**INDEX**

|  |  |  |
| --- | --- | --- |
| **EXP NO.** | **TITLE** | **PAGE NO.** |
| 1 | Lexical Analyser | 3 |
| 2 | Regular expression to NFA Conversion | 6 |
| 3 | NFA to DFA Conversion | 9 |
| 4 | Removal of Left Recursion and Left Factors | 13 |
| 5 | Computation of First and Follow | 20 |
| 6 | Construction of Predictive parsing table | 25 |
| 7 | Shift Reduce Parsing | 31 |
| 8 | Leading Trailing Computation | 33 |
| 9 | Computation of LR(0) Items | 36 |
| 10 | Conversion of Infix to Postfix, Infix to  prefix and Three Address Code generation | 40 |
| 11 | Intermediate code generation | 44 |

**Experiment 1**

**Topic: Lexical Analyzer**

**Aim:**

To design and implement a Lexical analyzer in Python.

**Algorithm:**

1. Start.
2. Get the input program from the lexical.txt document.
3. Read the program line by line and check if each word separated by whitespace in a line is a keyword, identifier, numerals, separators, special symbols, built-in functions, header file or an operator.
4. If the word read is an identifier, add the value to a set created for storing identifiers.
5. Repeat the same process for the rest of the categories.
6. Print all the set values along with their count.
7. Stop the program.

**Code:**

import re

keyword = ['break','case','char','const','continue','default','do','int','else','enum','extern','float','for','goto','if',

            'long','register','return','short','signed','sizeof','static','switch','typedef','union','unsigned','void','volatile','while']

built\_in\_functions = ['clrscr()','printf(','scanf(','getch()','main()']

operators = ['+','-','\*','/','%','==','!=','>','<','>=','<=','&&','||','!','&','|','^','~','>>','<<','=','+=','-=','\*=']

specialsymbol = ['@','#','$','\_','!']

separator = [',',':',';','\n','\t','{','}','(',')','[',']']

kw = set()

op = set()

sp = set()

bf = set()

sep =set()

hf = set()

num = set()

iden = set()

file = open('lexical.txt','r+')

contents = file.read()

splitCode = contents.split() #split program in word based on space

length = len(splitCode)      # count the number of word in program

for i in range(0,length):

    if splitCode[i] in keyword:

        kw.add(splitCode[i])

        continue

    if splitCode[i] in operators:

        op.add(splitCode[i])

        continue

    if splitCode[i] in specialsymbol:

        sp.add(splitCode[i])

        continue

    if splitCode[i] in built\_in\_functions:

        bf.add(splitCode[i])

        continue

    if splitCode[i] in separator:

        sep.add(splitCode[i])

        continue

    if re.match(r'(#include\*).\*', splitCode[i]):

        hf.add(splitCode[i])

        continue

    if re.match(r'^[-+]?[0-9]+$',splitCode[i]):

        num.add(splitCode[i])

        continue

    if re.match(r"^[^\d\W]\w\*\Z", splitCode[i]):

        iden.add(splitCode[i])

print("\nType\t\t\t|Count\t\t|Values")

print("-------------------------------------------------------------")

print("\nHeader File\t\t|",str(len(hf))+"\t\t|",hf)

print("\nBuilt-in functions\t|",str(len(bf))+"\t\t|",bf)

print("\nKeywords\t\t|",str(len(kw))+"\t\t|",kw)

print("\nOperators\t\t|",str(len(op))+"\t\t|",op)

print("\nSpecial Symbols\t\t|",str(len(sp))+"\t\t|",sp)

print("\nNumerals\t\t|",str(len(num))+"\t\t|",num)

print("\nIdentifiers\t\t|",str(len(iden))+"\t\t|",iden)

print("\nSeparators\t\t|",str(len(sep))+"\t\t|",sep)

**lexical.txt document**

#include<stdio.h>

int main()

{

int a , b , c ;

a = 15 ;

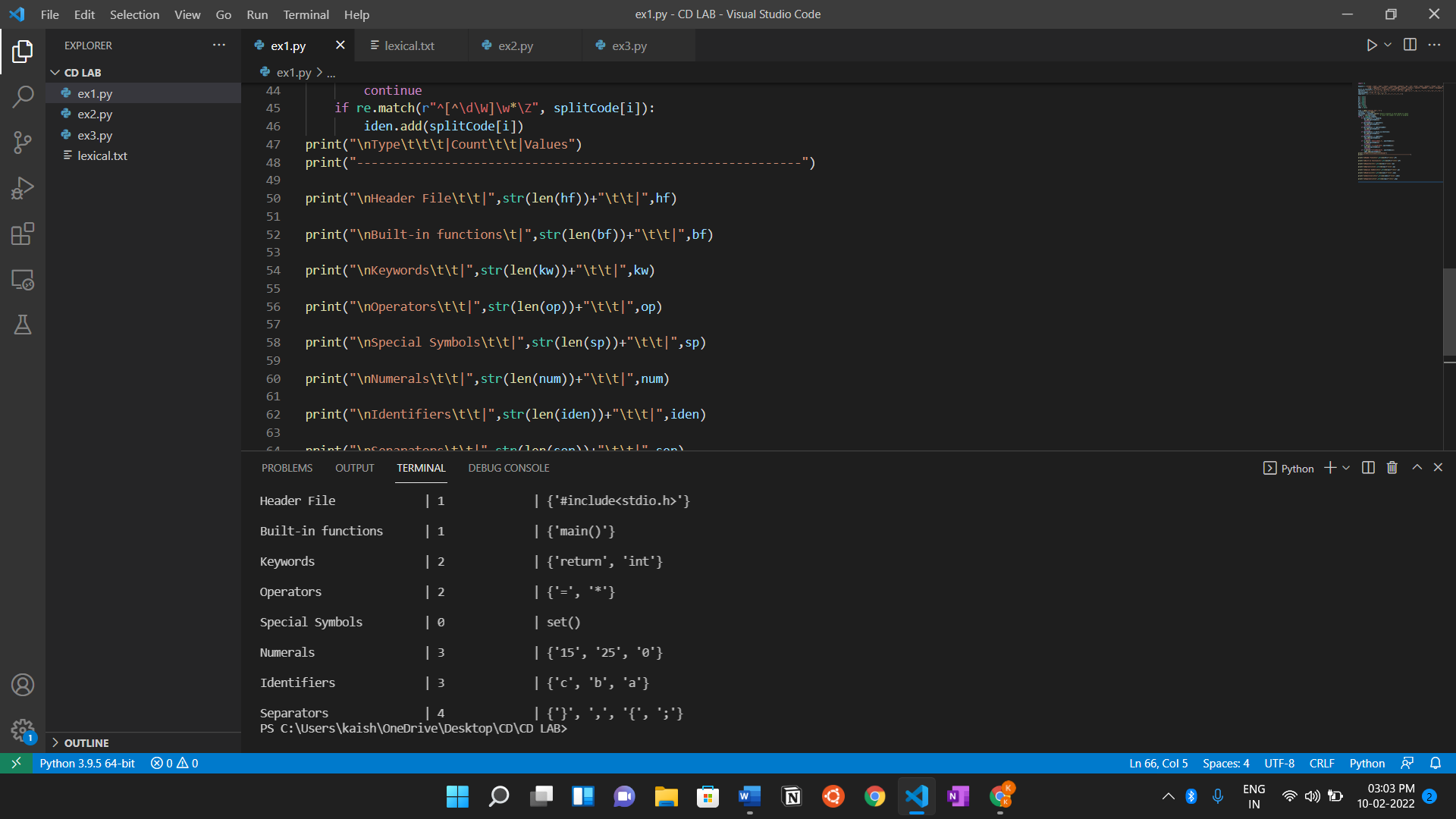
b = 25 ;

c = a \* b ;

return 0 ;

}

**Output:**



**Result:**

Thus, we have implemented Lexical Analyzer in Python.

**Experiment 2**

**Topic: Regular Expression to NFA**

**Aim:**

To design and implement a program which converts a Regular Expression to NFA in Python.

**Algorithm:**

1. Start.
2. Get the input Regular expression from the user.
3. Initialize the transition table as a list.
4. Making use of Thompson’s construction rules we write the algorithm by checking each and every character and updating the transition table along the way.
5. In the end we get the transition table and display it.
6. Stop the program.

**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):

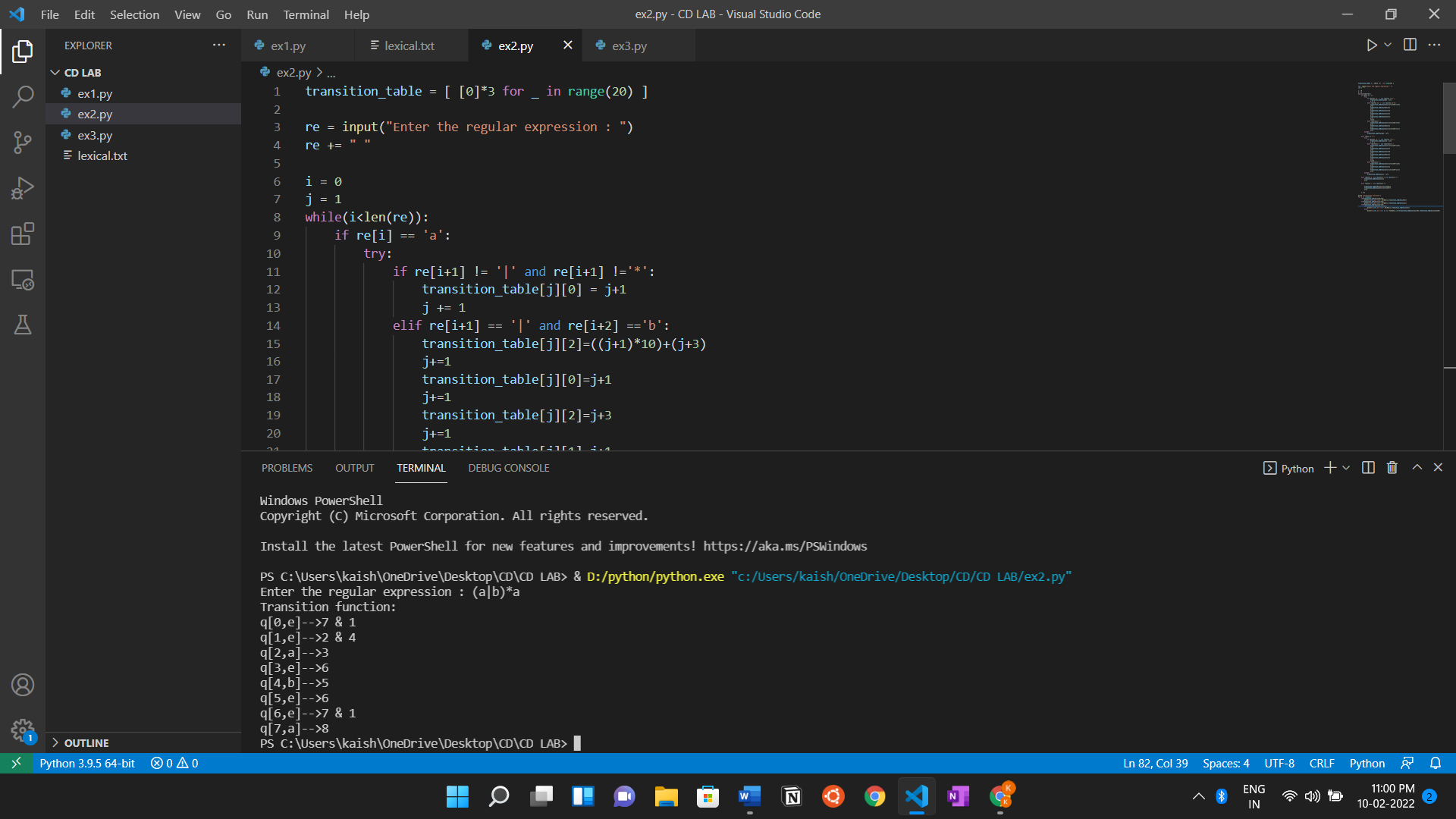
        if(transition\_table[i][2]<10):

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

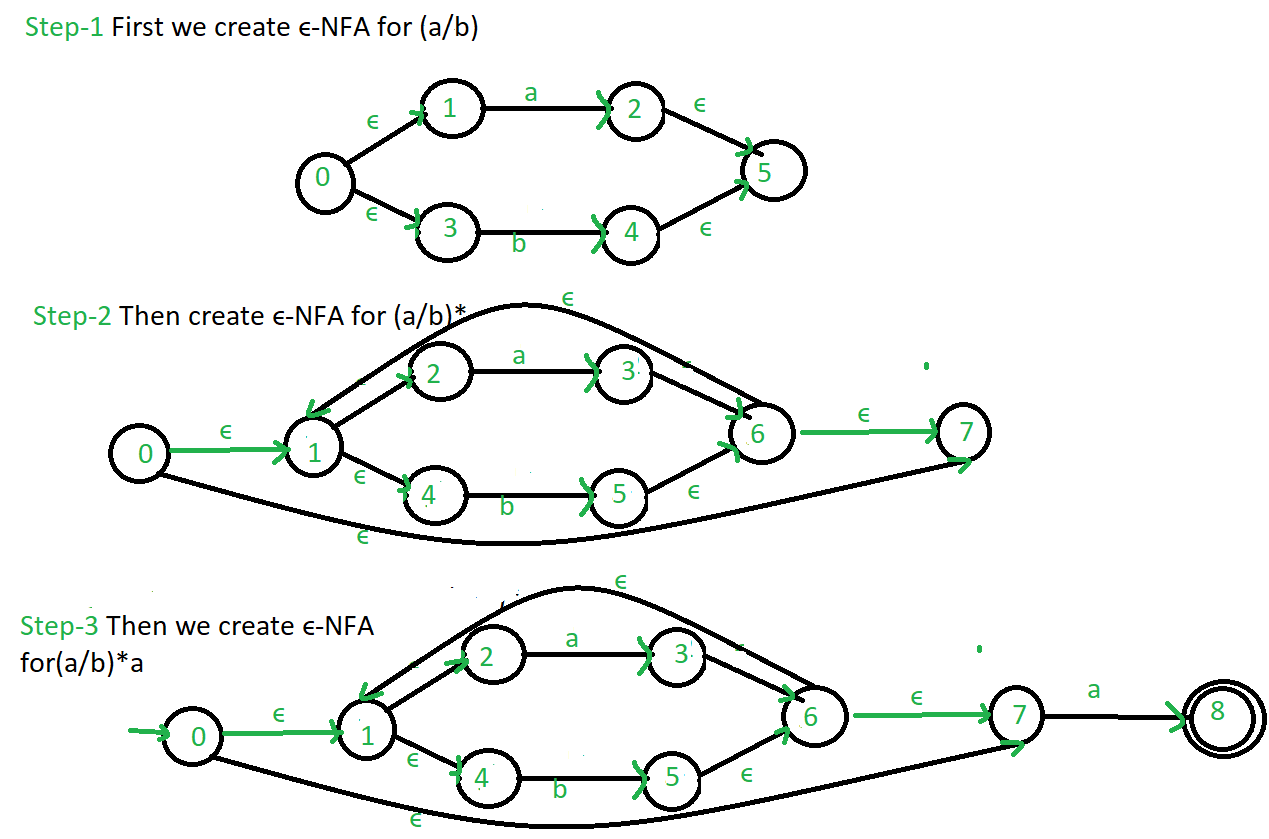
        else:

            print("q[{0},e]-->{1} & {2}".format(i,int(transition\_table[i][2]/10),transition\_table[i][2]%10))

**Output:**



**For this particular input: (a|b)\*a**



**Result:**

Thus, we have executed a program which converts a given Regular expression to NFA.

**Experiment 3**

**Topic: NFA to DFA**

**Aim:**

To design and implement a program which converts an NFA to DFA in Python.

**Algorithm:**

1. Start.
2. Get all the NFA input details from the user (number of states, number of transitions, transition table details, final state of NFA etc)
3. Print the NFA table in an orderly manner.
4. Create a new list to hold all the new states for the DFA, a dictionary for output table for DFA, a list containing all states in NFA and DFA and list of all paths.
5. Next we compute the first row of DFA transition table and then compute the other rows of DFA transition table.
6. Print the DFA created and its Final states computed.
7. Stop the program.

**Code:**

import pandas as pd

# Taking NFA input from User

nfa = {}

n = int(input("No. of states : "))            #Enter total no. of states

t = int(input("No. of transitions : "))       #Enter total no. of transitions/paths eg: a,b so input 2 for a,b,c input 3

for i in range(n):

    state = input("state name : ")            #Enter state name eg: A, B, C, q1, q2 ..etc

    nfa[state] = {}                           #Creating a nested dictionary

    for j in range(t):

        path = input("path : ")               #Enter path eg : a or b in {a,b} 0 or 1 in {0,1}

        print("Enter end state from state {} travelling through path {} : ".format(state,path))

        reaching\_state = [x for x in input().split()]  #Enter all the end states that

        nfa[state][path] = reaching\_state     #Assigning the end states to the paths in dicti4onary

print("\nNFA :- \n")

print(nfa)                                    #Printing 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()]      # Enter final state/states of NFA

###################################################

new\_states\_list = []                          #holds all the new states created in dfa

dfa = {}                                      #dfa dictionary/table or the output structure we needed

keys\_list = list(list(nfa.keys())[0])                  #conatins all the states in nfa plus the states created in dfa are also appended further

path\_list = list(nfa[keys\_list[0]].keys())    #list of all the paths eg: [a,b] or [0,1]

###################################################

# Computing first row of DFA transition table

dfa[keys\_list[0]] = {}                        #creating a nested dictionary in dfa

for y in range(t):

    var = "".join(nfa[keys\_list[0]][path\_list[y]])   #creating a single string from all the elements of the list which is a new state

    dfa[keys\_list[0]][path\_list[y]] = var            #assigning the state in DFA table

    if var not in keys\_list:                         #if the state is newly created

        new\_states\_list.append(var)                  #then append it to the new\_states\_list

        keys\_list.append(var)                        #as well as to the keys\_list which contains all the states

###################################################

# Computing the other rows of DFA transition table

while len(new\_states\_list) != 0:                     #consition is true only if the new\_states\_list is not empty

    dfa[new\_states\_list[0]] = {}                     #taking the first element of the new\_states\_list and examining it

    for \_ in range(len(new\_states\_list[0])):

        for i in range(len(path\_list)):

            temp = []                                #creating a temporay list

            for j in range(len(new\_states\_list[0])):

                temp += nfa[new\_states\_list[0][j]][path\_list[i]]  #taking the union of the states

            s = ""

            s = s.join(temp)                         #creating a single string(new state) from all the elements of the list

            if s not in keys\_list:                   #if the state is newly created

                new\_states\_list.append(s)            #then append it to the new\_states\_list

                keys\_list.append(s)                  #as well as to the keys\_list which contains all the states

            dfa[new\_states\_list[0]][path\_list[i]] = s   #assigning the new state in the DFA table

    new\_states\_list.remove(new\_states\_list[0])       #Removing the first element in the new\_states\_list

print("\nDFA :- \n")

print(dfa)                                           #Printing the DFA created

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:

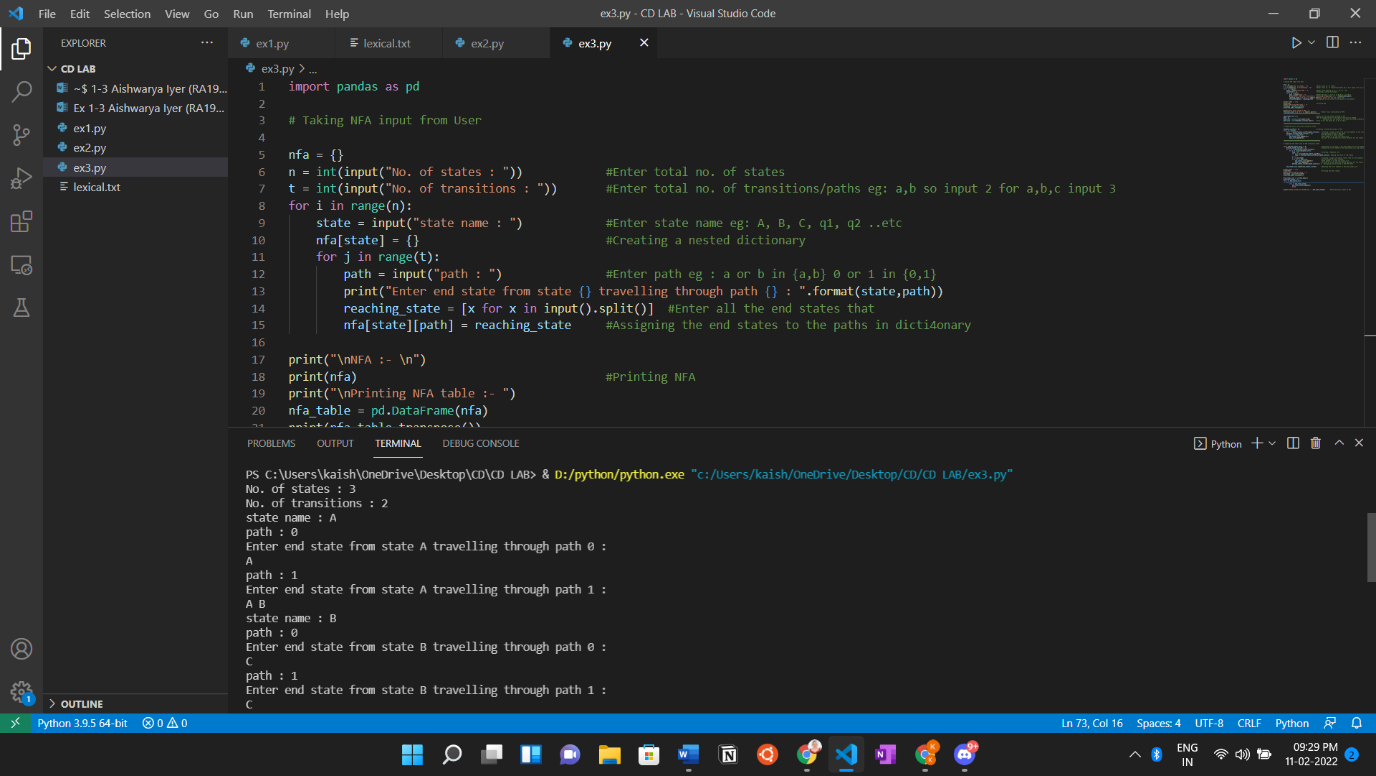
        if i in nfa\_final\_state:

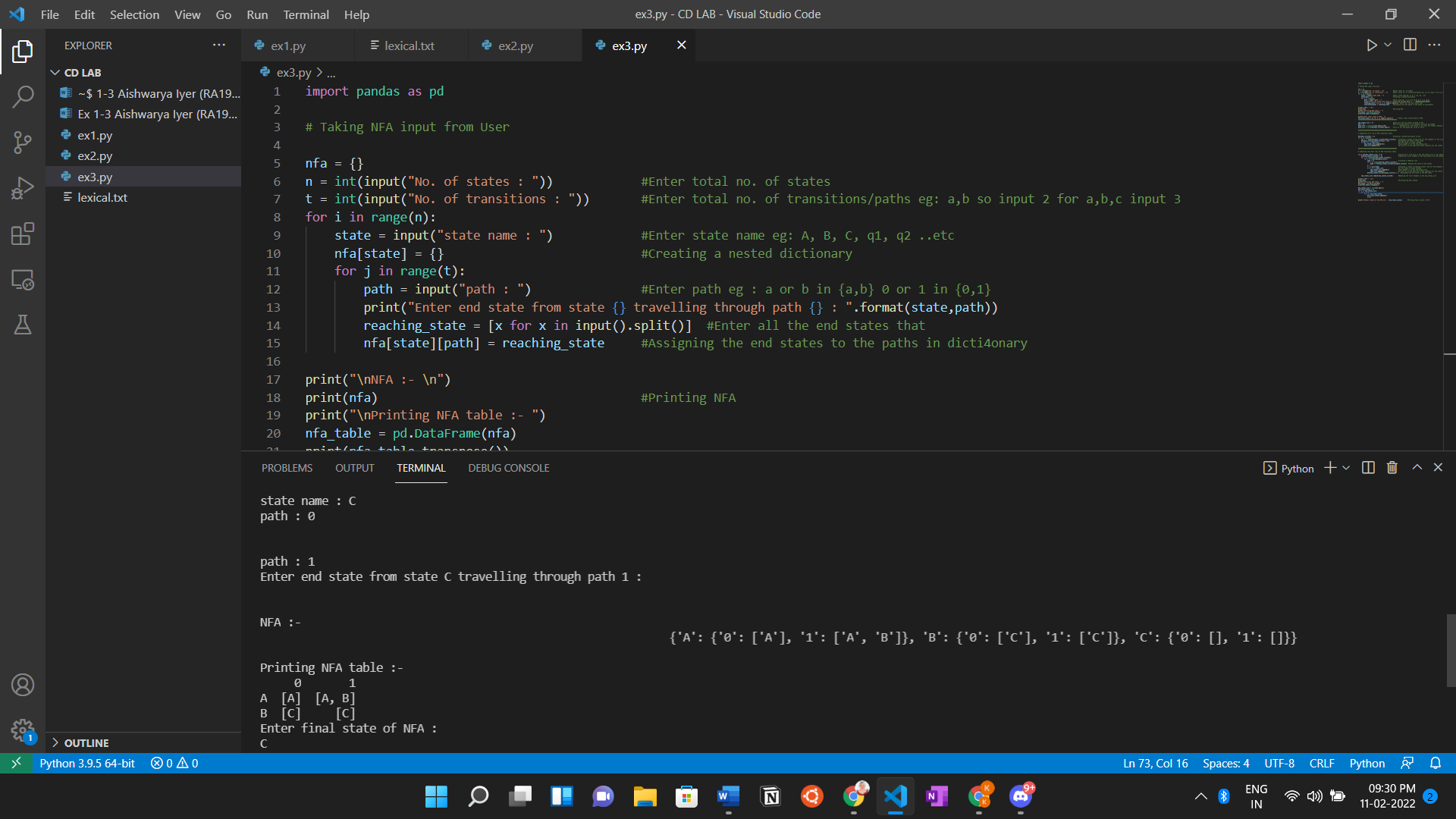
            dfa\_final\_states.append(x)

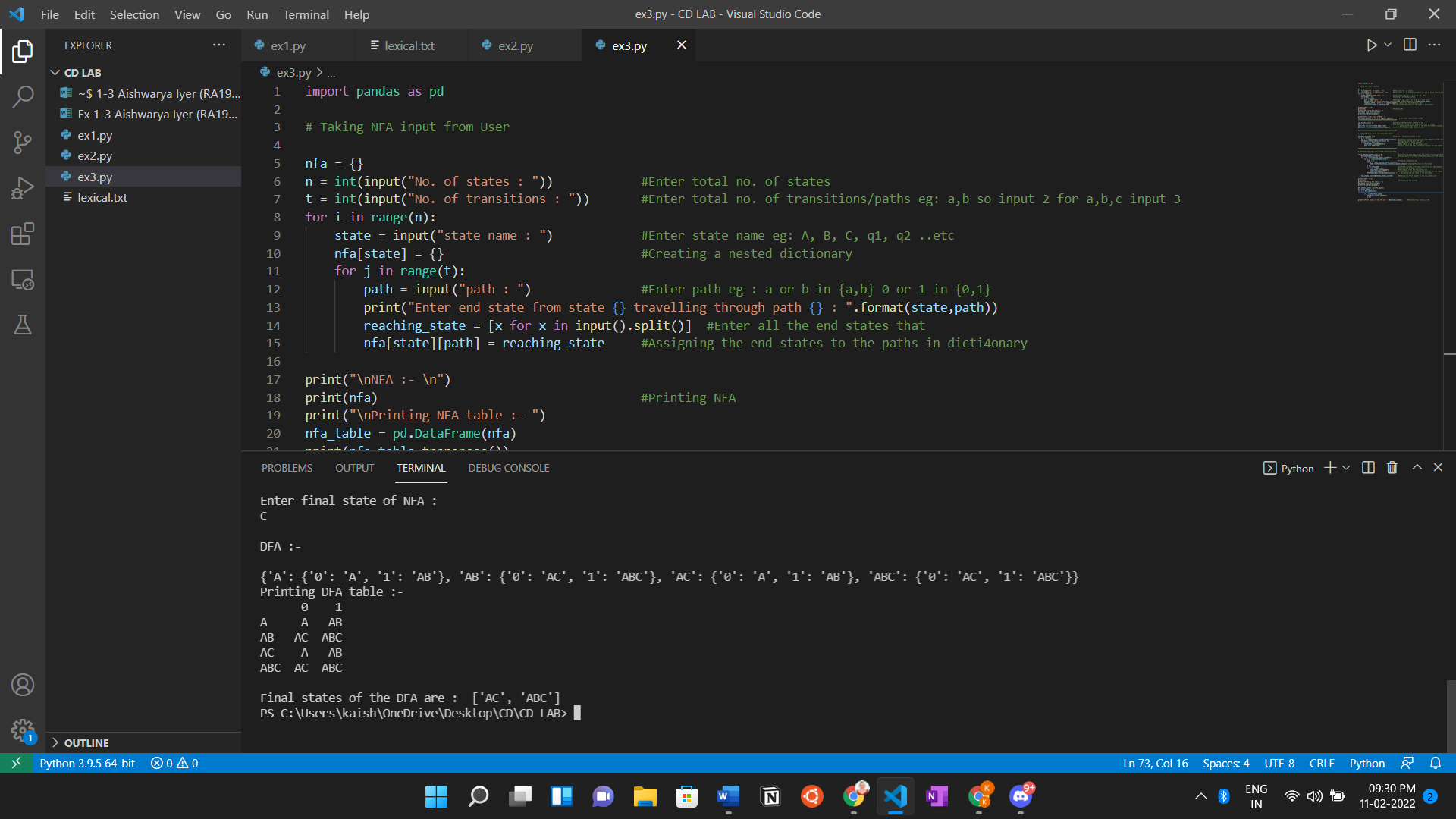
            break

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

**Output:**

****

****

****

**Result:**

Thus, we have designed and implemented a program which converts an NFA to DFA in Python.

**Experiment 4**

**Topic: Left Recursion and Left Factoring Elimination**

**4a. Topic : Left Recursion Elimination**

**Aim:**

To design and implement a Left Recursion elimination of productions in Python.

**Algorithm:**

1. Start.
2. Get the input production from the user.
3. Create a dictionary for the grammar.
4. Create a function add() to split the production into parts and add it to the grammar.
5. Create a function to remove direct left recursion following the rules.
   1. For eg. For production A ->Az | b
      1. A -> bA’
      2. A’ -> zA’ | e (Here e = Epsilon symbol)

(A, A’ = Non-Terminals ; b, z = Terminals)

1. Create a function to check Indirect left recursion.
2. Print the Productions after removing Left Recursion.
3. Stop the program.

**Code:**

gram = {}

def add(str):

    x = str.split("->")

    y = x[1]

    x.pop()

    z = y.split("|")

    x.append(z)

    gram[x[0]]=x[1]

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

n = int(input("Enter No of Production: "))

for i in range(n):

    txt=input()

    add(txt)

result = rem(gram)

i = 0

print("The Left Recursion for the given production is: ")

for x,y in result.items():

    if i == 0:

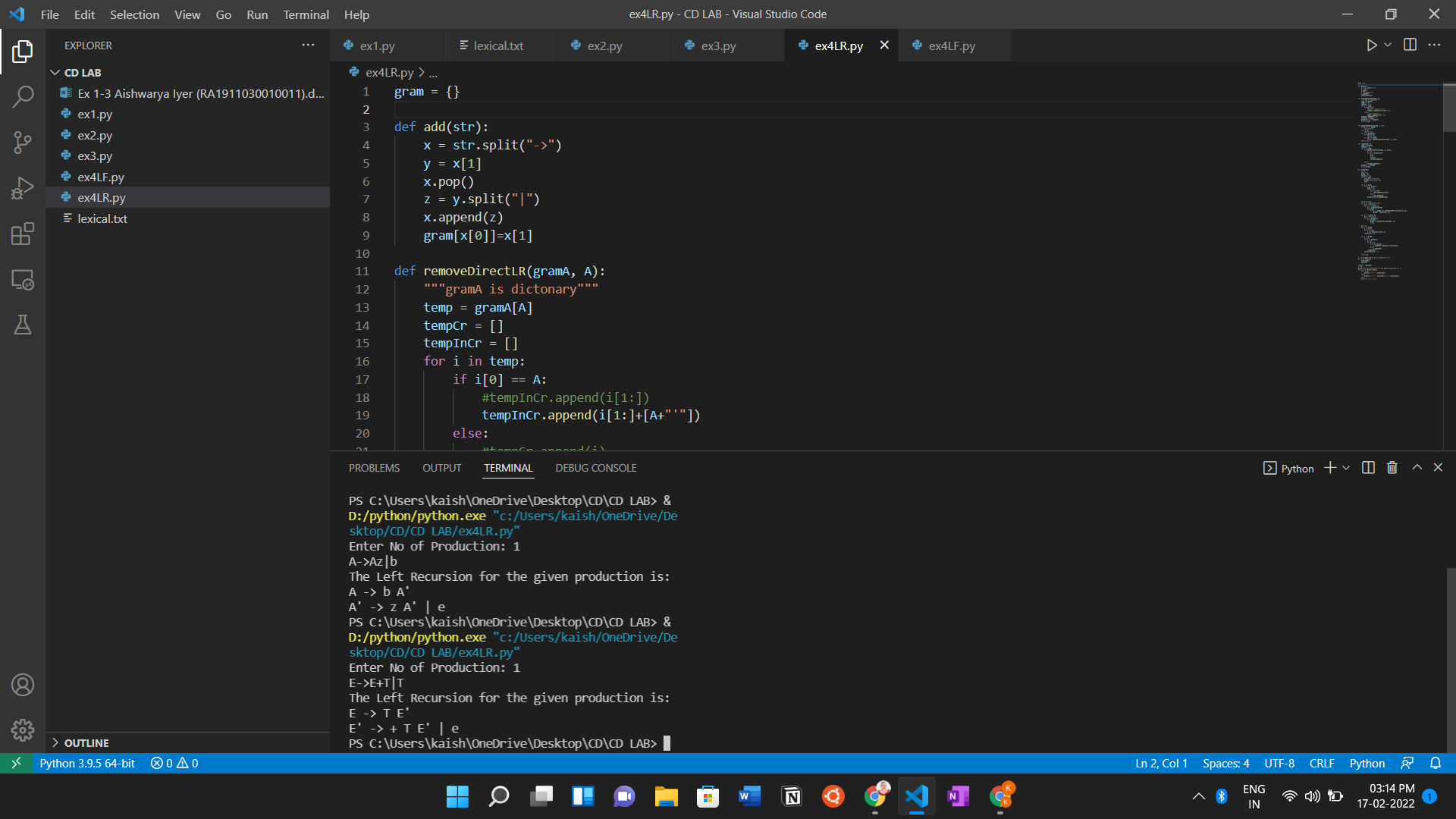
        print(x,"->"," ".join(y[0]))

    else:

        print(x,"->"," ".join(y[0]),"|","".join(y[1]))

    i+=1

**Output:**



**Result:**

Thus, we have designed and implement a program to remove Left Recursion from a production.

**4b. Topic : Left Factoring Elimination**

**Aim:**

To design and implement a Left Factoring Elimination of productions in Python.

**Algorithm:**

1. Start.
2. Enter the number of productions you wish to check Factoring for.
3. Create a dictionary for the grammar.
4. Create a function to split the production into parts and add it to the grammar.
5. Create a function to perform left factoring following the rules.
   1. For eg. For production A ->Az | Ab
      1. A -> AA’
      2. A’ -> z | b

(A, A’ = Non-Terminals ; b, z = Terminals)

1. Create a function to display Left Factoring.
2. Print the Productions after Left Factoring.
3. Stop the program.

**Code:**

#function for removing left Factoring

def fact(nonterm,transition):

    prod = transition[nonterm]

    prod1 = nonterm + "->"

    prod2 = nonterm + "'->"

    minn = 1000

    comp = prod[0]

    for i in prod:

        count=0

        for j in i:

            if j!=comp[i.index(j)]:

                minn=min(minn,count)

            count = count+1

    if minn==0:

        return 0

    count=0

    for i in comp:

        if count<minn:

            prod1 = prod1 + i

            count = count + 1

    prod1 = prod1 + nonterm + "'"

    for i in prod:

        count = 0

        for j in i:

            count = count + 1

            if count>minn:

                prod2 = prod2 + j

        if(prod.index(i) != (len(prod)-1)):

            prod2 = prod2 + "|"

    return(prod1 + "$" + prod2)

def split\_grammar(production):

    split\_prod = production.split("->")

    nonterm=split\_prod[0]

    temp = split\_prod[1].split("|")

    transition[nonterm]=temp

    return nonterm,transition

def print\_grammar(grammar):

    grammar = grammar.split("$")

    print(grammar[0])

    print(grammar[1])

###################   Starting of Code   ###################

n = int(input("\nEnter total number of productions : "))

char = '$'

transition = dict()

for iter in range(n):

    production = input("\nEnter Production " + str(iter+1) + " : ")

    temp\_production = production

    temp\_list = list()

    transition = dict()

    nonterm, transition = split\_grammar(production)

    for j in transition[nonterm]:

        if j[0] == char:

            for k in split\_grammar1:

                temp\_string = k + j[1::]

                temp\_list.append(temp\_string)

        else:

            temp\_list.append(j)

    transition[nonterm] = temp\_list

    production = fact(nonterm, transition)

    if production != 0:

        print("\nProduction " + str(iter+1) + " has Left Factoring\n")

        print\_grammar(production)

        production = production.split("$")

        production = production[0]

    else:

        print("\nProduction " + str(iter+1) + " doesn't have Left Factoring")

        production = temp\_production

    production = production.split("$")

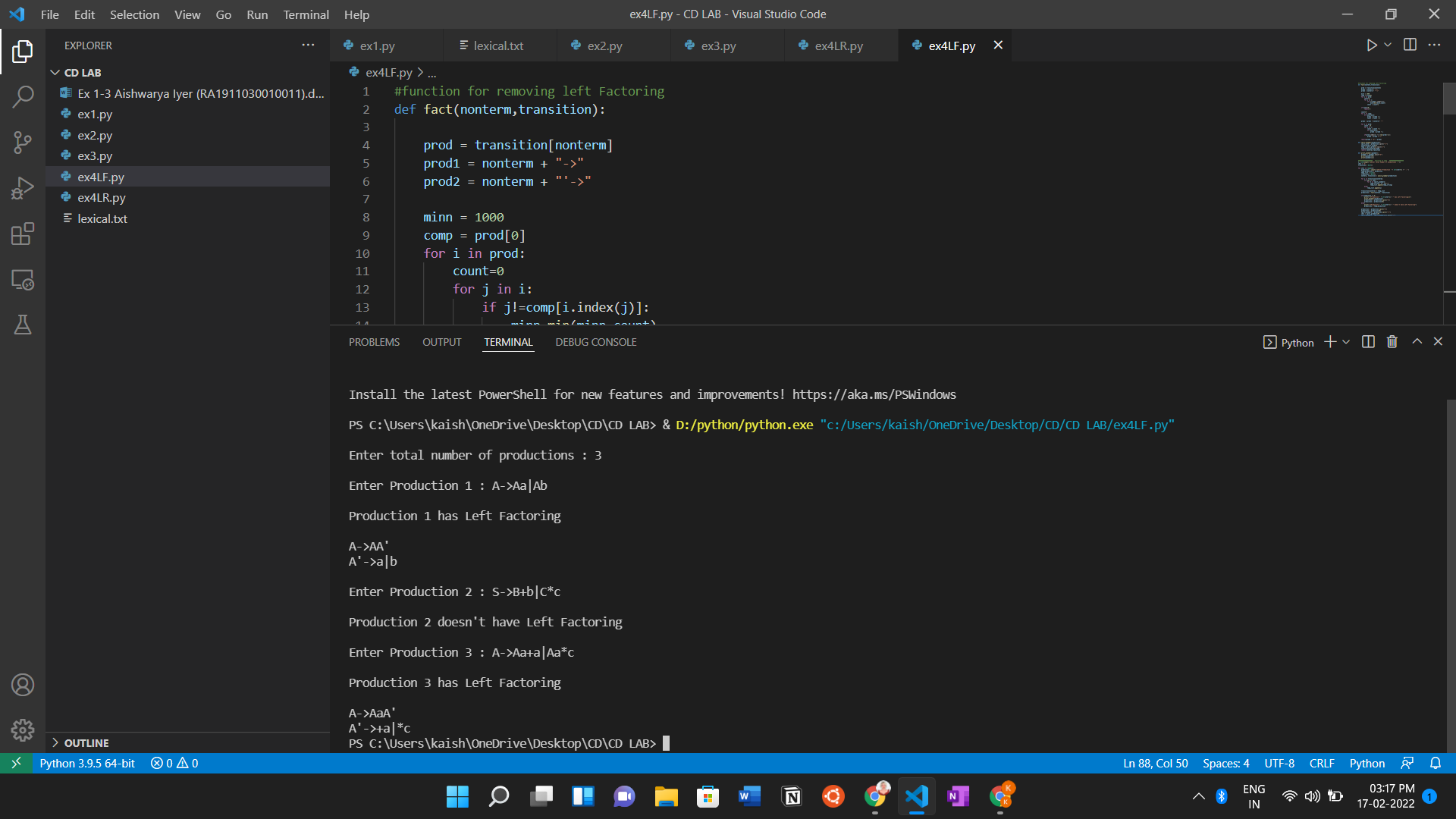
    production = production[0]

    split\_grammar1 = production.split("->")

    char = split\_grammar1[0]

    split\_grammar1 = split\_grammar1[1].split("|")

**Output:**

****

**Result:**

Thus, we have designed and implemented Left Factoring Elimination for productions in Python.

**Experiment 5**

**Topic: Computation of FIRST and FOLLOW**

**Aim:**

To design and implement a program that computes FIRST and FOLLOW for given productions in Python.

**Algorithm:**

1. Start.
2. Accept input from the user in the form of number of terminals and the said terminals, number of non-terminals and the said non-terminals, the starting symbol and the number of productions and said productions all in this order.
3. Create a dictionary for productions and store them after splitting the productions in it.
4. Create a function to compute FIRST(X) of the given productions according to the rules, where X is a grammar symbol:
   1. If X is a terminal, then FIRST(X) = {X}
   2. If X -> € is a production, then add € to FIRST(X).
   3. If X is a non-terminal and X-> Y1 Y2 … Yk is a production, then add FIRST(Y1) to FIRST(X). If Y1 derives €, then add FIRST(Y2) to FIRST(X).
5. Create a function to compute FOLLOW of the given productions according to rules:
   1. For the FOLLOW(start symbol) place $, where $ is the input end marker.
   2. If there is a production A->αBβ, then everything in FIRST(β) except € is in FOLLOW(B).
   3. If there is a production A->αB, or a production A->αBβ where FIRST(β) contains €, then everything in FOLLOW(A) is in FOLLOW(B).
6. Run the first and follow functions for the given productions and display the result in a table format.
7. Stop the program.

**Code:**

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\_

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)

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)

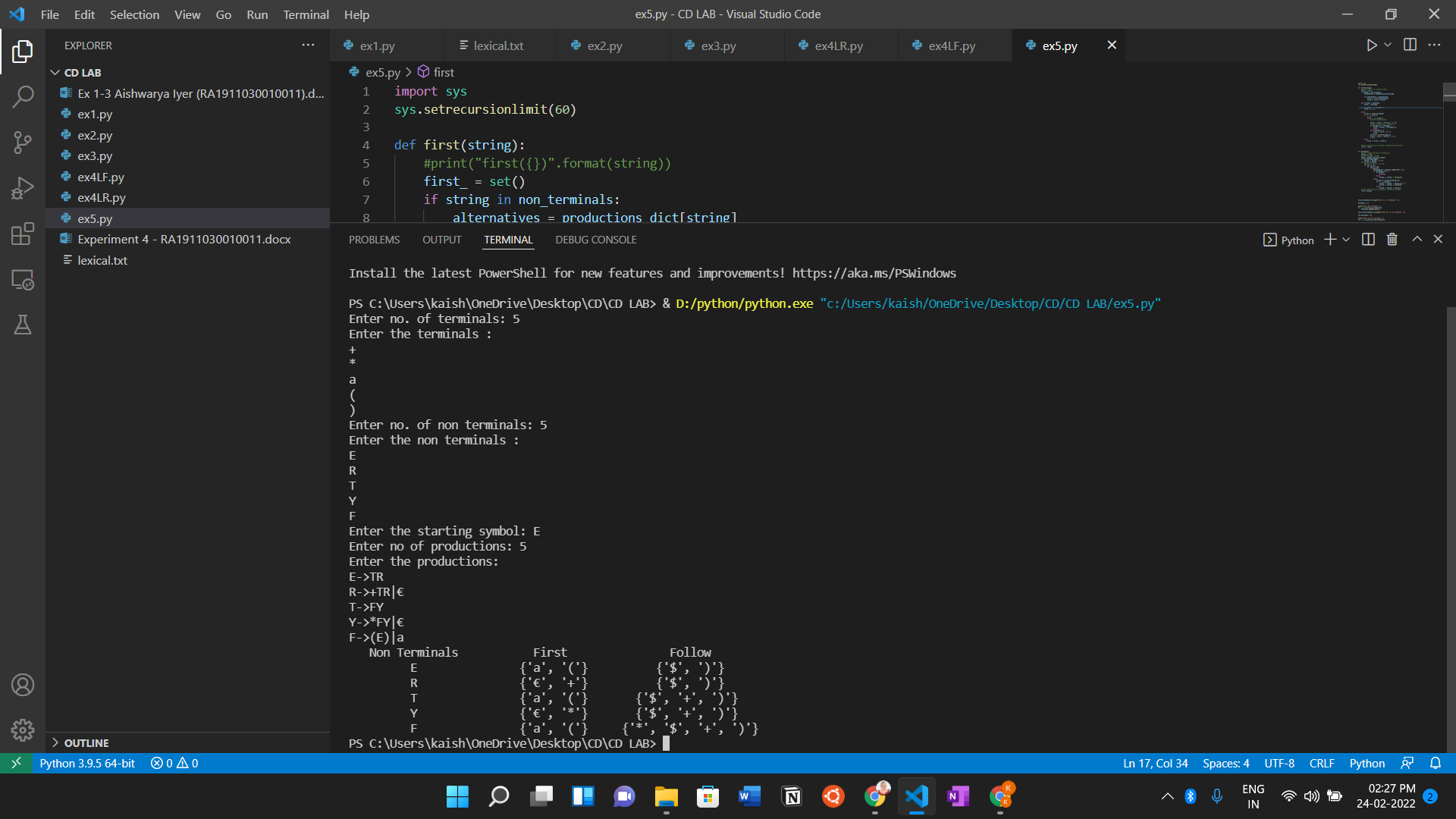
#print("FOLLOW", FOLLOW)

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])))

**Output:**

****

**Result:**

Thus, we have implemented a program to display the FIRST and FOLLOW for a given set of production in Python.

**Experiment 6**

**Topic: Predictive Parsing Table**

**Aim:**

To design and implement a program that displays the Predictive Parsing Table for given productions in Python.

**Algorithm:**

1. Start.
2. Decide the grammar and productions for the problem.
3. Write a function to remove the left recursion and left factoring from the given productions.
4. Write functions to compute the FIRST and FOLLOW of the given productions.
5. Write a function to construct the Predictive Parsing Table according to the following rules:
   1. For each a in FIRST(α), add A->α to M[A,a]
   2. If € is in FIRST(α), add A->α to M[A,b] for each token b in FOLLOW(A)
   3. If € is in FIRST(α) and $ is in FOLLOW(A) then add A->α to M[A,$]
   4. Mark all other entries of M as “error”
6. Display this Predictive Parsing Table.
7. Stop the program.

**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

    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+=["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}'

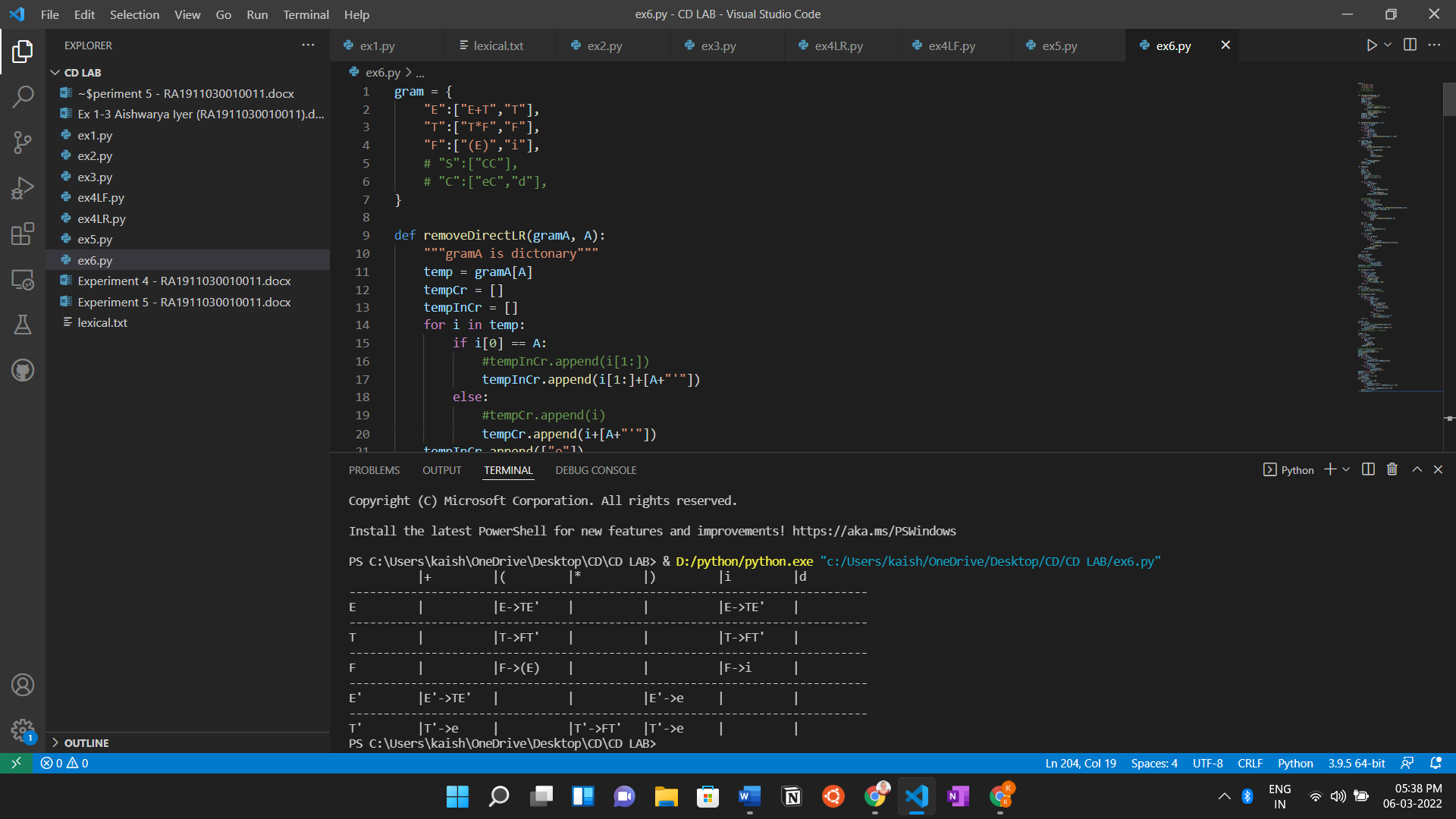
        else:

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

    print(f'{"-":-<76}')

    print(toprint)

**Output:**

****

**Result:**

Thus, we have successfully constructed a Predictive Parsing Table for the given set of productions.

**Experiment 7**

**Topic: Shift Reduce Parsing**

**Aim:**

To design and implement a program that performs Shift Reduce Parsing for given productions in Python.

**Algorithm:**

1. Start.
2. Get the input expression and store it in the input buffer.
3. Using stack (push & pop) operation shift and reduce symbols with respect to production rules available.
4. Continue the process till symbol shift and production rule reduce reaches the start symbol.
5. Display the Stack Implementation table with corresponding Stack actions with input symbols.

**Code:**

gram = {

    "E":["2E2","3E3","4"]

}

starting\_terminal = "E"

inp = "2324232$"

"""

# example 2

gram = {

    "S":["S+S","S\*S","i"]

}

starting\_terminal = "S"

inp = "i+i\*i"

"""

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')

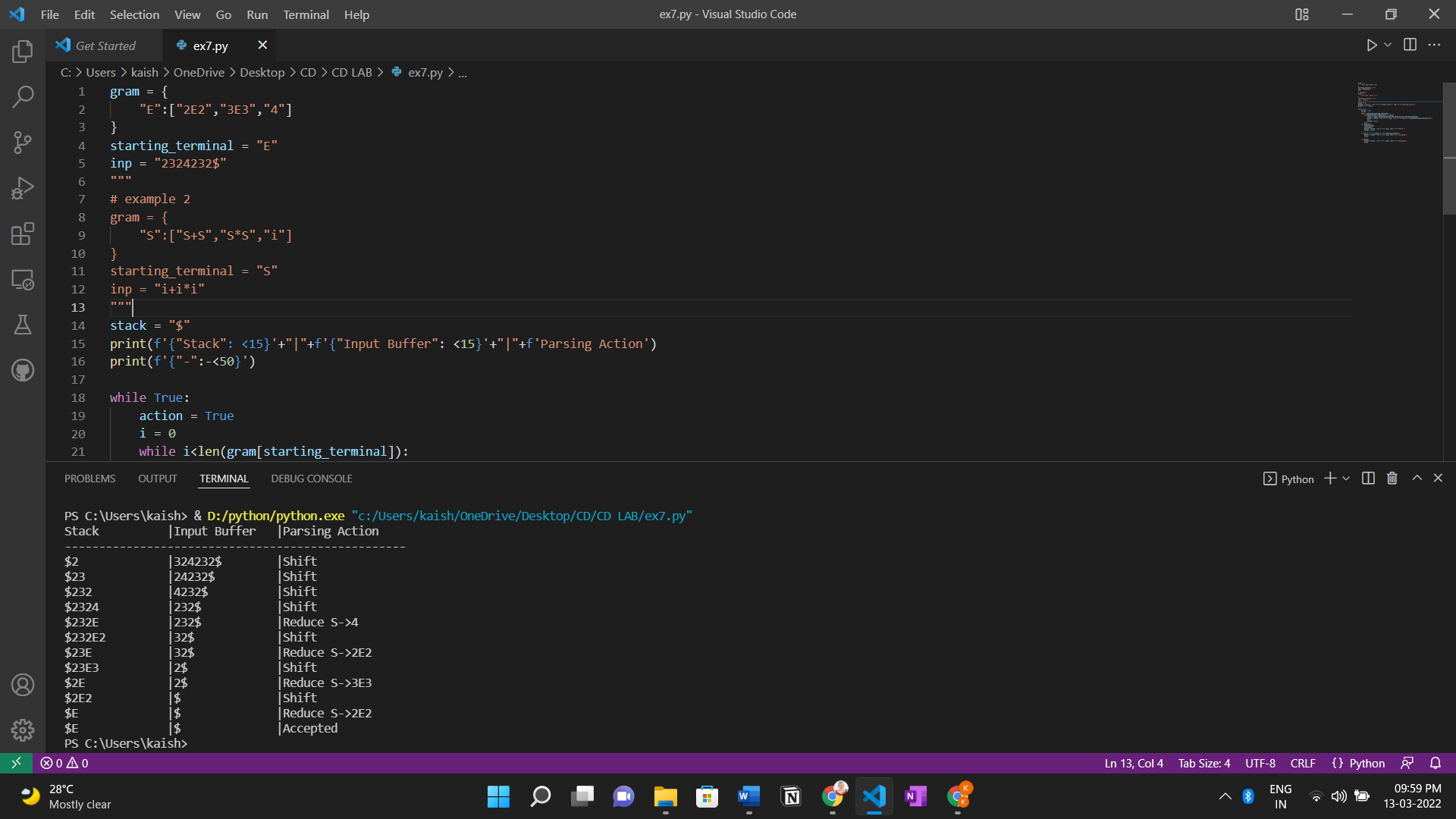
        break

**Output:**

**Grammar:**

"E":["2E2","3E3","4"]

input = "2324232$"



**Result:**

Thus, we have designed a program that performs Shift Reduce Parsing for the given set of productions in python.

**Experiment 8**

**Topic: Computation of LEADING and TRAILING**

**Aim:**

To design and implement a program that computes LEADING and TRAILING for given set of productions in Python.

**Algorithm:**

1. Start.

2. Get the grammar and productions.

3. For Computation of LEADING:

* ‘a’ is in Leading(A) if A -> γaδ where γ is ε or any Non-Terminal
* If’ ‘a’ is in Leading(B) and A -> Bα, then a in Leading(A)

4. For Computation of TRAILING:

* ‘a’ is in Trailing(A) if A -> γaδ where δ is ε or any Non-Terminal
* If ‘a’ is in Trailing(B) and A -> αB, then a in Trailing(A)

5. Display both of them respectively.

**Code:**

a = ["E->E+T",

"E->T",

"T->T\*F",

"T->F",

"F->(E)",

"F->i"]

rules = {}

terms = []

for i in a:

temp = i.split("->")

terms.append(temp[0])

try:

rules[temp[0]] += [temp[1]]

except:

rules[temp[0]] = [temp[1]]

terms = list(set(terms))

print(rules,terms)

def leading(gram, rules, term, start):

s = []

if gram[0] not in terms:

return gram[0]

elif len(gram) == 1:

return [0]

elif gram[1] not in terms and gram[-1] is not start:

for i in rules[gram[-1]]:

s+= leading(i, rules, gram[-1], start)

s+= [gram[1]]

return s

def trailing(gram, rules, term, start):

s = []

if gram[-1] not in terms:

return gram[-1]

elif len(gram) == 1:

return [0]

elif gram[-2] not in terms and gram[-1] is not start:

for i in rules[gram[-1]]:

s+= trailing(i, rules, gram[-1], start)

s+= [gram[-2]]

return s

leads = {}

trails = {}

for i in terms:

s = [0]

for j in rules[i]:

s+=leading(j,rules,i,i)

s = set(s)

s.remove(0)

leads[i] = s

s = [0]

for j in rules[i]:

s+=trailing(j,rules,i,i)

s = set(s)

s.remove(0)

trails[i] = s

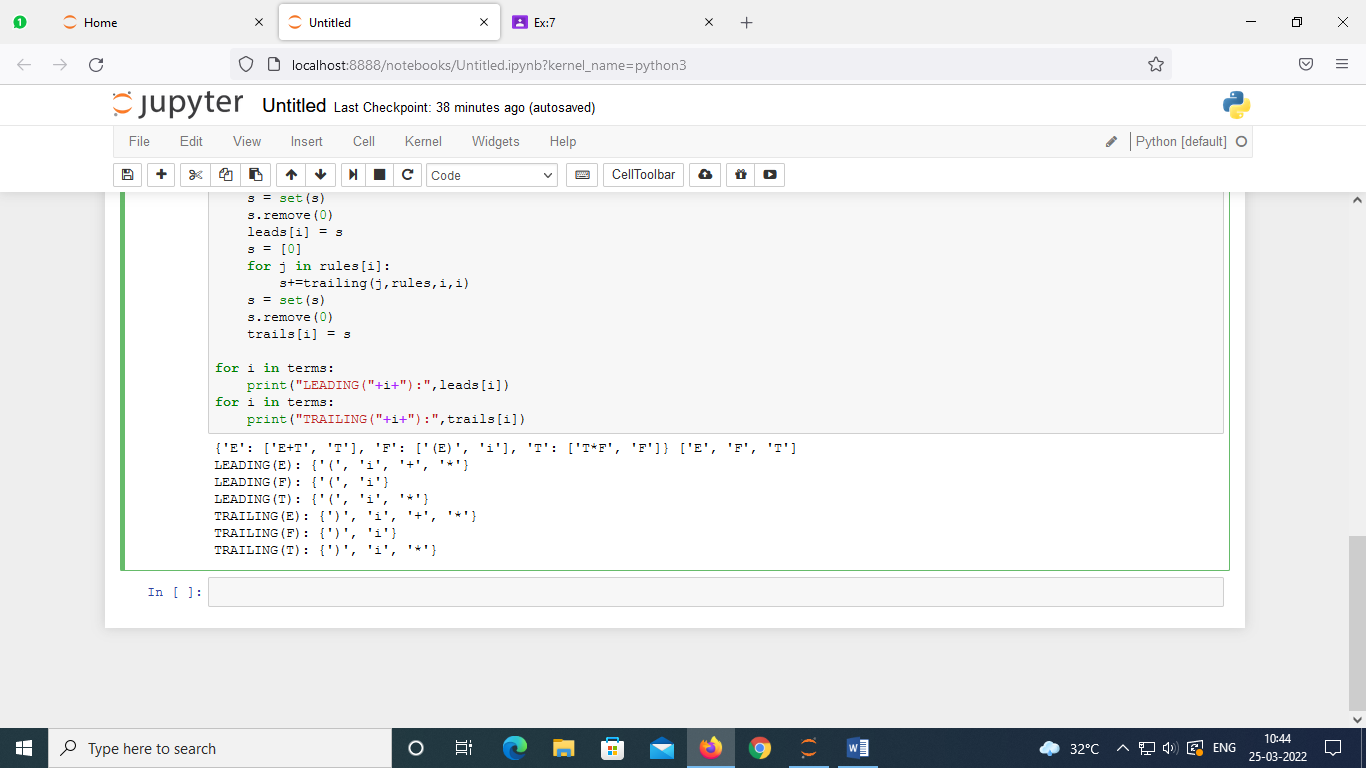
for i in terms:

print("LEADING("+i+"):",leads[i])

for i in terms:

print("TRAILING("+i+"):",trails[i])

**Output:**



**Result:**

Thus, we have designed a program that computes the LEADING and TRAILING for the given set of productions.

**Experiment 9**

**Topic: LR(0) Computation**

**Aim:**

To design and implement a program that performs LR(0) Computation for given productions in Python.

**Algorithm:**

The LR Parser is a Shift-reduce Parser that makes use of a Deterministic Finite Automata, recognizing the Set Of All Viable Prefixes by reading the stack from Bottom To Top.

If a Finite-State Machine that recognizes viable prefixes of the right sentential forms is constructed, it can be used to guide the handle selection in the Shift-reduce Parser.

Handle: Handle is a substring that matches the body of a production

Handle is a Right Sentential Form + position where reduction can be performed + production used for reduction.

LR(0) Items

-An LR(0) Item of a Grammar G is a Production of G with a Dot (.) at some position of the right side. Production A → XYZ yields the Four items:

1. A→•XYZ We hope to see a string derivable from XYZ next on the input.

2. A→X•YZ We have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ next on the input.

3. A→XY•Z

4. A→XYZ•

❖ The production A→ε generates only one item, A→•.

❖ Each of this item is a Viable prefixes

❖ Closure Item : An Item created by the closure operation on a state.

❖ Complete Item : An Item where the Item Dot is at the end of the RHS.

**Code:**

gram = {

    "S":["ab"]

}

start = "S"

terms = ["a","b","$"]

non\_terms = []

for i in gram:

    non\_terms.append(i)

gram["S'"]= [start]

new\_row = {}

for i in terms+non\_terms:

    new\_row[i]=""

non\_terms += ["S'"]

# each row in state table will be dictionary {nonterms ,term,$}

stateTable = []

# I = [(terminal, closure)]

# I = [("S","A.A")]

def Closure(term, I):

    if term in non\_terms:

        for i in gram[term]:

            I+=[(term,"."+i)]

    I = list(set(I))

    for i in I:

        # print("." != i[1][-1],i[1][i[1].index(".")+1])

        if "." != i[1][-1] and i[1][i[1].index(".")+1] in non\_terms and i[1][i[1].index(".")+1] != term:

            I += Closure(i[1][i[1].index(".")+1], [])

    return I

Is = []

Is+=set(Closure("S'", []))

countI = 0

omegaList = [set(Is)]

while countI<len(omegaList):

    newrow = dict(new\_row)

    vars\_in\_I = []

    Is = omegaList[countI]

    countI+=1

    for i in Is:

        if i[1][-1]!=".":

            indx = i[1].index(".")

            vars\_in\_I+=[i[1][indx+1]]

    vars\_in\_I = list(set(vars\_in\_I))

    # print(vars\_in\_I)

    for i in vars\_in\_I:

        In = []

        for j in Is:

            if "."+i in j[1]:

                rep = j[1].replace("."+i,i+".")

                In+=[(j[0],rep)]

        if (In[0][1][-1]!="."):

            temp = set(Closure(i,In))

            if temp not in omegaList:

                omegaList.append(temp)

            if i in non\_terms:

                newrow[i] = str(omegaList.index(temp))

            else:

                newrow[i] = "s"+str(omegaList.index(temp))

            print(f'Goto(I{countI-1},{i}):{temp} That is I{omegaList.index(temp)}')

        else:

            temp = set(In)

            if temp not in omegaList:

                omegaList.append(temp)

            if i in non\_terms:

                newrow[i] = str(omegaList.index(temp))

            else:

                newrow[i] = "s"+str(omegaList.index(temp))

            print(f'Goto(I{countI-1},{i}):{temp} That is I{omegaList.index(temp)}')

    stateTable.append(newrow)

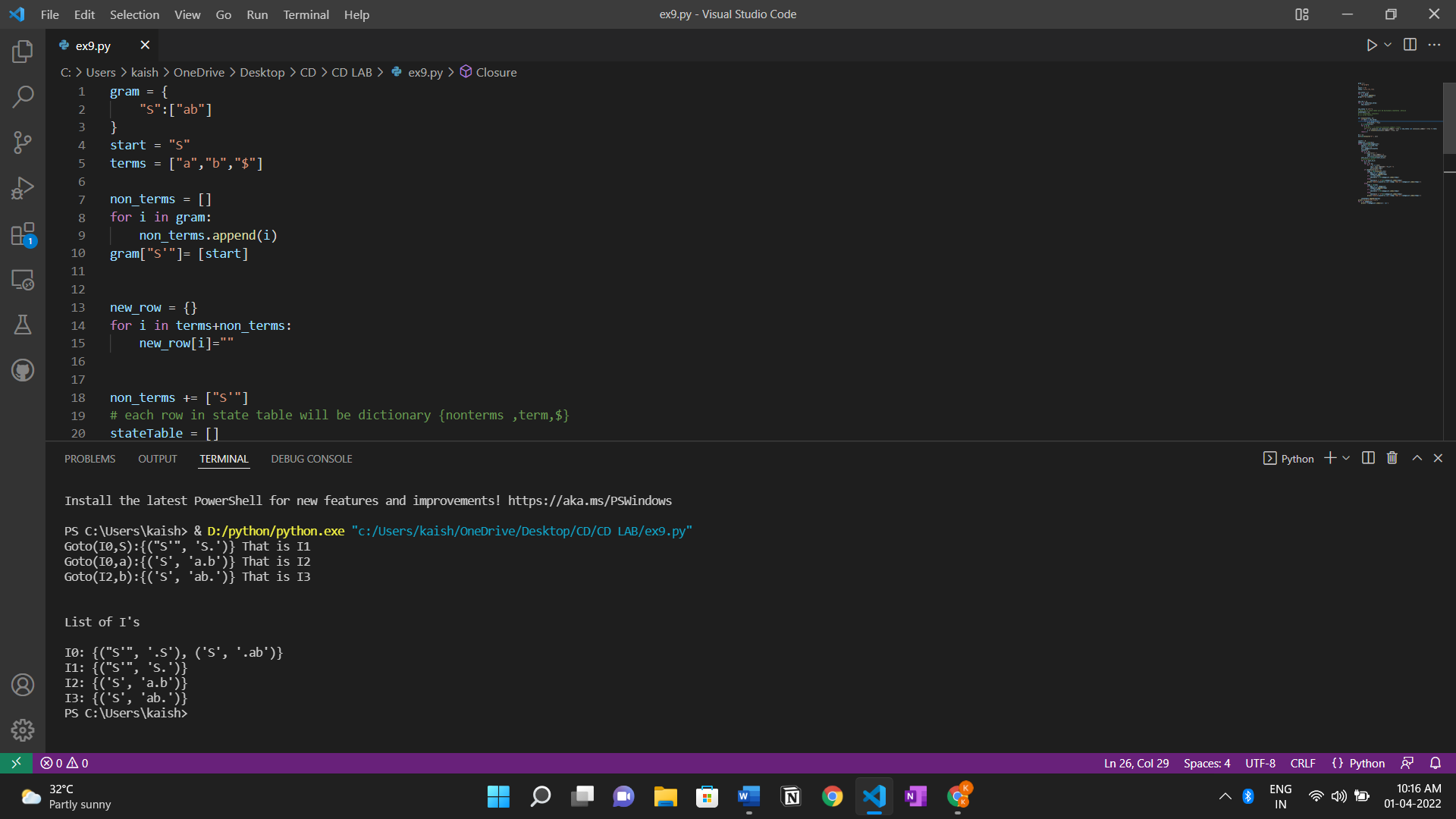
print("\n\nList of I's\n")

for i in omegaList:

    print(f'I{omegaList.index(i)}: {i}')

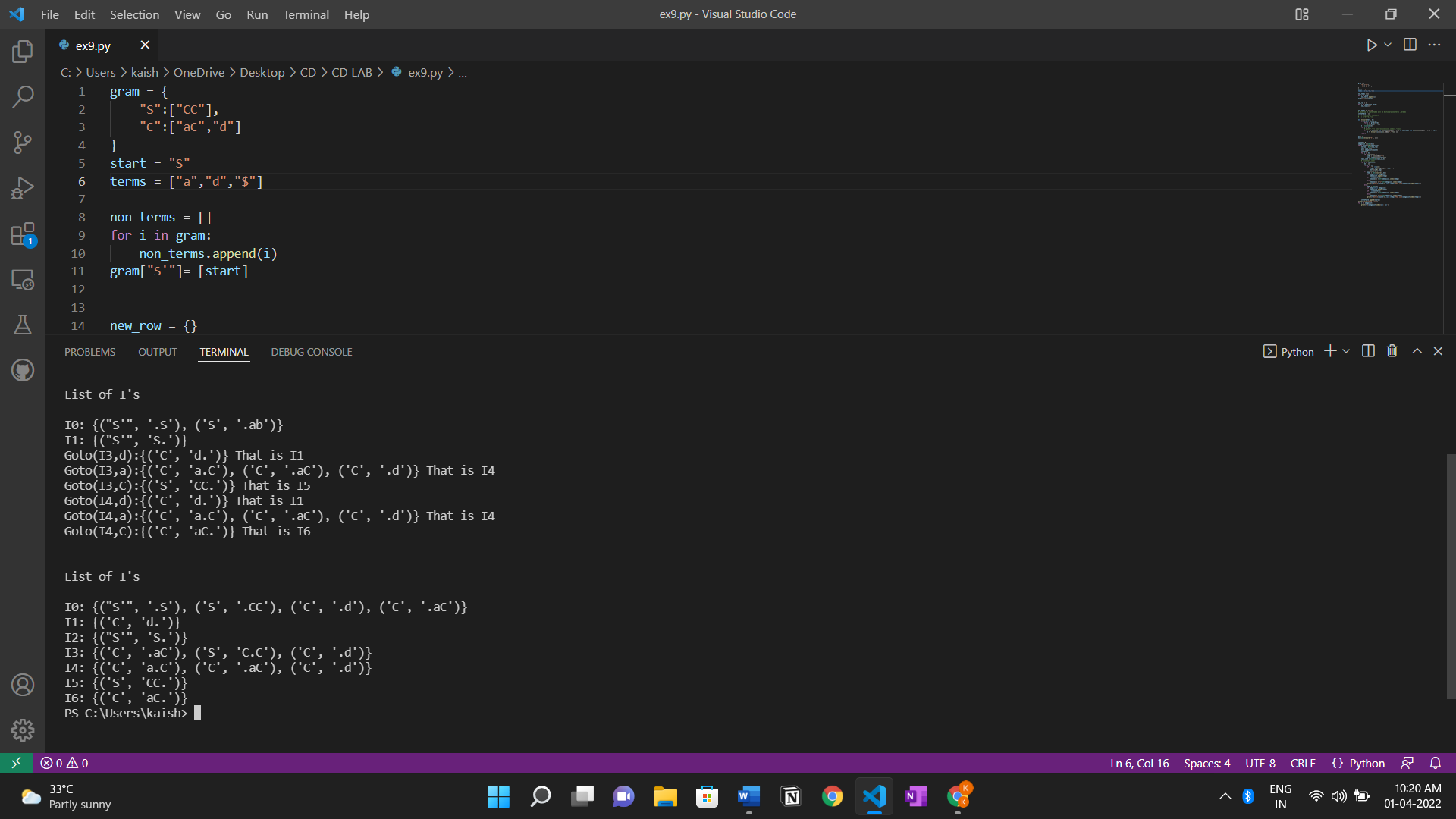
**Output:**

**Grammar: S->ab**



**Grammar: S->CC**

**C->aC|d**



**Result:**

Thus, we have implemented a program that performs LR(0) Computation for the given productions in python.

**Experiment 10**

**Topic: Intermediate code generation – Prefix and Postfix**

**Aim:**

To design and implement a program that performs Intermediate code generation – Prefix and Postfix in Python.

**Algorithm:**

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).

The benefits of using machine independent intermediate code are:

* Because of the machine independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
* Retargeting is facilitated
* It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.

**Postfix Notation –**

The ordinary (infix) way of writing the sum of a and b is with operator in the middle : a + b  
The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation the operator follows the operand.

If we generate machine code directly from source code then for n target machine we will have n optimizers and n code generators but if we will have a machine independent intermediate code,  
we will have only one optimizer. Intermediate code can be either language specific (e.g., Bytecode for Java) or language. independent (three-address code).

Example – The postfix representation of the expression (a – b) \* (c + d) + (a – b) is:   ab – cd + \*ab -+.

1. **Three-Address Code –**  
   A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.

Example – The three address code for the expression a + b \* c + d :

T 1 = b \* c  
T 2 = a + T 1  
T 3 = T 2 + d

T 1 , T 2 , T 3 are temporary variables.

1. **Syntax Tree –**Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.

Example –  
x = (a + b \* c) / (a – b \* c)

**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

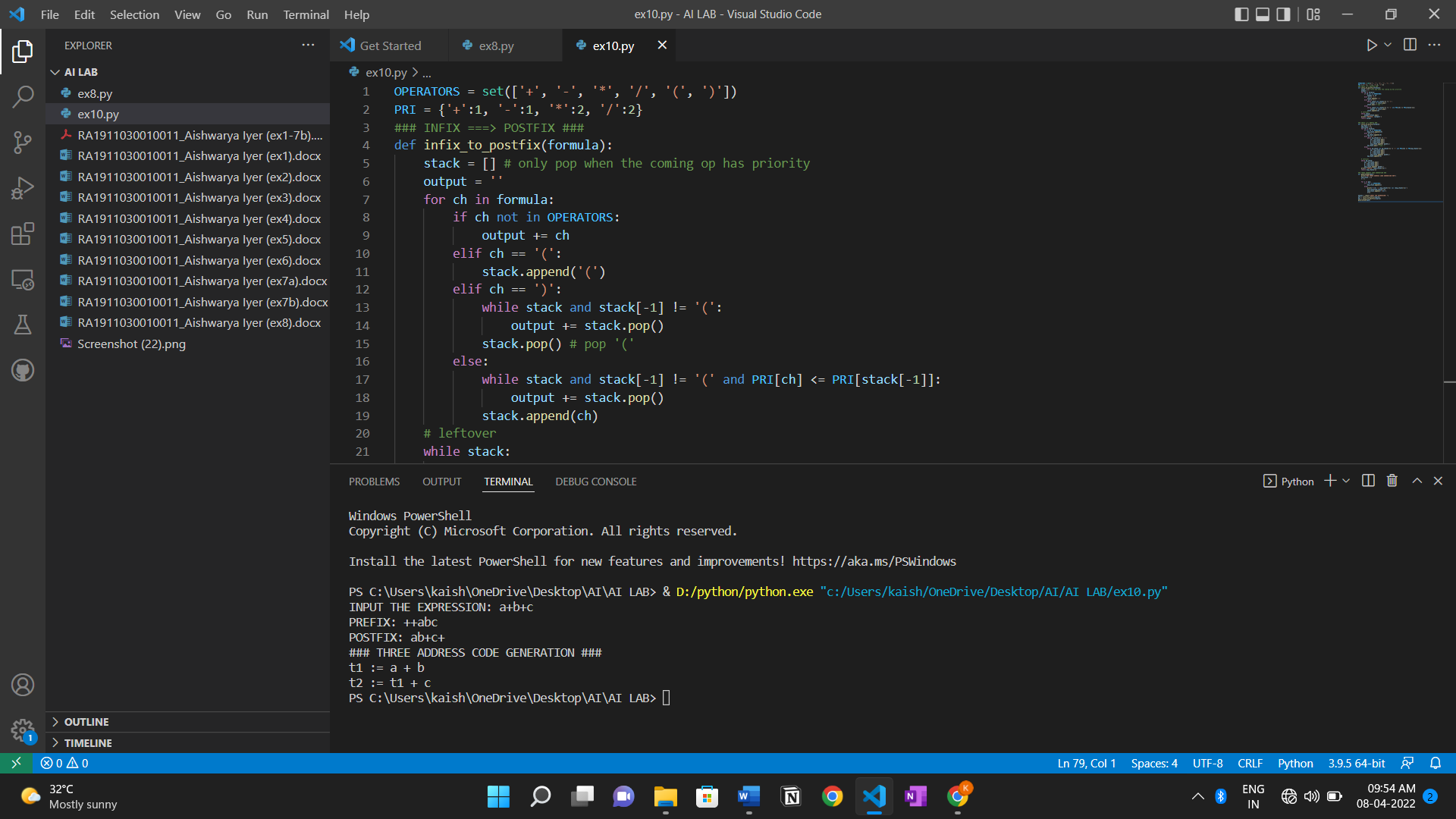
expres = input("INPUT THE EXPRESSION: ")

pre = infix\_to\_prefix(expres)

pos = infix\_to\_postfix(expres)

generate3AC(pos)

**Output:**



**Result:**

Thus, we have generated the Intermediate Code.

**Experiment 11**

**Topic: Intermediate code generation – Quadruple, Triple, Indirect triple**

**Aim:**

To design and implement a program that performs Intermediate code generation – Quadruple, Triple, Indirect triple in Python.

**Algorithm:**

1. **Quadruple** – It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.
2. **Triples** – This representation doesn’t make use of extra temporary variable to represent a single operation instead when a reference to another triple’s value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.
3. **Indirect** **Triples** – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

**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()

    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)

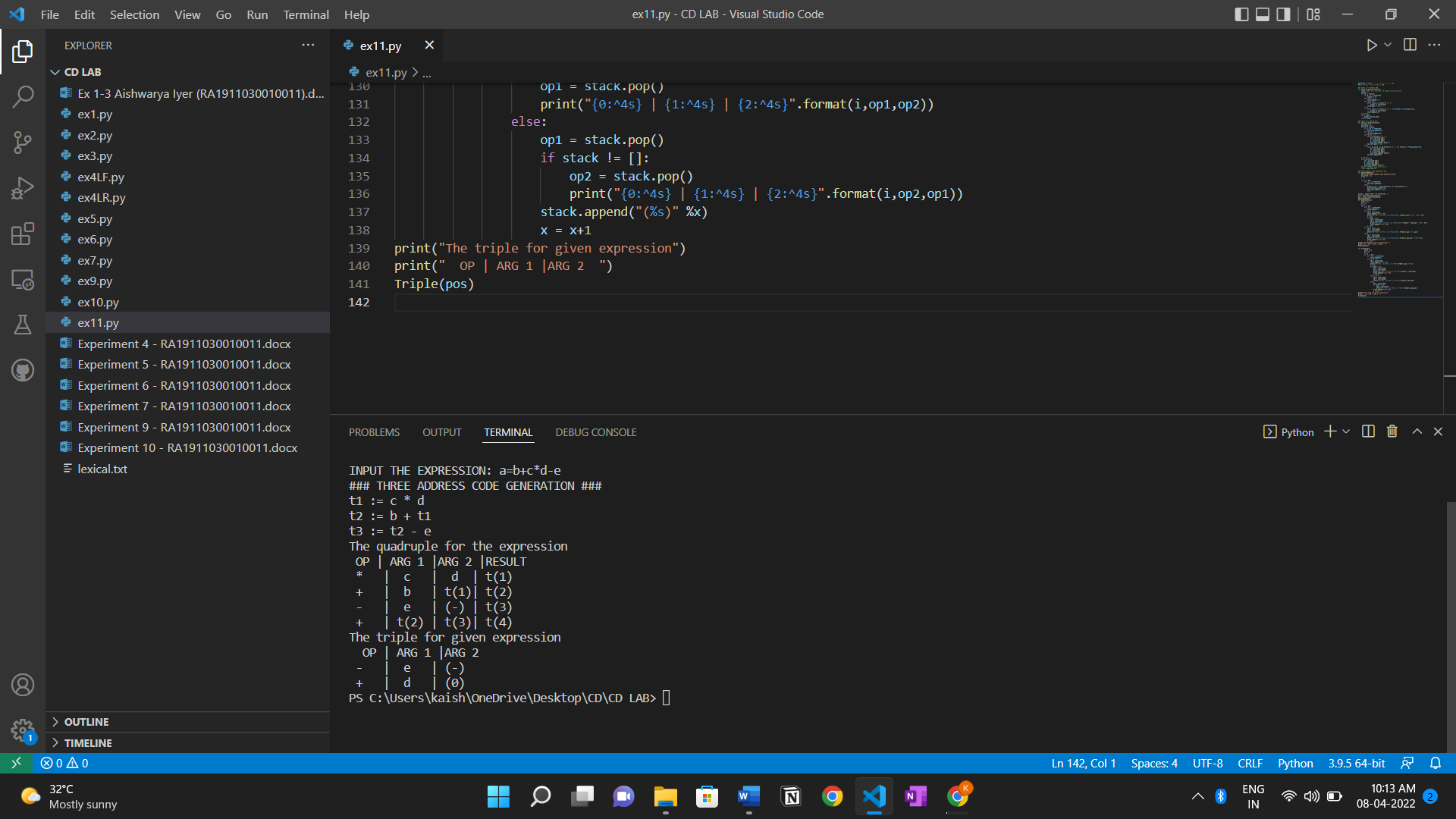
                    x = x+1

print("The triple for given expression")

print("  OP | ARG 1 |ARG 2  ")

Triple(pos)

**Output:**



**Result:**

Thus, we have generated Intermediate code for the given input expression.