Name: Bhakti Varadkar **Academic Year**: 2021-2022

Roll No.: 36 **Class**: M. Sc. C.S. Part 1

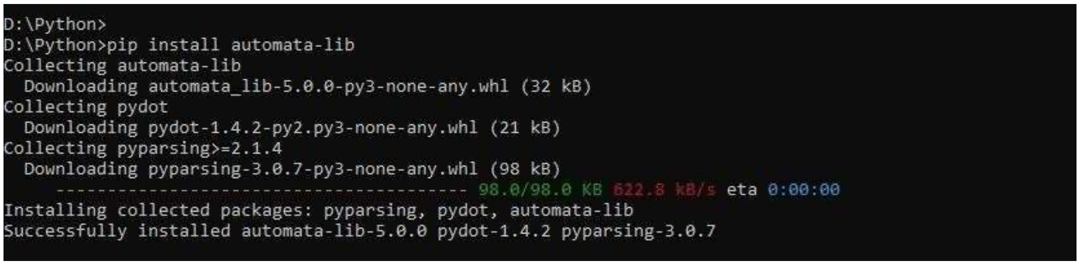
Subject: Design and Implementation of Modern Compilers

INDEX

Practical No. 1

Aim: Write a program to construct NDFA

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

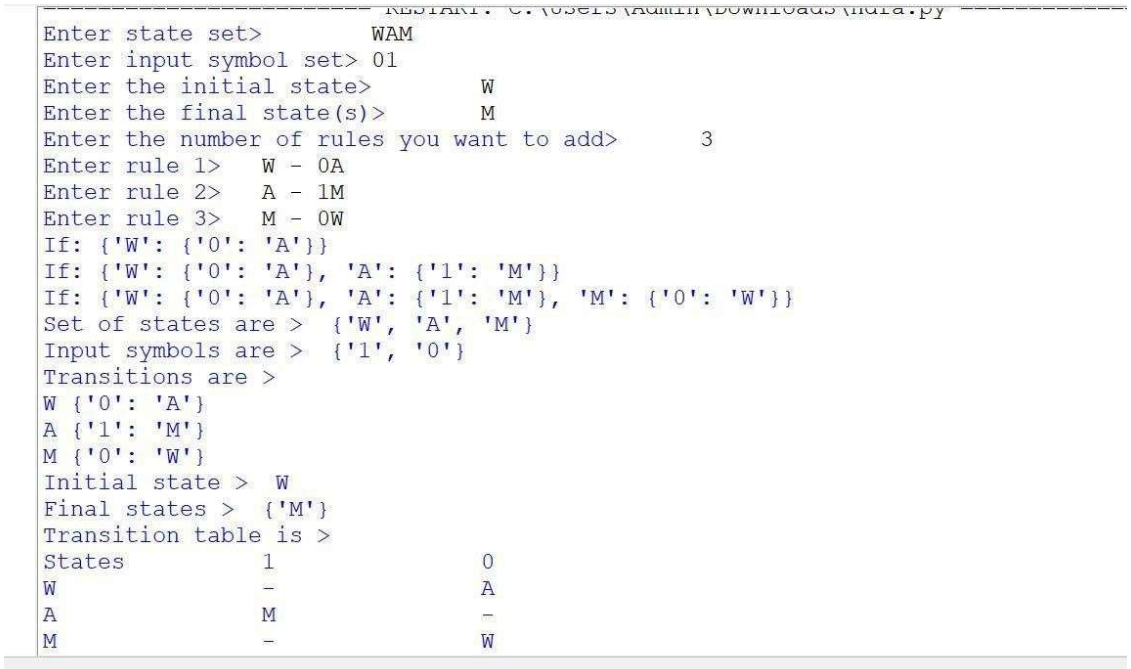


```
D:\Python>
D:\Python>pip install automata-lib
Collecting automata-lib
  Downloading automata_lib-5.0.0-py3-none-any.whl (32 kB)
Collecting pydot
  Downloading pydot-1.4.2-py2.py3-none-any.whl (21 kB)
Collecting pyparsing>=2.1.4
  Downloading pyparsing-3.0.7-py3-none-any.whl (98 kB)
     ---------------------------------------- 98.0/98.0 KB 622.8 kB/s eta 0:00:00
Installing collected packages: pyparsing, pydot, automata-lib
Successfully installed automata-lib-5.0.0 pydot-1.4.2 pyparsing-3.0.7
```

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][0]:rule[1][1]}
        else:
            rules_dict[rule[0]][rule[1][0]] = rule[1][1]
    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("States\t\t"+input_symbols[0]+"\t\t"+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(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:



```
======================== RESTART: C:\Users\Admin\Downloads\ndfa.py ============
Enter state set>        WAM
Enter input symbol set> 01
Enter the initial state>        W
Enter the final state(s)>       M
Enter the number of rules you want to add>      3
Enter rule 1>   W - 0A
Enter rule 2>   A - 1M
Enter rule 3>   M - 0W
If: {'W': {'0': 'A'}}
If: {'W': {'0': 'A'}, 'A': {'1': 'M'}}
If: {'W': {'0': 'A'}, 'A': {'1': 'M'}, 'M': {'0': 'W'}}
Set of states are >  {'W', 'A', 'M'}
Input symbols are >  {'1', '0'}
Transitions are >
W {'0': 'A'}
A {'1': 'M'}
M {'0': 'W'}
Initial state >  W
Final states >  {'M'}
Transition table is >
States          1               0
W               -               A
A               M               -
M               -               W
```

Practical No. 2

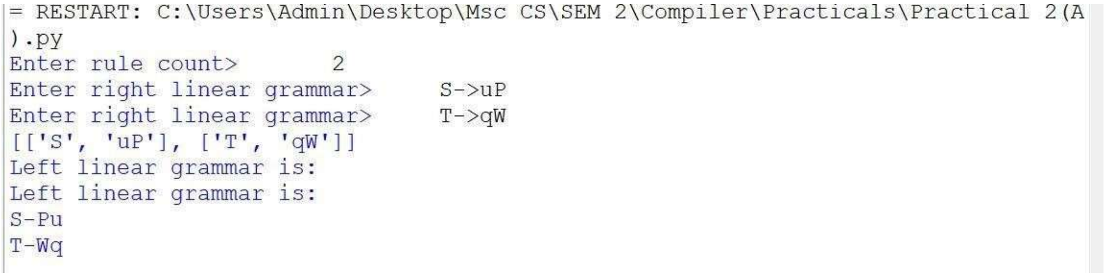
Aim: Write a program to convert the given Right linear grammar to Left Linear Grammar form.

Code:

```
def get_transitions(rules):
    my_dict = res = dict()
    ld = r = str()
    for i in rules:
        if i[0] not in my_dict:
            my_dict[i[0]] = []
        try:
            my_dict[i[0]].append([i[1][1], i[1][0]])
        except IndexError:
            continue
        print(my_dict)
    for sub in my_dict:
        for rule in my_dict[sub]:
            if isinstance(rule, list):
                if sub not in res:
                    res[sub] = []
                res[sub].append(ld.join([str(ele) for ele in rule]))
    print("Left Linear grammer is:")
    for item in res:
        for rhs in res[item]:
            if isinstance(rhs, str):
                print(r, item, "->", rhs)
if __name__ == "__main__":
    rule_count = int(input("Enter rule count>\t"))
    rules = []
```

```
for i in range(rule_count):
    rules.append(input("Enter right linear grammer" + str(i + 1) + ">\t"))
rules = [i.split("->") for i in rules]
get_transitions(rules)
```

Output:



```
= RESTART: C:\Users\Admin\Desktop\Msc CS\SEM 2\Compiler\Practicals\Practical 2(A
).py
Enter rule count>       2
Enter right linear grammar>     S->uP
Enter right linear grammar>     T->qW
[['S', 'uP'], ['T', 'qW']]
Left linear grammar is:
Left linear grammar is:
S-Pu
T-Wq
```

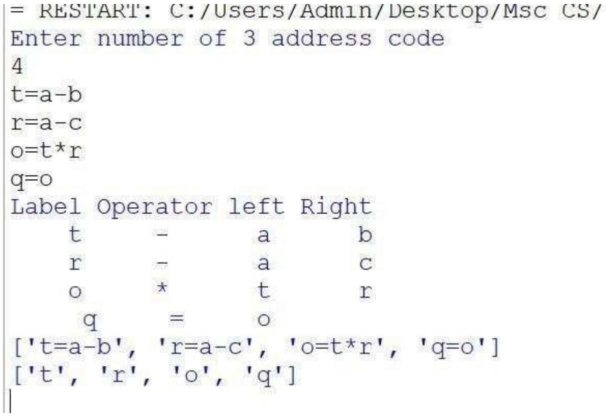
Practical No. 3

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

Code:

```
def func_1(x):
    main = []
    for i in range(0, x):
        main.append(input("Enter production " + str(i + 1) + " > ").replace(" ", ""))
    print("Label \t Operator \t Left \t Right")
    for i in range(x):
        q = main[i]
        if q[0] not in res:
            res.append(q[0])
        if(len(q) > 3):
            print(str(q[0]) + " \t " + str(q[3]) + " \t\t " + str(q[2]) + " \t " + str(q[4]))
        else:
            print(str(q[0]) + " \t " + str(q[1]) + " \t " + str(q[2]) + " \t " + "-")
    print(main)
    print(res)
x = int(input("Enter number of three address codes > "))
res = []
func_1(x)
```

Output:



```
= RESTART: C:/Users/Admin/Desktop/Msc CS/
Enter number of 3 address code
4
t=a-b
r=a-c
o=t*r
q=o
Label Operator left Right
    t     -      a      b
    r     -      a      c
    o     *      t      r
     q     =     o
['t=a-b', 'r=a-c', 'o=t*r', 'q=o']
['t', 'r', 'o', 'q']
```

Practical No. 4

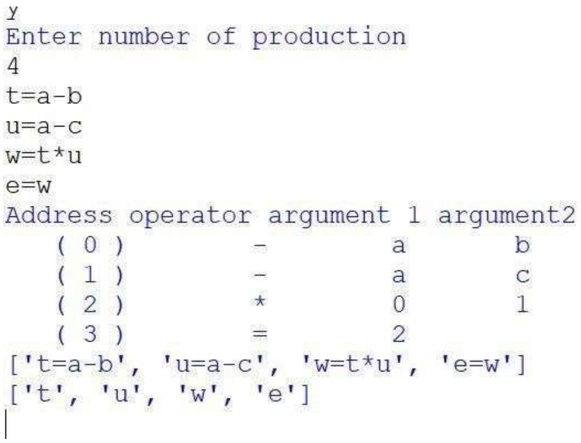
Aim: Write a code for triples.

Code:

```
def func_1(x):
    main = []
    for i in range(0, x):
        main.append(input("Enter production " + str(i + 1) + " > ").replace(" ", ""))
    print("Address \t Operator \t Argument1 \t Argument2")
    for i in range(x):
        q = main[i]
        if q[0] not in res:
            res.append(q[0])
        e = func_2(q[2])
        if(len(q) > 3):
            r = func_2(q[4])
            print(str(i) + " \t " + str(q[3]) + "\t\t " + str(e) + "\t\t " + str(r))
        else:
            print(str(i) + "\t\t " + str(q[1]) + "\t\t " + str(e) + "\t\t " + "-")
    print(main)
    print(res)


def func_2(q):
    try:
        return res.index(q)
    except:
        return q


x = int(input("Enter number of productions > "))
res = []
```

func_1(x)

Output:



```
y
Enter number of production
4
t=a-b
u=a-c
w=t*u
e=w
Address operator argument 1 argument2
   ( 0 )        -        a       b
   ( 1 )        -        a       c
   ( 2 )        *        0       1
   ( 3 )        =        2
['t=a-b', 'u=a-c', 'w=t*u', 'e=w']
['t', 'u', 'w', 'e']
```

Practical No. 5

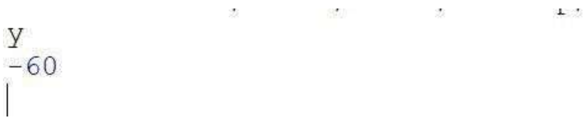
Aim: Write the code for Postfix Evaluation.

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

```
s="4 2 + 3 5 1 - * +"

val=postfix_evaluation(s)

print(val)
```

Output:



```
y
-60
```

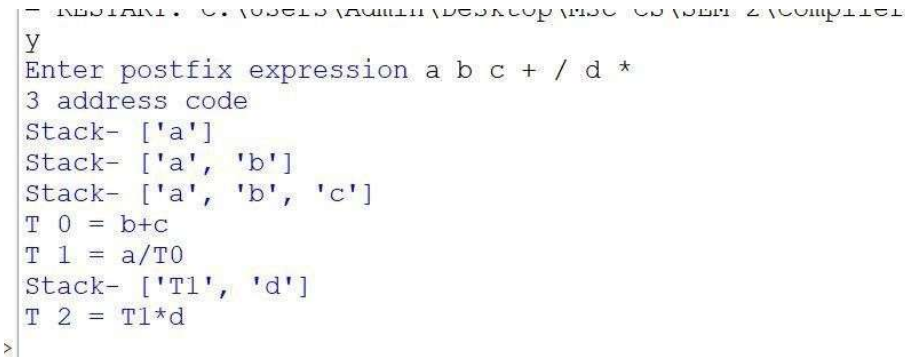
Practical No. 6

Aim: Write a code to generate 3 address code.

Code:

```
if __name__ == "__main__":
    postfix_expr = input("Enter postfix expression > ").split()
    operator_set = ('+', '-', '/', '*', '^')
    stack = []
    result = str1 = " "
    count = 0
    print("Three address code")
    for i in postfix_expr:
        if i not in operator_set:
            stack.append(i)
            print("Stack:", stack)
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            result = operand1 + i + operand2
            stack.append("T" + str(count))
            print("T", count, "=", result)
            count += 1
```

Output:



```
= RESTART: C:\Users\Admin\Desktop\MSC CS\SEM 2\Compiler
y
Enter postfix expression a b c + / d *
3 address code
Stack- ['a']
Stack- ['a', 'b']
Stack- ['a', 'b', 'c']
T 0 = b+c
T 1 = a/T0
Stack- ['T1', 'd']
T 2 = T1*d
>
```

Practical No. 7

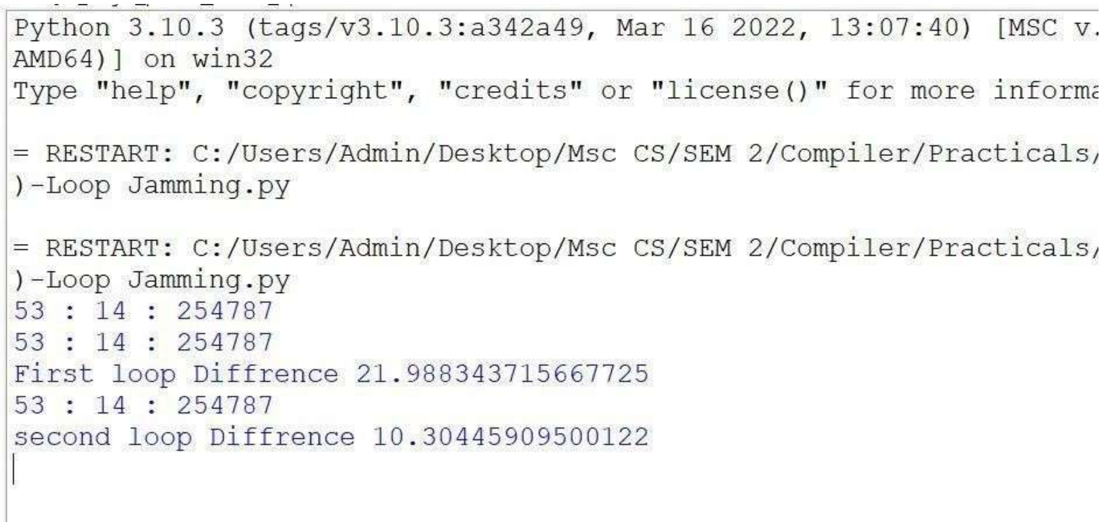
Aim: Write a program to demonstrate loop jamming for given code sequence containing loop.

Code:

```
import time
from datetime import datetime


def func1(arr1,arr2,arr3):
   t1 = datetime.now()
   print(t1.minute,":",t1.second,":",t1.microsecond)
   start=time.time()
   for i in range(0,100000):
      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()
   print(tm.minute,":",tm.second,":",tm.microsecond)
   end=time.time()
   diff=end - start
   print("time take by first loop",diff)
   start1=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()
    print(tn.minute,":",tn.second,":",tn.microsecond)
    end1=time.time()
    diff1=end1-start1
    print("time taken by second loop",diff1)


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

Output:



```
Python 3.10.3 (tags/v3.10.3:a342a49, Mar 16 2022, 13:07:40) [MSC v.
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more informa

= RESTART: C:/Users/Admin/Desktop/Msc CS/SEM 2/Compiler/Practicals/
)-Loop Jamming.py

= RESTART: C:/Users/Admin/Desktop/Msc CS/SEM 2/Compiler/Practicals/
)-Loop Jamming.py
53 : 14 : 254787
53 : 14 : 254787
First loop Diffrence 21.988343715667725
53 : 14 : 254787
second loop Diffrence 10.30445909500122
```

Practical No. 8

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

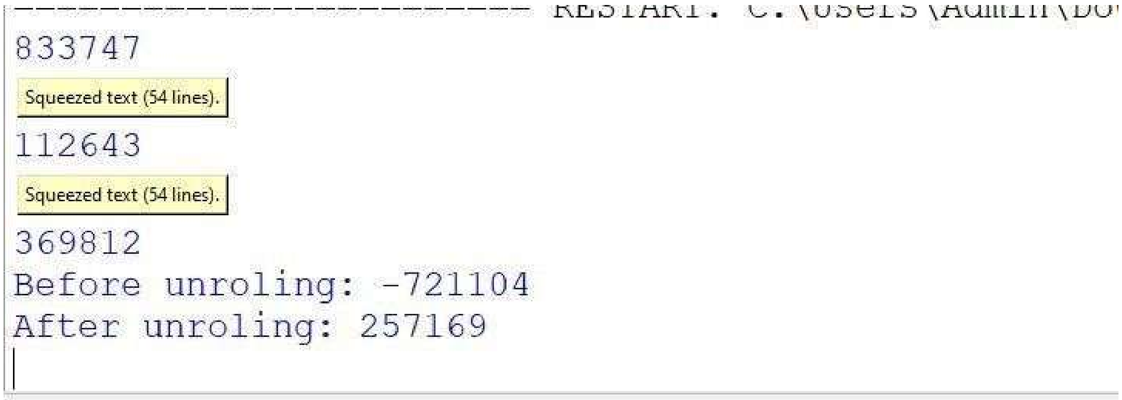
Code:

```
import time
from datetime import datetime
def func1():
    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 unrolling",end1-start)
    print("after unrolling",end2-end1)
```

func1()

#first loop should run more time then second but get the same output

Output:



```
======================== RESTART: C:\Users\Admin\Do
833747
Squeezed text (54 lines).
112643
Squeezed text (54 lines).
369812
Before unroling: -721104
After unroling: 257169
```