Exp No	Title	Language
1	Lexical Analyzer	PYTHON
2	RE to NFA	PYTHON
3	NFA –DFA	PYTHON
4	A)LEFT RECURSION B)LEFT FACTORING	C++
5	FIRST FOLLOW	PYTHON
6	PREDICTIVE PARSING	PYTHON
7	SHIFT REDUCE PARSER	PYTHON
8	LEADING AND TRAILING	C++
9	LR0	C++
10	PREFIX-POSTFIX	PYTHON
11	THREE ADDRESS CODE	C
12	SIMPLE CODE GENERATOR	PYTHON
13	DAG	C

EXP1

LEXICAL ANALYZER

CODE –

```
file  = open("./add.c", 'r')
lines = file.readlines()

keywords     = ["void", "main", "int", "float", "bool", "if", "for", "else",
"while", "char", "return"]
operators    = ["=", "==", "+", "-", "*", "/", "++", "--", "+=", "-=", "!=",
"||", "&&"]
punctuations= [";", "(", ")", "{", "}", "[", "]"]

def is_int(x):
    try:
        int(x)
        return True
    except:
        return False

for line in lines:
    for i in line.strip().split(" "):
        if i in keywords:
            print (i, " is a keyword")
        elif i in operators:
            print (i, " is an operator")
        elif i in punctuations:
            print (i, " is a punctuation")
        elif is_int(i):
            print (i, " is a number")
        else:
            print (i, " is an identifier")
```

EXP 2

RE TO NFA

CODE

```python
rows, cols = (20, 3)
q = [[0]*cols]*rows

reg = input('Enter your regular expression : ')
len = len(reg)
i = 0
j = 1
print( q)
while(i<len):
    if reg[i] == 'a':
        try:
            if reg[i+1] != '|' and reg[i+1] !='*':
                q[j][0] = j+1
                j += 1
        except:
            q[j][0] = j+1

    elif reg[i] == 'b':
        try:
            if reg[i+1] != '|' and reg[i+1] !='*':
                q[j][1] = j+1
                j += 1
        except:
            q[j][1] = j+1

    elif reg[i]=='e' and reg[i+1]!='|'and reg[i+1]!='*':
        q[j][2]=j+1
        j+=1

    elif reg[i] == 'a' and reg[i+1] == '|' and reg[i+2] =='b':
        q[j][2]=((j+1)*10)+(j+3)
        j+=1
        q[j][0]=j+1
        j+=1
        q[j][2]=j+3
        j+=1
        q[j][1]=j+1
        j+=1
        q[j][2]=j+1
        j+=1
        i=i+2

    elif reg[i]=='b'and reg[i+1]=='|' and reg[i+2]=='a':
        q[j][2]=((j+1)*10)+(j+3)
        j+=1
        q[j][1]=j+1
        j+=1
        q[j][2]=j+3
        j+=1
        q[j][0]=j+1
        j+=1
        q[j][2]=j+1
        j+=1
        i=i+2

    elif reg[i]=='a' and reg[i+1]=='*':

        q[j][2]=((j+1)*10)+(j+3)
        j+=1
        q[j][0]=j+1
        j+=1
        q[j][2]=((j+1)*10)+(j-1)
        j+=1

    elif reg[i]=='b' and reg[i+1]=='*':
        q[j][2]=((j+1)*10)+(j+3)
        j+=1
        q[j][1]=j+1
        j+=1
        q[j][2]=((j+1)*10)+(j-1)
        j+=1

    elif reg[i]==')' and reg[i+1]=='*':

        q[0][2]=((j+1)*10)+1
        q[j][2]=((j+1)*10)+1
        j+=1

    i +=1

print("Transition Function ==>")

for i in range(0,j):
    if q[i][0]!=0:

        print(f"\n {q[i]},a --> {q[i][0]}")

    elif q[i][1]!=0:
        print (f"\n {q[i]},b-->{q[i][1]}")
    elif q[i][2]!=0:

        if q[i][2]<10:
            print(f"\n {q[i]},e-->{q[i][2]}")
        else:
            print(f"\n {q[i]},e-->{q[i][2]}/10 and {q[i][2]}%10")
```

EXP 3

NFA TO DFA

CODE

```
import pandas as pd

nfa = {}
n = int(input("No. of states : "))
t = int(input("No. of transitions : "))
for i in range(n):
    state = input("state name : ")
    nfa[state] = {}
    for j in range(t):
        path = input("path : ")
        print("Enter end state from state {} travelling through path {} :
".format(state, path))
        reaching_state = [x for x in input().split()]
        nfa[state][path] = reaching_state

print("\nNFA :- \n")
print(nfa)
print("\nPrinting NFA table :- ")
nfa_table = pd.DataFrame(nfa)
print(nfa_table.transpose())

print("Enter final state of NFA : ")
nfa_final_state = [x for x in input().split()]

new_states_list = []

#-------------------------------------------------

dfa = {}
keys_list = list(
    list(nfa.keys())[0])
path_list = list(nfa[keys_list[0]].keys())

dfa[keys_list[0]] = {}
for y in range(t):
    var = "".join(nfa[keys_list[0]][
                  path_list[y]])
    dfa[keys_list[0]][path_list[y]] = var
    if var not in keys_list:
        new_states_list.append(var)
        keys_list.append(var)

while len(new_states_list) != 0:
```

```python
    dfa[new_states_list[0]] = {}
    for _ in range(len(new_states_list[0])):
        for i in range(len(path_list)):
            temp = []
            for j in range(len(new_states_list[0])):
                temp += nfa[new_states_list[0][j]][path_list[i]]
            s = ""
            s = s.join(temp)
            if s not in keys_list:
                new_states_list.append(s)
                keys_list.append(s)
            dfa[new_states_list[0]][path_list[i]] = s

    new_states_list.remove(new_states_list[0])

print("\nDFA :- \n")
print(dfa)
print("\nPrinting DFA table :- ")
dfa_table = pd.DataFrame(dfa)
print(dfa_table.transpose())

dfa_states_list = list(dfa.keys())
dfa_final_states = []
for x in dfa_states_list:
    for i in x:
        if i in nfa_final_state:
            dfa_final_states.append(x)
            break

print("\nFinal states of the DFA are : ", dfa_final_states)
```

EXP 4

LEFT RECURSION

LEFT FACTORING

CODE

```cpp
#include <iostream>
#include <math.h>
#include <vector>
#include <string>
#include <stdlib.h>
using namespace std;

int main()
{
    cout<<"\nEnter number of productions: ";
    int p;
    cin>>p;
    vector<string> prodleft(p),prodright(p);
    cout<<"\nEnter productions one by one: ";
    int i;
    for(i=0;i<p;++i) {
        cout<<"\nLeft of production "<<i+1<<": ";
        cin>>prodleft[i];
        cout<<"\nRight of production "<<i+1<<": ";
        cin>>prodright[i];
    }
    int j;
    int e=1;
    for(i=0;i<p;++i) {
        for(j=i+1;j<p;++j) {
            if(prodleft[j]==prodleft[i]) {
                int k=0;
                string com="";
                while(k<prodright[i].length()&&k<prodright[j].length()&&prodri
ght[i][k]==prodright[j][k]) {
                    com+=prodright[i][k];
                    ++k;
                }
                if(k==0)
                    continue;
                char* buffer;
                string comleft=prodleft[i];
                if(k==prodright[i].length()) {
                    prodleft[i]+=string(itoa(e,buffer,10));
                    prodleft[j]+=string(itoa(e,buffer,10));
                    prodright[i]="^";
                    prodright[j]=prodright[j].substr(k,prodright[j].length()-
k);
                }
                else if(k==prodright[j].length()) {
                    prodleft[i]+=string(itoa(e,buffer,10));
                    prodleft[j]+=string(itoa(e,buffer,10));
                    prodright[j]="^";
                    prodright[i]=prodright[i].substr(k,prodright[i].length()-
k);
                }
                else {
                    prodleft[i]+=string(itoa(e,buffer,10));
                    prodleft[j]+=string(itoa(e,buffer,10));
                    prodright[j]=prodright[j].substr(k,prodright[j].length()-
k);

                    prodright[i]=prodright[i].substr(k,prodright[i].length()-
k);
                }
                int l;
                for(l=j+1;l<p;++l) {
                    if(comleft==prodleft[l]&&com==prodright[l].substr(0,fmin(k
,prodright[l].length()))) {
                        prodleft[l]+=string(itoa(e,buffer,10));
                        prodright[l]=prodright[l].substr(k,prodright[l].length
()-k);
                    }
                }
                prodleft.push_back(comleft);
                prodright.push_back(com+prodleft[i]);
                ++p;
                ++e;
            }
        }
    }
    cout<<"\n\nNew productions";
    for(i=0;i<p;++i) {
        cout<<"\n"<<prodleft[i]<<"->"<<prodright[i];
    }
    return 0;
}
```

B)

```cpp
#include<iostream>

#include<string>
using namespace std;
int main()
{   string ip,op1,op2,temp;
    int sizes[10] = {};
    char c;
    int n,j,l;
    cout<<"Enter the Parent Non-Terminal : ";
    cin>>c;
    ip.push_back(c);
    op1 += ip + "\'->";
    ip += "->";
    op2+=ip;
    cout<<"Enter the number of productions : ";
    cin>>n;
    for(int i=0;i<n;i++)
    {   cout<<"Enter Production "<<i+1<<" : ";
        cin>>temp;
        sizes[i] = temp.size();
        ip+=temp;
        if(i!=n-1)
            ip += "|";
    }
    cout<<"Production Rule : "<<ip<<endl;
    for(int i=0,k=3;i<n;i++)
    {
        if(ip[0] == ip[k])
        {
            cout<<"Production "<<i+1<<" has left recursion."<<endl;
            if(ip[k] != '#')
            {
                for(l=k+1;l<k+sizes[i];l++)
                    op1.push_back(ip[l]);
                k=l+1;
                op1.push_back(ip[0]);
                op1 += "\'|";
            }
        }
        else
        {
            cout<<"Production "<<i+1<<" does not have left recursion."<<endl;
            if(ip[k] != '#')
            {
                for(j=k;j<k+sizes[i];j++)
                    op2.push_back(ip[j]);
                k=j+1;
                op2.push_back(ip[0]);
                op2 += "\'|";
            }
            else
            {
                op2.push_back(ip[0]);
                op2 += "\'";
            }}}
    op1 += "#";
    cout<<op2<<endl;
    cout<<op1<<endl;
    return 0;}
```

EXP 5

FIRST FOLLOW

CODE

```python
import sys
sys.setrecursionlimit(60)

def first(string):
    #print("first({})".format(string))
    first_ = set()
    if string in non_terminals:
        alternatives = productions_dict[string]

        for alternative in alternatives:
            first_2 = first(alternative)
            first_ = first_ |first_2

    elif string in terminals:
        first_ = {string}

    elif string=='' or string=='@':
        first_ = {'@'}

    else:
        first_2 = first(string[0])
        if '@' in first_2:
            i = 1
            while '@' in first_2:
                #print("inside while")

                first_ = first_ | (first_2 - {'@'})
                #print('string[i:]=', string[i:])
                if string[i:] in terminals:
                    first_ = first_ | {string[i:]}
                    break
                elif string[i:] == '':
                    first_ = first_ | {'@'}
                    break
                first_2 = first(string[i:])
                first_ = first_ | first_2 - {'@'}
                i += 1
        else:
            first_ = first_ | first_2


    #print("returning for first({})".format(string),first_)
    return  first_
```

```python
def follow(nT):
    #print("inside follow({})".format(nT))
    follow_ = set()
    #print("FOLLOW", FOLLOW)
    prods = productions_dict.items()
    if nT==starting_symbol:
        follow_ = follow_ | {'$'}
    for nt,rhs in prods:
        #print("nt to rhs", nt,rhs)
        for alt in rhs:
            for char in alt:
                if char==nT:
                    following_str = alt[alt.index(char) + 1:]
                    if following_str=='':
                        if nt==nT:
                            continue
                        else:
                            follow_ = follow_ | follow(nt)
                    else:
                        follow_2 = first(following_str)
                        if '@' in follow_2:
                            follow_ = follow_ | follow_2-{'@'}
                            follow_ = follow_ | follow(nt)
                        else:
                            follow_ = follow_ | follow_2
    #print("returning for follow({})".format(nT),follow_)
    return follow_





no_of_terminals=int(input("Enter no. of terminals: "))

terminals = []

print("Enter the terminals :")
for _ in range(no_of_terminals):
    terminals.append(input())

no_of_non_terminals=int(input("Enter no. of non terminals: "))

non_terminals = []

print("Enter the non terminals :")
for _ in range(no_of_non_terminals):
    non_terminals.append(input())
```

```
starting_symbol = input("Enter the starting symbol: ")

no_of_productions = int(input("Enter no of productions: "))

productions = []

print("Enter the productions:")
for _ in range(no_of_productions):
    productions.append(input())


#print("terminals", terminals)

#print("non terminals", non_terminals)

#print("productions",productions)


productions_dict = {}

for nT in non_terminals:
    productions_dict[nT] = []


#print("productions_dict",productions_dict)

for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("/")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)

#print("productions_dict",productions_dict)

#print("nonterm_to_prod",nonterm_to_prod)
#print("alternatives",alternatives)


FIRST = {}
FOLLOW = {}

for non_terminal in non_terminals:
    FIRST[non_terminal] = set()

for non_terminal in non_terminals:
    FOLLOW[non_terminal] = set()

#print("FIRST",FIRST)
```

```python
for non_terminal in non_terminals:
    FIRST[non_terminal] = FIRST[non_terminal] | first(non_terminal)

#print("FIRST",FIRST)


FOLLOW[starting_symbol] = FOLLOW[starting_symbol] | {'$'}
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = FOLLOW[non_terminal] | follow(non_terminal)

#print("FOLLOW", FOLLOW)

print("{: ^20}{: ^20}{: ^20}".format('Non Terminals','First','Follow'))
for non_terminal in non_terminals:
    print("{: ^20}{: ^20}{:
^20}".format(non_terminal,str(FIRST[non_terminal]),str(FOLLOW[non_terminal])))
```

EXP 6

PREDICTIVE PARSING

CODE

```
gram = {
    "E":["E+T","T"],
    "T":["T*F","F"],
    "F":["(E)","i"],
    # "S":["CC"],
    # "C":["eC","d"],
}

def removeDirectLR(gramA, A):
    """gramA is dictonary"""
    temp = gramA[A]
    tempCr = []
    tempInCr = []
    for i in temp:
        if i[0] == A:
            #tempInCr.append(i[1:])
            tempInCr.append(i[1:]+[A+"'"])
        else:
            #tempCr.append(i)
            tempCr.append(i+[A+"'"])
    tempInCr.append(["e"])
    gramA[A] = tempCr
    gramA[A+"'"] = tempInCr
    return gramA


def checkForIndirect(gramA, a, ai):
    if ai not in gramA:
        return False
    if a == ai:
        return True
    for i in gramA[ai]:
        if i[0] == ai:
            return False
        if i[0] in gramA:
            return checkForIndirect(gramA, a, i[0])
    return False

def rep(gramA, A):
    temp = gramA[A]
    newTemp = []
    for i in temp:
        if checkForIndirect(gramA, A, i[0]):
            t = []
            for k in gramA[i[0]]:
                t=[]
                t+=k
                t+=i[1:]
                newTemp.append(t)

        else:
            newTemp.append(i)
    gramA[A] = newTemp
    return gramA

def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1

    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)


    #print(gramA)
    for i in range(c-1,0,-1):
        ai = "A"+str(i)
        for j in range(0,i):
            aj = gramA[ai][0][0]
            if ai!=aj :
                if aj in gramA and checkForIndirect(gramA,ai,aj):
                    gramA = rep(gramA, ai)

    for i in range(1,c):
        ai = "A"+str(i)
        for j in gramA[ai]:
            if ai==j[0]:
                gramA = removeDirectLR(gramA, ai)
                break
```

```python
    op = {}
    for i in gramA:
        a = str(i)
        for j in conv:
            a = a.replace(conv[j],j)
        revconv[i] = a

    for i in gramA:
        l = []
        for j in gramA[i]:
            k = []
            for m in j:
                if m in revconv:
                    k.append(m.replace(m,revconv[m]))
                else:
                    k.append(m)
            l.append(k)
        op[revconv[i]] = l

    return op

result = rem(gram)
terminals = []
for i in result:
    for j in result[i]:
        for k in j:
            if k not in result:
                terminals+=[k]
terminals = list(set(terminals))
#print(terminals)

def first(gram, term):
    a = []
    if term not in gram:
        return [term]
    for i in gram[term]:
        if i[0] not in gram:
            a.append(i[0])
        elif i[0] in gram:
            a += first(gram, i[0])
    return a

firsts = {}
for i in result:
    firsts[i] = first(result,i)
#   print(f'First({i}):',firsts[i])
```

```python
def follow(gram, term):
    a = []
    for rule in gram:
        for i in gram[rule]:
            if term in i:
                temp = i
                indx = i.index(term)
                if indx+1!=len(i):
                    if i[-1] in firsts:
                        a+=firsts[i[-1]]
                    else:
                        a+=[i[-1]]
                else:
                    a+=["e"]
                if rule != term and "e" in a:
                    a+= follow(gram,rule)
    return a

follows = {}
for i in result:
    follows[i] = list(set(follow(result,i)))
    if "e" in follows[i]:
        follows[i].pop(follows[i].index("e"))
    follows[i]+=["$"]
#   print(f'Follow({i}):',follows[i])

resMod = {}
for i in result:
    l = []
    for j in result[i]:
        temp = ""
        for k in j:
            temp+=k
        l.append(temp)
    resMod[i] = l

# create predictive parsing table
tterm = list(terminals)
tterm.pop(tterm.index("e"))
tterm+=["d"]
pptable = {}
for i in result:
    for j in tterm:
        if j in firsts[i]:
            pptable[(i,j)]=resMod[i[0]][0]
        else:
            pptable[(i,j)]=""
    if "e" in firsts[i]:
        for j in tterm:
            if j in follows[i]:
                pptable[(i,j)]="e"
pptable[("F","i")] = "i"
toprint = f'{"": <10}'
for i in tterm:
    toprint+= f'|{i: <10}'
print(toprint)
for i in result:
    toprint = f'{i: <10}'
    for j in tterm:
        if pptable[(i,j)]!="":
            toprint+=f'|{i+"->"+pptable[(i,j)]: <10}'
        else:
            toprint+=f'|{pptable[(i,j)]: <10}'
    print(f'{"-":-<76}')
    print(toprint)
```

EXP 7

SHIFT REDUCE PARSER

CODE

```
gram = {
    "E":["E*E","E+E","i"]
}
starting_terminal = "E"

inp = input("Enter the string \n")
inp=inp+"$"

stack = "$"
print(f'{"Stack": <15}'+"|"+f'{"Input Buffer": <15}'+"|"+f'Parsing Action')
print(f'{"-":-<50}')

while True:
    action = True
    i = 0
    while i<len(gram[starting_terminal]):
        if gram[starting_terminal][i] in stack:
            stack =
stack.replace(gram[starting_terminal][i],starting_terminal)
            print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Reduce S-
>{gram[starting_terminal][i]}')
            i=-1
            action = False
        i+=1
    if len(inp)>1:
        stack+=inp[0]
        inp=inp[1:]
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Shift')
        action = False

    if inp == "$" and stack == ("$"+starting_terminal):
        print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Accepted')
        break
```

EXP 8

LEADING AND TRAILING

CODE

```
#include<iostream>
#include<conio.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
using namespace std;

int vars,terms,i,j,k,m,rep,cont,temp=-1;
char var[10],term[10],lead[10][10],trail[10][10];
struct grammar
{
    int prodno;
    char lhs,rhs[20][20];
}gram[50];
void get()
{
    cout<<"\nLEADING AND TRAILING\n";
    cout<<"\nEnter the no. of variables : ";
    cin>>vars;
    cout<<"\nEnter the variables : \n";
    for(i=0;i<vars;i++)
    {
        cin>>gram[i].lhs;
        var[i]=gram[i].lhs;
    }
    cout<<"\nEnter the no. of terminals : ";
    cin>>terms;
    cout<<"\nEnter the terminals : ";
    for(j=0;j<terms;j++)
        cin>>term[j];
    cout<<"\nPRODUCTION DETAILS\n";
    for(i=0;i<vars;i++)
    {
        cout<<"\nEnter the no. of production of "<<gram[i].lhs<<":";
        cin>>gram[i].prodno;
        for(j=0;j<gram[i].prodno;j++)
        {
            cout<<gram[i].lhs<<"->";
            cin>>gram[i].rhs[j];
        }
    }
}
void leading()
```

```
{
    for(i=0;i<vars;i++)
    {
        for(j=0;j<gram[i].prodno;j++)
        {
            for(k=0;k<terms;k++)
            {
                if(gram[i].rhs[j][0]==term[k])
                    lead[i][k]=1;
                else
                {
                    if(gram[i].rhs[j][1]==term[k])
                        lead[i][k]=1;
                }
            }
        }
    }
    for(rep=0;rep<vars;rep++)
    {
        for(i=0;i<vars;i++)
        {
            for(j=0;j<gram[i].prodno;j++)
            {
                for(m=1;m<vars;m++)
                {
                    if(gram[i].rhs[j][0]==var[m])
                    {
                        temp=m;
                        goto out;
                    }
                }
                out:
                for(k=0;k<terms;k++)
                {
                    if(lead[temp][k]==1)
                        lead[i][k]=1;
                }
            }
        }
    }
}
void trailing()
{
    for(i=0;i<vars;i++)
    {
        for(j=0;j<gram[i].prodno;j++)
        {
            cont=0;
            while(gram[i].rhs[j][cont]!='\x0')
                cont++;
            for(k=0;k<terms;k++)
            {
                if(gram[i].rhs[j][cont-1]==term[k])
                    trail[i][k]=1;
                else
                {
                    if(gram[i].rhs[j][cont-2]==term[k])
                        trail[i][k]=1;
                }
            }
        }
    }
    for(rep=0;rep<vars;rep++)
    {
        for(i=0;i<vars;i++)
        {
            for(j=0;j<gram[i].prodno;j++)
            {
                cont=0;
                while(gram[i].rhs[j][cont]!='\x0')
                    cont++;
                for(m=1;m<vars;m++)
                {
                    if(gram[i].rhs[j][cont-1]==var[m])
                        temp=m;
                }
                for(k=0;k<terms;k++)
                {
                    if(trail[temp][k]==1)
                        trail[i][k]=1;
                }
            }
        }
    }
}
void display()
{
    for(i=0;i<vars;i++)
    {
        cout<<"\nLEADING("<<gram[i].lhs<<") = ";
        for(j=0;j<terms;j++)
        {
            if(lead[i][j]==1)
                cout<<term[j]<<",";
        }
    }
    cout<<endl;
    for(i=0;i<vars;i++)
    {
        cout<<"\nTRAILING("<<gram[i].lhs<<") = ";
        for(j=0;j<terms;j++)
        {
            if(trail[i][j]==1)
                cout<<term[j]<<",";
        }
    }
}
int main()
{

    get();
    leading();
    trailing();
    display();

}
```

EXP 9

LR0

CODE

```
#include<iostream>
#include<conio.h>
#include<string.h>

using namespace std;

char prod[20][20],listofvar[26]="ABCDEFGHIJKLMNOPQR";
int novar=1,i=0,j=0,k=0,n=0,m=0,arr[30];
int noitem=0;

struct Grammar
{
    char lhs;
    char rhs[8];
}g[20],item[20],clos[20][10];

int isvariable(char variable)
{
    for(int i=0;i<novar;i++)
        if(g[i].lhs==variable)
            return i+1;
    return 0;
}
void findclosure(int z, char a)
{
    int n=0,i=0,j=0,k=0,l=0;
    for(i=0;i<arr[z];i++)
    {
        for(j=0;j<strlen(clos[z][i].rhs);j++)
        {
            if(clos[z][i].rhs[j]=='.' && clos[z][i].rhs[j+1]==a)
            {
                clos[noitem][n].lhs=clos[z][i].lhs;
                strcpy(clos[noitem][n].rhs,clos[z][i].rhs);
                char temp=clos[noitem][n].rhs[j];
                clos[noitem][n].rhs[j]=clos[noitem][n].rhs[j+1];
                clos[noitem][n].rhs[j+1]=temp;
                n=n+1;
            }
        }
    }
    for(i=0;i<n;i++)
    {
        for(j=0;j<strlen(clos[noitem][i].rhs);j++)
        {
            if(clos[noitem][i].rhs[j]=='.' &&
isvariable(clos[noitem][i].rhs[j+1])>0)
            {
                for(k=0;k<novar;k++)
                {
                    if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)
                    {
                        for(l=0;l<n;l++)
                            if(clos[noitem][l].lhs==clos[0][k].lhs &&
strcmp(clos[noitem][l].rhs,clos[0][k].rhs)==0)
                                break;
                        if(l==n)
                        {
                            clos[noitem][n].lhs=clos[0][k].lhs;
                        strcpy(clos[noitem][n].rhs,clos[0][k].rhs);
                            n=n+1;
                        }
                    }
                }
            }
        }
    }
    arr[noitem]=n;
    int flag=0;
    for(i=0;i<noitem;i++)
    {
        if(arr[i]==n)
        {
            for(j=0;j<arr[i];j++)
            {
                int c=0;
                for(k=0;k<arr[i];k++)
                    if(clos[noitem][k].lhs==clos[i][k].lhs &&
strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)
                        c=c+1;
                if(c==arr[i])
                {
                    flag=1;
                    goto exit;
                }
            }
        }
    }
    exit:;
    if(flag==0)
        arr[noitem++]=n;
```

```
}

int main()
{
    cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :\n";
    do
    {
        cin>>prod[i++];
    }while(strcmp(prod[i-1],"0")!=0);
    for(n=0;n<i-1;n++)
    {
        m=0;
        j=novar;
        g[novar++].lhs=prod[n][0];
        for(k=3;k<strlen(prod[n]);k++)
        {
            if(prod[n][k] != '|')
            g[j].rhs[m++]=prod[n][k];
            if(prod[n][k]=='|')
            {
                g[j].rhs[m]='\0';
                m=0;
                j=novar;
                g[novar++].lhs=prod[n][0];
            }
        }
    }
    for(i=0;i<26;i++)
        if(!isvariable(listofvar[i]))
            break;
    g[0].lhs=listofvar[i];
    char temp[2]={g[1].lhs,'\0'};
    strcat(g[0].rhs,temp);
    cout<<"\n\n augumented grammar \n";
    for(i=0;i<novar;i++)
        cout<<endl<<g[i].lhs<<"->"<<g[i].rhs<<" ";

    for(i=0;i<novar;i++)
    {
        clos[noitem][i].lhs=g[i].lhs;
        strcpy(clos[noitem][i].rhs,g[i].rhs);
        if(strcmp(clos[noitem][i].rhs,"ε")==0)
            strcpy(clos[noitem][i].rhs,".");
        else
        {
            for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--)
                clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];
            clos[noitem][i].rhs[0]='.';
        }
    }
    arr[noitem++]=novar;
    for(int z=0;z<noitem;z++)
    {
        char list[10];
        int l=0;
        for(j=0;j<arr[z];j++)
        {
            for(k=0;k<strlen(clos[z][j].rhs)-1;k++)
            {
                if(clos[z][j].rhs[k]=='.')
                {
                    for(m=0;m<l;m++)
                        if(list[m]==clos[z][j].rhs[k+1])
                            break;
                    if(m==l)
                        list[l++]=clos[z][j].rhs[k+1];
                }
            }
        }
        for(int x=0;x<l;x++)
            findclosure(z,list[x]);
    }
    cout<<"\n THE SET OF ITEMS ARE \n\n";
    for(int z=0; z<noitem; z++)
    {
        cout<<"\n I"<<z<<"\n\n";
        for(j=0;j<arr[z];j++)
            cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";

    }

}
```

EXP 10

PREFIX POSTFIX

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])

PRI = {'+': 1, '-': 1, '*': 2, '/': 2}
def infix_to_postfix(formula):
    stack = []  # only pop when the coming op has priority

    output = ''

    for ch in formula:

        if ch not in OPERATORS:

            output += ch

        elif ch == '(':

            stack.append('(')

        elif ch == ')':

            while stack and stack[-1] != '(':
                output += stack.pop()

            stack.pop()  # pop '('

        else:

            while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
                output += stack.pop()

            stack.append(ch)

            # leftover

    while stack:
        output += stack.pop()

    print(f'POSTFIX: {output}')

    return output


### INFIX ===> PREFIX ###

def infix_to_prefix(formula):
    op_stack = []

    exp_stack = []

    for ch in formula:

        if not ch in OPERATORS:

            exp_stack.append(ch)

        elif ch == '(':

            op_stack.append(ch)

        elif ch == ')':

            while op_stack[-1] != '(':
                op = op_stack.pop()

                a = exp_stack.pop()

                b = exp_stack.pop()

                exp_stack.append(op + b + a)

            op_stack.pop()  # pop '('

        else:

            while op_stack and op_stack[-1] != '(' and PRI[ch] <=
PRI[op_stack[-1]]:
                op = op_stack.pop()

                a = exp_stack.pop()

                b = exp_stack.pop()

                exp_stack.append(op + b + a)

            op_stack.append(ch)

            # leftover

    while op_stack:
        op = op_stack.pop()

        a = exp_stack.pop()
        b = exp_stack.pop()

        exp_stack.append(op + b + a)

    print(f'PREFIX: {exp_stack[-1]}')

    return exp_stack[-1]

expres = input("INPUT THE EXPRESSION: ")

pre = infix_to_prefix(expres)

pos = infix_to_postfix(expres)
```

EXP 11

THREE ADDRESS CODE

CODE-

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
void small();
void dove(int i);
int p[5]={0,1,2,3,4},c=1,i,k,l,m,pi;
char sw[5]={'=','-','+','/','*'},j[20],a[5],b[5],ch[2];
void main()
{
printf("Enter the expression:");
scanf("%s",j);
printf("\tThe Intermediate code is:\n");
small();
}
void dove(int i)
{
a[0]=b[0]='\0';
if(!isdigit(j[i+2])&&!isdigit(j[i-2]))
{
a[0]=j[i-1];
b[0]=j[i+1];
}
if(isdigit(j[i+2])){
a[0]=j[i-1];
b[0]='t';
b[1]=j[i+2];
}
if(isdigit(j[i-2]))
{
b[0]=j[i+1];
a[0]='t';
a[1]=j[i-2];
b[1]='\0';
}
if(isdigit(j[i+2]) &&isdigit(j[i-2]))
{
a[0]='t';
b[0]='t';
a[1]=j[i-2];
b[1]=j[i+2];
sprintf(ch,"%d",c);
j[i+2]=j[i-2]=ch[0];
```

```c
}
if(j[i]=='*')
printf("\tt%d=%s*%s\n",c,a,b);
if(j[i]=='/')
printf("\tt%d=%s/%s\n",c,a,b);
if(j[i]=='+')
printf("\tt%d=%s+%s\n",c,a,b);if(j[i]=='-')
printf("\tt%d=%s-%s\n",c,a,b);
if(j[i]=='=')
printf("\t%c=t%d",j[i-1],--c);
sprintf(ch,"%d",c);
j[i]=ch[0];
c++;
small();
}
void small()
{
pi=0;l=0;
for(i=0;i<strlen(j);i++)
{
for(m=0;m<5;m++)
if(j[i]==sw[m])
if(pi<=p[m])
{
pi=p[m];
 l=1;
 k=i;
}
}
if(l==1)
dove(k);
else
exit(0);}
```

EXP 12

SIMPLE CODE GENERATOR

CODE

```python
n=[]
di={'+':'ADD','-':'SUB','*':'MUL','/':'DIV'}
while True:
    n.append(input())
    if(n[-1]=="exit"):
        n.pop(-1)
        break
j=0
for i in n:
    op=di[i[3]]
    print("MOV",i[2],",R",str(j))
    print(op,i[4],",R",str(j))
    print("MOV R",str(j),",",i[0])
    j+=1
```

EXP 13

DAG

CODE-

```
#include<stdio.h>
#include<string.h>
int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp
{
 int pos;
 char op;
}k[15];
void main()
{

 printf("\t\tINTERMEDIATE CODE GENERATION OF DAG\n\n");

 scanf("%s",str);
 printf("The intermediate code:\t\tExpression\n");
 findopr();
 explore();

}
void findopr()
{
 for(i=0;str[i]!='\0';i++)
  if(str[i]==':')
  {
  k[j].pos=i;
  k[j++].op=':';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='/')
  {
  k[j].pos=i;
  k[j++].op='/';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='*')
  {
  k[j].pos=i;
  k[j++].op='*';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='+')
  {
  k[j].pos=i;
  k[j++].op='+';
  }
 for(i=0;str[i]!='\0';i++)
  if(str[i]=='-')
  {
  k[j].pos=i;
  k[j++].op='-';
  }
}
void explore()
{
 i=1;
 while(k[i].op!='\0')
 {
  fleft(k[i].pos);
  fright(k[i].pos);
  str[k[i].pos]=tmpch--;
  printf("\t%c := %s%c%s\t\t",str[k[i].pos],left,k[i].op,right);
  for(j=0;j <strlen(str);j++)
   if(str[j]!='$')
    printf("%c",str[j]);
  printf("\n");
  i++;
 }
 fright(-1);
 if(no==0)
 {
  fleft(strlen(str));
  printf("\t%s := %s",right,left);
 }
 printf("\t%s :=  %c",right,str[k[--i].pos]);

}
void fleft(int x)
{
 int w=0,flag=0;
 x--;
 while(x!= -1 &&str[x]!= '+'
&&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-
'&&str[x]!='/'&&str[x]!=':')
 {
  if(str[x]!='$'&& flag==0)
  {
```

```
  left[w++]=str[x];
  left[w]='\0';
  str[x]='$';
  flag=1;
  }
  x--;
 }
}
void fright(int x)
{
 int w=0,flag=0;
 x++;
 while(x!= -1 && str[x]!=
'+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-
'&&str[x]!='/')
 {
  if(str[x]!='$'&& flag==0)
  {
  right[w++]=str[x];
  right[w]='\0';
  str[x]='$';
  flag=1;
  }
  x++;
 }
}
```