M.Sc C.S - I
SEM II
Journal

Roll No.	027
Name	OJHA ABHISHEK DEVMANI
Subject	DESIGN AND IMPLEMENTATION OF MODERN COMPILER

CERTIFICATE

This is here to certify that Mr. **OJHA ABHISHEK DEVMANI**, Seat Number **027** of M.Sc. I Computer Science, has satisfactorily completed the required number of experiments prescribed by the **THAKUR COLLEGE OF SCIENCE & COMMERCE AUTONOMOUS COLLEGE, PERMANENTLY AFFILIATED TO UNIVERSITY OF MUMBAI** during the academic year 2021 – 2022.

Date: 28-03-2022
Place: Mumbai

Teacher In-Charge

Head of Department

External Examiner

INDEX

Fig. No.	List of Figures	Page No

Tb. No.	List of Tables	Page No

Experiment no – 01

Aim: Write a program to accept a string and validate using NFA.

Theory: -

NFA (Non-deterministic Finite automata) finite state machine that can move to any combination of states for an input symbol i.e. there is no exact state to which the machine will move.

NFA / NDFA (Non-deterministic Finite automata) can be represented by 5-tuple (Q, ∑, δ, q0, F) where −

- Q is a finite set of states.
- ∑ is a finite set of symbols called the alphabets.
- δ is the transition function where d: Q × ∑ → 2Q (Here the power set of Q (2Q) has been taken because in case of NDFA, from a state, transition can occur to any combination of Q states)
- q0 is the initial state from where any input is processed (q0 ∈ Q).
- F is a set of final state/states of Q (F ⊆ Q).

In programming, NFA is created using a directed graph. Each vertex of the graph denotes the states of NDA. The edges of the graph can have one of the two values 0 or 1. Edge labeled as 0 represents non-accepting transition whereas Edge labeled as 1 represents accepting transition.

There is an entry point to the graph generally vertex 1 from where it takes input string which is a binary array of finite length.

Let's see an NFA graphical form and then solve a grammar using it.

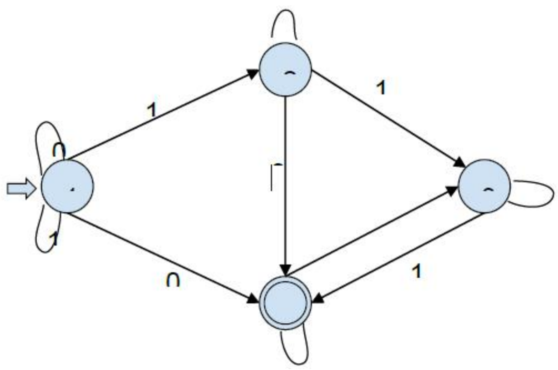


Fig: - NFA.

Starting state - vertex 1

Accepting states - Vertices with double circles(label 1) // Vertex 4

Non accepting states - single circles (label 0). // Vertices 1, 2 and 3.

For Input : 1010

In-state 1, we have two possibilities, either follow the self-loop and stay in state 1 or follow the edge labeled 1 and go to state 3.

{1} 1010 --> {1, 3} 010

In-state 3, there is no edge labeled 0, so the computation will die out.

In-state 1, we have two possibilities, either follow the self-loop and stay in state 1, or follow the edge labeled 0 to state 2.

{1, 3} 010 --> {1, 2} 10

Now there is no edge labeled 1 from state 2. The computation will die out. We have two possibilities: either follow the self-loop to state 1 or follow the edge labeled 1 to state 3.

{1, 2} 10 --> {1, 3} 0

In-state 3, there is no edge labeled 0. So the computation will die out. In-state 1, we have two possibilities: either follow the self-loop to state 1 or the edge labeled 0 to state 2.

{1, 3} 0 --> {1, 2}

Now the NFA has consumed the input. It can be either be in states 1 or 2, both of which are non - accepting. So the NFA has rejected the input 1010.

For Input: 1100

{1} 1100 --> {1, 3} 100 {1, 3} 100 --> {1, 3, 4} 00 {1, 3, 4}

00--> {1, 2, 4} 0 {1, 2, 4} 0--> {1, 2, 4}

Now the NFA has consumed the input. It can either be in states 1, 2, or 4. State 4 is an accepting state. So, the NFA accepts the string 1100.

We can easily verify that the given NFA accepts all binary strings with “00” and/or “11” as a substring.

Practical Implementation of Insertion Sort :-

Code:-

```
# Design to recognize strings NFA
nfa = 1
# This checks for invalid input.
flag = 0
# Function for the state Q2
def state1(c):
   global nfa,flag

   # State transitions
   # 'a' takes to Q4, and
   # 'b' and 'c' remain at Q2
   if (c == 'a'):
      nfa = 2
   elif (c == 'b' or c == 'c'):
      nfa = 1
   else:
      flag = 1

# Function for the state Q3
def state2(c):
   global nfa,flag

   # State transitions
   # 'a' takes to Q3, and
   # 'b' and 'c' remain at Q4
   if (c == 'a'):
      nfa = 3
   elif (c == 'b' or c == 'c'):
      nfa = 2
   else:
      flag = 1

# Function for the state Q4
def state3(c):
   global nfa,flag

   # State transitions
   # 'a' takes to Q2, and
   # 'b' and 'c' remain at Q3
   if (c == 'a'):
      nfa = 1
   elif (c == 'b' or c == 'c'):
      nfa = 3
   else:
      flag = 1
```

```
# Function for the state Q5
def state4(c):
    global nfa,flag

    # State transitions
    # 'b' takes to Q6, and
    # 'a' and 'c' remain at Q5
    if (c == 'b'):
        nfa = 5
    elif (c == 'a' or c == 'c'):
        nfa = 4
    else:
        flag = 1

# Function for the state Q6
def state5(c):
    global nfa, flag

    # State transitions
    # 'b' takes to Q7, and
    # 'a' and 'c' remain at Q7
    if (c == 'b'):
        nfa = 6
    elif (c == 'a' or c == 'c'):
        nfa = 5
    else:
        flag = 1

# Function for the state Q7
def state6(c):
    global nfa,flag

    # State transitions
    # 'b' takes to Q5, and
    # 'a' and 'c' remain at Q7
    if (c == 'b'):
        nfa = 4
    elif (c == 'a' or c == 'c'):
        nfa = 6
    else:
        flag = 1

# Function for the state Q8
def state7(c):
    global nfa,flag

    # State transitions
```

```
    # 'c' takes to Q9, and
    # 'a' and 'b' remain at Q8
    if (c == 'c'):
        nfa = 8
    elif (c == 'b' or c == 'a'):
        nfa = 7
    else:
        flag = 1

# Function for the state Q9
def state8(c):
    global nfa,flag

    # State transitions
    # 'c' takes to Q10, and
    # 'a' and 'b' remain at Q9
    if (c == 'c'):
        nfa = 9
    elif (c == 'b' or c == 'a'):
        nfa = 8
    else:
        flag = 1

# Function for the state Q10
def state9(c):
    global nfa,flag

    # State transitions
    # 'c' takes to Q8, and
    # 'a' and 'b' remain at Q10
    if (c == 'c'):
        nfa = 7
    elif (c == 'b' or c == 'a'):
        nfa = 9
    else:
        flag = 1

# Function to check for 3 a's
def checkA(s, x):
    global nfa,flag
    for i in range(x):
        if (nfa == 1):
            state1(s[i])
        elif (nfa == 2):
            state2(s[i])
        elif (nfa == 3):
            state3(s[i])
```

```
    if (nfa == 1):
        return True

    else:
        nfa = 4

# Function to check for 3 b's
def checkB(s, x):
    global nfa,flag
    for i in range(x):
        if (nfa == 4):
            state4(s[i])
        elif (nfa == 5):
            state5(s[i])
        elif (nfa == 6):
            state6(s[i])

    if (nfa == 4):
        return True
    else:
        nfa = 7

# Function to check for 3 c's
def checkC(s, x):
    global nfa, flag
    for i in range(x):
        if (nfa == 7):
            state7(s[i])
        elif (nfa == 8):
            state8(s[i])
        elif (nfa == 9):
            state9(s[i])

    if (nfa == 7):
        return True

# Driver Code

s = "bbbca"
x = 5

# If any of the states is True, that is, if either
# the number of a's or number of b's or number of c's
# is a multiple of three, then the is accepted
if (checkA(s, x) or checkB(s, x) or checkC(s, x)):
    print("ACCEPTED")

else:
```

```
if (flag == 0):
    print("NOT ACCEPTED")

else:
    print("INPUT OUT OF DICTIONARY.")
```

Output:

ACCEPTED

Conclusion: The entered string were identified as NFA or not based of the provided state diagram .

Experiment no – 02

Aim: Write a program to minimize given DFA.

Theory Explanation:

- DFA refers to deterministic finite automata. Deterministic refers to the uniqueness of the computation. The finite automata are called deterministic finite automata if the machine is read an input string one symbol at a time.
- In DFA, there is only one path for specific input from the current state to the next state.
- DFA does not accept the null move, i.e., the DFA cannot change state without any input character.
- DFA can contain multiple final states. It is used in Lexical Analysis in Compiler.

In the following diagram, we can see that from state q0 for input a, there is only one path which is going to q1. Similarly, from q0, there is only one path for input b going to q2.

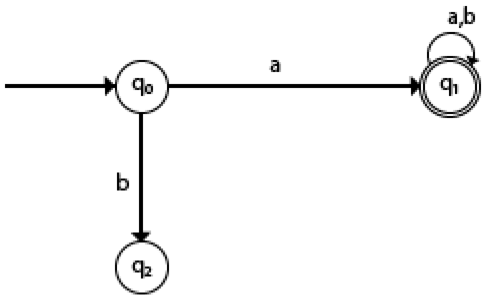


Fig 2.1: - DFA.

Formal Definition of DFA

A DFA is a collection of 5-tuples same as we described in the definition of FA.

1. Q: finite set of states
2. ∑: finite set of the input symbol
3. q0: initial state
4. F: **final** state
5. δ: Transition function

Transition function can be defined as:

1. $\delta: Q \times \sum \rightarrow Q$

Graphical Representation of DFA

A DFA can be represented by digraphs called state diagram. In which:

1. The state is represented by vertices.
2. The arc labeled with an input character show the transitions.
3. The initial state is marked with an arrow.
4. The final state is denoted by a double circle.

Example 1:

1. Q = {q0, q1, q2}
2. ∑ = {0, 1}
3. q0 = {q0}
4. F = {q2}

Solution:

Transition Diagram:

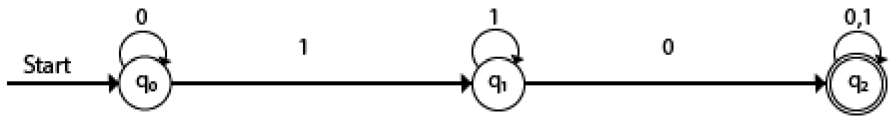


Fig 2.2: - Transition Diagram for DFA.

Transition Table:

PRESENT STATE	NEXT STATE FOR INPUT 0	NEXT STATE OF INPUT 1
→Q0	q0	q1
Q1	q2	q1
***Q2**	q2	q2

Table 2.1: - Transition Table for DFA.

Minimization of DFA

Minimization of DFA means reducing the number of states from given FA. Thus, we get the FSM(finite state machine) with redundant states after minimizing the FSM.

We have to follow the various steps to minimize the DFA. These are as follows:

Step 1: Remove all the states that are unreachable from the initial state via any set of the transition of DFA.

Step 2: Draw the transition table for all pair of states.

Step 3: Now split the transition table into two tables T1 and T2. T1 contains all final states, and T2 contains non-final states.

Step 4: Find similar rows from T1 such that:

1. 1. δ (q, a) = p
2. 2. δ (r, a) = p

That means, find the two states which have the same value of a and b and remove one of them.

Step 5: Repeat step 3 until we find no similar rows available in the transition table T1.

Step 6: Repeat step 3 and step 4 for table T2 also.

Step 7: Now combine the reduced T1 and T2 tables. The combined transition table is the transition table of minimized DFA.

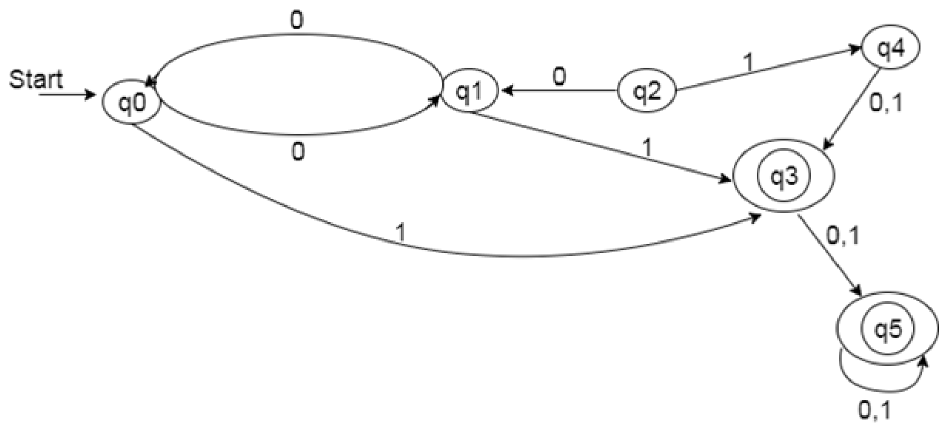


Fig 2.3: - Example for DFA.

Solution:

Step 1: In the given DFA, q2 and q4 are the unreachable states so remove them.

Step 2: Draw the transition table for the rest of the states.

STATE	0	1
→Q0	q1	q3
Q1	q0	q3
***Q3**	q5	q5
***Q5**	q5	q5

Table 2.2: - Transition Table for DFA.

Step 3: Now divide rows of transition table into two sets as:

1. One set contains those rows, which start from non-final states:

STATE	0	1
Q0	q1	q3
Q1	q0	q3

2. Another set contains those rows, which starts from final states.

STATE	0	1
Q3	q5	q5
Q5	q5	q5

Step 4: Set 1 has no similar rows so set 1 will be the same.

Step 5: In set 2, row 1 and row 2 are similar since q3 and q5 transit to the same state on 0 and 1. So skip q5 and then replace q5 by q3 in the rest.

STATE	0	1
Q3	q3	q3

Step 6: Now combine set 1 and set 2 as:

STATE	0	1
→Q0	q1	q3
Q1	q0	q3

*Q3	q3	q3

Now it is the transition table of minimized DFA.

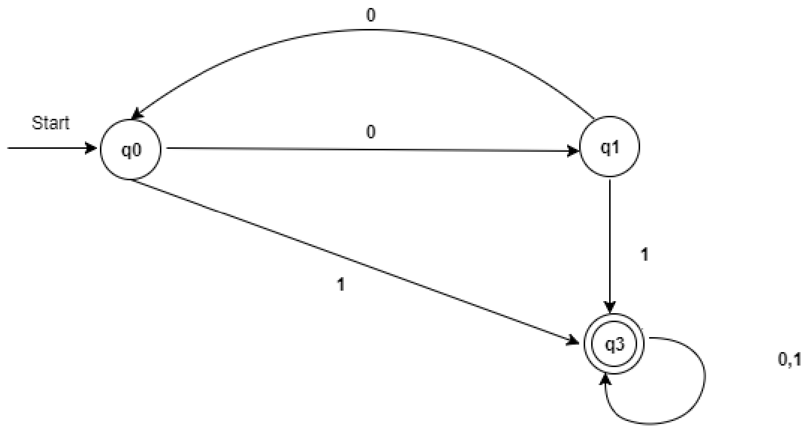


Fig 2.4: - Transition Table of Minimized DFA.

Program:

1. **DFA.Py**

```
from collections import defaultdict
from disjoint_set import DisjointSet


class DFA(object):

   def __init__(self,states_or_filename,terminals=None,start_state=None,
transitions=None,final_states=None):

      if terminals is None:
         self._get_graph_from_file(states_or_filename)
      else:
         assert isinstance(states_or_filename,list) or isinstance(states_or_filename,tuple)
         self.states = states_or_filename

         assert isinstance(terminals,list) or isinstance(terminals,tuple)
         self.terminals = terminals

         assert isinstance(start_state,str)
         self.start_state = start_state

         assert isinstance(transitions,dict)
         self.transitions = transitions

         assert isinstance(final_states,list) or isinstance(final_states,tuple)
         self.final_states = final_states
```

```
def _remove_unreachable_states(self):
    '''
    Removes states that are unreachable from the start state
    '''

    g = defaultdict(list)

    for k,v in self.transitions.items():
        g[k[0]].append(v)

    # do DFS
    stack = [self.start_state]

    reachable_states =  set()

    while stack:
        state = stack.pop()

        if state not in reachable_states:
            stack += g[state]

        reachable_states.add(state)

    self.states = [state for state in self.states \
                    if state in reachable_states]

    self.final_states = [state for state in self.final_states \
                    if state in reachable_states]


    self.transitions = { k:v for k,v in self.transitions.items() \
                   if k[0] in reachable_states}


def minimize(self):

    self._remove_unreachable_states()

    def order_tuple(a,b):
        return (a,b) if a < b else (b,a)

    table = {}

    sorted_states = sorted(self.states)

    # initialize the table
    for i,item in enumerate(sorted_states):
        for item_2 in sorted_states[i+1:]:
            table[(item,item_2)] = (item in self.final_states) != (item_2\
```

```
                        in self.final_states)

    flag = True

    # table filling method
    while flag:
        flag = False

        for i,item in enumerate(sorted_states):
            for item_2 in sorted_states[i+1:]:

                if table[(item,item_2)]:
                    continue

                # check if the states are distinguishable
                for w in self.terminals:
                    t1 = self.transitions.get((item,w),None)
                    t2 = self.transitions.get((item_2,w),None)

                    if t1 is not None and t2 is not None and t1 != t2:
                        marked = table[order_tuple(t1,t2)]
                        flag = flag or marked
                        table[(item,item_2)] = marked

                        if marked:
                            break

    d = DisjointSet(self.states)

    # form new states
    for k,v in table.items():
        if not v:
            d.union(k[0],k[1])

    self.states = [str(x) for x in range(1,1+len(d.get()))]
    new_final_states = []
    self.start_state = str(d.find_set(self.start_state))

    for s in d.get():
        for item in s:
            if item in self.final_states:
                new_final_states.append(str(d.find_set(item)))
                break

    self.transitions = {(str(d.find_set(k[0])),k[1]):str(d.find_set(v))
                        for k,v in self.transitions.items()}

    self.final_states = new_final_states

def __str__(self):
```

```
        '''
        String representation
        '''
        num_of_state = len(self.states)
        start_state = self.start_state
        num_of_final = len(self.final_states)

        return '{} states. {} final states. start state - {}'.format( \
                        num_of_state,num_of_final,start_state)


    def _get_graph_from_file(self,filename):
        '''
        Load the graph from file
        '''

        with open(filename,'r') as f:

            try:
                lines = f.readlines()
                states,terminals,start_state,final_states = lines[:4]

                if states:
                    self.states = states[:-1].split()
                else:
                    raise Exception('Invalid file format: cannot read states')

                if terminals:
                    self.terminals = terminals[:-1].split()
                else:
                    raise Exception('Invalid file format: cannot read terminals')

                if start_state:
                    self.start_state = start_state[:-1]
                else:
                    raise Exception('Invalid file format: cannot read start state')

                if final_states:
                    self.final_states = final_states[:-1].split()
                else:
                    raise Exception('Invalid file format: cannot read final states')

                lines = lines[4:]

                self.transitions = {}

                for line in lines:
                    current_state,terminal,next_state = line[:-1].split()

                    self.transitions[(current_state,terminal)] = next_state
```

```
    except Exception as e:
      print("ERROR: ",e)

if __name__ =="__main__":
    filename = 'graph'
    dfa = DFA(filename)
    print(dfa)
    dfa.minimize()
    print(dfa)
```

2. **Disjoint.py**

```
class DisjointSet(object):

  def __init__(self,items):

    self._disjoint_set = list()

    if items:
      for item in set(items):
        self._disjoint_set.append([item])

  def _get_index(self,item):
    for s in self._disjoint_set:
      for _item in s:
        if _item == item:
          return self._disjoint_set.index(s)
    return None

  def find(self,item):
    for s in self._disjoint_set:
      if item in s:
        return s
    return None

  def find_set(self,item):

    s = self._get_index(item)

    return s+1 if s is not None else None

  def union(self,item1,item2):
    i = self._get_index(item1)
    j = self._get_index(item2)
```

```
    if i != j:
      self._disjoint_set[i] += self._disjoint_set[j]
      del self._disjoint_set[j]

  def get(self):
    return self._disjoint_set
```

3. Graph Input

```
1 2 3 4 5
a b
1
1 5
1 a 3
1 b 2
2 b 1
2 a 4
3 b 4
3 a 5
4 a 4
4 b 4
5 a 3
5 b 2
```

OUTPUT:

```
('1', '3') : ['a']
('1', '2') : ['b']
('2', '1') : ['b']
('2', '4') : ['a']
('3', '4') : ['b']
('3', '5') : ['a']
('4', '4') : ['a', 'b']
('5', '3') : ['a']
('5', '2') : ['b']
5 states. 2 final states. start state - 1

('2', '4') : ['a']
('2', '1') : ['b']
('1', '2') : ['b']
('1', '3') : ['a']
('4', '3') : ['b']
('4', '2') : ['a']
('3', '3') : ['a', 'b']
4 states. 1 final states. start state - 2
```

Conclusion : A minimized DFA was successfully observed from 5 total states to 4 states.

References :-

https://github.com/navin-mohan/dfa-minimization

https://www.javatpoint.com/minimization-of-dfa

Experiment no - 03

Aim: Write a program to construct DFA using given regular expression.

Description:

Given a string S, the task is to design a Deterministic Finite Automata (DFA) for accepting the language L = C (A + B)+. If the given string is accepted by DFA, then print "Yes". Otherwise, print "No".

Examples:

Input: S = "CABABABAB"
Output: Yes
Explanation: The given string is of the form C(A + B)+ as the first character is C and it is followed by A or B.

Input: S = "ACCBBCCA"
Output: No

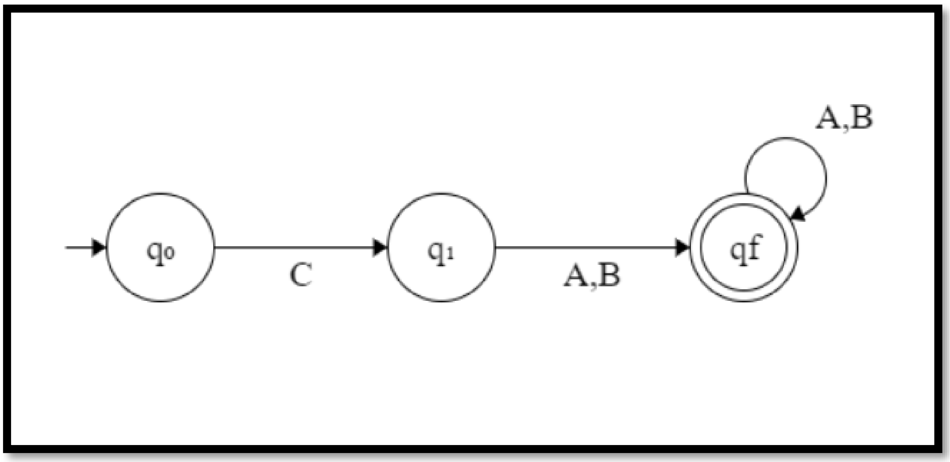


Fig 3.1: - DFA using REGEX.

- If the given string is of length less than equal to 1, then print "No".
- If the first character is always C, then traverse the remaining string and check if any of the characters is A or B.
- If there exists any character other than A or B while traversing in the above step, then print "No".
- Otherwise, print "Yes".
- Below is the implementation of the above approach:

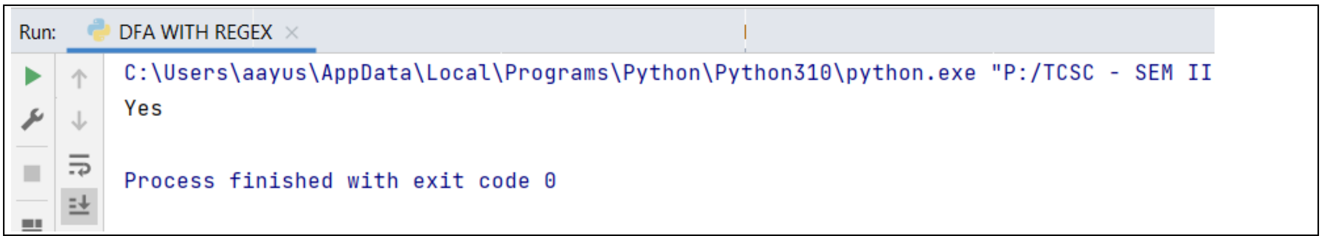
Program:

```python
#Program to Construct DFA using REGEX
def DFA(str, N):
    # If n <= 1, then prNo
    if (N <= 1):
        print("No")
        return
    # To count the matched characters
    count = 0
    # Check if the first character is C
    if (str[0] == 'C'):
        count += 1
        # Traverse the rest of string
        for i in range(1, N):

            # If character is A or B,
            # increment count by 1
            if (str[i] == 'A' or str[i] == 'B'):
                count += 1
            else:
                break
    else:
        # If the first character
        # is not C, pr-1
        print("No")
        return
    # If all characters matches
    if (count == N):
        print("Yes")
    else:
        print("No")
# Driver Code
if __name__ == '__main__':
    str = "CAABBAAB"
    N = len(str)
    DFA(str, N)
```

OUTPUT:

Given String as : “CAABBAAB”



Given String as : "ACCBBCCA"

```
if __name__ == '__main__':
    str = "ACCBBCCA"
    N = len(str)
    DFA(str, N)
```

```
if __name__ == '__main__'
Run:  DFA WITH REGEX
C:\Users\aayus\AppData\Local\Programs\Python\Python310\python.exe "P:/TCSC - SEM II/
No

Process finished with exit code 0
```

Conclusion : The given string is accepted by DFA as "CAABBAAB"

Experiment no – 04

Aim: Write a program to construct NFA using given regular expression.

Algorithm:

1. Create a menu for getting four regular expressions input as choice.
2. To draw NFA for a, a/b ,ab ,a* create a routine for each regular expression.
3. For converting from regular expression to NFA, certain transition had been made based on choice of input at the rumtime.
4. Each of the NFA will be displayed is sequential order.

Theory Explanation :

∈-NFA is similar to the NFA but have minor difference by epsilon move. This automaton replaces the transition function with the one that allows the empty string ∈ as a possible input. The transitions without consuming an input symbol are called ∈-transitions.

In the state diagrams, they are usually labeled with the Greek letter ∈. ∈-transitions provide a convenient way of modeling the systems whose current states are not precisely known: i.e., if we are modeling a system and it is not clear whether the current state (after processing some input string) should be q or q', then we can add an ∈-transition between these two states, thus putting the automaton in both states simultaneously.

Common regular expression used in make ∈-NFA:

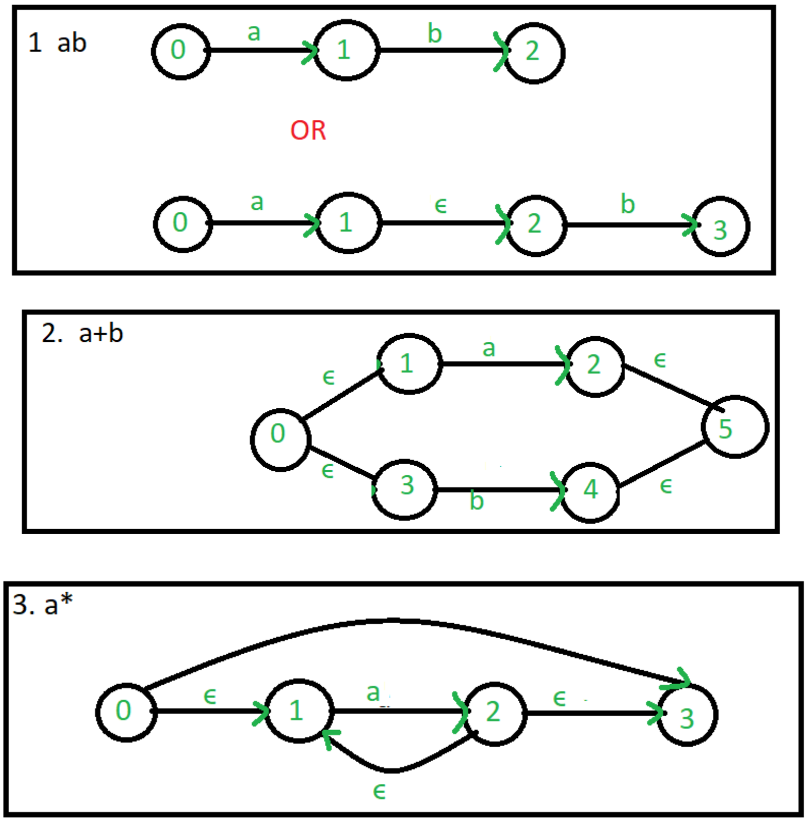
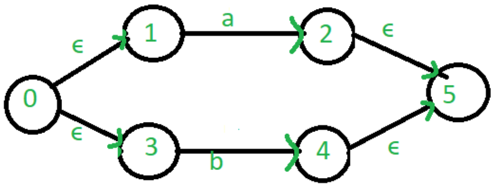


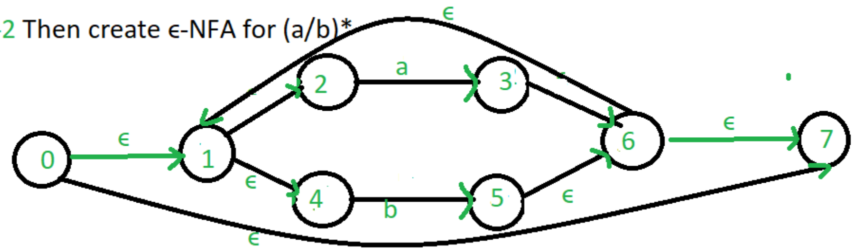
Fig 4.1: - NFA using REGEX.

Example: Create a ∈-NFA for regular expression: (a/b)*a

Step-1 First we create ϵ-NFA for (a/b)



Step-2 Then create ϵ-NFA for (a/b)*



Step-3 Then we create ϵ-NFA for(a/b)*a

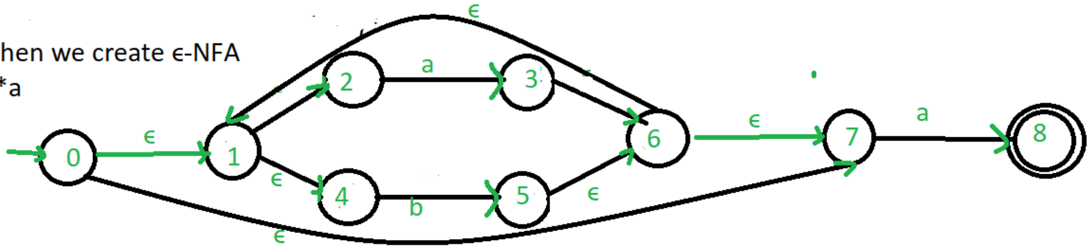


Fig 4.2: - Example of NFA using REGEX.

Program:

4. **Main.Py**

```
#Program to Construct NFA using REGEX


import sys
import nfa_utils
import time

# print the intro text block
with open("intro.dat") as intro_file:
    print(intro_file.read())

# regular expression string to compare against provided input
regex = None
regex_nfa = None
# last line of user input read from the command line
line_read = ""

# continuously parse and process user input
while True:
    # read in line of user input
    line_read = input("> ")
    # make a lowercase copy of the input for case insensitive comparisons
    line_read_lower = line_read.lower()

    if line_read_lower == "exit":
        # exit the program
        print("\nExiting...")
        sys.exit()

    if line_read_lower.startswith("regex="):
        # user wants to set the regex to a string they've provided
        regex = line_read[6:]
        print("New regex pattern:", regex, "\n")
        start_time = time.time()
        # turn regular expression string into an NFA object
        regex_nfa = nfa_utils.get_regex_nfa(regex)
        regex_nfa.reset()
        finish_time = time.time()
        ms_taken = (finish_time - start_time) * 1000

        print("\nBuilt NFA in {:.3f} ms.\n".format(ms_taken))
        print(regex_nfa)
    else:
        # assume the user intends to test this entered string against the regex
        if regex_nfa is None:
            # regex has not yet been set
```

```
        print("Please supply a regular expression string first, with regex=(regex here)")
    else:
        start_time = time.time()
        # feed input string into NFA
        regex_nfa.feed_symbols(line_read, return_if_dies=True)
        accepts = regex_nfa.is_accepting()
        finish_time = time.time()
        ms_taken = (finish_time - start_time) * 1000

        print("String was {} by NFA"
              .format("ACCEPTED" if accepts else "REJECTED"))

        print("Calculated in {:.3f} ms.".format(ms_taken))

        # print(regex_nfa)
        regex_nfa.reset()

# print a new line for aesthetics
print()
```

5. **NFA.py**

```
class NFA:
    """Class representing a non-deterministic finite automaton"""

    def __init__(self):
        """Creates a blank NFA"""

        # all NFAs have a single initial state by default
        self.alphabet = set()
        self.states = {0}
        self.transition_function = {}
        self.accept_states = set()

        # set of states that the NFA is currently in
        self.in_states = {0}

    def add_state(self, state, accepts=False):
        self.states.add(state)

        if accepts:
            self.accept_states.add(state)

    def add_transition(self, from_state, symbol, to_states):
        self.transition_function[(from_state, symbol)] = to_states

        if symbol != "":
```

```
            self.alphabet.add(symbol)

    def feed_symbol(self, symbol):
        """
        Feeds a symbol into the NFA, calculating which states the
        NFA is now in, based on which states it used to be in
        """

        # a dead NFA will not have any transitions after a symbol is fed in
        if self.is_dead():
            return

        new_states = set()

        # process each old state in turn
        for state in self.in_states:
            pair = (state, symbol)

            # check for a legal transition from the old state to a
            # new state, based on what symbol was fed in
            if pair in self.transition_function:
                # add the corresponding new state to the updated states list
                new_states |= self.transition_function[pair]

        self.in_states = new_states

        # feed the empty string through the nfa
        self.feed_empty()

    def feed_symbols(self, symbols, return_if_dies=False):
        """
        Feeds an iterable into the NFAs feed_symbol method

        :param symbols: Iterable of symbols to feed through the NFA
        :param return_if_dies: If true, ignore any further symbols after the NFA dies (for
efficiency),
        since a dead NFA will never accept, regardless of any further input.
        """

        for symbol in symbols:
            self.feed_symbol(symbol)

            if return_if_dies and self.is_dead():
                # NFA is dead; feeding further symbols will not change the NFA's state
                return

    def feed_empty(self):
        """
        Continuously feeds empty strings into the NFA until they fail
        to cause any further state transitions
```

```
        """

        # a dead NFA will not have any empty string transitions
        if self.is_dead():
            return

        old_states_len = None
        # set of states that will be fed the empty string on the next pass
        unproc_states = self.in_states
        first_run = True

        # keep feeding the empty string until no more new states are transitioned into
        while first_run or len(self.in_states) > old_states_len:
            old_states_len = len(self.in_states)
            # set of new states transitioned into after the empty string was fed
            new_states = set()

            # process each state in turn
            for state in unproc_states:
                pair = (state, "")

                # check if this state has a transition using the empty string
                # to another state
                if pair in self.transition_function:
                    # add the new state to a set to be added to self.in_states later
                    new_states |= self.transition_function[pair]

            # merge new states back into "in" states
            self.in_states |= new_states
            # all new states discovered will be fed the empty string on the next pass
            unproc_states = new_states
            first_run = False

    def is_accepting(self):
        # accepts if we are in ANY accept states
        # ie. if in_states and accept_states share any states in common
        return len(self.in_states & self.accept_states) > 0

    def is_dead(self):
        """
        Returns true if the NFA is not in ANY states.
        A "dead" NFA can never be in any states again.
        """
        return len(self.in_states) == 0

    def reset(self):
        """
        Resets the NFA by putting it back to it's initial state,
        and feeding the empty string through it
        """
```

```
        self.in_states = {0}
        self.feed_empty()

    def __str__(self):
        """
        String representation of this NFA.
        Useful for debugging.
        """
        return "NFA:\n" \
               "Alphabet: {}\n" \
               "States: {}\n" \
               "Transition Function: {}\n" \
               "Accept States: {}\n" \
               "In states: {}\n" \
               "Accepting: {}\n"\
            .format(self.alphabet,
                    self.states,
                    self.transition_function,
                    self.accept_states,
                    self.in_states,
                    "Yes" if self.is_accepting() else "No")

    def __eq__(self, other):
        """
        Checks if two NFAs are equal. Used for testing.

        Tests if they are structurally the same; does NOT check if they are in the same states.

        Also ignores alphabets.
        """
        return self.states == other.states \
          and self.transition_function == other.transition_function \
          and self.accept_states == other.accept_states
```

6. **NFA.UTILS.py**

```
from nfa import NFA
import copy


def get_single_symbol_regex(symbol):
    """ Returns an NFA that recognizes a single symbol """

    nfa = NFA()
    nfa.add_state(1, True)
    nfa.add_transition(0, symbol, {1})

    return nfa
```

```python
def shift(nfa, inc):
    """
    Increases the value of all states (including accept states and transition function etc)
    of a given NFA bya given value.

    This is useful for merging NFAs, to prevent overlapping states
    """
    # update NFA states
    new_states = set()
    for state in nfa.states:
        new_states.add(state + inc)
    nfa.states = new_states

    # update NFA accept states
    new_accept_states = set()
    for state in nfa.accept_states:
        new_accept_states.add(state + inc)
    nfa.accept_states = new_accept_states

    # update NFA transition function
    new_transition_function = {}
    for pair in nfa.transition_function:
        to_set = nfa.transition_function[pair]
        new_to_set = set()

        for state in to_set:
            new_to_set.add(state + inc)

        new_key = (pair[0] + inc, pair[1])
        new_transition_function[new_key] = new_to_set

    nfa.transition_function = new_transition_function


def merge(a, b):
    """Merges two NFAs into one by combining their states and transition function"""
    a.accept_states = b.accept_states
    a.states |= b.states
    a.transition_function.update(b.transition_function)
    a.alphabet |= b.alphabet


def get_concat(a, b):
    """ Concatenates two NFAs, ie. the dot operator """

    # number to add to each b state number
    # this is to ensure each NFA has separate number ranges for their states
    # one state overlaps; this is the state that connects a and b
```

```
    add = max(a.states)

    # shift b's state/accept states/transition function, etc.
    shift(b, add)

    # merge b into a
    merge(a, b)

    return a


def get_union(a, b):
    """Returns the resulting union of two NFAs (the '|' operator)"""

    # create a base NFA for the union
    nfa = NFA()

    # clear a and b's accept states
    a.accept_states = set()
    b.accept_states = set()

    # merge a into the overall NFA
    shift(a, 1)
    merge(nfa, a)

    # merge b into the overall NFA
    shift(b, max(nfa.states) + 1)
    merge(nfa, b)

    # add an empty string transition from the initial state to the start of a and b
    # (so that the NFA starts in the start of a and b at the same time)
    nfa.add_transition(0, "", {1, min(b.states)})

    # add an accept state at the end so if either a or b runs through,
    # this NFA accepts
    new_accept = max(nfa.states) + 1
    nfa.add_state(new_accept, True)
    nfa.add_transition(max(a.states), "", {new_accept})
    nfa.add_transition(max(b.states), "", {new_accept})

    return nfa


def get_kleene_star_nfa(nfa):
    """
    Wraps an NFA inside a kleene star expression
    (NFA passed in recognizes 0, 1 or many of the strings it originally recognized)
    """
    # clear old accept state
    nfa.accept_states = {}
```

```
    # shift NFA by 1 and insert new initial state
    shift(nfa, 1)
    nfa.add_state(0)

    # add new ending accept state
    last_state = max(nfa.states)
    new_accept = last_state + 1
    nfa.add_state(new_accept, True)
    nfa.add_transition(last_state, "", {new_accept})

    # add remaining empty string transitions
    nfa.add_transition(0, "", {1, new_accept})
    nfa.add_transition(last_state, "", {0})

    return nfa

def get_one_or_more_of_nfa(nfa):
    """
    Wraps an NFA inside the "one or more of" operator (plus symbol)

    Simply combines the concatenation operator and the kleene star operator.
    """

    # must make a copy of the nfa,
    # these functions operate on the nfa passed in, they do not make a copy
    return get_concat(copy.deepcopy(nfa), get_kleene_star_nfa(nfa))

def get_zero_or_one_of_nfa(nfa):
    """
    Wraps an NFA inside the "zero or one of" operator (question mark symbol)

    Simply uses the union operator, with one path for the empty string, and the other path
    for the NFA being wrapped.
    """

    return get_union(get_single_symbol_regex(""), nfa)

def get_regex_nfa(regex, indent=""):
    """Recursively builds an NFA based on the given regex string"""

    print("{0}Building NFA for regex:\n{0}({1})".format(indent, regex))
    indent += " " * 4

    # special symbols: +*.| (in order of precedence highest to lowest, symbols coming before that

    # union operator
    bar_pos = regex.find("|")
    if bar_pos != -1:
        # there is a bar in the string; union both sides
```

```
    # (uses the leftmost bar if there are more than 1)
    return get_union(
        get_regex_nfa(regex[:bar_pos], indent),
        get_regex_nfa(regex[bar_pos + 1:], indent)
    )

# concatenation operator
dot_pos = regex.find(".")
if dot_pos != -1:
    # there is a dot in the string; concatenate both sides
    # (uses the leftmost dot if there are more than 1)
    return get_concat(
        get_regex_nfa(regex[:dot_pos], indent),
        get_regex_nfa(regex[dot_pos + 1:], indent)
    )

# kleene star operator
star_pos = regex.find("*")
if star_pos != -1:
    # there is an asterisk in the string; wrap everything before it in a kleene star expression
    # (uses the leftmost dot if there are more than 1)
    star_part = regex[:star_pos]
    trailing_part = regex[star_pos + 1:]
    kleene_nfa = get_kleene_star_nfa(get_regex_nfa(star_part, indent))

    if len(trailing_part) > 0:
        return get_concat(
            kleene_nfa,
            get_regex_nfa(trailing_part, indent)
        )
    else:
        return kleene_nfa

# "one or more of" operator ('+' symbol)
plus_pos = regex.find("+")
if plus_pos != -1:
    # there is a plus in the string; wrap everything before it in the "one or more of" expression
    # (uses the leftmost plus if there are more than 1)

    plus_part = regex[:plus_pos]
    trailing_part = regex[plus_pos + 1:]
    plus_nfa = get_one_or_more_of_nfa(get_regex_nfa(plus_part, indent))

    if len(trailing_part) > 0:
        return get_concat(
            plus_nfa,
            get_regex_nfa(trailing_part, indent)
        )
    else:
        return plus_nfa
```

```
    # "zero or one of" operator ('?' symbol)
    qmark_pos = regex.find("?")
    if qmark_pos != -1:
        # there is a question mark in the string; wrap everything before it in the "zero or one of"
expression
        # (uses the leftmost question mark if there are more than 1)

        leading_part = regex[:qmark_pos]
        trailing_part = regex[qmark_pos + 1:]
        zero_or_one_of_nfa = get_zero_or_one_of_nfa(get_regex_nfa(leading_part, indent))

        if len(trailing_part) > 0:
            return get_concat(
                zero_or_one_of_nfa,
                get_regex_nfa(trailing_part, indent)
            )
        else:
            return zero_or_one_of_nfa

    # no special symbols left at this point

    if len(regex) == 0:
        # base case: empty nfa for empty regex
        return NFA()
    elif len(regex) == 1:
        # base case: single symbol is directly turned into an NFA
        return get_single_symbol_regex(regex)
    else:
        # multiple characters left; apply implicit concatenation between the first character
        # and the remaining characters
        return get_concat(
            get_regex_nfa(regex[0], indent),
            get_regex_nfa(regex[1:], indent)
        )
```

OUTPUT:

```
Run:   main ×
C:\Users\aayus\AppData\Local\Programs\Python\Python310\python.exe "P:/TCSC - SEM II/Semester - II/Practicals & Journals/D
== Regex NFA ==


Example: regex=011010

> regex=011010
New regex pattern: 011010

Building NFA for regex:
(011010)
    Building NFA for regex:
    (0)
    Building NFA for regex:
    (11010)
        Building NFA for regex:
        (1)
        Building NFA for regex:
        (1010)
            Building NFA for regex:
            (1)
            Building NFA for regex:
            (010)
                Building NFA for regex:
                (0)
                Building NFA for regex:
                (10)
                    Building NFA for regex:
                    (1)
                    Building NFA for regex:
                    (0)

Built NFA in 0.000 ms.

NFA:
Alphabet: {'1', '0'}
States: {0, 1, 2, 3, 4, 5, 6}
Transition Function: {(0, '0'): {1}, (1, '1'): {2}, (2, '1'): {3}, (3, '0'): {4}, (4, '1'): {5}, (5, '0'): {6}}
Accept States: {6}
In states: {0}
Accepting: No

Version Control   Run   TODO   Problems   Terminal   Python Packages   Python Console
```

Conclusion : Successfully construct NFA using given regular expression.

Reference:

https://www.geeksforgeeks.org/regular-expression-to-nfa/

https://userpages.umbc.edu/~squire/cs451_l7.html

Experiment no – 05

Aim: Write a program to check the syntax of looping statements in Python language.

Theory: - Python For Loops

A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the for keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Example 1:-

```
fruits = ["apple", "banana", "cherry"] for x
in fruits:   print(x)
```

Example 2:-

```
for x in range (0,x):   print(x)
```

Syntax:- for counter in

iterable:

where,

for and in are keywords

counter can be any variable you used to get the value from iterable

iterable is an object that can be “iterated over”

Syntax 2:- for counter in range (start_value,end_value)

where,

for and in and range are keywords

counter can be any variable you used to get the value from iterable

start is interger used to assign the index for looping over an iterable

end is interger used to assign the the end range for looping over an iterable.

Code:-

```
import sys

str = input("Enter for loop to check syntax: \n")
striped_string = str.replace(" ","") in_pos =
str.find("in") col_pos = str.find(":")
range_exist = [1 if striped_string.find("inrange") != -1 else 0][0]

#check if for exist print("Checking for
keyword exist.....") if str[0:3] != "for":
   print("What are you trying to do?\n")
sys.exit()

#check if space after for exist
print("Checking space after for
.....") if str[3] != " ":     print("Forgot
a space after for...")     sys.exit()

#check for counter variable  print("Checking if
counter variable exist .....") if
striped_string.find("forin") != -1:     print("Forgot
to give a counter varible to loop ..")     sys.exit()

#check if space before in exist print("Checking
if space before in exist .....") if str[in_pos-1] !=
" ":
   print("Forgot the space before in...")
sys.exit()

#check if in exist print("Checking if in
keyword exist .....") if in_pos == -1:
   print("Forgot the in...")
   sys.exit()

#check if space after for exist
print("Checking if sapace after in  exist .....")
if str[in_pos+2] != " ":     print("Forgot a
space after in...")     sys.exit()
```

```
#check if colon at the end exist print("Checking if in :
exist at the end of the string.....") if col_pos+1 !=
len(str.strip()):
    print("Forgot the : ...",)     sys.exit()

#checking if counter variable exist
print("Checking if counter variable exist.....")
for_in  = str[3:in_pos].replace(" ","") if
len(for_in) > 1:     print("Something is wrong")
    sys.exit()

#check for loop variable or range variable
print("Checking if loop type is a range based.....")
if range_exist == 1:     #check if range exist
print("Checking if range is provided or not....")
if striped_string.find("range:") != -1:
print("Forgot the give a the range")     else:
        #check if space after range exist
        print("Checking if range is provided after range keyword....")
range_pos = striped_string.find("range")        print("Checking range
type....")         if striped_string.find("range(") != -1:
            print("Checking if ( exist....")
open_pos = striped_string.find("range(")+5
print("Checking if ) exist....")            close_pos =
striped_string.rindex(")")           print("Checking if (
is before ) exist....")
            #if ( is after range           if open_pos <
range_pos:                print("Where did you even put
the ( ?")
            #if ) is afrer ( and range           elif close_pos <
range_pos or close_pos < open_pos:              print("Where
did you even put the ) ?")           else:
                print("Checking if two ranges exist....")
                list_range = striped_string[open_pos+1:close_pos].replace(" ","")
                #check if comma seprating is not at start
if list_range.find(",") == 0:
                    print("What is the start of the range ?")
#check if comma seprating is not at end              elif
list_range.find(",") == len(list_range)-1:
print("What is the end of the range ?")
                       sys.exit()
```

```
elif range_exist == 0:     print("Checking if loop
variable is provided.....")        if
striped_string.find("in:") != -1:
        print("Forgot to give a iterable..")
sys.exit()

print("No errors")
```

Output:-

Input:- for i in x:

Expected output:-

No error

```
Enter for loop to check syntax:
for i in x:
Checking for keyword exist.....
Checking space after for .....
Checking if counter variable exist .....
Checking if space before in exist .....
Checking if in keyword exist .....
Checking if sapace after in  exist .....
Checking if in : exist at the end of the string.....
Checking if counter variable exist.....
Checking if loop type is a range based.....
Checking if loop variable is provided.....
No errors
```

Input:- for i in range(x,y):

Expected output:-

No error

```
Enter for loop to check syntax:
for i in range(x,y):
Checking for keyword exist.....
Checking space after for .....
Checking if counter variable exist .....
Checking if space before in exist .....
Checking if in keyword exist .....
Checking if sapace after in  exist .....
Checking if in : exist at the end of the string.....
Checking if counter variable exist.....
Checking if loop type is a range based.....
Checking if range is provided or not....
Checking if range is provided after range keyword....
Checking range type....
Checking if ( exist....
Checking if ) exist....
Checking if ( is before ) exist....
Checking if two ranges exist....
No errors
```

Input:- for i in range(x,y)

Expected output:-

: missing

```
Enter for loop to check syntax:
for i in range(x,y)
Checking for keyword exist.....
Checking space after for .....
Checking if counter variable exist .....
Checking if space before in exist .....
Checking if in keyword exist .....
Checking if sapace after in  exist .....
Checking if in : exist at the end of the string.....
Forgot the : ...
```

Input:- for in x:

Expected output:-

Missing counter variable

```
Enter for loop to check syntax:
for in x:
Checking for keyword exist.....
Checking space after for .....
Checking if counter variable exist .....
Forgot to give a counter varible to loop ..
```

Input:-

For i in :

Expected output:-

Missing iterable

```
Enter for loop to check syntax:
for i in :
Checking for keyword exist.....
Checking space after for .....
Checking if counter variable exist .....
Checking if space before in exist .....
Checking if in keyword exist .....
Checking if sapace after in  exist .....
Checking if in : exist at the end of the string.....
Checking if counter variable exist.....
Checking if loop type is a range based.....
Checking if loop variable is provided.....
Forgot to give a iterable  ..
```

Input:-

For i in (x,): :

Expected output:-

Missing end of the range

```
Enter for loop to check syntax:
for i in range(x,):
Checking for keyword exist.....
Checking space after for .....
Checking if counter variable exist .....
Checking if space before in exist .....
Checking if in keyword exist .....
Checking if sapace after in  exist .....
Checking if in : exist at the end of the string.....
Checking if counter variable exist.....
Checking if loop type is a range based.....
Checking if range is provided or not....
Checking if range is provided after range keyword....
Checking range type....
Checking if ( exist....
Checking if ) exist....
Checking if ( is before ) exist....
Checking if two ranges exist....
What is the end of the range ?
```

Conclusion:-

We successfully checked the syntax of for loop in python.

References :-

https://www.geeksforgeeks.org/c-program-to-check-syntax-of-for-loop/

Experiment no – 06

Aim: Write a program to illustrate the generation of SPM for a given grammar.

Theory: -

Algorithm:-

1. Input the grammar from the user. Print the Terminals and Non-Terminals and Start state.
2. Obtain and print FIRST, FIRST+, LAST and LAST+ matrices and print them on the screen.
3. Compute FIRST* and LAST* and print them.
4. Calculate (±) , (ϵ) and (϶) matrices using suitable formula. Writ the formula separately.
5. Superimpose (±) , (ϵ) and (϶) matrices obtain SPM. (Find if It is SPG?)

Code:-

```
grammer = [["Z","bMb"],["M","(L"],['M',"a"],["L","Ma)"]]

lhs = [i[0] for i in grammer]
rhs = [i[1] for i in grammer]

#--------------------------------#
symbol = lhs + rhs
symbols = []
for i in symbol:
   for x in range(0,len(i)):
      if  i[x] not in symbols:
         symbols.append(i[x])

#symbols = ["Z","M","L","a","b","(",")"]
#--------------------------------#

def warshall(a):
   assert (len(row) == len(a) for row in a)
   n = len(a)
   for k in range(n):
      for i in range(n):
         for j in range(n):
            a[i][j] = a[i][j] or (a[i][k] and a[k][j])
   return a


def emptyMat():
   temp= []
   for i in range(0,len(symbols)):
      x = []
      for i in range(0,len(symbols)):
         x.append(0)
      temp.append(x)
   return temp
```

```
#making empty matrix
firstMatrix = emptyMat()
firstStar = emptyMat()

I = []
#making identity matrix
identityX=0
for i in range(0,len(symbols)):
    x = []
    for j in range(0,len(symbols)):
        if j == identityX:
            x.append(1)
        else:
            x.append(0)
    identityX += 1
    I.append(x)
#making empty matrix -end


#first matrix
i = 0
for j in range(0, len(I)):
        I[i][j] = 1
        i = i+1

for i in range(0,len(lhs)):
    left = lhs[i]
    right = rhs[i]
    #first
    right = right[0]
    for i in range(0,len(symbols)):
        if symbols[i] == left:
            findL = i
            break
    for i in range(0,len(symbols)):
        if symbols[i] == right:
            findR = i
            break
    firstMatrix[findL][findR] = 1
#first matrix end

#first+ = warshal(first)
firstPlus = warshall(firstMatrix)


#------------------------------------------------------------#

#last matrix
lastMatrix = emptyMat()
lastPlus = emptyMat()
```

```
for i in range(0,len(rhs)):
    left = lhs[i]
    right = rhs[i]
    right = right[-1]
    for i in range(0,len(symbols)):
        if symbols[i] == left:
            findL = i
            break
    for i in range(0,len(symbols)):
        if symbols[i] == right:
            findR = i
            break
    lastMatrix[findL][findR] = 1


#last+ = warshal(last)
lastPlus = warshall(lastMatrix)


#last+ transpose
lastPlusT = emptyMat()

for i in range(len(lastPlus)):
   # iterate through columns
   for j in range(len(lastPlus[0])):
       lastPlusT[j][i] = lastPlus[i][j]


#----------------------------------------------------------------#
equal = emptyMat()

#eq matrix
#equal = resultant matrix
print("")
eqSet=[]
for i in rhs:
    if len(i) > 1:
        #ceiling function
        items = -(-len(i)//2)
        x = 0
        y = 1
        for j in range(0,items):
            temp = i[x] + i [y]
            eqSet.append(temp)
            x += 1
            y += 1

for i in eqSet:
    left = i[0]
```

```
    right = i[1]
    #print(f"left = {left}  right={right}")
    for j in range(0,len(symbols)):
        if symbols[j] == left:
            findL = j
            break

    for j in range(0,len(symbols)):
        if symbols[j] == right:
            findR = j
            break
    equal[findL][findR] = 1



#---------------------------------------------------------------------#
#less then
# = eq * first+
# lessThen resultant matrix

lessThen = emptyMat()

for i in range(len(equal)):
    for j in range(len(firstPlus[0])):
        for k in range(len(firstPlus)):
            lessThen[i][j] += equal[i][k] * firstPlus[k][j]



#-----------------------------------------------------------#

#first* = first+ * Identity
for i in range(0,len(firstPlus)):
    for j in range(0,len(firstPlus[0])):
        #print(f"i={i}  j={j}")
        firstStar[i][j] = firstPlus[i][j] or  I[i][j]

#----------------------------------------------------------#

#Greater then
# = last+T * eq * first*
# greaterThen resultant matrix

greaterThen = emptyMat()
eqSfp = emptyMat()

for i in range(len(equal)):
    for j in range(len(firstStar[0])):
        for k in range(len(firstStar)):
            eqSfp[i][j] += equal[i][k] * firstStar[k][j]
```

```
for i in range(len(lastPlusT)):
    for j in range(len(eqSfp[0])):
        for k in range(len(eqSfp)):
            greaterThen[i][j] += lastPlusT[i][k] * eqSfp[k][j]

#-------------------------------------#

spm = []
for i in range(0,len(symbols)+1):
    x = []
    for i in range(0,len(symbols)+1):
        x.append(0)
    spm.append(x)
spm[0][0] = "`"

for i in range(1,len(spm)):
    spm[0][i] = symbols[i-1]
    spm[i][0] = symbols[i-1]

for i in range(1, len(lessThen)+1):
    for j in range(1, len(lessThen)+1):
        if(equal[i-1][j-1]==1):
            spm[i][j] = "="
        elif(lessThen[i-1][j-1]==1):
            spm[i][j] = "<"
        elif(greaterThen[i-1][j-1]==1):
            spm[i][j] = ">"

for i in spm:
    print ('  '.join(map(str, i)))
```

Output:-

```
`  Z  M  L  b  (  a  )
Z  0  0  0  0  0  0  0
M  0  0  0  =  0  =  0
L  0  0  0  >  0  >  0
b  0  =  0  0  <  <  0
(  0  <  =  0  <  <  0
a  0  0  0  >  0  >  =
)  0  0  0  >  0  >  0
```

Conclusion:-

We successfully constructed the simple precision matrix for the given grammar.

Experiment no – 07

Aim: Write a program to illustrate the generation of OPM for a given grammar.

Theory: -

Operator precedence grammar is kinds of shift reduce parsing method. It is applied to a small class of operator grammars.

A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a∈.
- No two non-terminals are adjacent.

Operator precedence can only established between the terminals of the grammar. It ignores the non-terminal.

There are the three operator precedence relations:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".

a ⋖ b means that terminal "a" has the lower precedence than terminal "b".

a ≐ b means that the terminal "a" and "b" both have same precedence.

We first calculate leading and trailing sets for the given grammer:

LEADING

If production is of form A → aα or A → Ba α where B is Non-terminal, and α can be any string, then the first terminal symbol on R.H.S is

Leading(A) = {a}

If production is of form A → Bα, if a is in LEADING (B), then a will also be in LEADING (A).

TRAILING

If production is of form A→ αa or A → αaB where B is Non-terminal, and α can be any string then,

TRAILING (A) = {a}

If production is of form A → αB. If a is in TRAILING (B), then a will be in TRAILING (A).

Algorithms:-

LEADING

- begin
- For each non-terminal A and terminal a

  L [A, a] = false ;

- For each production of form A ⟶ aα or A → B a α

  Install (A, a) ;

- While the stack not empty

  Pop top pair (B, a) form Stack ;

  For each production of form A → B α

  Install (A, a);

- end

TRAILING

- begin
- For each non-terminal A and terminal a

  T [A, a] = false ;

- For each production of form A ⟶ αa or A → α a B

  Install (A, a) ;

- While the stack not empty

  Pop top pair (B, a) form Stack ;

  For each production of form A → αB

  Install (A, a);

- End

Procedure Install (A, a)

- begin
- If not T [A, a] then

  T [A, a] = true

  push (A, a) onto stack.

- End

Operator Precedence Relations

- begin
- For each production A → B_1, B_2, … … … . B_n

  for i = 1 to n – 1

  If B_i and B_{i+1} are both terminals then

  set B_i = B_{i+1}

  If i ≤ n − 2 and B_i and $B_{i+2}$are both terminals and B_{i+1} is non-terminal then

  set B_i = B_{i+1}

  If $B_i$is terminal & $B_{i+1}$is non-terminal then for all a in LEADING (B_{i+1})

  set B_i <. a

  If $B_i$is non-terminal & B_{i+1} is terminal then for all a in TRAILING (B_i)

  set a . > B_{i+1}

- end

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))


#=====================================================#
x = list(rules.values())
prod_rules = []
for i in x:
   for j in i:
      prod_rules.append(j)
opr = []
list_oprs = ["+","-","*","/","(",")","i"]
```

```
for i in prod_rules:
   for x in range(0,len(i)):
      if  i[x] in list_oprs:
         opr.append(i[x])

opm= []
for i in range(0,len(opr)+1):
   x = []
   for j in range(0,len(opr)+1):
      x.append("0")
   opm.append(x)

#===========================================================#
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])

#===============================================================#


print("\nOperator Precedance Matrix")
opr = sorted(opr)

opm[0][0] = "`"

for i in range(1,len(opm)):
    opm[0][i] = opr[i-1]
    opm[i][0] = opr[i-1]

for i in a:
    temp = i.split("=")
    cur_prod = temp[1]
    for j in range (0,len(cur_prod)-1):
        if cur_prod[j] in opr and cur_prod[j+1] in opr:
            opm[opr.index(cur_prod[j]) +1][opr.index(cur_prod[j+1])+1] = "="
        if j < (len(cur_prod)-2):
            if cur_prod[j] in opr and cur_prod[j+2] in opr:
                if cur_prod[j+1] in terms:
                    opm[opr.index(cur_prod[j])+1][opr.index(cur_prod[j+2])+1] = "="
        if cur_prod[j] in opr and cur_prod[j+1] in terms:
            for k in leads[temp[0]]:
                opm[opr.index(cur_prod[j])+1][opr.index(k)+1] = "<"
        if cur_prod[j] in terms and cur_prod[j+1] in opr:
            for k in trails[cur_prod[j]]:
                opm[opr.index(k)+1][opr.index(cur_prod[j+1])+1] = ">"

for i in opm:
    print ('  '.join(map(str, i)))
```

Output:-

```
================ RESTART: C:\Users\ya
LEADING(T): {'i', '*', '('}
LEADING(E): {'i', '+', '(', '*'}
LEADING(F): {'i', '('}
TRAILING(T): {'i', ')', '*'}
TRAILING(E): {'i', '+', ')', '*'}
TRAILING(F): {'i', ')'}

Operator Precedance Matrix
`   (   )   *   +   i
(   <   =   0   0   <
)   0   >   >   >   0
*   <   >   <   >   <
+   <   >   <   <   <
i   0   >   >   >   0
```

Conclusion:-

We successfully constructed the operator precedence matrix for the given grammar.

Experiment no – 08

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

Theory: -

Directed Acyclic Graph:

The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. To apply an optimization technique to a basic block, a DAG is a three-address code that is generated as the result of an intermediate code generation.

- Directed acyclic graphs are a type of data structure and they are used to apply transformations to basic blocks.
- The Directed Acyclic Graph (DAG) facilitates the transformation of basic blocks.
- DAG is an efficient method for identifying common sub-expressions.
- It demonstrates how the statement's computed value is used in subsequent statements.

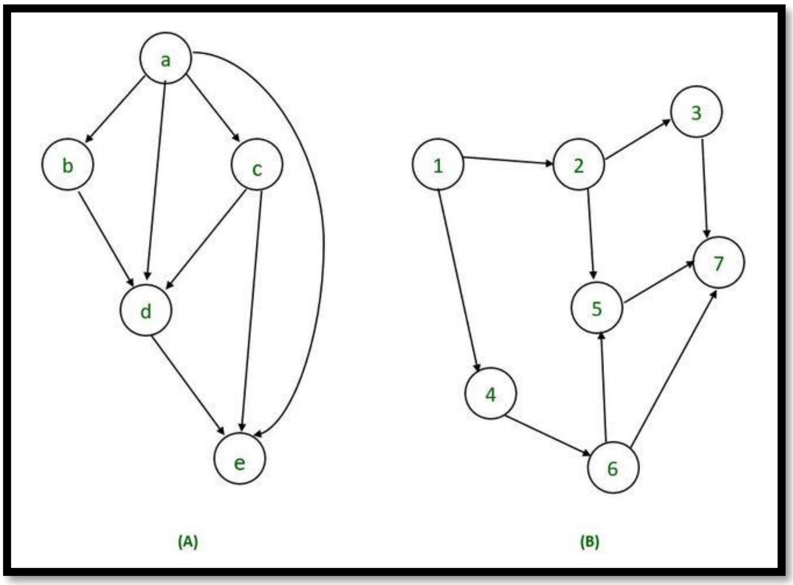


Fig: - 8.1 Examples of directed acyclic graph

Directed Acyclic Graph Characteristics:

A Directed Acyclic Graph for Basic Block is a directed acyclic graph with the following labels on nodes.

- The graph's leaves each have a unique identifier, which can be variable names or constants.
- The interior nodes of the graph are labelled with an operator symbol.
- In addition, nodes are given a string of identifiers to use as labels for storing the computed value.
- Directed Acyclic Graphs have their own definitions for transitive closure and transitive reduction.

- Directed Acyclic Graphs have topological orderings defined.

Algorithm for construction of Directed Acyclic Graph :
There are three possible scenarios for building a DAG on three address codes:

Case 1 – x = y op z
Case 2 – x = op y
Case 3 – x = y

Directed Acyclic Graph for the above cases can be built as follows :

Step 1 –

- If the y operand is not defined, then create a node (y).
- If the z operand is not defined, create a node for case(1) as node(z).

Step 2 –

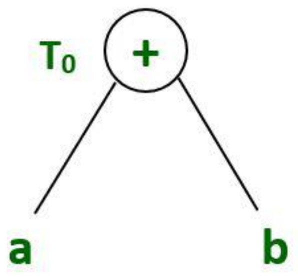
- Create node(OP) for case(1), with node(z) as its right child and node(OP) as its left child (y).
- For the case (2), see if there is a node operator (OP) with one child node (y).
- Node n will be node(y) in case (3).

Step 3 –
Remove x from the list of node identifiers. Step 2: Add x to the list of attached identifiers for node n.

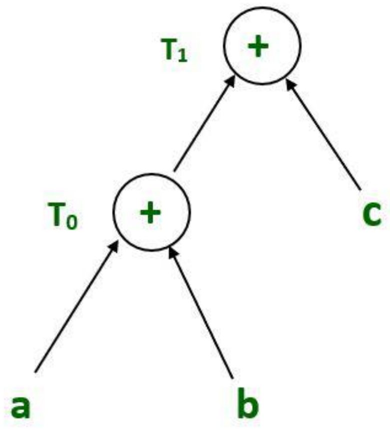
Fig:- 8.2 Example for DAG

$T_0 = a + b$ —Expression 1
$T_1 = T_0 + c$ —-Expression 2
$d = T_0 + T_1$ ——Expression 3

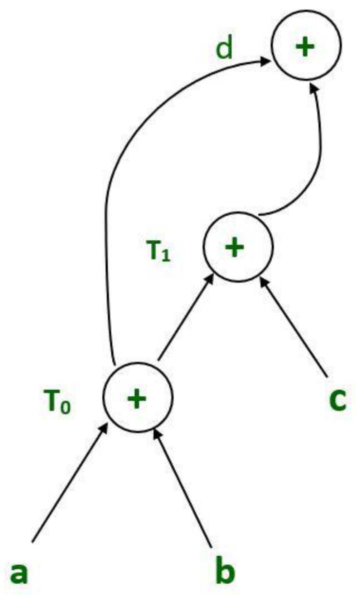
Expression 1 : ***$T_0 = a + b$***



Expression 2: **$T_1 = T_0 + c$**

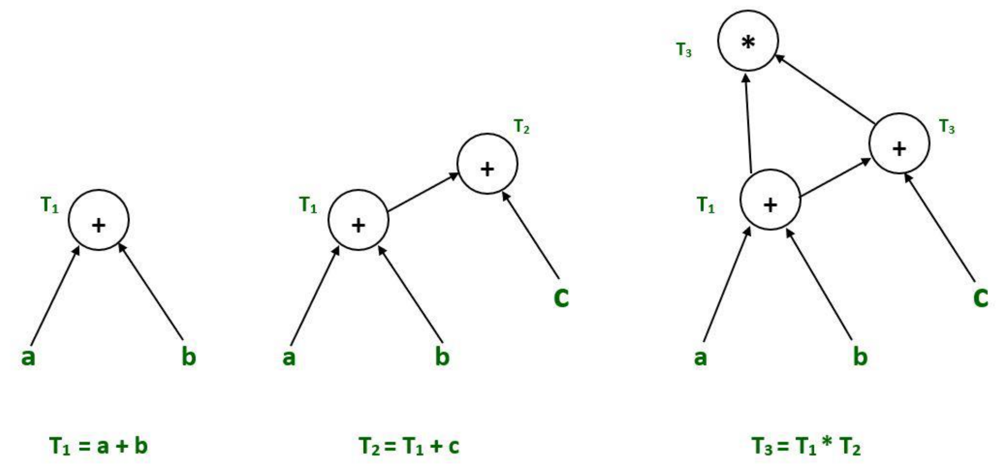


Expression 3 : $\mathbf{d = T_0 + T_1}$



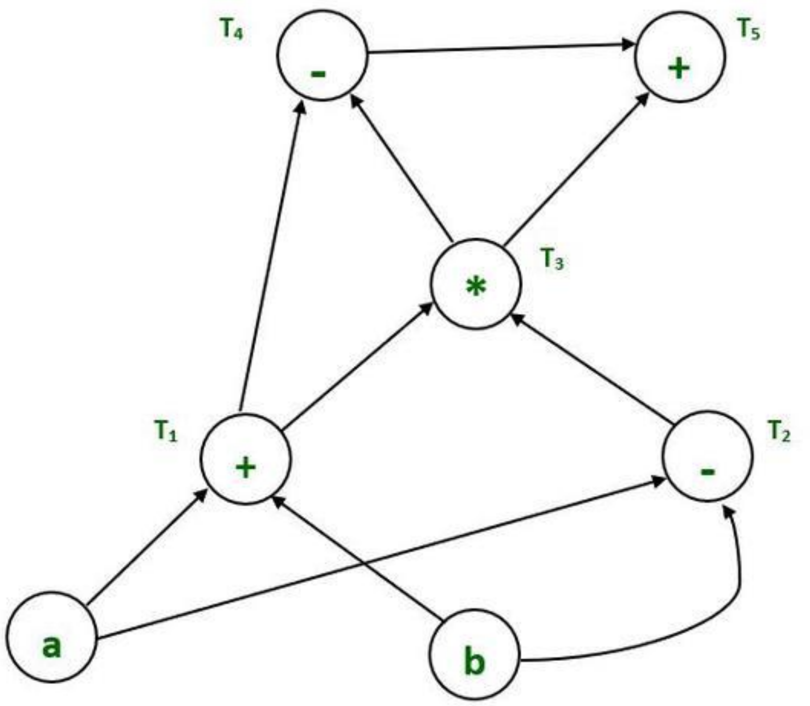
Example :

$T_1 = a + b$
$T_2 = T1 + c$
$T_3 = T1 \times T2$



Example :

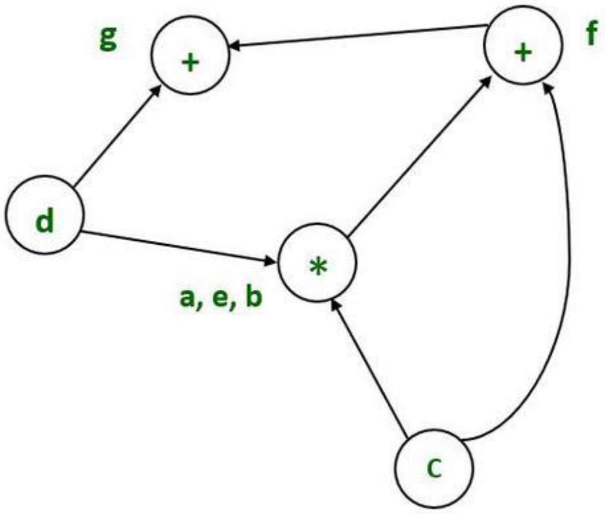
$T_1 = a + b$
$T_2 = a - b$
$T_3 = T_1 * T_2$
$T_4 = T_1 - T_3$
$T_5 = T_4 + T_3$



Example :

$a = b \times c$
$d = b$

e = d x c
b = e
f = b + c
g = f + d



Example :

$T_1 := 4*I_0$

$T_2 := a[T_1]$

$T_3 := 4*I_0$

$T_4 := b[T_3]$

$T_5 := T_2 * T_4$

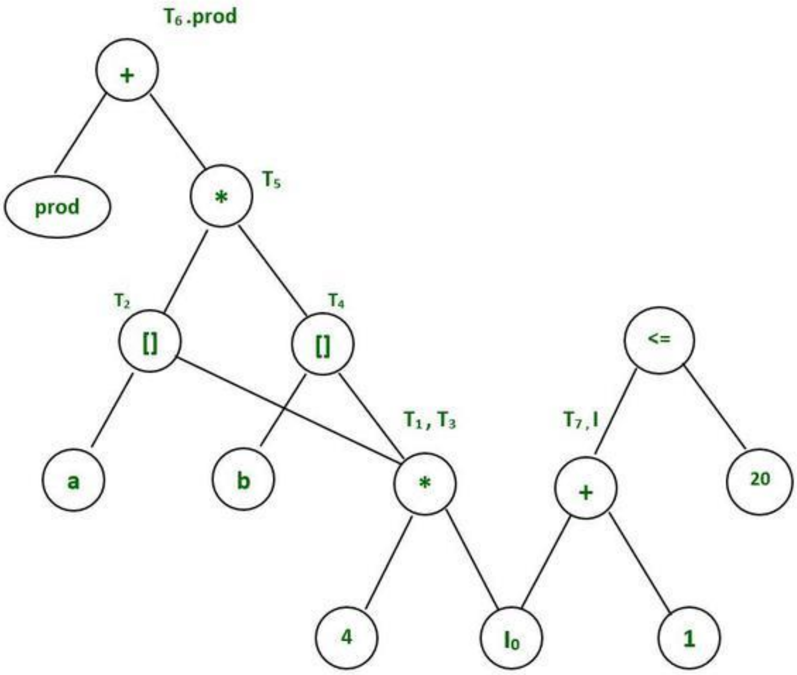
$T_6 := prod + T_5$

$prod := T_6$

$T_7 := I_0 + 1$

$I_0 := T_7$

$\text{if } I_0 <= 20 \text{ goto } 1$



Code:

 # Code to generate the DAG for the input arithmetic expression

```
#Python Program for Implementation oF DAG (Directed Acyclic Graph)
import re
grammer = ["D=B*C", "E=A+B", "B=B*C", "A=E-D"]
x=[]

opr = []
left =[]
right =[]
temp=[]
val =[]
tempNew=[]
valNew =[]

for i in grammer:
    a = i.split("=")
    val.append(a[0])
    temp.append(a[1])


loopTime = len(temp)
i = 0;
j = 0;
while i < loopTime:
    while j < loopTime:
        print(j)
```

```
        if i == j:
            j=j+1
        if temp[j] == temp[i]:
            x= val[i] + " " + val[j]
            val.pop(j)
            temp.pop(j)
            val[i] = x
            print(val)
            print(temp)
            j+=1
        j+=1
    i+=1


count = 0
for i in temp:
    if len(i) == 3:
        re.split('[+-]{1}', i)
        opr.append(i[1])
        left.append(i[0])
        right.append(i[2])
        count += 1
        continue

    if len(i) == 2:
        i.split("+")
        opr.append(i[0])
        left.append("-")
        right.append(i[1])
        count += 1
        continue
    if len(i) == 1:
        x = val[count]
        x = x + " " + i
        for k in (0,len(val)-1):
            if val[k] == i:
                temp2 = k
                val[temp2] = x
        count += 1
        continue



print("VAl\tLeft\tOperator\tRight")
for i in range(0,count):
    print(f"{val[i]}\t{left[i]}\t{opr[i]}\t\t{right[i]}")
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS P:\TCSC - SEM II\Semester - II\Practicals & Journals\Design and implementation of Modern Compilers\Practicals\Experiment no #9>  p:; cd
'p:\TCSC - SEM II\Semester - II\Practicals & Journals\Design and implementation of Modern Compilers\Practicals\Experiment no #9'; & 'C:\Use
rs\aayus\AppData\Local\Programs\Python\Python310\python.exe' 'c:\Users\aayus\.vscode\extensions\ms-python.python-2022.4.1\pythonFiles\lib\p
ython\debugpy\launcher' '55997' '--' 'p:\TCSC - SEM II\Semester - II\Practicals & Journals\Design and implementation of Modern Compilers\Pr
acticals\Experiment no #9\dag.py'
0
2
['D B', 'E', 'A']
['B*C', 'A+B', 'E-D']
VAl     Left    Operator        Right
D B     B       *               C
E       A       +               B
A       E       -               D
PS P:\TCSC - SEM II\Semester - II\Practicals & Journals\Design and implementation of Modern Compilers\Practicals\Experiment no #9>
```

Conclusion:-

We successfully constructed and checked DAG for the input arithmetic expression.

Reference:

https://www.techiedelight.com/check-given-digraph-dag-directed-acyclic-graph-not/

Experiment no – 09

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

Theory: -

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Before
```
for (int i = 0; i < n; ++i) {
  a[i] = b[i] * 7 + c[i] / 13;
}
```

After
```
for (int i = 0; i < n % 3; ++i) {
   a[i] = b[i] * 7 + c[i] / 13;
}
for (; i < n; i += 3) {
  a[i] = b[i] * 7 + c[i] / 13;
  a[i + 1] = b[i + 1] * 7 + c[i + 1] / 13;
  a[i + 2] = b[i + 2] * 7 + c[i + 2] / 13;
```

- If fixed number of iterations, maybe turn loop into sequence of statements!
- Before
```
for (int i = 0; i < 6; ++i) {
  if (i % 2 == 0) foo(i); else bar(i);
}
```
- After
```
foo(0);
bar(1);
foo(2);
bar(3);
foo(4);
bar(5);
```

Example:

// This program does not uses loop unrolling.

```
#include<stdio.h>
int main(void)
{
	for (int i=0; i<5; i++)
		printf("Hello\n"); //print hello 5 times
	return 0;
}
```

// **This program uses loop unrolling.**

```
#include<stdio.h>
```

```
int main(void)
{
    // unrolled the for loop in program 1
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    printf("Hello\n");
    return 0;
}
```

Loop splitting (or loop peeling) is a compiler optimization technique. It attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range.

A useful special case is loop peeling, which can simplify a loop with a problematic first (or first few) iteration by performing that iteration separately before entering the loop.

Here is an example of loop peeling. Suppose the original code looks like this:

p = 10; for (i=0; i<10; ++i) { y [i] = x [i] + x [p] ; p = i; }

In the above code, only in the 1st iteration is p=10. For all other iterations p=i-1. We get the following after loop peeling:

y [0] = x [0] + x [10] ; for (i=1; i<10; ++i) { y [i] = x [i] + x [i-1] ; }

#LOOP_UNROLLING

```
public
class loop_unrolling
{
public
   static void main(String[] args)
   {
      int[] array1 = new int[5];
      long t1 = System.currentTimeMillis();
      // Version 1: assign elements in a loop.
      for (int I = 0; i < 10000000; i++)
      {
         for (int x = 0; x < array1.length; x++)
         {
            array1[x] = x;
         }
      }
      long t2 = System.currentTimeMillis();
      // Version 2: unroll the loop and use a list of statements.
      for (int i = 0; i < 10000000; i++)
      {
         array1[0] = 0;
         array1[1] = 1;
         array1[2] = 2;
         array1[3] = 3;
         array1[4] = 4;
      }
      long t3 = System.currentTimeMillis();
      // ... Times
      System.out.println("Time taken by processor before loop unrolling:--> " + (t2 - t1));
      System.out.println("Time taken by processor after loop unrolling:--> " + (t3 - t2));
   }
}
```

Output: -

```
Time taken by processor before loop unrolling:--> 71

Time taken by processor after loop unrolling:--> 20
```

#LOOP_JAMMING

```
public
class loop_jamming
{
public
   static void main(String[] args)
   {
      int[] array1 = {10, 20, 30};
      int[] array2 = {20, 10, 30};
      int[] array3 = {40, 40, 10};
      long t1 = System.currentTimeMillis();
      // Version 1: loop over each array separately. for (int i
      = 0;
      i < 10000000; i++)
      {
         int sum = 0;
         for (int x = 0; x < array1.length; x++)
         {
            sum += array1[x];
         }
         for (int x = 0; x < array2.length; x++)
         {
            sum += array2[x];
         }
         for (int x = 0; x < array3.length; x++)
         {
            sum += array3[x];
         }
         if (sum != 210)
         {
            System.out.println(false);
         }
      }
      long t2 = System.currentTimeMillis();
      // Version 2: jam loops together. for (int i
      = 0;
      i < 10000000; i++)
      {
         int sum = 0;
         for (int x = 0; x < array1.length; x++)
         {
            sum += array1[x];
            sum +=
               array2[x];
            sum +=
               array3[x];
         }
```

```
            if (sum != 210)
            {
                System.out.println(false);
            }
        }
        long t3 = System.currentTimeMillis();
        // ... Times.
        System.out.println("Before loop jamming --- >" + (t2 - t1));
        System.out.println("After loop jamming --- >" + (t3 - t2));
    }
}
```

Output:

Before loop jamming --- >157

After loop jamming --- >104

Conclusion: Successfully implemented loop unrolling and loop splitting.