CHRIST

COLLEGE OF ENGINEERING

LABORATORY RECORD

CS431 COMPILER DESIGN

CERTIFICATE

Name:……

Branch:…….Semester:……..Admission No. :…….

University Register No.:……

This is a bonafide record of practical laboratory work done by Mr./Ms. …… of 7th Semester Computer Science and Engineering in CS431 – Compiler Design Laboratory

HOD

Faculty in-charge

CONTENTS

Experiment No. :
Date:

Familiarization of Lexical Analyser

Lexical analysis is the very first phase in the compiler designing. It takes the modified source code which is written in the form of sentences. In other words, it helps you to convert a sequence of characters into a sequence of tokens. The lexical analysis breaks this syntax into a series of tokens. It removes any extra space or comment written in the source code.
Programs that perform lexical analysis are called lexical analyzers or lexers. A lexer contains tokenizer or scanner. If the lexical analyzer detects that the token is invalid, it generates an error. It reeads character streams from the source code, checks for legal tokens and pass the data to the syntax analyzer when it demands.

Lexeme: A lexeme is a sequence of characters included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

Token: The token is a sequence of characters representing a unit of information in the source program. Common token names are:

- Identifier: Names the programmer chooses
- Keyword: Names already in the programming language
- Separator (also known as punctuators): Punctuation characters and paired-delimiters
- Operator: Symbols that operate on arguments and produce results
- Literal: Numeric, logical, textual, reference literals
- Comment: Line, block

Lexical Grammar: Lexical grammar is a formal grammar defining the syntax of tokens. The program is written using characters that are defined by the lexical structure of the language used. The character set is equivalent to the alphabet used by any written language.

Pattern: A pattern is a description used by the token. In case of a keyword which uses as a token, the pattern is a sequence of characters.

Experiment No. :
Date:

Program to Design and Implement a Lexical Analyzer for given Language using C

AIM:
To design and implement a lexical analyzer for given language using C and the lexical analyzer should ignore redundant spaces, tabs and new lines.

PROGRAM:

```
#include<stdio.h>
#include<string.h>
FILE *fp;
int lineno=0;
char c;
char lexbuf[50],symtab[50][20];
int i=0,x;
char
kw[30][20]={"void","int","float","double","short","long","if","else","switch","c
ase","break","ret
urn","main","static","goto"};
char delim[]={'(',')','{','}','[',']',';',','};
char oper[]={'+','=','-','*','/','<','>'};
int isdelim(char);
int isoper(char);
int iskw(char[]);
void main()
{
fp=fopen("sample.c","r");
c=getc(fp);
while(c!=EOF)
{
if(c==' '||c=='\t');
else if(c=='\n')
{
lineno++;
```

```
}
else if((x=isdelim(c))!=-1)
{
printf("%c\t\tDelimiter\n",c);
}
else if((x=isoper(c))!=-1)
{
printf("%c\t\tOperator\n",c);
}
else if(isdigit(c))
{
int b=0;
while(isdigit(c))
{
lexbuf[b++]=c;
c=getc(fp);
}
ungetc(c,fp);
lexbuf[b]='\0';
printf("%s\t\tDigit\n",lexbuf);
}
else if(isalpha(c))
{
int b=0,k;
while(isalpha(c)||isdigit(c)||c=='_')
{
lexbuf[b++]=c;
c=getc(fp);
}
ungetc(c,fp);
lexbuf[b]='\0';
if((!(lookup(lexbuf)))&&(!iskw(lexbuf)))
{
strcpy(symtab[i++],lexbuf);
}
if((k=iskw(lexbuf))!=0)
{
```

```
printf("%s\t\t Keyword\n",lexbuf);
}
else
{
printf("%s\t\tIdentifier\n",lexbuf);
}
}
c=getc(fp);
}
fclose(fp);
printf("\nNumber of lines=%d\n",lineno-1);
}
//Is delimiter
int isdelim(char d)
{
int k;
for(k=0;k<8;k++)
{
if(d==delim[k])
{
return k;
}
}
return
-1;
}
//Is operator
int isoper(char op) {
int k;
//printf("%c
\n",op);
for(k=0;k<7;k++) {
if(op==oper[k]) {
return k; }}
return
-1;
}
```

```
int lookup(char s[]) {
int k;
for(k=0;k<i;k++) {
if((strcmp(s,symtab[k]))==0) {
return k+1; }
return 0; }}
int iskw(char s[]) {
int k;
for(k=0;k<15;k++) {
if(strcmp(s,kw[k])==0)
return k+1; }
return 0; }
```

RESULT:
Successfully executed the program

Experiment No. :
Date:

Program to Implement Lexical Analyser using Lex Tool

AIM:
To implement lexical analyser using Lex tool

PROGRAMS:
1. Lex program to count the number of characters in a file

```
%{
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;
}
```

2. Lex program to count the digits in a file

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

3. Lex program to count number of a's in a program

```
%{
int c=0,d=0;
%}
digit [0-9]
%%
a c++;
. ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count is %d\n",c);
}
int yywrap()
{
return 1;
}
%{
int negative=0;
int positive=0;
int positivefraction=0;
%}
%%
[-][0-9]+ {negative=negative+1;}
[+][0-9]+ {positive=positive+1;}
[+][0-9]+[.][0-9]+ {positivefraction=positivefraction+1;}
. ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count of negative no is %d\n",negative);
printf("count of positive no is %d\n",positive);
printf("count of positive fraction is %d\n",positivefraction);
```

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

4. Lex program to remove comment line

```
%{
int c=0;
%}
%%
\/\/.* ;
%%
int main()
{
yyin=fopen("b.c","r");
yyout=fopen("c.c","w");
yylex();
}
int yywrap()
{
return 1;
}
```

5. Lex program to count the number of identifiers

```
%{
int c=0;
%}
%%
[a-z_][a-z_0-9]* {c=c+1;}
.|\n ;
%%
int main()
{
yyin=fopen("b.c","r");
yylex();
printf("count is %d\n",c);
}
int yywrap()
{
return 1;
}
```

RESULT:

Successfully executed the programs

Experiment No. :
Date:

Program to Generate YACC Specification for a few Syntactic Categories

AIM:
To generate YACC specification for a few syntactic categories

PROGRAMS:
1. Program to implement a calculator using LEX and YACC
calculator.y

```
%{
#include <stdio.h>
//extern FILE *yyin;
%}
%token NUMBER
%start S
%%
S : E { printf("Expression_value= %d\n", $1); }
;
E : E '+' NUMBER { $$ = $1 + $3;
printf ("Recognized '+' expression.\n");
}
| E '-' NUMBER { $$ = $1 - $3;
printf ("Recognized '-' expression.\n");
}
| E '*' NUMBER { $$ = $1 * $3;
printf ("Recognized '*' expression.\n");
}
| E '/' NUMBER { if($3==0)
{
printf("Cannot divide by 0");
break;
}
else
$$ = $1 / $3;
```

```
printf ("Recognized '/' expression.\n");
 }
| NUMBER { $$ = $1;
printf ("Recognized a number.\n");
}
;
%%
int main ()
{
//yyin=fopen("s.txt","r");
do
{printf("Enter the expression\n");
yyparse();
}while(1);
return 1;
}
int yyerror (char *msg)
{
printf("Invalid Expression\n");
}
yywrap()
{
return(1);
}
```

calculator.l

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ { yylval = atoi (yytext);
printf ("scanned the number %d\n", yylval);
return NUMBER; }
[ \t] { printf ("skipped whitespace\n"); }
\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 '=' */
}
%%
```

2. Program to recognize a valid variable which starts with a letter followed by any number of 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 '=' */
 }
```

3. Program to recognize a valid arithmetic expression that uses operator +,-,*and /

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 '=' */
 }
```

RESULT:

Successfully executed the program

Experiment No. :
Date:

Program to find ε – closure of all states of any given NFA with ε transition

AIM:
To find ε – closure of all states of any given NFA with ε transition.

PROGRAM:

```
#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;

}
```

RESULT:
Successfully executed the program

Experiment No. :
Date:

Program to convert NFA to DFA

AIM:
To convert NFA to DFA

PROGRAM:

```
#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;
}
```

RESULT:
Successfully executed the program