ASSIGNMENTS SOLUTION

COURSE CODE : CSE/PC/B/S/322

COMPILER DESIGN LAB

By

SHAHIR HABIB

002210501006

Department of Computer Scirence & Engineering,

Jadavpur University , Kolkata

January-April, 2025

Assignment2\index.html

```html
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DFA Generator</title>
    <link rel = "stylesheet" type = "text/css" href = "style.css">
</head>

<body>
    <div class="container">
        <h1>DFA Generator from Regular Expression</h1>
        <label for="alphabet">Enter Alphabet (comma-separated, e.g., a,b):</label>
        <input type="text" id="alphabet" placeholder="a,b">

        <label for="regex">Enter Regular Expression:</label>
        <input type="text" id="regex" placeholder="(a+b)*">

        <button onclick="generateDFA()">Generate DFA</button>

        <h2>DFA State Table</h2>
        <table id="dfaTable">
            <thead>
                <tr>
                    <th>State</th>
                    <!-- Alphabet symbols will be dynamically added here -->
                    <th>Accepting?</th>
                </tr>
            </thead>
            <tbody>
                <!-- DFA rows will be dynamically added here -->
            </tbody>
        </table>
        <h2>DFA State Transition Diagram</h2>
        <img id="dfaDiagram" src= alt="DFA Diagram" style="max-width: 100%; height: auto;">

        <label for="inputString">Enter a string to simulate:</label>
        <input type="text" id="inputString" placeholder="e.g., aab">
        <button onclick="simulateString()">Simulate String</button>

        <p id="simulationResult"></p>

    </div>

    <script>
        function simulateString() {
            const inputString = document.getElementById('inputString').value.trim();
            const dfaTable = window.dfaTable; // Assuming you store the DFA table globally after fetching it
            const acceptingStates = window.acceptingStates; // Assuming you store the accepting states globally after fetching them
```

```
51
52              if (!dfaTable || !acceptingStates) {
53                  alert("DFA not generated yet. Please generate the DFA first.");
54                  return;
55              }
56
57              let currentState = 0; // Start from the initial state (q0)
58              for (const symbol of inputString) {
59                  if (!dfaTable[currentState].hasOwnProperty(symbol)) {
60                      // If the symbol is not in the alphabet, the string is rejected
61                      document.getElementById('simulationResult').innerText = `String
"${inputString}" is NOT accepted by the DFA.`;
62                      return;
63                  }
64                  currentState = dfaTable[currentState][symbol];
65              }
66
67              // Check if the final state is an accepting state
68              if (acceptingStates.includes(currentState)) {
69                  document.getElementById('simulationResult').innerText = `String
"${inputString}" is ACCEPTED by the DFA.`;
70              } else {
71                  document.getElementById('simulationResult').innerText = `String
"${inputString}" is NOT accepted by the DFA.`;
72              }
73          }
74
75          // Modify the generateDFA function to store the DFA table and accepting states
globally
76          async function generateDFA() {
77              const alphabetInput = document.getElementById('alphabet').value.trim();
78              const alphabet = alphabetInput.split(',').map(s => s.trim()).filter(s => s !==
'');
79              const regex = document.getElementById('regex').value.trim();
80
81              const response = await fetch('http://127.0.0.1:5000/generate-dfa', {
82                  method: 'POST',
83                  headers: {
84                      'Content-Type': 'application/json',
85                  },
86                  body: JSON.stringify({ alphabet, regex }),
87              });
88
89              const result = await response.json();
90              if (result.error) {
91                  alert(result.error);
92                  return;
93              }
94
95              const { dfa_table, accepting_states, diagram_url } = result;
96              const diagramImg = document.getElementById('dfaDiagram');
97              diagramImg.src = "./dfa.png?t=" + new Date().getTime(); // Force reload the
image
98              console.log(result);
```

```
            // Store the DFA table and accepting states globally
            window.dfaTable = dfa_table;
            window.acceptingStates = accepting_states;

            // Update DFA table display
            const table = document.getElementById('dfaTable');
            const thead = table.querySelector('thead');
            const tbody = table.querySelector('tbody');

            thead.innerHTML = `<tr><th>State</th>${alphabet.map(s => `<th>${s}</th>`).join('')}<th>Accepting?</th></tr>`;
            tbody.innerHTML = Object.keys(dfa_table).map(state => {
                const isAccepting = accepting_states.includes(Number(state));
                const rowClass = isAccepting ? 'accepting' : 'not-accepting'; // Apply the respective class based on the state type

                return `<tr class="${rowClass}">
            <td>q${state}</td>
            ${alphabet.map(s => `<td>q${dfa_table[state][s]}</td>`).join('')}
            <td>${isAccepting ? 'Yes' : 'No'}</td>
        </tr>`;
            }).join('');
        }
    </script>
</body>

</html>
```

Assignment2\dfa_generator.py

```python
from flask import Flask, request, jsonify
from flask_cors import CORS
import re

app = Flask(__name__)
CORS(app)

class State:
    def __init__(self, is_accepting=False):
        self.transitions = {}  # Key: symbol (str) or None for epsilon, Value: set of States
        self.is_accepting = is_accepting

class NFA:
    def __init__(self, start, end):
        self.start = start
        self.end = end  # End state

def epsilon_closure(state):
    closure = set()
    stack = [state]
    closure.add(state)
    while stack:
        s = stack.pop()
        for eps_trans in s.transitions.get(None, set()):
            if eps_trans not in closure:
                closure.add(eps_trans)
                stack.append(eps_trans)
    return closure

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

def insert_concat_operators(regex):
    if len(regex) == 0:
        return regex
    new_regex = [regex[0]]
    for i in range(1, len(regex)):
        prev = regex[i - 1]
        curr = regex[i]
        # Handle character classes and escaped characters
        if curr == '[' or (prev == '\\' and curr in {'d', 's', 'w', 'b', 'D', 'S', 'W'}):
            new_regex.append(curr)
            continue
        # Insert concatenation operator where necessary
        if (prev in [')', '*', '?', '+'] or (prev not in {'|', '(', ')', '*', '+', '?'}))
and \
           (curr == '(' or (curr not in {'|', ')', '*', '+', '?'})):
            new_regex.append('.')
```

```
        new_regex.append(curr)
    return ''.join(new_regex)

def shunting_yard(regex):
    precedence = {'*': 4, '?': 4, '+': 4, '.': 3, '|': 2, '(': 1}
    output = []
    stack = []
    i = 0
    while i < len(regex):
        token = regex[i]
        if token == '\\':
            # Handle escaped characters like \d, \s, \w, \b, etc.
            if i + 1 >= len(regex):
                raise ValueError("Invalid escape sequence")
            output.append(regex[i:i+2])
            i += 2
            continue
        elif token == '[':
            # Handle character classes like [a-z], [0-9_], etc.
            j = i + 1
            while j < len(regex) and regex[j] != ']':
                j += 1
            if j >= len(regex):
                raise ValueError("Unclosed character class")
            output.append(regex[i:j+1])
            i = j + 1
            continue
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            if not stack:
                raise ValueError("Mismatched parentheses")
            stack.pop()  # Pop the '('
        elif token in precedence:
            while stack and stack[-1] != '(' and precedence[stack[-1]] >= precedence[token]:
                output.append(stack.pop())
            stack.append(token)
        else:
            output.append(token)
        i += 1
    while stack:
        if stack[-1] == '(':
            raise ValueError("Mismatched parentheses")
        output.append(stack.pop())
    return ''.join(output)

def build_nfa(postfix):
    stack = []
    for token in postfix:
        if token == '*':
            nfa = stack.pop()
            new_start = State()
```

```
            new_end = State(is_accepting=True)
            new_start.transitions[None] = {nfa.start, new_end}
            nfa.end.is_accepting = False
            nfa.end.transitions[None] = {nfa.start, new_end}
            stack.append(NFA(new_start, new_end))
        elif token == '.':
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            nfa1.end.is_accepting = False
            nfa1.end.transitions[None] = {nfa2.start}
            stack.append(NFA(nfa1.start, nfa2.end))
        elif token == '|':
            nfa2 = stack.pop()
            nfa1 = stack.pop()
            new_start = State()
            new_end = State(is_accepting=True)
            new_start.transitions[None] = {nfa1.start, nfa2.start}
            nfa1.end.is_accepting = False
            nfa2.end.is_accepting = False
            nfa1.end.transitions[None] = {new_end}
            nfa2.end.transitions[None] = {new_end}
            stack.append(NFA(new_start, new_end))
        elif token == '?':
            nfa = stack.pop()
            new_start = State()
            new_end = State(is_accepting=True)
            new_start.transitions[None] = {nfa.start, new_end}
            nfa.end.is_accepting = False
            nfa.end.transitions[None] = {new_end}
            stack.append(NFA(new_start, new_end))
        elif token == '+':
            nfa = stack.pop()
            new_start = State()
            new_end = State(is_accepting=True)
            new_start.transitions[None] = {nfa.start}
            nfa.end.is_accepting = False
            nfa.end.transitions[None] = {new_end, nfa.start}
            stack.append(NFA(new_start, new_end))
        elif token.startswith('[') and token.endswith(']'):
            # Handle character classes like [a-z], [0-9_], etc.
            char_class = token[1:-1]
            start = State()
            end = State(is_accepting=True)
            if '-' in char_class:
                # Handle ranges like a-z, 0-9
                ranges = char_class.split(',')
                for r in ranges:
                    if '-' in r:
                        start_char, end_char = r.split('-')
                        for c in range(ord(start_char), ord(end_char) + 1):
                            start.transitions[chr(c)] = {end}
                    else:
                        start.transitions[r] = {end}
            else:
```

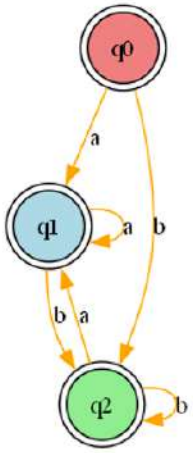
```
                for c in char_class:
                    start.transitions[c] = {end}
            stack.append(NFA(start, end))
        elif token.startswith('\\'):
            # Handle escaped characters like \d, \s, \w, \b, etc.
            escape_char = token[1]
            start = State()
            end = State(is_accepting=True)
            if escape_char == 'd':
                # \d matches any digit
                for c in range(ord('0'), ord('9') + 1):
                    start.transitions[chr(c)] = {end}
            elif escape_char == 's':
                # \s matches any whitespace character
                for c in [' ', '\t', '\n', '\r']:
                    start.transitions[c] = {end}
            elif escape_char == 'w':
                # \w matches any word character (alphanumeric + underscore)
                for c in range(ord('0'), ord('9') + 1):
                    start.transitions[chr(c)] = {end}
                for c in range(ord('A'), ord('Z') + 1):
                    start.transitions[chr(c)] = {end}
                for c in range(ord('a'), ord('z') + 1):
                    start.transitions[chr(c)] = {end}
                start.transitions['_'] = {end}
            elif escape_char == 'b':
                # \b matches a word boundary
                pass
            else:
                # Handle other escaped characters
                start.transitions[escape_char] = {end}
            stack.append(NFA(start, end))
        else:
            # Handle single characters
            start = State()
            end = State(is_accepting=True)
            start.transitions[token] = {end}
            stack.append(NFA(start, end))
    return stack.pop()

def nfa_to_dfa(nfa, alphabet):
    initial = frozenset(epsilon_closure(nfa.start))
    dfa_states = [initial]
    dfa_transitions = {}
    state_map = {initial: 0}
    accepting_states = []
    if any(s.is_accepting for s in initial):
        accepting_states.append(0)
    queue = [initial]
    state_counter = 1

    while queue:
        current = queue.pop(0)
        current_idx = state_map[current]
```

```
214
215         for symbol in alphabet:
216             moved = move(current, symbol)
217             closure = set()
218             for s in moved:
219                 closure.update(epsilon_closure(s))
220             closure_frozen = frozenset(closure)
221             if not closure_frozen:
222                 continue
223             if closure_frozen not in state_map:
224                 state_map[closure_frozen] = state_counter
225                 dfa_states.append(closure_frozen)
226                 if any(s.is_accepting for s in closure_frozen):
227                     accepting_states.append(state_counter)
228                 queue.append(closure_frozen)
229                 state_counter += 1
230             dfa_transitions[(current_idx, symbol)] = state_map[closure_frozen]
231
232     # Handle dead state
233     dead_state = None
234     all_states = list(state_map.values())
235     for state_idx in all_states.copy():
236         for symbol in alphabet:
237             if (state_idx, symbol) not in dfa_transitions:
238                 if dead_state is None:
239                     dead_state = state_counter
240                     state_counter += 1
241                     for s in alphabet:
242                         dfa_transitions[(dead_state, s)] = dead_state
243                 dfa_transitions[(state_idx, symbol)] = dead_state
244     if dead_state is not None:
245         for symbol in alphabet:
246             dfa_transitions.setdefault((dead_state, symbol), dead_state)
247
248     # Build DFA table
249     dfa_table = {}
250     max_state = state_counter if dead_state is not None else state_counter
251     for state in range(max_state):
252         dfa_table[state] = {}
253         for symbol in alphabet:
254             dfa_table[state][symbol] = dfa_transitions.get((state, symbol), dead_state)
255
256     return dfa_table, accepting_states
257
258 def validate_regex(regex, alphabet):
259     valid_symbols = set(alphabet) | {'|', '*', '+', '?', '(', ')', '[', ']', '\\', '.'}
260     stack = []
261     i = 0
262     while i < len(regex):
263         c = regex[i]
264         if c == '\\':
265             # Handle escaped characters
266             if i + 1 >= len(regex):
267                 return False, "Invalid escape sequence"
```

```
            next_char = regex[i + 1]
            if next_char not in {'d', 's', 'w', 'b', 'D', 'S', 'W', '\\', '.', '|', '*', '+', '?', '(', ')', '[', ']'}:
                return False, f"Invalid escape sequence '\\{next_char}'"
            i += 2
        elif c == '[':
            # Handle character classes
            j = i + 1
            while j < len(regex) and regex[j] != ']':
                j += 1
            if j >= len(regex):
                return False, "Unclosed character class"
            i = j + 1
        elif c == '(':
            stack.append(c)
            i += 1
        elif c == ')':
            if not stack:
                return False, "Unbalanced parentheses"
            stack.pop()
            i += 1
        elif c not in valid_symbols:
            return False, f"Invalid character '{c}' in regular expression"
        else:
            i += 1
    if stack:
        return False, "Unbalanced parentheses"
    return True, ""
from graphviz import Digraph
# def generate_dfa_diagram(dfa_table, accepting_states, alphabet, output_file='dfa'):
#     # Create a directed graph
#     dot = Digraph(comment='DFA State Transition Diagram')

#     # Add states
#     for state in dfa_table:
#         if state in accepting_states:
#             # Double circle for accepting states
#             dot.node(f'q{state}', shape='doublecircle')
#         else:
#             # Single circle for non-accepting states
#             dot.node(f'q{state}', shape='circle')

#     # Add transitions
#     for state, transitions in dfa_table.items():
#         for symbol, next_state in transitions.items():
#             dot.edge(f'q{state}', f'q{next_state}', label=symbol)

#     # Render the graph to a file
#     dot.render(output_file, format='png', cleanup=False)
def generate_dfa_diagram(dfa_table, accepting_states, alphabet, output_file='dfa'):

    dot = Digraph(comment='Colorful DFA State Transition Diagram')

```

```
    state_colors = ['lightcoral', 'lightblue', 'lightgreen', 'plum', 'lightsalmon', 'lightgoldenrodyellow']
    edge_colors = ['red', 'blue', 'green', 'purple', 'orange', 'gold']

    for i, state in enumerate(dfa_table):
        color = state_colors[i % len(state_colors)]
        if state in accepting_states:

            dot.node(f'q{state}', shape='doublecircle', style='filled', fillcolor=color)
        else:

            dot.node(f'q{state}', shape='circle', style='filled', fillcolor=color)

    # Add transitions with different colors and styles
    for state, transitions in dfa_table.items():
        for symbol, next_state in transitions.items():
            edge_color = edge_colors[hash(symbol) % len(edge_colors)]
            dot.edge(f'q{state}', f'q{next_state}', label=symbol, color=edge_color)

    # Render the graph to a file
    dot.render(output_file, format='png')

@app.route('/generate-dfa', methods=['POST'])
def generate_dfa():
    data = request.json
    alphabet = data['alphabet']
    regex = data['regex']

    is_valid, msg = validate_regex(regex, alphabet)
    if not is_valid:
        return jsonify({"error": msg}), 400

    modified_regex = insert_concat_operators(regex)
    try:
        postfix = shunting_yard(modified_regex)
    except ValueError as e:
        return jsonify({"error": str(e)}), 400

    try:
        nfa = build_nfa(postfix)
    except Exception as e:
        return jsonify({"error": str(e)}), 400

    dfa_table, accepting_states = nfa_to_dfa(nfa, alphabet)

    # Generate the DFA diagram
    generate_dfa_diagram(dfa_table, accepting_states, alphabet)

    return jsonify({
        "dfa_table": dfa_table,
        "accepting_states": accepting_states,
        "diagram_url": "./dfa.png"  # Serve the diagram image
    })

```

DFA State Transition Diagram



Enter a string to simulate:

abbabbb

Simulate String

String "abbabbb" is ACCEPTED by the DFA.

DFA Generator from Regular Expression

Enter Alphabet (comma-separated, e.g., a,b):

a,b

Enter Regular Expression:

(a|b)*

Generate DFA

DFA State Table

State	a	b	Accepting?
q0	q1	q2	✓ Yes
q1	q1	q2	✓ Yes
q2	q1	q2	✓ Yes