School Of Computing

LAB MANUAL

III Year / VI SEM (CSE -CYS)

20CYS315 – AUTOMATA THEORY AND COMPILER DESIGN

Amrita Vishwa Vidyapeetham Chennai – 601 103, Tamil Nadu, India.

BONAFIDE CERTIFICATE

University Register Number: CH.EN.U4CYS22044

This is to certify that this is a bona fide record of work done by Mr. Shabhareash S, a B.Tech Computer Science and Engineering (Cyber Security) student from the 2022-2026 batch at Amrita Vishwa Vidyapeetham, Chennai Campus.

Internal Examiner 1 **Internal Examiner 2**

List of Experiments

Experiment – 1: Construct the DFA for given string

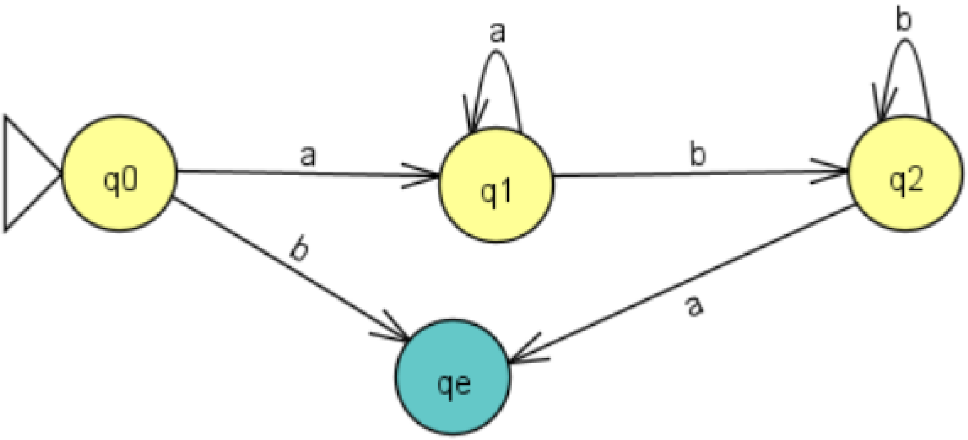
Aim: writing a program to construct DFA for given string

CODE:

```python
class State:
    Q0 = 0
    Q1 = 1
    Q2 = 2
def is_accepted(input_string):
    current_state = State.Q0

    for symbol in input_string:
        if current_state == State.Q0:
            if symbol == 'a':
                current_state = State.Q1
            else:
                return False

        elif current_state == State.Q1:
            if symbol == 'a':
                current_state = State.Q1
            elif symbol == 'b':
                current_state = State.Q2
            else:
                return False

        elif current_state == State.Q2:
            if symbol == 'b':
                current_state = State.Q2
            else:
                return False

    return current_state == State.Q2

if __name__ == "__main__":
    input_string = input("Enter a string: ")
    if is_accepted(input_string):
        print("String is accepted.")
    else:
        print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q1.py
Enter a string: aaaba
String is not accepted.
PS D:\automata> python q1.py
Enter a string: aab
String is accepted.
```



Experiment-2: Construct the NFA for given string

Aim: writing a program to construct the NFA for given string

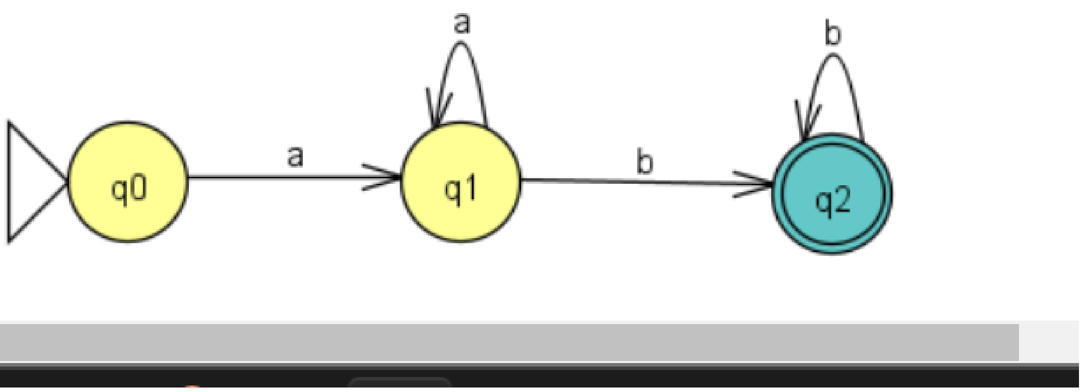
CODE:

```
from collections import defaultdict

class NFA:
    def __init__(self):
        self.transitions = defaultdict(set)
        self.start_state = 0
        self.final_states = set()

    def add_transition(self, from_state, symbol, to_state):
        self.transitions[(from_state, symbol)].add(to_state)

    def add_final_state(self, state):
        self.final_states.add(state)

    def epsilon_closure(self, states):
        stack = list(states)
        closure = set(states)

        while stack:
            state = stack.pop()
            if (state, 'ε') in self.transitions:
                for next_state in self.transitions[(state, 'ε')]:
                    if next_state not in closure:
                        closure.add(next_state)
                        stack.append(next_state)
        return closure

    def is_accepted(self, input_string):
        current_states = self.epsilon_closure({self.start_state})

        for symbol in input_string:
            next_states = set()
            for state in current_states:
                if (state, symbol) in self.transitions:
                    next_states.update(self.transitions[(state, symbol)])
            current_states = self.epsilon_closure(next_states)

        return any(state in self.final_states for state in current_states)
nfa = NFA()
Q0, Q1, Q2 = 0, 1, 2
nfa.add_transition(Q0, 'a', Q1)
nfa.add_transition(Q1, 'a', Q1)
nfa.add_transition(Q1, 'b', Q2)
```

```
nfa.add_transition(Q2, 'b', Q2)
nfa.add_final_state(Q2)
input_string = input("Enter a string: ")
if nfa.is_accepted(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q2.py
Enter a string: aaaaaabbb
String is accepted.
PS D:\automata> python q2.py
Enter a string: abaaab
String is not accepted.
```



Result :

Thus a program to construct a NFA for a given string has been done and verified.

Experiment-3: Verify String acceptance using DFA

Aim: writing a program to verify string acceptance using DFA

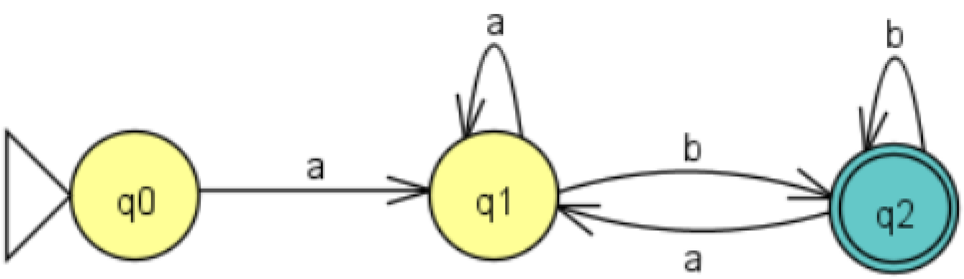
CODE:

```
from collections import defaultdict

class Automaton:
    def __init__(self, is_dfa=True):
        self.transitions = defaultdict(set)
        self.start_state = 0
        self.final_states = set()
        self.is_dfa = is_dfa

    def add_transition(self, from_state, symbol, to_state):
        if self.is_dfa:
            self.transitions[(from_state, symbol)] = {to_state}
        else:
            self.transitions[(from_state, symbol)].add(to_state)

    def add_final_state(self, state):
        self.final_states.add(state)

    def is_accepted(self, input_string):
        current_states = {self.start_state}

        for symbol in input_string:
            next_states = set()
            for state in current_states:
                if (state, symbol) in self.transitions:
                    next_states.update(self.transitions[(state, symbol)])
            if not next_states:
                return False
            current_states = next_states

        return any(state in self.final_states for state in current_states)

automaton_type = input("Choose Automaton Type (DFA/NFA): ").strip().upper()
is_dfa = automaton_type == "DFA"

automaton = Automaton(is_dfa)

Q0, Q1, Q2 = 0, 1, 2

automaton.add_transition(Q0, 'a', Q1)
automaton.add_transition(Q1, 'a', Q1)
```

```
automaton.add_transition(Q1, 'b', Q2)
automaton.add_transition(Q2, 'b', Q2)

automaton.add_final_state(Q2)

input_string = input("Enter a string: ")
if automaton.is_accepted(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q3.py
Choose Automaton Type (DFA/NFA): DFA
Enter a string: abab
String is not accepted.
PS D:\automata> python q3.py
Choose Automaton Type (DFA/NFA): NFA
Enter a string: aaabb
String is accepted.
```



Result :

Thus a program to construct a NFA for a given string has been done and verified.

Experiment-4: Conversion of NFA to DFA

Aim: writing a program to convert NFA to DFA

CODE:

```
from collections import defaultdict

class FiniteAutomaton:
    def __init__(self, is_dfa=True):
        self.transitions = defaultdict(set)
        self.start_state = 0
        self.final_states = set()
        self.is_dfa = is_dfa

    def add_transition(self, from_state, symbol, to_state):
        self.transitions[(from_state, symbol)].add(to_state)

    def add_final_state(self, state):
        self.final_states.add(state)

    def epsilon_closure(self, states):
        stack = list(states)
        closure = set(states)

        while stack:
            state = stack.pop()
            if (state, 'ε') in self.transitions:
                for next_state in self.transitions[(state, 'ε')]:
                    if next_state not in closure:
                        closure.add(next_state)
                        stack.append(next_state)
        return closure

    def is_accepted(self, input_string):
        current_states = {self.start_state}

        if not self.is_dfa:
            current_states = self.epsilon_closure(current_states)

        for symbol in input_string:
            next_states = set()
            for state in current_states:
                if (state, symbol) in self.transitions:
                    next_states.update(self.transitions[(state, symbol)])

            if not next_states:
                return False
```

```
            if not self.is_dfa:
                next_states = self.epsilon_closure(next_states)

            current_states = next_states

        return any(state in self.final_states for state in current_states)

    def display_transitions(self):
        print("Transition Table:")
        for (state, symbol), next_states in self.transitions.items():
            print(f"δ({state}, '{symbol}') -> {next_states}")
        print(f"Start State: {self.start_state}")
        print(f"Final States: {self.final_states}")
automaton = FiniteAutomaton(is_dfa=False)
Q0, Q1, Q2 = 0, 1, 2
automaton.add_transition(Q0, 'a', Q1)
automaton.add_transition(Q1, 'a', Q1)
automaton.add_transition(Q1, 'b', Q2)
automaton.add_transition(Q2, 'b', Q2)
automaton.add_final_state(Q2)
automaton.display_transitions()
input_string = input("Enter a string to check: ")
if automaton.is_accepted(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q4.py
Enter number of states in NFA: 3
Enter transition for state ('q0', 0) : q1
Enter transition for state ('q0', 1) : q2
Enter transition for state ('q1', 0) : q1
Enter transition for state ('q1', 1) : q2
Enter transition for state ('q2', 0) : q2
Enter transition for state ('q2', 1) : q1

Transistion Diagram for NFA :
('q0', 0)  :  ['q1']
('q0', 1)  :  ['q2']
('q1', 0)  :  ['q1']
('q1', 1)  :  ['q2']
('q2', 0)  :  ['q2']
('q2', 1)  :  ['q1']

Transistion Diagram for DFA :
('q0', 0)  :  ['q1']
('q0', 1)  :  ['q2']
('q1', 0)  :  ['q1']
('q1', 1)  :  ['q2']
('q2', 0)  :  ['q2']
('q2', 1)  :  ['q1']
PS D:\automata>
```

Result :

Thus a program to convert an NFA to DFA has been done and verified.

Experiment-5: Conversion of Epsilon NFA to NFA

Aim: Write a program to Convert the Epsilon NFA into NFA

CODE:

```
def add_epsilon_closure(epsilon_nfa, state, temp):
    if state != '':
        temp.add(state)
        for i in epsilon_nfa.get((state, 'E'), []):
            add_epsilon_closure(epsilon_nfa, i, temp)

def calculate_epsilon_closure(epsilon_nfa):
    epsilon_closures = {}
    for state in states:
        temp = set()
        add_epsilon_closure(epsilon_nfa, state, temp)
        epsilon_closures[state] = temp
    return epsilon_closures

def calculate_transitions(epsilon_nfa, epsilon_closures):
    transitions = {}
    for state in states:
        transitions[state] = {}
        for symbol in alphabet:
            reachable_states = set()
            for s in epsilon_closures[state]:
                targets = epsilon_nfa.get((s, symbol), [])
                reachable_states.update(targets)
            closure = set()
            for s in reachable_states:
                if s in epsilon_closures:
                    closure.update(epsilon_closures[s])
            transitions[state][symbol] = closure
    return transitions

def convert_epsilon_nfa_to_nfa(epsilon_nfa):
    epsilon_closures = calculate_epsilon_closure(epsilon_nfa)
    transitions = calculate_transitions(epsilon_nfa, epsilon_closures)
    return transitions

alphabet = ['0', '1']

epsilon_nfa = {}
n = int(input("Enter number of states in Epsilon-NFA: "))
s = []
states = []

for i in range(n):
```

```python
    s.append(('q'+str(i), 'E'))
    s.append(('q'+str(i), '0'))
    s.append(('q'+str(i), '1'))
    states.append('q'+str(i))

for i in s:
    transitions = list(map(str, input(f"Enter transitions for state {i} :
").split(' ')))
    epsilon_nfa[i] = transitions

print("Epsilon-NFA : ", epsilon_nfa)
print("\nTransition Diagram for Epsilon-NFA : ")
for i in epsilon_nfa:
    print(i, " : ", epsilon_nfa[i])

print(states, alphabet)

nfa = convert_epsilon_nfa_to_nfa(epsilon_nfa)
print("Transition Table for NFA:")
for state, transitions in nfa.items():
    print(f"State {state}: {transitions}")
```

OUTPUT:

```
PS D:\automata> python q5.py
Enter number of states in Epsilon-NFA: 3
Enter transitions for state ('q0', 'E') : q1
Enter transitions for state ('q0', '0') : q2
Enter transitions for state ('q0', '1') : q1
Enter transitions for state ('q1', 'E') : q2
Enter transitions for state ('q1', '0') : q2
Enter transitions for state ('q1', '1') : q1
Enter transitions for state ('q2', 'E') : q2
Enter transitions for state ('q2', '0') : q1
Enter transitions for state ('q2', '1') : q2
Epsilon-NFA :  {('q0', 'E'): ['q1'], ('q0', '0'): ['q2'], ('q0', '1'): ['q1'], ('q1', 'E'): ['q2'], ('q1', '0'): ['q2'], ('
q1', '1'): ['q1'], ('q2', 'E'): ['q2'], ('q2', '0'): ['q1'], ('q2', '1'): ['q2']}

Transition Diagram for Epsilon-NFA :
('q0', 'E')  :  ['q1']
('q0', '0')  :  ['q2']
('q0', '1')  :  ['q1']
('q1', 'E')  :  ['q2']
('q1', '0')  :  ['q2']
('q1', '1')  :  ['q1']
('q2', 'E')  :  ['q2']
('q2', '0')  :  ['q1']
('q2', '1')  :  ['q2']
```

Result:

Thus a program to convert Epsilon NFA to NFA has been done sucessfully

Experiment-6: Convert the Epsilon NFA to DFA

Aim: Write a Python program to Write a program to Convert the Epsilon NFA to DFA

CODE:

```
def fillStates(dfa, state, input):
    l = []
    for i in dfa[state]:
        if (i, input) in dfa:
            l.append(dfa[(i, input)])
    newEntry = [element for innerList in l for element in innerList]
    newEntry = list(filter(lambda a: a != '', newEntry))
    return newEntry
nfa = {}
n = int(input("Enter number of states in NFA: "))
state = []
for i in range(n):
    state.append(('q'+str(i), 0))
    state.append(('q'+str(i), 1))
for i in state:
    nfa[i] = list(map(str, input(f"Enter transition for state {i} : ").split('
')))
print("\nTransistion Diagram for NFA : ")
for i in nfa:
    print(i, " : ", nfa[i])

dfa = nfa
states = set()
for i in dfa.keys():
    states.add(i[0])
newDfa = {}
for i in dfa:
    newState = ""
    for j in dfa[i]:
        newState += j
    if newState not in states and newState != "":
        states.add(newState)
        newDfa[(newState, 0)] = fillStates(dfa, i, 0)
        newDfa[(newState, 1)] = fillStates(dfa, i, 1)
for i in newDfa:
    dfa[i] = newDfa[i]
print("\nTransistion Diagram for DFA : ")
for i in dfa:
    print(i, " : ", dfa[i])
```

OUTPUT:

```
PS D:\automata> python q6.py
Enter number of states in NFA: 3
Enter transition for state ('q0', 0) : q1
Enter transition for state ('q0', 1) :
Enter transition for state ('q1', 0) : q0
Enter transition for state ('q1', 1) : q2
Enter transition for state ('q2', 0) : q1
Enter transition for state ('q2', 1) :

Transistion Diagram for NFA :
('q0', 0)  :  ['q1']
('q0', 1)  :  ['']
('q1', 0)  :  ['q0']
('q1', 1)  :  ['q2']
('q2', 0)  :  ['q1']
('q2', 1)  :  ['']

Transistion Diagram for DFA :
('q0', 0)  :  ['q1']
('q0', 1)  :  ['']
('q1', 0)  :  ['q0']
('q1', 1)  :  ['q2']
('q2', 0)  :  ['q1']
('q2', 1)  :  ['']
PS D:\automata>
```

Result:

Thus a program to convert Epsilon NFA to DFA has been done sucessfully

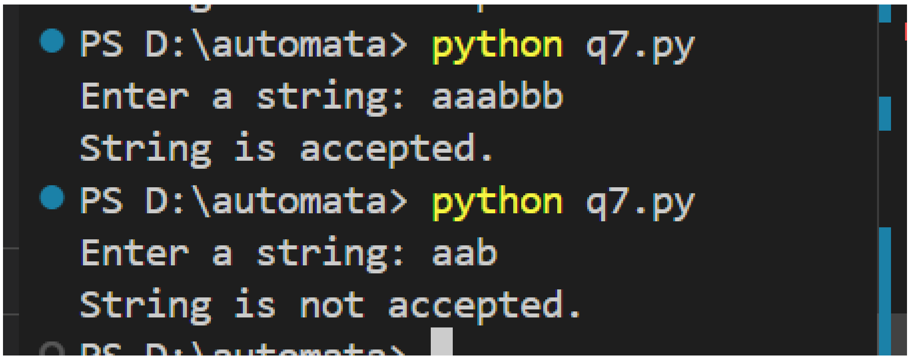
Experiment-7: construct a Pushdown Automaton

Aim: Write a Python program to construct a Pushdown Automaton (PDA) for the following Language L={a^n,b^n|n>=1}

CODE:

```python
def is_accepted_by_pda(string):
    stack = []

    for symbol in string:
        if symbol == 'a':
            stack.append('a')
        elif symbol == 'b':
            if stack:
                stack.pop()
            else:
                return False
        else:
            return False

    return len(stack) == 0

input_string = input("Enter a string: ")

if is_accepted_by_pda(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:



Result:

Thus a program to construct a Push Down automation has been done sucessfully

Experiment-8: construct a Pushdown Automation

Aim: Write a Python program to construct a Pushdown Automaton (PDA) for the following Language L={ww^r|w=(a+b)^+}

CODE:

```
def is_accepted_by_pda(string):
    stack = []
    length = len(string)

    if length % 2 != 0:
        return False

    mid = length // 2

    for i in range(mid):
        stack.append(string[i])

    for i in range(mid, length):
        if not stack or stack.pop() != string[i]:
            return False

    return len(stack) == 0

input_string = input("Enter a string: ")

if is_accepted_by_pda(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q8.py
Enter a string: baab
String is accepted.
PS D:\automata> python q8.py
Enter a string: abcba
String is not accepted.
PS D:\automata>
```

Result:

Thus a program to construct a push down automation for this string

L={ww^r|w=(a+b)^+} has been done successfully.

Experiment-9: construct a Pushdown Automaton

Aim: Write a Python program to construct a Pushdown Automaton (PDA) for the following Language L={wCw^r|w belongs to (a+b)^*}

CODE:

```
def is_accepted_by_pda(string):
    stack = []
    found_C = False
    index = 0

    while index < len(string):
        char = string[index]

        if not found_C:
            if char == 'C':
                found_C = True
            else:
                stack.append(char)
        else:
            if not stack or stack.pop() != char:
                return False

        index += 1

    return len(stack) == 0 and found_C

input_string = input("Enter a string: ")

if is_accepted_by_pda(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q9.py
Enter a string: abCba
String is accepted.
PS D:\automata> python q9.py
Enter a string: abCCba
String is not accepted.
```

Result:

Thus a program to construct a push down automation for this string

L={wCw^r|w belongs to (a+b)^*} has been done successfully.

Experiment-10: construct a Pushdown Automaton

Aim: Write a Python program to construct a Pushdown Automaton (PDA) for the following Language L={0^n1^m2^m3^n|n,m>=1}

CODE:

```
def is_accepted_by_pda(string):
    stack1 = []
    stack2 = []
    stage = 1

    for symbol in string:
        if stage == 1:
            if symbol == '0':
                stack1.append('0')
            elif symbol == '1':
                stack2.append('1')
                stage = 2
            else:
                return False

        elif stage == 2:
            if symbol == '1':
                stack2.append('1')
            elif symbol == '2':
                if stack2:
                    stack2.pop()
                else:
                    return False
                stage = 3
            else:
                return False

        elif stage == 3:
            if symbol == '2':
                if stack2:
                    stack2.pop()
                else:
                    return False
            elif symbol == '3':
                if stack1:
                    stack1.pop()
                else:
                    return False
```

```python
                stage = 4
            else:
                return False

        elif stage == 4:
            if symbol == '3':
                if stack1:
                    stack1.pop()
                else:
                    return False
            else:
                return False

    return len(stack1) == 0 and len(stack2) == 0

input_string = input("Enter a string: ")

if is_accepted_by_pda(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q10.py
Enter a string: 011223
String is accepted.
PS D:\automata> python q10.py
Enter a string: 0012233
String is not accepted.
```

Result:

Thus a program to construct a push down automation for this string

L={0^n1^m2^m3^n|n,m>=1} has been done successfully.

Experiment-11: construct a Pushdown Automaton

Aim: Write a Python program to construct a Pushdown Automaton (PDA) for the following Language L={a^nb^2n|n>+1}

```
def is_accepted_by_pda(string):
    stack = []
    count_b = 0

    for symbol in string:
        if symbol == 'a':
            stack.append('a')
        elif symbol == 'b':
            count_b += 1
            if stack:
                stack.pop()
            elif count_b <= len(stack) * 2:
                continue
            else:
                return False
        else:
            return False

    return len(stack) == 0 and count_b == 2 * (count_b // 2)

input_string = input("Enter a string: ")

if is_accepted_by_pda(input_string):
    print("String is accepted.")
else:
    print("String is not accepted.")
```

OUTPUT:

```
PS D:\automata> python q11.py
Enter a string: aabb
String is accepted.
PS D:\automata> python q11.py
Enter a string: aba
String is not accepted.
```

Result:

Thus a program to construct a push down automation for this string

L={a^nb^2n|n>+1} has been done successfully.

Experiment – 12: Find the first and follow for the given CFG

Aim:

Write a program to find the first and follow for the given CFG

CODE:

```
from collections import defaultdict

class Grammar:
    def __init__(self, productions):
        self.productions = productions
        self.first = defaultdict(set)
        self.follow = defaultdict(set)
        self.start_symbol = list(productions.keys())[0]
        self.compute_first()
        self.compute_follow()

    def compute_first(self):
        for non_terminal in self.productions:
            self.first[non_terminal] = self.find_first(non_terminal)

    def find_first(self, symbol):
        if symbol.islower() or symbol in "()+*":
            return {symbol}
        first_set = set()
        for production in self.productions[symbol]:
            if production == "":
                first_set.add("ε")
                continue
            for char in production:
                char_first = self.find_first(char)
                first_set.update(char_first - {"ε"})
                if "ε" not in char_first:
                    break
            else:
                first_set.add("ε")
        return first_set

    def compute_follow(self):
        self.follow[self.start_symbol].add("$")
        changed = True
        while changed:
            changed = False
            for non_terminal, productions in self.productions.items():
                for production in productions:
                    trailer = self.follow[non_terminal].copy()
                    for symbol in reversed(production):
```

```
                        if symbol.isupper():
                            if trailer - self.follow[symbol]:
                                self.follow[symbol].update(trailer)
                                changed = True
                            if "ε" in self.first[symbol]:
                                trailer.update(self.first[symbol] - {"ε"})
                            else:
                                trailer = self.first[symbol]
                        else:
                            trailer = {symbol}

    def display_results(self):
        for non_terminal in self.productions:
            print(f"First({non_terminal}) = {self.first[non_terminal]}")
        print()
        for non_terminal in self.productions:
            print(f"Follow({non_terminal}) = {self.follow[non_terminal]}")

if __name__ == "__main__":
    n = int(input("Enter the number of productions: "))
    productions = defaultdict(list)
    for _ in range(n):
        lhs, rhs = input("Enter production (A=xyz format): ").split("=")
        productions[lhs].extend(rhs.split("|"))

    grammar = Grammar(productions)
    grammar.display_results()
```

OUTPUT:

```
PS D:\automata> python q12.py
Enter the number of productions: 3
Enter production (A=xyz format): A=abc
Enter production (A=xyz format): A=def
Enter production (A=xyz format): A=g
First(A) = {'d', 'g', 'a'}
```

Result:

Thus a program to find the first and follow for the given CFG has been done successfully.

Experiment – 13: Generate Three Address Code for the given Expression

Aim:

Write a program to generate Three Address Code for the given Expression

CODE:

```
def generate_tac(expression):
    tokens = expression.split("+")
    tokens = [token.strip() for token in tokens]
    tac_code = []
    temp_vars = []

    temp_var_count = 1
    t = f"t{temp_var_count} = {tokens[0]} + {tokens[1]}"
    tac_code.append(t)
    temp_vars.append(f"t{temp_var_count}")

    for i in range(2, len(tokens)):
        temp_var_count += 1
        t = f"t{temp_var_count} = {temp_vars[-1]} + {tokens[i]}"
        tac_code.append(t)
        temp_vars.append(f"t{temp_var_count}")

    tac_code.append(f"x = {temp_vars[-1]}")

    return tac_code

expression = input("Enter an arithmetic expression : ")

tac_result = generate_tac(expression)

print("\nThree-Address Code:")
for line in tac_result:
    print(line)
```

OUTPUT:

```
PS D:\automata> python q13.py
Enter an arithmetic expression : a + b + c + d

Three-Address Code:
t1 = a + b
t2 = t1 + c
t3 = t2 + d
x = t3
```

Result:

Thus a program to generate Three Address Code for the given Expression has been done successfully.