# COMPILER DESIGN LAB MANUAL

## 1.DESIGN AND IMPLEMENT A LEXICAL ANALYSER FOR GIVEN LANGUAGE USING C AND THE LEXICAL ANALYSER SHOULD IGNORE REDUNDANT SPACES, TABS AND NEW LINES.

## SOURCE CODE:

```
#include<string.h>
#include<conio.h>
#include<ctype.h>
#include<stdio.h>
void main()
{
FILE *f1;
char c,str[10];
int lineno=1,num=0,i=0;
clrscr();
printf("\nEnter the c program\n");
f1=fopen("input.txt","w");
while((c=getchar())!=EOF)
putc(c,f1);
fclose(f1);
f1=fopen("input.txt","r");
while((c=getc(f1))!=EOF) // TO READ THE GIVEN FILE
{
if(isdigit(c)) // TO RECOGNIZE NUMBERS
{
 num=c-48;
 c=getc(f1);
 while(isdigit(c))
 {
 num=num*10+(c-48);
 c=getc(f1);
 }
 printf("%d is a number \n",num);
 ungetc(c,f1);
}
else if(isalpha(c)) // TO RECOGNIZE KEYWORDS AND IDENTIFIERS
```

```
{
 str[i++]=c;
 c=getc(f1);
 while(isdigit(c)||isalpha(c)||c=='_'||c=='$')
 {
 str[i++]=c;
 c=getc(f1);
 }
 str[i++]='\0';
 if(strcmp("for",str)==0||strcmp("while",str)==0||strcmp("do",str)==0||
 strcmp("int",str)==0||strcmp("float",str)==0||strcmp("char",str)==0||
 strcmp("double",str)==0||strcmp("static",str)==0||
 strcmp("switch",str)==0||strcmp("case",str)==0) // TYPE 32 KEYWORDS
 printf("%s is a keyword\n",str);
Blog - https://anilkumarprathipati.wordpress.com/ 6
 else
 printf("%s is a identifier\n",str);
 ungetc(c,f1);
 i=0;
}
else if(c==' '||c=='\t') // TO IGNORE THE SPACE
 printf("\n");
else if(c=='\n') // TO COUNT LINE NUMBER
 lineno++;
else // TO FIND SPECIAL SYMBOL
 printf("%c is a special symbol\n",c);
}
printf("Total no. of lines are: %d\n",lineno);
fclose(f1);
getch();
}
```

**INPUT :**

Enter the c program

```
int main()
{
int a=10,20;
charch;
float f;
}^Z
```

**OUTPUT:**

The numbers in the program are: 10 20

The keywords and identifiersare:

int is a keyword

main is an identifier

int is a keyword

a is an identifier

char is a keyword

ch is an identifier

float is a keyword

f is an identifier

Special characters are ( ) { = , ; ; ; }

Total no. of lines are:5

## 2. IMPLEMENTATION OF LEXICAL ANALYZER USING LEX TOOL.

## SOURCECODE:

```
#include<stdio.h>
#include<ctype.h>
#include<conio.h>
#include<string.h>
FILE *fp, *fp1;
void main()
{
 char s[10];
 clrscr();
 fp=fopen("i.dat","r");
 fp1=fopen("o.dat","w");
 while(!feof(fp))
 {
 fscanf(fp,"%s",&s);
 check(s);
 }
}
check(char s[10])
{
 fp1=fopen("o.dat","a+");
 if(strcmp (s, "read")==0||strcmp (s, "write")==0||
 strcmp (s, "while")==0||strcmp (s, "for")==0||
 strcmp (s, "if")==0||strcmp (s, "else")==0||
 strcmp (s, "endif")==0||strcmp (s, "then")==0)
 fprintf(fp1,"%s->keyword \n", s);
 else
 if (!isalpha(s[0]))
 {
 if(strcmp(s,",")==0)
fprintf (fp1,", ->comma\n");
 if(strcmp(s,"(")==0)
```

```
fprintf(fp1,"(->openbrace\n");
 if(strcmp(s,")")==0)
fprintf(fp1,")->closebrace\n");
 if(strcmp(s,";")==0)
fprintf(fp1,";->semicolon\n");
 if(strcmp(s,">")==0)
fprintf (fp1,">->greaterthan->\n");
 if (strcmp (s,"<")==0)
fprintf (fp1,"<-> less than \n");
 }
 else
 fprintf (fp1,"%s->identifier\n", s);
 fclose (fp1);
return 0;
}
```

## INPUT:

type i.dat

read a, b

if(a>b)

write a

else

write b

## OUTPUT:

type o.dat

read->keyword

a, b ->identifier

if(a>b)-> identifier

write-> keyword

a-> identifier

else-> keyword

write -> keyword

b-> identifier

b->identifier

## 3.(A). PROGRAM TO RECOGNIZE A VALID ARITHMETIC EXPRESSION THAT USES OPERATORS +, - , * AND /.

**SOURCE CODE:**

LEX PART:

```
%{
 #include "y.tab.h"
%}
%%
[a-zA-Z_][a-zA-Z_0-9]* return id;
[0-9]+(\.[0-9]*)? return num;
[+/*] return op;
. return yytext[0];
\n return 0;
%%
int yywrap()
{
return 1;
}
```

YACC PART:

```
%{
 #include<stdio.h>
 int valid=1;
%}
%token num id op
%%
start : id '=' s ';'
s : id x
 | num x
 | '-' num x
 | '(' s ')' x ;
```

```
x : op s
 | '-' s
 |
 ;
%%
int yyerror()
{
 valid=0;
 printf("\nInvalid expression!\n");
 return 0;
}
int main()
{
 printf("\nEnter the expression:\n");
 yyparse();
 if(valid) {
printf("\nValid expression!\n");
 }
}
```

**OUTPUT:**

Enter the expression:

a=b+c;

valid expression!

Enter the expression:

a=b+c;

Invalid expression!

Enter the expression:

a=b;

**valid expression!**

## 3.(B). PROGRAM TO RECOGNIZE A VALID VARIABLE WHICH STARTS WITH A LETTER FOLLOWED BY ANY NUMBER OF LETTERS OR DIGITS.

## SOURCE CODE:

```
//Program to recognize a valid variable
LEX PART:
%{
 #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;
}
YACC PART:
%{
 #include<stdio.h>
 int valid=1;
%}
%token digit letter
%%
start : letter s
s : letter s
 | digit s
 |
 ;
%%
int yyerror()
{
 printf("\nIts not a identifier!\n");
```

```
 valid=0;
 return 0;
}
int main()
{
 printf("\nEnter a name to tested for identifier ");
 yyparse();
 if(valid)
 {
 printf("\nIt is a identifier!\n");
 }
}
```

**OUTPUT:**

Enter a name to tested for identifier abc

It is a identifier!

Enter a name to tested for identifier _abc

It is a identifier!

Enter a name to tested for identifier 848_f

Its not a identifier!

## 3.(C). IMPLEMENTATION OF CALCULATOR USING LEX AND YACC.

## SOURCE CODE:

LEX PART:

```
%{
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ {
 yylval=atoi(yytext);
 return NUMBER;
 }
[\t] ;
[\n] return 0;
. return yytext[0];
%%
int yywrap()
{
return 1;
}
```

YACC PART:

```
%{
 #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 which can have operations Addition,
Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");
 yyparse();
 if(flag==0)
 printf("\nEntered arithmetic expression is Valid\n\n");
}
void yyerror()
{
 printf("\nEntered arithmetic expression is Invalid\n\n");
 flag=1;
}
```

**OUTPUT:**

enter any arithmetic expression which can have operations addition, subtraction, multiplication, division, modulus and round brackets:

result=0

entered arithmetic expression is valid

enter any arithmetic expression which can have operations addition, subtraction, multiplication, division, modulus, and round brackets:

(9=0)

entered arithmetic expression is invalid.

## 3.(D). CONVERTS THE BNF RULES INTO YACC FORM AND WRITE CODE TO GENERATE ABSTRACT SYNTAX TREE

## SOURCE CODE:

```
//Convert the BNF rules into YACC form and
//write code to generate Abstract Syntax Tree
LEX PART:
%{
#include"y.tab.h"
#include<stdio.h>
#include<string.h>
int LineNo=1;
%}
identifier [a-zA-Z][_a-zA-Z0-9]*
number [0-9]+|([0-9]*\.[0-9]+)
%%
main\(\) return MAIN;
if return IF;
else return ELSE;
while return WHILE;
int |
char |
float return TYPE;
{identifier} {strcpy(yylval.var,yytext);
return VAR;}
{number} {strcpy(yylval.var,yytext);
return NUM;}
\< |
\> |
\>= |
\<= |
== {strcpy(yylval.var,yytext);
return RELOP;}
[ \t] ;
\n LineNo++;
```

```
. return yytext[0];
%%
YACC PART:
%{
#include<string.h>
#include<stdio.h>
struct quad
{
char op[5];
char arg1[10];
char arg2[10];
char result[10];
}QUAD[30];
struct stack
{
int items[100];
int top;
}stk;
int Index=0,tIndex=0,StNo,Ind,tInd;
extern int LineNo;
%}
%union
{
char var[10];
}
%token <var> NUM VAR RELOP
%token MAIN IF ELSE WHILE TYPE
%type <var> EXPR ASSIGNMENT CONDITION IFST ELSEST WHILELOOP
%left '-' '+'
%left '*' '/'
%%
PROGRAM : MAIN BLOCK
;
BLOCK: '{' CODE '}'
```

```
;
CODE: BLOCK
| STATEMENT CODE
| STATEMENT
;
STATEMENT: DESCT ';'
| ASSIGNMENT ';'
| CONDST
| WHILEST
;
DESCT: TYPE VARLIST
;
VARLIST: VAR ',' VARLIST
| VAR
;
ASSIGNMENT: VAR '=' EXPR{
strcpy(QUAD[Index].op,"=");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,$1);
strcpy($$,QUAD[Index++].result);
};
EXPR: EXPR '+' EXPR {AddQuadruple("+",$1,$3,$$);}
| EXPR '-' EXPR {AddQuadruple("-",$1,$3,$$);}
| EXPR '' EXPR {AddQuadruple("",$1,$3,$$);}
| EXPR '/' EXPR {AddQuadruple("/",$1,$3,$$);}
| '-' EXPR {AddQuadruple("UMIN",$2,"",$$);}
| '(' EXPR ')' {strcpy($$,$2);}
| VAR
| NUM
;
CONDST: IFST{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
```

```
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
| IFST ELSEST
;
IFST: IF '(' CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK { strcpy(QUAD[Index].op,"GOTO"); strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
};
ELSEST: ELSE{
tInd=pop();
Ind=pop();
push(tInd);
sprintf(QUAD[Ind].result,"%d",Index);
}
BLOCK{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
};
CONDITION: VAR RELOP VAR {AddQuadruple($2,$1,$3,$$);
StNo=Index-1;
}
| VAR
| NUM
```

```
;
WHILEST: WHILELOOP{
Ind=pop();
sprintf(QUAD[Ind].result,"%d",StNo);
Ind=pop();
sprintf(QUAD[Ind].result,"%d",Index);
}
;
WHILELOOP: WHILE'('CONDITION ')' {
strcpy(QUAD[Index].op,"==");
strcpy(QUAD[Index].arg1,$3);
strcpy(QUAD[Index].arg2,"FALSE")
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
BLOCK {
strcpy(QUAD[Index].op,"GOTO");
strcpy(QUAD[Index].arg1,"");
strcpy(QUAD[Index].arg2,"");
strcpy(QUAD[Index].result,"-1");
push(Index);
Index++;
}
;
%%
extern FILE *yyin;
int main(int argc,char *argv[])
{
FILE *fp;
int i;
if(argc>1)
{
fp=fopen(argv[1],"r");
```

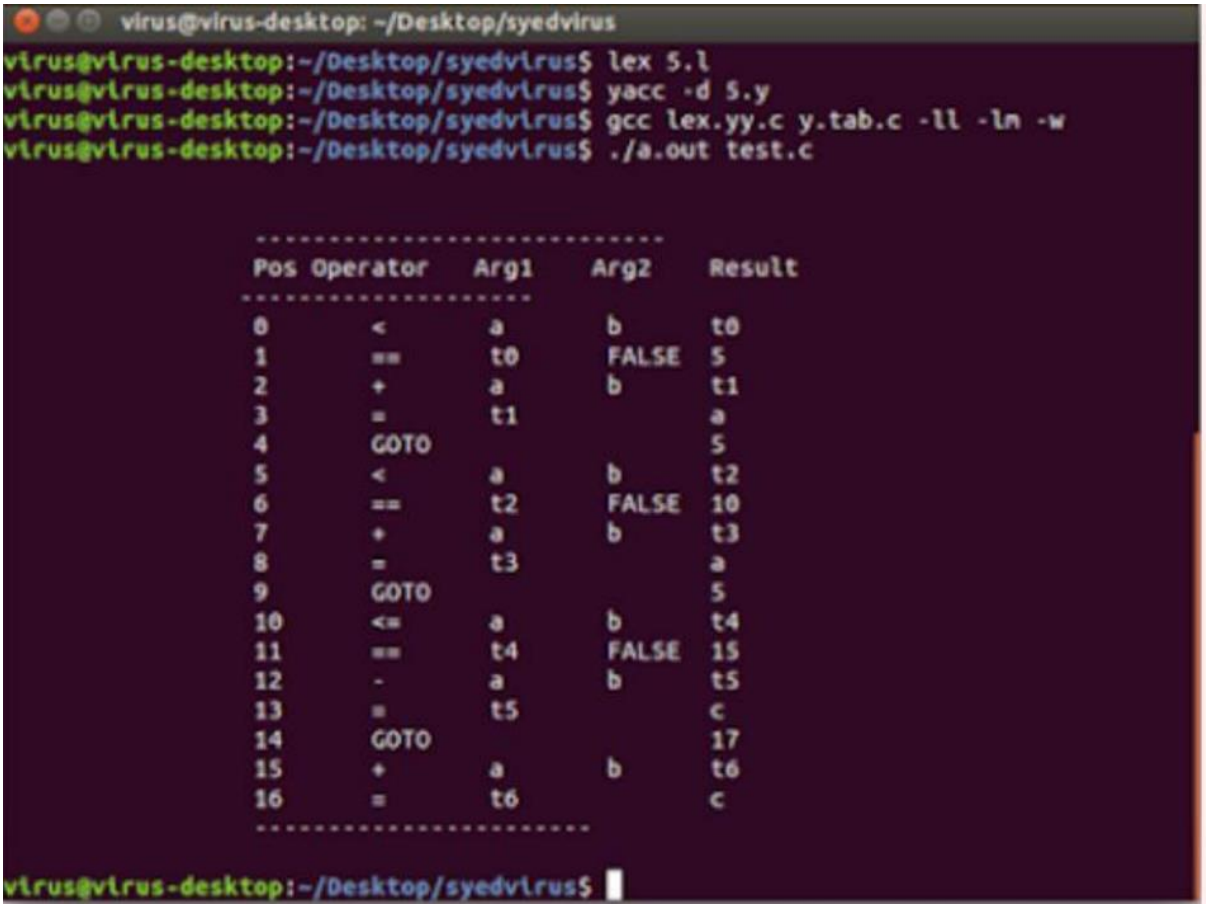
```
if(!fp)
{
printf("\n File not found");
exit(0);
}
yyin=fp;
}
yyparse();
printf("\n\n\t\t ----------""\n\t\t Pos Operator \tArg1 \tArg2 \tResult"
"\n\t\t----------");
for(i=0;i<Index;i++)
{
printf("\n\t\t %d\t %s\t %s\t
%s\t%s",i,QUAD[i].op,QUAD[i].arg1,QUAD[i].arg2,QUAD[i].result);
}
printf("\n\t\t ----------");
printf("\n\n"); return 0; }
void push(int data)
{ stk.top++;
if(stk.top==100)
{
printf("\n Stack overflow\n");
exit(0);
}
stk.items[stk.top]=data;
}
int pop()
{
int data;
if(tk.top==-1)
{
printf("\n Stack underflow\n");
exit(0);
}
```

```
data=stk.items[stk.top--];
return data;
}
void AddQuadruple(char op[5],char arg1[10],char arg2[10],char result[10])
{
strcpy(QUAD[Index].op,op);
strcpy(QUAD[Index].arg1,arg1);
strcpy(QUAD[Index].arg2,arg2);
sprintf(QUAD[Index].result,"t%d",tIndex++);
strcpy(result,QUAD[Index++].result);
}
yyerror()
{
printf("\n Error on line no:%d",LineNo);
}
```

## INPUT:

```
main()
{
int a,b,c;
if(a<b)
{
a=a+b;
}
while(a<b)
{
a=a+b;
}
if(a<=b)
{
c=a-b;
}
else
{
c=a+b;
}
}
```

## OUTPUT:


virus@virus-desktop: ~/Desktop/syedvirus
virus@virus-desktop:~/Desktop/syedvirus$ lex 5.l
virus@virus-desktop:~/Desktop/syedvirus$ yacc -d 5.y
virus@virus-desktop:~/Desktop/syedvirus$ gcc lex.yy.c y.tab.c -ll -lm -w
virus@virus-desktop:~/Desktop/syedvirus$ ./a.out test.c
Pos Operator Arg1 Arg2 Result
0 < a b t0
1 == t0 FALSE 5
2 + a b t1
3 = t1 a
4 GOTO 5
5 < a b t2
6 == t2 FALSE 10
7 + a b t3
8 = t3 a
9 GOTO 5
10 <= a b t4
11 == t4 FALSE 15
12 - a b t5
13 = t5 c
14 GOTO 17
15 + a b t6
16 = t6 c
virus@virus-desktop:~/Desktop/syedvirus$

## 4.WRITE PROGRAM TO FIND € CLOSURE OF ALL STATES OF ANY GIVEN NFA WITH TRANSITION.

## SOURCE CODE:

```
#include <stdio.h>
#include <stdbool.h>
#define max_states 100
// data structure to represent a state in the nfa
typedef struct {
 int statenum;
 int transitions[max_states]; // stores the epsilon transitions from this state
} state;
// function to perform depth-first search to find epsilon closure of a state
void findepsilonclosure(state nfa[], int currentstate, bool visited[]) {
 if (visited[currentstate])
 return;
visited[currentstate] = true;
 printf("%d ", currentstate);
 // recursively find epsilon closure for each epsilon transition
 for (int i = 0; nfa[currentstate].transitions[i] != -1; i++) {
 findepsilonclosure(nfa, nfa[currentstate].transitions[i], visited);
}
}
// function to find epsilon closure for all states in the nfa
void findepsilonclosures(state nfa[], int numstates) {
 for (int i = 0; i < numstates; i++) {
 printf("epsilon closure of state %d: ", i);
 bool visited[max_states] = {false};
 findepsilonclosure(nfa, i, visited);
 printf("\n");
 }
}
int main() {
 // example nfa representation with epsilon transitions
 // modify this based on your input nfa.
```

```
 state nfa[] = {
 {0, {1, 2, -1}}, // state 0 with epsilon transitions to states 1 and 2
 {1, {3, -1}}, // state 1 with epsilon transition to state 3
 {2, {4, -1}}, // state 2 with epsilon transition to state 4
 {3, {-1}}, // state 3 with no epsilon transitions
 {4, {-1}} // state 4 with no epsilon transitions
 };
 int numstates = sizeof(nfa) / sizeof(nfa[0]);
// find epsilon closure for all states
 findepsilonclosures(nfa, numstates);
return 0;
}
```

**Output:**

epsilon closure of state 0: 0 1 3 2 4

epsilon closure of state 1: 1 3

epsilon closure of state 2: 2 4

epsilon closure of state 3: 3

epsilon closure of state 4: 4

## 5.WRITE A PROGRAM TO CONVERT NFA WITH € TRANSITION TO NFA WITHOUT € TRANSITION.

## SOURCE CODE:

```
#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]={0};
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("\n states :" );
for(i=1;i<=nostate;i++)
print_e_closure(i);
printf("\ntnransitions 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!=0)
{
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("\n final 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]!=0)
 {
 temp=transition[x][noalpha-1];
 while(temp!=0)
 {
 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;
 printf("{");
 for(j=0;e_closure[i][j]!=0;j++)
 printf("q%d,",e_closure[i][j]);
 printf("}\t");
}
```

## OUTPUT:

```
enter the number of alphabets?
4
note:- [ use letter e as epsilon]
note:- [e must be last character ,if it is present]

enter alphabets?
a
b
c
e
enter the number of states?
3
enter the start state?
1
enter the number of final states?
1
enter the final states?
3
enter no of transition?
5
note:- [transition is in the form--> qno alphabet qno]
note:- [states number must be greater than zero]

enter transition?
1 a 1
1 e 2
2 b 2
2 e 3
3 c 3

equivalent nfa without epsilon
-----------------------------------
```

start state:{q1,q2,q3,}

alphabets:a b c e

states :{q1,q2,q3,} {q2,q3,} {q3,}

tnransitions are...:

{q1,q2,q3,} a {q1,q2,q3,}

{q1,q2,q3,} b {q2,q3,}

{q1,q2,q3,} c {q3,}

{q2,q3,} a {}

{q2,q3,} b {q2,q3,}

{q2,q3,} c {q3,}

{q3,} a {}

{q3,} b {}

{q3,} c {q3,}

final states:{q1,q2,q3,} {q2,q3,} {q3,}

## 6.WRITE A PROGRAM TO CONVERT NFA TO DFA.

## SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
#include<math.h>
int ninputs;
int dfa [100][2][100] = {0};
int state [10000] = {0};
char ch [10], str [1000];
int go [10000][2] = {0};
int arr [10000] = {0};
int main ()
{
int st, fin, in;
int f [10];
int i, j=3, s=0, final=0,flag=0,curr1,curr2,k,l;
int c;
printf ("\n follow the one based indexing\n");
printf ("\n enter the number of states::");
scanf ("%d", &st);
printf ("\n give state numbers from 0 to %d", st-1);
for (i=0; i<st; i++)
state[(int) (pow (2, i))] = 1;
printf ("\n enter number of final states\t");
scanf ("%d", &fin);
printf ("\n enter final states::");
for (i=0; i<fin; i++)
{
scanf ("%d", &f[i]);
}
int p, q, r, rel;
printf ("\n enter the number of rules according to nfa::");
scanf ("%d", &rel);
```

```
printf ("\n\n define transition rule as \n");
for (i=0; i<rel ; i++)
{
scanf ("%d%d%d", &p,&q,&r);
if (q==0)
dfa[p][0][r] = 1;
else
dfa[p][1][r] = 1;
}
printf ("\n enter initial state::");
scanf ("%d", &in);
in = pow (2, in);
i=0;
printf ("\n solving according to dfa");
int x=0;
for (i=0; i<st; i++)
{
for (j=0; j<2; j++)
{
int stf=0;
for (k=0; k<st; k++)
{
if (dfa [i][j][k] ==1)
stf = stf + pow (2, k);
}
go[(int) (pow (2, i))][j] = stf;
printf("%d-%d-->%d\n", (int) (pow (2, i)), j, stf);
if(state[stf]==0)
arr[x++] = stf;
state[stf] = 1;
}
}
//for new states
for (i=0; i<x; i++)
```

```
{
printf ("for %d ---- ", arr[x]);
for (j=0; j<2; j++)
{
int new=0;
for (k=0; k<st; k++)
{
if(arr[i] & (1<<k))
{
int h = pow (2, k);
if(new==0)
new = go[h][j];
new = new | (go[h][j]);
}
}
if(state[new]==0)
{
arr[x++] = new;
state[new] = 1;
}
}
}
printf ("\n the total number of distinct states are::\n");
printf ("state 0 1\n");
for (i=0; i<10000; i++)
{
if(state[i]==1)
{
//printf ("%d**", i);
int y=0;
if(i==0)
printf ("q0 ");
else
for (j=0; j<st; j++)
```

```
{
int x = 1<<j;
if (x& i)
{
printf ("q %d ", j);
y = y+ pow (2, j);
//printf ("y=%d ", y);
}
}
//printf ("%d", y);
printf (" %d %d", go[y][0], go[y][1]);
printf("\n");
}
}
 j=3;
while(j--)
{
printf ("\n enter string");
scanf ("%s", str);
l = strlen(str);
curr1 = in;
flag = 0;
printf ("\n string takes the following path-->\n");
printf ("%d-",curr1);
for (i=0; i<l; i++)
{
curr1 = go[curr1] [str[i]-'0'];

printf ("%d-", curr1);
}
 printf ("\n final state - %d\n", curr1);
for (i=0; i<fin; i++)
{
if (curr1 & (1<<f[i]))
```

```
{
flag = 1;
break;
}
}
if(flag)
printf ("\n string accepted");
else
printf ("\n string rejected");
}
return 0;
}
```

## OUTPUT:

```
follow the one based indexing
 enter the number of states::2
 give state numbers from 0 to 1
 enter number of final states  1
 enter final states::1
 enter the number of rules according to nfa::2
 define transition rule as
0 0 0
0 1 1
 enter initial state::0
 solving according to dfa1-0-->1
1-1-->2
2-0-->0
2-1-->0
for 0 ----
 the total number of distinct states are::
state 0 1
q0  0 0
q 0  1 2
q 1  0 0
enter string
00
 string takes the following path-->
1-1-1-
 final state - 1
 string rejected
 enter string
01
 string takes the following path-->
1-1-2-
 final state - 2
 string accepted
```

## 7.Write a program to minimize any given DFA.

## Source Code:

```
#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;
}
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';
}
 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 */
}
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)++;
   }
}
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;
}
 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;
}
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);
         }
}
 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) */
}
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");
}
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 */
}
int is_subset(char *s, char *t)
{
int i;
 for (i = 0; *t; i++)
       if (!strchr(s, *t++)) return 0;
   return 1;
}
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);
}
```

## OUTPUT:

DFA: STATE TRANSITION TABLE

```
   |  0   1
-----+------------
 A  |  B   C
 B  |  E   F
 C  |  A   A
 D  |  F   E
 E  |  D   F
 F  |  D   E
```

Final states = EF

EQUIV. CLASS CANDIDATE ==> 0:[ABCD] 1:[EF]

0:[ABCD] --> [BEAF] (0101)

0:[ABCD] --> [CFAE] (0101)

1:[EF] --> [DD] (00)

1:[EF] --> [FE] (11)

EQUIV. CLASS CANDIDATE ==> 0:[AC] 1:[BD] 2:[EF]

0:[AC] --> [BA] (10)

0:[AC] --> [CA] (00)

1:[BD] --> [EF] (22)

1:[BD] --> [FE] (22)

2:[EF] --> [DD] (11)

2:[EF] --> [FE] (22)

EQUIV. CLASS CANDIDATE ==> 0:[A] 1:[BD] 2:[C] 3:[EF]

0:[A] --> [B] (1)

0:[A] --> [C] (2)

1:[BD] --> [EF] (33)

1:[BD] --> [FE] (33)

2:[C] --> [A] (0)

2:[C] --> [A] (0)

3:[EF] --> [DD] (11)

3:[EF] --> [FE] (33)

DFA: STATE TRANSITION TABLE

| 0 1

-----+----------

A | B C

B | D D

C | A A

D | B D

Final states = D

## 8. DEVELOP AN OPERATOR PRECEDENCE PARSER FOR A GIVEN LANGUAGE.

## SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
void main()
{
char stack[20],ip[20],opt[10][10][1],ter[10];
inti,j,k,n,top=0,row,col;
clrscr();
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:");
scanf("%d",&n);
printf("\nEnter the terminals:");
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("\nOPERATOR PRECEDENCE TABLE:\n");
for(i=0;i<n;i++)
```

```
{
printf("\t%c",ter[i]);
}
printf("\n_____");
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("\n\nEnter the input string(append with $):");
scanf("%s",ip);
i=0;
printf("\nSTACK\t\t\tINPUT STRING\t\t\tACTION\n");
printf("\n%s\t\t\t%s\t\t\t",stack,ip);
while(i<=strlen(ip))
{
for(k=0;k<n;k++)
{
if(stack[top]==ter[k])
row=k;
if(ip[i]==ter[k])
col=k;
}
if((stack[top]=='$')&&(ip[i]=='$'))
{
printf("String is ACCEPTED");
break;
}
else if((opt[row][col][0]=='<') ||(opt[row][col][0]=='='))
```

```
{
stack[++top]=opt[row][col][0];
stack[++top]=ip[i];
ip[i]=' ';
printf("Shift %c",ip[i]);
i++;
}
else
{
if(opt[row][col][0]=='>')
{
while(stack[top]!='<')
{
--top;
}
top=top-1;
printf("Reduce");
}
else
{
printf("\nString is not accepted");
break;
}
}
printf("\n");
printf("%s\t\t\t%s\t\t\t",stack,ip);
}
getch();
}
```

## OUTPUT:

```
Enter the no.of terminals:4
Enter the terminals:i+*$
Enter the table values:
Enter the value for i i:
-
Enter the value for i +:>
Enter the value for i *:>
Enter the value for i $:>
Enter the value for + i:<
Enter the value for + +:>
Enter the value for + *:<
Enter the value for + $:>
Enter the value for * i:<
Enter the value for * +:>
Enter the value for * *:>
Enter the value for * $:>
Enter the value for $ i:<
Enter the value for $ +:<
Enter the value for $ *:<
Enter the value for $ $:-
OPERATOR PRECEDENCE TABLE:
 i + * $
_____________________________________
i | - > > >
+ | < > < >
* | < > > >
$ | < < < -
Enter the input string(append with $):i+i*i$
STACK INPUT STRING ACTION
$ i+i*i$ Shift
$<i +i*i$ Reduce
$<i +i*i$ Shift
```

$<+ i*i$ Shift

$<+<i *i$ Reduce

$<+<i *i$ Shift

$<+<* i$ Shift

$<+<*<i $ Reduce

$<+<*<i $ Reduce

$<+<*<i $ Reduce

$<+<*<i $ String is ACCEPTED

## 9.WRITE A PROGRAM TO FIND SIMULATE FIRST AND FOLLOW OF ANY GIVEN GRAMMAR

**SOURCE CODE:**

```
#include<stdio.h>
#include<math.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>
#include<conio.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()
{
int i,z;
char c,ch;
clrscr();
printf("Enter the no of prooductions:\n");
scanf("%d",&n);
printf("Enter the productions:\n");
for(i=0;i<n;i++)
scanf("%s%c",a[i],&ch);
do{
m=0;
printf("Enter the elements whose first&follow is to be found:");
scanf("%c",&c);
first(c);
printf("First(%c)={",c);
for(i=0;i<n;i++);
printf("%c",f[i]);
printf("}\n");
strcpy(f," ");
m=0;
follow(c);
printf("Follow(%c)={",c);
for(i=0;i<n;i++);
printf("%c",f[i]);
printf("}\n");printf("Continue(0/t)?");
```

```
scanf("%d%c",&z,&ch);
}
while(z==1);
}
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]=='$')
follow(a[k][0]);
else if(islower(a[k][2]))
f[m++]=a[k][2];
else first(a[k][2]);
}
}
}
void follow(char c)
{
if(a[0][0]==c)
f[m++]='$';
for(i=0;i<n;i++)
{
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]);
}
}
}
getch();
}
```

**OUTPUT:**

```
Enter the no of productions:
3
S=AB
A=a
B=b
Enter the elements whose first&follow is to be found:S
First(S)={a}
Follow(S)={$}
Continue(0/t)?1
Enter the elements whose first&follow is to be found:A
First(A)={a}
Follow(A)={b}
Continue(0/t)?1
Enter the elements whose first&follow is to be found:S
First(B)={b}
Follow(B)={$}
Continue(0/t)?0
```

**10. CONSTRUCT A RECURSIVE DESCENT PARSER FOR AN EXPRESSION.**

**SOURCECODE:**

```
#include<stdio.h>
#include<ctype.h>
char str [15];
int p, chk=1;
void main ()
{
 printf ("enter the input string:");
 scanf ("%s", &str);
 printf ("%s", str);
 p=0;
 e();
 if(p==strlen(str)&&chk)
 printf ("\n string is acceptable");
 else
 {
 printf ("\n the given input string: %s", str);
 printf ("\n the string is not acceptable");
 }
 getch ();
}
e ()
{
 printf("\n e->te'");
 t ();
 eprime ();
}
eprime ()
{
 if(str[p]=='+')
 {
 printf ("\n e'->+te'");
 p++;
 t ();
 eprime ();
 }
 else
 printf("\n e'->#");
}
```

```
t ()
{
 printf ("\n t->ft'");
 f ();
 tprime ();
}
tprime ()
{
 if(str[p]=='*')
 {
 printf ("\n t'->*ft'");
 p++;
 f ();
 tprime ();
 }
 else
 printf ("\n t'->#");
}
f ()
{
 if(isalpha(str[p]))
 {
 p++;
 printf ("\n f->i");
 }
 else
 if(str[p]=='(')
 {
 printf("\nf->(e)");
 p++;
 e ();
if(str[p]==')')
p++;
 else
chk=1;
 }
 else
 chk=1;
}
```

**OUTPUT:**

enter the input string: i+i*i

i+i*i

e-->te'

t-->ft'

f-->i

t'-->#

e'-->+te'

t-->ft'

f-->i

t'-->*ft'

f-->i

t'-->#

e'-->#

string is acceptable

## 11. CONSTRUCT A SHIFT REDUCE PARSER FOR A GIVEN LANGUAGE.

**SOURCE CODE**:

```
#include<stdio.h>
int a [10] = {121,131,514,6,181};
int stack [20], top=-1;
void push (int item)
{
 if(top>=20)
 {
 printf("overflow");
 exit (1);
 }
 top++;
 stack[top]=item;
}
char convert (int item)
{
 char ch;
 switch(item)
 {
 case 1:ch='e'; break;
 case 2:ch='*'; break;
 case 3:ch='+'; break;
 case 4:ch='('; break;
 case 5:ch=')'; break;
 case 6:ch='i'; break;
 case 7:ch='$'; break;
 case 8:ch='='; break;
 }
 return(ch);
```

```
}
int action (int item)
{
 int i;
 for (i=0; i<=4; i++)
 {
 if(a[i]==item)
 return 1;
 }
 return -1;
}
void main ()
{
 char ipstr [20];
 int ip [20],i,s,sum,j , l,m,n, cnt;
 clrscr ();
 printf ("\n enter the input string:");
 scanf ("%s", ipstr);
 for (i=0; ipstr[i]!='\0';i++)
 {
 switch(ipstr[i])
 {
 case 'e': s=1; break;
 case '*': s =2; break;
 case '+': s=3; break;
 case '(': s=4; break;
 case ')': s=5; break;
 case 'i': s=6; break;
 case '$': s=7; break;
 case '=': s=8; break;
```

```
default: printf ("error");
exit (1);
}
ip[i]=s;
}
ip[i]=-1;
i=0;
push (7);
printf ("\t stack \t input \t action \n");
printf("\t----------------------------\n\n");
while (1)
{
printf("\t");
for (m=0; m<=top; m++)
printf ("%c", convert (stack[m]));
printf("\t");
for (n=i; ip[n]!= -1;n++)
printf ("%c", convert(ip[n]));
printf("\t");
sum=0;
if((stack[top]==1) &&(stack[top-1] ==7) &&(ip[i]==7))
{
printf ("\t accept");
exit (1);
}
else
{
l=1;
while(l<=top)
{
```

```
 j=l;
 sum=0; cnt=0;
 for (; j<=top; j++)
 {
sum=sum*10+stack[j];
cnt++;
 }
 s=action(sum);
 if(s==1)
 {
top=top-cnt;
push (1);
printf ("\t reduce");
break;
 }
 l++;
 }
 if(s==-1&&ip[i]==7)
 {
 printf("\terror");
 exit (1);
 }
 }
 if(ip[i]! =7)
 {
 push(ip[i]);
 i++;
 printf ("\t shift");
 }
 printf("\n");
```

```
 }
}
```

**OUTPUT:**

enter the input string: i*i+i$

stack input action

---

$ i*i+i$ shift

$i *i+i$ reduce shift

$e* i+i$ shift

$e*i +i$ reduce shift

$e*e+ i$ shift

$e*e+i $ reduce

$e*e+e $ reduce

$e*e $ reduce

$e $ accept

## 12. WRITE A PROGRAM TO PERFORM LOOP UNROLLING.

## SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
int countbit1(unsigned int),countbit2(unsigned int);
void main() {
 unsigned int n;
 int x;
 int ch;
 printf("\nenter n\n");
 scanf("%u", & n);
 printf("\n1. loop roll\n2. loop unroll\n");
 printf("\nenter ur choice\n");
 scanf(" %d", & 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:

```
enter n
3
1. loop roll
2. loop unroll
enter ur choice
2
no of iterations 1
loop unroll: count of 1's :2
```

## 13.WRITE A PROGRAM TO PERFORM CONSTANT PROPAGATION.

## SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<conio.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()
{
clrscr();
input();
constant();
output();
getch();
}
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);
}
}
```

## INPUT:

enter the maximum number of expressions : 4

enter the input :

= 3 - a

+ a b t1

+ a c t2

+ t1 t2 t3

## OUTPUT:

optimized code is :

+ 3 b t1

+ 3 c t2

+ t1 t2 t3

## 14.IMPLEMENT INTERMEDIATE CODE GENERATION FOR SIMPLE EXPRESSION.

## SOURCE CODE:

```
#include<stdio.h>
#include<string.h>
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp
{
int pos;
char op;
}k[15];
void main()
{
printf("\t\tintermediate code generation\n\n");
printf("enter the expression :");
scanf("%s",str);
printf("the intermediate code:\n");
findopr();
explore();
}
void findopr()
{
for(i=0;str[i]!='\0';i++)
if(str[i]==':')
{
k[j].pos=i;
k[j++].op=':';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='/')
```

```
{
k[j].pos=i;
k[j++].op='/';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='*')
{
k[j].pos=i;
k[j++].op='*';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='+')
{
k[j].pos=i;
k[j++].op='+';
}
for(i=0;str[i]!='\0';i++)
if(str[i]=='-')
{
k[j].pos=i;
k[j++].op='-';
}
}
void explore()
{
i=1;
while(k[i].op!='\0')
{
fleft(k[i].pos);
fright(k[i].pos);
str[k[i].pos]=tmpch--;
printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
printf("\n");
i++;
```

```
}
fright(-1);
if(no==0)
{
fleft(strlen(str));
printf("\t%s := %s",right,left);
exit(0);
}
printf("\t%s := %c",right,str[k[--i].pos]);
getch();
}
void fleft(int x)
{
int w=0,flag=0;
x--;
while(x!= -1 &&str[x]!= '+' &&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-
'&&str[x]!='/'&&str[x]!=':')
{
if(str[x]!='$'&& flag==0)
{
left[w++]=str[x];
left[w]='\0';
str[x]='$';
flag=1;
}
x--;
}
}
void fright(int x)
{
int w=0,flag=0;
x++;
while(x!= -1 && str[x]!=
'+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'&&str[x]!='/')
```

```
{
if(str[x]!='$'&& flag==0)
{
right[w++]=str[x];
right[w]='\0';
str[x]='$';
flag=1;
}
x++;
}
}
```

**OUTPUT:**

enter the expression: w:=a*b+c/d-e/f+g*h

the intermediate code:

z: =c/d

y: =e/f

x: =a*b

w: =g*h

v: =x+z

u: =y+w

t: =v-u

w: =t