**Name: Sreejith Nair**

**Roll No: 19**

**Design and implementation of Modern Compilers**

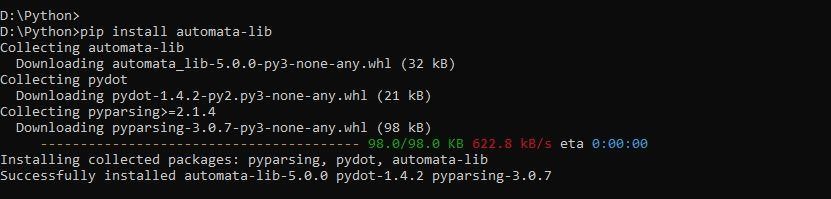
INDEX

|  |  |
| --- | --- |
| SR  NO | TITLE |
| 1 | Write a program to construct NDFA |
| 2 | Write a program to convert the given Right linear Grammar to Left Linear Grammar. |
| 3 | Write a code to generate DAG for input arithmetic expression. |
| 4 | Write a code to for triples. |
| 5 | Write a code for postfix evaluation. |
| 6 | Write a code to generate 3 address code |
| 7 | Write a code to demonstrate loop jamming for the given code sequence containing loop. |
| 8 | Write a code to demonstrate loop unrolling for the given code sequence containing loop. |

# Practical NO 1

Aim : Write a program to construct NDFA

Install package automata-lib By using the following command: pip install automata-lib



Code: from automata.fa.nfa import NFA class NDFA: def \_\_init\_\_(self):

state\_set = set(input("Enter state set>\t"))

input\_symbols = set(input("Enter input symbol set>\t")) initial\_state = input("Enter the initial state>\t") final\_states = set(input("Enter the final state(s)>\t")) rule\_count = int(input("Enter the number of rules you want to add>\t"))

rules = []

for counter in range(rule\_count):

rules.append(input("Enter rule " + str(counter + 1) +

">\t").replace(" ", ""))

rules = self.get\_transitions(rules)

self.nfa = NFA( states = state\_set,

input\_symbols = input\_symbols, transitions = rules, initial\_state = initial\_state, final\_states = final\_states

)

del state\_set, input\_symbols, initial\_state, final\_states, rules.

def get\_transitions(self, rules): rules = [i.split("-") for i in rules] rules\_dict = {}

for rule in rules: if rule[0] not in rules\_dict:

rules\_dict[rule[0]] = {rule[1][1]:rule[1][0]} print("If:", rules\_dict) else:

rules\_dict[rule[0]][rule[1][0]] = rule[1][1] print("Else:", rules\_dict) return rules\_dict

def print\_stats(self):

print("\n\nSet of states are > ", self.nfa.states) print("Input symbols are > ", self.nfa.input\_symbols) print("Transitions are > ") for transition in self.nfa.transitions:

print(transition, self.nfa.transitions[transition]) print("Initial state > ", self.nfa.initial\_state) print("Final states > ", self.nfa.final\_states)

def print\_transition\_table(self):

input\_symbols = list(self.nfa.input\_symbols) transitions = self.nfa.transitions

print("\n\nTransition table is > ")

#print(f"States\t\t{input\_symbols[0]}\t\t{input\_symbols[1]}") print("States\t\t" + str(input\_symbols[0]) + "\t\t" + str(input\_symbols[1])) for transition in transitions: for input\_symbol in input\_symbols: try:

temp = transitions[transition][input\_symbol] del temp except KeyError:

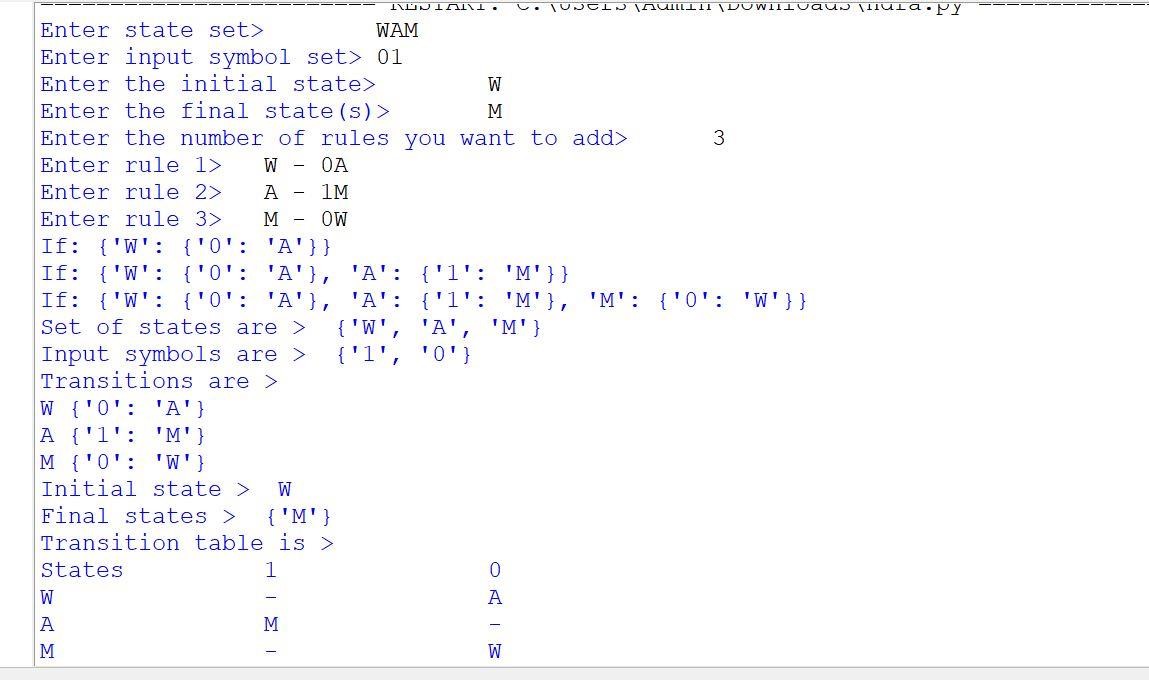
transitions[transition][input\_symbol] = "-"

#print(f"{transition}\t\t{transitions[transition][input\_symbols [0]]}\t\t{transitions[transition][input\_symbols[1]]}")

print(transition + "\t\t" +

transitions[transition][input\_symbols[0]] + "\t\t" + transitions[transition][input\_symbols[1]]) del input\_symbols, transitions if \_\_name\_\_ == "\_\_main\_\_":

ndfa = NDFA() ndfa.print\_stats() ndfa.print\_transition\_table() Output:



# PRACTICAL NO 2

Aim: Write a program to convert the given Right linear grammar to Left Linear Grammar form.

CODE:

def get\_transitions(rules):

my\_dict={}

ld='' res=dict()

r='' for i in rules:

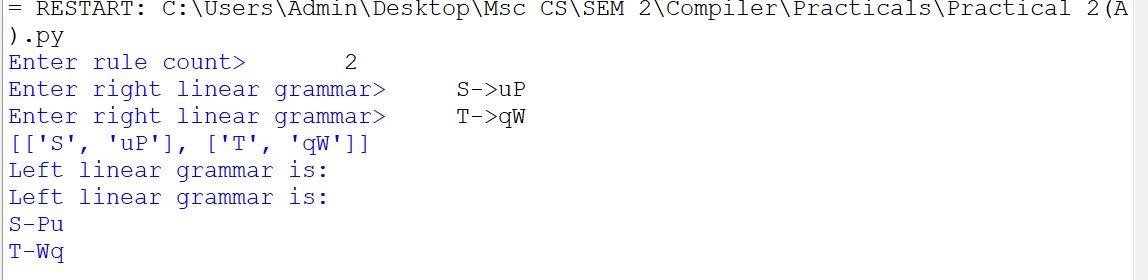
my\_dict[i[0]]=[i[1][1],i[1][0]] for sub in my\_dict: if isinstance(my\_dict[sub],list): res[sub]=ld.join([str(ele) for ele in my\_dict[sub]]) print("Left linear grammar is:") for item in res: r+=item+"-"+str(res[item])+"\n" print(str(r))

rule\_count=int(input("Enter rule count>\t"))

rules=[] for i in range(rule\_count):

rules.append(input("Enter right linear grammar"+">\t")) rules=[i.split("->") for i in rules] print(rules) get\_transitions(rules)

OUTPUT:



# PRACTICAL NO 3

Aim: Write a code to generate DAG for input arithmetic expression.

CODE:

def funct1(x): main=[] for i in range(0,x): y=input() main.append(y)

print("Label Operator left Right") for i in range(0,x):

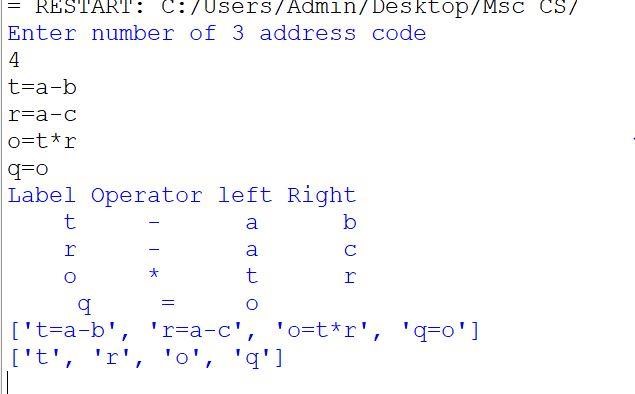
q=main[i] if q[0] not in res: res.append(q[0]) if(len(q)>3):

print(" ",q[0]," ",q[3]," ",q[2]," ",q[4]) else:

print(" ",q[0]," ",q[1]," ",q[2]," ") print(main) print(res)

print("Enter number of 3 address code") x=input() x=int(x) res=[] funct1(x)

Output:



# PRACTICAL NO 4

Aim: Write a code for triples.

Code:

def funct1(x): main=[] for i in range(0,x): y=input() main.append(y)

print("Address operator argument 1 argument2")

for i in range(0,x):

g=main[i] if g[0] not in res: res.append(g[0]) e=funct2(g[2]) if(len(g)>3):

r=funct2(g[4])

print(" (",i,")"," ",g[3]," ",e," ",r) else:

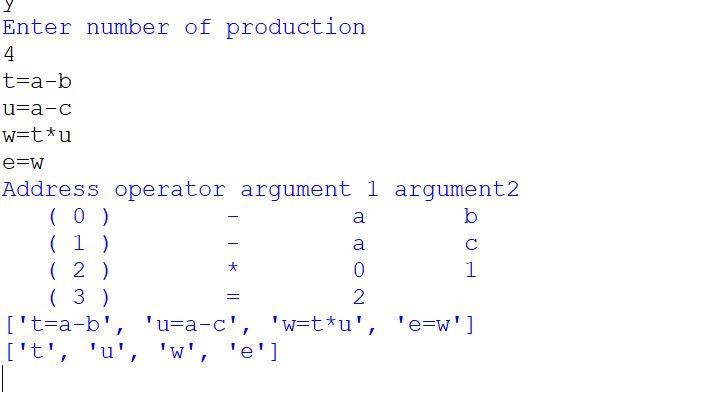
print(" (",i,")"," ",g[1]," ",e," ") print(main) print(res) def funct2(g):

try:

z=res.index(g) return(z) except: return(g)

print("Enter number of production") x=input() x=int(x) res=[] funct1(x)

Output:



# PRACTICAL NO 5

Aim: Write the code for Postfix Evaluation CODE:

def postfix\_evaluation(s):

s=s.split() n=len(s) stack=[] for i in range(n): if s[i].isdigit(): stack.append(int(s[i])) elif s[i]=="+": a=stack.pop() b=stack.pop() stack.append(int(a)+int(b)) elif s[i]=="\*": a=stack.pop() b=stack.pop() stack.append(int(a)\*int(b)) elif s[i]=="/":

a=stack.pop() b=stack.pop() stack.append(int(a)/int(b)) elif s[i]=="-": a=stack.pop() b=stack.pop() stack.append(int(a)-int(b)) return stack.pop()

s="8 7 8 \* + 4 -" val=postfix\_evaluation(s) print(val)

OUTPUT:



# PRACTICAL NO 6

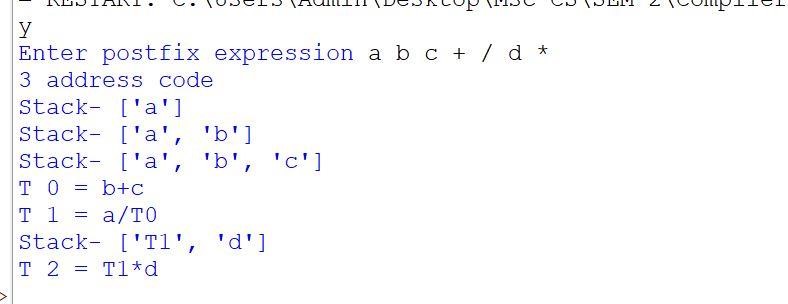
Aim: Write a code to generate 3 address code Code:

postfix=input("Enter postfix expression").split() operators=['+','-','/','\*','^'] stack=[] result='' str1=''

count=0 print("3 address code") for i in postfix: if i not in operators: stack.append(i) print("Stack-",stack) else:

op1=stack.pop() op2=stack.pop() result=op2+i+op1 str1='T'+str(count) stack.append(str1) print("T",count,"=",result) count+=1

Output:



# PRACTICAL NO 7

Aim: Write a program to demonstrate loop jamming for given code sequence containing loop. Code: Loop Jamming import time

from datetime import datetime def func1(arr1,arr2,arr3):

t1=datetime.now() start=time.time()

print(t1.minute,":",t1.second,":",t1.microsecond) for i in range (0,10000000):

sum=0 for j in range(0,len(arr1)):

sum=sum+arr1[j] for k in range(0,len(arr2)):

sum=sum+arr2[k] for l in range(0,len(arr3)):

sum=sum+arr3[l] if(sum!=210): print(false)

tm=datetime.now() done=time.time() elapsed=done-start

print(t1.minute,":",t1.second,":",t1.microsecond) print("First loop Difference",elapsed)

start=time.time() for i in range(0,10000000):

sum=0 for j in range(0,len(arr1)):

sum=sum+arr1[j] sum=sum+arr2[j] sum=sum+arr3[j] if(sum!=210): print(false)

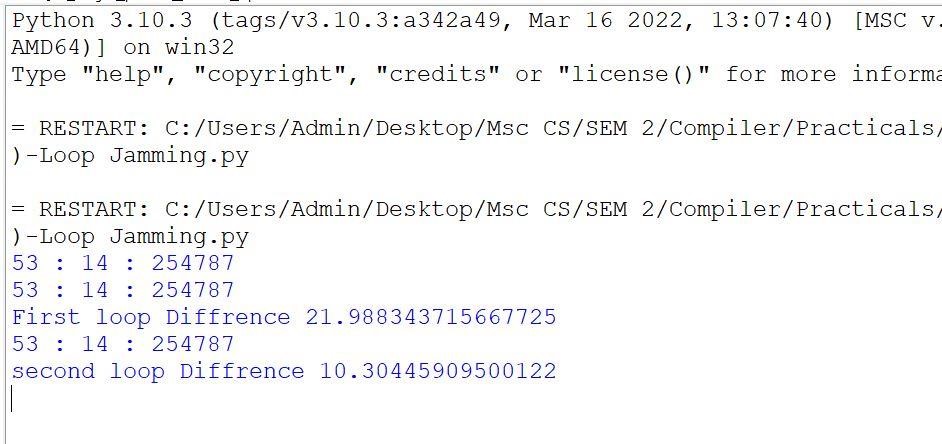
tn=datetime.now()

done=time.time() elapsed=done-start

print(t1.minute,":",t1.second,":",t1.microsecond) print("second loop Diffrence",elapsed)

arr1=[10,20,30] arr2=[20,10,30] arr3=[40,40,10] func1(arr1,arr2,arr3)

OUTPUT:



# PRACTICAL NO 8

Aim: Write a program to demonstrate loop unrolling for given code sequence containing loop.

Loop Unrolling Code:

import time

from datetime import datetime def funct1(): arr=[] arr1=[] t1=datetime.now() start=t1.microsecond print(start) for i in range(0,1000):

arr.insert(0,i) print(arr) t2=datetime.now() end1=t2.microsecond print(end1)

for i in range(0,1000,4):

arr1.insert(0,i) arr1.insert(0,i+1) arr1.insert(0,i+2) arr1.insert(0,i+3) print(arr1) t3=datetime.now() end2=t3.microsecond print(end2)

print("Before unroling:",end1-start) print("After unroling:",end2-end1) funct1()

OUTPUT:

