, &NFA_TABLE[i][symbols]);
check(closure_ar, buffer);
int z = closure(closure_ar, states);
// till closure get completely saturated
while (z != 100)
{
if (strcmp(&NFA_TABLE[z][symbols], "-") !=
0) {
strcpy(buffer, &NFA_TABLE[z][symbols]);
// call the check function
check(closure_ar, buffer);
}
closure_ar[z]++;
z = closure(closure_ar, states);
}
}
// print the e closure for every states of NFA
printf("\n e-Closure (%c) :\t", (char)(65 + i));
bzero((void *)buffer, MAX_LEN);
state(closure_ar, states, buffer);
strcpy(&closure_table[i], buffer);
printf("%s\n", &closure_table[i]);
}
}
/* To check New States in DFA */
int new_states(struct DFA *dfa, char S[]) {
int i;
// To check the current state is already
// being used as a DFA state or not in
// DFA transition table
for (i = 0; i < last_index; i++) {
if (strcmp(&dfa[i].states, S) == 0)
return 0;
}
// push the new
strcpy(&dfa[last_index++].states, S);
// set the count for new states entered
// to zero
dfa[last_index - 1].count = 0;
return 1;
}
// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int
st,
char *NFT[][symbols + 1], char TB[]) {
int len = strlen(S);
int i, j, k, g;
int arr[st];
int sz;
reset(arr, st);
char temp[MAX_LEN], temp2[MAX_LEN];
char *buff;
// Transition function from NFA to DFA
for (i = 0; i < len; i++) {
j = ((int)(S[i] - 65));
strcpy(temp, &NFT[j][M]);
if (strcmp(temp, "-") != 0) {
sz = strlen(temp);
g = 0;
while (g < sz) {
k = ((int)(temp[g] - 65));
strcpy(temp2, &clsr_t[k]);
check(arr, temp2);
g++;
}
}
}
bzero((void *)temp, MAX_LEN);
state(arr, st, temp);
if (temp[0] != '\0') {
strcpy(TB, temp);
} else
strcpy(TB, "-");
}
/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA
*dfa_states,
char *DFA_TABLE[][symbols]) {
int i, j;
printf("\n\n**************************************
******************\n\n");
printf("\t\t DFA TRANSITION STATE TABLE
\t\t \n\n");
printf("\n STATES OF DFA :\t\t");
for (i = 1; i < last_index; i++)
printf("%s, ", &dfa_states[i].states);
```

```
printf("\n");
printf("\n GIVEN SYMBOLS FOR DFA: \t");
for (i = 0; i < symbols; i++)
printf("%d, ", i);
printf("\n\n");
printf("STATES\t");
for (i = 0; i < symbols; i++)
printf("|%d\t", i);
printf("\n");
// display the DFA transition state table
printf("--------+----------------------\n");
for (i = 0; i < zz; i++) {
printf("%s\t", &dfa_states[i + 1].states);
for (j = 0; j < symbols; j++) {
printf("|%s \t", &DFA_TABLE[i][j]);
}
printf("\n");
}
}
// Driver Code
int main() {
int i, j, states;
char T_buf[MAX_LEN];
// creating an array dfa structures
struct DFA *dfa_states = malloc(MAX_LEN *
(sizeof(dfa)));
states = 6, symbols = 2;
printf("\n STATES OF NFA :\t\t");
for (i = 0; i < states; i++)
printf("%c, ", (char)(65 + i));
printf("\n");
printf("\n GIVEN SYMBOLS FOR NFA: \t");
for (i = 0; i < symbols; i++)
printf("%d, ", i);
printf("eps");
printf("\n\n");
char *NFA_TABLE[states][symbols + 1];
// Hard coded input for NFA table
char *DFA_TABLE[MAX_LEN][symbols];
strcpy(&NFA_TABLE[0][0], "FC");
strcpy(&NFA_TABLE[0][1], "-");
strcpy(&NFA_TABLE[0][2], "BF");
strcpy(&NFA_TABLE[1][0], "-");
strcpy(&NFA_TABLE[1][1], "C");
strcpy(&NFA_TABLE[1][2], "-");
strcpy(&NFA_TABLE[2][0], "-");
strcpy(&NFA_TABLE[2][1], "-");
strcpy(&NFA_TABLE[2][2], "D");
strcpy(&NFA_TABLE[3][0], "E");
strcpy(&NFA_TABLE[3][1], "A");
strcpy(&NFA_TABLE[3][2], "-");
strcpy(&NFA_TABLE[4][0], "A");
strcpy(&NFA_TABLE[4][1], "-");
strcpy(&NFA_TABLE[4][2], "BF");
strcpy(&NFA_TABLE[5][0], "-");
strcpy(&NFA_TABLE[5][1], "-");
strcpy(&NFA_TABLE[5][2], "-");
printf("\n NFA STATE TRANSITION TABLE
\n\n\n");
printf("STATES\t");
for (i = 0; i < symbols; i++)
printf("|%d\t", i);
printf("eps\n");
// Displaying the matrix of NFA transition
table
printf("--------+-----------------------------------\n"
);
for (i = 0; i < states; i++) {
printf("%c\t", (char)(65 + i));
for (j = 0; j <= symbols; j++) {
printf("|%s \t", &NFA_TABLE[i][j]);
}
printf("\n");
}
int closure_ar[states];
char *closure_table[states];
Display_closure(states, closure_ar,
closure_table, NFA_TABLE, DFA_TABLE);
strcpy(&dfa_states[last_index++].states,
"-");
dfa_states[last_index - 1].count = 1;
bzero((void *)buffer, MAX_LEN);
strcpy(buffer, &closure_table[0]);
strcpy(&dfa_states[last_index++].states,
buffer);
int Sm = 1, ind = 1;
int start_index = 1;
}
```

Convert NFA to DFA

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>+
#define MAX_LEN 100

char NFA_FILE[MAX_LEN];
char buffer[MAX_LEN];
int zz = 0;
// Structure to store DFA states and their
// status ( i.e new entry or already present)
struct DFA {
char *states;
int count;
} dfa;
int last_index = 0;
FILE *fp;
int symbols;

/* reset the hash map*/
void reset(int ar[], int size) {
int i;
// reset all the values of
// the mapping array to zero
for (i = 0; i < size; i++) {
ar[i] = 0;
}
}
// Check which States are present in the
e-closure
/* map the states of NFA to a hash set*/
void check(int ar[], char S[]) {
int i, j;
// To parse the individual states of NFA
int len = strlen(S);
for (i = 0; i < len; i++) {
// Set hash map for the position
// of the states which is found
j = ((int)(S[i]) - 65);
ar[j]++;
}
}
// To find new Closure States
void state(int ar[], int size, char S[]) {
int j, k = 0;
// Combine multiple states of NFA
// to create new states of DFA
for (j = 0; j < size; j++) {
if (ar[j] != 0)
S[k++] = (char)(65 + j);
}
// mark the end of the state
S[k] = '\0';
}
// To pick the next closure from closure set
int closure(int ar[], int size) {
int i;
// check new closure is present or not
for (i = 0; i < size; i++) {
if (ar[i] == 1)
return i;
}
return (100);
}
// Check new DFA states can be
// entered in DFA table or not
int indexing(struct DFA *dfa) {
int i;
for (i = 0; i < last_index; i++) {
if (dfa[i].count == 0)
return 1;
}
return -1;
}
/* To Display epsilon closure*/
void Display_closure(int states, int
closure_ar[],

char *closure_table[],
char *NFA_TABLE[][symbols + 1],
char *DFA_TABLE[][symbols]) {

int i;
```

```
for (i = 0; i < states; i++) {
reset(closure_ar, states);
closure_ar[i] = 2;
// to neglect blank entry
if (strcmp(&NFA_TABLE[i][symbols], "-") !=
0) {
// copy the NFA transition state to buffer
strcpy(buffer, &NFA_TABLE[i][symbols]);
check(closure_ar, buffer);
int z = closure(closure_ar, states);

// till closure get completely saturated
while (z != 100)
{
if (strcmp(&NFA_TABLE[z][symbols], "-") !=
0) {
strcpy(buffer, &NFA_TABLE[z][symbols]);
// call the check function
check(closure_ar, buffer);
}
closure_ar[z]++;
z = closure(closure_ar, states);
}
}
// print the e closure for every states of NFA
printf("\n e-Closure (%c) :\t", (char)(65 + i));
bzero((void *)buffer, MAX_LEN);
state(closure_ar, states, buffer);
strcpy(&closure_table[i], buffer);
printf("%s\n", &closure_table[i]);
}
}
/* To check New States in DFA */
int new_states(struct DFA *dfa, char S[]) {
int i;
// To check the current state is already
// being used as a DFA state or not in
// DFA transition table
for (i = 0; i < last_index; i++) {
if (strcmp(&dfa[i].states, S) == 0)
return 0;
}
// push the new
strcpy(&dfa[last_index++].states, S);
// set the count for new states entered
// to zero
dfa[last_index - 1].count = 0;
return 1;
}
// Transition function from NFA to DFA
// (generally union of closure operation )
void trans(char S[], int M, char *clsr_t[], int
st,

char *NFT[][symbols + 1], char TB[]) {

int len = strlen(S);
int i, j, k, g;
int arr[st];
int sz;
reset(arr, st);
char temp[MAX_LEN], temp2[MAX_LEN];
char *buff;
// Transition function from NFA to DFA
for (i = 0; i < len; i++) {
j = ((int)(S[i] - 65));
strcpy(temp, &NFT[j][M]);
if (strcmp(temp, "-") != 0) {
sz = strlen(temp);
g = 0;
while (g < sz) {
k = ((int)(temp[g] - 65));
strcpy(temp2, &clsr_t[k]);
check(arr, temp2);
g++;
}
}
}
bzero((void *)temp, MAX_LEN);
state(arr, st, temp);
if (temp[0] != '\0') {
strcpy(TB, temp);
} else
strcpy(TB, "-");
}
/* Display DFA transition state table*/
void Display_DFA(int last_index, struct DFA
*dfa_states,
```

```
char *DFA_TABLE[][symbols]) {

int i, j;
printf("\n\n**************************************
******************\n\n");
printf("\t\t DFA TRANSITION STATE TABLE
\t\t \n\n");
printf("\n STATES OF DFA :\t\t");
for (i = 1; i < last_index; i++)
printf("%s, ", &dfa_states[i].states);
printf("\n");
printf("\n GIVEN SYMBOLS FOR DFA: \t");
for (i = 0; i < symbols; i++)
printf("%d, ", i);
printf("\n\n");
printf("STATES\t");
for (i = 0; i < symbols; i++)
printf("|%d\t", i);
printf("\n");
// display the DFA transition state table
printf("--------+----------------------\n");
for (i = 0; i < zz; i++) {
printf("%s\t", &dfa_states[i + 1].states);
for (j = 0; j < symbols; j++) {
printf("|%s \t", &DFA_TABLE[i][j]);
}
printf("\n");
}
}
// Driver Code
int main() {
int i, j, states;
char T_buf[MAX_LEN];
// creating an array dfa structures
struct DFA *dfa_states = malloc(MAX_LEN *
(sizeof(dfa)));
states = 6, symbols = 2;
printf("\n STATES OF NFA :\t\t");
for (i = 0; i < states; i++)
printf("%c, ", (char)(65 + i));
printf("\n");
printf("\n GIVEN SYMBOLS FOR NFA: \t");
for (i = 0; i < symbols; i++)
printf("%d, ", i);
printf("eps");
printf("\n\n");
char *NFA_TABLE[states][symbols + 1];
// Hard coded input for NFA table
char *DFA_TABLE[MAX_LEN][symbols];
strcpy(&NFA_TABLE[0][0], "FC");
strcpy(&NFA_TABLE[0][1], "-");
strcpy(&NFA_TABLE[0][2], "BF");
strcpy(&NFA_TABLE[1][0], "-");
strcpy(&NFA_TABLE[1][1], "C");
strcpy(&NFA_TABLE[1][2], "-");
strcpy(&NFA_TABLE[2][0], "-");
strcpy(&NFA_TABLE[2][1], "-");
strcpy(&NFA_TABLE[2][2], "D");
strcpy(&NFA_TABLE[3][0], "E");
strcpy(&NFA_TABLE[3][1], "A");
strcpy(&NFA_TABLE[3][2], "-");
strcpy(&NFA_TABLE[4][0], "A");
strcpy(&NFA_TABLE[4][1], "-");
strcpy(&NFA_TABLE[4][2], "BF");
strcpy(&NFA_TABLE[5][0], "-");
strcpy(&NFA_TABLE[5][1], "-");
strcpy(&NFA_TABLE[5][2], "-");
printf("\n NFA STATE TRANSITION TABLE
\n\n\n");
printf("STATES\t");
for (i = 0; i < symbols; i++)
printf("|%d\t", i);
printf("eps\n");
// Displaying the matrix of NFA transition
table
printf("--------+---------------------------------\n"
);
for (i = 0; i < states; i++) {
printf("%c\t", (char)(65 + i));
for (j = 0; j <= symbols; j++) {
printf("|%s \t", &NFA_TABLE[i][j]);
}
printf("\n");
}
int closure_ar[states];
char *closure_table[states];
Display_closure(states, closure_ar,
closure_table, NFA_TABLE, DFA_TABLE);
```

```
strcpy(&dfa_states[last_index++].states,
"-");
dfa_states[last_index - 1].count = 1;
bzero((void *)buffer, MAX_LEN);
strcpy(buffer, &closure_table[0]);
strcpy(&dfa_states[last_index++].states,
buffer);
int Sm = 1, ind = 1;
int start_index = 1;
// Filling up the DFA table with transition
values
// Till new states can be entered in DFA
table
while (ind != -1) {
dfa_states[start_index].count = 1;
Sm = 0;
for (i = 0; i < symbols; i++) {
trans(buffer, i, closure_table, states,
NFA_TABLE, T_buf);
// storing the new DFA state in buffer
strcpy(&DFA_TABLE[zz][i], T_buf);
// parameter to control new states
Sm = Sm + new_states(dfa_states, T_buf);
}
ind = indexing(dfa_states);
if (ind != -1)
strcpy(buffer,
&dfa_states[++start_index].states);
zz++;
}
// display the DFA TABLE
Display_DFA(last_index, dfa_states,
DFA_TABLE);
return 0;
}
```

Output:

Develop a program 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);
}
```