BHARATIYA VIDYA BHAVAN'S

SARDAR PATEL INSTITUTE OF TECHNOLOGY

(Empowered Autonomous Institute Affiliated to University of Mumbai)

[Knowledge is Nectar]

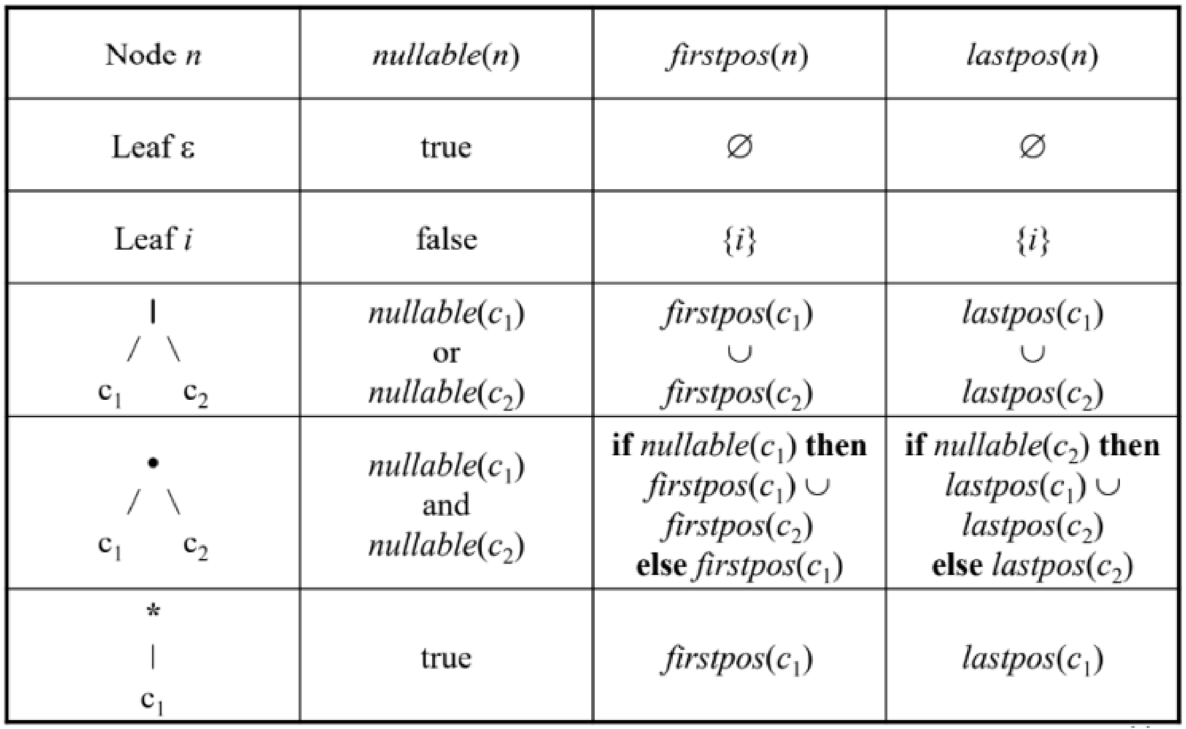
Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300108
Name	Hatim Sawai
Class and Batch	TE Computer Engineering Class B – Batch C
Date	12/2/2024
Lab #	2
Aim	Write a program to implement optimization of DFA-Based Pattern Matchers
Objective	To build a parse tree and to find firstpos, followpos, lastpos and nullable for a given expression.
Theory	**REGULAR GRAMMAR** Regular grammar generates regular language. They have a single non-terminal on the left-hand side and a right-hand side consisting of a single terminal or single terminal followed by a non-terminal. The productions must be in the form: A ⇢ xB A ⇢ x A ⇢ Bx where A, B ∈ Variable(V) and x ∈ T* i.e. string of terminals. **TYPES OF REGULAR GRAMMAR** 1. Left Linear Grammar In LLG, the productions are of the form A ⇢ Bx A ⇢ x where A,B ∈ V and x ∈ T* 2. Right Linear Grammar In RLG, the productions are in the form if all the productions are of the form A ⇢ xB A ⇢ x where A,B ∈ V and x ∈ T* **PARSE TREE** Parse tree is the hierarchical representation of terminals or non-terminals. These symbols (terminals or non-terminals) represent the derivation of the grammar to yield input strings. In parsing, the string springs using the beginning symbol. The starting symbol of the grammar must be used as the root of the Parse Tree. Leaves of parse trees represent terminals. Each interior node represents productions of a grammar. **REGULAR EXPRESSION TO PARSE TREE CONVERSION** Creating a parse tree for a regular expression using the FollowPos algorithm involves several steps. FollowPos is commonly used in the context of constructing a syntax tree for regular expressions. The basic idea is to build a tree that represents the structure of the regular expression and the relationships between the different elements.

FUNCTIONS COMPUTED FROM THE SYNTAX TREE

1. nullable(n): is true for a syntax-tree node n if and only if the subexpression represented by n has in its language.
2. firstpos(n): set of positions that can match the first symbol of a string generated by the subexpression represented by node.
3. lastpos(n): the set of positions that can match the last symbol of a string generated by the subexpression represented by node n
4. followpos(p): the set of positions that can follow position p in the syntax-tree



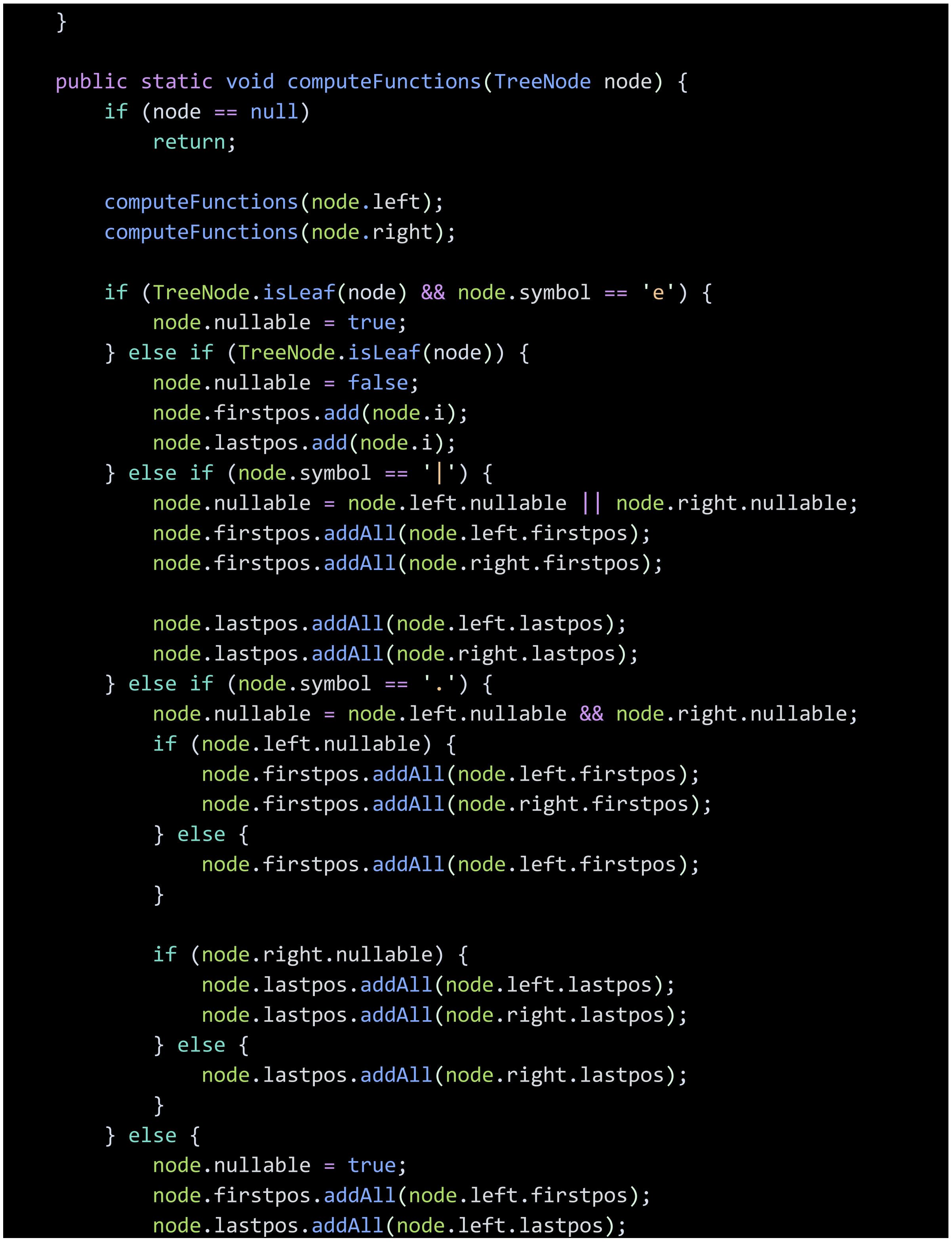
Node n	$nullable(n)$	$firstpos(n)$	$lastpos(n)$
Leaf ε	true	$\emptyset$	$\emptyset$
Leaf i	false	$\{i\}$	$\{i\}$
| / \ c_1 c_2	$nullable(c_1)$ or $nullable(c_2)$	$firstpos(c_1) \cup firstpos(c_2)$	$lastpos(c_1) \cup lastpos(c_2)$
• / \ c_1 c_2	$nullable(c_1)$ and $nullable(c_2)$	**if** $nullable(c_1)$ **then** $firstpos(c_1) \cup firstpos(c_2)$ **else** $firstpos(c_1)$	**if** $nullable(c_2)$ **then** $lastpos(c_1) \cup lastpos(c_2)$ **else** $lastpos(c_2)$
* | c_1	true	$firstpos(c_1)$	$lastpos(c_1)$

Implementation / Code

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

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 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) {
                g.drawString(Character.toString(node.symbol), x, y);
                if (node.left != null) {
                    g.drawLine(x + 10, y + 10, x - xOffset + 10, y + 50);
                }
                if (node.right != null) {
                    g.drawLine(x + 10, y + 10, x + xOffset + 10, y + 50);
                }
                drawTree(g, node.left, x - xOffset, y + 50, level + 1,
xOffset / 2);
                drawTree(g, node.right, x + xOffset, y + 50, level + 1,
xOffset / 2);
            }
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            drawTree(g, root, getWidth() / 2, 30, 0, getWidth() / 4);
        }
    }

    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;
    }
```

```java
    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]);
```

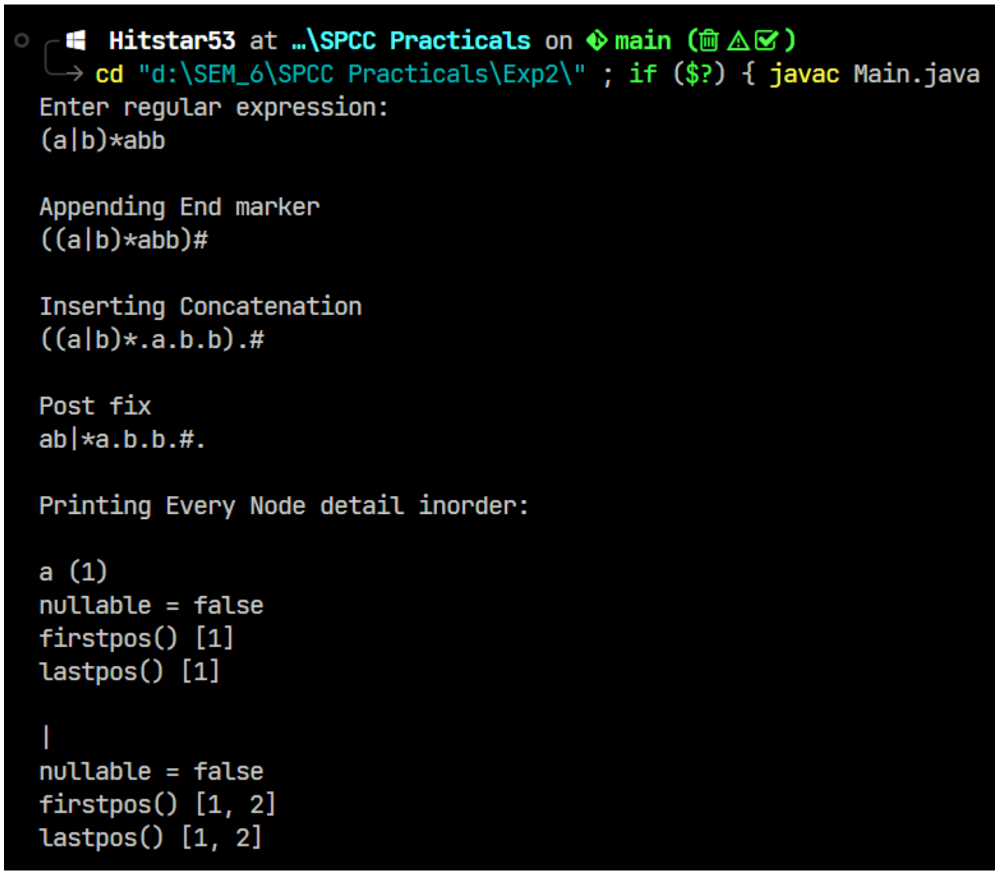
```
            boolean starTerm = arr[i - 1] == '*' &&
TreeNode.isTerminal(arr[i]);
            boolean cbraceTerm = arr[i - 1] == ')' &&
TreeNode.isTerminal(arr[i]);
            boolean cbraceObrace = arr[i - 1] == ')' && arr[i] == '(';
            boolean termObrace = TreeNode.isTerminal(arr[i - 1]) && 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();
```



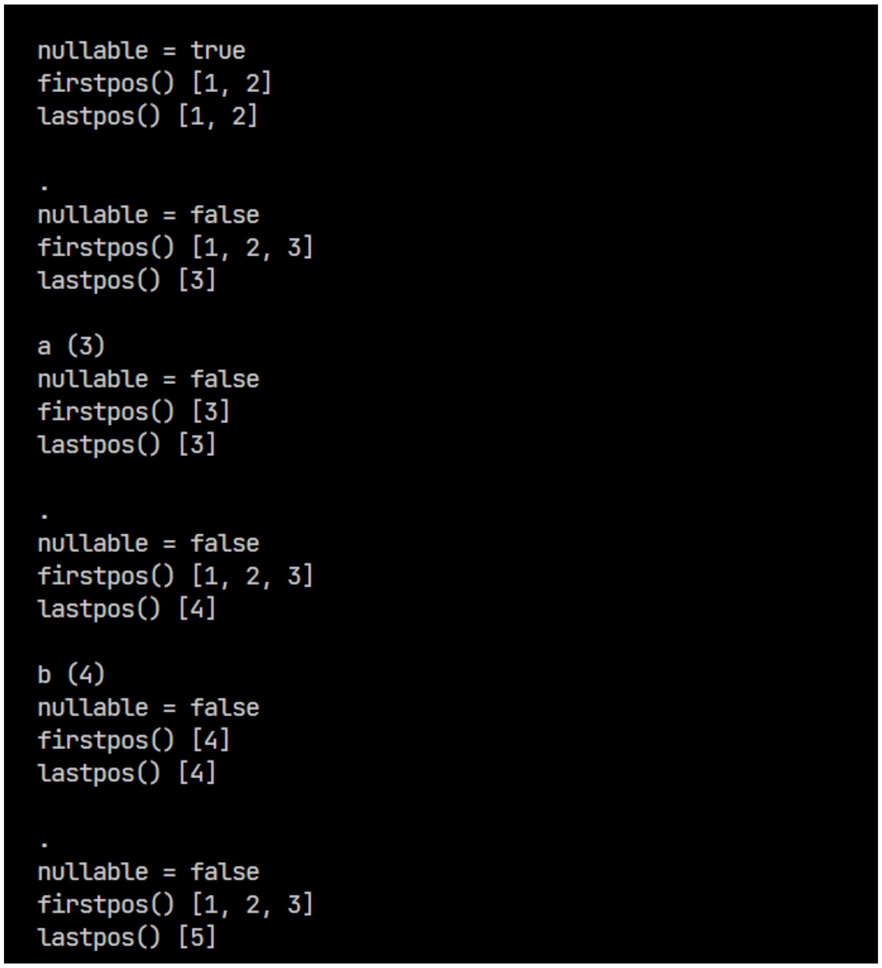
```
    }

    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));
```

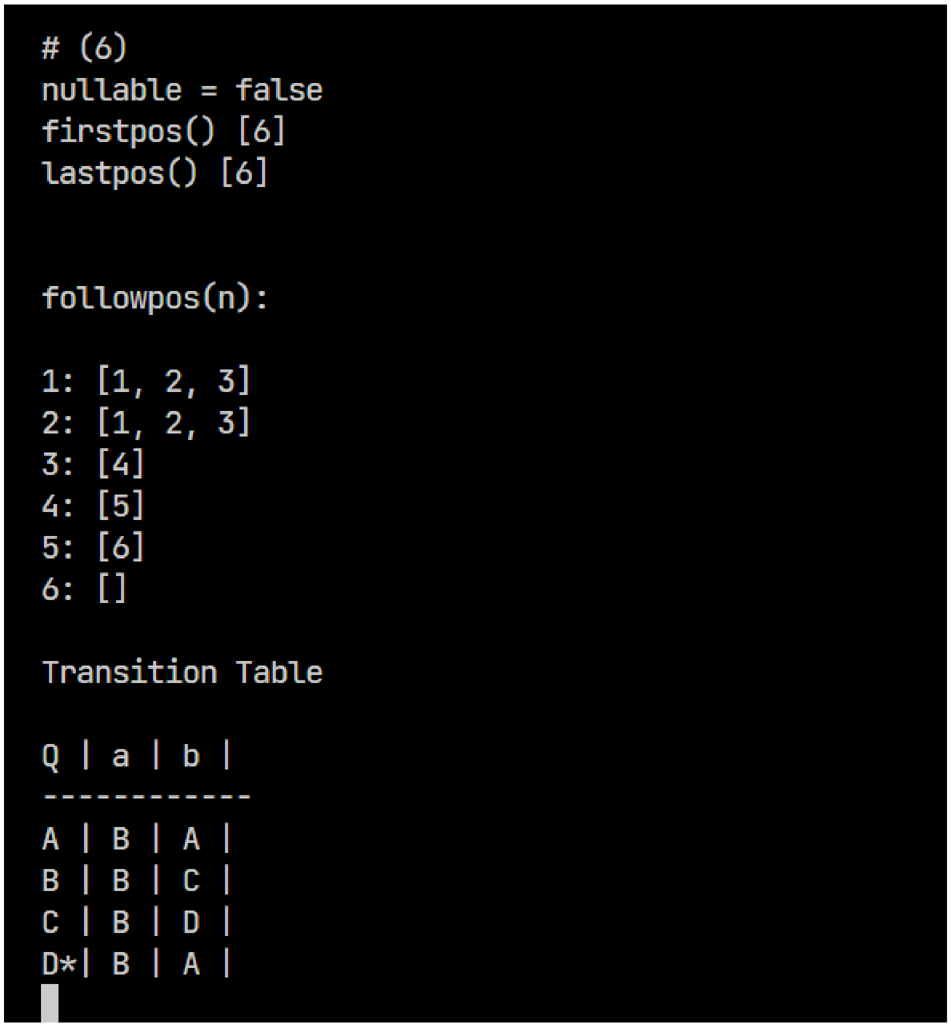
```java
            }
        }

        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();

        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("followpos(n):\n");

        for (int n : followposMap.keySet()) {
            System.out.println(n + ": " + followposMap.get(n).toString());
        }

        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));
    }

    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(800, 600);
            setLocationRelativeTo(null);
            setVisible(true);
        }
    }
}
```

Output	

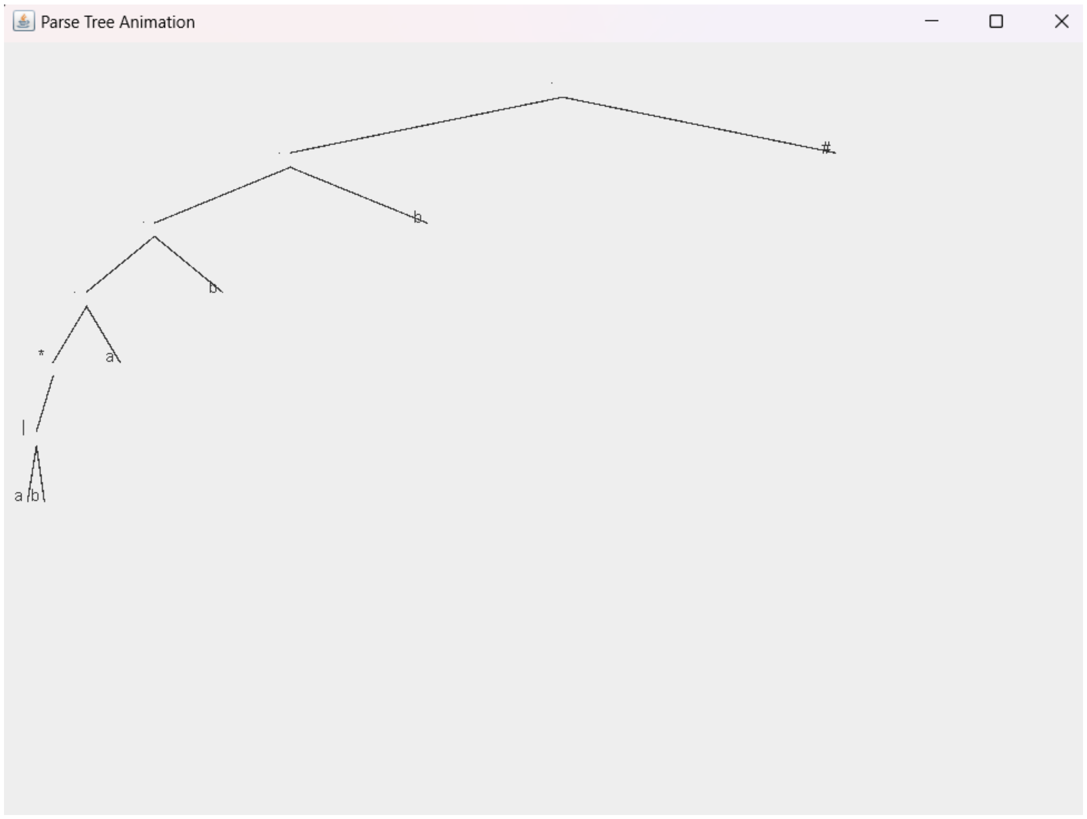
```
Hitstar53 at …\SPCC Practicals on main
→ cd "d:\SEM_6\SPCC Practicals\Exp2\" ; if ($?) { javac Main.java
Enter regular expression:
(a|b)*abb

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

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

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

Printing Every Node detail inorder:

a (1)
nullable = false
firstpos() [1]
lastpos() [1]

|
nullable = false
firstpos() [1, 2]
lastpos() [1, 2]
```

```
nullable = true
firstpos() [1, 2]
lastpos() [1, 2]

.
nullable = false
firstpos() [1, 2, 3]
lastpos() [3]

a (3)
nullable = false
firstpos() [3]
lastpos() [3]

.
nullable = false
firstpos() [1, 2, 3]
lastpos() [4]

b (4)
nullable = false
firstpos() [4]
lastpos() [4]

.
nullable = false
firstpos() [1, 2, 3]
lastpos() [5]
```



```
# (6)
nullable = false
firstpos() [6]
lastpos() [6]


followpos(n):

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:  Parse Tree Animation # b b * a | a b
Conclusion	In this experiment, I implemented construction of DFA for lexical analysers. I also implemented and displayed the firstpos, followpos, lastpos and nullable for an expression. I also constructed the parse tree for the same and displayed it using Java-Swing
References	[1] Goutam Nagpal (2022): https://www.geeksforgeeks.org/regular-grammar-model-regular-grammars/ GeeksforGeeks [2] Deepanshu Rustagi (2022): https://www.geeksforgeeks.org/parse-tree-in-compiler-design/ GeeksforGeeks [3] Dr. Xeujun Liang (2019): Compiler Theory [4] AI LLM (2024): ChatGPT