1. Implement a YACC specification for simple arithmetic calculations.

**yacc1.y**

```
%{
 #include<stdio.h
 >
 #include<ctype.h
 >
 %}
%token NUM
%%
cmd:E {printf("%d\n",$1);}
E: E'+'T {$$ = $1 + $3;}
    |T {$$ = $1;}
;
 E: E'-'T {$$ = $1 - $3;}
 ;
 T: T'*'F {$$ = $1 * $3;}
| F {$$ = $1;}
 ;
 T: T'/'F {$$ = $1 / $3;}
 ;
 F: '('E')' {$$ = $2;}
|  NUM {$$ = $1;}
 ;%%
  int yyerror(char *s)
 {printf("%s \n", s); return
 0;}int main(void)
```

{ yyparse();return 0;}

**yacclex1.l**

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

```
"yacclex1.l" 15L, 153C written
[cse19022@CCLINUXSERVER cclab4]$ yacc -d yacc1.y
[cse19022@CCLINUXSERVER cclab4]$ lex yacclex1.l
[cse19022@CCLINUXSERVER cclab4]$ cc lex.yy.c y.tab.c
[cse19022@CCLINUXSERVER cclab4]$ ./a.out
2+3
5
```

## 2. Implement the Three address code using YACC.

**3add.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=operand
    2; code[nIndex].opera=opera;
    nIndex++; temp++; return
    temp;}
 threeadresscode()
{
  int nCnt=0; char temp='A';
printf("\n\n\t three adress
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();
threeadresscode();}
yywrap()
{
return 1;
}
```

**3addlex.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];}
%%
```

```
"3add.y" 80L, 1408C written
[cse19022@CCLINUXSERVER cclab4]$ yacc -d 3add.y
[cse19022@CCLINUXSERVER cclab4]$ lex 3addlex.l
[cse19022@CCLINUXSERVER cclab4]$ cc lex.yy.c y.tab.c
[cse19022@CCLINUXSERVER cclab4]$ ./a.out
enter expression(a+b)*(c+d)

        three adress codes

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

## 3. Implement the Three address code in form of quadruples.

```
#include<iostream>
#include<string>
using namespace std;
string inttostring(int n){
	string ans="";
	while(n){
		ans+=char('0'+n%10);
		n/=10;
	}
	string t="";
	for(int i=ans.size()-1;i>-1;i--){
		t+=ans[i];
	}
	return t;
}
int main(){
	int a=0,b=0,c=0,d=0,e=0,f=0;
    cout<<"\nThe set of expression: a+b+c*d/e+f\n"<<endl;
	cout<<"Enter value of a,b,c,d,e and f for above expression:";
	cout<<"\na:";
	cin>>a;
	cout<<"b:";
	cin>>b;
	cout<<"c:";
	cin>>c;
	cout<<"d:";
	cin>>d;
```

```
        cout<<"e:";
        cin>>e;
        cout<<"f:";
        cin>>f;
        int res1=c*d,res3=a+b;
        int res2=res1/e;
        int res4=res3+res2;
        int res5=res4+f,ans=0;
        ans=res5;
    cout<<"\nQuadraple format representation of given expression is:\n";
        cout<<"\nOperator\tArg1\tArg2\tResult"<<endl;
        string result[6][4];

        for(int i=0;i<6;i++){
            for(int j=0;j<4;j++){
                result[i][j]="",result[0][0]="*";
                                result[0][1]=inttostring(c),result[0][2]=inttostring(d);
                                result[0][3]=inttostring(c*d),result[1][0]="/";
                                result[1][1]=inttostring(res1),result[1][2]=inttostring(e);
result[1][3]=inttostring(res1/e),result[2][0]="+",result[2][1]=inttostring(a);
                                result[2][2]=inttostring(b),result[2][3]=inttostring(a+b);
                                result[3][0]="+",result[3][1]=inttostring(res3);
result[3][2]=inttostring(res2),result[3][3]=inttostring(res3+res2),result[4][0]="+";
                result[4][1]=inttostring(res4),result[4][2]=inttostring(f);
                result[4][3]=inttostring(res4+f),result[5][0]="=";
                result[5][1]=inttostring(res5),result[5][2]="";
                result[5][3]=inttostring(res5);
```

```
        }
    }
    int answer=res5;
        for(int i=0;i<6;i++){
            cout<<"   "<<result[i][0]<<"      \t "<<result[i][1]<<" \t "<<result[i][2]<<" \t "<<result[i][3]<<endl;
        }
        cout<<"\nResult of above expression  is: "<<answer<<endl;
}
```

```
The set of expression: a+b+c*d/e+f

Enter value of a,b,c,d,e and f for above expression:
a:2
b:3
c:2
d:4
e:7
f:1

Quadraple format representation of given expression is:

Operator        Arg1    Arg2    Result
   *             2       4       8
   /             8       7       1
   +             2       3       5
   +             5       1       6
   +             6       1       7
   =             7               7

Result of above expression  is: 7


...Program finished with exit code 0
Press ENTER to exit console.
```

# 4. Implement the Dependency graph using YACC.

## sdd.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);
 }
```

## Sdd.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];}

%%

```
"sdd.y" 63L, 738C written
[cse19022@LINUXSERVER cc]$ lex sdd.l
[cse19022@LINUXSERVER cc]$ yacc -d sdd.y
[cse19022@LINUXSERVER cc]$ gcc lex.yy.c y.tab.c
[cse19022@LINUXSERVER cc]$ ./a.out
5+9*7
1 . F.val = 5
2 . T.val = 5
3 . E.val = 5
4 . F.val = 9
5 . T.val = 9
6 . F.val = 7
7 . T.val = 63
8 . E.val = 68
68
9 . E.val = 68
```

## 5. Implement the DAG using YACC.

File: C9.h

```
#include<string>
using namespace std;
// Node structure of a node of the DAG.
struct node
{
int number;
node *left, *right;
bool printed;
const char *value;
};
```

File: C9.y

```
%{
#include<iostream>
#include<vector>
#include<string.h>
#include"C9.h"
using namespace std;
vector<node*> nodelist;
extern int yylex();
int node_count = 0;
void yyerror(const char *str)
{
cerr<<"error : "<<str<<endl;
}
// This function creates a node with the value passed as input with
its left and right children as the nodes passed as parameters.
// After that the node is added into the nodelist vector.
node* make_node(node *left, const char *value, node *right)
{
node *n = new node;
int size = nodelist.size();
for(int i = 0; i < size; ++i)
{
node *x = nodelist[i];
if(strcmp(x->value, value) == 0 && x->left == left && x-
>right == right)
{
```

```
return x;
}
}
n->left = left;
n->value = value;
n->right = right;
n->number = node_count++;
n->printed = false;
nodelist.push_back(n);
return n;
}
// This function is used to print the DAG tree recursively.
void print_tree(node *n)
{
if(!n || (n->printed))
{
return;
}
n->printed = true;
cout<<"Node : "<<n->number<<" value : "<<n->value<<flush;
if(n->left)
{
cout<<" left child at : "<<n->left->number<<flush;
}
if(n->right)
{
cout<<" right child at : "<<n->right->number<<flush;
}
```

```
cout<<endl;
print_tree(n->left);
print_tree(n->right);
}
%}
/* Start statet is S. */
%start S
%union
{
char *text;
node *n;
}
/* init the different tokens that will be used. */
%token <text> NUMBER
%token ADD SUB MUL DIV POW OPEN CLOSE
%type <n> S E T P F
%%
/* Start parsing the tree. And print at the end. */
S:
E
3
{
print_tree($$);
}
;
/* If an ADD or SUB are encountered, split it into two halves and
create the nodes of the DAG tree for the operator. */
/* Lowest precedence to ADD and SUB. */
```

```
/* Check for other operators with higher precedence in the expression
using T and for ADD or SUB again using E. */
/* If ADD or SUB isnt't there, going to operators with higher
precedence. */
E:
E ADD T
{
$$ = make_node($1, "+", $3);
}
|
E SUB T
{
$$ = make_node($1, "-", $3);
}
|
T
{
$$ = $1;
}
;
/* If an MUL or DIV are encountered, split it into two halves and
create the nodes of the DAG tree for the operator. */
/* Second lowest precedence to MUL and DIV. */
/* Check for other operators with higher precedence in the expression
using P and for MUL or DIV again using T. */
/* If MUL or DIV isnt't there, going to operators with higher
precedence. */
T:
```

```
T MUL P
{
$$ = make_node($1, "*", $3);
}
|
T DIV P
{
$$ = make_node($1, "/", $3);
}
|
P
{
$$ = $1;
}
;
4
/* If an POW is encountered, split it into two halves and create the
nodes of the DAG tree for the operator. */
/* Second highest precedence to POW */
/* Check for other operators with higher precedence in the expression
using F and for POW again using P. */
/* If POW isnt't there, going to operator with higher precedence. */
P:
F POW P
{
$$ = make_node($1, "^", $3);
}
|
```

```
F
{
$$ = $1;
}
;
/* If an OPWN and CLOSE are encountered, recursively call state E for
parsing the expression inside the brackets. */
/* Highest precedence to OPEN and CLOSE. */
/* If OPEN and CLOSE isnt't there, make node for the number with no
children. */
F:
OPEN E CLOSE
{
$$ = $2;
}
|
NUMBER
{
$$ = make_node(NULL, $1, NULL);
}
;
%%
int main()
{
cout<<"Enter an expression\n";
 yyparse();
return 0;
}
```

5

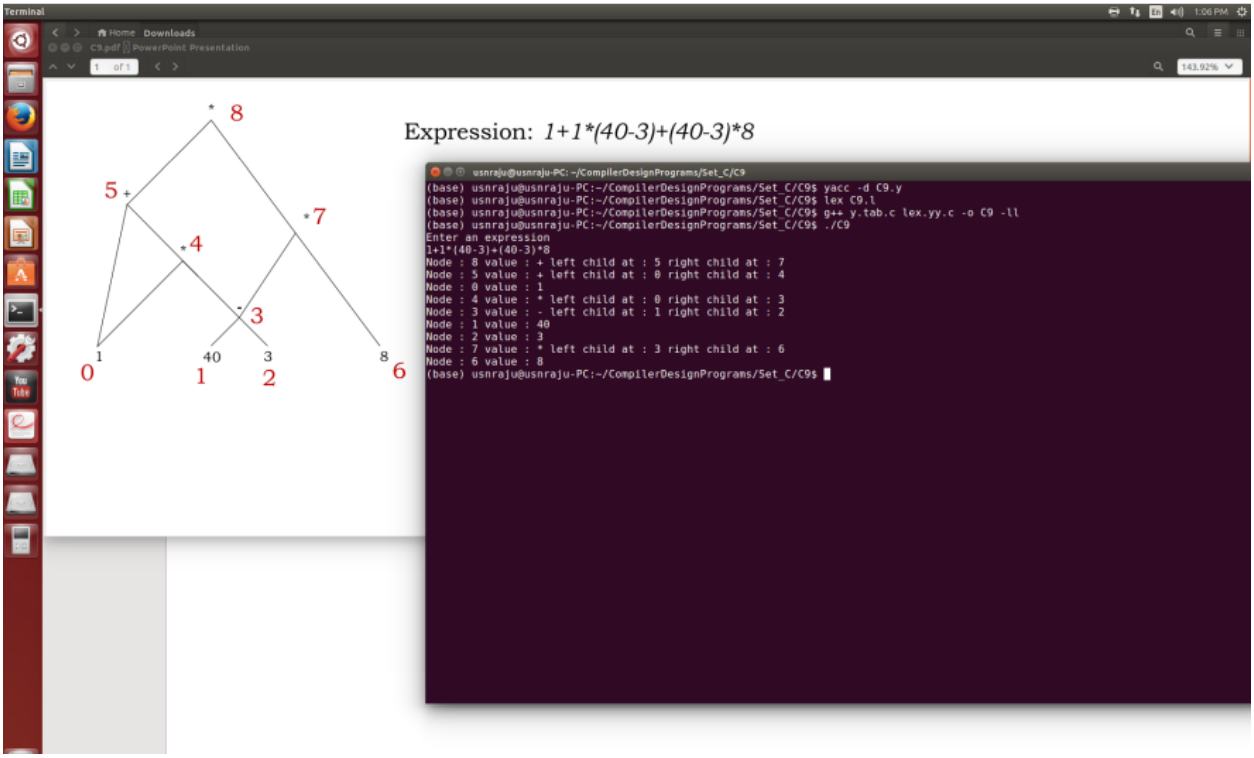
File: C9.l

```
%{
#include<string.h>
#include"C9.h"
#include"y.tab.h"
using namespace std;
%}
/*
 Rules:
 If any number is matched, the number is sent as the token.
 If a '+' is matched, send ADD as token.
 If a '-' is matched, send SUB as token.
 If a '*' is matched, send MUL as token.
 If a '\' is matched, send DIV as token.
 If a '^' is matched, send POW as token.
 If a '(' is matched, send OPEN as token.
 If a ')' is matched, send CLOSE as token.
 If a space, tab or new line character is matched, end the
program.
*/
%%
[0-9]+ { yylval.text = strdup(yytext); return NUMBER; }
\+ { return ADD; }
\- { return SUB; }
\* { return MUL; }
\\ { return DIV; }
\^ { return POW; }
```

\( { return OPEN; }

\) { return CLOSE; }

[ \n\t] { return 0;




6. Implement the SLR (1) parsing table for the given grammar

E -> E+T | T

T-> T*F | T

F->id| (E)

```
# pip install firfol==0.2.1

from collections import deque
from collections import OrderedDict
from pprint import pprint
from firfol import makeGrammar, findFirsts, findFollows
```

```
rules   = ['S->AA', 'A->aA|b']
start   = 'S'
aug     = ''
nt_list = ['S', 'A']
t_list  = ['a', 'b', '$']
g       = makeGrammar(rules)
firsts  = findFirsts(g)
follows = findFollows(g, start)

class State:
    _id=0
    def __init__(self, closure):
        self.closure=closure
        self.no=State._id
        State._id+=1

class Item(str):
    def __new__(cls, item):
        self=str.__new__(cls, item)
        return self
    def __str__(self):
        return super(Item, self).__str__()

def closure(items):
    def exists(newitem, items):
        for i in items:
            if i==newitem:
                return True
        return False

    global g
    while True:
        flag=0
        for i in items:
            if i.index('.')==len(i)-1: continue
            Y=i.split('->')[1].split('.')[1][0]
            if i.index('.')+1<len(i)-1:
                lastr=list(firsts[i[i.index('.')+2]]-set(chr(1013)))
            for prod in g.keys():
```

```
                head, body=prod, g[prod]
                if head!=Y: continue
                for b in body:
                    newitem=Item(Y+'->.'+b)
                    if not exists(newitem, items):
                        items.append(newitem)
                        flag=1
        if flag==0: break
    return items

def goto(items, symbol):
    initial=[]
    for i in items:
        if i.index('.')==len(i)-1: continue
        head, body=i.split('->')
        seen, unseen=body.split('.')
        if unseen[0]==symbol and len(unseen) >= 1:
            initial.append(Item(head+'->'+seen+unseen[0]+'.'+unseen[1:]))
    return closure(initial)

def calc_states():
    def contains(states, t):
        for s in states:
            if len(s) != len(t): continue
            if sorted(s)==sorted(t):
                for i in range(len(s)):
                    if s[i]!=t[i]: break
                else: return True
        return False
    global g, nt_list, t_list, aug

    head, body=aug, g[aug]
    for b in body:
        states=[closure([Item(head+'->.'+b)])]
    while True:
        flag=0
        for s in states:
            for e in nt_list+t_list:
                t=goto(s, e)
```

```
                if t == [] or contains(states, t): continue
                states.append(t)
                flag=1
        if not flag: break
    return states

def make_table(states):
    global nt_list, t_list
    def getstateno(t):
        for s in states:
            if len(s.closure) != len(t): continue
            if sorted(s.closure)==sorted(t):
                for i in range(len(s.closure)):
                    if s.closure[i]!=t[i]: break
                else: return s.no
        return -1
    def getprodno(closure):
        closure=''.join(closure).replace('.', '')
        return list(g.keys()).index(closure.split('->')[0])
    SLR_Table=OrderedDict()
    for i in range(len(states)):
        states[i]=State(states[i])
    for s in states:
        SLR_Table[s.no]=OrderedDict()
        for item in s.closure:
            head, body=item.split('->')
            if body=='.':
                for term in follows[item.split('->')[0]]:
                    if term not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                    else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}

                continue
            nextsym=body.split('.')[1]
            if nextsym=='':
                if getprodno(item)==0:
                    SLR_Table[s.no]['$']='accept'
                else:
                    for term in follows[item.split('->')[0]]:
```

```
                    if term not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                    else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
                continue
            nextsym=nextsym[0]
            t=goto(s.closure, nextsym)
            if t != []:
                if nextsym in t_list:
                    if nextsym not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][nextsym]={'s'+str(getstateno(t))}
                    else: SLR_Table[s.no][nextsym] |= {'s'+str(getstateno(t))}
                else: SLR_Table[s.no][nextsym] = str(getstateno(t))
    return SLR_Table

def augment_grammar():
    global start, aug
    for i in range(ord('Z'), ord('A')-1, -1):
        if chr(i) not in nt_list:
            g[chr(i)]=start
            aug = chr(i)
            return

def main():
    global ntl, nt_list, tl, t_list
    augment_grammar()
    follows[aug] = ['$']
    nt_list = list(g.keys())
    j = calc_states()
    ctr=0
    for s in j:
        print("Item{}:".format(ctr))
        for i in s:

        print("\t", i)
        ctr+=1
    table=make_table(j)

print('_____
_')
```

```
    print("\n\tSLR(1) TABLE\n")
    sym_list = nt_list + t_list

print('_____
_')
    print('\t|  ','\t|  '.join(sym_list),'\t\t|')

print('_____')
    for i, j in table.items():
        print(i, "\t|  ", '\t|  '.join(list(j.get(sym,' ') if type(j.get(sym))in (str , None) else
next(iter(j.get(sym,' ')))  for sym in sym_list)),'\t\t|')
        s, r=0, 0
        for p in j.values():
            if p!='accept' and len(p)>1:
                p=list(p)
                if('r' in p[0]): r+=1
                else: s+=1
                if('r' in p[1]): r+=1
                else: s+=1

print('_____
_')
    return
main()
```

```
Item0:
     Z->.S
     S->.AA
     A->.aA
     A->.b
Item1:
     Z->S.
Item2:
     S->A.A
     A->.aA
     A->.b
Item3:
     A->a.A
     A->.aA
     A->.b
Item4:
     A->b.
Item5:
     S->AA.
Item6:
     A->aA.
------------------------------------------------------------
```

```
    SLR(1) TABLE


-------------------------------------------------------------------
    |   S   |  A    |  Z    |  a    |  b    |  $        |
-------------------------------------------------------------------
0   |   1   |  2    |       |  s3   |  s4   |           |
1   |       |       |       |       |       |  r2       |
2   |       |  5    |       |  s3   |  s4   |           |
3   |       |  6    |       |  s3   |  s4   |           |
4   |       |       |       |  r1   |  r1   |           |
5   |       |       |       |       |       |  accept       |
6   |       |       |       |  r1   |  r1   |           |
-------------------------------------------------------------------

Process finished with exit code 0
```

## 7. Construct target code for the given expression.

```
import time
def findoperation(stmt, op, label):
    if(op == ">"):
        cmp = "BGT "+label
        print("ARM STATEMENT: ", cmp)
        time.sleep(0.02)
        stmt.append(cmp)
    elif(op == "<"):
        cmp = "BLT "+label
        print("ARM STATEMENT: ", cmp)
        time.sleep(0.02)
        stmt.append(cmp)
```

```
    elif(op == ">="):
        cmp = "BGE "+label
        print("ARM STATEMENT: ", cmp)
        time.sleep(0.02)
        stmt.append(cmp)
    elif(op == "<="):
        cmp = "BLE "+label
        print("ARM STATEMENT: ", cmp)
        time.sleep(0.02)
        stmt.append(cmp)
    elif(op == "=="):
        cmp = "BEQ "+label
        print("ARM STATEMENT: ", cmp)
        time.sleep(0.02)
        stmt.append(cmp)
    elif(op == "!="):
        cmp = "BNE "+label
        print("ARM STATEMENT: ", cmp)
        time.sleep(0.02)
        stmt.append(cmp)
    return stmt

def loadconstant(stmt, regval, value):
    lstmt = "MOV "+"R"+str(regval)+"," + "#" + value
    stmt.append(lstmt)
    print("ARM STATEMENT: ", lstmt)
    time.sleep(0.02)
    r1 = regval
    regval = (regval + 1)%13
    return stmt, regval, r1


def loadvariable(stmt, regval, value, isarr, offset=None):
    if(isarr == 0):
        st1 = "MOV "+"R" + str(regval) + ","+"="+str(value)
        r1 = regval
        regval = (regval + 1)%13

        print("ARM STATEMENT: ", st1)
```

```
        time.sleep(0.02)
        stmt.append(st1)

        st2 = "MOV "+"R" + str(regval) +","+ "[R" + str(r1) + "]"
        stmt.append(st2)
        print("ARM STATEMENT: ", st2)
        time.sleep(0.02)
        r2 = regval
        regval = (regval + 1)%13
        return stmt, regval, r1, r2
    else:
        st1 = "MOV "+"R" + str(regval) + ","+"="+str(value)
        r1 = regval
        regval = (regval + 1)%13

        print("ARM STATEMENT: ", st1)
        time.sleep(0.02)
        stmt.append(st1)
        if(not offset.isdigit()):
            st2 = "MOV "+"R" + str(regval) +","+ "[R" + str(r1) + "," + str(offset)+"]"
        else:
            st2 = "MOV "+"R" + str(regval) +","+ "[R" + str(r1) + "," + " #" +
str(offset)+"]"
        stmt.append(st2)
        print("ARM STATEMENT: ", st2)
        time.sleep(0.02)
        r2 = regval
        regval = (regval + 1)%13
        return stmt, regval, r1, r2
def binaryoperation(stmt, lhs, arg1, op, arg2):
    if(op == "+"):
        st = "ADD "+"R"+str(lhs)+","+"R"+str(arg1)+",R"+str(arg2)
        print("ARM STATEMENT: ", st)
        time.sleep(0.02)
        stmt.append(st)

    elif(op == "-"):
        st = "SUBS "+"R"+str(lhs)+","+"R"+str(arg1)+",R"+str(arg2)
        print("ARM STATEMENT: ", st)
```

```
            time.sleep(0.02)
            stmt.append(st)

        elif(op == "*"):
            st = "MUL "+"R"+str(lhs)+","+"R"+str(arg1)+",R"+str(arg2)
            print("ARM STATEMENT: ", st)
            time.sleep(0.02)
            stmt.append(st)

        elif(op == "/"):
            st = "SDIV "+"R"+str(lhs)+","+"R"+str(arg1)+",R"+str(arg2)
            print("ARM STATEMENT: ", st)
            time.sleep(0.02)
            stmt.append(st)
        return stmt
    offset = 0
    def genAssembly(lines, file):
        vardec = []
        stmt = []
        varlist = []
        regval = 0
        for i in lines:
            i = i.strip("\n")

            if(len(i.split()) == 2):
                if(i.split()[0] == "GOTO"):
                    st = "B " + i.split()[1]
                    print("ARM STATEMENT: ", st)
                    time.sleep(0.02)
                    stmt.append(st)
                else:
                    st = i
                    print("ARM STATEMENT: ", st)
                    time.sleep(0.02)
                    stmt.append(st)
            if(len(i.split()) == 5):
                lhs, ass, arg1, op, arg2 = i.split()
                if(lhs[0] == '*' and arg1[0] == '*'):
                    if(arg2.isdigit()):
```

```
            offset = arg2
        else:
            stmt, regval, r1,r2 = loadvariable(stmt, regval, arg2, 0)
            offset = "R"+str(r2)

    elif(arg1.isdigit() and arg2.isdigit()):

        stmt, regval, r1 = loadconstant(stmt, regval, arg1)
        stmt, regval, r2 = loadconstant(stmt, regval, arg2)
        if(lhs[0] == '*'):
            stmt, regval, r3, r4 = loadvariable(stmt, regval, lhs[1:], 1, offset)
        else:
            stmt, regval, r3, r4 = loadvariable(stmt, regval, lhs,0)
        stmt = binaryoperation(stmt, r4, r1, op, r2)
        if(lhs[0] == '*'):
            st = "STR R"+str(r4) + ", [R" + str(r3) + ", #", str(offset)+"]"
        else:
            st = "STR R"+str(r4) + ", [R" + str(r3) + "]"
        print("ARM STATEMENT: ", st)
        time.sleep(0.02)
        stmt.append(st)

    elif(arg1.isdigit()):
        stmt, regval, r1 = loadconstant(stmt, regval, arg1)
        if(arg2[0] == '*'):
            stmt, regval, r2, r3 = loadvariable(stmt, regval, arg2[1:], 1, offset)
        else:
            stmt, regval, r2, r3 = loadvariable(stmt, regval, arg2, 0)
        if(lhs[0] == '*'):
            stmt, regval, r4, r5 = loadvariable(stmt, regval, lhs[1:], 1, offset)
        else:
            stmt, regval, r4, r5 = loadvariable(stmt, regval, lhs, 0)
        stmt = binaryoperation(stmt, r5, r1, op, r3)
        if(lhs[0] == '*'):
            st = "STR R"+str(r5) + ", [R" + str(r4) + ", #"+str(offset)+"]"
        else:
            st = "STR R"+str(r5) + ", [R" + str(r4) + "]"
        print("ARM STATEMENT: ", st)
        time.sleep(0.02)
```

```
        stmt.append(st)
        #STR Op
    elif(arg2.isdigit()):
        if(arg1[0] == '*'):
            stmt, regval, r1, r2 = loadvariable(stmt, regval, arg1[1:], 1, offset)
        else:
            stmt, regval, r1,r2 = loadvariable(stmt, regval, arg1, 0)
        stmt, regval, r3 = loadconstant(stmt, regval, arg2)
        if(lhs[0] == '*'):
            stmt, regval, r4, r5 = loadvariable(stmt, regval, lhs[1:], 1, offset)
        else:
            stmt, regval, r4, r5 = loadvariable(stmt, regval, lhs,0)
        stmt = binaryoperation(stmt, r5, r2, op, r3)
        if(lhs[0] == '*'):
            st = "STR R"+str(r5) + ", [R" + str(r4) + ", #"+str(offset)+"]"
        else:
            st = "STR R"+str(r5) + ", [R" + str(r4) + "]"
        print("ARM STATEMENT: ", st)
        time.sleep(0.02)
        stmt.append(st)
    else:
        if(arg1[0] == '*'):
            stmt, regval, r1,r2 = loadvariable(stmt, regval, arg1[1:], 1, offset)
        else:
            stmt, regval, r1,r2 = loadvariable(stmt, regval, arg1, 0)
        if(arg2[0] == '*'):
            stmt, regval, r3,r4 = loadvariable(stmt, regval, arg2[1:], 1, offset)
        else:
            stmt, regval, r3,r4 = loadvariable(stmt, regval, arg2)
        if(lhs[0] == '*'):
            stmt, regval, r5,r6 = loadvariable(stmt, regval, lhs[1:], 1, offset)
        else:
            stmt, regval, r5,r6 = loadvariable(stmt, regval, lhs, 0)
        stmt = binaryoperation(stmt, r6, r2, op, r4)
        if(lhs[0] == '*'):
            st = "STR R"+str(r6) + ", [R" + str(r5) + ", #"+str(offset)+"]"
        else:
            st = "STR R"+str(r6) + ", [R" + str(r5) + "]"
        print("ARM STATEMENT: ", st)
```

```
            time.sleep(0.02)
            stmt.append(st)
    if(len(i.split())==4 and i.split()[0]=="ARR"):
        variable = i.split()[1]
        value = i.split()[3].split(",")
        if(variable not in varlist):
            out = ""
            out = out + variable + ":" + " .WORD "
            vals = ""
            for x in value:
                vals = vals + x + " "
            out = out + vals
            print("ARM DECLARATION :", out)
            time.sleep(0.02)
            vardec.append(out)
            varlist.append(variable)

    if(len(i.split()) == 4 and i.split()[0]!="ARR"):

        condition = i.split()[1]
        label = i.split()[3]
        flag = 0
        lhs = ""
        rhs = ""
        operator = [">", "<", ">=", "<=", "=", "!"]
        op = ""
        for j in condition:
            if(j in operator):
                op = op + j
                flag = 1
                continue
            if(j == "="):
                op = op + j
                continue
            if(flag == 0):
                lhs += j
            else:
                rhs+=j
```

```
if(rhs.isdigit() and lhs.isdigit()):
    stmt, regval, r1 = loadconstant(stmt, regval, lhs)
    stmt, regval, r2 = loadconstant(stmt, regval, rhs)
    cmp = "CMP R"+str(r1)+", "+"R"+str(r2)
    print("ARM STATEMENT: ", cmp)
    time.sleep(0.02)
    stmt.append(cmp)
    stmt = findoperation(stmt, op, label)

elif(lhs.isdigit()):
    stmt, regval, r1 = loadconstant(stmt, regval, lhs)
    if(rhs[0] == '*'):
        stmt, regval, r2, r3 = loadvariable(stmt, regval, rhs[1:], 1, offset)
    else:
        stmt, regval, r2, r3 = loadvariable(stmt, regval, rhs, 0)

    st4 = "CMP " + "R"+str(r1) + "," + "R" + str(r3)
    print("ARM STATEMENT: ", st4)
    time.sleep(0.02)
    stmt.append(st4)
    stmt = findoperation(stmt, op, label)
elif(rhs.isdigit()):
    if(lhs[0] == '*'):
        stmt, regval, r1, r2 = loadvariable(stmt, regval, lhs[1:], 1, offset)
    else:
        stmt, regval, r1, r2 = loadvariable(stmt, regval, lhs, 0)
    stmt, regval, r3 = loadconstant(stmt, regval, rhs)
    st4 = "CMP " + "R"+str(r2) + "," + "R" + str(r3)
    print("ARM STATEMENT: ", st4)
    time.sleep(0.02)
    stmt.append(st4)
    stmt = findoperation(stmt, op, label)
else:
    if(lhs[0] == '*'):
        stmt, regval, r1, r2 = loadvariable(stmt, regval, lhs[1:], 1, offset)
    else:
        stmt, regval, r1, r2 = loadvariable(stmt, regval, lhs, 0)
    if(rhs[0] == '*'):
        stmt, regval, r1, r2 = loadvariable(stmt, regval, lhs[1:], 1, offset)
```

```
            else:
                stmt, regval, r3, r4 = loadvariable(stmt, regval, rhs, 0)

            st4 = "CMP " + "R"+str(r2) + "," + "R" + str(r4)
            print("ARM STATEMENT: ", st4)
            time.sleep(0.02)
            stmt.append(st4)
            stmt = findoperation(stmt, op, label)



    if(len(i.split()) == 3):
        variable = i.split()[0]
        value = i.split()[2]
        variable = str(variable)
        if variable not in varlist:
            out = ""
            out = out + variable + ":" + " .WORD " + str(value)
            print("ARM DECLARATION :", out)
            time.sleep(0.02)
            vardec.append(out)
            varlist.append(variable)
        else:
            if(variable[0] == '*'):
                stmt, regval, r1, r2 = loadvariable(stmt, regval, variable[1:], 1, offset)
            else:
                stmt, regval, r1, r2 = loadvariable(stmt, regval, variable, 0)
            stmt, regval, r3 = loadconstant(stmt, regval, value)
            if(variable[0] == '*'):
                st = "STR R"+str(r3)+", [R" + str(r1) + ", #"+str(offset)+"]"
            else:
                st = "STR R"+str(r3)+", [R" + str(r1) + "]"
            print("ARM STATEMENT: ", st)
            time.sleep(0.02)
            stmt.append(st)
return vardec, stmt
```

```
def writeassembly(stmt, vardec, File):
    File.write(".text\n")
    for i in stmt:
        time.sleep(0.001)
        File.write("%s\n"%(i))
    File.write("SWI 0x011\n")
    File.write(".DATA\n")
    for i in vardec:

        time.sleep(0.01)
        File.write("%s\n"%(i))

    print("Written to File")

fin = open("input.txt", "r")
fout = open("output.s", "w")

lines = fin.readlines()
print("Generating Assembly ... ")
vardec, stmt = genAssembly(lines, fout)
print("Assembly Code Generated")
print("Writing to File")
print("---------------")
writeassembly(stmt, vardec, fout)
print("---------------")
print("Compilation Succesful")
fin.close()
fout.close()

fin.close()
fout.close()
```

---

**Input**

```
c = 0
if (a < b) goto (4)
goto (7)
```

**T1 = x + 1**
**x = T1**
**goto (9)**
**T2 = x − 1**
**x = T2**
**T3 = c + 1**
**c = T3**
**if (c < 5) goto (2)**

---

**Output**

```
.text
goto (7)
MOV R0,=x
MOV R1,[R0]
MOV R2,#1
MOV R3,=T1
MOV R4,[R3]
ADD R4,R1,R2
STR R4, [R3]
goto (9)
MOV R5,=x
MOV R6,[R5]
MOV R7,#1
MOV R8,=T2
MOV R9,[R8]
STR R9, [R8]
MOV R10,=x
MOV R11,[R10]
MOV R12,#T2
STR R12, [R10]
MOV R0,=c
MOV R1,[R0]
MOV R2,#1
MOV R3,=T3
MOV R4,[R3]
```

```
ADD R4,R1,R2
STR R4, [R3]
MOV R5,=c
MOV R6,[R5]
MOV R7,#T3
STR R7, [R5]
SWI 0x011
.DATA
c: .WORD 0
x: .WORD T1
```

## 8. Implement the CLR parsing table for the given grammar

S->CC

C->aC|d

```
pip install firfol==0.2.1

from collections import deque
from collections import OrderedDict
from pprint import pprint
from firfol import makeGrammar, findFirsts, findFollows

rules   = ['S->AA', 'A->aA|b']
start   = 'S'
aug     = ''
nt_list = ['S', 'A']
t_list  = ['a', 'b', '$']
g       = makeGrammar(rules)
firsts  = findFirsts(g)
follows = findFollows(g, start)

class State:
    _id=0
    def __init__(self, closure):
        self.closure=closure
```

```
        self.no=State._id
        State._id+=1

class Item(str):
    def __new__(cls, item, lookahead=list()):
        self=str.__new__(cls, item)
        self.lookahead=lookahead
        return self
    def __str__(self):
        return super(Item, self).__str__()+", "+'|'.join(self.lookahead)

def closure(items):
    def exists(newitem, items):
        for i in items:
            if i==newitem and
sorted(set(i.lookahead))==sorted(set(newitem.lookahead)):
                return True
        return False

    global g
    while True:
        flag=0
        for i in items:
            if i.index('.')==len(i)-1: continue
            Y=i.split('->')[1].split('.')[1][0]
            if i.index('.')+1<len(i)-1:
                lastr=list(firsts[i[i.index('.')+2]]-set(chr(1013)))
            else:
                lastr=i.lookahead
            for prod in g.keys():
                head, body=prod, g[prod]
                if head!=Y: continue
                for b in body:
                    newitem=Item(Y+'->.'+b, lastr)
                    if not exists(newitem, items):
                        items.append(newitem)
                        flag=1
        if flag==0: break
    return items
```

```
def goto(items, symbol):
    initial=[]
    for i in items:
        if i.index('.')==len(i)-1: continue
        head, body=i.split('->')
        seen, unseen=body.split('.')
        if unseen[0]==symbol and len(unseen) >= 1:
            initial.append(Item(head+'->'+seen+unseen[0]+'.'+unseen[1:],
i.lookahead))
    return closure(initial)

def calc_states():
    def contains(states, t):
        for s in states:
            if len(s) != len(t): continue
            if sorted(s)==sorted(t):
                for i in range(len(s)):
                        if s[i].lookahead!=t[i].lookahead: break
                else: return True

        return False
    global g, nt_list, t_list, aug
    head, body=aug, g[aug]
    for b in body:
        states=[closure([Item(head+'->.'+b, ['$'])])]
    while True:
        flag=0
        for s in states:
            for e in nt_list+t_list:
                t=goto(s, e)
                if t == [] or contains(states, t): continue
                states.append(t)
                flag=1
        if not flag: break
    return states

def make_table(states):
    global nt_list, t_list
```

```
def getstateno(t):
    for s in states:
        if len(s.closure) != len(t): continue
        if sorted(s.closure)==sorted(t):
            for i in range(len(s.closure)):
                if s.closure[i].lookahead!=t[i].lookahead: break
            else: return s.no
    return -1
def getprodno(closure):
    closure=''.join(closure).replace('.', '')
    return list(g.keys()).index(closure.split('->')[0])
SLR_Table=OrderedDict()
for i in range(len(states)):
    states[i]=State(states[i])
for s in states:
    SLR_Table[s.no]=OrderedDict()
    for item in s.closure:
        head, body=item.split('->')
        if body=='.':
            for term in item.lookahead:
                if term not in SLR_Table[s.no].keys():

                    SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
            continue
        nextsym=body.split('.')[1]
        if nextsym=='':
            if getprodno(item)==0:
                SLR_Table[s.no]['$']='accept'
            else:
                for term in item.lookahead:
                    if term not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][term]={'r'+str(getprodno(item))}
                    else: SLR_Table[s.no][term] |= {'r'+str(getprodno(item))}
            continue
        nextsym=nextsym[0]
        t=goto(s.closure, nextsym)
        if t != []:
            if nextsym in t_list:
```

```
                    if nextsym not in SLR_Table[s.no].keys():
                        SLR_Table[s.no][nextsym]={'s'+str(getstateno(t))}
                    else: SLR_Table[s.no][nextsym] |= {'s'+str(getstateno(t))}
                else: SLR_Table[s.no][nextsym] = str(getstateno(t))
    return SLR_Table

def augment_grammar():
    global start, aug
    for i in range(ord('Z'), ord('A')-1, -1):
        if chr(i) not in nt_list:
            g[chr(i)]=start
            aug = chr(i)
            return

def main():
    global ntl, nt_list, tl, t_list
    augment_grammar()
    nt_list = list(g.keys())
    j = calc_states()
    ctr=0
    for s in j:
        print("Item{}:".format(ctr))

        for i in s:
            print("\t", i)
        ctr+=1
    table=make_table(j)

print('_____
_')
    print("\n\tCLR(1) TABLE\n")
    sym_list = nt_list + t_list

print('_____
_')
    print('\t|  ','\t|  '.join(sym_list),'\t\t|')

print('_____
_')
```

```
    for i, j in table.items():
        print(i, "\t|  ", '\t|  '.join(list(j.get(sym,' ') if type(j.get(sym))in (str , None) else
next(iter(j.get(sym,' ')))  for sym in sym_list)),'\t\t|')
        s, r=0, 0
        for p in j.values():
            if p!='accept' and len(p)>1:
                p=list(p)
                if('r' in p[0]): r+=1
                else: s+=1
                if('r' in p[1]): r+=1
                else: s+=1

print('_____
_')
    return

main()
```

```
Item0:
        Z->.S, $
        S->.AA, $
        A->.aA, b|a
        A->.b, b|a
Item1:
        Z->S., $
Item2:
        S->A.A, $
        A->.aA, $
        A->.b, $
Item3:
        A->a.A, b|a
        A->.aA, b|a
        A->.b, b|a
Item4:
        A->b., b|a
Item5:
        S->AA., $
Item6:
        A->a.A, $
        A->.aA, $
        A->.b, $
Item7:
        A->b., $
Item8:
        A->aA., b|a
Item9:
        A->aA., $
```

CLR(1) TABLE

| | S | A | Z | a | b | $ | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | | s3 | s4 | | |
| 1 | | | | | | r2 | |
| 2 | | 5 | | s6 | s7 | | |
| 3 | | 8 | | s3 | s4 | | |
| 4 | | | | r1 | r1 | | |
| 5 | | | | | | accept | |
| 6 | | 9 | | s6 | s7 | | |
| 7 | | | | | | r1 | |
| 8 | | | | r1 | r1 | | |
| 9 | | | | | | r1 | |

9. Construct DAG for the given expression using value number method.

```
#include<stdio.h>
main(){
struct da{
int ptr,left,right;
char label;
} dag[25];
int ptr,l,j,change,n=0,i=0,state=1,x,y,k;
char store,*input1,input[25],var;clrscr();
for(i=0;i<25;i++){
dag[i].ptr=NULL;
dag[i].left=NULL;
dag[i].right=NULL;
dag[i].label=NULL;}
printf("\n\nENTER THE EXPRESSION\n\n");
scanf("%s",input1);/EX:((a*b-c))+((b-c)*d)) like this give with paranthesis.limitis 25
char ucan change that/
for(i=0;i<25;i++)
input[i]=NULL;
l=strlen(input1);
a:for(i=0;input1[i]!=')';i++);
for(j=i;input1[j]!='(';j--);
for(x=j+1;x<i;x++)
if(isalpha(input1[x]))
input[n++]=input1[x];
elseif(input1[x]!='0')
store=input1[x];
input[n++]=store;
for(x=j;x<=i;x++)
input1[x]='0';
if(input1[0]!='0')
goto a;for(i=0;i<n;i++){
dag[i].label=input[i];
dag[i].ptr=i;
if(!isalpha(input[i])&&!isdigit(input[i])){
```

```
dag[i].right=i-1;
ptr=i;
var=input[i-1];
if(isalpha(var))
ptr=ptr-2;
else{ptr=i-1;
b:if(!isalpha(var)&&!isdigit(var))
{ptr=dag[ptr].left;var=input[ptr];goto b;}
elseptr=ptr-1;}dag[i].left=ptr;}}
printf("\n SYNTAX TREE FOR GIVEN EXPRESSION\n\n");
printf("\n\n PTR \t\t LEFT PTR \t\t RIGHT PTR \t\t LABEL\n\n");
for(i=0;i<n;i++)/* draw the syntax tree for the followingoutput with pointer value*/
printf("\n%d\t%d\t%d\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].label);
getch();
for(i=0;i<n;i++){
for(j=0;j<n;j++){
if((dag[i].label==dag[j].label&&dag[i].left==dag[j].left)&&dag[i].right==dag[j].right
){
for(k=0;k<n;k++){
if(dag[k].left==dag[j].ptr)
dag[k].left=dag[i].ptr;
if(dag[k].right==dag[j].ptr)
dag[k].right=dag[i].ptr;}
dag[j].ptr=dag[i].ptr;}}}
printf("\n DAG FOR GIVEN EXPRESSION\n\n");
printf("\n\n PTR \t LEFT PTR \t RIGHT PTR \t LABEL \n\n");
for(i=0;i<n;i++)/draw DAG for the following output withpointer value/
printf("\n
%d\t\t%d\t\t%d\t\t%c\n",dag[i].ptr,dag[i].left,dag[i].right,dag[i].label);getch();
}
```

10. Identify the common subexpression from the given expression using DAG.

A.

```
#include<stdio.h>
#include<string.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];
operatormajid(char haj,char haj1)
{
 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;
}
int 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)
```

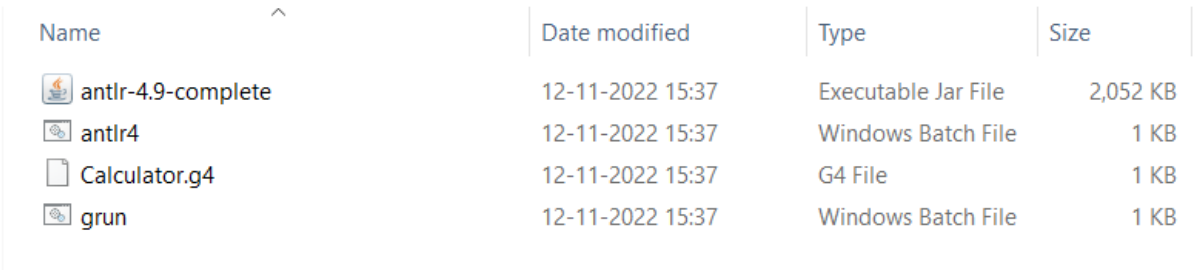
```
{
temp[j]='\0';
for(i=0;temp[i]!='\0';i++)
m[i]=temp[i];
m[i]='\0';
}
} /* end of evluating sub exp */
a=operatormajid('*','/'); /* operator fun call depends on priority */
d=operatormajid('+','-');
if(a==0&&d==0&&m[1]=='=')
printf("\n%s%d",m,k-1);
getch();
}
```

```
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)

The Intermediate Code may generated as-
T0=b*c
T1=d/T0
T2=a+T0
T3=T2-T1

...Program finished with exit code 0
Press ENTER to exit console.
```

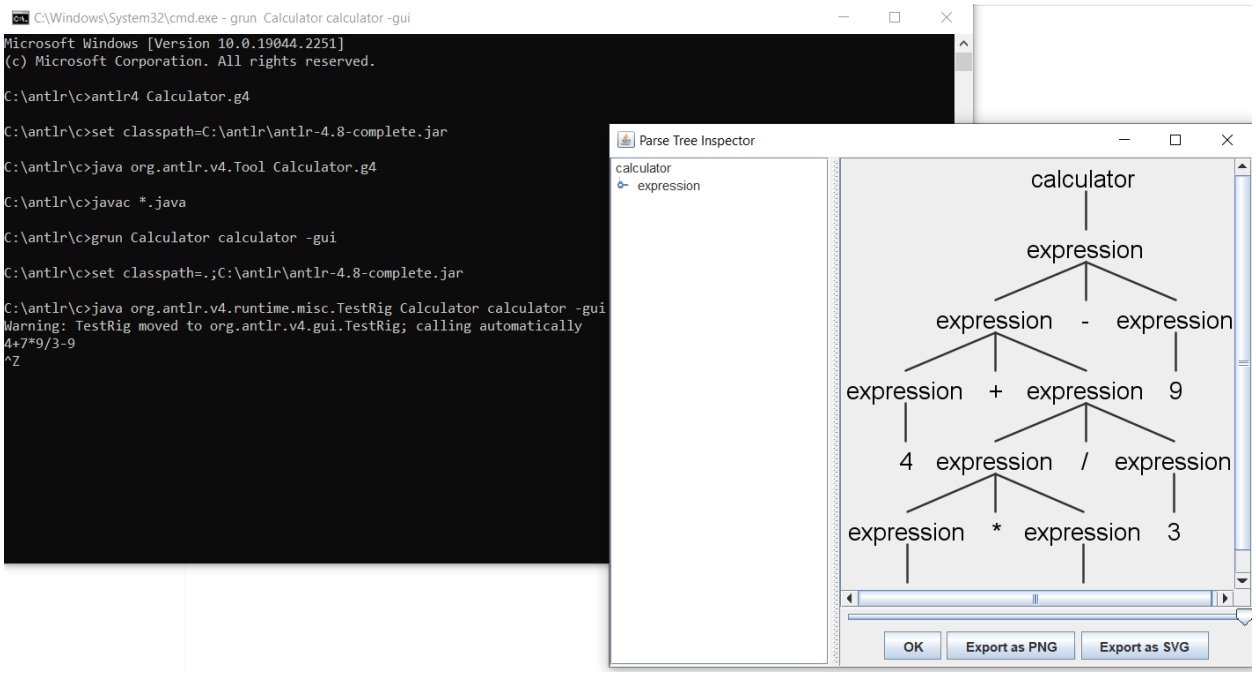
11. Construct syntax tree for expression grammar by using ANTLR.

## Calculator.g4

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

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



| Name | Date modified | Type | Size |
|---|---|---|---|
| antlr-4.9-complete | 12-11-2022 15:37 | Executable Jar File | 2,052 KB |
| antlr4 | 12-11-2022 15:37 | Windows Batch File | 1 KB |
| Calculator.g4 | 12-11-2022 15:37 | G4 File | 1 KB |
| grun | 12-11-2022 15:37 | Windows Batch File | 1 KB |




## antlr4.bat

set classpath=C:\antlr\antlr-4.8-complete.jar  (May change depending on machine)
java org.antlr.v4.Tool %*

## grun.bat

set classpath=.;C:\antlr\antlr-4.8-complete.jar (May change depending on machine)
java org.antlr.v4.runtime.misc.TestRig %*