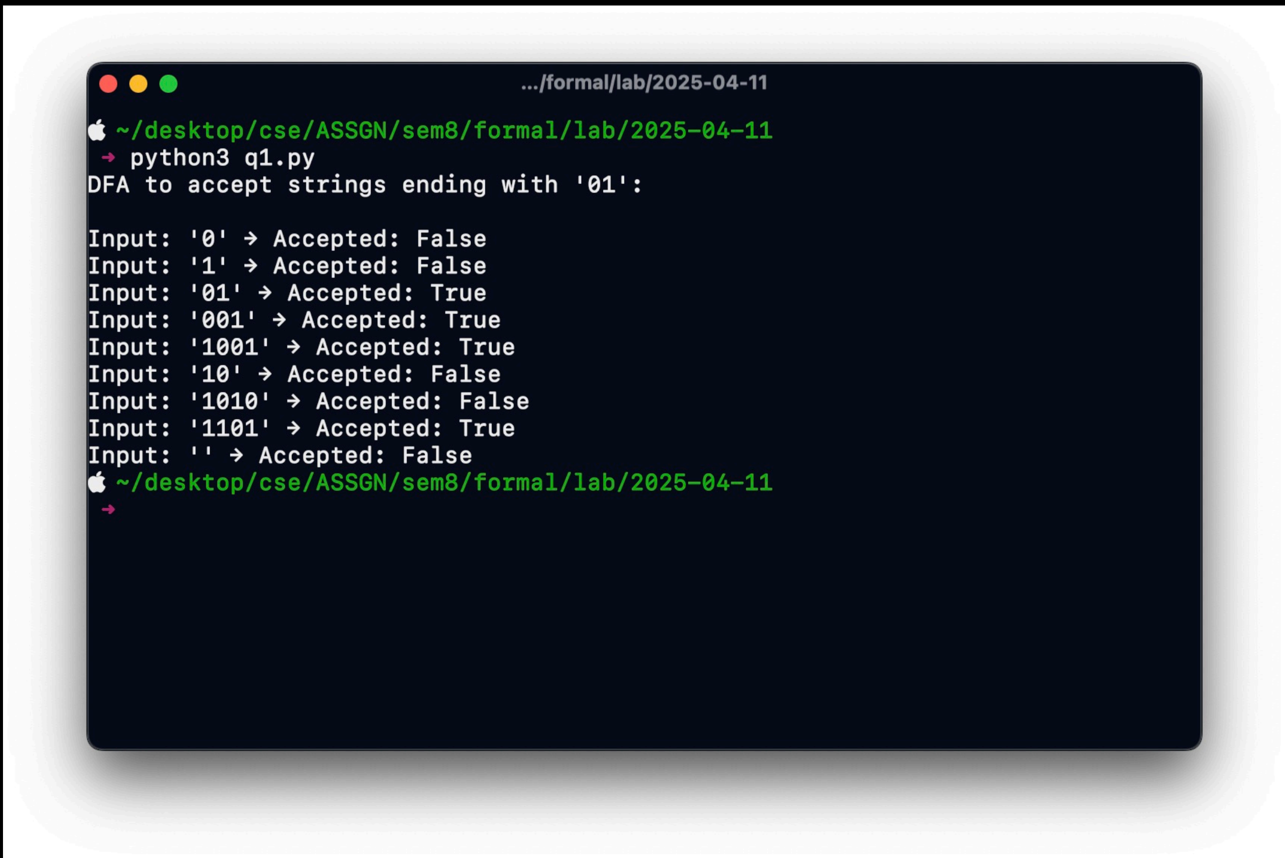
ASSIGNMENT 8
11/04/25

NAME : SHRESTH SONKAR
REGNO : 20214272
GROUP : CS8D
TOPIC : FORMAL METHOD LAB
CODE : CS-18201

```python
# Q1 Implement a Deterministic Finite Automaton (DFA)
in Python and verify its language acceptance
properties.

class DFA:
    def __init__(self, states, alphabet,
transition_function, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states

    def accepts(self, input_string):
        current_state = self.start_state
        for symbol in input_string:
            if symbol not in self.alphabet:
                print(f"Invalid symbol: {symbol}")
                return False
            current_state =
self.transition_function.get((current_state, symbol))
            if current_state is None:
                return False
        return current_state in self.accept_states


if __name__ == "__main__":
    states = {'q0', 'q1', 'q2'}
    alphabet = {'0', '1'}
    start_state = 'q0'
    accept_states = {'q2'}

    transition_function = {
        ('q0', '0'): 'q1',
        ('q0', '1'): 'q0',
        ('q1', '0'): 'q1',
        ('q1', '1'): 'q2',
        ('q2', '0'): 'q1',
        ('q2', '1'): 'q0'
    }

    dfa = DFA(states, alphabet, transition_function,
start_state, accept_states)
    test_strings = ['0', '1', '01', '001', '1001',
'10', '1010', '1101', '']
```

```
    print("DFA to accept strings ending with '01':\n")
    for s in test_strings:
        result = dfa.accepts(s)
        print(f"Input: '{s}' → Accepted: {result}")
```




```
# Q2 Develop a simulation tool for Nondeterministic
Finite Automata (NFA) and check equivalence with a DFA.

from collections import defaultdict
from itertools import product

EPSILON = 'ε'

class NFA:
    def __init__(self, states, alphabet,
transition_function, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states
```

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

        while stack:
            state = stack.pop()
            for next_state in
self.transition_function.get((state, EPSILON), []):
                if next_state not in closure:
                    closure.add(next_state)
                    stack.append(next_state)
        return closure

    def move(self, state_set, symbol):
        result = set()
        for state in state_set:

result.update(self.transition_function.get((state,
symbol), []))
        return result

    def accepts(self, input_string):
        current_states =
self.epsilon_closure({self.start_state})
        for symbol in input_string:
            current_states =
self.epsilon_closure(self.move(current_states, symbol))
        return any(state in self.accept_states for
state in current_states)


class DFA:
    def __init__(self, states, alphabet,
transition_function, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transition_function = transition_function
        self.start_state = start_state
        self.accept_states = accept_states

    def accepts(self, input_string):
        state = self.start_state
        for symbol in input_string:
            state =
self.transition_function.get((state, symbol))
```

```
            if state is None:
                return False
        return state in self.accept_states


def nfa_to_dfa(nfa):
    state_map = {}
    dfa_states = set()
    dfa_start =
frozenset(nfa.epsilon_closure({nfa.start_state}))
    unmarked_states = [dfa_start]
    dfa_trans = {}
    dfa_accepts = set()

    while unmarked_states:
        current = unmarked_states.pop()
        if current not in dfa_states:
            dfa_states.add(current)
            if any(state in nfa.accept_states for state
in current):
                dfa_accepts.add(current)

            for symbol in nfa.alphabet:
                move_result = nfa.move(current, symbol)
                closure =
frozenset(nfa.epsilon_closure(move_result))
                if closure:
                    dfa_trans[(current, symbol)] =
closure
                    if closure not in dfa_states and
closure not in unmarked_states:
                        unmarked_states.append(closure)

    return DFA(
        states=dfa_states,
        alphabet=nfa.alphabet,
        transition_function=dfa_trans,
        start_state=dfa_start,
        accept_states=dfa_accepts
    )


def generate_all_strings(alphabet, max_length):
    result = set()
    for l in range(max_length + 1):
```

```python
        for p in product(alphabet, repeat=l):
            result.add(''.join(p))
    return result


def check_equivalence(nfa, dfa, test_depth=4):
    test_set = generate_all_strings(nfa.alphabet,
test_depth)
    for test_str in test_set:
        if nfa.accepts(test_str) !=
dfa.accepts(test_str):
            print(f"Mismatch found for input:
'{test_str}'")
            return False
    return True


if __name__ == "__main__":
    states = {'q0', 'q1', 'q2'}
    alphabet = {'0', '1'}
    transition_function = {
        ('q0', '0'): {'q0', 'q1'},
        ('q0', '1'): {'q0'},
        ('q1', '1'): {'q2'},
    }
    start_state = 'q0'
    accept_states = {'q2'}

    nfa = NFA(states, alphabet, transition_function,
start_state, accept_states)
    dfa = nfa_to_dfa(nfa)

    print("Testing equivalence of NFA and converted
DFA...")
    equivalent = check_equivalence(nfa, dfa)

    if equivalent:
        print("✅ NFA and DFA are equivalent (within
test depth).")
    else:
        print("❌ NFA and DFA are NOT equivalent.")
```


```python
# Q3 Write a Python-based tool to transform a regular
expression into an equivalent automaton.

from collections import defaultdict
from itertools import count

EPSILON = 'ε'

class State:
    def __init__(self):
        self.transitions = defaultdict(list)


class Fragment:
    def __init__(self, start, out_states):
        self.start = start
        self.out_states = out_states


class NFA:
    def __init__(self, start, accept):
        self.start = start
```

```
        self.accept = accept
        self.states = set()
        self._collect_states(start)

    def _collect_states(self, state):
        if state in self.states:
            return
        self.states.add(state)
        for targets in state.transitions.values():
            for t in targets:
                self._collect_states(t)

    def epsilon_closure(self, states):
        stack = list(states)
        closure = set(states)
        while stack:
            state = stack.pop()
            for next_state in
state.transitions.get(EPSILON, []):
                if next_state not in closure:
                    closure.add(next_state)
                    stack.append(next_state)
        return closure

    def move(self, states, symbol):
        result = set()
        for state in states:
            result.update(state.transitions.get(symbol,
[]))
        return result

    def accepts(self, input_string):
        current_states =
self.epsilon_closure({self.start})
        for symbol in input_string:
            current_states =
self.epsilon_closure(self.move(current_states, symbol))
        return self.accept in current_states


class RegexToNFA:
    def __init__(self):
        self.state_id = count()

    def new_state(self):
```

```
        return State()

    def re_to_nfa(self, regex):
        postfix = self.infix_to_postfix(regex)
        stack = []

        for token in postfix:
            if token == '*':
                frag = stack.pop()
                start = self.new_state()
                accept = self.new_state()

start.transitions[EPSILON].extend([frag.start, accept])
                for out in frag.out_states:

out.transitions[EPSILON].extend([frag.start, accept])
                stack.append(Fragment(start, [accept]))
            elif token == '.':
                frag2 = stack.pop()
                frag1 = stack.pop()
                for out in frag1.out_states:

out.transitions[EPSILON].append(frag2.start)
                stack.append(Fragment(frag1.start,
frag2.out_states))
            elif token == '|':
                frag2 = stack.pop()
                frag1 = stack.pop()
                start = self.new_state()
                accept = self.new_state()

start.transitions[EPSILON].extend([frag1.start,
frag2.start])
                for out in frag1.out_states +
frag2.out_states:

out.transitions[EPSILON].append(accept)
                stack.append(Fragment(start, [accept]))
            else:
                start = self.new_state()
                accept = self.new_state()
                start.transitions[token].append(accept)
                stack.append(Fragment(start, [accept]))

        final_frag = stack.pop()
```

```
        return NFA(final_frag.start,
final_frag.out_states[0])

    def infix_to_postfix(self, regex):
        precedence = {'*': 3, '.': 2, '|': 1}
        output = []
        stack = []

        new_regex = []
        prev = None
        for c in regex:
            if prev and (prev.isalnum() or prev == ')'
or prev == '*') and (c.isalnum() or c == '('):
                new_regex.append('.')
            new_regex.append(c)
            prev = c

        for c in new_regex:
            if c.isalnum():
                output.append(c)
            elif c == '(':
                stack.append(c)
            elif c == ')':
                while stack and stack[-1] != '(':
                    output.append(stack.pop())
                stack.pop()
            else:
                while stack and stack[-1] != '(' and
precedence[c] <= precedence[stack[-1]]:
                    output.append(stack.pop())
                stack.append(c)

        while stack:
            output.append(stack.pop())
        return output


if __name__ == "__main__":
    converter = RegexToNFA()
    regex = "(a|b)*abb"
    nfa = converter.re_to_nfa(regex)
    test_strings = ["abb", "aabb", "abababb", "ab",
"bba", "", "abbbb"]

    print(f"Testing regex: {regex}\n")
```

```
    for s in test_strings:
        result = nfa.accepts(s)
        print(f"Input: '{s}' → Accepted: {result}")
```


```
# Q4 Model and analyze a simple text parser using
formal grammar and automata theory.

import re

class Token:
    def __init__(self, type_, value):
        self.type = type_
        self.value = value

    def __repr__(self):
        return f"{self.type}({self.value})"


class Lexer:
    def __init__(self, input_text):
        self.input_text = input_text
        self.tokens = []
        self.tokenize()
```

```
    def tokenize(self):
        token_spec = [
            ('NUMBER', r'\d+'),
            ('PLUS', r'\+'),
            ('MINUS', r'-'),
            ('MULT', r'\*'),
            ('DIV', r'/'),
            ('LPAREN', r'\('),
            ('RPAREN', r'\)'),
            ('SKIP', r'[ \t]+'),
            ('MISMATCH', r'.'),
        ]

        tok_regex = '|'.join(f'(?P<{name}>{regex})' for
name, regex in token_spec)
        for mo in re.finditer(tok_regex,
self.input_text):
            kind = mo.lastgroup
            value = mo.group()
            if kind == 'NUMBER':
                self.tokens.append(Token('NUMBER',
int(value)))
            elif kind in ('PLUS', 'MINUS', 'MULT',
'DIV', 'LPAREN', 'RPAREN'):
                self.tokens.append(Token(kind, value))
            elif kind == 'SKIP':
                continue
            else:
                raise SyntaxError(f"Unexpected
character: {value}")
        self.tokens.append(Token('EOF', None))


class Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0

    def current(self):
        return self.tokens[self.pos]

    def eat(self, type_):
        if self.current().type == type_:
            self.pos += 1
```

```
        else:
            raise SyntaxError(f"Expected {type_}, got
{self.current()}")

    def parse(self):
        result = self.E()
        if self.current().type != 'EOF':
            raise SyntaxError("Unexpected input after
complete parsing.")
        return result

    def E(self):
        node = self.T()
        return self.E_prime(node)

    def E_prime(self, left):
        tok = self.current()
        if tok.type == 'PLUS':
            self.eat('PLUS')
            right = self.T()
            return self.E_prime(('Add', left, right))
        elif tok.type == 'MINUS':
            self.eat('MINUS')
            right = self.T()
            return self.E_prime(('Sub', left, right))
        return left  # ε

    def T(self):
        node = self.F()
        return self.T_prime(node)

    def T_prime(self, left):
        tok = self.current()
        if tok.type == 'MULT':
            self.eat('MULT')
            right = self.F()
            return self.T_prime(('Mul', left, right))
        elif tok.type == 'DIV':
            self.eat('DIV')
            right = self.F()
            return self.T_prime(('Div', left, right))
        return left  # ε

    def F(self):
        tok = self.current()
```

```python
        if tok.type == 'NUMBER':
            self.eat('NUMBER')
            return ('Num', tok.value)
        elif tok.type == 'LPAREN':
            self.eat('LPAREN')
            node = self.E()
            self.eat('RPAREN')
            return node
        else:
            raise SyntaxError(f"Unexpected token:
{tok}")


def evaluate(ast):
    if ast[0] == 'Num':
        return ast[1]
    elif ast[0] == 'Add':
        return evaluate(ast[1]) + evaluate(ast[2])
    elif ast[0] == 'Sub':
        return evaluate(ast[1]) - evaluate(ast[2])
    elif ast[0] == 'Mul':
        return evaluate(ast[1]) * evaluate(ast[2])
    elif ast[0] == 'Div':
        return evaluate(ast[1]) / evaluate(ast[2])
    else:
        raise ValueError("Invalid AST")

if __name__ == "__main__":
    input_expr = "3 + 5 * (2 - 1)"
    print(f"Input Expression: {input_expr}")
    lexer = Lexer(input_expr)
    parser = Parser(lexer.tokens)
    ast = parser.parse()
    print("Parsed AST:", ast)
    print("Evaluation Result:", evaluate(ast))
```

```
# Q5 Implement Minimization of Finite State Machines
(FSMs) and verify equivalence between two FSMs.

from collections import defaultdict
from typing import Set, Dict, Tuple, List, FrozenSet

class FSM:
    def __init__(self, states: Set[str], alphabet:
Set[str], transition: Dict[Tuple[str, str], str],
                 start_state: str, accept_states:
Set[str]):
        self.states = states
        self.alphabet = alphabet
        self.transition = transition
        self.start_state = start_state
        self.accept_states = accept_states

    def __repr__(self):
        return f"FSM(start={self.start_state},
accept={self.accept_states})"

    def minimize(self):
        partition = [self.accept_states, self.states -
self.accept_states]
        stable = False

        while not stable:
            new_partition = []
            stable = True

            for group in partition:
                group_dict = defaultdict(set)
                for state in group:
                    key =
tuple(self.get_target_group(state, sym, partition) for
sym in sorted(self.alphabet))
                    group_dict[key].add(state)


new_partition.extend(group_dict.values())

                if len(group_dict) > 1:
                    stable = False

            partition = new_partition
```

```
        new_state_map = {}
        for i, group in enumerate(partition):
            for state in group:
                new_state_map[state] = f'q{i}'

        new_states = set(new_state_map.values())
        new_start = new_state_map[self.start_state]
        new_accept = {new_state_map[s] for s in
self.accept_states}
        new_trans = {}

        for (state, sym), target in
self.transition.items():
            if state in new_state_map and target in
new_state_map:
                new_trans[(new_state_map[state], sym)]
= new_state_map[target]

        return FSM(new_states, self.alphabet,
new_trans, new_start, new_accept)

    def get_target_group(self, state, symbol,
partition):
        target = self.transition.get((state, symbol))
        for i, group in enumerate(partition):
            if target in group:
                return i
        return -1

    def is_equivalent(self, other) -> bool:
        min_self = self.minimize()
        min_other = other.minimize()

        return (min_self.states == min_other.states and
                min_self.alphabet == min_other.alphabet
and
                min_self.start_state ==
min_other.start_state and
                min_self.accept_states ==
min_other.accept_states and
                min_self.transition ==
min_other.transition)

if __name__ == "__main__":
```

```
    states1 = {'A', 'B', 'C', 'D'}
    alphabet = {'0', '1'}
    transitions1 = {
        ('A', '0'): 'B', ('A', '1'): 'C',
        ('B', '0'): 'A', ('B', '1'): 'D',
        ('C', '0'): 'D', ('C', '1'): 'A',
        ('D', '0'): 'C', ('D', '1'): 'B'
    }
    start1 = 'A'
    accept1 = {'A'}

    states2 = {'X', 'Y', 'Z'}
    transitions2 = {
        ('X', '0'): 'Y', ('X', '1'): 'Z',
        ('Y', '0'): 'X', ('Y', '1'): 'Z',
        ('Z', '0'): 'Z', ('Z', '1'): 'X',
    }
    start2 = 'X'
    accept2 = {'X'}

    fsm1 = FSM(states1, alphabet, transitions1, start1,
accept1)
    fsm2 = FSM(states2, alphabet, transitions2, start2,
accept2)

    print("FSM1 (minimized):",
fsm1.minimize().__dict__)
    print("FSM2 (minimized):",
fsm2.minimize().__dict__)
    print("Are FSM1 and FSM2 equivalent?",
fsm1.is_equivalent(fsm2))
```

```
.../formal/lab/2025-04-11
~/desktop/cse/ASSGN/sem8/formal/lab/2025-04-11
→ python3 q5.py
FSM1 (minimized): {'states': {'q1', 'q2', 'q0', 'q3'}, 'alphabet': {'0', '1'},
 'transition': {('q0', '0'): 'q1', ('q0', '1'): 'q3', ('q1', '0'): 'q0', ('q1'
, '1'): 'q2', ('q3', '0'): 'q2', ('q3', '1'): 'q0', ('q2', '0'): 'q3', ('q2',
'1'): 'q1'}, 'start_state': 'q0', 'accept_states': {'q0'}}
FSM2 (minimized): {'states': {'q1', 'q2', 'q0'}, 'alphabet': {'0', '1'}, 'tran
sition': {('q0', '0'): 'q2', ('q0', '1'): 'q1', ('q2', '0'): 'q0', ('q2', '1')
: 'q1', ('q1', '0'): 'q1', ('q1', '1'): 'q0'}, 'start_state': 'q0', 'accept_st
ates': {'q0'}}
Are FSM1 and FSM2 equivalent? False
~/desktop/cse/ASSGN/sem8/formal/lab/2025-04-11
→
```