COMPILER LAB

REPORT ASSIGNMENT-2

NAME: SOHAM LAHIRI

CLASS: BCSE UG-III 6TH SEMESTER

ROLL NO: 002210501107

GROUP: A3

SUBMISSION DATE: 21/02/2025

Problem Statement: Write a code that will accept a regular expression and generate a DFA.

This report provides an analysis of a C++ program that converts a regular expression (in postfix notation) into a **Deterministic Finite Automaton (DFA)**. The program constructs a syntax tree, computes first and last positions, determines follow positions, and builds the DFA.

Components of the Program:
The program consists of the following key components:

1. Class Definition: treenode
A class treenode is defined to represent nodes in the syntax tree of the regular expression.
Attributes:

- char alpha: Character at the node.
- int pos: Position of the character in the expression.
- bool nullable: Indicates whether the node represents a nullable subexpression.
- vector<int> fpos, lpos: Vectors storing first and last positions in the node.
- treenode *left, *right: Pointers to left and right children.

Constructors:

- Default constructor.
- Parameterized constructor initializing alpha and pos, with children set to NULL.

2. Utility Functions

- **isOperator(char ch):** Determines whether a character is an operator (|, /, *, .).
- **printVector(const vector<int>& v):** Prints the elements of a vector.
- **postOrder(treenode *root):** Performs a post-order traversal to print details of each node in the syntax tree.
- **Overloaded + operator:** Computes the union of two vectors and ensures uniqueness.

3. Follow Position Computation
A global array vector<int> followPos[30] stores the follow positions for each position in the regular expression.

4. DFA Construction (buildDFA)
This function constructs a DFA from the syntax tree.

Key Data Structures:

- vector<int> Dstate[30]: Stores DFA states.
- map<int, char> intMapsChar: Maps positions to characters.
- map<vector<int>, char> out_state: Maps state sets to DFA state labels.
- map<vector<int>, map<char, vector<int>>> re_dfa: Stores DFA transitions.

Working of the Program:

The program follows a structured approach:

Step 1: Input Handling and Syntax Tree Construction

- The user inputs a regular expression in **postfix notation**.
- A stack-based approach constructs the syntax tree.
- Each character is processed as follows:
    - If it's an operand (symbol), a new treenode is created with position posIndex.
    - If it's an operator (*, ., |), a new treenode is created with left and/or right children from the stack.
    - For * (Kleene star), follow positions are updated.
    - For . (concatenation), left lpos are linked to right fpos.

Step 2: Syntax Tree Traversal and Computation

- A post-order traversal prints details of each node: char, pos, nullable, firstpos, and lastpos.

Step 3: Follow Position Calculation

- Follow positions are stored in followPos[i].
- For each node with a concatenation (.), the last position of the left child is mapped to the first position of the right child.
- For Kleene star (*), last positions are mapped to first positions.

Step 4: DFA Construction

- DFA states are initialized using root->fpos.
- New states are assigned unique labels (A, B, C, ...).
- The program iterates over existing states, processing transitions for each symbol.
- If a new transition set is found, it is assigned a new state.
- The constructed DFA is displayed in a readable format.

4. Output and Interpretation

The program generates the following outputs:

1. **Syntax Tree Details:**
    - Displays each node’s position, nullability, first and last positions.
2. **Follow Position Table:**
    - Shows follow positions for each position in the regular expression.
3. **DFA Transition Table:**
    - Represents the DFA in terms of states and transitions for each input symbol.

Key Features:

1. **Postfix Expression Handling**
    - Many implementations use infix parsing, while this directly **builds the tree from postfix notation**.

2. **Vector Overloading for FollowPos Computation**
    - The custom **+ operator for vectors** ensures that unique elements are merged without duplicates.

3. **Efficient DFA State Generation**
    - The DFA **avoids redundant states** by tracking seen states in out_state.

4. **Detailed Debugging Output**
    - Shows **each node's properties** (nullable, firstpos, lastpos).
    - Prints **Follow Position table** clearly.
    - Outputs **formatted DFA transitions**.

Implementation:

ASSIGNMENT2.cpp:

```
#include <bits/stdc++.h>
#include <iomanip>  // Include for proper formatting

using namespace std;

class treenode {
public:
    char alpha;
    int pos;
    bool nullable;
    vector<int> fpos, lpos;
    treenode *left, *right;

    treenode() {}

    treenode(char ch, int p) {
        alpha = ch;
        pos = p;
        left = right = NULL;
        fpos.clear();
        lpos.clear();
    }
};

// Function to check if a character is an operator
bool isOperator(char ch) {
    return (ch == '|' || ch == '/' || ch == '*' || ch == '.');
}

// Function to print a vector
void printVector(const vector<int>& v) {
    for (int num : v) cout << num << " ";
}

// Post-order traversal with formatted output
void postOrder(treenode *root) {
    if (!root) return;
    postOrder(root->left);
    postOrder(root->right);

    cout << setw(5) << root->alpha
         << setw(10) << root->pos
```

```
        << setw(10) << root->nullable
        << setw(15) << "{ ";
    printVector(root->fpos);
    cout << " }" << setw(15) << "{ ";
    printVector(root->lpos);
    cout << " }" << endl;
}

int posIndex = 0, followPosIndex = 0;
vector<int> followPos[30], Dstate[30];
map<int, char> intMapsChar;
set<char> charSet;

// Overloading `+` operator for vector union
vector<int> operator+(vector<int> a, vector<int> b) {
    a.insert(a.end(), b.begin(), b.end());
    sort(a.begin(), a.end());
    a.erase(unique(a.begin(), a.end()), a.end());
    return a;
}

// DFA construction
void buildDFA(treenode *root, string input) {
    int num_state = 1, cur_state = 1;
    char ch = 'A';
    vector<int> temp;
    map<vector<int>, char> out_state;
    map<vector<int>, map<char, vector<int>>> re_dfa;

    Dstate[num_state++] = root->fpos;
    out_state[root->fpos] = ch++;

    while (1) {
        for (char i : input) {
            for (int j : Dstate[cur_state]) {
                if (intMapsChar[j] == i)
                    temp = temp + followPos[j];
                if (out_state[temp] == 0 && !temp.empty()) {
                    out_state[temp] = ch++;
                    Dstate[num_state++] = temp;
                }
            }
            re_dfa[Dstate[cur_state]][i] = temp;
            temp.clear();
        }
        if (cur_state == num_state - 1)
```

```
            break;
        cur_state++;
    }

    // Display DFA states
    cout << "\n\nDeterministic Finite Automaton (DFA):\n\n";
    for (const auto& a : re_dfa) {
        cout << "{ ";
        for (int b : a.first)
            cout << b << " ";
        cout << "} ";
        for (const auto& b : a.second) {
            cout << " on input: " << b.first << " { ";
            for (int c : b.second)
                cout << c << " ";
            cout << "} ";
        }
        cout << endl;
    }
}

int main() {
    treenode *tnode;
    stack<treenode *> s;
    string str, input;

    cout << "\nInput a Regular Expression (in postfix form): ";
    cin >> str;

    for (char c : str) {
        if (!isOperator(c)) {
            posIndex++;
            if (c != '#') {
                followPosIndex++;
                intMapsChar[followPosIndex] = c;
                charSet.insert(c);
            }
            tnode = new treenode(c, posIndex);
            tnode->nullable = false;
            tnode->fpos.push_back(posIndex);
            tnode->lpos.push_back(posIndex);
        } else if (c == '*') {
            tnode = new treenode(c, 0);
            tnode->left = s.top(), s.pop();
            tnode->nullable = true;
            tnode->fpos = tnode->left->fpos;
```

```
            tnode->lpos = tnode->left->lpos;
            for (int i : tnode->lpos)
                followPos[i] = followPos[i] + tnode->fpos;
        } else if (c == '.') {
            tnode = new treenode(c, 0);
            tnode->right = s.top(), s.pop();
            tnode->left = s.top(), s.pop();
            tnode->nullable = tnode->right->nullable && tnode->left->nullable;
            if (tnode->left->nullable)
                tnode->fpos = tnode->right->fpos + tnode->left->fpos;
            else
                tnode->fpos = tnode->left->fpos;
            if (tnode->right->nullable)
                tnode->lpos = tnode->right->lpos + tnode->left->lpos;
            else
                tnode->lpos = tnode->right->lpos;
            for (int i : tnode->left->lpos)
                followPos[i] = followPos[i] + tnode->right->fpos;
        } else {
            tnode = new treenode(c, 0);
            tnode->right = s.top(), s.pop();
            tnode->left = s.top(), s.pop();
            tnode->nullable = tnode->right->nullable && tnode->left->nullable;
            tnode->fpos = tnode->right->fpos + tnode->left->fpos;
            tnode->lpos = tnode->right->lpos + tnode->left->lpos;
        }
        s.push(tnode);
    }

    for (char t : charSet)
        input.push_back(t);

    // Print Header
    cout << "\n-----------------------------------------------------------------------------------------";
    cout << "\n\nNODE" << setw(10) << "Pos" << setw(10) << "Nullable"
         << setw(20) << "First Pos" << setw(20) << "Last Pos" << endl;

    // Print Parse Tree Nodes
    postOrder(tnode);

    cout << "\n-----------------------------------------------------------------------------------------";
    cout << "\n\nFollow Position Table:\n";

    // Print Follow Position Table
    for (int i = 1; i <= followPosIndex; i++) {
        cout << setw(5) << i
```

```
            << setw(5) << intMapsChar[i]
            << setw(10) << "{ ";
        printVector(followPos[i]);
        cout << " }" << endl;
    }

    cout << "\n----------------------------------------------------------------------------------------";
    buildDFA(tnode, input);
    cout << "\n----------------------------------------------------------------------------------------";

    return 0;
}
```

Output:

```
Input a Regular Expression (in postfix form): ab.c*|

-------------------------------------------------------------------------------------

NODE            Position       Nullable        First position              Last position
a               1               0               { 1  }                          { 1  }
b               2               0               { 2  }                          { 2  }
.               0               0               { 1  }                          { 2  }
c               3               0               { 3  }                          { 3  }
*               0               1               { 3  }                          { 3  }
|               0               0               { 1 3  }                                  { 2 3  }

-------------------------------------------------------------------------------------

Follow Position Table :
1               a               { 2  }
2               b               {  }
3               c               { 3  }

-------------------------------------------------------------------------------------

Deteministic Finite Automata :

{ 1 3  }  on input: a { 2  }  on input: b {  }  on input: c { 3  }
{ 2  }  on input: a {  }  on input: b {  }  on input: c {  }
{ 3  }  on input: a {  }  on input: b {  }  on input: c { 3  }

-------------------------------------------------------------------------------------
```