Experiment-I: Scanner programs using C

1. **Problem Statement :** ‘C’ Program to recognize Comments

Description : C program using switch and if statements to recognize Comment line which starts with // (Singleline Comment) or enclosing in /* and */ symbols (Multi line Comments)

Source Code:

```
#include<stdio.h>
main(){
 int state=1;
 int ip=0;
 char input[20];
 printf("Enter the input string:\n");
 scanf("%s",input);
 while((state>=1)&&(state<=4))  {
   switch(state)    {
     case 1:if(input[ip]=='/')    {
              ip++;
              state=2;
            }
            else
              state=6;
            break;
     case 2:if(input[ip]=='*') {
              ip++;
              state=3;
            }
            else if(input[ip]=='/'){
               ip++;
               state=5;
               }
               else{ip++;state=6;}
```

```
        break;
 case 3:if(input[ip]=='*') {
          ip++;
          state=4;
        }
        else {
           if(input[ip]=='\0')
                   state=6;
           else{
          ip++;
          state=3;}
        }
        break;
 case 4:if(input[ip]=='/') {
          ip++;
          state=5;
        }
        else
        {
          if(input[ip]=='*')  {
            ip++;
            state=4; }
          else  {
            ip++;
            state=3;
          } }
        break;
  }    }
if(state==5)
  printf("It is a comment line\n");
else
```

```
       printf("It is not a comment line\n");
 }
```

Input / Output:

```
$ gcc commentsinc.c
$ ./a.out
Enter the input string:
//sddd
It is a comment line
$ ./a.out
Enter the input string:
/*sdsdsd
It is not a comment line
$ ./a.out
Enter the input string:
/*dsdsd*/
It is a comment line
$ ./a.out
Enter the input string:
sdsd
It is not a comment line
$
```

2. **Problem Statement :** ‘C’ Program to check whether a given string is a keyword or an identifier.

Description : C program to read the input string and check whether the given string is a keyword or an identifier. First the program checks if the given input is the keyword and laterTo check for an identifier , if the initial character of the string is numerical or any special character except ‘_’ then print it is “Neither an identifier nor a keyword “. Otherwise print it as valid identifier if remaining characters of string doesn’t contains any special characters except ‘_’.

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
int keyword(char b[])
{
int i=0,l;
char c;
for(i=0;i<10;i++)
{
	l=strcmp(key[i],b);
	if(l==0)	{
		printf("%s is an keyword",b);
		exit(0);	}
}
}
void main()
{
char a[10];
int flag, i=1;
printf("\n Enter an identifier:");
gets(a);
keyword(a);
if(isalpha(a[0]))
 flag=1;
 else
 printf("\n Neither an identifier nor a keyword");
while(a[i]!='\0')
{
 if(!isdigit(a[i])&&!isalpha(a[i]))
 {
 flag=0;
 break;
 }
 i++;
}
if(flag==1)
 printf("\n Valid identifier");
}
```

Input / Output:

```
$ ./a.out
 Enter an identifier:if
if is an keyword
$ ./a.out
 Enter an identifier:vasavi
Valid identifier
$ ./a.out
 Enter an identifier:89vasavi
Neither an identifier nor a keyword
```

3. **Problem Statement :** Implement Lexical analyzer / Scanner using C

Description : Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer, lexer or scanner. A token is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). The process of forming tokens from an input stream of characters is called tokenization and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

Source Code:

```
 #include<stdio.h>
#include<ctype.h>
#include<string.h>
#define NONE -1
#define EOS '\0'
#define NUM 256
#define KEYWORD 257
#define PAREN 258
#define ID 259
#define ASSIGN 260
#define REL_OP 261
#define DONE 262
#define MAX 999
char lexemes[MAX];
char buffer[SIZE];
int lastchar=-1;
int lastentry =0;
int tokenval=NONE;
int lineno=1;
struct entry
{
char *lexptr;
```

```
int token;
}
symtable[100];
/*stores the list of keywords*/
struct entry keywords[]=
{
0,0};
/*THIS FUNCTION ISSSUE A COMPILER ERROR*/
void Error_message(char *m)
{
fprintf(stderr,"line %d: %s\n",lineno,m);
exit(1);

}
/*this function is used to search the symbol table for particular entry*/
int look_up(char s[])
{
int k;
for(k=lastentry;k>0;k=k-1)
if(strcmp(symtable[k].lexptr,s)==0)
return k;
return 0;
}
int insert(char s[],int tok)
{
int len;
len=strlen(s);
if((lastentry+1)>=MAX)
Error_message("Symbol table is full");
if((lastchar+len+1)>=MAX)
Error_message("lexemas array is full");
```

```
lastentry=(lastentry +1);
symtable[lastentry].token=tok;
symtable[lastentry].lexptr=&lexemes[lastchar+1];
lastchar=lastchar + len +1;
strcpy(symtable[lastentry].lexptr,s);
return lastentry;
}
void Initialize()
{
struct entry *ptr;
for(ptr=keywords;ptr->token;ptr++)
insert(ptr->lexptr,ptr->token);
}
int lexer()
{
int t;
int val,i=0;
while(1)
{
t=getchar();
if(t==' '||t=='\t')
 ;
else
 if(t=='\n')
lineno=lineno+1;
else if(t=='('||t==')')
return PAREN;
else if(t=='<'||t=='>'||t==">="||t=='<='||t=='!=')
return REL_OP;
else if(t=='=')
return ASSIGN;
```

```
else if(isdigit(t))
{
ungetc(t,stdin);
scanf("%d",&tokenval);
return NUM;
}
else if(isalpha(t))
{
while(isalnum(t))
{
buffer[i]=t;
t=getchar();
i=i+1;
if(i>=SIZE)
Error_message("compiler error");
}
buffer[i]=EOS;
if(t!=EOF)
ungetc(t,stdin);
val=look_up(buffer);
if(val==0)
val= insert(buffer,ID);
tokenval=val;
return symtable[val].token;
}
else if(t==EOF)
return DONE;
else
{
tokenval=NONE;
return t;
}
}
}
 main()
{
int lookahead;
char ans;
printf("\n\t\t  program for lexical analysis \n");
Initialize();
printf("\n Enter the expresssion  and put ; at the end");
printf("\n press ctrl z to terminate ....\n");
lookahead=lexer();
while(lookahead!=DONE)
{
if(lookahead==NUM)
{
printf("\n NUmber:");
printf("%d",tokenval);
}
if(lookahead=='+'||lookahead=='-'||lookahead=='*'||lookahead=='/')
printf("\n operator");
if(lookahead==PAREN)
printf("\n parenthesis");
if(lookahead==ID)
{
printf("\n Identifier:");
printf("%s",symtable[tokenval].lexptr);
}
if(lookahead==KEYWORD)
printf("\n keyword");
if(lookahead==ASSIGN)
```

```
printf("\n Assignment operator");
if(lookahead==REL_OP)
printf("\n relational operator");
lookahead=lexer();
}
}
```

Input / Output:

```
$ gcc scanner.c
$ ./a.out
                program for lexical analysis
 Enter the expresssion  and put ; at the end
 press ctrl z to terminate ....
a=10
 Identifier:a
 Assignment operator
 NUmber:10
main()
 Identifier:main
 parenthesis
 parenthesis
for
 keyword
return
 keyword
```

Experiment-II: Scanner programs using Lex

1.Problem Statement : Lex programs to recognize Keywords.

Description : token can look like anything that is useful for processing an input text stream or text file.

Source Code:

```
%{
#include<stdio.h>
%}
%%
int|float|char { printf(" data type: %s",yytext); }
%%
main()
{
yylex();
return(0);
}
```

Input / Output:

```
int

datatype: int
```

2.Problem Statement : Lex programs to recognize String ending with 00.

Description : token can look like anything that is useful for processing an input text stream or text file.

Source Code:

```
%{
 #include<stdio.h>
%}
%%
[a-z A-Z 0-9]++00 { printf("string is acepted",yytext);}
.* { printf("not acepted",yytext);}
%%
main()
{
yylex();
return(0);
}
```

Input / Output:

```
as100
String is accepted
```

3.Problem Statement : Program to recognize the strings which are starting or ending with 'k'.

Description : token can look like anything that is useful for processing an input text stream or text file.

Source Code:

```
%{
 #include<stdio.h>
%}
begin-with-k k.*
end-with-k .*k
%%
{begin-with-k} { printf(" %s is a word that begin with k",yytext);}
{end-with-k} { printf(" %s is a word that end with k",yytext);}
%%
main()
{
yylex();
return(0);
}
```

Input / Output:

```
kishore
kishore is a word that begin with k
shaik
shaik is a word that end with k
```

4.Problem Statement : program to assign line numbers for source code.

Description : token can look like anything that is useful for processing an input text stream or text file.

Source Code:

```
%{
 #include<stdio.h>
 int lineno=0;
%}
line .*\n
%%
{line}   { printf("%d .%s",lineno++,yytext); }
%%
main()
{
yylex();
return (0);
}
```

Input / Output:

```
abc
def
ghi

1.abc
2.def
3.ghi
```

5.Problem Statement : Program to recognize the numbers which has 1 in its 5th position from right.

Description : token can look like anything that is useful for processing an input text stream or text file.

Source Code:

```
%{
#include<stdio.h>
%}
%%
[0-9]*1[0-9]{4} { printf("acepted:");}
[0-9]*1[0-9] { printf("NOT acepted:");}
%%
main()
{
yylex();
return(0);
}
```

Input / Output:

```
3410001
accepted
123456
NOT accepted
```

6.Problem Statement : Implement lexical analyzer in Lex.

Description : The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial look a head is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

The general format of Lex source is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

Source Code:

```
%{
int COMMENT=0;
%}
identifier [a-z A-Z][a-z A-Z 0-9]*
%%
#.* {printf("\n %s is a PREPROCESSOR DIRECTIVE",yytext);}
int|float|char|double|while|for|do|if|break|continue|void|switch|case|long|struc
t|const|typedef|return|else|goto|main {printf("\n\t %s is a KEYWORD",yytext);}
"%*" {COMMENT=1;}
"*/" {COMMENT=0;}
{identifier}\( {if(!COMMENT) printf("\n\n FUNCTION \n\t%s",yytext); }
\{ {if(!COMMENT) printf("\n BLOCK BEGINS");}
\} {if(!COMMENT) printf("\n BLOCK ENDS");}
{identifier}(\ [[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
\".*\" { if(!COMMENT) printf("\n\t%s is a STRING",yytext);}
[0-9]+ { if(!COMMENT) printf("\n\t%s is a NUMBER",yytext);}
\)(\;)? {if(!COMMENT) printf("\n\t");
ECHO;
printf("\n");}
\(ECHO;
= {if(!COMMENT) printf("\n\t is an ASSIGNMENT OPERATOR",yytext);}
\<=|\>=|\<|==|\> {if(!COMMENT) printf("\n\t %s is a RELATIONAL OPERATOR",yytext)
;}
\b|\t {if(!COMMENT) printf("\n white space",yytext);}
\n { ; }
```

```
%%
int main(int argc,char **argv)
{
if(argc>1)
{
FILE *file;
file=fopen(argv[1],"r");
if(!file)
{
printf("could not open %s\n",argv[1]);
exit(0);
}
yyin=file;
}
yylex();
printf("\n\n");
return 0;
}
int yywrap()
{
return 0;
}
```

INPUT: file input.c

```
#include<stdio.h>
Main()
{ int i=10;
}
```

OUTPUT:

```
#include<stdio.h> is a  PREPROCESSOR DIRECTIVE
Main    KEYWORD
(
)
 BLOCK BEGINS
 int  KEYWORD ; IDENTIFIER
     i IDENTIFIER
   =  is an ASSIGNMENT OPERATOR
   10 is a NUMBER
BLOCK ENDS
```

Experiment-III: Find first set and follow set

1.Problem Statement :C program to find first set of the variable in the given productions.

Description : token can look like anything that is useful for processing an input text stream or text

The rules for finding FIRST of a given grammar is:

1.If X is terminal, then FIRST(X) is {X}.
2.If X --> ε is a production, then add ε to FIRST (X).
3.If X is a nonterminal and X--> Y1Y2 . . . Yk is a production, then if Y1 doesnt derive ε all in Y1 will come under X else move to Y2 and so on.

file.

Source Code:

```
#include<stdio.h>
#include<ctype.h>
void FIRST(char );
int count,n=0;
char prodn[10][10], first[10];

main()
{
int i,choice;
char c,ch;
printf("How many productions ? :");
scanf("%d",&count);
printf("Enter %d productions epsilon= $ :\n\n",count);
for(i=0;i<count;i++)
scanf("%s%c",prodn[i],&ch);
do
{
n=0;
printf("Element :");
scanf("%c",&c);
FIRST(c);
printf("\n FIRST(%c)= { ",c);
for(i=0;i<n;i++)
printf("%c ",first[i]);
printf("}\n");

printf("press 1 to continue : ");
scanf("%d%c",&choice,&ch);
}
while(choice==1);
}

void FIRST(char c)
{
int j;
if(!(isupper(c)))first[n++]=c;
for(j=0;j<count;j++)
{
if(prodn[j][0]==c)
{
if(prodn[j][2]=='$') first[n++]='$';
else if(islower(prodn[j][2]))first[n++]=prodn[j][2];
else FIRST(prodn[j][2]);
}
}
}
```

Input / Output:

```
$ gcc first.c
$ ./a.out
How many productions ? :8
```

```
Enter 8 productions epsilon= $ :

E=TD
D=+TD
D=$
T=FS
S=*FS
S=$
F=(E)
F=a
Element :E

 FIRST(E)= { ( a }
press 1 to continue : 1
Element :D

 FIRST(D)= { + $ }
press 1 to continue : 1
Element :D

 FIRST(D)= { + $ }
press 1 to continue : 0
$
```

2.Problem Statement :C program to find follow set of the variable in the given productions.

Description : token can look like anything that is useful for processing an input text stream or text

Source Code:

```
#include"stdio.h"
#define max 10
#define MAX 15
char array[max][MAX],temp[max][MAX];
int c,n,t;void fun(int,int[]);
int fun2(int i,int j,int p[],int key)
{
int k;
if(!key)
{
for(k=0;k<n;k++)
if(array[i][j]==array[k][0])
break;
p[0]=i;p[1]=j+1;
fun(k,p);
return 0;
}
else
{
for(k=0;k<=c;k++)
{
if(array[i][j]==temp[t][k])
break;
}
if(k>c)return 1;
```

```
else return 0;
}
}
void fun(int i,int p[])
{
int j,k,key;
for(j=2;array[i][j]!='\0';j++)
{
if(array[i][j-1]=='/')
{
if(array[i][j]>='A'&&array[i][j]<='Z')
{
key=0;
fun2(i,j,p,key);
}
else
{key=1;
if(fun2(i,j,p,key))
temp[t][++c]=array[i][j];
if(array[i][j]=='[at]'&&p[0]!=-1)
{ //taking ,[at], as null symbol.
if(array[p[0]][p[1]]>='A'&&array[p[0]][p[1]]<='Z')
{
key=0;
fun2(p[0],p[1],p,key);
}
else
if(array[p[0]][p[1]]!='/'&&array[p[0]][p[1]]!='\0')
{
if(fun2(p[0],p[1],p,key))
temp[t][++c]=array[p[0]][p[1]];
}
} } } } }
char fol[max][MAX],ff[max];int f,l,ff0;
void ffun(int,int);
void follow(int i)
{
int j,k;
for(j=0;j<=ff0;j++)
if(array[i][0]==ff[j])
return 0;
if(j>ff0)ff[++ff0]=array[i][0];
if(i==0)fol[l][++f]='$';
for(j=0;j<n;j++)
for(k=2;array[j][k]!='\0';k++)
if(array[j][k]==array[i][0])
ffun(j,k);
}
void ffun(int j,int k)
{
int ii,null=0,tt,cc;
if(array[j][k+1]=='/'||array[j][k+1]=='\0')
null=1;
for(ii=k+1;array[j][ii]!='/'&&array[j][ii]!='\0';ii++)
{
if(array[j][ii]<='Z'&&array[j][ii]>='A')
{
for(tt=0;tt<n;tt++)
if(temp[tt][0]==array[j][ii])break;
for(cc=1;temp[tt][cc]!='\0';cc++)
{
if(temp[tt][cc]=='[at]')null=1;
```

```
else fol[l][++f]=temp[tt][cc];
}
}
else fol[l][++f]=array[j][ii];
}
if(null)follow(j);
}
main()
{
int p[2],i,j;

printf("Enter the no. of productions :");
scanf("%d",&n);
printf("Enter the productions :\n");
for(i=0;i<n;i++)
scanf("%s",array[i]);
for(i=0,t=0;i<n;i++,t++)
{
c=0,p[0]=-1,p[1]=-1;
temp[t][0]=array[i][0];
fun(i,p);
temp[t][++c]='\0';
}
/* Follow Finding */
for(i=0,l=0;i<n;i++,l++)
{
f=-1;ff0=-1;
fol[l][++f]=array[i][0];
follow(i);
fol[l][++f]='\0';
}
```

```
for(i=0;i<n;i++)
{
printf("\nFollow[%c] : [ ",fol[i][0]);
for(j=1;fol[i][j]!='\0';j++)
printf("%c,",fol[i][j]);
printf("\b ]");
}
}
```

Input / Output:

```
$ ./a.out
Enter the no. of productions :3
Enter the productions :
A->aBb
C->dBc
B->dCc

Follow[A] : [ $ ]
Follow[C] : [ c ]
Follow[B] : [ b,c ]
$
```

Experiment-IV: Implementation of Recursive decent parser

1.Problem Statement : C program for Recursive descent Parsing.

Description : A recursive descent parser is a static top down parser. For every non terminal Procedure must be implemented to recognize its production body. The first procedure invoked by main function is the procedure for start symbol of the grammar.

Source Code:

```
/*For grammar E->TE', E'->+TE', T->FT', T'->*FT'*/
/*F->(E)/id*/
#include<stdio.h>
#include<stdlib.h>
int i=0;
char a[10];
void e();
void e1();
void t();
void t1();
void f();
int main()
{
		printf("Enter the string:");
		scanf("%s",a);
		e();
		if(a[i]=='$')
				printf("Succesful Parse");
		else
				printf("Unsuccessful Parse");
		return 0;
}
void e(void)
{
		t();
		e1();
}
void e1(void)
{
		if(a[i]=='+')
		{
				i++;
				t();
				e1();
		}
}
```

```
void t(void)
{
        f();
        t1();
}
void t1(void)
{
        if(a[i]=='*')
        {
                i++;
                f();
                t1();
        }
}
void f(void)
{
        if(a[i]=='(')
        {
                i++;
                e();
                if(a[i]==')')
                        i++;
                else
                {
                        printf("')' expected\n");
                        exit(0);
                }
        }
        else if(a[i]=='i')
    i++;
        else
        {
                printf("Invalid Expresion\n");
                exit(0);
        }}
```

Input / Output:

```
Enter the string  i+i$
Successful Parse
ii$
Unsuccessful Parse.
```

Experiment-V: Implementation of LL(1) parser.

1.Problem Statement : C program for LL(1) Parser.

Description : A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1 If X= a=$, the parser halts and announces successful completion of parsing.
2 If X=a!=$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3 If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a]={X->UVW}, the parser replaces X on top of the stack by WVU(with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine.

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
char l[20]={'E','G','G','T','U','U','F','F'};
char r[20][20]={"TG","+TG","@","FU","*FU","@","(E)","i"};
char x[5]={'E','G','T','U','F'};
char y[6]={'i','+','*','(',')','$'};
int tab[5][6],np[10],ra=100,z;
static char temp[10];
char ch[20],t1[10];
int i,j,k=0,m,n,p,h,m1,b=0;
void FIRST(char);
void FOLLOW(char);
int main()
{

   printf("\n\n THE GRAMMER IS...:\n\n");
   for(i=0;i<8;i++)
   {
     printf("%c----->",l[i]);
     for(j=0;j<4;j++)
        printf("%c",r[i][j]);
     printf("\n");
}
for(p=0;p<5;p++)
{
    for(m=0;m<8;m++)
    {   if(x[p]==l[m])
        {
            n=m;
            break;
        }
    }
    k=0;ra=100;z=0;
    FIRST(x[p]);
    for(j=0;j<k;j++)
    {
         for(h=0;h<6;h++)
         {
             if(temp[j]==y[h]&&j<ra)
             {
                  tab[p][h]=n+1;
                  if(r[n][0]==temp[j])
                       tab[p][h]=n+1;
                  else if(r[n+1][0]==temp[j])
                       tab[p][h]=n+2;
             }
             if(temp[j]==y[h]&&j>=ra)
             {
                  tab[p][h]=n+2;
                  if(r[n][0]==temp[j])
                       tab[p][h]=n+2;
                  else if(r[n+1][0]==temp[j])
                       tab[p][h]=n+3;
             }
         }
    }
}
printf("\n");
for(i=0;i<6;i++)
    printf("\t %c",y[i]);
for(i=0;i<5;i++)
{
    printf("\n\t %c",x[i]);
    for(j=0;j<6;j++)
```

```
            printf("%d \t",tab[i][j]);
    }

    return 0;
}
void FOLLOW(char c)
{
    for(i=0;i<8;i++)
    {
       for(j=0;j<8;j++)
       {
          if(c==r[i][j])
          {
             if(r[i][j+1]>='A' && r[i][j+1]<='Z')
                 FIRST(r[i][j+1]);
             else if(r[i][j+1]=='\0')
                 FOLLOW(l[i]);
             else
             {
                 temp[k]=r[i][j+1];
                 k++;
             }
          }
       }
    }
    if(c==l[0])
    {
        temp[k]='$';
        k++;
    }
}
void FIRST(char c)
{
    for(i=0;i<20;i++)
    {
        if(c==l[i])
        {
            if(r[i][0]>='A' && r[i][0]<='Z')
                  FIRST(r[i][0]);
            else if(r[i][0]=='@')
            {
                  if(z==0)
                  {
                      ra=k;
                      z++;
                  }
```

```
                FOLLOW(l[i]);
            }
        }
        else
        {
                temp[k]=r[i][0];
                k++;
        }
    }}
```

Input / Output:

```
 THE GRAMMER IS...:


E----->TG

G----->+TG

G----->@

T----->FU

U----->*FU

U----->@

F----->(E)

F----->i


         i      +      *      (      )      $

         E0     2      1      0      0      0

         G0     0      0      0      3      3

         T0     4      5      0      0      0

         U0     5      0      0      6      6

         F0     7      7      0      0      0
```

Experiment-VI: Implementation of SLR parser.

1.Problem Statement : C program for SLR parser to parse a given string.

Description : A Grammar is said to be SLR(1) if and only if, for any state *s* in the SLR(1) automaton, the following conditions are met:

1. For any item *A* -> *a.Xb* in *s* (*X* is a terminal), there is no complete item *B* -> *y.* in *s* with *X* in the follow set of *B*. Violation of this rule is a **Shift-Reduce Conflict**.
2. For any two complete items *A* -> *a.* and *B* -> *b.* in *s*, *Follow(A)* and *Follow(B)* are disjoint (their intersection is the empty set). Violation of this rule is a **Reduce-Reduce Conflict**.

A grammar is said to be SLR(1) if the following Simple LR parser algorithm results in no ambiguity.

1. If state *s* contains any time of the form *A* -> *a.Xb*, where *X* is a terminal, and *X* is the next token in the input string, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item *A* -> *aX.b*.
2. If state *s* contains the complete item *A* -> *y.*, and the next token in the input string is in *Follow(A)*, then the action is to reduce by the rule *A* -> *y*. A reduction by the rule *S'* -> *S*, where *S* is the start state, is equivalent to acceptance; this will happen only if the next input token is *$*. In all other cases, the new state in computed as follows. Remove the string *y* and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of *y* began. By construction, this state must contain an item of the form *B* -> *a.Ab*. Push *A* on to the stack, and push the state containing the item *B* -> *aA.b*.
3. If the next input token is such that neither of the above two cases apply, an error is declared.

Source Code:

```
int axn[][6][2]= { {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
                    {{-1,-1},{100,6},{-1,-1},{-1,-1},{-1,-1},{102,102}},
                    {{-1,-1},{101,2},{100,7},{-1,-1},{101,2},{101,2}},
                    {{-1,-1},{101,4},{101,4},{-1,-1},{101,4},{101,4}},
                    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
                    {{-1,-1},{101,6},{101,6},{-1,-1},{101,6},{101,6}},
                    {{100,5},{-1,-1},{-1,-1},{-1,-1},{-1,-1},{-1,-1}},
                    {{100,5},{-1,-1},{-1,-1},{100,4},{-1,-1},{-1,-1}},
                    {{-1,-1},{100,6},{-1,-1},{-1,-1},{100,11},{-1,-1}},
                    {{-1,-1},{101,1},{100,7},{-1,-1},{101,1},{101,1}},
                    {{-1,-1},{101,3},{101,3},{-1,-1},{101,3},{101,3}},
                    {{-1,-1},{101,5},{101,5},{-1,-1},{101,5},{101,5}}
                     };//axn Table
int gotot[12][3]={1,2,3,-1,-1,-1,-1,-1,-1,-1,-1,-1,8,2,3,-1,-1,-1,
          -1,9,3,-1,-1,10,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}; //GoTo table
int a[10];char b[10];int top=-1,btop=-1,i;
void push(int k)
{
 if(top<9)
 a[++top]=k;
}
 void pushb(char k)
{
if(btop<9)
b[++btop]=k;
}
 char TOS()
{
return a[top];
}
void pop()
{
if(top>=0)
top--;
}
void popb()
{
if(btop>=0)
b[btop--]='\0';
}
void display()
{
for(i=0;i<=top;i++)
 printf("%d%c",a[i],b[i]);
}
 void display1(char p[],int m) //Displays The Present Input String
{
int l;

for(l=m;p[l]!='\0';l++)
{
   printf("%c",p[l]);
}
  printf("\n");
}
void error()
{
 printf("Syntax Error");
}
 void reduce(int p)
{
int len,k,ad;
char src,*dest;
switch(p)
```

```
 {
case 1:dest="E+T";
             src='E';
             break;
case 2:dest="T";
       src='E';
       break;
case 3:dest="T*F";
       src='T';
       break;
case 4:dest="F";
       src='T';
       break;
case 5:dest="(E)";
       src='F';
       break;
case 6:dest="i";
       src='F';
       break;
default:dest="\0";
            src='\0';
            break;
 }
for(k=0;k<strlen(dest);k++)
 {
 pop();
 popb();
 }
  pushb(src);
   switch(src)
   {
case 'E':ad=0;
             break;
case 'T':ad=1;
             break;
case 'F':ad=2;
             break;
default: ad=-1;
             break;
  }
 push(gotot[TOS()][ad]);}
int main()
{
 int j,st,ic;
 char ip[20]="\0",an;
 printf("Enter any String");
 gets(ip);
 push(0);
 display();
 printf("\t%s\n",ip);
 for(j=0;ip[j]!='\0';)
   {
            st=TOS();
            an=ip[j];
            if(an>='a'&&an<='z')
            ic=0;
            else if(an=='+')
            ic=1;
            else if(an=='*')
            ic=2;
            else if(an=='(')
            ic=3;
            else if(an==')')
            ic=4;
            else if(an=='$')
            ic=5;
            else
            {
              error();
               break;
            }
 if(axn[st][ic][0]==100)
    {
 pushb(an);
 push(axn[st][ic][1]);
 display();
 j++;

 display1(ip,j);
 }
if(axn[st][ic][0]==101)
   {
  reduce(axn[st][ic][1]);
      display();

            display1(ip,j);
    }


 if(axn[st][ic][1]==102)
    {
```

```
   printf("Given String is accepted\n");
     break;

    }

}
          return 0;
          }
```

Input / Output:

```
Enter the string          i+i$
0                         i+i$
0i5                      +i$
0F3                      +i$
0T2                      +i$
0E1                      +i$
0E1+6                     i$
0E1+6i5                   $
0E1+6F3                   $
0E1+6T9                   $
0E1                       $
Given string is accepted.
```

Experiment-VII: Implementation of CLR parser.

1.Problem Statement : program to implement CLR for the following Grammar

S->CC
C->aC/d

Description : token can look like anything that is useful for processing an input text stream or text

Source Code:

```
#include<stdio.h>
#include<string.h>
int i,j,z;
void printtable();
main()
{
char b[5]={'a','d','$','s','c'};
printf("\n------------------clr----------------------\n");
printf("\nthe given grammer\n");
printf("\nS->CC\nC->aC\nC->d\n");
printf("\n the parsing table of the grammer\n");
for(z=0;z<5;z++)
{
printf("\t%c",b[z]);
}
printtable();


}
 void printtable()
{
#include<stdio.h>
#include<string.h>
```

```
int i,j,z;
void printtable();
main()
{
char b[5]={'a','d','$','s','c'};
printf("\n------------------clr----------------------\n");
printf("\nthe given grammer\n");
printf("\nS->CC\nC->aC\nC->d\n");
printf("\n the parsing table of the grammer\n");
for(z=0;z<5;z++)
{
printf("\t%c",b[z]);
}
printtable();

}
 void printtable()
{
for(i=0;i<=9;i++)
{
printf("\n---------------------------------------------------\n");
printf("%d",i);
for(j=0;j<=4;j++)
{

 if(t[i][j]==-5)
printf("\t");
else if(t[i][j]>=1&&t[i][j]<10)
printf("\t%d",t[i][j]);
else if(t[i][j]==99)
printf("\taccepted\t");
//else if(t[i][j]%2==0)
else if(t[i][j]%2==0)
printf("\tr%d",t[i][j]/10);
else if(t[i][j]%2==1)
{
printf("\ts%d",t[i][j]/10);
continue;
}
/*else
printf("\t");   */

}
}
}
```

Input / Output:

```
$ gcc clr.c
$ ./a.out

------------------clr----------------------


the given grammer


S->CC
C->aC
C->d
 the parsing table of the grammer
        a       d       $       s       c
---------------------------------------------------
0       s3      s4              1       2
---------------------------------------------------
1                       accepted
```

```
--------------------------------------------------------
2      s6      s7                  5
--------------------------------------------------------
3      s3      s4                  8
--------------------------------------------------------
4      r3            r3
--------------------------------------------------------
5      s6            r1
--------------------------------------------------------
6                               9
--------------------------------------------------------
7      r2      s7      r3
--------------------------------------------------------
8                   r2
--------------------------------------------------------
9                   r2


$
```

Experiment-VIII: Implementation of LALR Parser using ANTLR

1.Problem Statement : Implementation of LALR Parser using ANTLR

Description : ANTLR(Another Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

Steps to parse a grammer using antlr tool:

1. Create a folder in c drive called antlr.
2. Copy grammar file (for ex: Calculator.g4 file) , antlr-4.5-complete.jar file and two batch files (antlr4.bat and grun.bat) files in to c:\antlr folder.
3. go to command prompt.
4. set classpath=.;C:\antlr\antlr-4.5-complete.jar.
5. compile a grammer file for ex take Calculator grammer file

4. antlr4 Calculator.g4 // on success java files will be generated.

5. javac *.java //compile java files.then

6. grun Calculator calculator –gui.

7. ex: 3*(4+5) enter and press ctrl+z.

8. Finally a parse tree inspector generates for the given grammer.

9. Click on save png then a png image file will be created for the given grammer.

Source Code:

Calculator.g4:

```
grammar Calculator;
calculator : expression;
expression
  : expression operator=('*'|'/') expression  # MultiplyDivide
  | expression operator=('+'|'-') expression  # AddSubtract
  | '-' expression                            # Negate
  | Number                                    # Number
  | '(' expression ')'                        # Parenthesis
  ;
Number : DIGIT+ '.' DIGIT*
  | '.' DIGIT+
  | DIGIT+
  ;



DIGIT: ('0'..'9');
WS: [ \t\r\n]+ -> skip;
```

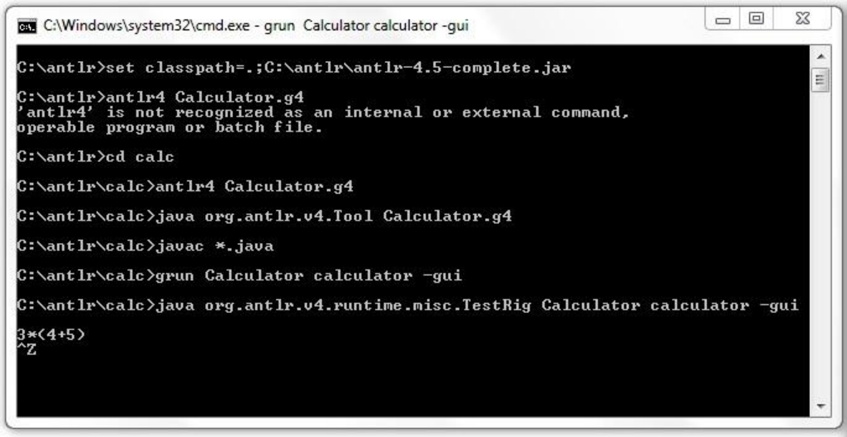
antlr4.bat:

```
java org.antlr.v4.Tool %*
```

grun.bat:

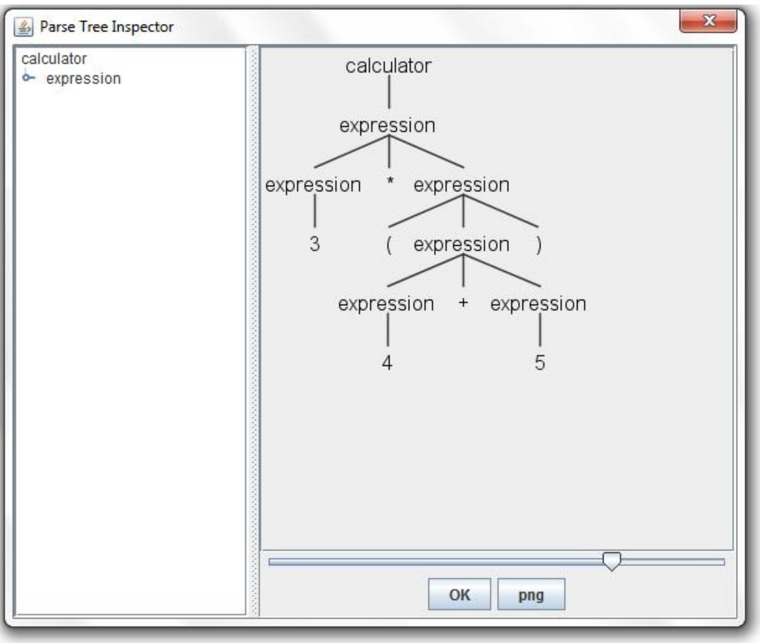
```
java org.antlr.v4.runtime.misc.TestRig %*.
```

Input / Output::



C:\Windows\system32\cmd.exe - grun Calculator calculator -gui

```
C:\antlr>set classpath=.;C:\antlr\antlr-4.5-complete.jar

C:\antlr>antlr4 Calculator.g4
'antlr4' is not recognized as an internal or external command,
operable program or batch file.

C:\antlr>cd calc

C:\antlr\calc>antlr4 Calculator.g4

C:\antlr\calc>java org.antlr.v4.Tool Calculator.g4

C:\antlr\calc>javac *.java

C:\antlr\calc>grun Calculator calculator -gui

C:\antlr\calc>java org.antlr.v4.runtime.misc.TestRig Calculator calculator -gui

3*(4+5)
^Z
```



Experiment-IX: Construct dependency graph for the given SDD

1.Problem Statement : YACC program to Construct dependency graph for the given SDD

Description : Dependency graphs tool for determining an evaluation order for the attributes instances in a given parse tree.

Annotated parse tree shows the values of attributes, a dependency graph helps to determine how those values can be computed.

Dependency graphs

1. Edges express constraints implied by the semantic rules.
2. Each attribute is associated to a node
3. If a semantic rule associated with a production p defines the value of synthesized attribute A.b in terms of the value of X.c, then graph has an edge from X.c to A.b

If a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of value of X.a, then graph has an edge from X.a to B.c

Source Code:

dep.l

```
%{
#include "y.tab.h"
#include<math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {yylval.dval=atof(yytext);
return NUMBER;
}
log|LOG {return LOG;}
sin|SIN {return SIN;}
cos|COS {return COS;}
[\t];
\$; {return 0;}
\n|. {return yytext[0];}
%%
```

dep.y

```
%{
#include<math.h>
#include<stdio.h>
int sno=1;
%}
%union{
double dval;
}
%type<dval>Expr
%type<dval> T
%type<dval> F
%token <dval>NUMBER
%token LOG,SIN,COS
%left '+','-'
%left '*','/'
%left SIN COS
%%
S :Expr'\n' {printf("%g \n",$1); printf("%d . E.val = %g\n",sno,$1); sno++;}
;
Expr:Expr '+' T {$$=$1+$3; printf("%d . E.val = %g\n",sno,$$); sno++;}
   |T     {printf("%d . E.val = %g\n",sno,$$); sno++;}
;
T: T '*' F {$$=$1*$3; printf("%d . T.val = %g\n",sno,$$); sno++;}
  |F       {printf("%d . T.val = %g\n",sno,$$); sno++;}
;
F :  NUMBER {$$=$1;  printf("%d . F.val = %g\n",sno,$$); sno++; }
  ;
%%
```

```
int main()
{
yyparse();
return 0;
}
int yywrap()
{
return 1;
}
int yyerror(char *err)
{
printf("%s",err);
}
```

Input / Output:

```
$ lex dep.l
$ yacc -d dep.y
$ gcc lex.yy.c y.tab.c -ll -lm
$ ./a.out
6+8*7
1 . F.val = 6
2 . T.val = 6
3 . E.val = 6
4 . F.val = 8
5 . T.val = 8
6 . F.val = 7
7 . T.val = 56
8 . E.val = 62
```

Experiment-X: Intermediate Code generation using YACC

1.Problem Statement : C program for Three address code generation

Description : Three-address code is a sequence of statements of the general form

X:= Op Z

Where x, y, and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

Source Code:

three.l

```
%{
#include "y.tab.h"
extern char yyval;
%}
number [0-9]+
letter [a-zA-Z]+
%%
{number} {yylval.sym=(char)yytext[0];return number;}
{letter} {yylval.sym=(char)yytext[0]; return letter; }
\n {return 0;}
.  {return yytext[0];}
%%
```

Three.y

```
%{
#include<stdio.h>
#include<string.h>
int nIndex=0;
```

```
struct Intercode
{
char operand1;
char operand2;
char opera;
};
%}
%union
{
char sym;
}
%token <sym> letter number
%type  <sym> expr
%left '-' '+'
%right '*' '/'
%%
statement: letter '=' expr ';' { addtotable((char)$1,(char)$3,'=' ); }
              | expr ;
              ;
expr: expr '+' expr  { $$=addtotable((char)$1,(char)$3,'+');}
         | expr '-'  expr { $$=addtotable((char)$1,(char)$3,'-');}
         | expr '*'  expr { $$=addtotable((char)$1,(char)$3, '*');}
         | expr '/'  expr { $$=addtotable((char)$1,(char)$3,'/');}
         | '(' expr ')' { $$= (char)$2;}
         |  number  { $$= (char)$1;}
         | letter { $$= (char)$1;}
%%
yyerror(char *s)
{
printf("%s",s);
exit (0);
```

```
}
struct Intercode code[20];
char addtotable(char operand1, char operand2,char opera)
{
char temp = 'A';
code[nIndex].operand1 = operand1;
code[nIndex].operand2 = operand2;
code[nIndex].opera = opera;
nIndex++;
temp++;
return temp;
}
threeaddresscode()
{
int nCnt=0;
char temp='A';
printf("\n\n\t three addrtess codes\n\n");
temp++;
while(nCnt<nIndex)
{
printf("%c:=\t",temp);
if (isalpha(code[nCnt].operand1))
printf("%c\t", code[nCnt].operand1);
else
printf("%c\t",temp);
printf("%c\t", code[nCnt].opera);
if (isalpha(code[nCnt].operand2))
printf("%c\t", code[nCnt].operand2);
else
printf("%c\t",temp);
printf("\n");
nCnt++;
temp++;
}}
main()
{
printf("enter expression");
yyparse();
threeaddresscode();
}
yywrap()
{
return 1;
}
```

Input / Output:

```
$ lex three.l
$ yacc -d three.y
$ gcc lex.yy.c y.tab.c -ll -lm
$ ./a.out
enter expression (a*b)+(c*d)
        three addrtess codes

B:=    a     *     b
C:=    c     *     d
D:=    B     +     B
```

2.Problem Statement :YACC program for generate Quadruple.

Description : A quadruple is a record structure with four fields, which we call op, arg l, arg 2, and result. The op field contains an internal code for the operator. The three-address statement x:= y op z is represented by placing y in arg 1. z in arg 2. and x in result. Statements with unary operators like x: = – y or x: = y do not use arg 2. Operators like param use neither arg2 nor result. Conditional and unconditional jumps put the target label in result.

Source Code:

/* lex program */

```
%{
#include "y.tab.h"
extern char yyval;
%}
number [0-9]+
letter [a-zA-Z]+
%%
{number} {yylval.sym=(char)yytext[0];return number;}
{letter} {yylval.sym=(char)yytext[0]; return letter; }
\n {return 0;}
.  {return yytext[0];}
%%
```

/*yaac program */

```
%{
#include<stdio.h>
#include<string.h>
int nIndex=0;
struct Intercode
{
char operand1;
char operand2;
char opera;
};
%}
%union
{
char sym;
}
```

```
%token <sym> letter number
%type  <sym> expr
%left '-' '+'
%right '*' '/'
%%
Statement: letter '=' expr ';' { addtotable((char)$1,(char)$3,'=' ); }
             | expr ;
             ;
expr: expr '+' expr  { $$=addtotable((char)$1,(char)$3,'+');}
        | expr '-'  expr { $$=addtotable((char)$1,(char)$3,'-');}
        | expr '*'  expr { $$=addtotable((char)$1,(char)$3, '*');}
        | expr '/'  expr { $$=addtotable((char)$1,(char)$3,'/');}
        | '(' expr ')' { $$= (char)$2;}
        |  number  { $$= (char)$1;}
        | letter { $$= (char)$1;}
%%

yyerror(char *s)
{
printf("%s",s);
exit (0);
}
struct Intercode code[20];
char addtotable(char operand1, char operand2,char opera)
{
char temp = 'A';
code[nIndex].operand1 = operand1;
code[nIndex].operand2 = operand2;
code[nIndex].opera = opera;
nIndex++;
temp++;
return temp;
}
threeaddresscode()
{
int nCnt=0;
char temp='A';
printf("\n\n\t three addrtess codes\n\n");
temp++;
while(nCnt<nIndex)
{
printf("%c:=\t",temp);
if (isalpha(code[nCnt].operand1))
printf("%c\t", code[nCnt].operand1);
else
printf("%c\t",temp);
printf("%c\t", code[nCnt].opera);
if (isalpha(code[nCnt].operand2))
printf("%c\t", code[nCnt].operand2);
else
printf("%c\t",temp);
printf("\n");
nCnt++;
temp++;
}
}
void quadruples()
{
int nCnt=0;
char temp = 'A';
temp++;
printf("\n\n\t Quardruples \n");
printf("\n ID OPERATOR OPERAND1 OPERAND2\n");
while(nCnt<nIndex)
{
printf("\n (%d) \t %c \t",nCnt,code[nCnt].opera);
if(isalpha(code[nCnt].operand1))
printf("%c\t", code[nCnt].operand1);
else
printf("%c\t",temp);
printf("%c\t", code[nCnt].opera);
if(isalpha(code[nCnt].operand2))
printf("%c\t", code[nCnt].operand2);
else
printf("%c\t",temp);
printf("%c\t",temp);
printf("\n");
nCnt++;
temp++;
}
}
main()
{
printf("enter expression");
yyparse();
threeaddresscode();
quadruples();
}
yywrap()
{
return 1;
}
```

Input / Output:

```
$ lex lexfile.l
    $ yacc –d  yaccfile.y
    $cc lex.yy.c y.tab.c –ll –ly –lm
    $./a.out
    Enter expr a+b*c+d
    Three address code
    B=b*c
    C=a+B
    D=B+d

    ID    OPERATOR       OPERAND1     OPERAND2
    (0)       *            b                  c           B
    (1)       +            a                  B           C
    (2)       +            B                  d           D
```

Experiment-III: Construct the DAG for given three address code.

1.Problem Statement :C program for Directed Acyclic Graph

Description : A directed acyclic graph (DAG) is a directed graph that contains no cycles. a DAG gives a picture of how the value computed by a statement in a basic block is used in subsequent statements of the block.

Constructing a DAG from three-address statements is a good way of determining common sub-expressions (expressions computed more than once) within a block, determining which names are used inside the block but evaluated outside the block, and determining which statements of the block could have their computed value used outside the block.

i. A DAG for a basic block has following labels on the nodes

j. Leaves are labeled by unique identifiers, either variable names or constants.

k. Interior nodes are labeled by an operator symbol.

Nodes are also optionally given a sequence of identifiers for labels.

Source Code:

dag.c

```
#include<stdio.h>
#include<ctype.h>
#define size 20
typedef struct node{
char data;
struct node *left;
struct node *right;
}btree;
btree *stack[size];
int top;
main() {
btree *root;
char exp[80];
```

```
btree *create(char exp[80]);
void dag(btree *root);
printf("\nEnter the postfix expression:\n");
scanf("%s",exp);
top=-1;
root=create(exp);
printf("\nThe tree is created.....\n");
printf("\nInorder DAG is : \n\n");
dag(root);
return 0; }
btree *create(char exp[]) {
btree *temp;
int pos;
char ch;
void push(btree*);
btree *pop();
pos=0;
ch=exp[pos];
printf("%c\t",ch);
while(ch!='\0') {
temp=((btree*)malloc(sizeof(btree)));
temp->left=temp->right=NULL;
temp->data=ch;
printf("%c",temp->data);
if(isalpha(ch))
push(temp);
else if(ch=='+' ||ch=='-' || ch=='*' || ch=='/') {
temp->right=pop();
temp->left=pop();
push(temp); }
else
printf("\n Invalid char Expression\n");
pos++;
ch=exp[pos]; }
temp=pop();
return(temp); }
void push(btree *Node)
{
if(top+1 >=size)
printf("Error:Stack is full\n");
top++;
stack[top]=Node;
}
btree* pop()
{
btree *Node;
if(top==-1)
printf("\nerror: stack is empty..\n");
Node=stack[top];
top--;
return(Node);
}
void dag(btree *root)
{
btree *temp;
temp=root;
if(temp!=NULL)
{
dag(temp->left);
printf("%c",temp->data);
dag(temp->right);
}}
```

Input / Output:

```
$ gcc dag.c
$ ./a.out

Enter the postfix expression:
abcd+*-
a       abcd+*-
The tree is created.....
Inorder DAG is :
a-b*c+d
```

2.Problem Statement : Write a C Program to implement a code optimization method "common sub expression elimination".

Description : An "optimization" must not change the output produced by a program for a given input, or cause an error, such as a division by zero, that was not present in the original version of the source program, speed up programs by a measurable amount, and it must be worth the effort. On of the Optimization technique is eliminate Common sub expressions. Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where its referenced when encountered again – of course providing the variable values in the expression still remain constant.

Source Code:

```
#include<stdio.h>

int tc[10],fb=0,i=0,j=0,k=0,p=0,fstar=0,c=-1,c1=0,c2=0,t1,t2,t3,t4,fo=0;
char m[30],temp[30],opt[10][4];
 main(){
int a,d;
for(i=0;i<10;i++)
tc[i]=-1;
```

```
printf("\n Code stmt evaluation follow following precedence: ");
printf("\n   1.( ) within the () stmt should be of the form:  x op z");
printf("\n   2.*,/ equal precedence");
printf("\n   3.+,- equal precedence");
printf("\n Enter ur Code Stmt-");
gets(m);
i=0;
while(m[i]!='\0'){
            if(m[i++]=='('){
            fb++;
            break;
            }
            }
i=0;
printf("\nThe Intermediate Code may generated as-");
if(fb==1){                                      /* evaluating sub exp */
            while(m[i]!='\0')
                        if(m[i]=='('){
                        temp[j++]='T';
                        i++;
                        t3=i;                   /* optimising the code */
                while(m[i]!=')')
                        opt[c1][c2++]=m[i++];
                        for(t4=c1-1;t4>=0;t4--)
                        if(strcmp(opt[c1],opt[t4])==0){
                        tc[p++]=t4;
                        fo=1;
                        }                       /* end of optimising   */
                        if(fo==0){
                        tc[p++]=k++;
                printf("\nT%d=",k-1);
```

```
                    while(m[t3]!=')')
                    printf("%c",m[t3++]);
                    }
                    i++;
                    c1++;
                    c2=fo=0;
                    }
                    else if(m[i]!='(')
                    temp[j++]=m[i++];
            if(fb==1){
        temp[j]='\0';
            for(i=0;temp[i]!='\0';i++)
            m[i]=temp[i];
        m[i]='\0';
            }

}                           /* end of evluating sub exp */
a=operatormajid('*','/');     /* operator fun call depends on priority */
d=operatormajid('+','-');
if(a==0&&d==0&&m[1]=='=')
            printf("\n%s%d",m,k-1);

getch();
}
/*  ****************************     */
operatormajid(char haj,char haj1){     /* function to evaluate operators */
m1:     for(i=0;m[i]!='\0';i++)
            if(m[i]==haj||m[i]==haj1){
            fstar++;
            break;
            }
            if(fstar==1){
            for(j=0;j<i;j++)
            if(m[j]=='T')c++;
            printf("\nT%d=",k);
            if(m[i-1]=='T'&&m[i+1]=='T'){
            printf("%c%d%c%c%d",m[i-1],tc[c],m[i],m[i+1],tc[c+1]);
         tc[c]=k++;
            for(t2=c+1;t2<9;t2++)
            tc[t2]=tc[t2+1];
            }
            else if(m[i-1]!='T'&&m[i+1]!='T'){
            printf("%c%c%c",m[i-1],m[i],m[i+1]);
            if(c==-1){
            for(t1=9;t1>0;t1--)
            tc[t1]=tc[t1-1];
            tc[0]=k++;
            }
            else if(c>=0){
            for(t1=9;t1>c+1;t1--)
            tc[t1]=tc[t1-1];
            tc[t1]=k++;
}
            }
            else if(m[i-1]=='T'&&m[i+1]!='T'){
            printf("%c%d%c%c",m[i-1],tc[c],m[i],m[i+1]);
            tc[c]=k++;
            }
            else if(m[i-1]!='T'&&m[i+1]=='T'){
            printf("%c%c%c%d",m[i-1],m[i],m[i+1],tc[c+1]);
            tc[c+1]=k++;
            }
```

```
    for(t1=0;t1<i-1;t1++)
    temp[t1]=m[t1];
    temp[t1++]='T';
    for(t2=i+2;m[t2]!='\0';t2++)
    temp[t1++]=m[t2];
    temp[t1++]='\0';

    fstar=0;
  for(i=0;temp[i]!='\0';i++)
    m[i]=temp[i];
    m[i]='\0';
    c=-1;
    goto m1;
    }
    else return 0;
     }
```

INPUT/OUTPUT:

```
Code stmt evaluation follow following precedence:
 1.( ) within the () stmt should be of the form:  x op z
 2.*,/ equal precedence
 3.+,- equal precedence
 Enter ur Code Stmt

       a+(b*c)-d/(b*c)

  T0=b*C
  T1= d/T1
  T2=a+T0
  T3=T2-T1
```

Experiment-XII: Build a Tiny compiler for the C language using LEX and YAAC

1.Problem Statement : Parser generator using YACC (Calculator)

Description : generates a parser (the part of a compiler that tries to make syntactic sense of the source code) based on an analytic grammar written in a notation similar to BNF. Yacc generates the code for the parser in the C programming language.

The parser generated by yacc requires a lexical analyzer.

Source Code:

Calculator.l

```
%{
#include "y.tab.h"
#include<math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {yylval.dval=atof(yytext);
return NUMBER;
}
log|LOG {return LOG;}
ln {return nLOG;}
sin|SIN {return SINE;}
cos|COS {return COS;}
tan|TAN {return TAN;}
mem {return MEM;}
[\t];
\$; {return 0;}
\n|. {return yytext[0];}
%%
```

Calculator.y

```
%{
#include<stdio.h>
#include<math.h>
double memvar;
%}
%union
{
double dval;
}
%token<dval>NUMBER
%token<dval>MEM
%token LOG SINE nLOG COS TAN
%left '-''+'
```

```
%left '*''/'
%right '^'
%left LOG SINE nLOG COS TAN
%nonassoc UMINUS
%type<dval> expression
%%
start: statement '\n'
|start statement '\n'
;
statement: MEM'='expression { memvar=$3;}
|expression {printf("answer=%g\n",$1);}
;
expression:expression'+'expression {$$=$1+$3;}
|expression'-'expression {$$=$1+$3;}
|expression'*'expression {$$=$1*$3;}
|expression'/'expression
{
if($3==0)
yyerror("divide by zero");
else
$$=$1/$3;}
|expression'^'expression {$$=pow($1,$3);}
;
expression: '-' expression %prec UMINUS {$$=-$2;}
|'('expression')' {$$=$2;}
|LOG expression {$$=log($2)/log(10);}
|nLOG expression {$$=log($2);}
|SINE expression {$$=sin($2*3.14159/180);}
|COS expression {$$=cos($2*3.14159/180);}
|TAN expression {$$=tan($2*3.14159/180);}
|NUMBER { $$ = $1;}
|MEM {$$=memvar;}
;
%%
main()
{
printf("enter expression:");
yyparse();
}
int yyerror(char *error)
{
fprintf(stderr,"%s\n",error);
}
yywrap() {  return 1; }
```

Input / Output:

```
$ lex lex.l
$ yacc –d yacc.y
$cc lex.yy.c y.tab.c –ll –ly –lm
$./a.out
2+3
Answer=5
SIN 90
Answer=1
```