**Compiler Design Lab 1**

**Implementation of lexical analyser**

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

#include <stdbool.h>

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

bool isDelimiter(char ch)

{

if (ch == ' ' || ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == ',' || ch == ';' || ch == '>' ||

ch == '<' || ch == '=' || ch == '(' || ch == ')' ||

ch == '[' || ch == ']' || ch == '{' || ch == '}')

return (true);

return (false);

}

bool isOperator(char ch)

{

if (ch == '+' || ch == '-' || ch == '\*' ||

ch == '/' || ch == '>' || ch == '<' ||

ch == '=')

return (true);

return (false);

}

bool validIdentifier(char\* str)

{

if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||

str[0] == '3' || str[0] == '4' || str[0] == '5' ||

str[0] == '6' || str[0] == '7' || str[0] == '8' ||

str[0] == '9' || isDelimiter(str[0]) == true)

return (false);

return (true);

}

bool isKeyword(char\* str)

{

if (!strcmp(str, "if") || !strcmp(str, "else") || !strcmp(str, "while") || !strcmp(str, "do") ||

!strcmp(str, "break") || !strcmp(str, "continue") || !strcmp(str, "int")

|| !strcmp(str, "double") || !strcmp(str, "float") || !strcmp(str, "return") || !strcmp(str, "char")

|| !strcmp(str, "case") || !strcmp(str, "char") || !strcmp(str, "sizeof") || !strcmp(str, "long")

|| !strcmp(str, "short") || !strcmp(str, "typedef") || !strcmp(str, "switch") || !strcmp(str, "unsigned")

|| !strcmp(str, "void") || !strcmp(str, "static") || !strcmp(str, "struct") || !strcmp(str, "goto"))

return (true);

return (false);

}

bool isInteger(char\* str)

{

int i, len = strlen(str);

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' || (str[i] == '-' && i > 0))

return (false);

}

return (true);

}

bool isRealNumber(char\* str)

{

int i, len = strlen(str);

bool hasDecimal = false;

if (len == 0)

return (false);

for (i = 0; i < len; i++) {

if (str[i] != '0' && str[i] != '1' && str[i] != '2'

&& str[i] != '3' && str[i] != '4' && str[i] != '5'

&& str[i] != '6' && str[i] != '7' && str[i] != '8'

&& str[i] != '9' && str[i] != '.' ||

(str[i] == '-' && i > 0))

return (false);

if (str[i] == '.')

hasDecimal = true;

}

return (hasDecimal);

}

char\* subString(char\* str, int left, int right)

{

int i;

char\* subStr = (char\*)malloc(

sizeof(char) \* (right - left + 2));

for (i = left; i <= right; i++)

subStr[i - left] = str[i];

subStr[right - left + 1] = '\0';

return (subStr);

}

void parse(char\* str)

{

int left = 0, right = 0;

int len = strlen(str);

while (right <= len && left <= right) {

if (isDelimiter(str[right]) == false)

right++;

if (isDelimiter(str[right]) == true && left == right) {

if (isOperator(str[right]) == true)

printf("'%c' IS AN OPERATOR\n", str[right]);

right++;

left = right;

} else if (isDelimiter(str[right]) == true && left != right

|| (right == len && left != right)) {

char\* subStr = subString(str, left, right - 1);

if (isKeyword(subStr) == true)

printf("'%s' IS A KEYWORD\n", subStr);

else if (isInteger(subStr) == true)

printf("'%s' IS AN INTEGER\n", subStr);

else if (isRealNumber(subStr) == true)

printf("'%s' IS A REAL NUMBER\n", subStr);

else if (validIdentifier(subStr) == true

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS A VALID IDENTIFIER\n", subStr);

else if (validIdentifier(subStr) == false

&& isDelimiter(str[right - 1]) == false)

printf("'%s' IS NOT A VALID IDENTIFIER\n", subStr);

left = right;

}

}

return;

}

// DRIVER FUNCTION

int main()

{

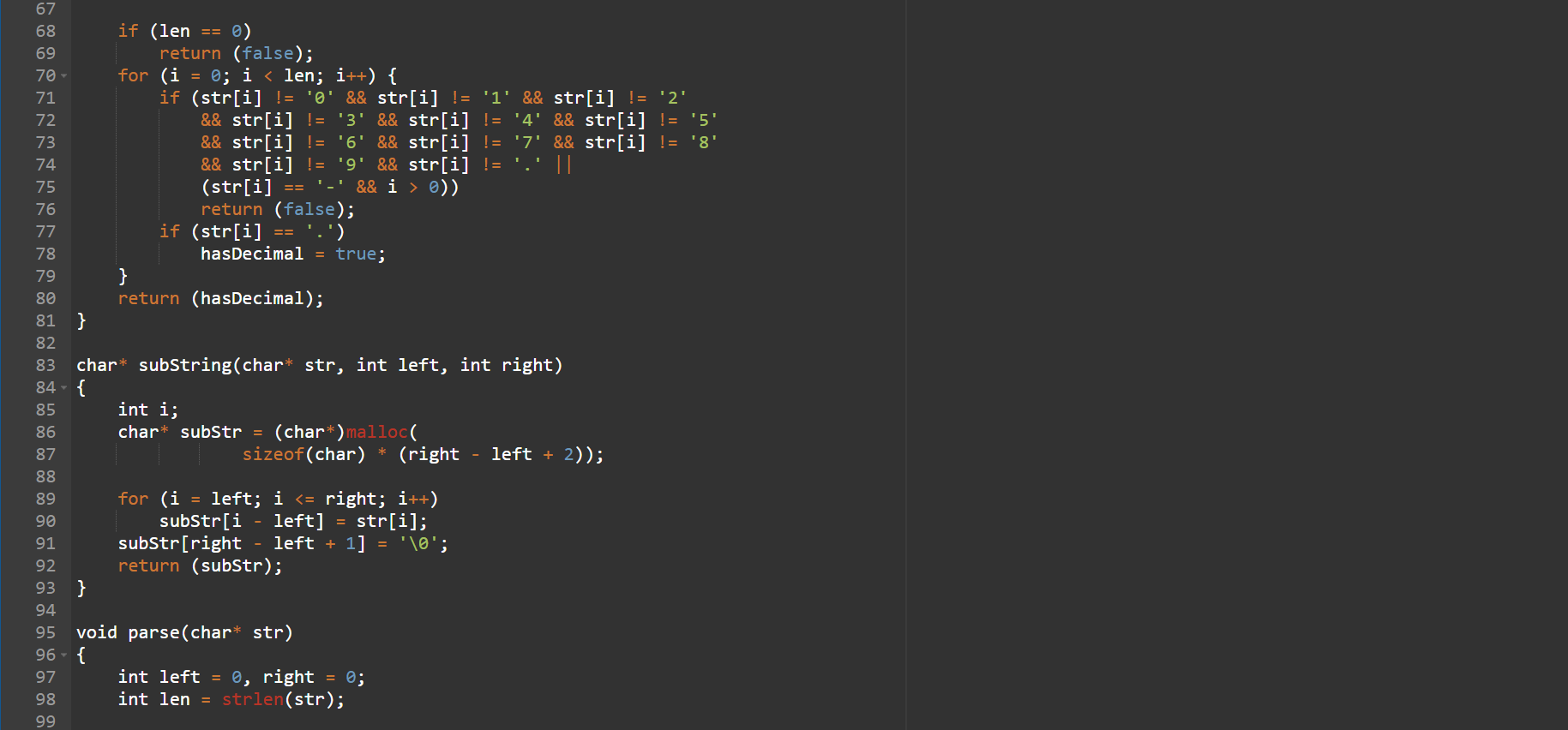
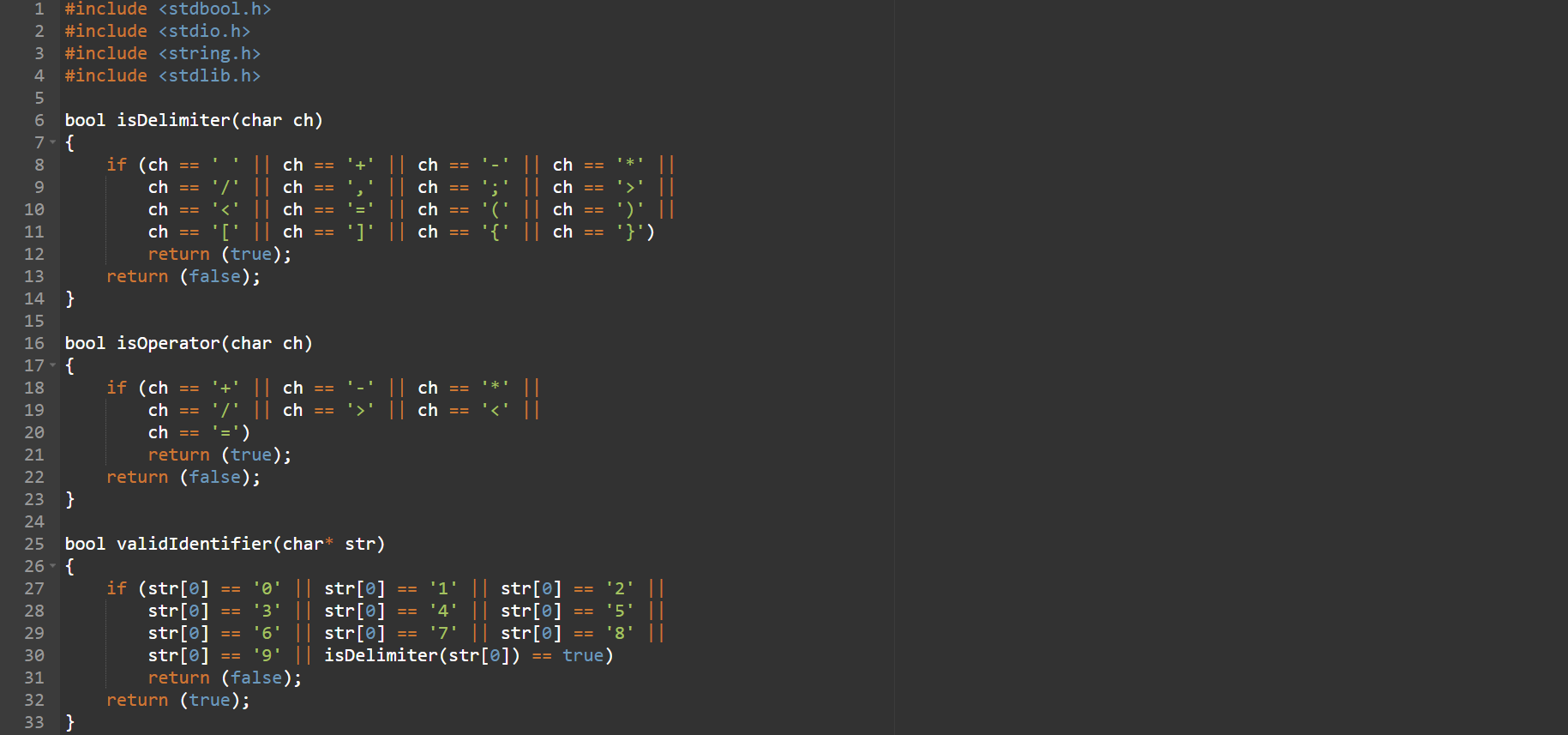
// maximum length of string is 100 here

char str[100] = "int x = a + 956 - 4z";

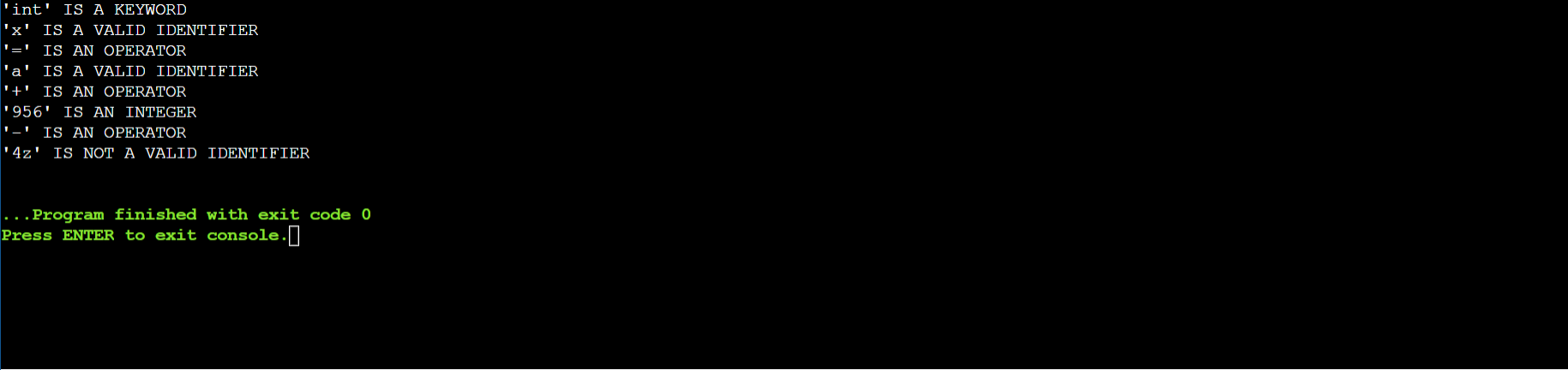
parse(str); // calling the parse function

return (0);

}



Output:



**Compiler Design Lab 2**

**Conversion from Regular Expression to NFA**

Name – Rohit Kothari

Reg no - RA1911003010350

Code:

#include <stdio.h>

#include <string.h>

int main()

{

char reg[20];

int q[20][3],i,j,len,a,b;

for(a=0;a<20;a++)

{

for(b=0;b<3;b++)

{

q[a][b]=0;

}

}

printf("Enter the regular expression: ");

scanf("%s",reg);

len = strlen(reg);

i=0;j=1;

while(i<len)

{

if(reg[i]=='a'&&reg[i+1]!='|'&&reg[i+1]!='\*')

{

q[j][0]=j+1;

j++;

}

if(reg[i]=='b'&&reg[i+1]!='|'&&reg[i+1]!='\*')

{

q[j][1]=j+1;

j++;

}

if(reg[i]=='e'&&reg[i+1]!='|'&&reg[i+1]!='\*')

{

q[j][2]=j+1;

j++;

}

if(reg[i]=='a'&&reg[i+1]=='|'&&reg[i+2]=='b')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][0]=j+1;

j++;

q[j][2]=j+3;

j++;

q[j][1]=j+1;

j++;

q[j][2]=j+1;

j++;

i=i+2;

}

if(reg[i]=='b'&&reg[i+1]=='|'&&reg[i+2]=='a')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][1]=j+1;

j++;

q[j][2]=j+3;

j++;

q[j][0]=j+1;

j++;

q[j][2]=j+1;

j++;

i=i+2;

}

if(reg[i]=='a'&&reg[i+1]=='\*')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][0]=j+1;

j++;

q[j][2]=((j+1)\*10)+(j-1);

j++;

}

if(reg[i]=='b'&&reg[i+1]=='\*')

{

q[j][2]=((j+1)\*10)+(j+3);

j++;

q[j][1]=j+1;

j++;

q[j][2]=((j+1)\*10)+(j-1);

j++;

}

if(reg[i]==')'&&reg[i+1]=='\*')

{

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

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

j++;

}

i++;

}

printf("Transition function \n");

for(i=0;i<=j;i++)

{

if(q[i][0]!=0)

printf("\n q[%d,a]-->%d",i,q[i][0]);

if(q[i][1]!=0)

printf("\n q[%d,b]-->%d",i,q[i][1]);

if(q[i][2]!=0)

{

if(q[i][2]<10)

printf("\n q[%d,e]-->%d",i,q[i][2]);

else

printf("\n q[%d,e]-->%d & %d",i,q[i][2]/10,q[i][2]%10);

}

}

return 0;

}

Output:

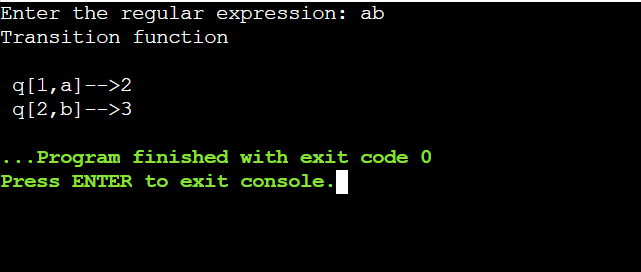
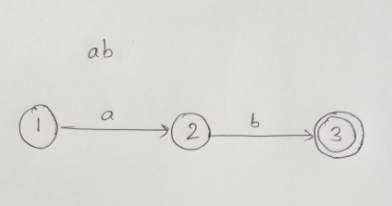


Diagram:



**Compiler Design Lab 3**

**Conversion from NFA to DFA**

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

import pandas as pd

# Taking NFA input from User

nfa = {}

n = int(input("No. of states : "))

t = int(input("No. of input states : "))

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:

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:

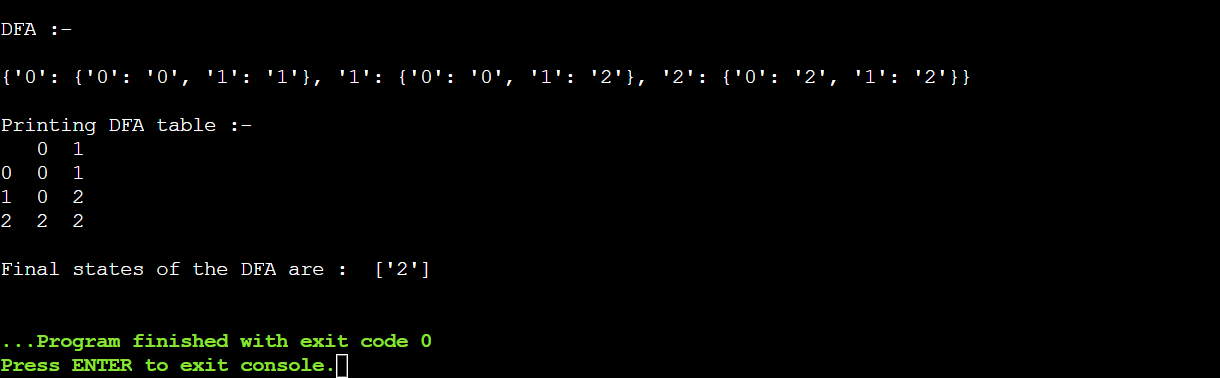
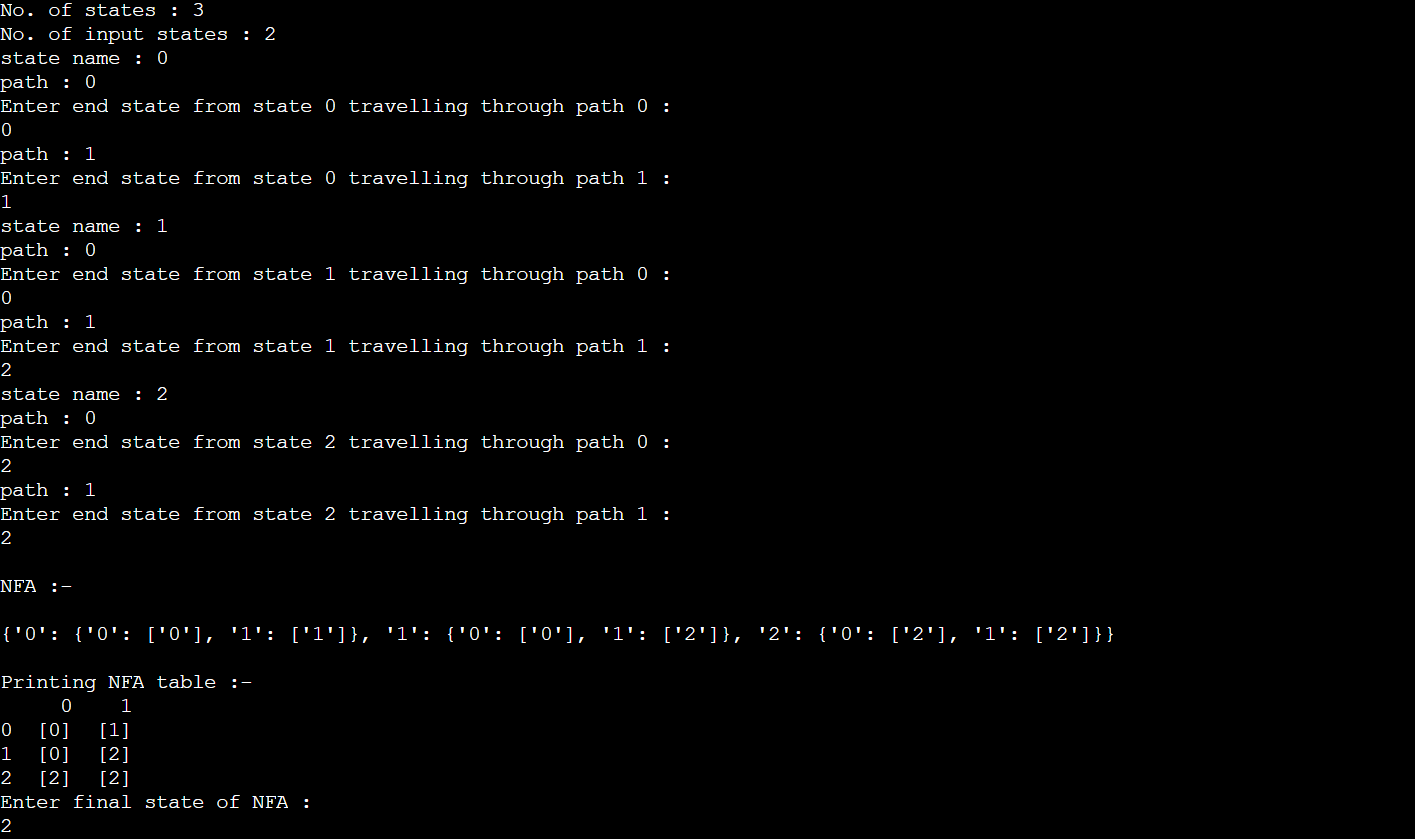
if i in nfa\_final\_state:

dfa\_final\_states.append(x)

break

print("\nFinal states of the DFA are : ",dfa\_final\_states)

Output:



**Compiler Design Lab 4**

Elimination of ambiguity, left factoring, and left recursion

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

Left recursion elimination:

#include <iostream>

#include <string>

using namespace std;

int main()

{

int n, j, l, i, k;

int length[10] = {};

string d, a, b, flag;

char c;

cout<<"Enter Parent Non-Terminal: ";

cin >> c;

d.push\_back(c);

a += d + "\'->";

d += "->";

b += d;

cout<<"Enter productions: ";

cin >> n;

for (int i = 0; i < n; i++)

{

cout<<"Enter Production ";

cout<<i + 1<<" :";

cin >> flag;

length[i] = flag.size();

d += flag;

if (i != n - 1)

{

d += "|";

}

}

cout<<"The Production Rule is: "; cout<<d<<endl;

for (i = 0, k = 3; i < n; i++)

{

if (d[0] != d[k])

{

cout<<"Production: "<< i + 1; cout<<" does not have left recursion."; cout<<endl;

if (d[k] == '#')

{

b.push\_back(d[0]);

b += "\'";

}

else

{

for (j = k; j < k + length[i]; j++) {

b.push\_back(d[j]);

}

k = j + 1;

b.push\_back(d[0]);

b += "\'|";

}

}

else

{

cout<<"Production: "<< i + 1 ;

cout<< " has left recursion";

cout<< endl;

if (d[k] != '#')

{

for (l = k + 1; l < k + length[i]; l++) {

a.push\_back(d[l]);

}

k = l + 1;

a.push\_back(d[0]);

a += "\'|";

}

}

}

a += "#";

cout << b << endl;

cout << a << endl;

return 0;

}

Left factoring elimination:

#include <iostream>

#include <string>

using namespace std;

int main()

{

int n,j,l,i,m;

int len[10] = {};

 string a, b1, b2, flag;

 char c;

 cout << "Enter the Parent Non-Terminal : ";  cin >> c;

 a.push\_back(c);

 b1 += a + "\'->";

 b2 += a + "\'\'->";;

 a += "->";

 cout << "Enter total number of productions : ";  cin >> n;

 for (i = 0; i < n; i++)

 {

 cout << "Enter the Production " << i + 1 << " : ";  cin >> flag;

 len[i] = flag.size();

 a += flag;

 if (i != n - 1)

 {

 a += "|";

}

 }

 cout << "The Production Rule is : " << a << endl;  char x = a[3];

 for (i = 0, m = 3; i < n; i++)

 {

 if (x != a[m])

 {

 while (a[m++] != '|');

 }

 else

 {

 if (a[m + 1] != '|')

 {

 b1 += "|" + a.substr(m + 1, len[i] - 1);  a.erase(m - 1, len[i] + 1);

 }

 else

 {

 b1 += "#";

 a.insert(m + 1, 1, a[0]);

 a.insert(m + 2, 1, '\'');

 m += 4;

 }

 }

 }

 char y = b1[6];

 for (i = 0, m = 6; i < n - 1; i++)

 {

 if (y == b1[m])

 {

 if (b1[m + 1] != '|')

 {

 flag.clear();

 for (int s = m + 1; s < b1.length(); s++)  {

 flag.push\_back(b1[s]);

}

 b2 += "|" + flag;

 b1.erase(m - 1, flag.length() + 2);  }

 else

 {

 b1.insert(m + 1, 1, b1[0]);  b1.insert(m + 2, 2, '\'');

 b2 += "#";

 m += 5;

 }

 }

 }

 b2.erase(b2.size() - 1);

 cout << "After Left Factoring : " << endl;  cout << a << endl;

 cout << b1 << endl;

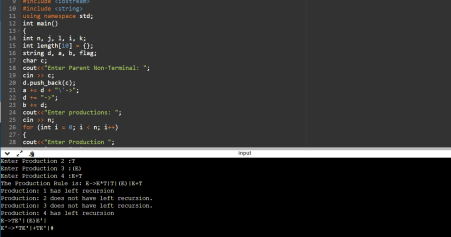
 cout << b2 << endl;

 return 0;

}

Output:

Left recursion elimination:

****

Left factoring elimination:

****

**Compiler Design Lab 5**

First and Follow Computation

Name – Rohit Kothari

Reg no. – RA1911003010350

**Code:**

#include<stdio.h>

#include<string.h>

#include<ctype.h>

int n,m=0,p,i=0,j=0;

char a[10][10],f[10];

void follow(char c);

void first(char c);

int main()

{

int i,z;

char c,ch;

printf("Enter the number of productions:\n");

scanf("%d",&n);

printf("Enter the productions:\n");

for(i=0;i<n;i++)

scanf("%s%c",a[i],&ch);

do

{

m=0;

printf("Enter the elements whose first & follow is to be found:");

scanf("%c",&c);

first(c);

printf("First(%c)={",c);

for(i=0;i<m;i++)

printf("%c",f[i]);

printf("}\n");

strcpy(f," ");

m=0;

follow(c);

printf("Follow(%c)={",c);

for(i=0;i<m;i++)

printf("%c",f[i]);

printf("}\n");

printf("Continue(0/1)?");

scanf("%d%c",&z,&ch);

}

while(z==1);

return(0);

}

void first(char c)

{

int k;

if(!isupper(c))

f[m++]=c;

for(k=0;k<n;k++)

{

if(a[k][0]==c)

{

if(a[k][2]=='$')

follow(a[k][0]);

else if(islower(a[k][2]))

f[m++]=a[k][2];

else first(a[k][2]);

}

}

}

void follow(char c)

{

if(a[0][0]==c)

f[m++]='$';

for(i=0;i<n;i++){

for(j=2;j<strlen(a[i]);j++)

{

if(a[i][j]==c)

{

if(a[i][j+1]!='\0')

first(a[i][j+1]);

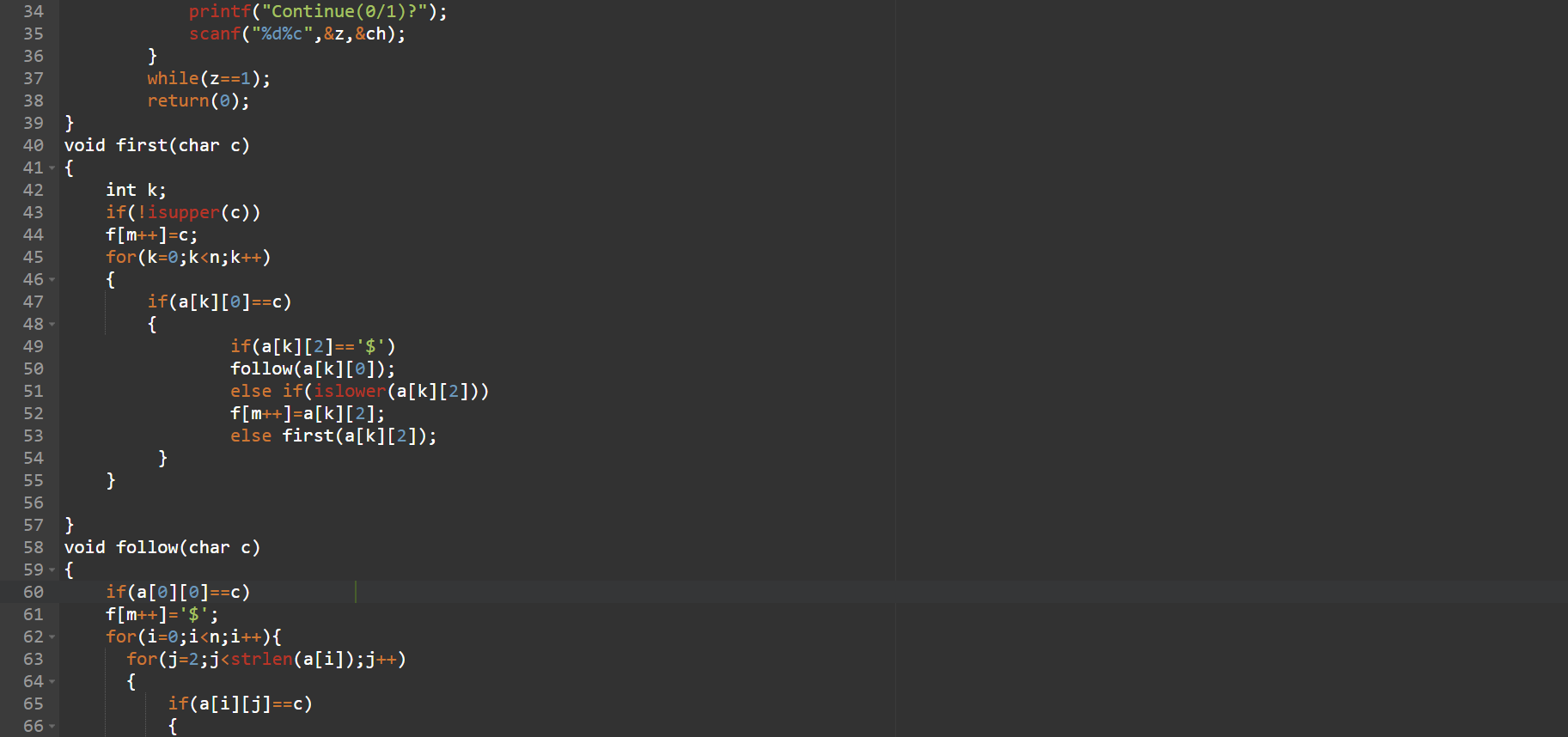
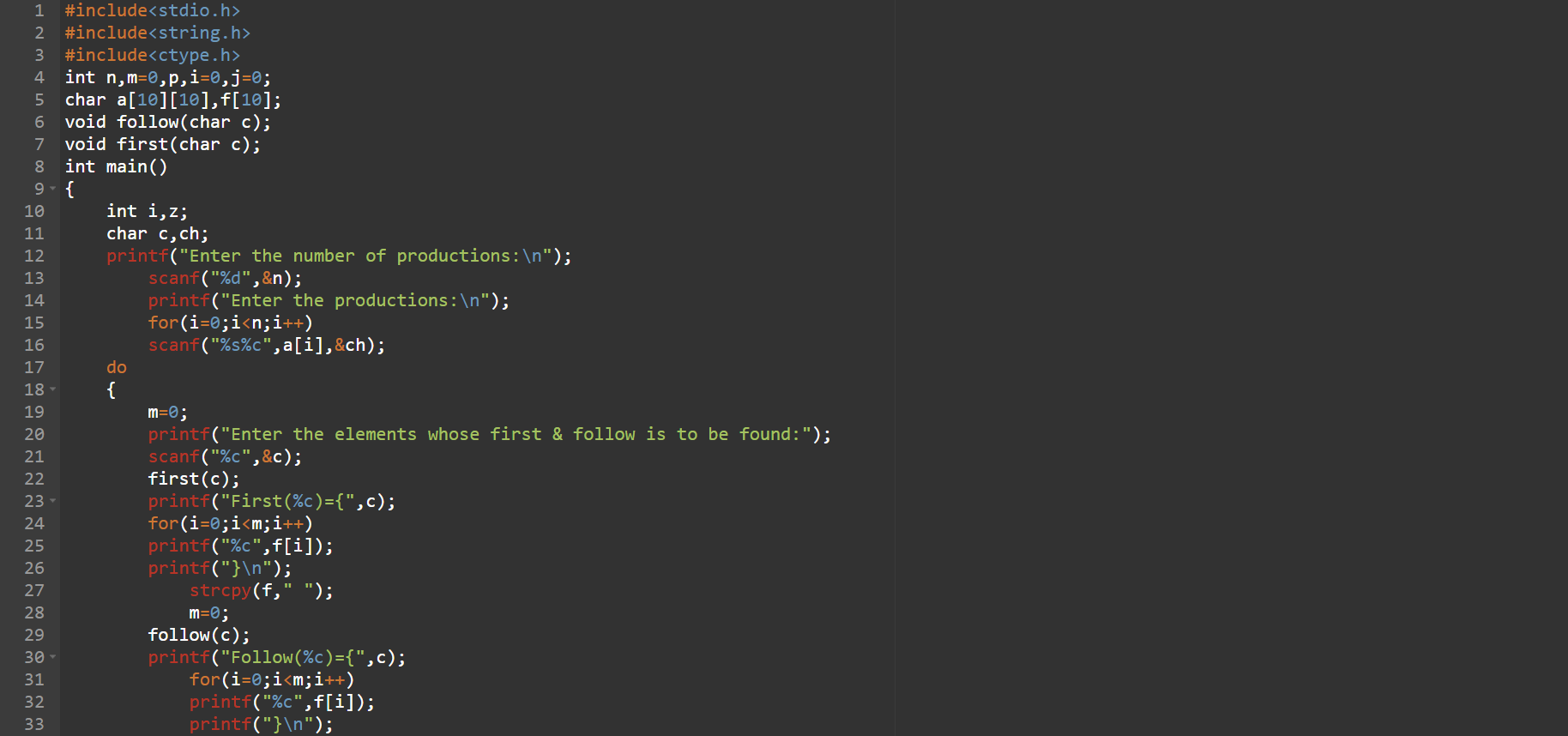
if(a[i][j+1]=='\0' && c!=a[i][0])

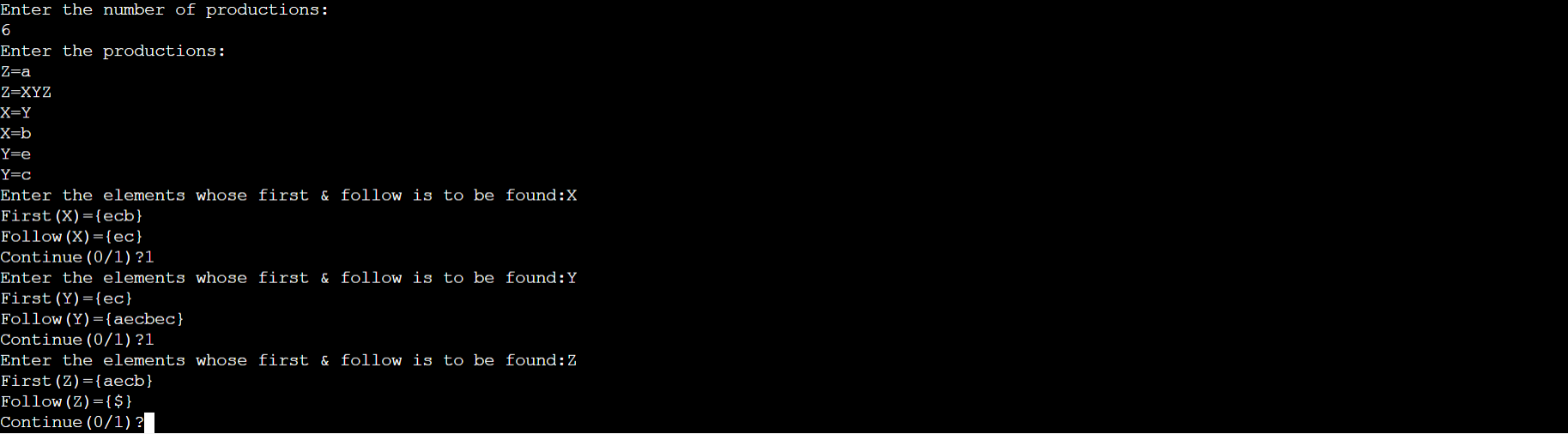
follow(a[i][0]);

}

} } }

**Output:**





**Compiler Design Lab 6**

Construct predictive parsing table

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

n =int(input()) # Number of production

non\_terminals = []

production=[]

product1=[]

print("Enter the non terminals :")

for i in range(n):

non\_terminals.append(input())

print("Enter production ")

for i in range(n):

print("1st")

production.append(input())

print("2nd")

product1.append(input())

lo=list (zip(production,product1))

gram = dict(zip(non\_terminals, lo))

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

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])

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+=["$"]

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}'

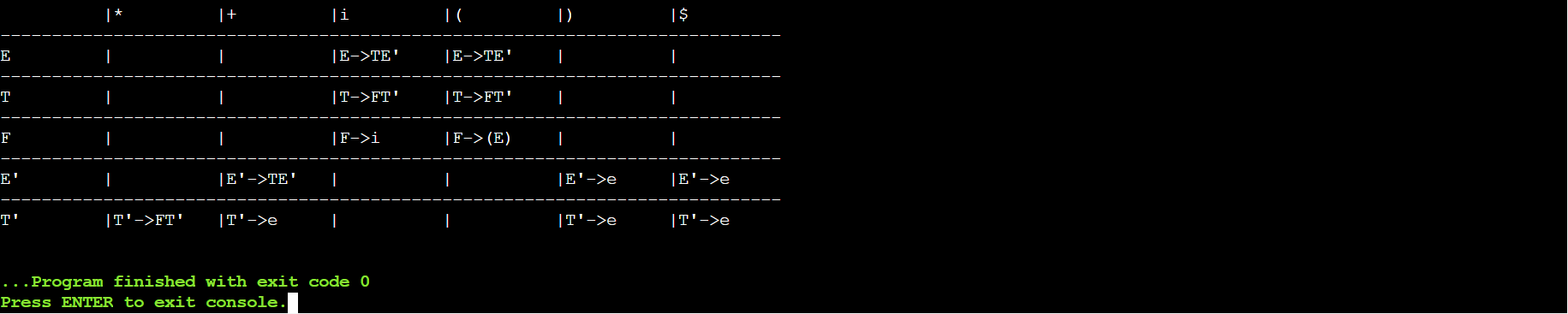
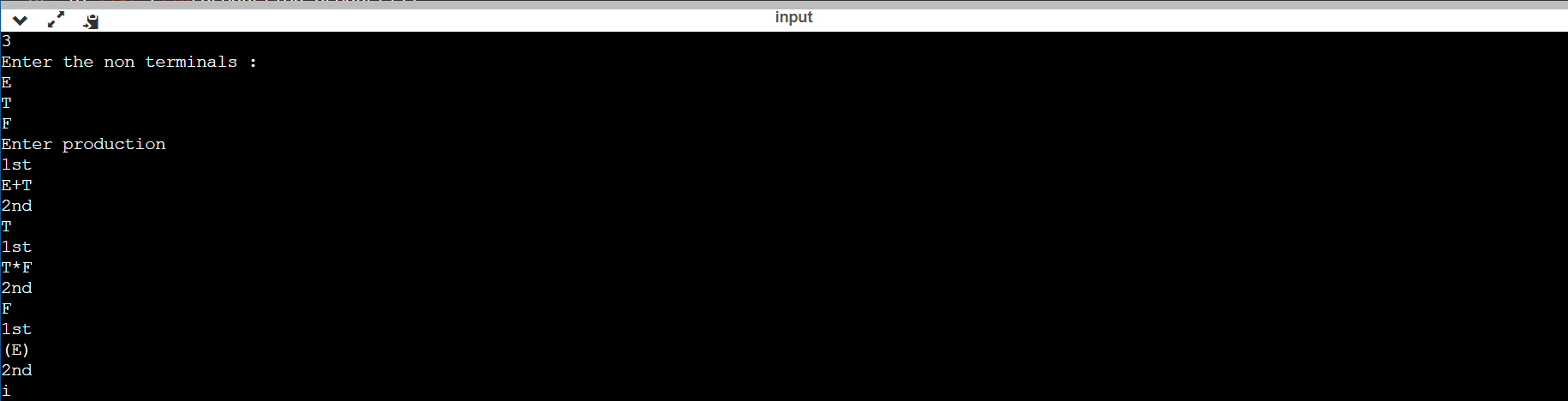
else:

toprint+=f'|{pptable[(i,j)]: <10}'

print(f'{"-":-<76}')

print(toprint)

Output:



**C.D. Lab 7**

Shift Reduce Parsing

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

non\_terminals = []

production=[]

product1=[]

product2=[]

print("Enter the non terminals :")

non\_terminals.append(input())

print("1st")

production.append(input())

print("2nd")

product1.append(input())

print("3rd")

product2.append(input())

lo =[]

lo.append(production)

lo.append(product1)

lo.append(product2)

lo

l= list(zip( lo[0],lo[1],lo[2]))

gram = dict(zip(non\_terminals,l))

print(gram)

starting\_terminal =non\_terminals[0]

print("starting-terminal "+ str(starting\_terminal))

print("input string")

inp=input()

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')

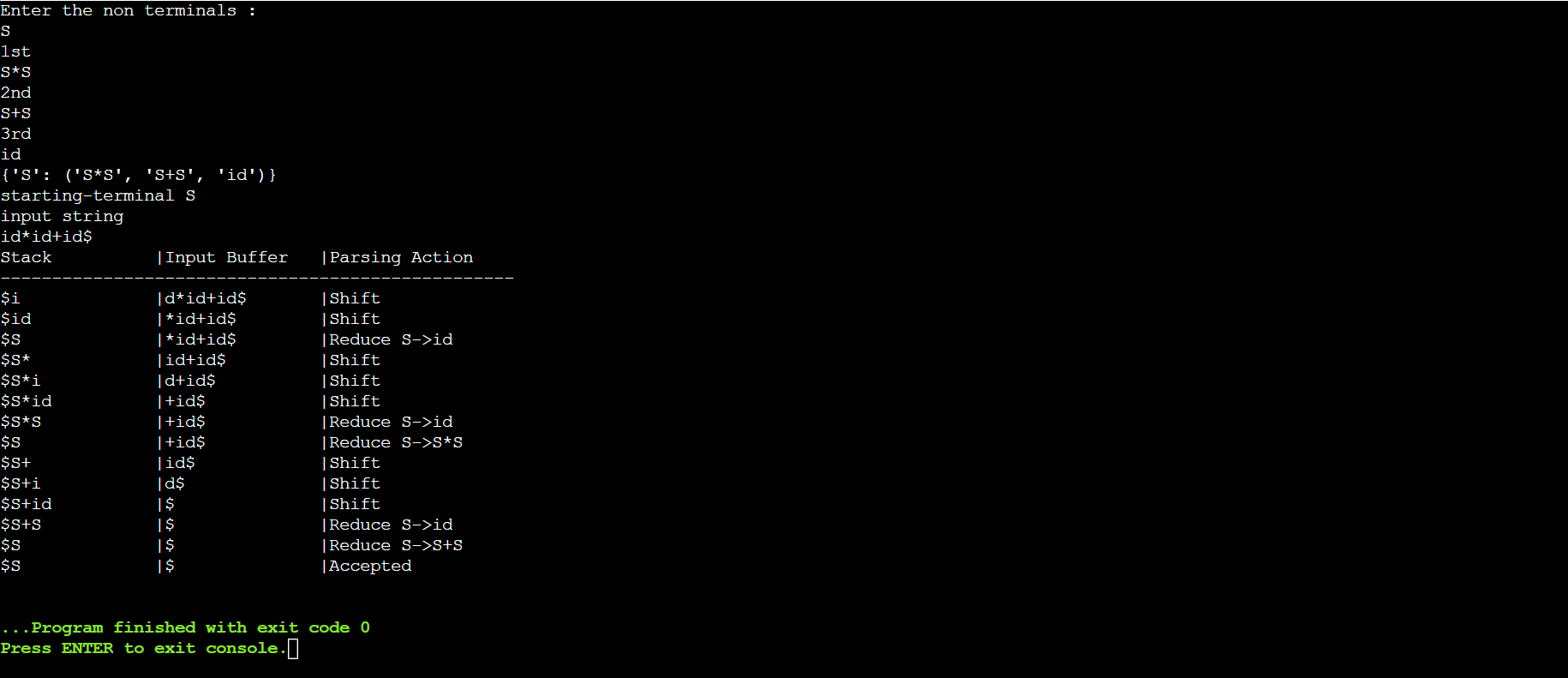
break

if action:

print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Rejected')

break

Output:



**Compiler Design Lab 8**

Leading and Trailing

Name – Rohit Kothari

Reg no. – RA1911003010350

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,count,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<<"\n------------- LEADING 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<<"\n------------- PRODUCTION 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++)

{

count=0;

while(gram[i].rhs[j][count]!='\x0')

count++;

for(k=0;k<terms;k++)

{

if(gram[i].rhs[j][count-1]==term[k])

trail[i][k]=1;

else

{

if(gram[i].rhs[j][count-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++)

{

count=0;

while(gram[i].rhs[j][count]!='\x0')

count++;

for(m=1;m<vars;m++)

{

if(gram[i].rhs[j][count-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();

getch();

}

Output:



**Compiler Design Lab 9**

Computation of LR(0) items

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

char items[30][100][100];

char augmented\_grammar[100][100], terminals[10], nonterminals[10];

int no\_of\_productions = 0, no\_of\_states = 0, no\_of\_items[30], no\_of\_terminals = 0,

no\_of\_nonterminals = 0;

char FIRST[2][10][10];

char FOLLOW[10][10];

int state\_index = 0, goto\_state\_index = 0, closure\_item\_index = 0;

int check(char c)

{

int i;

for(i = 0; i < no\_of\_terminals; i++)

if(terminals[i] == c)

return 1;

return 0;

}

void generate\_terminals()

{

int i, j;

int index = 0;

for(i = 0; i < no\_of\_productions; i++)

{

for(j = 0; augmented\_grammar[i][j] != '>'; j++);

j++;

for(; augmented\_grammar[i][j] != '\0'; j++)

{

if(augmented\_grammar[i][j] < 65 || augmented\_grammar[i][j] > 90)

{

if(!check(augmented\_grammar[i][j]))

{

terminals[index] = augmented\_grammar[i][j];

no\_of\_terminals++;

index++;

}

}

}

}

terminals[index] = '$';

no\_of\_terminals++;

index++;

terminals[index] = '\0';

}

int check2(char c, int index)

{

int i;

for(i = 0; i < index; i++)

if(nonterminals[i] == c)

return 1;

return 0;

}

void generate\_nonterminals()

{

int i, index = 0;

for(i = 0; i < no\_of\_productions; i++)

if(!check2(augmented\_grammar[i][0], index))

{

nonterminals[index] = augmented\_grammar[i][0];

index++;

}

no\_of\_nonterminals = index;

nonterminals[index] = '\0';

}

void initialize\_items()

{

generate\_terminals();

generate\_nonterminals();

int i;

for(i = 0; i < 30; i++)

no\_of\_items[i] = 0;

}

void generate\_item(char \*s, char \*t)

{

int i;

for(i = 0; i < 3; i++)

t[i] = s[i];

t[i] = '.';

if(s[i] != '@')

for(; i < strlen(s); i++)

t[i+1] = s[i];

t[i+1] = '\0';

}

int item\_found(char \*s)

{

int i;

for(i = 0; i < closure\_item\_index; i++)

{

if(!strcmp(s, items[state\_index][i]))

return 1;

}

return 0;

}

int isterminal(char s)

{

int i;

for(i = 0; i < no\_of\_terminals; i++)

if(s == terminals[i])

return 1;

return 0;

}

void closure(char \*s)

{

int i, j;

for(i = 0; s[i] != '.'; i++);

i++;

if(!item\_found(s))

{

strcpy(items[state\_index][closure\_item\_index], s);

closure\_item\_index++;

}

if(s[i] == s[0] && s[i-2] == '>')

return;

if(isterminal(s[i]))

return;

else

{

for(j = 0; j < no\_of\_productions; j++)

{

char temp[100];

if(augmented\_grammar[j][0] == s[i])

{

generate\_item(augmented\_grammar[j], temp);

closure(temp);

}

}

}

}

int Goto1(char s, char temp[][100])

{

int i, j;

int n = 0;

char t, temp2[100];

if(s == '\0')

{

return n;

}

for(i = 0; i < no\_of\_items[goto\_state\_index]; i++)

{

strcpy(temp2, items[goto\_state\_index][i]);

for(j = 0; temp2[j] != '.'; j++);

if(temp2[j+1] == '\0')

continue;

if(temp2[j+1] == s)

{

t = temp2[j];

temp2[j] = temp2[j+1];

temp2[j+1] = t;

strcpy(temp[n], temp2);

n++;

}

}

return n;

}

int state\_found(char \*s)

{

int i;

for(i = 0; i < state\_index; i++)

{

if(!strcmp(s, items[i][0]))

return 1;

}

return 0;

}

int transition\_item\_found(char \* t\_items, char s, int t\_index)

{

int i;

for(i = 0; i < t\_index; i++)

if(s == t\_items[i])

return 1;

return 0;

}

void compute\_closure\_goto()

{

char temp[100][100], transition\_items[100];

int i, no\_of\_goto\_items,j, transition\_index = 0;

generate\_item(augmented\_grammar[0], temp[0]);

closure(temp[0]);

no\_of\_items[state\_index] = closure\_item\_index;

closure\_item\_index = 0;

state\_index++;

while(goto\_state\_index < 30)

{

transition\_index = 0;

transition\_items[transition\_index] = '\0';

for(i = 0; i < no\_of\_items[goto\_state\_index]; i++)

{

for(j = 0; items[goto\_state\_index][i][j] != '.'; j++);

j++;

if(!transition\_item\_found(transition\_items,items[goto\_state\_index][i][j], transition\_index))

{

transition\_items[transition\_index] = items[goto\_state\_index][i][j];

transition\_index++;

}

}

transition\_items[transition\_index] = '\0';

for(i = 0; i < transition\_index; i++)

{

int add\_flag = 0;

no\_of\_goto\_items = Goto1(transition\_items[i], temp);

for(j = 0; j < no\_of\_goto\_items; j++)

{

if(!state\_found(temp[j]))

{

add\_flag = 1;

closure(temp[j]);

}

else

break;

}

if(add\_flag)

{

no\_of\_items[state\_index] = closure\_item\_index;

closure\_item\_index = 0;

state\_index++;

}

}

goto\_state\_index++;

}

no\_of\_states = state\_index;

}

void print()

{

int i, j;

printf("\nNumber of states = %d.\n", no\_of\_states);

for(i = 0; i < no\_of\_states; i++)

{

printf("\n\nItems in State %d...\n\n", i);

for(j = 0; j < no\_of\_items[i]; j++)

printf("%s\n", items[i][j]);

}

}

void start()

{

char str[100];

printf("Enter number of productions:");

scanf("%d", &no\_of\_productions);

printf("Enter the productions...\n");

int i;

for(i = 1; i <= no\_of\_productions; i++)

scanf("%s", augmented\_grammar[i]);

printf("\n\nAugmented Grammar is...\n\n");

strcpy(augmented\_grammar[0], "Z->");

str[0] = augmented\_grammar[1][0];

str[1] = '\0';

strcat(augmented\_grammar[0], str);

no\_of\_productions++;

for(i = 0; i < no\_of\_productions; i++)

printf("%s\n", augmented\_grammar[i]);

initialize\_items();

compute\_closure\_goto();

print();

}

int main()

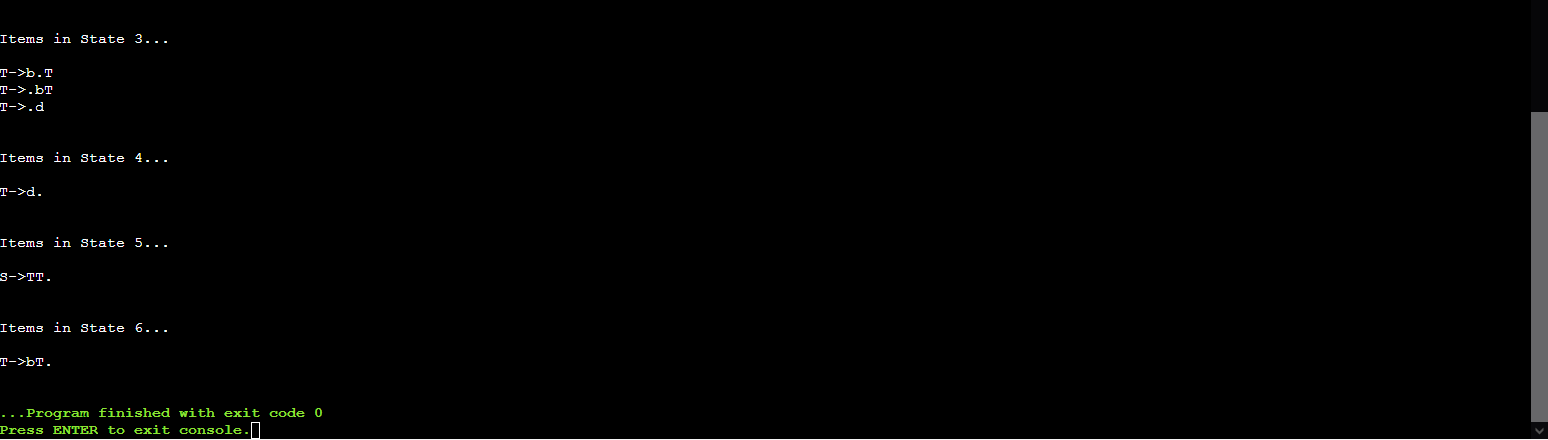
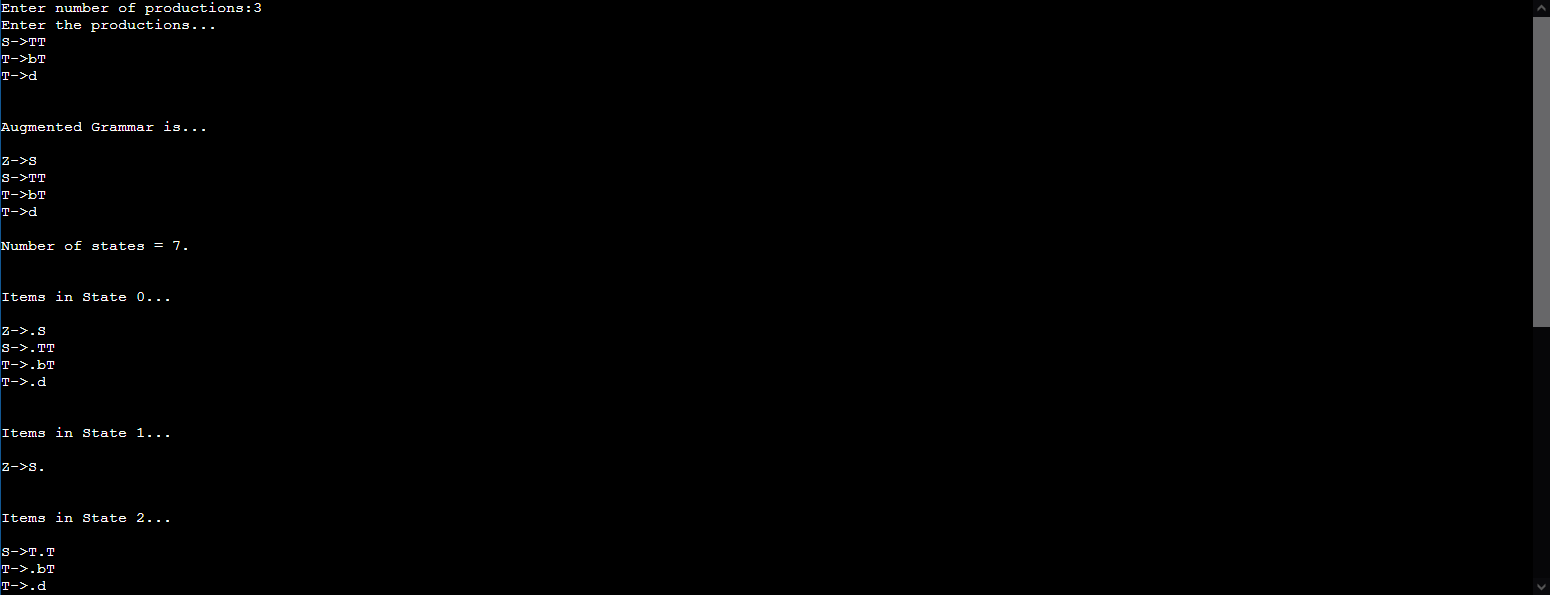
{

start();

return 0;

}

Output:



**Compiler Design Lab 10**

Intermediate code generation – Postfix, Prefix

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

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

PRI = {'+':1, '-':1, '\*':2, '/':2}

### INFIX ===> POSTFIX ###

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]

### THREE ADDRESS CODE GENERATION ###

def generate3AC(pos):

print("### THREE ADDRESS CODE GENERATION ###")

exp\_stack = []

t = 1

for i in pos:

if i not in OPERATORS:

exp\_stack.append(i)

else:

print(f't{t} := {exp\_stack[-2]} {i} {exp\_stack[-1]}')

exp\_stack=exp\_stack[:-2]

exp\_stack.append(f't{t}')

t+=1

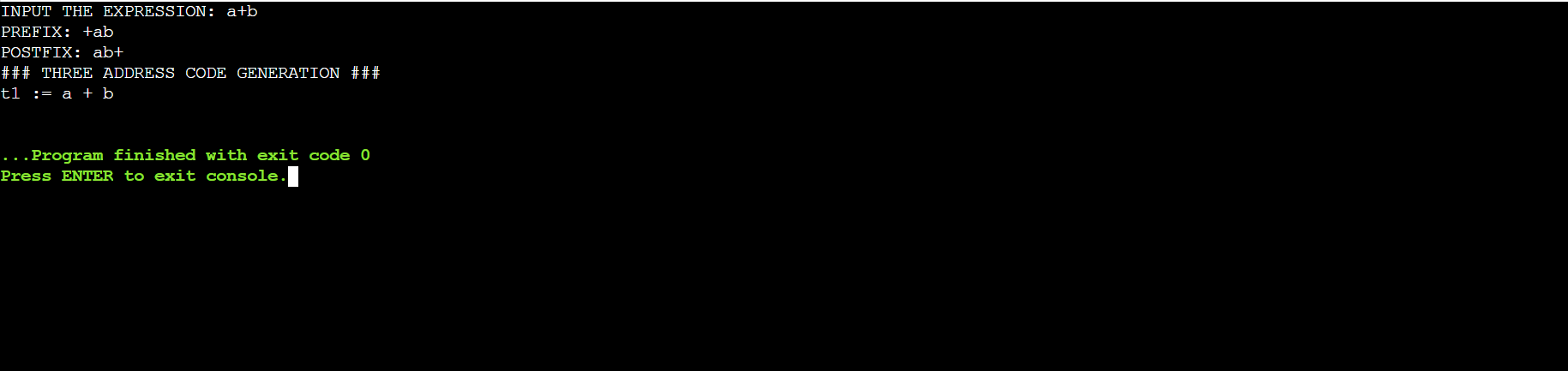
expres = input("INPUT THE EXPRESSION: ")

pre = infix\_to\_prefix(expres)

pos = infix\_to\_postfix(expres)

generate3AC(pos)

Output:



**Compiler Design Lab 11**

Intermediate code generation – Quadruple, Triple, Indirect Triple

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

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

PRI = {'+':1, '-':1, '\*':2, '/':2}

### INFIX ===> POSTFIX ###

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]

### THREE ADDRESS CODE GENERATION ###

def generate3AC(pos):

print("### THREE ADDRESS CODE GENERATION ###")

exp\_stack = []

t = 1

for i in pos:

if i not in OPERATORS:

exp\_stack.append(i)

else:

print(f't{t} := {exp\_stack[-2]} {i} {exp\_stack[-1]}')

exp\_stack=exp\_stack[:-2]

exp\_stack.append(f't{t}')

t+=1

expres = input("INPUT THE EXPRESSION: ")

pre = infix\_to\_prefix(expres)

pos = infix\_to\_postfix(expres)

generate3AC(pos)

def Quadruple(pos):

stack = []

op = []

x = 1

for i in pos:

if i not in OPERATORS:

stack.append(i)

elif i == '-':

op1 = stack.pop()

stack.append("t(%s)" %x)

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op1,"(-)"," t(%s)" %x))

x = x+1

if stack != []:

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format("+",op1,op2," t(%s)" %x))

stack.append("t(%s)" %x)

x = x+1

elif i == '=':

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,"(-)",op1))

else:

op1 = stack.pop()

op2 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}|{3:4s}".format(i,op2,op1," t(%s)" %x))

stack.append("t(%s)" %x)

x = x+1

print("The quadruple for the expression ")

print(" OP | ARG 1 |ARG 2 |RESULT ")

Quadruple(pos)

def Triple(pos):

stack = []

op = []

x = 0

for i in pos:

if i not in OPERATORS:

stack.append(i)

elif i == '-':

op1 = stack.pop()

stack.append("(%s)" %x)

print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,"(-)"))

x = x+1

if stack != []:

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}".format("+",op1,op2))

stack.append("(%s)" %x)

x = x+1

elif i == '=':

op2 = stack.pop()

op1 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op1,op2))

else:

op1 = stack.pop()

if stack != []:

op2 = stack.pop()

print("{0:^4s} | {1:^4s} | {2:^4s}".format(i,op2,op1))

stack.append("(%s)" %x)

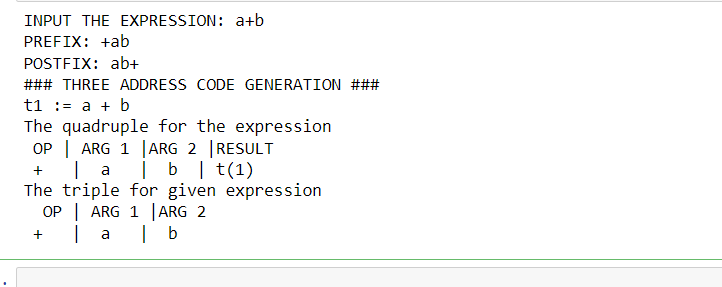
x = x+1

print("The triple for given expression")

print(" OP | ARG 1 |ARG 2 ")

Triple(pos)

Output:



**Compiler Design Lab 12**

A Simple Code Generator

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

import pandas as pd

import copy

try:

    a=pd.read\_csv("WEEK12.txt")

    print("Input given in the txt file is in following manner: ")

    print(a)

    print('\nSimple Code Generated :\n')

    c=a.shape# It will gives an tuple of numbers of rows and columns

    #print(c)

    l=[]

    o=list("+-\*/")#If you want to add more operator youn can use that as well

    o1=[]

    r=[]

    for i in range(c[0]):# Here c[0] is 0th element of tuple c, which is a.shape (c=a.shape)

        l=l+[a['left'][i]]

        d=a['right'][i]

        x=d.split()

        l=l+x

    #print(l)

    sizel=len(l)

    for z in range(sizel):

        #print(sizel)

        if(l[z] in o):

            o1=o1+[l[z]]

    o1=list(set(o1))

    #print(o1)

    li=copy.deepcopy(l)# if you use li=l then it may occures some un usual error further in program.

    for x in o1:

        if(x in li):

            li.remove(x)

    li=list(set(li))

    #print(li)

    for b in range(len(li)):

        r=r+["R"+str(b)]

    #print(r)

    i=1

    ak=0

    z=0

    ACounter=0

    akm=[]

    while(i):

        if(ak==len(l)):

            i=0

        elif(l[ak].isalpha() and l[ak]==a['left'][z]):

            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))

            akm=akm+[r[li.index(l[ak])]]

            ak+=1

        elif(((l[ak].isalpha()) and (l[ak] in a['right'][z]))and (l[ak] not in o1)):

            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))

            akm=akm+[r[li.index(l[ak])]]

            ak+=1

            ACounter+=1

            if((len(a['right'][z])==1)and (len(akm)==2)):

                   print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

               #print(akm)

                   akm.clear()

                   z+=1

            print("\t")

        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="+")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("ADD "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

            #print(akm)

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

            #print(ak)

            #print(ACounter)

            ak+=2

            ACounter+=2

            #print(ACounter)

            if(len(a['right'][z].split(" "))==ACounter):

                   #print(akm)

                    akm.clear()

                    z+=1

                    ACounter=0

                    #print(z)

                    print("\t")

        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="-")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("SUB "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

            ak+=2

            ACounter+=2

            #print(ACounter)

            if(len(a['right'][z].split(" "))==ACounter):

                   #print(akm)

                    akm.clear()

                    z+=1

                    ACounter=0

                   #print(z)

                    print("\t")

        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="\*")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("MUL "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

            ak+=2

            ACounter+=2

            #print(ACounter)

            if(len(a['right'][z].split(" "))==ACounter):

                   #print(akm)

                    akm.clear()

                    z+=1

                    ACounter=0

                   #print(z)

                    print("\t")

        elif((l[ak] in a['right'][z]) and ((l[ak]in o1)and l[ak]=="/")):

            print("MOV "+str(l[ak+1])+' , '+str(r[li.index(l[ak+1])]))

            akm=akm+[r[li.index(l[ak+1])]]

            print("DIV "+str(akm[len(akm)-2])+' , '+str(akm[len(akm)-1]))

            akm.pop(len(akm)-2)

            print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

            akm.clear()

            ak+=2

            ACounter+=2

            if(len(a['right'][z].split(" "))==ACounter):

                   #print(akm)

                    akm.clear()

                    z+=1

                    ACounter=0

                   #print(z)

                    print("\t")

        elif((l[ak].isnumeric())and(l[ak] in a['right'][z])):

            print("MOV "+str(l[ak])+' , '+str(r[li.index(l[ak])]))

            akm=akm+[r[li.index(l[ak])]]

            ak+=1

            ACounter+=1

            if((len(akm)==2)and (a['right'][z]==l[ak-1])):

                print("STOR "+str(akm[len(akm)-1])+' , '+str(akm[0]))

                akm.clear()

                z+=1

                ACounter=0

               #print(z)

                print("\t")

        elif((l[ak] not in o1)or (l[ak] not in string.ascii\_lowercase)):

            print("\f Error!\n\f Please enter valid syntax for three address code.\n\f Check your csv file...")

            print(f"\f Error description...\nError in line number {z} and place number {ak}.")

            print(f"\f Error element is {a['right'][z]}.")

            break

except (FileNotFoundError):

    print("Please check you input file. It may possible that file doesn't exist.")

    print("Also check the file name that is given in input section at the starting place.")

except(ArithmeticError):

    print("An arithmetic error is caused due to which program is not proceed futher.Please check for the solution.")

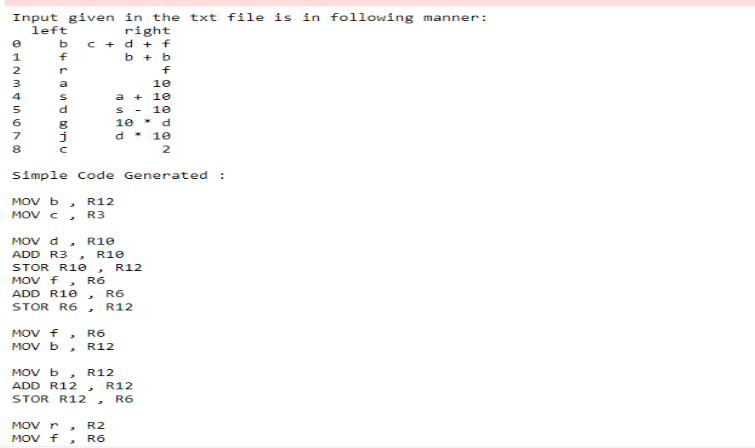
except(IndexError):

    print("List index out of range.")

except:

    print("An exceptions occurred.")

Output:



**Compiler Design Lab 13**

Implement any one storage allocation strategies

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

stack = []

# append() function to push element in the stack

print("Enter Number/Alphabets for the Stack to push : ")

for x in range(0,5):

    d=input()

    stack.append(d)

#stack.append('b')

print('\nInitial stack')

print(stack)

# pop() function to pop element from stack in LIFO order

print('\nElements popped from stack:')

print(stack.pop())

print(stack.pop())

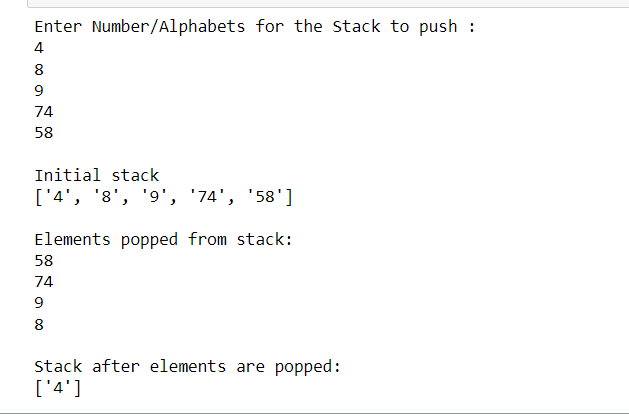
print(stack.pop())

print(stack.pop())

print('\nStack after elements are popped:')

print(stack)

Output:



**Compiler Design Lab 14**

Implementation of DAG

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

import networkx as nx

from matplotlib import pyplot as plt

class DAG:

def \_\_init\_\_(self):

self.graph=nx.DiGraph()

def addEdges(self,edges):

"""Function to add one edge at a time and check if the graph is acyclic post insertion"""

self.graph.add\_edge(edges)

if nx.is\_directed\_acyclic\_graph(self.graph):

pass

else:

raise "Unable to insert "+str(edges)+"This is an Acyclic graph"

self.graph.remove\_edge(edges)

def AddSetofEdges(self,listt):

"""Function to all a list of edges and check is the graph is an DAG for furthur details refer networkx"""

self.graph.add\_edges\_from(listt)

if nx.is\_directed\_acyclic\_graph(self.graph):

pass

else:

raise "This is an acyclic graph check your edges"

self.graph.remove\_edge(listt)

def Visualise(self,location="home"):

"""It uses Matplotlib to visualise the DAG .

The graph is stored in a PNG format . So name the file accourdingly

eg

>>> DAG.Visualise(home/img.png)"""

if self.graph==None:

return "There is no graph consider adding edges to visualise"

plt.tight\_layout()

nx.draw\_networkx(self.graph,arrows=True,node\_size=800)

plt.savefig(location,format="PNG")

plt.clf()

return "Graph generated"

graph = DAG()

graph.AddSetofEdges([("root", "a"), ("a", "b"), ("a", "e"), ("b", "c"), ("b", "d"), ("d", "f"),("c","f")])

graph.Visualise("graph.png")

from PIL import Image # to load images

from IPython.display import display

print("DAG image")

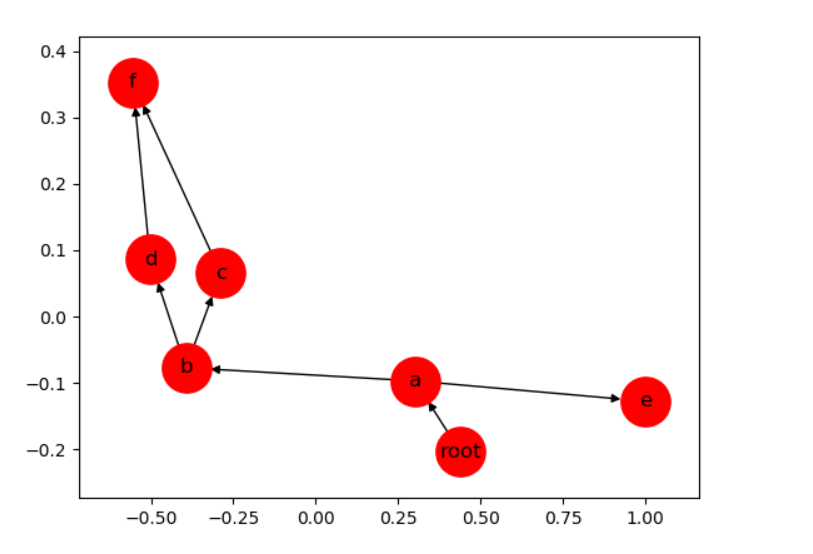
print("")

print("")

pil\_im = Image.open('graph.png')

display(pil\_im)

Output:



**Compiler Design Lab 15**

Implementation of Global Data Flow Analysis

Name – Rohit Kothari

Reg no. – RA1911003010350

Code:

#include <stdio.h>

#include <conio.h>

#include <string.h>

struct op

{

char l[20];

char r[20];

} op[10], pr[10];

int main()

{

int a, i, k, j, n, z = 0, m, q, lineno = 1;

char \*p, \*l;

char temp, t;

char \*tem;

char \*match;

printf("enter no of values");

scanf("%d", &n);

for (i = 0; i < n; i++)

{

printf("\tleft\t");

scanf("%s", op[i].l);

printf("\tright:\t");

scanf("%s", op[i].r);

}

printf("intermediate Code\n");

for (i = 0; i < n; i++)

{

printf("Line No=%d\n", lineno);

printf("\t\t\t%s=", op[i].l);

printf("%s\n", op[i].r);

lineno++;

}

printf("\*\*\*Data Flow Analysis for the Above Code \*\*\*\n");

for (i = 0; i < n; i++)

{

for (j = 0; j < n; j++)

{

match = strstr(op[j].r, op[i].l);

if (match)

{

printf("\n %s is live at %s \n ", op[i].l, op[j].r);

}

}

}

return 0;

}

Output:

