Assignment 1- Generate a Deterministic Finite State Automata from Regular Expression using Thomson's rule and Subset Construction Algorithm.

Name-Anvit Kumar

Roll no-102203161

Group-4C14

This assignment involves designing a system to convert a regular expression into a Minimized Deterministic Finite Automaton (DFA). The process is broken down into three fundamental steps:

1. Thompson's Construction: Converting the regular expression (a/b)*abb into a Non-deterministic Finite Automaton (NFA).
2. Subset Construction: Converting the NFA into a DFA.
3. DFA Minimization: Reducing the DFA to its smallest equivalent form using the partition refinement algorithm.

The implementation is done in Python. Below is the breakdown of the code with explanations.

Part 1: Data Structures

We first define two classes, NFA and DFA, to represent our automata. Both classes store the set of states, the alphabet, the transition function (a dictionary), the start state, and the set of final states.

```python
# Epsilon symbol
EPSILON = 'ε'

class NFA:
    """Class to represent a Non-deterministic Finite Automaton."""
    def __init__(self, states, alphabet, transitions, start_state, final_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.start_state = start_state
        self.final_states = final_states

    def __str__(self):
        return (
            f"NFA:\n"
            f"  States: {self.states}\n"
            f"  Alphabet: {self.alphabet}\n"
            f"  Start State: {self.start_state}\n"
            f"  Final States: {self.final_states}\n"
            f"  Transitions: {self.transitions}"
        )
```

```python
class DFA:
    """Class to represent a Deterministic Finite Automaton."""
    def __init__(self, states, alphabet, transitions, start_state, final_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.start_state = start_state
        self.final_states = final_states

    def __str__(self):
        return (
            f"DFA:\n"
            f"  States: {self.states}\n"
            f"  Alphabet: {self.alphabet}\n"
            f"  Start State: {self.start_state}\n"
            f"  Final States: {self.final_states}\n"
            f"  Transitions: {self.transitions}"
        )
```

Part 2: Step 1 - Thompson's Construction (RE to NFA)

2.1 Infix to Postfix Conversion

Thompson's construction is significantly easier to implement using a postfix (Reverse Polish Notation) expression. The preprocess_regex function inserts explicit concatenation operators (.), and infix_to_postfix converts the standard infix notation.

```python
def preprocess_regex(regex):
    """Inserts explicit concatenation operators '.' into the regex."""
    output = []
    for i in range(len(regex)):
        output.append(regex[i])
        if i + 1 < len(regex):
            # Insert '.' if current and next chars are operands or specific operators
            if (regex[i].isalnum() or regex[i] in ')*') and \
               (regex[i+1].isalnum() or regex[i+1] == '('):
                output.append('.')
    return "".join(output)
```

```python
def infix_to_postfix(regex):
    """Converts an infix regular expression to postfix."""
    # Use '|' for union to avoid confusion with file paths
    regex = regex.replace('/', '|')
    preprocessed = preprocess_regex(regex)

    precedence = {'|': 1, '.': 2, '*': 3}
    postfix = []
    operator_stack = []

    for char in preprocessed:
        if char.isalnum():
            postfix.append(char)
        elif char == '(':
            operator_stack.append(char)
        elif char == ')':
            while operator_stack and operator_stack[-1] != '(':
                postfix.append(operator_stack.pop())
            operator_stack.pop() # Pop '('
        else: # Operator
            while (operator_stack and operator_stack[-1] != '(' and
                   precedence.get(operator_stack[-1], 0) >= precedence.get(char, 0)):
                postfix.append(operator_stack.pop())
            operator_stack.append(char)

    while operator_stack:
        postfix.append(operator_stack.pop())

    return "".join(postfix)
```

2.2 Building the NFA

The thompson_construction function iterates through the postfix expression, using a stack to build the NFA.

- Base Case (Literal): Pushes a two-state NFA for a single character.
- Concatenation (.): Pops two NFAs and links them with an ε-transition.
- Union (|): Pops two NFAs and creates a new start and end state with ε-transitions to/from both.
- Kleene Star (*): Pops one NFA and creates a looping structure with ε-transitions.

```python
def thompson_construction(postfix_regex):
    """Builds an NFA from a postfix regular expression."""
    nfa_stack = []
    state_counter = 0

    def new_state():
        nonlocal state_counter
        state = state_counter
        state_counter += 1
        return state

    for char in postfix_regex:
        if char.isalnum():
            # Base case: create NFA for a single character
            start = new_state()
            final = new_state()
            nfa = NFA(
                states={start, final},
                alphabet={char},
                transitions={start: {char: {final}}},
                start_state=start,
                final_states={final}
            )
            nfa_stack.append(nfa)
        elif char == '.':
            # Concatenation
            nfa2 = nfa_stack.pop()
            nfa1 = nfa_stack.pop()

            # Merge final state of nfa1 with start state of nfa2
            nfa1.transitions.update(nfa2.transitions)
            for s in nfa1.final_states:
                if s not in nfa1.transitions: nfa1.transitions[s] = {}
                nfa1.transitions[s][EPSILON] = {nfa2.start_state}
```

```
            nfa1.states.update(nfa2.states)
            nfa1.alphabet.update(nfa2.alphabet)
            nfa1.final_states = nfa2.final_states
            nfa_stack.append(nfa1)

        elif char == '|':
            # Union
            nfa2 = nfa_stack.pop()
            nfa1 = nfa_stack.pop()
            start = new_state()
            final = new_state()

            transitions = {start: {EPSILON: {nfa1.start_state, nfa2.start_state}}}
            transitions.update(nfa1.transitions)
            transitions.update(nfa2.transitions)
            for s in nfa1.final_states.union(nfa2.final_states):
                if s not in transitions: transitions[s] = {}
                transitions[s][EPSILON] = {final}

            nfa = NFA(
                states=nfa1.states.union(nfa2.states).union({start, final}),
                alphabet=nfa1.alphabet.union(nfa2.alphabet),
                transitions=transitions,
                start_state=start,
                final_states={final}
            )
            nfa_stack.append(nfa)

        elif char == '*':
            # Kleene Star
            nfa1 = nfa_stack.pop()
            start = new_state()
            final = new_state()

            transitions = {start: {EPSILON: {nfa1.start_state, final}}}
            transitions.update(nfa1.transitions)
            for s in nfa1.final_states:
                if s not in transitions: transitions[s] = {}
                transitions[s][EPSILON] = {nfa1.start_state, final}

            nfa = NFA(
                states=nfa1.states.union({start, final}),
                alphabet=nfa1.alphabet,
                transitions=transitions,
                start_state=start,
                final_states={final}
            )
            nfa_stack.append(nfa)

    return nfa_stack.pop()
```

Part 3: Step 2 - Subset Construction (NFA to DFA)

The Subset Construction algorithm converts the NFA into a DFA where each DFA state represents a set of possible NFA states.

3.1 Helper Functions

- epsilon_closure(states): Computes the set of all NFA states reachable from a given set through any number of ε-transitions.
- move(states, char): Computes the set of all NFA states reachable by consuming an input character char from any state in the given set.

```
# --- Step 2: Subset Construction (NFA -> DFA) ---

def epsilon_closure(nfa, states):
    """Computes the epsilon closure for a set of NFA states."""
    closure = set(states)
    stack = list(states)
    while stack:
        state = stack.pop()
        # Find states reachable by epsilon transitions
        epsilon_neighbors = nfa.transitions.get(state, {}).get(EPSILON, set())
        for neighbor in epsilon_neighbors:
            if neighbor not in closure:
                closure.add(neighbor)
                stack.append(neighbor)
    return frozenset(closure)

def move(nfa, states, char):
    """Computes the set of states reachable on a character from a set of states."""
    reachable_states = set()
    for state in states:
        reachable_states.update(nfa.transitions.get(state, {}).get(char, set()))
    return frozenset(reachable_states)
```

3.2 The Algorithm

The process starts with the ε-closure of the NFA's start state. It then systematically explores transitions for all discovered DFA states, adding new "subset" states to a worklist until all reachable states are found.

```python
def subset_construction(nfa):
    """Converts an NFA to a DFA using the subset construction algorithm."""
    # The DFA states are sets of NFA states
    dfa_states = set()
    dfa_transitions = {}

    # The initial DFA state is the epsilon closure of the NFA's start state
    initial_dfa_state = epsilon_closure(nfa, {nfa.start_state})
    dfa_states.add(initial_dfa_state)

    worklist = [initial_dfa_state]

    while worklist:
        current_dfa_state = worklist.pop(0)
        dfa_transitions[current_dfa_state] = {}

        for char in nfa.alphabet:
            next_nfa_states = move(nfa, current_dfa_state, char)
            next_dfa_state = epsilon_closure(nfa, next_nfa_states)

            if not next_dfa_state: # Trap state if empty
                continue

            if next_dfa_state not in dfa_states:
                dfa_states.add(next_dfa_state)
                worklist.append(next_dfa_state)

            dfa_transitions[current_dfa_state][char] = next_dfa_state

    # Map frozensets to integer state names for clarity
    state_map = {state: i for i, state in enumerate(dfa_states)}

    # Determine final states of the DFA
    dfa_final_states = {
        state_map[s] for s in dfa_states if not nfa.final_states.isdisjoint(s)
    }

    # Build the final DFA object
    return DFA(
        states=set(state_map.values()),
        alphabet=nfa.alphabet,
        transitions={state_map[s]: {c: state_map[t] for c, t in trans.items()} for s, trans in dfa_transitions.items()},
        start_state=state_map[initial_dfa_state],
        final_states=dfa_final_states
    )
```

Part 4: Step 3 - DFA Minimization

The final step uses the Partition Refinement Algorithm (Hopcroft's Algorithm) to merge equivalent states.

Algorithm Logic:

- Initial Partition: Divide states into two groups: Final and Non-Final.
- Refinement Loop: For each group and each input symbol, check if all states in the group transition to the same partition. If they transition to different partitions, the group must be split.
- Termination: The process repeats until no more splits are possible. The resulting partitions are the states of the minimized DFA.

```python
# --- Step 3: DFA Minimization ---

def minimize_dfa(dfa):
    """Minimizes a DFA using the partition refinement algorithm."""
    # Initial partition: final states and non-final states
    final_states = frozenset(dfa.final_states)
    non_final_states = frozenset(dfa.states - dfa.final_states)
    partitions = {final_states, non_final_states}
    worklist = {final_states, non_final_states}

    # Filter out empty sets if all states are final or non-final
    partitions = {p for p in partitions if p}
    worklist = {w for w in worklist if w}

    while worklist:
        A = worklist.pop()
        for char in dfa.alphabet:
            # X is the set of states that transition into A on character 'char'
            X = frozenset({s for s in dfa.states if dfa.transitions.get(s, {}).get(char) in A})

            new_partitions = set()
            for Y in partitions:
                intersection = Y.intersection(X)
                difference = Y.difference(X)
                if intersection and difference:
                    # Split Y
                    new_partitions.add(intersection)
                    new_partitions.add(difference)
                    if Y in worklist:
                        worklist.remove(Y)
                        worklist.add(intersection)
                        worklist.add(difference)
                    else:
                        if len(intersection) <= len(difference):
                            worklist.add(intersection)
                        else:
                            worklist.add(difference)
                else:
                    # No split, keep Y as is
                    new_partitions.add(Y)
            partitions = new_partitions
```

```python
    # Create the minimized DFA from the final partitions
    min_state_map = {frozenset(p): i for i, p in enumerate(partitions)}
    min_states = set(min_state_map.values())
    min_alphabet = dfa.alphabet

    min_start_state = None
    for p, i in min_state_map.items():
        if dfa.start_state in p:
            min_start_state = i
            break

    min_final_states = {min_state_map[p] for p in partitions if not p.isdisjoint(dfa.final_states)}

    min_transitions = {}
    for p, i in min_state_map.items():
        # Pick a representative state from the partition
        representative = next(iter(p))
        min_transitions[i] = {}
        for char in min_alphabet:
            target_state = dfa.transitions.get(representative, {}).get(char)
            if target_state is not None:
                for target_p, target_i in min_state_map.items():
                    if target_state in target_p:
                        min_transitions[i][char] = target_i
                        break

    return DFA(min_states, min_alphabet, min_transitions, min_start_state, min_final_states)
```

Part 5: Simulation and Execution

The simulate_dfa function tests a string against the final DFA. The main execution block runs the entire pipeline for the assignment's regular expression and prints the results.

```python
def simulate_dfa(dfa, input_string):
    """Simulates a DFA on an input string."""
    current_state = dfa.start_state
    for char in input_string:
        if char not in dfa.alphabet:
            return False # Character not in alphabet

        current_state = dfa.transitions.get(current_state, {}).get(char)
        if current_state is None:
            return False # No transition defined (implicit trap state)

    return current_state in dfa.final_states

# --- Main Execution ---

if __name__ == "__main__":
    # The regular expression from the assignment
    regex = "(a/b)*abb"
    print(f"Regular Expression: {regex}\n")

    # --- Step 1 ---
    print("--- Step 1: RE to NFA (Thompson's Construction) ---")
    postfix = infix_to_postfix(regex)
    print(f"Postfix Expression: {postfix}")
    nfa = thompson_construction(postfix)
    print(nfa)
    print("-" * 50)

    # --- Step 2 ---
    print("--- Step 2: NFA to DFA (Subset Construction) ---")
    unminimized_dfa = subset_construction(nfa)
    print("Generated (Unminimized) DFA:")
    print(unminimized_dfa)
    print("-" * 50)

    # --- Step 3 ---
    print("--- Step 3: DFA Minimization ---")
    minimized_dfa = minimize_dfa(unminimized_dfa)
    print("Minimized DFA:")
    print(minimized_dfa)
    print("-" * 50)

    # --- Final Output and Testing ---
    print("--- Testing the Minimized DFA ---")
    test_strings = ["abb", "aabb", "babb", "ab", "a", "banana", "", "abbabb"]

    for s in test_strings:
        result = simulate_dfa(minimized_dfa, s)
        output = "Accept" if result else "Not Accepted"
        print(f"Input: '{s}' -> Output: '{output}'")
```

Output:-

Regular Expression: (a/b)*abb

--- Step 1: RE to NFA (Thompson's Construction) ---

Postfix Expression: ab|*a.b.b.

NFA:

States: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}

Alphabet: {'a', 'b'}

Start State: 6

Final States: {13}

Transitions: {6: {'ε': {4, 7}}, 4: {'ε': {0, 2}}, 0: {'a': {1}}, 2: {'b': {3}}, 1: {'ε': {5}}, 3: {'ε': {5}}, 5: {'ε': {4, 7}}, 8: {'a': {9}}, 7: {'ε': {8}},

10: {'b': {11}}, 9: {'ε': {10}}, 12: {'b': {13}}, 11: {'ε': {12}}}

--- Step 2: NFA to DFA (Subset Construction) ---

Generated (Unminimized) DFA:

DFA:

States: {0, 1, 2, 3, 4}

Alphabet: {'a', 'b'}

Start State: 2

Final States: {4}

Transitions: {2: {'a': 1, 'b': 0}, 1: {'a': 1, 'b': 3}, 0: {'a': 1, 'b': 0}, 3: {'a': 1, 'b': 4}, 4: {'a': 1, 'b': 0}}

--- Step 3: DFA Minimization ---

DFA:

States: {0, 1, 2, 3}

Alphabet: {'a', 'b'}

Start State: 2

Final States: {3}

Transitions: {0: {'a': 1, 'b': 3}, 1: {'a': 1, 'b': 0}, 2: {'a': 1, 'b': 2}, 3: {'a': 1, 'b': 2}}

--- Testing the Minimized DFA ---

Input: 'abb' -> Output: 'Accept'

Input: 'aabb' -> Output: 'Accept'

Input: 'babb' -> Output: 'Accept'

Input: 'ab' -> Output: 'Not Accepted'

Input: 'a' -> Output: 'Not Accepted'

Input: 'banana' -> Output: 'Not Accepted'

Input: '' -> Output: 'Not Accepted'

Input: 'abbabb' -> Output: 'Accept'

Generate a Minimized DFA from a generic Regular Expression

```
# --- Main Execution (Generic Version) ---

if __name__ == "__main__":
    # Ask the user for a generic regular expression
    # Use '|' for union/OR, for example: (a|b)*abb
    regex_input = input("Enter the Regular Expression (use '|' for union): ")
    print(f"\nProcessing Regular Expression: {regex_input}\n")

    # --- Step 1: RE to NFA ---
    print("--- Step 1: RE to NFA (Thompson's Construction) ---")
    try:
        postfix = infix_to_postfix(regex_input)
        print(f"Postfix Expression: {postfix}")
        nfa = thompson_construction(postfix)
        print(nfa)
    except Exception as e:
        print(f"Error parsing Regex. Please check your syntax. Details: {e}")
        exit()
    print("-" * 50)

    # --- Step 2: NFA to DFA ---
    print("--- Step 2: NFA to DFA (Subset Construction) ---")
    unminimized_dfa = subset_construction(nfa)
    print("Generated (Unminimized) DFA:")
    print(unminimized_dfa)
    print("-" * 50)

    # --- Step 3: DFA Minimization ---
    print("--- Step 3: DFA Minimization ---")
    minimized_dfa = minimize_dfa(unminimized_dfa)
    print("Minimized DFA:")
    print(minimized_dfa)
    print("-" * 50)

    # --- Final Output and Testing ---
    print(f"--- Testing the Minimized DFA for RE: '{regex_input}' ---")
    print("Enter strings to test (or type 'quit' to exit):")

    while True:
        test_string = input(f"Test string for '{regex_input}'> ")
        if test_string.lower() == 'quit':
            break

        # Check if all characters in the string are in the DFA's alphabet
        is_valid_string = all(char in minimized_dfa.alphabet for char in test_string)

        if not is_valid_string:
            print(f"  Error: Input string contains characters not in the alphabet {minimized_dfa.alphabet}")
            continue

        result = simulate_dfa(minimized_dfa, test_string)
        output = "Accept" if result else "Not Accepted"
        print(f"  -> Output: '{output}'")
```

Output:-

Enter the Regular Expression (use '|' for union): a(b|c)*

Processing Regular Expression: a(b|c)*

--- Step 1: RE to NFA (Thompson's Construction) ---

Postfix Expression: abc|*.

NFA:

States: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

Alphabet: {'b', 'c', 'a'}

Start State: 0

Final States: {9}

Transitions: {0: {'a': {1}}, 8: {'ε': {9, 6}}, 6: {'ε': {2, 4}}, 2: {'b': {3}}, 4: {'c': {5}}, 3: {'ε': {7}}, 5: {'ε': {7}}, 7: {'ε': {9, 6}}, 1: {'ε': {8}}}

--- Step 2: NFA to DFA (Subset Construction) ---

Generated (Unminimized) DFA:

DFA:

States: {0, 1, 2, 3}

Alphabet: {'b', 'c', 'a'}

Start State: 2

Final States: {0, 1, 3}

Transitions: {2: {'a': 3}, 3: {'b': 1, 'c': 0}, 1: {'b': 1, 'c': 0}, 0: {'b': 1, 'c': 0}}

--- Step 3: DFA Minimization ---

Minimized DFA:

DFA:

```
--------------------------------------------------
--- Step 3: DFA Minimization ---
Minimized DFA:
DFA:
--- Step 3: DFA Minimization ---
Minimized DFA:
DFA:
Minimized DFA:
DFA:
DFA:
  States: {0, 1}
  Alphabet: {'b', 'c', 'a'}
  Start State: 0
  Final States: {1}
  Transitions: {0: {'a': 1}, 1: {'b': 1, 'c': 1}}
--------------------------------------------------
--- Testing the Minimized DFA for RE: 'a(b|c)*' ---
Enter strings to test (or type 'quit' to exit):
Test string for 'a(b|c)*'>
  Alphabet: {'b', 'c', 'a'}
  Start State: 0
  Final States: {1}
  Transitions: {0: {'a': 1}, 1: {'b': 1, 'c': 1}}
--------------------------------------------------
--- Testing the Minimized DFA for RE: 'a(b|c)*' ---
Enter strings to test (or type 'quit' to exit):
Test string for 'a(b|c)*'>
```

```
  Start State: 0
  Final States: {1}
  Transitions: {0: {'a': 1}, 1: {'b': 1, 'c': 1}}
--------------------------------------------------
--- Testing the Minimized DFA for RE: 'a(b|c)*' ---
Enter strings to test (or type 'quit' to exit):
Test string for 'a(b|c)*'>
  Transitions: {0: {'a': 1}, 1: {'b': 1, 'c': 1}}
--------------------------------------------------
--- Testing the Minimized DFA for RE: 'a(b|c)*' ---
Enter strings to test (or type 'quit' to exit):
Test string for 'a(b|c)*'>
--- Testing the Minimized DFA for RE: 'a(b|c)*' ---
Enter strings to test (or type 'quit' to exit):
Test string for 'a(b|c)*'> abcbc
  -> Output: 'Accept'
  -> Output: 'Accept'
```