Bhartiya Vidya Bhavan's, Sardar Patel Institute of Technology
(An Autonomous Institute Affiliated to University of Mumbai)
Bhavan's Campus, Munshi Nagar, Andheri (West),
Mumbai-400058-India

Academic Year: 2025-26 **Class: TE Comp** **Div: B** **Batch:- D**
Subject: SPCC **Sem: 6**

Name	**Raj Kalpesh Mathuria**
UID no.	**2023300139**
Aim:	**Optimization of DFA**

Theory:

The goal of this assignment is to turn a text pattern (Regular Expression) into a state machine (DFA) that a computer can run. To make this work, we use a specific algorithm (often called the Syntax Tree Method):

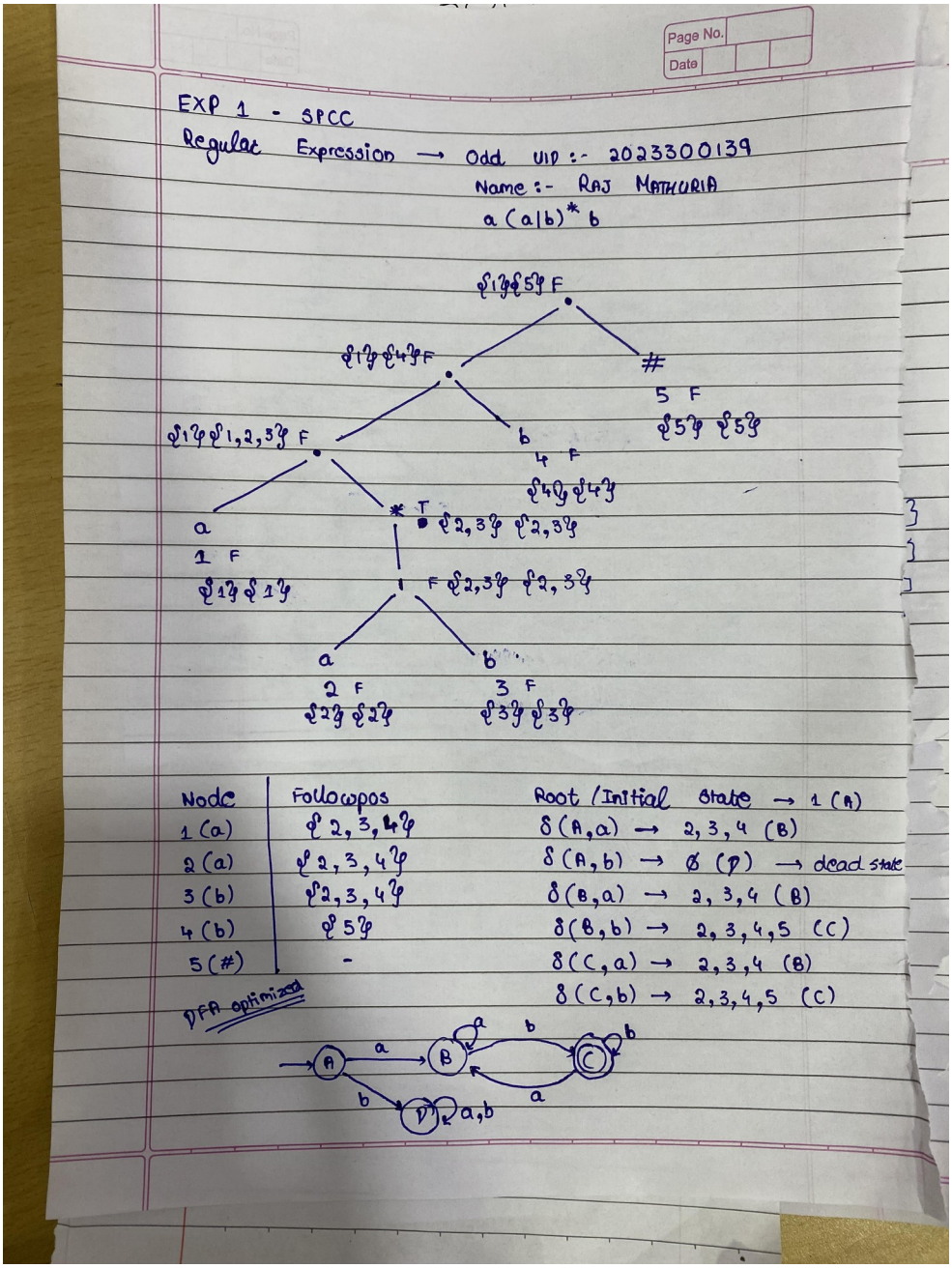
1. Augmentation: We first attach a special end-marker # to the expression (e.g., (a|b)* becomes (a|b)*#). This tells the machine exactly when a valid string has finished.

2. The Four Functions: To build the logic, we calculate four specific properties for every node in our tree:

- Nullable: Can this part of the regex be an empty string?
- Firstpos: If we enter this part of the tree, what are the first characters we might see?
- Lastpos: What are the possible "final" characters of this section?
- Followpos: The most crucial function. It answers the question: "After reading character X, what valid characters can come next?"

Building the States

We don't guess the states; we calculate them. The initial state is simply the firstpos of the whole tree. From there, we use followpos to determine where to go next based on input. Each "State" in our DFA is actually a set of tree positions grouped together.

Manually Solved Problem	
Code	#include <iostream> #include <stack> #include <string> #include <vector> #include <set> #include <map> #include <algorithm> #include <iomanip>

```
using namespace std;

set<int> followpos[100];
map<int, char> posToSymbol;
int leafNodeCount = 0;

struct Node {
char value;
int position;
Node *left, *right;
bool nullable;
set<int> firstpos;
set<int> lastpos;

Node(char val) : value(val), position(0), left(nullptr), right(nullptr),
nullable(false) {}
};

bool isOperator(char c) {
return c == '*' || c == '|' || c == '.';
}

bool isOperand(char c) {
return !isOperator(c) && c != '(' && c != ')';
}

string addConcatSymbol(string regex) {
string res = "";
for (int i = 0; i < regex.length(); i++) {
char c1 = regex[i];
res += c1;
if (i + 1 < regex.length()) {
char c2 = regex[i + 1];
bool c1IsOp = isOperand(c1);
bool c2IsOp = isOperand(c2);
if ((c1IsOp && c2IsOp) ||
(c1IsOp && c2 == '(') ||
(c1 == '*' && c2IsOp) ||
(c1 == '*' && c2 == '(') ||
(c1 == ')' && c2IsOp) ||
(c1 == ')' && c2 == '(')) {
res += '.';
}
```

```
}
}
return res;
}

int precedence(char c) {
if (c == '*') return 3;
if (c == '.') return 2;
if (c == '|') return 1;
return 0;
}

string infixToPostfix(string regex) {
string postfix = "";
stack<char> s;
for (char c : regex) {
if (isOperand(c)) {
postfix += c;
} else if (c == '(') {
s.push(c);
} else if (c == ')') {
while (!s.empty() && s.top() != '(') {
postfix += s.top();
s.pop();
}
s.pop();
} else {
while (!s.empty() && precedence(s.top()) >= precedence(c)) {
postfix += s.top();
s.pop();
}
s.push(c);
}
}
while (!s.empty()) {
postfix += s.top();
s.pop();
}
return postfix;
}

Node* buildTree(string postfix) {
stack<Node*> st;
leafNodeCount = 0;
```

	for (char c : postfix) {
	if (isOperand(c)) {
	Node* node = new Node(c);
	node->position = ++leafNodeCount;
	posToSymbol[node->position] = c;
	node->nullable = false;
	node->firstpos.insert(node->position);
	node->lastpos.insert(node->position);
	st.push(node);
	} else if (c == '*') {
	Node* node = new Node(c);
	node->left = st.top(); st.pop();
	node->nullable = true;
	node->firstpos = node->left->firstpos;
	node->lastpos = node->left->lastpos;
	st.push(node);
	} else {
	Node* node = new Node(c);
	node->right = st.top(); st.pop();
	node->left = st.top(); st.pop();
	if (c == '|') {
	node->nullable = node->left->nullable || node->right->nullable;
	node->firstpos = node->left->firstpos;
	node->firstpos.insert(node->right->firstpos.begin(), node->right-
	>firstpos.end());
	node->lastpos = node->left->lastpos;
	node->lastpos.insert(node->right->lastpos.begin(), node->right-
	>lastpos.end());
	} else if (c == '.') {
	node->nullable = node->left->nullable && node->right->nullable;
	if (node->left->nullable) {
	node->firstpos = node->left->firstpos;
	node->firstpos.insert(node->right->firstpos.begin(), node->right-
	>firstpos.end());
	} else {
	node->firstpos = node->left->firstpos;
	}
	if (node->right->nullable) {
	node->lastpos = node->left->lastpos;
	node->lastpos.insert(node->right->lastpos.begin(), node->right-
	>lastpos.end());
	} else {
	node->lastpos = node->right->lastpos;
	}

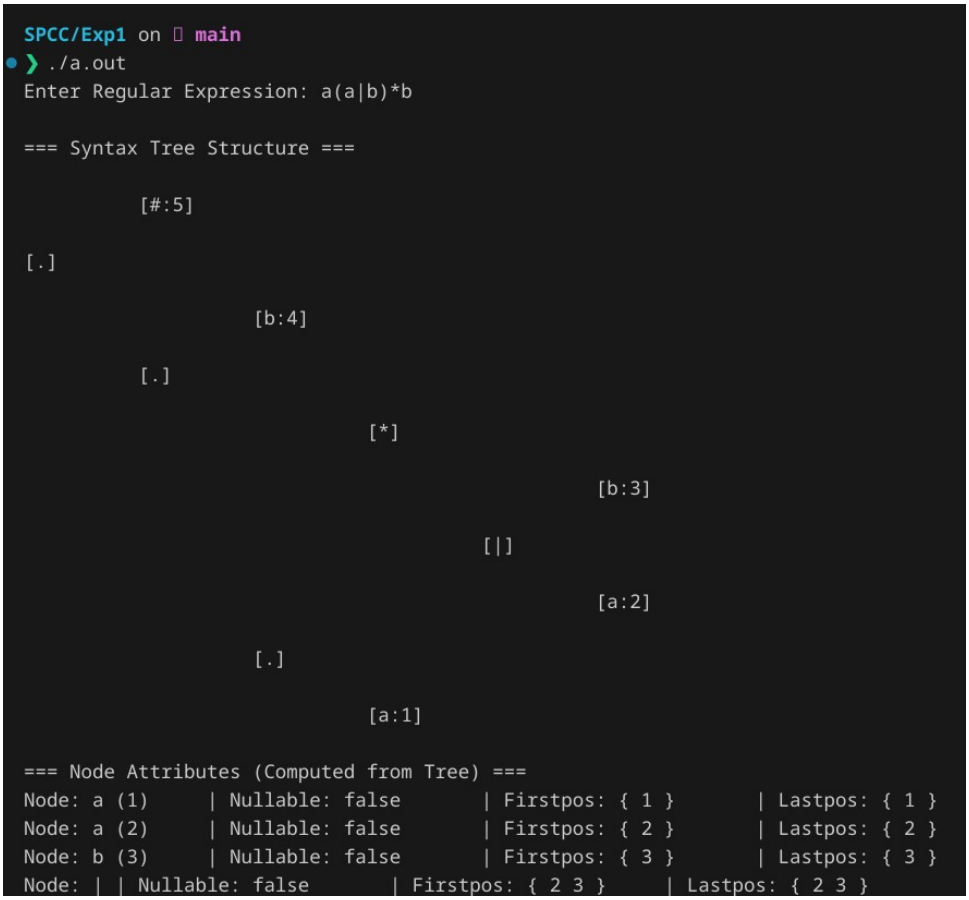
```
}
st.push(node);
}
}
return st.top();
}

void computeFollowpos(Node* node) {
if (!node) return;
if (node->value == '.') {
for (int i : node->left->lastpos) {
followpos[i].insert(node->right->firstpos.begin(), node->right-
>firstpos.end());
}
}
else if (node->value == '*') {
for (int i : node->lastpos) {
followpos[i].insert(node->firstpos.begin(), node->firstpos.end());
}
}
computeFollowpos(node->left);
computeFollowpos(node->right);
}

void printSet(set<int> s) {
cout << "{ ";
for (int i : s) cout << i << " ";
cout << "}";
}

void printTreeDetails(Node* node) {
if (!node) return;
printTreeDetails(node->left);
printTreeDetails(node->right);
cout << "Node: " << node->value;
if(isOperand(node->value)) cout << " (" << node->position << ")";
cout << "\t| Nullable: " << (node->nullable ? "true" : "false");
cout << "\t| Firstpos: "; printSet(node->firstpos);
cout << "\t| Lastpos: "; printSet(node->lastpos);
cout << endl;
}

void printPrettyTree(Node* root, int space = 0) {
const int COUNT = 10;
```

```
if (root == nullptr) return;
space += COUNT;
printPrettyTree(root->right, space);
cout << endl;
for (int i = COUNT; i < space; i++) cout << " ";
cout << "[" << root->value;
if (isOperand(root->value)) cout << ":" << root->position;
cout << "]" << "\n";
printPrettyTree(root->left, space);
}

void buildDFA(Node* root, vector<char> alphabet) {
map<set<int>, int> stateMapping;
vector<set<int>> Dstates;
vector<vector<int>> transitionTable;
set<int> startState = root->firstpos;
Dstates.push_back(startState);
stateMapping[startState] = 0;
int processedCount = 0;
while (processedCount < Dstates.size()) {
set<int> currentSet = Dstates[processedCount];
int currentStateID = processedCount;
processedCount++;
vector<int> row;
for (char sym : alphabet) {
set<int> newSet;
for (int p : currentSet) {
if (posToSymbol[p] == sym) {
newSet.insert(followpos[p].begin(), followpos[p].end());
}
}
if (newSet.empty()) {
row.push_back(-1);
} else {
if (stateMapping.find(newSet) == stateMapping.end()) {
stateMapping[newSet] = Dstates.size();
Dstates.push_back(newSet);
}
row.push_back(stateMapping[newSet]);
}
}
transitionTable.push_back(row);
}
cout << "\n\n=== DFA State Transition Table ===\n\n";
```

```
cout << setw(10) << "State" << " | ";
for (char c : alphabet) {
cout << setw(10) << c << " | ";
}
cout << "\n" << string(10 + (alphabet.size() * 13), '_') << "\n";
for (int i = 0; i < transitionTable.size(); i++) {
bool isFinal = false;
for (int p : Dstates[i]) {
if (posToSymbol[p] == '#') {
isFinal = true;
break;
}
}
string stateLabel = (isFinal ? "*" : "") + to_string(i);
cout << setw(10) << stateLabel << " | ";
for (int nextState : transitionTable[i]) {
if (nextState == -1) cout << setw(10) << "-" << " | ";
else cout << setw(10) << nextState << " | ";
}
cout << "\n";
}
cout << "\n(* denotes Accepting State)\n";
}

int main() {
string regex;
cout << "Enter Regular Expression: ";
cin >> regex;
regex = "(" + regex + ")#";
string explicitRegex = addConcatSymbol(regex);
string postfix = infixToPostfix(explicitRegex);
Node* root = buildTree(postfix);
cout << "\n=== Syntax Tree Structure ===\n";
printPrettyTree(root);
cout << "\n=== Node Attributes (Computed from Tree) ===\n";
printTreeDetails(root);
computeFollowpos(root);
cout << "\n=== Followpos Table ===\n";
for (int i = 1; i <= leafNodeCount; i++) {
cout << "Position " << i << " (" << posToSymbol[i] << "):\t";
printSet(followpos[i]);
cout << endl;
}
set<char> symbols;
```

```
for(int i=1; i<=leafNodeCount; i++) {
if(posToSymbol[i] != '#') symbols.insert(posToSymbol[i]);
}
vector<char> alphabet(symbols.begin(), symbols.end());
buildDFA(root, alphabet);
return 0;
}
```

Snapshot of Output



```
SPCC/Exp1 on  main
❯ ./a.out
Enter Regular Expression: a(a|b)*b

=== Syntax Tree Structure ===

          [#:5]

[.]

                    [b:4]

          [.]

                              [*]

                                                  [b:3]

                                        [|]

                                                  [a:2]

                    [.]

                              [a:1]

=== Node Attributes (Computed from Tree) ===
Node: a (1)     | Nullable: false       | Firstpos: { 1 }       | Lastpos: { 1 }
Node: a (2)     | Nullable: false       | Firstpos: { 2 }       | Lastpos: { 2 }
Node: b (3)     | Nullable: false       | Firstpos: { 3 }       | Lastpos: { 3 }
Node: | | Nullable: false       | Firstpos: { 2 3 }     | Lastpos: { 2 3 }
```

```
=== Node Attributes (Computed from Tree) ===
Node: a (1)     | Nullable: false       | Firstpos: { 1 }       | Lastpos: { 1 }
Node: a (2)     | Nullable: false       | Firstpos: { 2 }       | Lastpos: { 2 }
Node: b (3)     | Nullable: false       | Firstpos: { 3 }       | Lastpos: { 3 }
Node: | | Nullable: false       | Firstpos: { 2 3 }     | Lastpos: { 2 3 }
Node: * | Nullable: true        | Firstpos: { 2 3 }     | Lastpos: { 2 3 }
Node: . | Nullable: false       | Firstpos: { 1 }       | Lastpos: { 1 2 3 }
Node: b (4)     | Nullable: false       | Firstpos: { 4 }       | Lastpos: { 4 }
Node: . | Nullable: false       | Firstpos: { 1 }       | Lastpos: { 4 }
Node: # (5)     | Nullable: false       | Firstpos: { 5 }       | Lastpos: { 5 }
Node: . | Nullable: false       | Firstpos: { 1 }       | Lastpos: { 5 }

=== Followpos Table ===
Position 1 (a): { 2 3 4 }
Position 2 (a): { 2 3 4 }
Position 3 (b): { 2 3 4 }
Position 4 (b): { 5 }
Position 5 (#): { }


=== DFA State Transition Table ===

     State |          a |          b |
_____________________________________
         0 |          1 |          - |
         1 |          1 |          2 |
        *2 |          1 |          2 |

(* denotes Accepting State)
```

conclusion

Implementing this converter bridged the gap between abstract theory and practical coding. By constructing the syntax tree and coding the logic for firstpos, lastpos, and followpos, I was able to see exactly how a compiler understands patterns. Main benefits are as follows:

- No Backtracking: Unlike an NFA, the DFA we generated is efficient. It knows exactly where to go for every input without guessing.
- The Power of Trees: Converting the linear regex string into a tree structure was the critical step that made the complex logic manageable.
- Automation: The final output—the State Transition Table—is the proof of concept. It represents a fully functioning machine generated entirely from a string of rules.

This project demonstrated that lexical analysis isn't magic; it's a systematic process of breaking down rules and calculating the possible paths through them.