

**PRACTICAL 3**

**AIM: Write a program to validate DFA.**

# starts with 0

m=[['B','C'],['B','B'],['C','C']]

inp=input("enter string : ")

start='A'

current=start

for l in list(inp):

i=int(l)

print("Current",current)

if current=='A':

current=m[0][i]

if current=='B':

current=m[1][i]

if current=='C':

current=m[2][i]

if current=='B':

print('valid')

else:

print('invalid')

#pip install automata-lib

from automata.fa.dfa import DFA

dfa = DFA(states={'A', 'B', 'C'},

input\_symbols={'0', '1'}, transitions={

'A': {'0': 'B', '1': 'C'},

'B': {'0': 'B', '1': 'B'},

'C': {'0': 'C', '1': 'C'}

},

initial\_state='A',

final\_states={'B'}

)

inputstring=input("Enter string: ")

if(dfa.accepts\_input(inputstring)):

print('valid')

else:

print('invalid')

#Code :2

# starts with 0 ends with 1

m=[['B','D'],['B','C'],['B','C'],['D','D']]

inp=input("enter string : ")

start='A'

current=start

print("Start:",current)

for l in list(inp):

i=int(l)

if current=='A':

current=m[0][i]

if current=='B':

current=m[1][i]

if current=='C':

current=m[2][i]

if current=='D':

current=m[3][i]

print("Current:",current)

if current=='C':

print('valid')

else:

print('invalid')

#pip install automata-lib

from automata.fa.dfa import DFA

dfa = DFA(states={'A', 'B', 'C', 'D'},

input\_symbols={'0', '1'},

transitions={'A': {'0': 'B', '1': 'D'},

'B': {'0': 'B', '1': 'C'}, 'C': {'0': 'B', '1': 'C'},

'D': {'0': 'D', '1': 'D'}}, initial\_state='A',

final\_states={'C'})

inputstring=input("Enter string: ")

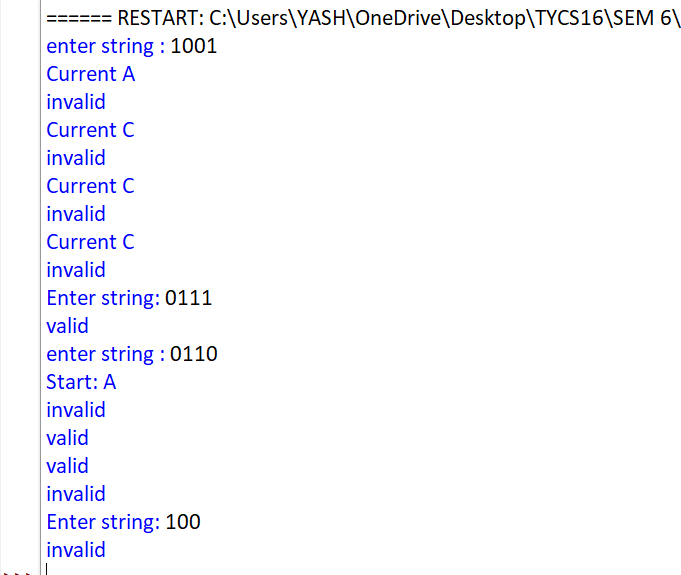
if(dfa.accepts\_input(inputstring)):

print('valid')

else:

print('invalid')

OUTPUT:



**PRACTICAL 4**

**AIM: Write a program to implement 3-Address Code.**

l=[]

def push(a):

l.append(a)

def pop():

p=l[-1]

del l[-1]

return p

def is\_empty():

if len(l)==0:

return True

else:

return False

def three\_add\_code(expr):

ts=[]

op=[]

arg1=[]

arg2=[]

i=0

for t1 in expr:

if(t1 in "ABCDEFGHIJKLMNOPQRSTUVWXTZ"):

push(t1)

else:

arg2.append(pop())

arg1.append(pop())

op.append(t1)

s="t"+str(i)

push(s)

ts.append(s)

i=i+1

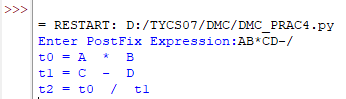
for a in range (0,i):

print(ts[a],"=",arg1[a],"",op[a],"",arg2[a])

e=input("Enter PostFix Expression:")

three\_add\_code(e)

**OUTPUT:**

****

**PRACTICAL NO 5**

**AIM:WRITE A PROGRAM TO IMPLEMENT TRIPLE.**

**CODE:**

l=[]

def push(a):

l.append(a)

def pop():

p=l[-1]

del l[-1]

return p

def is\_empty():

if len(l)==0:

return True

else:

return False

def triple(expr):

ts=[]

op=[]

arg1=[]

arg2=[]

i=0

for t1 in expr:

if(t1 in "ABCDEFGHIJKLMNOPQRSTUVWXTZ"):

push(t1)

else:

arg2.append(pop())

arg1.append(pop())

op.append(t1)

s="("+str(i)+")"

push(s)

ts.append(s)

i=i+1

print("\t arg1\t op\t arg2\t ")

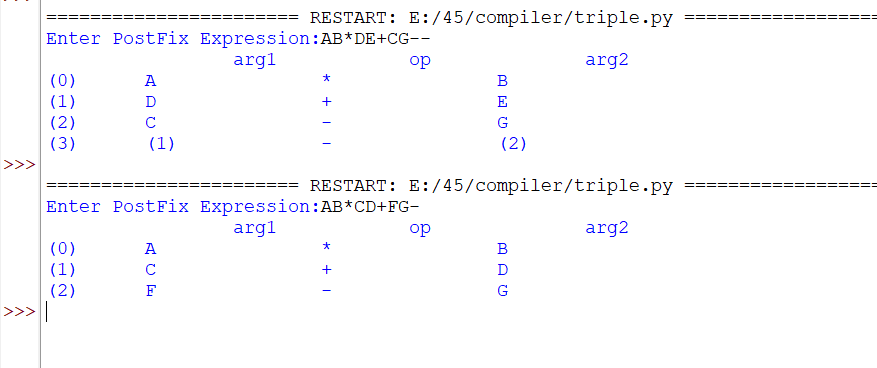
for a in range(0,i):

print (ts[a],"\t",arg1[a],"\t",op[a],"\t",arg2[a],"\t")

e=input("Enter triple: ")

print(triple(e))

**OUTPUT:**



**Practical 6**

**Aim: Write a program to implement Quadruple.**

**CODE:**

l=[]

def push(a):

l.append(a)

def pop():

p=l[-1]

del l[-1]

return p

def is\_empty():

if len(1)==0:

return True

else:

return False

def quadruple(expr):

ts=[]

op=[]

arg1=[]

arg2=[]

i=0

for t1 in expr:

if(t1 in "ABCDEFGHIJKLMNOPQRSTUVWXYZ"):

push(t1)

else:

arg2.append(pop())

arg1.append(pop())

op.append(t1)

s="t"+str(i)

push(s)

ts.append(s)

i=i+1

print("arg\t op\t arg2\t result")

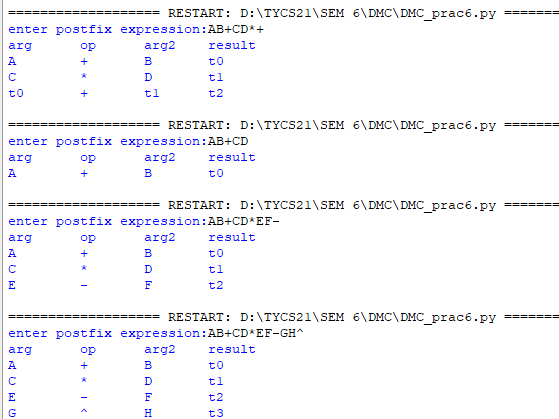
for a in range(0,i):

print(arg1[a],"\t",op[a],"\t",arg2[a],"\t",ts[a])

e=input("enter postfix expression:")

quadruple(e)

**OUTPUT:**



**Practical 7**

**Aim: Write a program to implement Infix to Postfix.**

**CODE:**

l=[]

def push(a):

l.append(a)

def pop():

p=l[-1]

del l[-1]

return p

def is\_empty():

if len(l)==0:

return True

else:

return False

def peek():

s=l[-1]

return s

def in\_to\_post(expr):

prec = {}

prec["\*"] = 3

prec["/"] = 3

prec["+"] = 2

prec["-"] = 2

prec["("] = 1

postfix = []

stack = []

for token in expr:

if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789" or token in "abcdefghijklmnopqrstuvwxyz":

postfix.append(token)

elif token == "(":

stack.append(token)

elif token == ")":

top\_token = stack.pop()

while top\_token != "(":

postfix.append(top\_token)

top\_token = stack.pop()

else:

while stack and prec[stack[-1]] >= prec[token]:

postfix.append(stack.pop())

stack.append(token)

while stack:

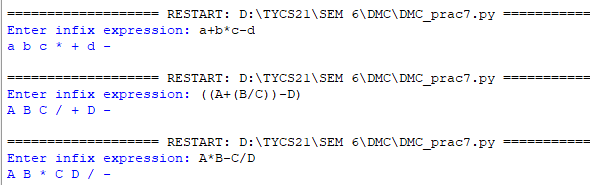
postfix.append(stack.pop())

return " ".join(postfix)

e = input("Enter infix expression: ")

print(in\_to\_post(e))

**OUTPUT:**



**Practical 8**

**Aim: Write a program to implement Postfix to Infix.**

**CODE:**

l=[]

def push(a):

l.append(a)

def pop():

p=l[-1]

del l[-1]

return p

def is\_empty():

if len(l)==0:

return True

else:

return False

def post\_to\_in(expr):

postfix = []

for t1 in expr:

if (t1 in "ABCDEFGHIJKLMNOPQRSTUVWXYZ") or (t1 in "0123456789"):

push(t1)

else:

a=pop()

b=pop()

s=str(b)+t1+str(a)

push(s)

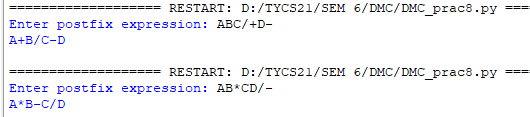
postfix=pop()

return postfix

e=input("Enter postfix expression: ")

print(post\_to\_in(e))

**OUTPUT:**



**Practical 10**

**Aim: Generation of machine code.**

**CODE:**

expns=["T=a+b","T=T\*c","T=T/d"]

for a in expns:

print(a) #loop iterates over each expression in the list

print("\*"\*50)

def cd(op,x,y): #takes 3 arguments (operator, x and y operands)

if op=='+': #Move register add register(assembly language)

print("MOV REG, "+x+"\nADD REG, "+y) #print assembly instructions (move operands into

if op=='\*': the register a and add register b)

print("MOV REG, "+x+"\nMUL REG, "+y)

if op=='/':

print("MOV REG, "+x+"\nDIV REG, "+y)

print("ST T,REG") #store it into T then move register T

for i in range(0,len(expns)):

s=expns[i].split('=') # t,a+b

exp=s[1] #a+b

op=exp[1] #+

x=exp[0] #a

y=exp[2] #b

cd(op,x,y)

**OUTPUT:**

