

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.U4CYS22026

This is to certify that this is a bonafide record work done
by Mr.Kiruthik Pranav P V studying B.Tech Computer Science
Engineering in Cyber Security in 2022-26 in Amrita Vishwa
Vidyapeetham, Chennai Campus.

Date: 19-03-2025
Examiner

List of Experiments

S.NO	TOPICS
1	Write a program to construct the DFA for the given Strings.
2	Write a program to construct the NFA for the given Strings.
3	Write a program for String Acceptance using NFA or DFA.
4	Write a program to Convert the NFA into DFA.
5	Write a program to Convert the Epsilon NFA into NFA
6	Write a program to Convert the Epsilon NFA to DFA
7	Write a Python program to construct a Pushdown Automaton (PDA) for the following Language $L=\{a^n, b^n n \geq 1\}$
8	Write a Python program to construct a Pushdown Automaton (PDA) for the following Language $L=\{ww^r w=(a+b)^+\}$
9	Write a Python program to construct a Pushdown Automaton (PDA) for the following Language $L=\{wCw^r w \text{ belongs to } (a+b)^*\}$
10	Write a Python program to construct a Pushdown Automaton (PDA) for the following Language $L=\{0^n 1^m 2^m 3^n n, m \geq 1\}$
11	Write a Python program to construct a Pushdown Automaton (PDA) for the following Language $L=\{a^n b^{2n} n \geq 1\}$
12	Write a program to find the first and follow for the given CFG
13	Write a program to generate Three Address Code for the given Expression

1. Write a program to construct the DFA for the given Strings.

CODE:

```
from enum import Enum

class State(Enum):
    Q0 = 0
    Q1 = 1
    Q2 = 2

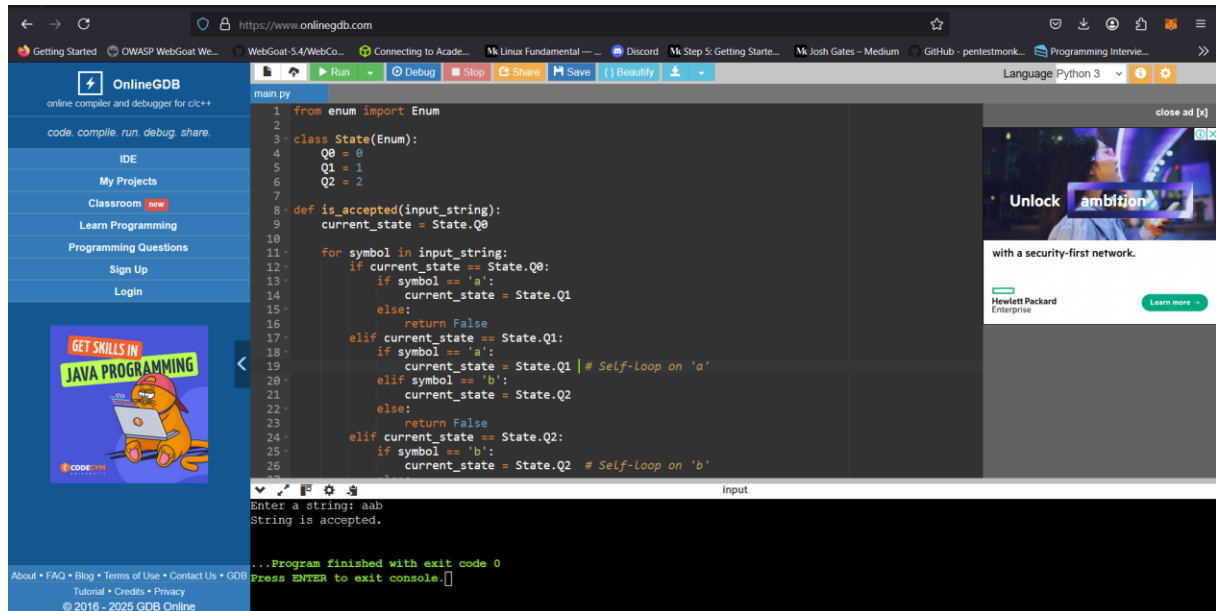
def is_accepted(input_string):
    current_state = State.Q0

    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 # Self-loop on 'a'
            elif symbol == 'b':
                current_state = State.Q2
            else:
                return False
        elif current_state == State.Q2:
            if symbol == 'b':
                current_state = State.Q2 # Self-loop on 'b'
            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:

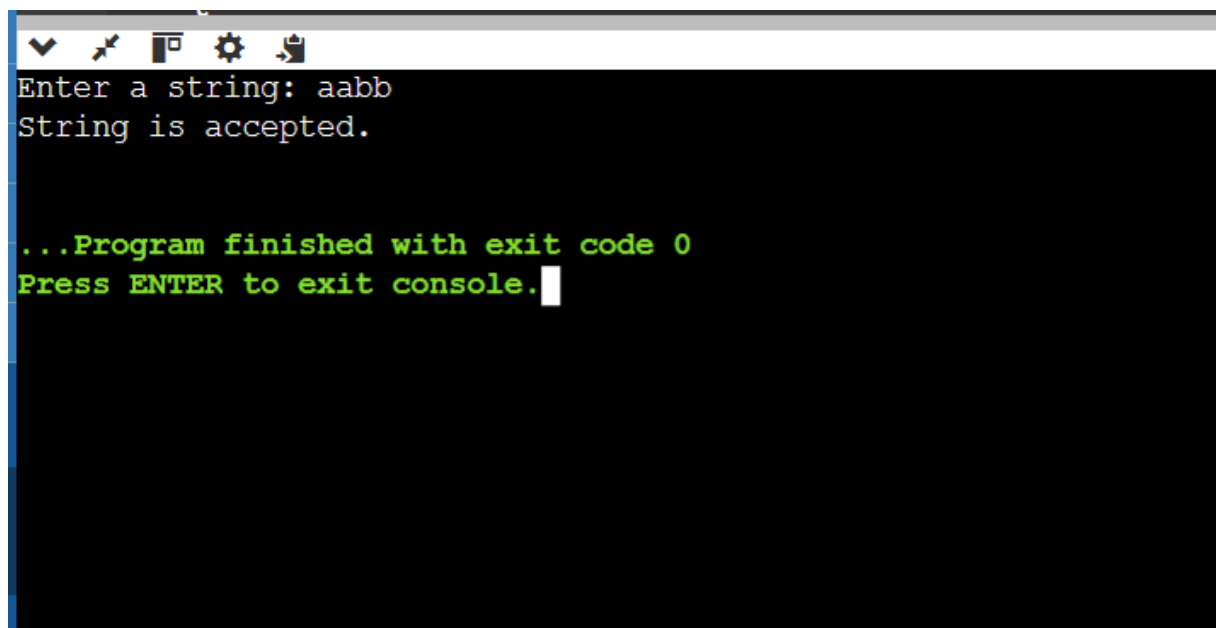


The screenshot shows the OnlineGDB web interface. The left sidebar contains navigation links: OnlineGDB, code.compile.run.debug.share, IDE, My Projects, Classroom, Learn Programming, Programming Questions, Sign Up, and Login. The main area displays a Python script in a dark-themed editor. The script defines a finite state machine with three states (Q0, Q1, Q2) and transitions based on input 'a' and 'b'. It includes a function `is_accepted` that returns True if the string ends in 'a' or 'ba', and False otherwise. The output console shows the program running successfully with the input 'aab' and the message 'String is accepted.'.

```
1 from enum import Enum
2
3 class State(Enum):
4     Q0 = 0
5     Q1 = 1
6     Q2 = 2
7
8 def is_accepted(input_string):
9     current_state = State.Q0
10
11     for symbol in input_string:
12         if current_state == State.Q0:
13             if symbol == 'a':
14                 current_state = State.Q1
15             else:
16                 return False
17         elif current_state == State.Q1:
18             if symbol == 'a':
19                 current_state = State.Q1 # Self-Loop on 'a'
20             elif symbol == 'b':
21                 current_state = State.Q2
22             else:
23                 return False
24         elif current_state == State.Q2:
25             if symbol == 'b':
26                 current_state = State.Q2 # Self-Loop on 'b'
```

Enter a string: aab
String is accepted.

...Program finished with exit code 0
Press ENTER to exit console.



This is a close-up view of the output console from the previous screenshot. It shows the input 'aab' and the confirmation 'String is accepted.' followed by the program's completion message.

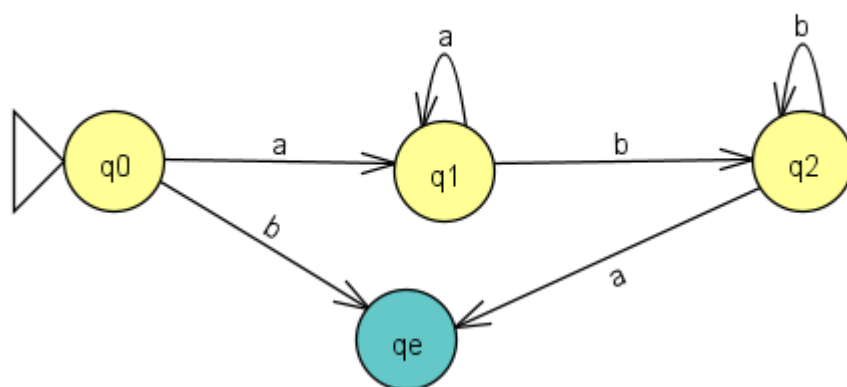
```
Enter a string: aabb
String is accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter a string: abab
String is not accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```

DIAGRAM:



2. Write a program to construct the NFA for the given Strings.

CODE:

```
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':
```

```

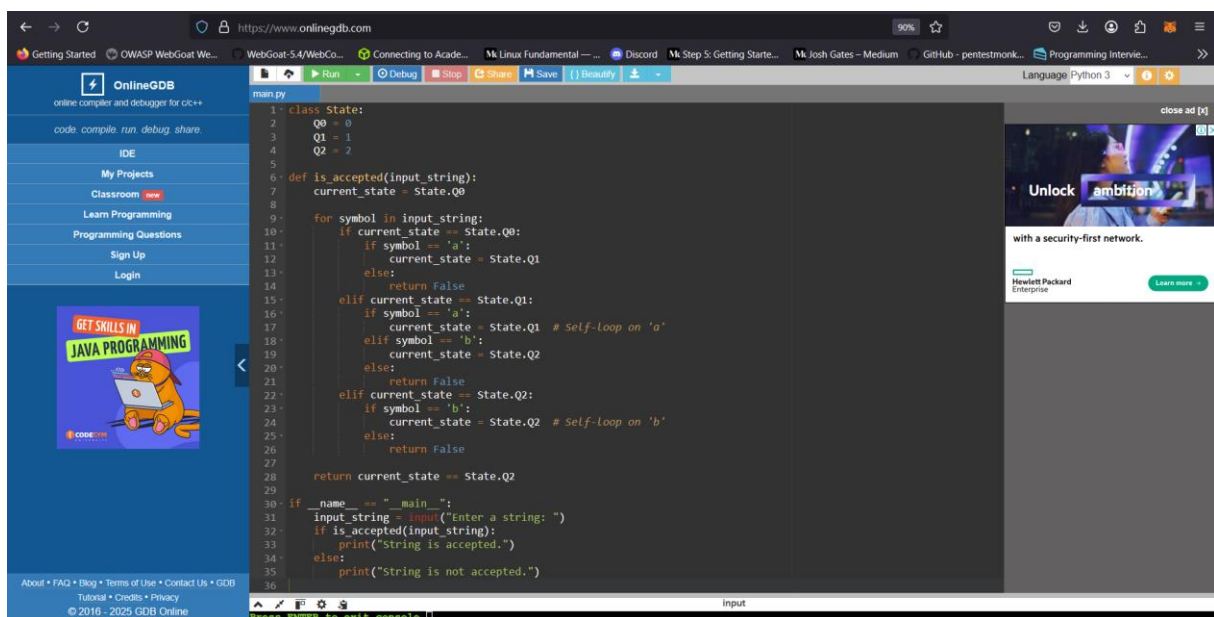
        current_state = State.Q1 # Self-loop on 'a'
    elif symbol == 'b':
        current_state = State.Q2
    else:
        return False
    elif current_state == State.Q2:
        if symbol == 'b':
            current_state = State.Q2 # Self-loop on 'b'
        else:
            return False

    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:



The screenshot shows the OnlineGDB web IDE interface. The code editor displays the Python program from the previous block. The left sidebar contains navigation links like 'OnlineGDB', 'IDE', 'My Projects', 'Classroom', 'Learn Programming', 'Programming Questions', 'Sign Up', and 'Login'. The bottom status bar indicates 'Press ENTER to exit console'. The output console at the bottom shows the result of the program execution.

```

main.py
1 class State:
2     Q0 = 0
3     Q1 = 1
4     Q2 = 2
5
6 def is_accepted(input_string):
7     current_state = State.Q0
8
9     for symbol in input_string:
10        if current_state == State.Q0:
11            if symbol == 'a':
12                current_state = State.Q1
13            else:
14                return False
15        elif current_state == State.Q1:
16            if symbol == 'a':
17                current_state = State.Q1 # Self-loop on 'a'
18            elif symbol == 'b':
19                current_state = State.Q2
20            else:
21                return False
22        elif current_state == State.Q2:
23            if symbol == 'b':
24                current_state = State.Q2 # Self-loop on 'b'
25            else:
26                return False
27
28    return current_state == State.Q2
29
30 if __name__ == "__main__":
31     input_string = input("Enter a string: ")
32     if is_accepted(input_string):
33         print("String is accepted.")
34     else:
35         print("String is not accepted.")
36
Input
Press ENTER to exit console

```

String is accepted.

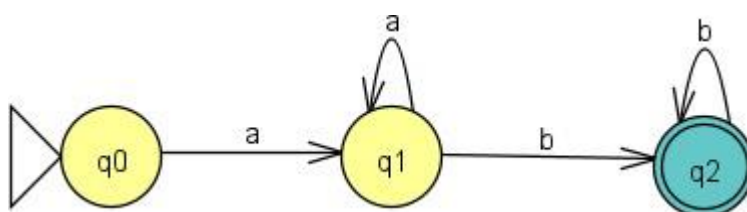
```
Enter a string: abbba
String is not accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Enter a string: aab
String is accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```

DIAGRAM:



3. Write a program for String Acceptance using NFA or DFA.

CODE:

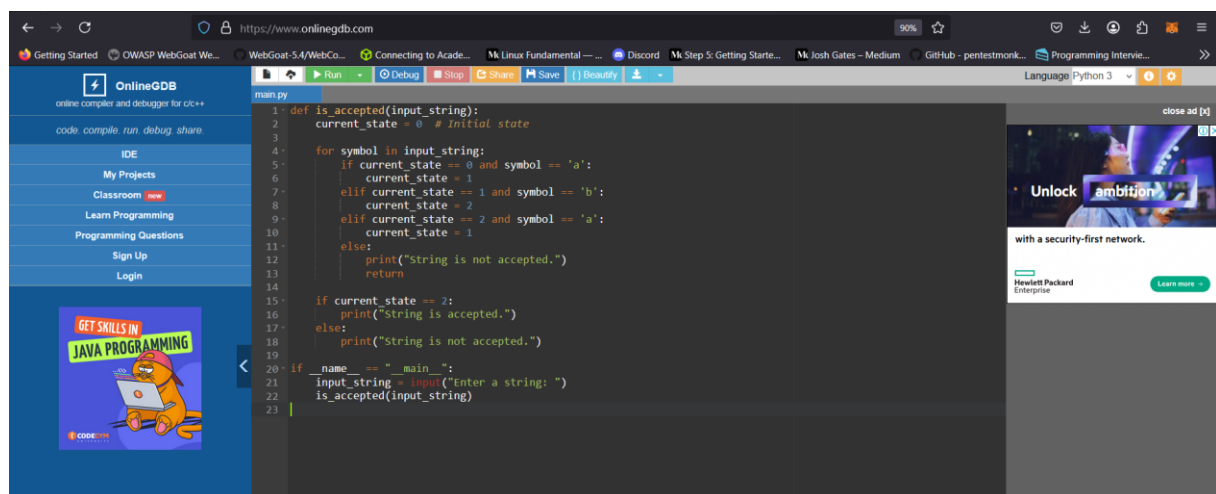
```
def is_accepted(input_string):
    current_state = 0 # Initial state

    for symbol in input_string:
        if current_state == 0 and symbol == 'a':
            current_state = 1
        elif current_state == 1 and symbol == 'b':
            current_state = 2
        elif current_state == 2 and symbol == 'a':
            current_state = 1
        else:
            print("String is not accepted.")
            return

    if current_state == 2:
        print("String is accepted.")
    else:
        print("String is not accepted.")

if __name__ == "__main__":
    input_string = input("Enter a string: ")
    is_accepted(input_string)
```

OUTPUT:

The screenshot shows the OnlineGDB website in a web browser. The browser's address bar displays 'https://www.onlinegdb.com'. The page has a dark theme. On the left, there is a sidebar with navigation links: 'Getting Started', 'OWASP WebGoat We...', 'WebGoat-5.4/WebCo...', 'Connecting to Acade...', 'My Linux Fundamental...', 'Discord', 'My Step 5: Getting Starte...', 'My Josh Gates - Medium', 'GitHub - pentestmonk...', and 'Programming Intervie...'. Below these are buttons for 'code', 'compile', 'run', 'debug', and 'share'. The main area shows a Python script being executed. The script is the same as the one in the previous block. The output of the program is visible in the console area on the right, showing 'String is accepted.' for the input 'abababab'. There is also a sidebar on the right with a 'close ad [x]' button and a 'Learn more' button for 'Newlett Pichard Enterprise'.

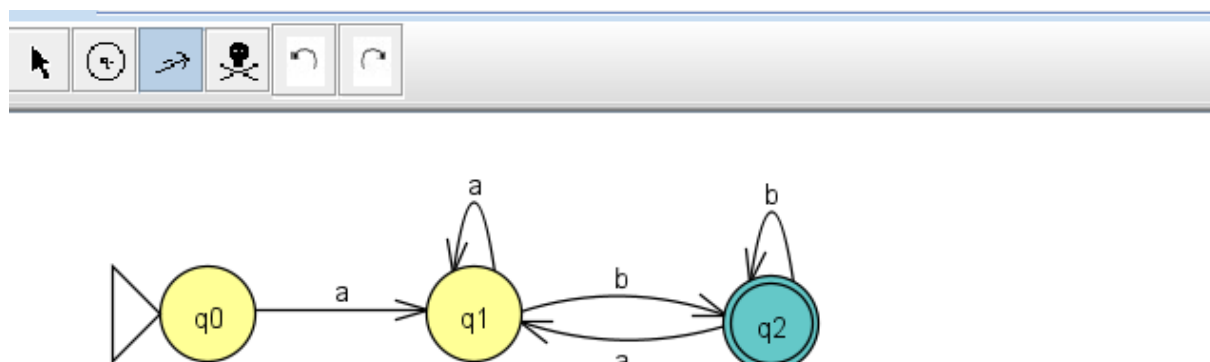
```
Enter a string: abab
String is accepted.

...Program finished with exit code 0
Press ENTER to exit console.

Enter a string: aab
String is not accepted.

...Program finished with exit code 0
Press ENTER to exit console.
```

DIAGRAM:



4. Write a program to Convert the NFA into DFA.

CODE:

```
def fillStates(nfa, state_set, input_symbol):
    next_state_set = set()
    for sub_state in state_set:
        if (sub_state, input_symbol) in nfa:
            next_state_set.update(nfa[(sub_state, input_symbol)])

    return frozenset(next_state_set) # Ensure unique state representation

# Read NFA transitions
nfa = {}
n = int(input("Enter number of states in NFA: "))

for i in range(n):
    for symbol in [0, 1]:
        key = ('q' + str(i), symbol)
        nfa[key] = input(f"Enter transition for state {key}: ").split()

# Print NFA transition diagram
print("\nTransition Diagram for NFA:")
for key, value in nfa.items():
    print(key, ":", value)

# Convert NFA to DFA
dfa = {}
new_states = set()
queue = []

start_state = frozenset(['q0'])
queue.append(start_state)
new_states.add(start_state)

while queue:
    current_state = queue.pop(0)

    dfa[current_state] = {}

    for symbol in [0, 1]:
        next_state = fillStates(nfa, current_state, symbol)
        dfa[current_state][symbol] = next_state

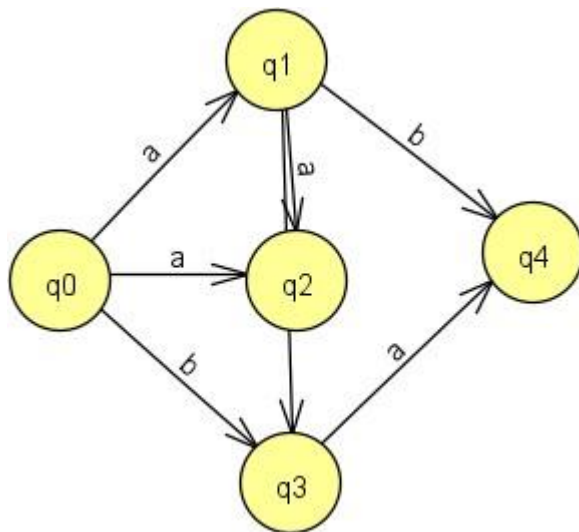
        if next_state and next_state not in new_states:
            new_states.add(next_state)
            queue.append(next_state)

# Print DFA transition diagram (cleaned output)
print("\nTransition Diagram for DFA:")
for state, transitions in dfa.items():
    print(state, ":", transitions)
```

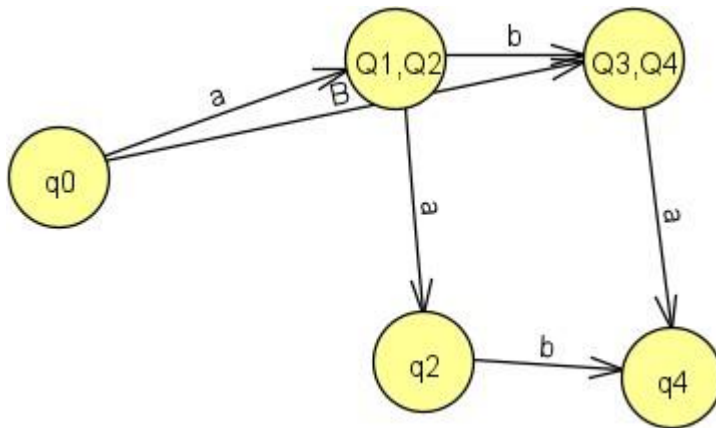
```
state_name = "{" + ", ".join(state) + "}"  
print(f"{state_name} -> { {symbol: '{' + ', '.join(transitions[symbol]) +  
'}' for symbol in transitions} }")
```

DIAGRAM:

STATE DIAGRAM FOR NFA:



DFA STATE TABLE:



OUTPUT:

The screenshot shows the OnlineGDB IDE interface. The main editor contains Python code for converting an NFA to a DFA. The code defines a function `fillStates` to handle state transitions, initializes an NFA, takes user input for the number of states and transitions, and then converts the NFA to a DFA using the subset construction algorithm. The final DFA states are stored in `new_states`.

```
1 def fillStates(dfa, state, input_symbol):
2     l = []
3     for sub_state in state:
4         if (sub_state, input_symbol) in dfa:
5             l.extend(dfa[(sub_state, input_symbol)])
6
7     newEntry = list(set(l)) # Remove duplicates
8     return newEntry if newEntry else []
9
10 # Initialize NFA
11 nfa = {}
12 n = int(input("Enter number of states in NFA: "))
13
14 # Create states
15 states = []
16 for i in range(n):
17     states.append(('q' + str(i), 0))
18     states.append(('q' + str(i), 1))
19
20 # Input transitions
21 for i in states:
22     nfa[i] = list(map(str, input(f"Enter transition for state {i}: ").split()))
23
24 # Print NFA transition diagram
25 print("\nTransition Diagram for NFA:")
26 for i in nfa:
27     print(i, ":", nfa[i])
28
29 # Convert NFA to DFA
30 dfa = {}
31 new_states = set()
32
33 queue = []
34 start_state = ('q0', 0) # Assuming 'q0' is the start state
35 queue.append(frozenset([start_state])) # Using frozenset to represent DFA states
36 new_states.add(frozenset([start_state]))
37
```

```
input
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

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

Transition Diagram for DFA:
{q0} -> {0: '{q1}', 1: '{q2}'}
{q1} -> {0: '{q1}', 1: '{q2}'}
{q2} -> {0: '{q2}', 1: '{q1}'}

...Program finished with exit code 0
Press ENTER to exit console.
```

5. 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 next_state in epsilon_nfa.get((state, 'E'), []):
            add_epsilon_closure(epsilon_nfa, next_state, temp)

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

            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, states, alphabet):
    epsilon_closures = calculate_epsilon_closure(epsilon_nfa, states)
    transitions = calculate_transitions(epsilon_nfa, epsilon_closures, states,
    alphabet)
    return transitions

# Input Section
alphabet = ['0', '1']
epsilon_nfa = {}

n = int(input("Enter number of states in Epsilon-NFA: "))
states = ['q' + str(i) for i in range(n)]

state_transitions = []
for i in range(n):
    state_transitions.append(('q' + str(i), 'E'))
    state_transitions.append(('q' + str(i), '0'))
    state_transitions.append(('q' + str(i), '1'))

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

# Display Epsilon-NFA
print("\nEpsilon-NFA Transition Table:")
for key, value in epsilon_nfa.items():
    print(key, ":", value)

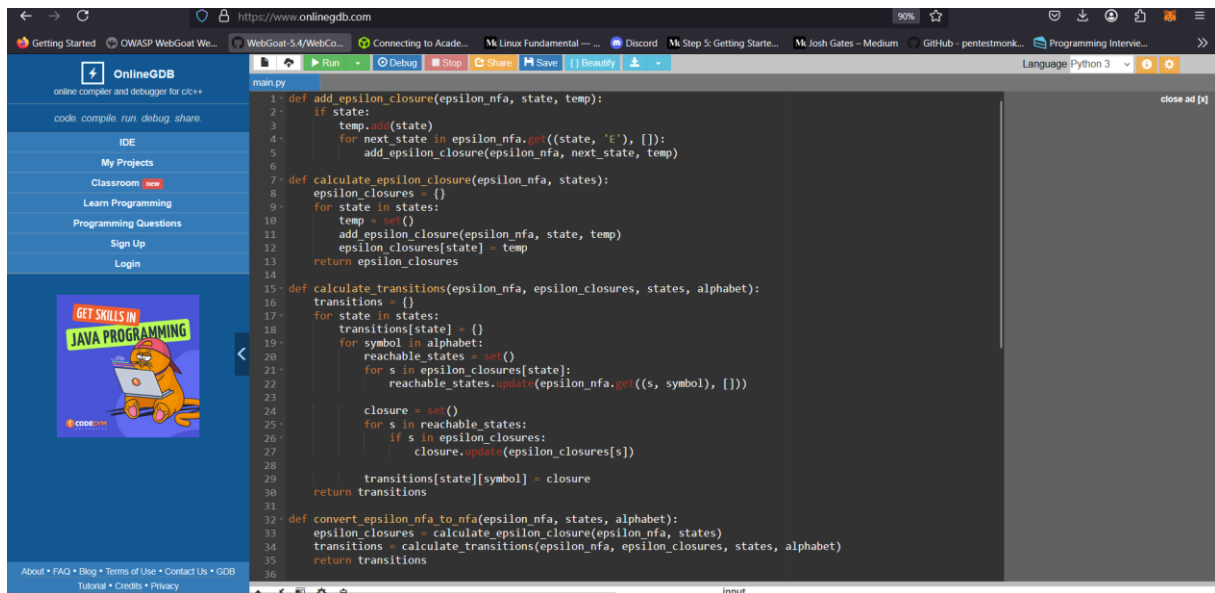
# Convert Epsilon-NFA to NFA
nfa = convert_epsilon_nfa_to_nfa(epsilon_nfa, states, alphabet)

# Display NFA Transition Table
print("\nTransition Table for NFA:")

```

```
for state, trans in nfa.items():
    print(f"State {state}: {trans}")
```

OUTPUT:



The screenshot shows the OnlineGDB web interface. The left sidebar contains navigation links: 'OnlineGDB', 'code compile run debug share', 'IDE', 'My Projects', 'Classroom', 'Learn Programming', 'Programming Questions', 'Sign Up', and 'Login'. Below these is a 'GET SKILLS IN JAVA PROGRAMMING' banner. The main area displays a Python script named 'main.py' with the following code:

```
1 def add_epsilon_closure(epsilon_nfa, state, temp):
2     if state:
3         temp.add(state)
4         for next_state in epsilon_nfa.get((state, 'ε'), []):
5             add_epsilon_closure(epsilon_nfa, next_state, temp)
6
7 def calculate_epsilon_closure(epsilon_nfa, states):
8     epsilon_closures = {}
9     for state in states:
10         temp = set()
11         add_epsilon_closure(epsilon_nfa, state, temp)
12         epsilon_closures[state] = temp
13     return epsilon_closures
14
15 def calculate_transitions(epsilon_nfa, epsilon_closures, states, alphabet):
16     transitions = {}
17     for state in states:
18         transitions[state] = {}
19         for symbol in alphabet:
20             reachable_states = set()
21             for s in epsilon_closures[state]:
22                 reachable_states.update(epsilon_nfa.get((s, symbol), []))
23
24             closure = set()
25             for s in reachable_states:
26                 if s in epsilon_closures:
27                     closure.update(epsilon_closures[s])
28
29             transitions[state][symbol] = closure
30     return transitions
31
32 def convert_epsilon_nfa_to_nfa(epsilon_nfa, states, alphabet):
33     epsilon_closures = calculate_epsilon_closure(epsilon_nfa, states)
34     transitions = calculate_transitions(epsilon_nfa, epsilon_closures, states, alphabet)
35     return transitions
36
```

The bottom of the interface shows a footer with links: 'About', 'FAQ', 'Blog', 'Terms of Use', 'Contact Us', 'GDB Tutorials', 'Credits', and 'Privacy'. The 'Input' field is visible at the bottom right.


```

input
Enter transitions for state ('q0', '0') : q0
Enter transitions for state ('q0', '1') :
Enter transitions for state ('q1', 'E') : q2
Enter transitions for state ('q1', '0') :
Enter transitions for state ('q1', '1') : q1
Enter transitions for state ('q2', 'E') :
Enter transitions for state ('q2', '0') : q0
Enter transitions for state ('q2', '1') : q1

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

Transition Table for NFA:
State q0: {'0': {'q2', 'q0', 'q1'}, '1': {'q2', 'q1'}}
State q1: {'0': {'q2', 'q0', 'q1'}, '1': {'q2', 'q1'}}
State q2: {'0': {'q2', 'q0', 'q1'}, '1': {'q2', 'q1'}}

...Program finished with exit code 0
Press ENTER to exit console.

```

6. Write a program to Convert the Epsilon NFA to DFA

CODE:

```

def add_epsilon_closure(epsilon_nfa, state, temp):
    if state:
        temp.add(state)
        for next_state in epsilon_nfa.get((state, 'E'), []):
            add_epsilon_closure(epsilon_nfa, next_state, temp)

def calculate_epsilon_closure(epsilon_nfa, states):
    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, states, alphabet):
    transitions = {}
    for state in states:
        transitions[state] = {}
        for symbol in alphabet:
            reachable_states = set()

```

```

        for s in epsilon_closures[state]:
            reachable_states.update(epsilon_nfa.get((s, symbol), []))

        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, states, alphabet):
    epsilon_closures = calculate_epsilon_closure(epsilon_nfa, states)
    transitions = calculate_transitions(epsilon_nfa, epsilon_closures, states,
    alphabet)
    return transitions

def fillStates(dfa, state, input_symbol):
    new_entry = set()
    for i in dfa[state]:
        if (i, input_symbol) in dfa:
            new_entry.update(dfa[(i, input_symbol)])
    return new_entry

def convert_to_tuple_dict(input_dict):
    result = {}
    for state, transitions in input_dict.items():
        for symbol, targets in transitions.items():
            for target in targets:
                if (state, symbol) in result:
                    result[(state, symbol)].add(target)
                else:
                    result[(state, symbol)] = {target}
    return result

# Input Section
epsilon_nfa = {}
n = int(input("Enter number of states in Epsilon-NFA: "))
states = ['q' + str(i) for i in range(n)]
alphabet = ['0', '1']

state_transitions = []
for i in range(n):
    state_transitions.append(('q' + str(i), 'E'))
    state_transitions.append(('q' + str(i), '0'))
    state_transitions.append(('q' + str(i), '1'))

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

```

```

# Display Epsilon-NFA
print("\nEpsilon-NFA Transition Table:")
for key, value in epsilon_nfa.items():
    print(key, ":", value)

# Convert Epsilon-NFA to NFA
nfa = convert_epsilon_nfa_to_nfa(epsilon_nfa, states, alphabet)

# Display NFA Transition Table
print("\nTransition Table for NFA:")
for state, trans in nfa.items():
    print(f"State {state}: {trans}")

nfa = convert_to_tuple_dict(nfa)
dfa = nfa

# Deterministic Finite Automaton (DFA) Construction
dfa_states = set()
for i in dfa.keys():
    dfa_states.add(i[0])

new_dfa = {}
for i in dfa:
    new_state = ''.join(dfa[i])
    if new_state not in dfa_states and new_state:
        dfa_states.add(new_state)
    new_dfa[(new_state, '0')] = fillStates(dfa, i, '0')
    new_dfa[(new_state, '1')] = fillStates(dfa, i, '1')

for i in new_dfa:
    dfa[i] = new_dfa[i]

# Display DFA Transition Table
print("\nTransition Table for DFA:")
for state, transitions in dfa.items():
    if isinstance(transitions, dict):
        for symbol, targets in transitions.items():
            target_string = ''.join(targets)
            print(f"({state[0]}, '{symbol}') : {target_string}")
    else:
        target_string = ''.join(transitions)
        print(f"({state[0]}, '{state[1]}') : {target_string}")

```

OUTPUT:

```
input
Enter number of states in Epsilon-NFA: 3
Enter transitions for state ('q0', 'E') : q1
Enter transitions for state ('q0', '0') :
Enter transitions for state ('q0', '1') : q0
Enter transitions for state ('q1', 'E') : q2
Enter transitions for state ('q1', '0') : q1
Enter transitions for state ('q1', '1') :
Enter transitions for state ('q2', 'E') :
Enter transitions for state ('q2', '0') : q1
Enter transitions for state ('q2', '1') : q0

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

Transition Table for NFA:
State q0: {'0': {'q2', 'q1'}, '1': {'q2', 'q1', 'q0'}}
State q1: {'0': {'q2', 'q1'}, '1': {'q2', 'q1', 'q0'}}
State q2: {'0': {'q2', 'q1'}, '1': {'q2', 'q1', 'q0'}}

Transition Table for DFA:
(q0, '0') : q2q1
(q0, '1') : q2q1q0
(q1, '0') : q2q1
(q1, '1') : q2q1q0
(q2, '0') : q2q1
(q2, '1') : q2q1q0
(q2q1, '0') : q2q1
(q2q1, '1') : q2q1q0
(q2q1q0, '0') : q2q1
(q2q1q0, '1') : q2q1q0
```

7. Write a Python program to construct a Pushdown Automaton (PDA) for the following Language

$L = \{a^n, b^n \mid n \geq 1\}$

CODE:

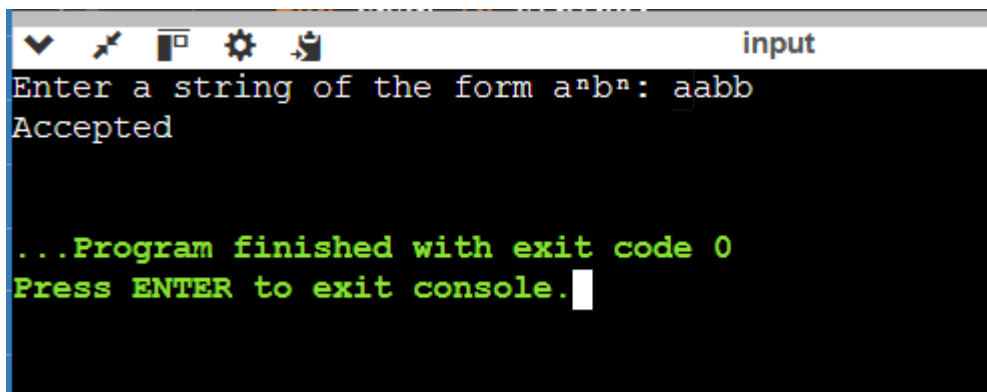
```
class PDA:
    def __init__(self):
        self.stack = []

    def process(self, string):
        for char in string:
            if char == 'a':
                self.stack.append('a') # Push 'a' onto the stack
            elif char == 'b':
                if not self.stack or self.stack[-1] != 'a':
                    return False # Invalid sequence (more 'b's than 'a's)
                self.stack.pop() # Pop 'a' for each 'b'

        return len(self.stack) == 0 # Accept if stack is empty

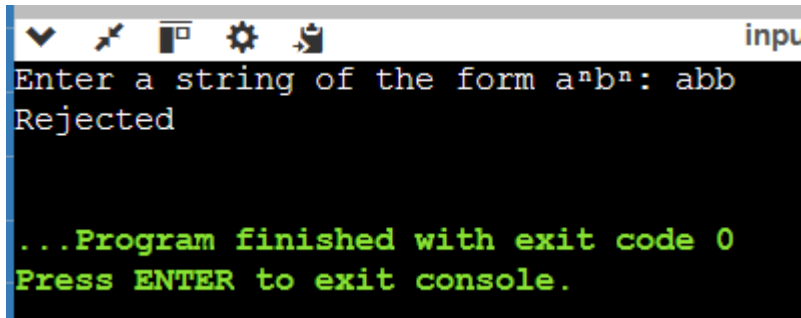
# Input and Execution
pda = PDA()
input_string = input("Enter a string of the form a^n b^n: ")
if pda.process(input_string):
    print("Accepted")
else:
    print("Rejected")
```

OUTPUT:



```
input
Enter a string of the form a^n b^n: aabb
Accepted

...Program finished with exit code 0
Press ENTER to exit console.
```



```
input
Enter a string of the form anbn: abb
Rejected

...Program finished with exit code 0
Press ENTER to exit console.
```

8. Write a Python program to construct a Pushdown Automaton (PDA) for the following Language

$L = \{ww^r \mid w = (a+b)^+\}$

CODE:

```
class PDA:
    def __init__(self):
        self.stack = []

    def process(self, string):
        n = len(string)

        if n % 2 != 0:
            return False # Length must be even for wwr

        half = n // 2 # Find the midpoint

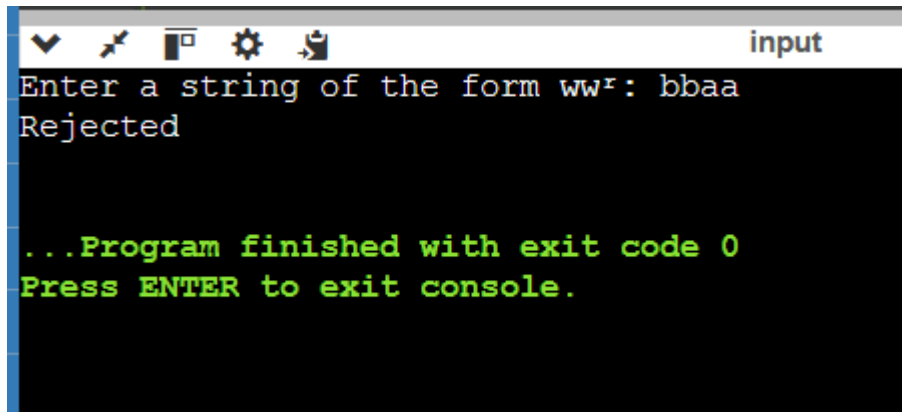
        # Push the first half of the string onto the stack
        for i in range(half):
            self.stack.append(string[i])

        # Check if the second half is the reverse of the first half
        for i in range(half, n):
            if not self.stack or self.stack[-1] != string[i]:
                return False # Mismatch found
            self.stack.pop() # Pop from stack

        return len(self.stack) == 0 # Accept if stack is empty

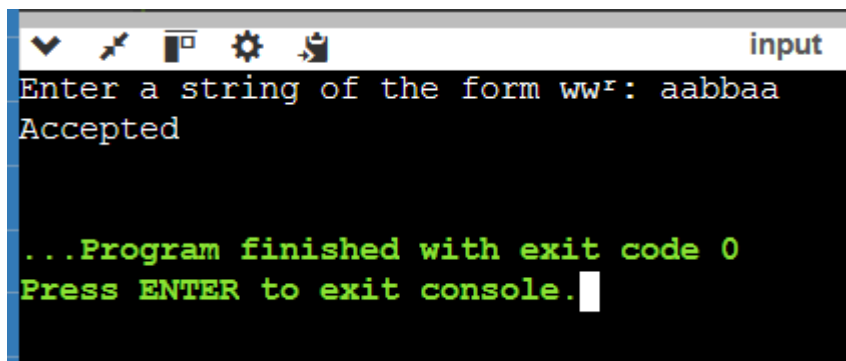
# Input and Execution
pda = PDA()
input_string = input("Enter a string of the form wwr: ")
if pda.process(input_string):
    print("Accepted")
else:
    print("Rejected")
```

OUTPUT:



```
input
Enter a string of the form wwr: bbaa
Rejected

...Program finished with exit code 0
Press ENTER to exit console.
```



```
input
Enter a string of the form wwr: aabbbaa
Accepted

...Program finished with exit code 0
Press ENTER to exit console.
```

9. Write a Python program to construct a Pushdown Automaton (PDA) for the following Language

$L = \{wCw^r \mid w \text{ belongs to } (a+b)^*\}$

CODE:

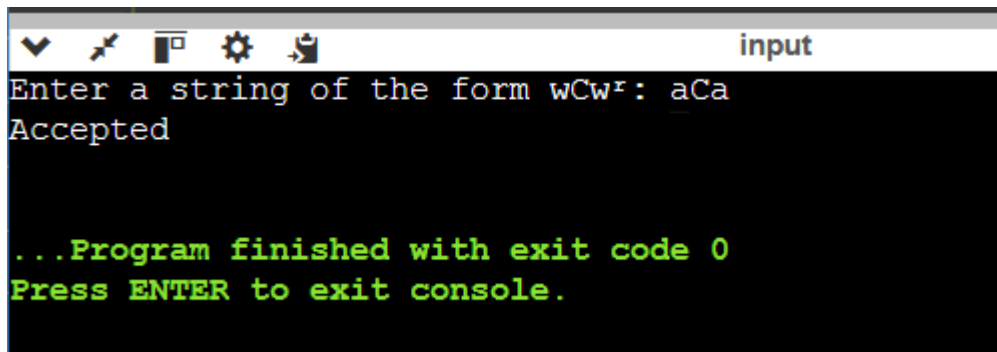
```
class PDA:
    def __init__(self):
        self.stack = []
        self.found_C = False # Track if 'C' has been encountered

    def process(self, string):
        for char in string:
            if char != 'C' and not self.found_C:
                self.stack.append(char) # Push w onto stack
            elif char == 'C':
                if self.found_C: # Multiple 'C' are not allowed
                    return False
                self.found_C = True
            elif self.found_C: # Matching w^r with stack
                if not self.stack or self.stack[-1] != char:
                    return False # Reverse check failed
                self.stack.pop()

        return self.found_C and len(self.stack) == 0 # Accept if stack is
empty and 'C' was found

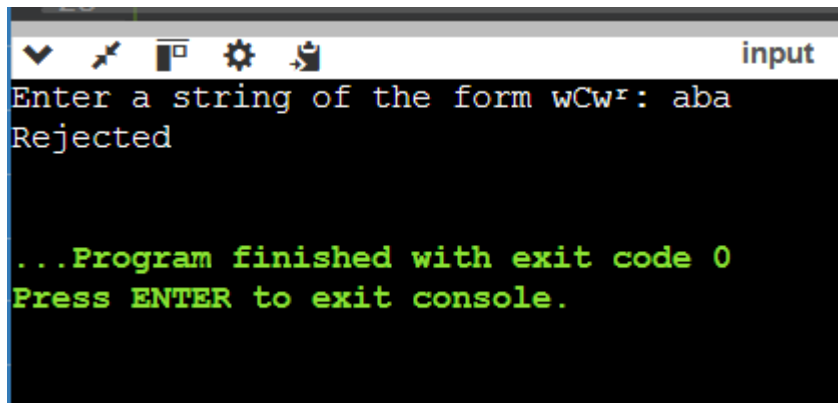
# Input and Execution
pda = PDA()
input_string = input("Enter a string of the form wCw^r: ")
if pda.process(input_string):
    print("Accepted")
else:
    print("Rejected")
```

OUTPUT:



```
input
Enter a string of the form wCw^r: aCa
Accepted

...Program finished with exit code 0
Press ENTER to exit console.
```

```
input
Enter a string of the form wCw^r: aba
Rejected

...Program finished with exit code 0
Press ENTER to exit console.
```

10. Write a Python program to construct a Pushdown Automaton (PDA) for the following Language
 $L = \{0^n 1^m 2^m 3^n \mid n, m \geq 1\}$

CODE:

```
class PDA:
    def __init__(self):
        self.stack = []

    def process(self, string):
        n = string.count('0')
        m = string.count('1')

        # Condition to check the count match
        if string.count('3') != n or string.count('2') != m:
            return False # Count mismatch

        step = 0 # Track transitions
        for char in string:
            if char == '0':
                self.stack.append('0')
                step = 1
            elif char == '1':
                self.stack.append('1')
                step = 2
            elif char == '2':
                if not self.stack or self.stack[-1] != '1':
                    return False # '1' must match with '2'
                self.stack.pop()
                step = 3
            elif char == '3':
                if not self.stack or self.stack[-1] != '0':
                    return False # '0' must match with '3'
                self.stack.pop()
                step = 4
            else:
                return False # Invalid character

        return len(self.stack) == 0 # Stack should be empty if valid

# Input and Execution
pda = PDA()
input_string = input("Enter a string of the form 0^n1^m2^m3^n: ")
if pda.process(input_string):
    print("Accepted")
else:
    print("Rejected")
```

OUTPUT:

```
input
Enter a string of the form 0^n1^m2^m3^n: 0112333
Rejected

...Program finished with exit code 0
Press ENTER to exit console.
```

```
input
Enter a string of the form 0^n1^m2^m3^n: 00112233
Accepted

...Program finished with exit code 0
Press ENTER to exit console.
```

11. Write a Python program to construct a Pushdown Automaton (PDA) for the following Language

$L = \{a^n b^{2n} \mid n \geq 1\}$

CODE:

```
class PDA:
    def __init__(self):
        self.stack = []
        self.b_count = 0 # Count of 'b's encountered

    def process(self, string):
        n = 0 # Count of 'a's

        for char in string:
            if char == 'a':
                self.stack.append('a') # Push 'a' onto the stack
                n += 1 # Track count of 'a's

            elif char == 'b':
                self.b_count += 1 # Count each 'b'

                if self.stack:
                    self.stack.pop() # Match the first batch of 'b's with
'a's

                else:
                    pass # Second batch of 'b's (after stack is empty)
```

```

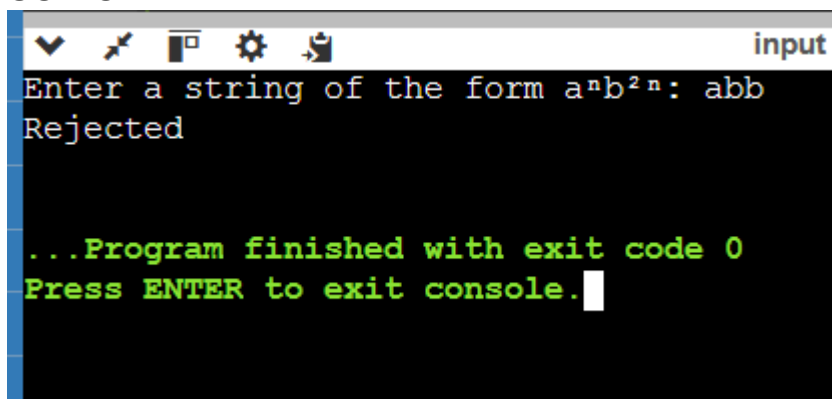
        else:
            return False # Reject if an invalid character is found

        # Condition for acceptance:
        return self.b_count == 2 * n and not self.stack # 2n 'b's and stack
must be empty

# Input and Execution
pda = PDA()
input_string = input("Enter a string of the form anb2n: ")
if pda.process(input_string):
    print("Accepted")
else:
    print("Rejected")

```

OUTPUT:



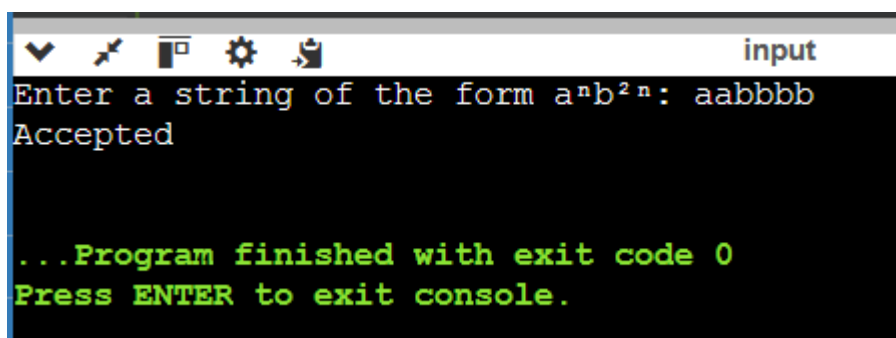
A terminal window titled 'input' with a dark background and light green text. The prompt 'Enter a string of the form aⁿb²ⁿ: ' is followed by the input 'abb'. The output is 'Rejected'. At the bottom, it says '...Program finished with exit code 0' and 'Press ENTER to exit console.' with a cursor.

```

input
Enter a string of the form anb2n: abb
Rejected

...Program finished with exit code 0
Press ENTER to exit console.

```



A terminal window titled 'input' with a dark background and light green text. The prompt 'Enter a string of the form aⁿb²ⁿ: ' is followed by the input 'aabbbb'. The output is 'Accepted'. At the bottom, it says '...Program finished with exit code 0' and 'Press ENTER to exit console.' with a cursor.

```

input
Enter a string of the form anb2n: aabbbb
Accepted

...Program finished with exit code 0
Press ENTER to exit console.

```

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

CODE:

```

from collections import defaultdict

def compute_first(symbol):
    if symbol in first_sets: # If already computed, return it
        return first_sets[symbol]

```

```

first = set()

if not symbol.isupper(): # Terminal case
    first.add(symbol)
else:
    for production in grammar[symbol]:
        for char in production:
            first_char = compute_first(char)
            first.update(first_char - {'ε'}) # Add FIRST(char) except ε

            if 'ε' not in first_char:
                break # Stop if epsilon is not in FIRST(char)
        else:
            first.add('ε') # If all have ε, add it to FIRST(symbol)

first_sets[symbol] = first
return first

def compute_follow(symbol):
    if symbol in follow_sets: # If already computed, return it
        return follow_sets[symbol]

    follow = set()

    if symbol == start_symbol:
        follow.add('$') # Add '$' for the start symbol

    for lhs, productions in grammar.items():
        for production in productions:
            if symbol in production:
                index = production.index(symbol)

                while index < len(production) - 1:
                    next_symbol = production[index + 1]
                    first_next = compute_first(next_symbol)

                    follow.update(first_next - {'ε'})

                    if 'ε' not in first_next:
                        break
                    index += 1
                else:
                    if lhs != symbol:
                        follow.update(compute_follow(lhs))

    follow_sets[symbol] = follow
    return follow

```

```

# Input Grammar
grammar = defaultdict(list)
first_sets = {}
follow_sets = {}

num_productions = int(input("Enter the number of productions: "))

for _ in range(num_productions):
    lhs, rhs = input("Enter production (A=abc format): ").split("=")
    grammar[lhs].extend(rhs.split("|")) # Handle multiple productions

start_symbol = list(grammar.keys())[0] # Assuming first entered symbol as start

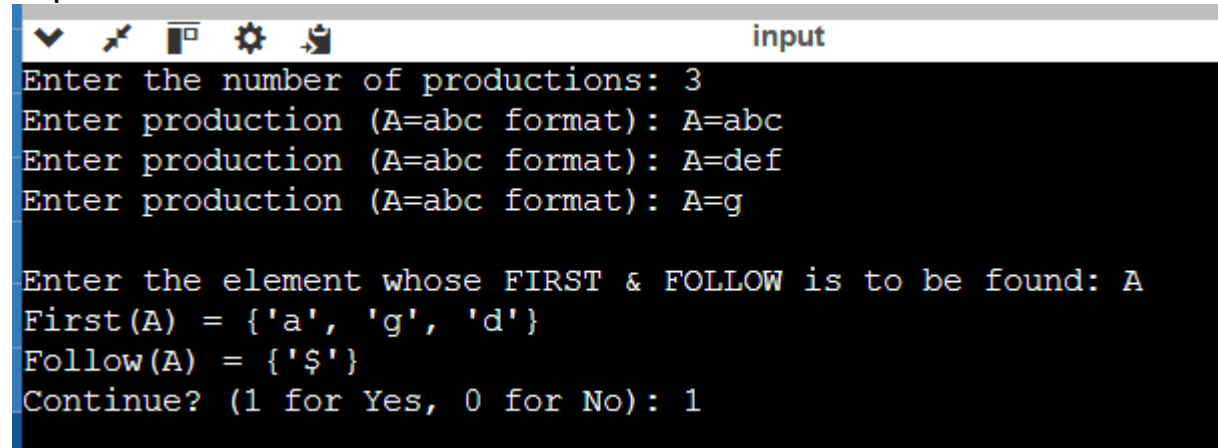
# Compute FIRST and FOLLOW sets
for non_terminal in grammar:
    compute_first(non_terminal)
for non_terminal in grammar:
    compute_follow(non_terminal)

# Querying FIRST and FOLLOW sets
while True:
    query = input("\nEnter the element whose FIRST & FOLLOW is to be found: ")
    if query not in grammar:
        print("Invalid non-terminal.")
    else:
        print(f"First({query}) = {first_sets[query]}")
        print(f"Follow({query}) = {follow_sets[query]}")

    cont = input("Continue? (1 for Yes, 0 for No): ")
    if cont == "0":
        break

```

Output:



```

input
Enter the number of productions: 3
Enter production (A=abc format): A=abc
Enter production (A=abc format): A=def
Enter production (A=abc format): A=g

Enter the element whose FIRST & FOLLOW is to be found: A
First(A) = {'a', 'g', 'd'}
Follow(A) = {'$'}
Continue? (1 for Yes, 0 for No): 1

```

13. Generating Three Address Code (TAC) for a Given Expression

Code:

```
def generate_tac(expression):
    tokens = expression.replace(" ", "").split("=")
    if len(tokens) != 2:
        print("Invalid expression format!")
        return []

    var = tokens[0] # Left-hand side variable
    expr = tokens[1] # Right-hand side expression
    operands = expr.split("+") # Splitting the expression based on '+'

    if len(operands) < 2:
        print("Expression must contain at least one '+' operation!")
        return []

    temp_count = 1
    tac_code = []

    # First operation
    temp_var = f"t{temp_count}"
    tac_code.append(f"{temp_var} = {operands[0]} + {operands[1]}")

    # Processing remaining operands
    for i in range(2, len(operands)):
        temp_count += 1
        new_temp_var = f"t{temp_count}"
        tac_code.append(f"{new_temp_var} = {temp_var} + {operands[i]}")
        temp_var = new_temp_var

    # Final assignment
    tac_code.append(f"{var} = {temp_var}")

    return tac_code

# Taking input from the user
expression = input("Enter an arithmetic expression (e.g., x = a + b + c + d): ")

# Generating and printing TAC
tac_output = generate_tac(expression)

if tac_output:
    print("\nGenerated Three-Address Code (TAC):")
    for line in tac_output:
        print(line)
```

Output:



```
input
Enter an arithmetic expression (e.g., x = a + b + c + d): x = a + b + c + d

Generated Three-Address Code (TAC):
t1 = a + b
t2 = t1 + c
t3 = t2 + d
x = t3

...Program finished with exit code 0
Press ENTER to exit console.
```