BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

**Department of Computer Engineering**

## Course - System Programming and Compiler Construction (SPCC)

| | |
|---|---|
| **Aim** | To create a Parse Tree and compute the sets firstpos, lastpos, and followpos for each node in the Parse Tree for a given regular expression or grammar. |
| **Objective** | The objective is to construct a Parse Tree from a given regular expression, employing parsing techniques. Calculate firstpos, lastpos, and followpos sets for each node in the Parse Tree, enabling efficient analysis of positions where symbols can begin, end, and follow within the parsed expression, contributing to language parsing and analysis. |
| **Theory** | **Deterministic Finite Automaton (DFA):**<br><br>A Deterministic Finite Automaton (DFA) is a theoretical construct used in computer science and formal language theory to model and recognize patterns within strings of symbols. It consists of five components:<br><br>• States (Q): A finite set of states representing different configurations or conditions of the automaton during its operation.<br>• Alphabet (Σ): A finite set of input symbols that the DFA recognizes. These symbols are the building blocks of the strings being processed.<br>• Transition Function (δ): A function that maps a state and an input symbol to a new state. It defines the behavior of the DFA as it processes symbols from the input string.<br>• Start State (q0): The initial state from which the DFA begins its operation.<br>• Accepting States (F): A subset of states that are designated as accepting or final states. If the DFA reaches one of these states after processing the entire input string, it recognizes the string as belonging to the language described by the DFA.[1]<br><br>DFA operates by consuming input symbols one at a time and transitioning between states according to the transition function. The process continues until the DFA either reaches an accepting state, indicating that the input string matches the pattern described by the DFA, or enters a non-accepting state, indicating that the string does not match the pattern.<br><br>DFAs are particularly useful for pattern matching tasks, such as lexical analysis in compilers, string searching algorithms, and regular expression evaluation. They offer a deterministic and efficient approach to recognizing patterns in strings, making them a fundamental concept in theoretical computer science and practical algorithm design.<br><br>Converting a Regular Expression to a Deterministic Finite Automaton (DFA):<br>• It is a critical process in automata theory and compiler design, enabling the efficient matching of patterns described by regular expressions.<br>• This conversion involves a series of steps, starting with the construction of a Non-deterministic Finite Automaton (NFA) that directly corresponds to the structure of the regular expression.<br>• The NFA, which allows multiple transitions for a single input symbol and can include |

ε-transitions (transitions that do not consume any input), captures the nondeterministic nature of regular expression pattern matching.

- Subsequently, the NFA is transformed into an equivalent DFA using the subset construction algorithm, which systematically eliminates nondeterminism by creating states in the DFA that represent sets of NFA states.
- The resulting DFA is a deterministic system that can efficiently process input strings to determine match with the pattern defined by the original regular expression, leveraging the DFA's deterministic nature for direct and efficient pattern matching.[2]

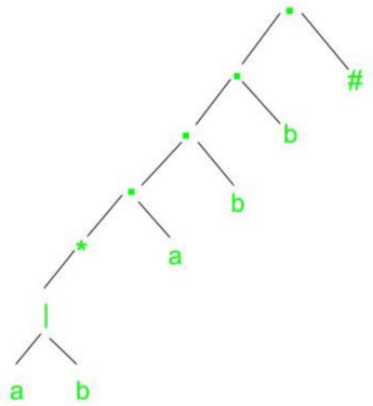
But to avoid this two-step procedure, the other way round is to directly construct a DFA for the given expression.
In order to construct a DFA directly from a regular expression, we need to follow the steps listed below:

Example: Let's consider the regular expression r = (a|b)*abb

**STEP 1**: Begin by creating the augmented regular expression by appending a unique right-end marker '#' to the given expression. This step ensures that the accepting state for r has a transition on '#' to mark it as a crucial state in the NFA for r#.

So, r' = (a|b)*abb#

**STEP 2**: Build the syntax tree for r#.




*Syntax tree for (a|b)*abb#*

**STEP 3:** Next we need to evaluate four functions nullable, firstpos, lastpos, and followpos.

1. nullable(n) is true for a syntax tree node n if and only if the regular expression represented by n has € in its language.
2. firstpos(n) gives the set of positions that can match the first symbol of a string generated by the subexpression rooted at n.
3. lastpos(n) gives the set of positions that can match the last symbol of a string generated by the subexpression rooted at n.
4. We refer to an interior node as a cat-node, or-node, or star-node if it is labeled by a concatenation, | or * operator, respectively.

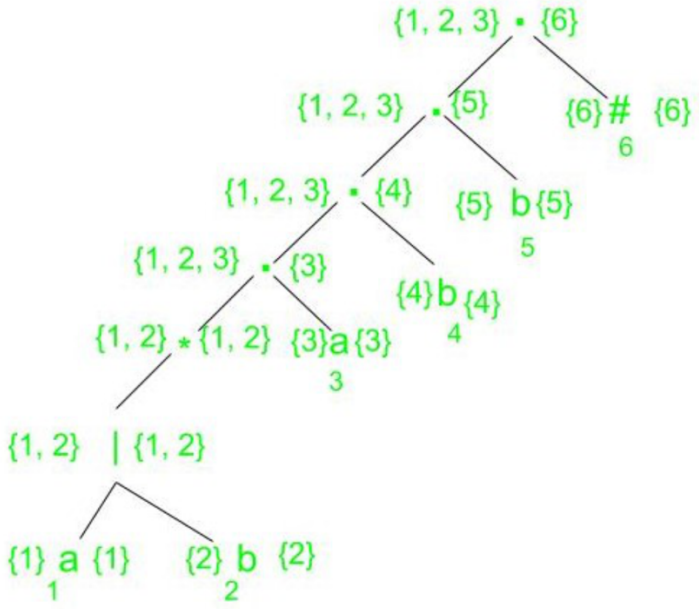
**Rules for computing nullable, firstpos, and lastpos:**

| Node n | nullable(n) | firstpos(n) | lastpos(n) |
|---|---|---|---|
| **n is a leaf node labeled €** | true | ⌀ | ⌀ |
| **n is a leaf node labelled with position i** | false | { i } | { i } |
| **n is an or node with left child c1 and right child c2** | nullable(c1) or nullable(c2) | firstpos(c1) ∪ firstpos(c2) | lastpos(c1) ∪ lastpos(c2) |
| **n is a cat node with left child c1 and right child c2** | nullable(c1) and nullable(c2) | If nullable(c1) then firstpos(c1) ∪ firstpos(c2) else firstpos(c1) | If nullable(c2) then lastpos(c2) ∪ lastpos(c1) else lastpos(c2) |
| **n is a star node with child node c1** | true | firstpos(c1) | lastpos(c1) |

**Rules for computing followpos:**

1. If n is a cat-node with left child c1 and right child c2 and i is a position in lastpos(c1), then all positions in firstpos(c2) are in followpos(i).
2. If n is a star-node and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i).

3. Now that we have seen the rules for computing firstpos and lastpos, we now proceed to calculate the values of the same for the syntax tree of the given regular expression (a|b)*abb#.




Let us now compute the followpos bottom up for each node in the syntax tree.

| NODE | followpos |
|---|---|
| 1 | {1, 2, 3} |
| 2 | {1, 2, 3} |
| 3 | {4} |
| 4 | {5} |
| 5 | {6} |
| 6 | ø |

**STEP 4:** Now we construct Dstates, the set of states of DFA D and Dtran, the transition table for D. The start state of DFA D is firstpos(root) and the accepting states are all those containing the position associated with the endmarker symbol #.

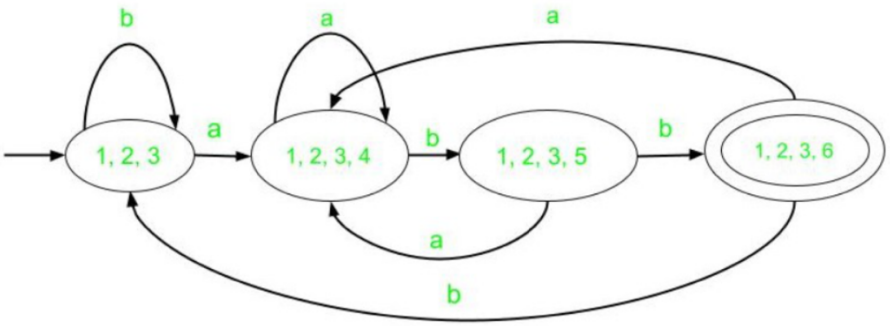
According to our example, the firstpos of the root is {1, 2, 3}. Let this state be A and consider the input symbol a. Positions 1 and 3 are for a, so let B = followpos(1) ∪ followpos(3) = {1, 2, 3, 4}. Since this set has not yet been seen, we set Dtran[A, a] := B.

When we consider input b, we find that out of the positions in A, only 2 is associated with b, thus we consider the set followpos(2) = {1, 2, 3}. Since this set has already been seen before, we do not add it to Dstates but we add the transition Dtran[A, b]:= A.

Continuing like this with the rest of the states, we arrive at the below transition table.

| | Input | |
|---|---|---|
| State | a | b |
| → A | B | A |
| B | B | C |
| C | B | D |
| D | B | A |

Here, A is the start state and D is the accepting state.
**STEP 5:** Finally we draw the DFA for the above transition table. The final DFA will be :




*DFA for (a|b)*abb*

**Implementation / Code**

```
import javax.swing.*;
import java.awt.*;
import java.util.*;
import java.util.List;
import java.util.Queue;

public class main {

    static class TreeNode {
```

```
        public char symbol;
        public Set<Integer> firstpos;
        public Set<Integer> lastpos;

        public int i;
        public boolean nullable;

        public TreeNode left;
        public TreeNode right;

        TreeNode() {
            symbol = ' ';
            i = 0;
            firstpos = new HashSet<>();
            lastpos = new HashSet<>();

            nullable = false;
            left = null;
            right = null;
        }

        TreeNode(char ch) {
            symbol = ch;
            i = 0;
            firstpos = new HashSet<>();
            lastpos = new HashSet<>();

            nullable = false;
            left = null;
            right = null;
        }

        TreeNode(char ch, int num) {
            symbol = ch;
            i = num;
            firstpos = new HashSet<>();
            lastpos = new HashSet<>();

            nullable = false;
            left = null;
            right = null;
        }

        public static boolean isOperand(char ch) {
            return ch == '|' || ch == '.' || ch == '*';
        }
```

```
        public static boolean isTerminal(char ch) {
            return !isOperand(ch) && ch != ')' && ch != '(';
        }

        public static boolean isLeaf(TreeNode node) {
            return node.left == null && node.right == null;
        }

        public void print() {
            if (isTerminal(symbol))
                System.out.println(symbol + " (" + i + ") " + "\nnullable = " +
nullable);
            else
                System.out.println(symbol + " " + "\nnullable = " + nullable);

            System.out.println("firstpos() " + firstpos.toString());
            System.out.println("lastpos() " + lastpos.toString());
            System.out.println();
        }
    }
    static void printTree(TreeNode root) {
        System.out.println("-----------------------------------------------------
-----------------------------");
        System.out.printf("| %-5s | %-14s | %-15s | %-9s | %-12s | %-8s |\n",
                "Value", "Left Child", "Right Child", "Nullable", "Firstpos",
"Lastpos");
        System.out.println("-----------------------------------------------------
-----------------------------");
        printTee(root);
        System.out.println("-----------------------------------------------------
-----------------------------");
    }

    static void printTee(TreeNode n) {
        if (n == null) {
            return;
        }
        System.out.printf("| %-5s | %-14s | %-15s | %-9s | %-12s | %-8s |\n",
                n.symbol, (n.left != null) ? n.left.symbol : "null",
                (n.right != null) ? n.right.symbol : "null", n.nullable,
                n.firstpos, n.lastpos);
        printTee(n.left);
        printTee(n.right);
    }

    static class ParseTreePanel extends JPanel {
        private TreeNode root;
```

```
        public ParseTreePanel(TreeNode root) {
            this.root = root;
        }

        private void drawTree(Graphics g, TreeNode node, int x, int y, int level,
int xOffset) {
            if (node != null) {
                // Draw the oval
                int ovalWidth = 30;
                int ovalHeight = 30;
                g.drawOval(x - ovalWidth / 2, y, ovalWidth, ovalHeight);

                // Draw the node symbol
                g.drawString(Character.toString(node.symbol), x - 3, y + 15);
                g.setColor(new Color(148, 0, 211));
                // Draw the list next to the node
                List<Integer> list = List.copyOf(node.firstpos);
                g.drawString(list.toString(), x + 25, y);
                g.setColor(Color.red);
                // Draw the lastpos below the node
                List<Integer> last = List.copyOf(node.lastpos);
                g.drawString(last.toString(), x - 30, y + ovalHeight + 12);
                g.setColor(Color.blue);
                // Draw 'T' or 'F' based on nullable
                g.drawString(node.nullable ? "T" : "F", x + 25, y + ovalHeight +
22);
                g.setColor(Color.black);
                // Draw lines and connect child nodes
                if (node.left != null) {
                    int childX = x - xOffset / 2;
                    int childY = y + 50;
                    g.drawLine(x, y + ovalHeight, childX, childY);
                    drawTree(g, node.left, childX, childY, level + 1, xOffset );
                }
                if (node.right != null) {
                    int childX = x + xOffset / 2;
                    int childY = y + 50;
                    g.drawLine(x, y + ovalHeight, childX, childY);
                    drawTree(g, node.right, childX, childY, level + 1, xOffset);
                }
            }
        }



        @Override
```

```
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            drawTree(g, root,getWidth()- getWidth()/3, 30, 0, getWidth() /6);
        }
    }

    public static int precedence(char ch) {
        if (ch == '(' || ch == ')')
            return 6;

        if (!TreeNode.isOperand(ch))
            return 0;

        if (ch == '*')
            return 5;
        if (ch == '.')
            return 4;

        return 3;
    }

    public static String toPostFix(String input) {
        Stack<Character> stack = new Stack<>();
        StringBuilder str = new StringBuilder();

        for (char ch : input.toCharArray()) {
            if (ch == '(') {
                stack.push(ch);
            } else if (ch == ')') {
                while (!stack.isEmpty() && stack.peek() != '(') {
                    str.append(stack.pop());
                }
                if (!stack.isEmpty() && stack.peek() == '(')
                    stack.pop();
            } else if (TreeNode.isOperand(ch)) {
                while (!stack.isEmpty() && TreeNode.isOperand(stack.peek())
                        && precedence(ch) <= precedence(stack.peek())) {
                    str.append(stack.pop());
                }
                stack.push(ch);
            } else {
                str.append(ch);
            }
        }

        while (!stack.isEmpty()) {
            str.append(stack.pop());
```

```
        }

        return str.toString();
    }

    public static String insertConcat(String input) {
        StringBuilder str = new StringBuilder();
        char[] arr = input.toCharArray();

        str.append(arr[0]);

        for (int i = 1; i < input.length(); i++) {
            boolean termTerm = TreeNode.isTerminal(arr[i - 1]) &&
TreeNode.isTerminal(arr[i]);
            boolean starTerm = arr[i - 1] == '*' && TreeNode.isTerminal(arr[i]);
            boolean cbraceTerm = arr[i - 1] == ')' && TreeNode.isTerminal(arr[i]);
            boolean cbraceObrace = arr[i - 1] == ')' && arr[i] == '(';
            boolean termObrace = TreeNode.isTerminal(arr[i - 1]) && arr[i] == '(';

            if (termTerm || cbraceObrace || starTerm || cbraceTerm || termObrace)
{
                str.append('.');
            }

            str.append(arr[i]);
        }

        return str.toString();
    }

    public static TreeNode createSyntaxTree(String postfix) {
        Stack<TreeNode> stack = new Stack<>();
        int termcount = 0;

        for (char ch : postfix.toCharArray()) {
            if (TreeNode.isTerminal(ch)) {
                stack.push(new TreeNode(ch, ++termcount));
            } else {
                TreeNode op = new TreeNode(ch);

                if (ch != '*') {
                    op.right = stack.pop();
                    op.left = stack.pop();
                } else {
                    op.left = stack.pop();
                }
```

```
                stack.push(op);
            }
        }

        return stack.pop();
    }

    public static void computeFunctions(TreeNode node) {
        if (node == null)
            return;

        computeFunctions(node.left);
        computeFunctions(node.right);

        if (TreeNode.isLeaf(node) && node.symbol == 'e') {
            node.nullable = true;
        } else if (TreeNode.isLeaf(node)) {
            node.nullable = false;
            node.firstpos.add(node.i);
            node.lastpos.add(node.i);
        } else if (node.symbol == '|') {
            node.nullable = node.left.nullable || node.right.nullable;
            node.firstpos.addAll(node.left.firstpos);
            node.firstpos.addAll(node.right.firstpos);

            node.lastpos.addAll(node.left.lastpos);
            node.lastpos.addAll(node.right.lastpos);
        } else if (node.symbol == '.') {
            node.nullable = node.left.nullable && node.right.nullable;
            if (node.left.nullable) {
                node.firstpos.addAll(node.left.firstpos);
                node.firstpos.addAll(node.right.firstpos);
            } else {
                node.firstpos.addAll(node.left.firstpos);
            }

            if (node.right.nullable) {
                node.lastpos.addAll(node.left.lastpos);
                node.lastpos.addAll(node.right.lastpos);
            } else {
                node.lastpos.addAll(node.right.lastpos);
            }
        } else {
            node.nullable = true;
            node.firstpos.addAll(node.left.firstpos);
            node.lastpos.addAll(node.left.lastpos);
        }
```

```
    }

    public static void inorder(TreeNode node) {
        if (node == null)
            return;

        inorder(node.left);
        node.print();
        inorder(node.right);
    }

    public static int countLeaves(TreeNode node) {
        if (node == null)
            return 0;

        if (TreeNode.isLeaf(node))
            return 1;

        return countLeaves(node.left) + countLeaves(node.right);
    }

    public static void computeFollowpos(TreeNode node, Map<Integer, Set<Integer>>
map) {
        if (node == null)
            return;

        computeFollowpos(node.left, map);
        computeFollowpos(node.right, map);

        if (TreeNode.isTerminal(node.symbol) || node.symbol == '|') {
            return;
        }

        if (node.symbol == '*') {
            for (int i : node.lastpos) {
                map.get(i).addAll(node.firstpos);
            }
            return;
        }

        for (int i : node.left.lastpos) {
            map.get(i).addAll(node.right.firstpos);
        }
    }

    public static void mapSymbolToIndices(TreeNode node, Map<Character,
Set<Integer>> map) {
```

```
        if (node == null)
            return;

        mapSymbolToIndices(node.left, map);
        mapSymbolToIndices(node.right, map);

        if (TreeNode.isLeaf(node)) {
            if (!map.containsKey(node.symbol)) {
                map.put(node.symbol, new HashSet<>());
            }

            map.get(node.symbol).add(node.i);
        }
    }

    public static Map<String, Map<Character, Character>> computeTransitions(
            Map<Integer, Set<Integer>> followposMap,
            Map<Character, Set<Integer>> symbolIndexMap, Set<Integer>
rootFirstpos) {

        Set<Set<Integer>> states = new HashSet<>();
        Queue<Set<Integer>> queue = new LinkedList<>();
        Map<Set<Integer>, String> stateChar = new HashMap<>();
        Map<String, Map<Character, Character>> table = new HashMap<>();

        char startStateChar = 'A';

        queue.offer(rootFirstpos);
        states.add(rootFirstpos);

        if (rootFirstpos.containsAll(symbolIndexMap.get('#'))) {
            stateChar.put(rootFirstpos, String.valueOf(startStateChar) + "*");
            table.put(String.valueOf(startStateChar) + "*", new HashMap<>());
        } else {
            stateChar.put(rootFirstpos, String.valueOf(startStateChar));
            table.put(String.valueOf(startStateChar), new HashMap<>());
        }

        while (!queue.isEmpty()) {
            Set<Integer> popped = queue.poll();

            for (char terminal : symbolIndexMap.keySet()) {
                if (terminal == '#')
                    continue;

                Set<Integer> containsTerminal = new HashSet<>(popped);
                containsTerminal.retainAll(symbolIndexMap.get(terminal));
```

```
                Set<Integer> genState = new HashSet<>();

                for (int n : containsTerminal) {
                    genState.addAll(followposMap.get(n));
                }

                if (!states.contains(genState)) {
                    queue.offer(genState);
                    states.add(genState);
                    startStateChar = (char) ((int) startStateChar + 1);
                    if (genState.containsAll(symbolIndexMap.get('#'))) {
                        stateChar.put(genState, String.valueOf(startStateChar) +
"*");
                        table.put(String.valueOf(startStateChar) + "*", new
HashMap<>());
                    } else {
                        stateChar.put(genState, String.valueOf(startStateChar));
                        table.put(String.valueOf(startStateChar), new
HashMap<>());
                    }
                }

                table.get(stateChar.get(popped)).put(terminal,
stateChar.get(genState).charAt(0));
            }
        }

        return table;
    }

    public static void printTransitionTable(Map<String, Map<Character, Character>>
table, Set<Character> c) {
        System.out.println();
        System.out.println("Transition Table");
        System.out.println();
        System.out.print("Q | ");
        for (char ch : c) {
            if (ch != '#')
                System.out.print(ch + " | ");
        }
        System.out.println();
        for (int i = 0; i < c.size(); i++) {
            System.out.print("----");
        }
        System.out.println();
        ArrayList<String> sortedStates = new ArrayList<>(table.keySet());
        Collections.sort(sortedStates);
```

```
        for (String state : sortedStates) {
            if (state.length() == 2) {
                System.out.print(state + "| ");
            } else {
                System.out.print(state + " | ");
            }
            for (char ch : c) {
                if (ch != '#') {
                    System.out.print(table.get(state).get(ch) + " | ");
                }
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter regular expression: ");
        String input = scanner.nextLine();
        input = "(" + input + ")" + "#";
        scanner.close();

        System.out.println("\nAppending End marker");
        System.out.println(input);

        String concat = insertConcat(input);
        System.out.println("\nInserting Concatenation");
        System.out.println(concat);

        String postfix = toPostFix(concat);
        System.out.println("\nPost fix");
        System.out.println(postfix);

        TreeNode root = createSyntaxTree(postfix);

        computeFunctions(root);

        // System.out.println("\nPrinting Every Node detail inorder:\n");
        // inorder(root);
        System.out.println();
        printTree(root);
        Map<Integer, Set<Integer>> followposMap = new HashMap<>();
        int leaves = countLeaves(root);

        for (int i = 1; i <= leaves; i++) {
            followposMap.put(i, new HashSet<>());
        }
```

```
        computeFollowpos(root, followposMap);

        System.out.println("\nFollowpos Table");
        System.out.println("------------------------------");
        System.out.printf("| %-9s | %-12s |\n", "Position", "Followpos");
        System.out.println("------------------------------");

        for (int position : followposMap.keySet()) {
            System.out.printf("| %-9d | %-12s |\n", position,
followposMap.get(position));
        }

        System.out.println("------------------------------");

        Map<Character, Set<Integer>> symbolIndexMap = new HashMap<>();
        mapSymbolToIndices(root, symbolIndexMap);

        Map<String, Map<Character, Character>> table = computeTransitions(
                followposMap,
                symbolIndexMap,
                root.firstpos);

        printTransitionTable(table, symbolIndexMap.keySet());


        // Parse Tree Animation
        SwingUtilities.invokeLater(() -> new ParseTreeAnimation(root));

        DFAVisualization dfaVisualization = new DFAVisualization(table,
symbolIndexMap.keySet());
        SwingUtilities.invokeLater(() -> dfaVisualization.showDFA());
    }
    static class ParseTreeAnimation extends JFrame {
        public ParseTreeAnimation(TreeNode root) {
            setTitle("Parse Tree Animation");
            setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            setLayout(new BorderLayout());

            ParseTreePanel treePanel = new ParseTreePanel(root);
            add(treePanel, BorderLayout.CENTER);

            setSize(2400, 800);
            setLocationRelativeTo(null);
            setVisible(true);
        }
    }
}
```

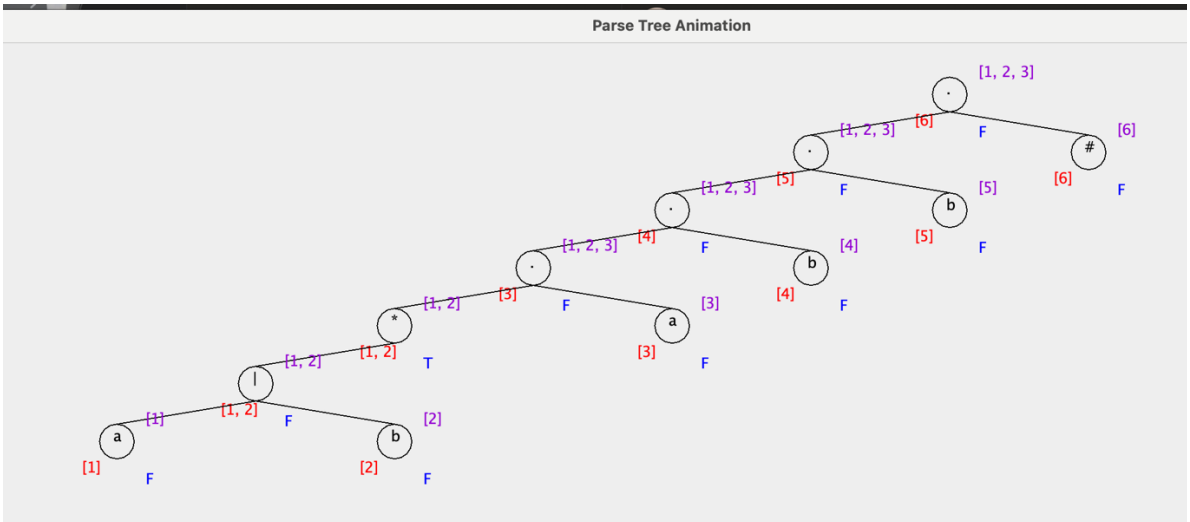
```
class DFAVisualization extends JFrame {
    private final Map<String, Map<Character, Character>> transitionTable;
    private final Set<Character> alphabet;

    public DFAVisualization(Map<String, Map<Character, Character>>
transitionTable, Set<Character> alphabet) {
        this.transitionTable = transitionTable;
        this.alphabet = alphabet;
    }

    public void showDFA() {
        setTitle("DFA Visualization");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new BorderLayout());

        DFAPanel dfaPanel = new DFAPanel(transitionTable, alphabet);
        add(dfaPanel, BorderLayout.CENTER);

        setSize(800, 600);
        setLocationRelativeTo(null);
        setVisible(true);
    }
    class DFAPanel extends JPanel {
        private final Map<String, Map<Character, Character>> transitionTable;
        private final Set<Character> alphabet;

        public DFAPanel(Map<String, Map<Character, Character>> transitionTable,
Set<Character> alphabet) {
            this.transitionTable = transitionTable;
            this.alphabet = alphabet;
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);

            int startX = 50;
            int startY = 50;
            int stateWidth = 50;
            int stateHeight = 30;

            // Draw states
            for (String state : transitionTable.keySet()) {
                g.drawOval(startX, startY, stateWidth, stateHeight);
                g.drawString(state, startX + stateWidth / 3, startY + stateHeight
/ 2);
                startX += 100;
```

```
            }

            // Draw transitions
            startX = 75;
            startY += stateHeight+40;

            for (char symbol : alphabet) {
                if (symbol == '#')
                    continue;
                for (String currentState : transitionTable.keySet()) {
                    String nextState =
String.valueOf(transitionTable.get(currentState).get(symbol));
                    g.drawLine(startX, startY, startX + stateWidth, startY);
                    g.drawString(String.valueOf(symbol), startX + stateWidth / 2,
startY - 5);

                    int nextX = startX + stateWidth / 2;
                    int nextY = startY + 30;

                    g.drawString(nextState, nextX - stateWidth / 3, nextY -
stateHeight / 2);
                    startX += 100;
                }

                startX = 75;
                startY += 60;
            }
        }
    }

}
```

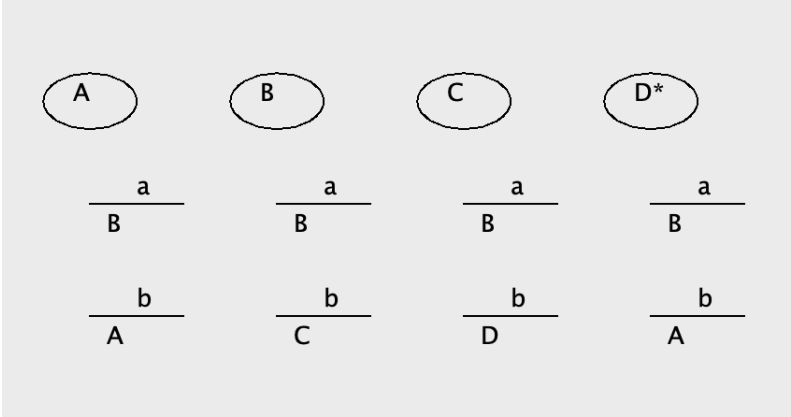
**Output**

```
Enter regular expression:
(a|b)*abb

Appending End marker
((a|b)*abb)#

Inserting Concatenation
((a|b)*.a.b.b).#

Post fix
ab|*a.b.b.#.

-------------------------------------------------------------------------------
| Value | Left Child     | Right Child     | Nullable  | Firstpos     | Lastpos  |
-------------------------------------------------------------------------------
| .     | .              | #               | false     | [1, 2, 3]    | [6]      |
| .     | .              | b               | false     | [1, 2, 3]    | [5]      |
| .     | .              | b               | false     | [1, 2, 3]    | [4]      |
| .     | *              | a               | false     | [1, 2, 3]    | [3]      |
| *     | |              | null            | true      | [1, 2]       | [1, 2]   |
| |     | a              | b               | false     | [1, 2]       | [1, 2]   |
| a     | null           | null            | false     | [1]          | [1]      |
| b     | null           | null            | false     | [2]          | [2]      |
| a     | null           | null            | false     | [3]          | [3]      |
| b     | null           | null            | false     | [4]          | [4]      |
| b     | null           | null            | false     | [5]          | [5]      |
| #     | null           | null            | false     | [6]          | [6]      |
-------------------------------------------------------------------------------

Followpos Table
----------------------------
| Position  | Followpos    |
----------------------------
| 1         | [1, 2, 3]    |
| 2         | [1, 2, 3]    |
| 3         | [4]          |
| 4         | [5]          |
| 5         | [6]          |
| 6         | []           |
----------------------------

Transition Table

Q | a | b |
------------
A | B | A |
B | B | C |
C | B | D |
D*| B | A |
```

## Parse Tree Animation:




## DFA Visualization:




| | |
|---|---|
| **Conclusion** | In this experiment, we learned to optimize DFA-based pattern matchers. We calculated sets like firstpos, lastpos, nullable, and followpos to effectively build parse trees and DFAs |
| **References** | [1] *What is Deterministic Finite Automata? - Sisense*. (2019, December 18). Sisense. https://www.sisense.com/glossary/deterministic-finite-automata/<br><br>[2] Automata conversion of RE to FA - Javatpoint. (n.d.). www.javatpoint.com. https://www.javatpoint.com/automata-conversion-of-re-to-fa<br><br>[3] GeeksforGeeks. (2022a, February 22). *Regular expression to DFA*. https://www.geeksforgeeks.org/regular-expression-to-dfa/ |