# Exp -1 - Implementation of Lexical Analyzer

# Code in C++ :

#include<iostream> #include<cstring> #include<ctype.h> #include<fstream> using namespace std;

string arr[] = { "void", "using", "namespace", "int", "include", "iostream", "std", "main",

"cin", "cout", "return", "float", "double", "string" };

bool

isKeyword (string a)

{

for (int i = 0; i < 14; i++)

{

if (arr[i] == a)

{

return true;

}

}

return false;

}

int main()

{

fstream file; string s, filename;

filename = "./add.c";

file.open(filename.c\_str());

while (file >> s)

{

if (s == "+" || s == "-" || s == "" || s == "/" || s == "^" || s == "&&" || s == "||" || s == "=" || s == "==" || s == "&" || s == "|" || s == "%" || s == "++" || s == "--" || s == "+=" || s == "-=" || s

== "/=" || s == "=" || s == "%=")

{

cout << s << " is an operator\n"; s = "";

}

else if (isKeyword (s))

{

cout << s << " is a keyword\n"; s = "";

}

else if (s == "(" || s == "{" || s == "[" || s == ")" || s == "}" || s == "]" || s == "<" || s ==

">" || s == "()" || s == ";" || s == "<<" || s == ">>" || s == "," || s == "#")

{

cout << s << " is a symbol\n"; s = "";

}

else if (s == "\n" || s == " " || s == "")

{

s = "";

}

else if (isdigit (s[0]))

{

int x = 0;

if (!isdigit (s[x++]))

{

}

else

{

}

}

continue;

cout << s << " is a constant\n"; s = "";

else

{

cout << s << " is an identifier\n"; s = "";

}

}

return 0;

}

# Add.c – file as input :

#include <stdio.h>

void main ( )

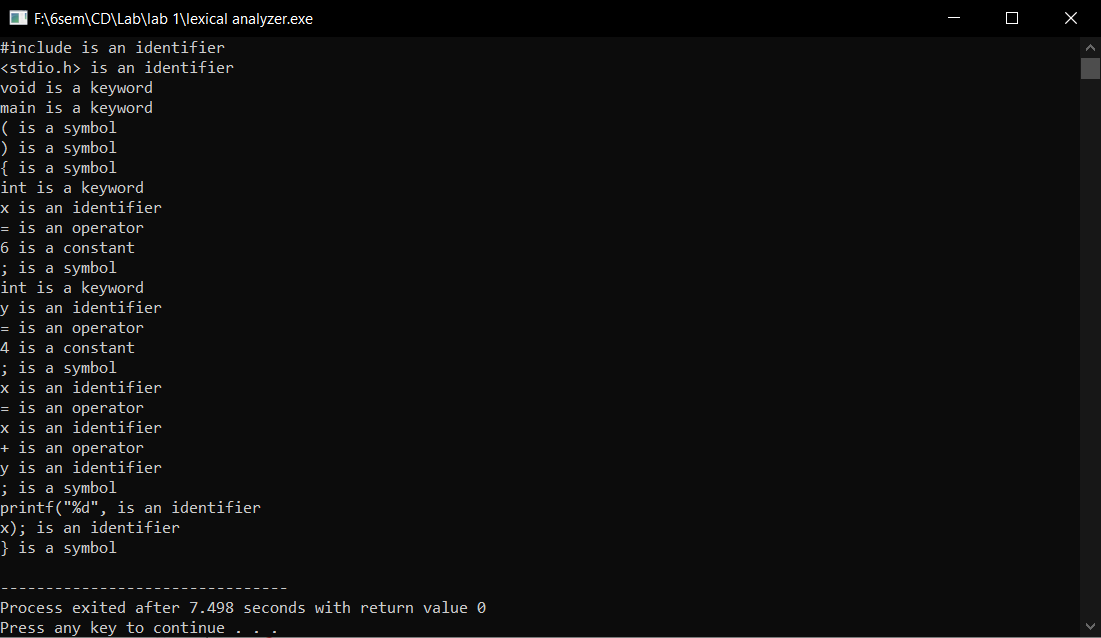
{

int x = 6 ; int y = 4 ; x = x + y ;

printf("%d", x);

}

# Ouptut:



**Result:**

Lexical Analyzer is successfully implemented.

Exp 2Exp 2 – Regular Expression to NFA

Code:

transition\_table = [ [0]\*3 for \_ in range(20) ]

re = input("Enter the regular expression : ") re += " "

i = 0

j = 1 while(i<len(re)):

if re[i] == 'a': try:

if re[i+1] != '|' and re[i+1] !='\*': transition\_table[j][0] = j+1

j += 1

elif re[i+1] == '|' and re[i+2] =='b': transition\_table[j][2]=((j+1)\*10)+(j+3) j+=1

transition\_table[j][0]=j+1 j+=1

transition\_table[j][2]=j+3 j+=1

transition\_table[j][1]=j+1 j+=1

transition\_table[j][2]=j+1 j+=1

i=i+2

elif re[i+1]=='\*': transition\_table[j][2]=((j+1)\*10)+(j+3) j+=1

transition\_table[j][0]=j+1 j+=1

transition\_table[j][2]=((j+1)\*10)+(j-1) j+=1

except:

transition\_table[j][0] = j+1

elif re[i] == 'b': try:

if re[i+1] != '|' and re[i+1] !='\*': transition\_table[j][1] = j+1

j += 1

elif re[i+1]=='|' and re[i+2]=='a': transition\_table[j][2]=((j+1)\*10)+(j+3) j+=1

transition\_table[j][1]=j+1 j+=1

transition\_table[j][2]=j+3

j+=1

transition\_table[j][0]=j+1 j+=1

transition\_table[j][2]=j+1 j+=1

i=i+2

elif re[i+1]=='\*': transition\_table[j][2]=((j+1)\*10)+(j+3) j+=1

transition\_table[j][1]=j+1 j+=1

transition\_table[j][2]=((j+1)\*10)+(j-1) j+=1

except:

transition\_table[j][1] = j+1

elif re[i]=='e' and re[i+1]!='|'and re[i+1]!='\*': transition\_table[j][2]=j+1

j+=1

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

transition\_table[0][2]=((j+1)\*10)+1 transition\_table[j][2]=((j+1)\*10)+1 j+=1

i +=1

print ("Transition function:") for i in range(j):

if(transition\_table[i][0]!=0):

print("q[{0},a]-->{1}".format(i,transition\_table[i][0])) if(transition\_table[i][1]!=0):

print("q[{0},b]-->{1}".format(i,transition\_table[i][1])) if(transition\_table[i][2]!=0):

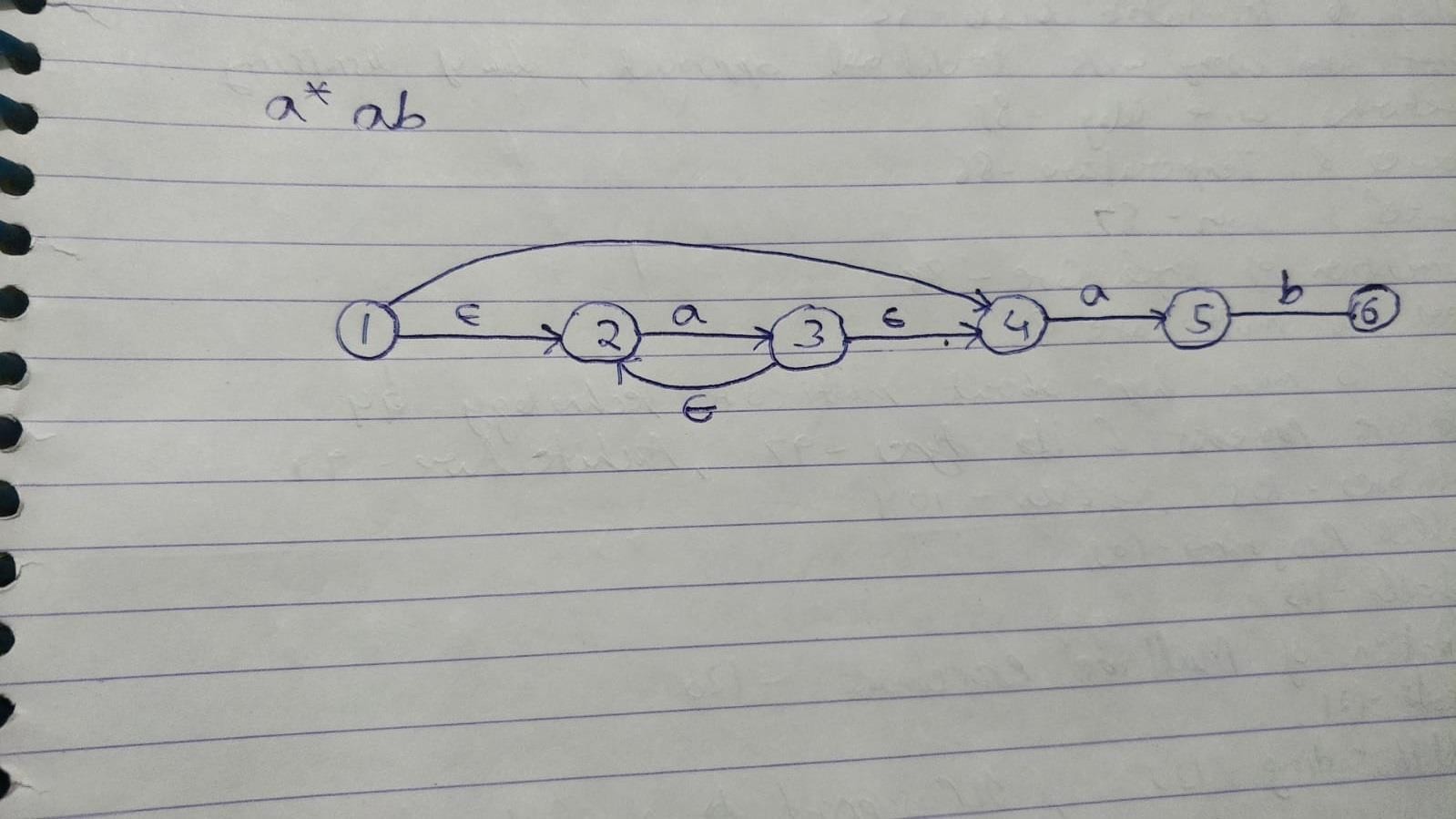
if(transition\_table[i][2]<10):

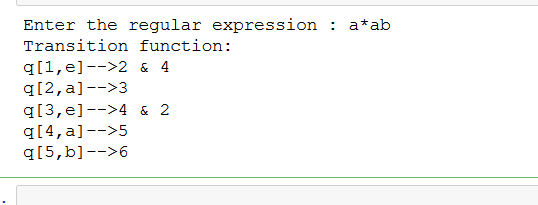
print("q[{0},e]-->{1}".format(i,transition\_table[i][2])) else:

print("q[{0},e]-->{1} &

{2}".format(i,int(transition\_table[i][2]/10),transition\_table[i][2]%10))

Output:





Result:

Regular Expression to NFA is successfully implemented.

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: 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("\nDFA table :- ")

dfa\_table = pd.DataFrame(dfa) print(dfa\_table.transpose())

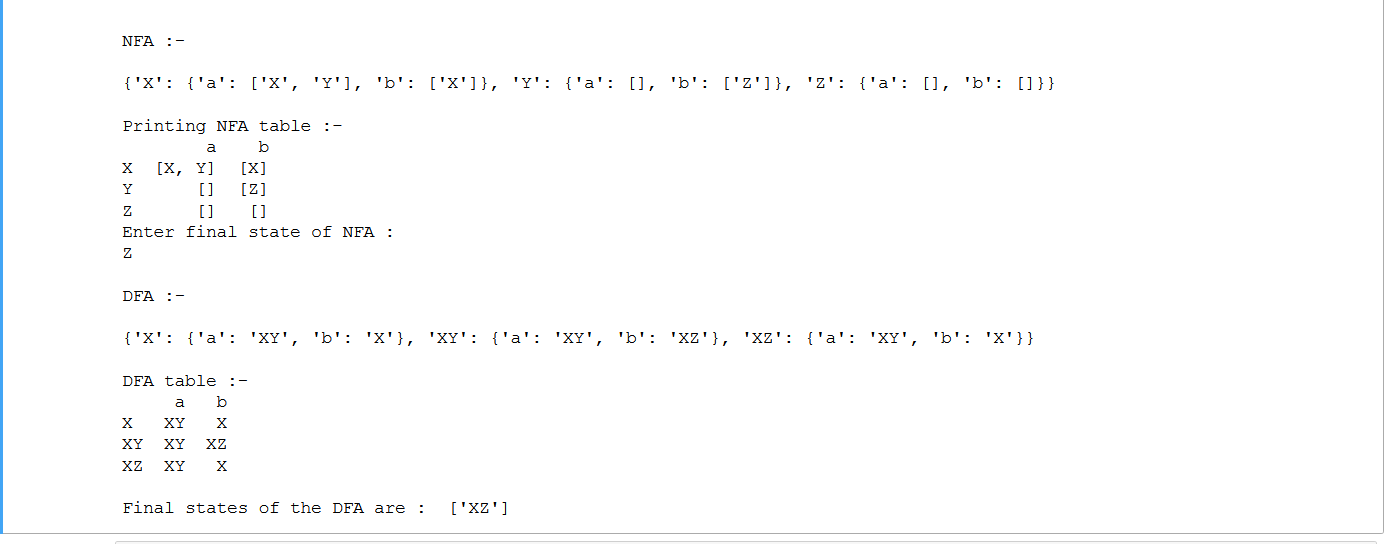
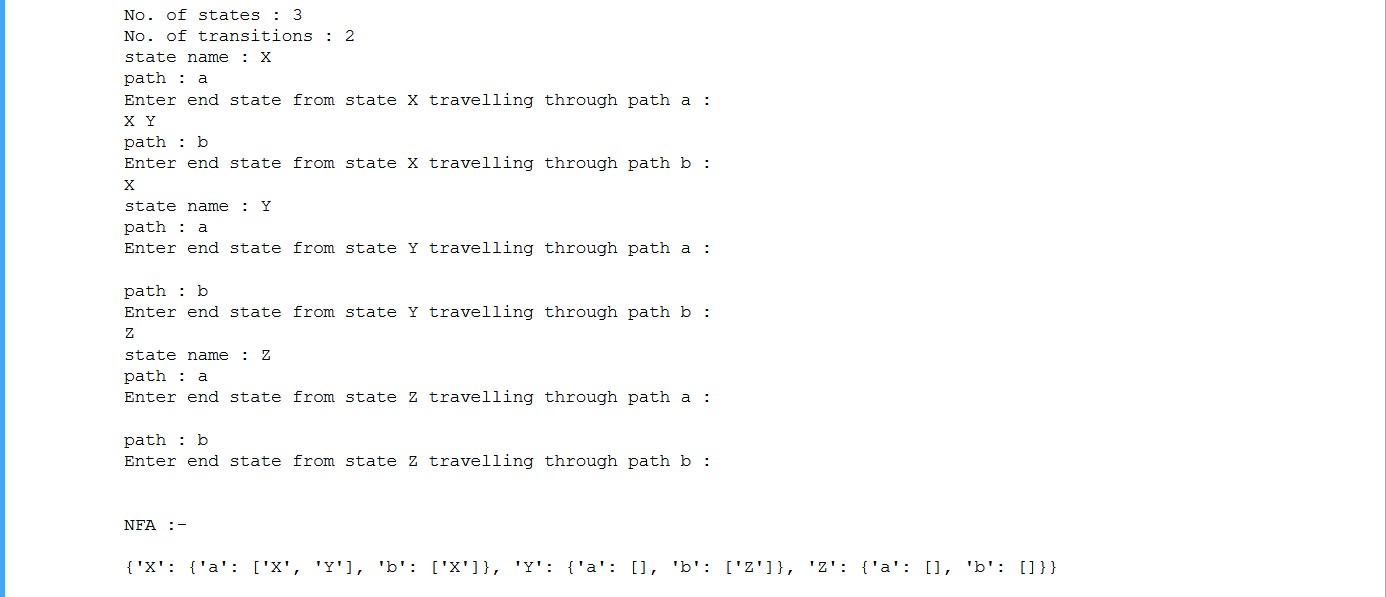
dfa\_states\_list = list(dfa.keys()) dfa\_final\_states = []

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:



Result:

NFA to DFA is successfully implemented.

# Exp 4 - Elimination of Left Recursion, Left Factoring and Ambiguity

**Ambiguity and Left Recursion**

# Code:

#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 a non-terminal: "; cin >> c;

d.push\_back(c); a += d + "\'->";

d += "->";

b += d;

cout << "Enter no of 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 contain 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 << " contains 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 += "\'|";

}

}

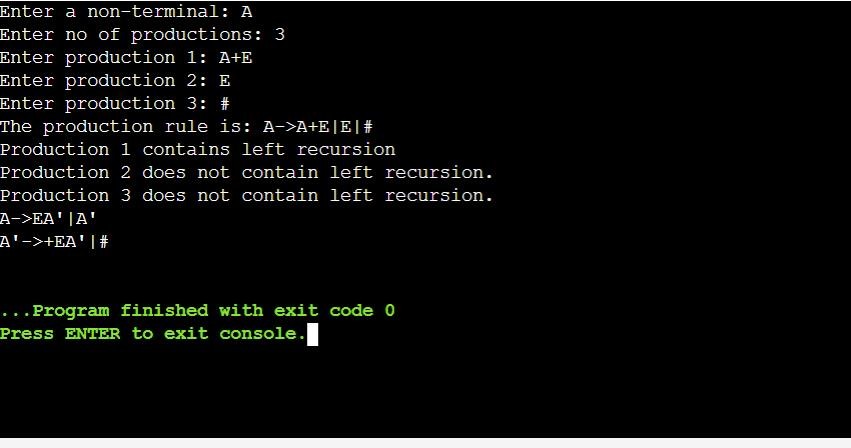
}

a += "#";

cout << b << endl; cout << a << endl; return 0;

}

# Output:



# Exp -5– Generating left factored grammar

# Code in C++ :

**Left Factoring**

# Code:

from itertools import takewhile def groupby(ls):

d = {}

ls = [ y[0] for y in rules ] initial = list(set(ls))

for y in initial: for i in rules:

if i.startswith(y): if y not in d:

d[y] = []

d[y].append(i) return d

def prefix(x):

return len(set(x)) == 1

starting="" rules=[] common=[]

alphabetset=["A'","B'","C'","D'","E'","F'","G'","H'","I'","J'","K'","L'","M'","N'",

"O'","P'","Q'","R'","S'","T'","U'","V'","W'","X'","Y'","Z'"]

s=input() while(True):

rules=[] common=[] split=s.split("->") starting=split[0]

for i in split[1].split("|"): rules.append(i)

for k, l in groupby(rules).items():

r = [l[0] for l in takewhile(prefix, zip(\*l))] common.append(''.join(r))

for i in common: newalphabet=alphabetset.pop() print(starting+"->"+i+newalphabet) index=[]

for k in rules: if(k.startswith(i)):

index.append(k) print(newalphabet+"->",end="") for j in index[:-1]:

stringtoprint=j.replace(i,"", 1)+"|" if stringtoprint=="|":

print("\u03B5","|",end="") else:

print(j.replace(i,"", 1)+"|",end="") stringtoprint=index[-1].replace(i,"", 1)+"|"

if stringtoprint=="|": print("\u03B5","",end="")

else:

print(index[-1].replace(i,"", 1)+"",end="") print("")

break

# Ouptut:



Result:

Elimination of Left Recursion, Left Factoring and Ambiguity is successfully implemented.

**Exp -6 & 7 - Program to perform FIRST AND FOLLOW computation**

# Code:

gram = { "A":["A+B","B"],

"B":["B\*C","C"],

"C":["(A)","D"], "D":["id"]

}

def removeDirectLR(gramA, A): temp = gramA[A]

tempCr = [] tempInCr = [] for i in temp:

if i[0] == A: tempInCr.append(i[1:]+[A+"'"])

else:

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)

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)

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)

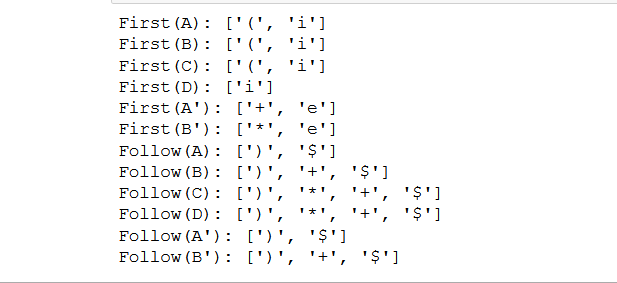
return a

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])

# Output:



**Result:**

First and Follow computation is successfully implemented

**Exp – 8 – Implement Shit Reduce Parsing**

**Code:**

gram = {

"S":["S-S","S\*S","a"]

}

starting\_terminal = "S" inp = "a-a\*a$"

stack = "$"

print(f'{"Stack": <15}'+"|"+f'{"Input": <15}'+"|"+f'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:]

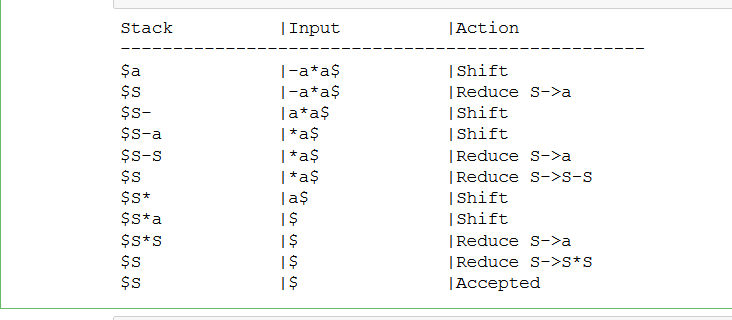
print(f'{stack: <15}'+"|"+f'{inp: <15}'+"|"+f'Shift') action = False

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:**



**Result:**

Shift Reduce Parsing is successfully implemented.

**Exp – 9&10 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,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<<"\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;

break;

}

}

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 << "}";

}

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]<<",";

}

cout << "}";

}

}

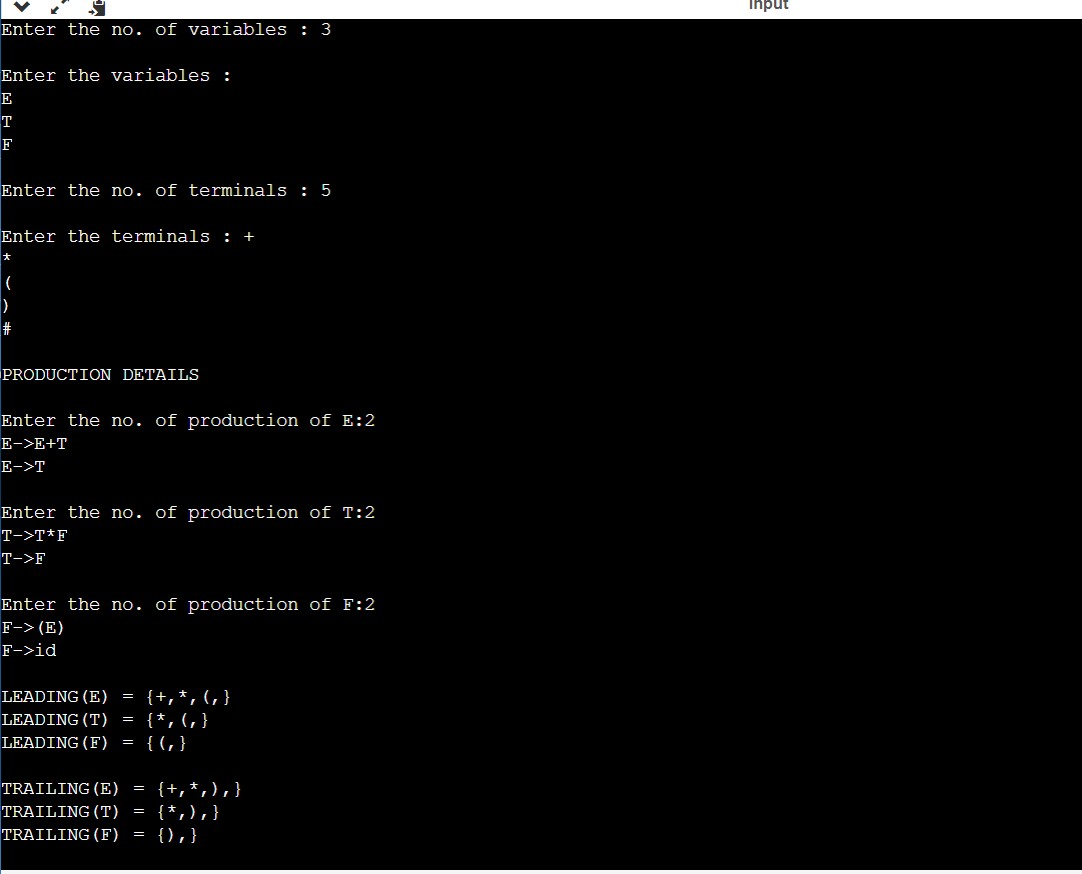
int main()

{

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

}

**Output:**



**Result:**

Leading and Trailing is successfully implemented.

**Lab 11&12 – Intermediate Code Generation – Prefix, Postfix**

**Code:**

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

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:

1]]:

while op\_stack and op\_stack[-1] != '(' and PRI[ch] <= PRI[op\_stack[- 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]}')

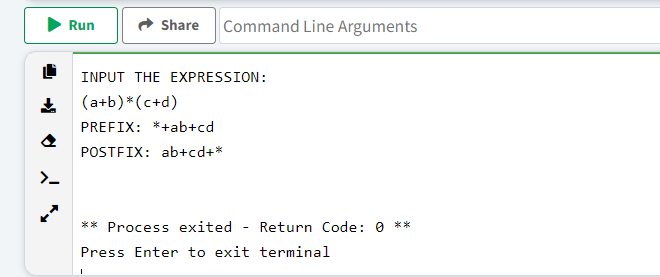
return exp\_stack[-1]

expres = input("INPUT THE EXPRESSION: ")

pre = infix\_to\_prefix(expres)

pos = infix\_to\_postfix(expres)

**Output:**



**Result:**

Intermediate Code Generation – Prefix, Postfix is successfully implemented.