Name: Shreeya Chatterji
Roll Number: 102103447
Class: COE16/32

UCS802 COMPILER CONSTRUCTION LAB ASSIGNMENT 1

Design a Minimized DFA for the Regular Expression (a/b)*abb i.e. All strings ending with abb.
This will involve three steps:
Generate the NFA using Thomson' s Construction (3 Marks) (2 Lab of 2 Hrs.)
Generate the DFA using Subset Construction (5 marks) (3 Labs of 2 Hrs. each)
Minimize the DFA generated (2 Marks)(2 Labs of 2 Hrs. each)

Step 1: Creating NFA from Regex using Thompson's Construction

```
class NFAState:
    def __init__(self):
      # Dictionary of inputs
        self.transitions = {}
        # list of epsilon transitions
        self.epsilon_transitions = []

    def add_transition(self, input_symbol, state):
        if input_symbol not in self.transitions:
            self.transitions[input_symbol] = set()
        self.transitions[input_symbol].add(state)

    def add_epsilon_transition(self, state):
        self.epsilon_transitions.append(state)

class NFA:
    def __init__(self):
        self.start_state = NFAState()
        self.accept_state = NFAState()

    @staticmethod
    def from_single_symbol(symbol):
        nfa = NFA()
        nfa.start_state.add_transition(symbol, nfa.accept_state)
        print(f"NFA for symbol '{symbol}':")
        print(f"Start state transitions:
{nfa.start_state.transitions}\n")
        return nfa

    @staticmethod
    def concatenate(nfa1, nfa2):
        nfa1.accept_state.add_epsilon_transition(nfa2.start_state)
        nfa = NFA()
        nfa.start_state = nfa1.start_state
        nfa.accept_state = nfa2.accept_state
        print(f"NFA after concatenation:")
```

```
        print(f"Start state transitions:
{nfa.start_state.transitions}\n")
        return nfa

    @staticmethod
    def union(nfa1, nfa2):
        nfa = NFA()
        nfa.start_state.add_epsilon_transition(nfa1.start_state)
        nfa.start_state.add_epsilon_transition(nfa2.start_state)
        nfa1.accept_state.add_epsilon_transition(nfa.accept_state)
        nfa2.accept_state.add_epsilon_transition(nfa.accept_state)
        print(f"NFA after union:")
        print(f"Start state transitions:
{nfa.start_state.epsilon_transitions}\n")
        return nfa

    @staticmethod
    def kleene_star(nfa):
        new_nfa = NFA()
        new_nfa.start_state.add_epsilon_transition(nfa.start_state)
        new_nfa.start_state.add_epsilon_transition(new_nfa.accept_state
)
        nfa.accept_state.add_epsilon_transition(nfa.start_state)
        nfa.accept_state.add_epsilon_transition(new_nfa.accept_state)
        print(f"NFA after Kleene star:")
        print(f"Start state epsilon transitions:
{new_nfa.start_state.epsilon_transitions}\n")
        return new_nfa

# Test building NFA for the regex (a/b)*abb
# We can create another regex expression to compare
def build_nfa_for_expression():
    a_nfa = NFA.from_single_symbol('a')
    b_nfa = NFA.from_single_symbol('b')

    union_nfa = NFA.union(a_nfa, b_nfa)
    kleene_nfa = NFA.kleene_star(union_nfa)

    a_nfa2 = NFA.from_single_symbol('a')
    b_nfa2 = NFA.from_single_symbol('b')
    b_nfa3 = NFA.from_single_symbol('b')

    abb_nfa = NFA.concatenate(a_nfa2, NFA.concatenate(b_nfa2, b_nfa3))

    final_nfa = NFA.concatenate(kleene_nfa, abb_nfa)
    return final_nfa

nfa = build_nfa_for_expression()
```

Name: Shreeya Chatterji
Roll Number: 102103447
Class: COE16/32

Step 2: Generate DFA using Subset Construction

```
class DFAState:
    def __init__(self, nfa_states):
        self.nfa_states = nfa_states
        self.transitions = {}

    def __repr__(self):
        return f"DFAState({[id(state) for state in self.nfa_states]})"

class DFA:
    def __init__(self, nfa):
        self.start_state = None
        self.states = []
        self.final_states = set()
        self.nfa_accept_state = nfa.accept_state
        self.build_from_nfa(nfa)

    def epsilon_closure(self, nfa_states):
        closure = set(nfa_states)
        stack = list(nfa_states)
        while stack:
            state = stack.pop()
            for next_state in state.epsilon_transitions:
                if next_state not in closure:
                    closure.add(next_state)
                    stack.append(next_state)
        return closure

    def move(self, nfa_states, symbol):
        next_states = set()
        for state in nfa_states:
            if symbol in state.transitions:
                next_states.update(state.transitions[symbol])
        return self.epsilon_closure(next_states)

    def build_from_nfa(self, nfa):
        start_closure = self.epsilon_closure([nfa.start_state])
        start_state = DFAState(start_closure)
        self.start_state = start_state
        self.states = [start_state]
        unprocessed_states = [start_state]
        state_mapping = {frozenset(start_closure): start_state}

        print(f"Initial DFA state: {start_state}")

        while unprocessed_states:
            current_dfa_state = unprocessed_states.pop()
```

Name: Shreeya Chatterji
Roll Number: 102103447
Class: COE16/32

```
            if self.nfa_accept_state in current_dfa_state.nfa_states:
                self.final_states.add(current_dfa_state)

            for symbol in ['a', 'b']:
                next_closure = self.move(current_dfa_state.nfa_states,
symbol)

                if frozenset(next_closure) not in state_mapping:
                    new_dfa_state = DFAState(next_closure)
                    self.states.append(new_dfa_state)
                    state_mapping[frozenset(next_closure)] =
new_dfa_state
                    unprocessed_states.append(new_dfa_state)

                current_dfa_state.transitions[symbol] =
state_mapping[frozenset(next_closure)]

            print(f"Processed DFA state: {current_dfa_state}")
            print(f"Transitions: {current_dfa_state.transitions}\n")

    def is_final(self, state):
        return state in self.final_states

    def simulate(self, input_string):
        """Simulate the DFA with an input string."""
        current_state = self.start_state
        for symbol in input_string:
            if symbol not in current_state.transitions:
                return False  # Invalid transition
            current_state = current_state.transitions[symbol]

        # Check if current state is a final state
        return self.is_final(current_state)


dfa = DFA(nfa)
```

Step 3: Minimize DFA

```
def minimize_dfa(dfa):
    final_states = [state for state in dfa.states if
dfa.is_final(state)]
    non_final_states = [state for state in dfa.states if not
dfa.is_final(state)]

    partition = [set(final_states), set(non_final_states)]
    new_partition = []
```

Name: Shreeya Chatterji
Roll Number: 102103447
Class: COE16/32

```
    print(f"Initial partition: {partition}\n")

    while partition != new_partition:
        if new_partition:
            partition = new_partition
        new_partition = []

        for group in partition:
            subsets = {}
            for state in group:
                transition_signature = tuple(
                    next((i for i, grp in enumerate(partition) if
state.transitions.get(symbol) in grp), None)
                    for symbol in ['a', 'b']
                )
                if transition_signature not in subsets:
                    subsets[transition_signature] = set()
                subsets[transition_signature].add(state)

            new_partition.extend(subsets.values())

        print(f"New partition: {new_partition}\n")

    # Build a minimized DFA (actual merging logic omitted for brevity)

# Minimize the DFA and print partitioning
minimize_dfa(dfa)
```

Step 4: Using the functions to generate required output

```
import re
import itertools

# Function to generate example strings that match the regex
def generate_strings(regex, num_strings=20):
    examples = []
    # Start generating strings with increasing lengths
    for length in range(1, num_strings + 1):
        for combination in itertools.product('ab', repeat=length):
            test_str = ''.join(combination)
            # Check if the generated string matches the regex
            if re.fullmatch(regex, test_str):
                examples.append(test_str)
                if len(examples) >= num_strings:
                    return examples
    return examples
```

Name: Shreeya Chatterji
Roll Number: 102103447
Class: COE16/32

```
# Ensure all regex characters are lowercase
def convert_regex_to_lowercase(regex):
    return regex.lower()

# Get user input for regular expression
user_regex = input("Enter a regular expression (e.g., (a|b)*abb):
").strip()
user_regex = convert_regex_to_lowercase(user_regex)  # Convert to
lowercase

# Generate example strings that match the user-provided regex
test_strings = generate_strings(user_regex)
print(f"Generated strings for '{user_regex}':", test_strings)


# Check if the DFA accepts all example strings
accepted = all(dfa.simulate(test_string) for test_string in
test_strings)

# Print result
if accepted:
    print(f"All generated strings matching the regex '{user_regex}' are
accepted by the DFA.")
    print("ACCEPT")
else:
    print(f"Not all strings matching the regex '{user_regex}' are
accepted by the DFA.")
    print("REJECT")
```

Final Output:

Prompt to enter Regular Expression to compare with (a|b)*abb:

```
Enter a regular expression (e.g., (a|b)*abb):
```

Outputs with different inputs:

- **(a|b) –** Since all strings generated by this RE will not end with abb the output should be “REJECTED”.

```
Enter a regular expression (e.g., (a|b)*abb): (a|b)
Generated strings for '(a|b)': ['a', 'b']
Not all strings matching the regex '(a|b)' are accepted by the DFA.
REJECT
```

Name: Shreeya Chatterji
Roll Number: 102103447
Class: COE16/32

- **(a|b)* -** Since all strings generated by this RE will not end with abb the output should be "REJECTED".

```
Enter a regular expression (e.g., (a|b)*abb): (a|b)*
Generated strings for '(a|b)*': ['a', 'b', 'aa', 'ab', 'ba', 'bb', 'aaa', 'aab', 'aba', 'abb',
Not all strings matching the regex '(a|b)*' are accepted by the DFA.
REJECT
```

- **aaa -** Since all strings generated by this RE will not end with abb the output should be "REJECTED".

```
Enter a regular expression (e.g., (a|b)*abb): aaa
Generated strings for 'aaa': ['aaa']
Not all strings matching the regex 'aaa' are accepted by the DFA.
REJECT
```

- **(a|b)*ab -** Since all strings generated by this RE will not end with abb the output should be "REJECTED".

```
Enter a regular expression (e.g., (a|b)*abb): (a|b)*ab
Generated strings for '(a|b)*ab': ['ab', 'aab', 'bab', 'aaab', 'abab', 'baa
Not all strings matching the regex '(a|b)*ab' are accepted by the DFA.
REJECT
```

- **a*abb -** Since all strings generated by this RE will end with abb the output should be "ACCEPTED".

```
Enter a regular expression (e.g., (a|b)*abb): a*abb
Generated strings for 'a*abb': ['abb', 'aabb', 'aaabb', 'aaaabb', 'aaaaabb'
All generated strings matching the regex 'a*abb' are accepted by the DFA.
ACCEPT
```

- **(a|b)*abb -** Since all strings generated by this RE will end with abb the output should be "ACCEPTED".

```
Enter a regular expression (e.g., (a|b)*abb): (a|b)*abb
Generated strings for '(a|b)*abb': ['abb', 'aabb', 'babb', 'aaabb', 'ababb'
All generated strings matching the regex '(a|b)*abb' are accepted by the DF
ACCEPT
```