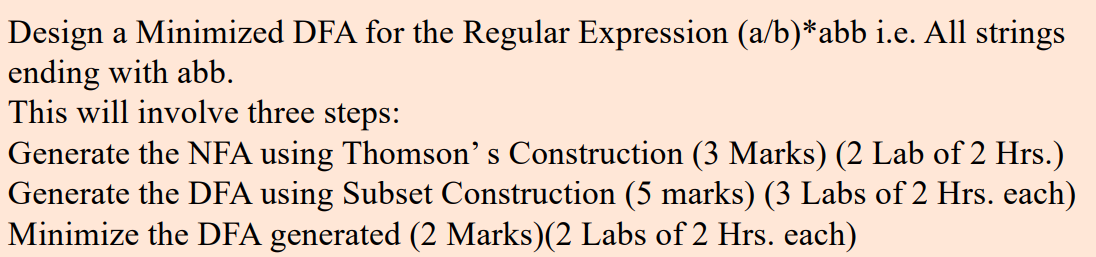
**UCS802 COMPILER CONSTRUCTION LAB ASSIGNMENT 1**



**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()

**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()

            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 = []

    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

# 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")

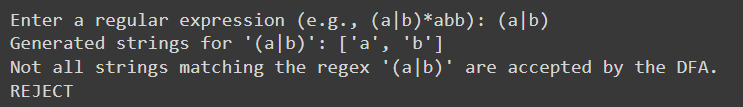
**Final Output:**

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

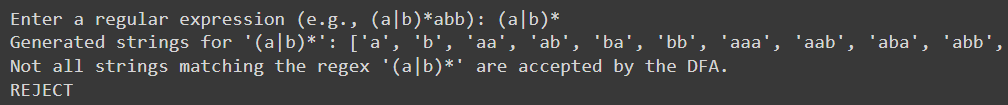


Outputs with different inputs:

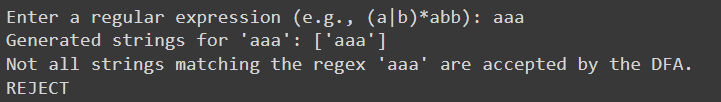
* + **(a|b) –** Since all strings generated by this RE will not end with abb the output should be “REJECTED”.



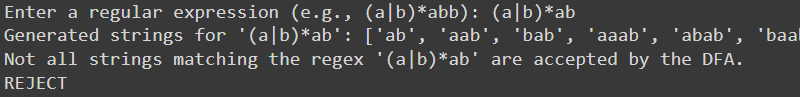
* + **(a|b)\* -** Since all strings generated by this RE will not end with abb the output should be “REJECTED”.



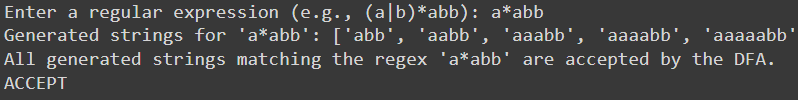
* + **aaa -** Since all strings generated by this RE will not end with abb the output should be “REJECTED”.



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



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



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

