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

|  |  |
| --- | --- |
| **UID** | 2021300126 |
| **Name** | Pranay Singhvi |
| **Class and Batch** | TE Computer Engineering - Batch C |
| **Date** | 07-02-2024 |
| **Lab #** | 2 |
| **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** | 1. Parse Tree Construction:  A Parse Tree is a hierarchical representation of the syntactic structure of a given expression or grammar. It visually illustrates how the input string is parsed according to the grammar rules. Recursive descent parsing or other algorithms can be employed to build a Parse Tree, where non-terminals are represented by internal nodes and terminals by leaves.  2. Firstpos Calculation:  The `firstpos` of a node in the Parse Tree is the set of positions in the input string where the node can be the first symbol of a valid parse. For terminal nodes, `firstpos` is simply the set containing the node's position. For non-terminals, it is the union of `firstpos` sets of its child nodes.  3. Lastpos Calculation:  The `lastpos` of a node is the set of positions where it can be the last symbol of a valid parse. For terminals, `lastpos` is the set containing the node's position. For non-terminals, it is the union of `lastpos` sets of its child nodes. The `lastpos` of a node is essential for handling concatenation.  4. Followpos Calculation:  The `followpos` of a non-terminal node represents the set of positions that can follow any occurrence of the non-terminal in a valid parse. This is determined by examining the `firstpos` sets of nodes following the non-terminal in the Parse Tree, considering concatenation and closure operations.  Procedure:  - Parse Tree Construction: Utilize recursive descent parsing or other parsing algorithms to construct a Parse Tree, adhering to grammar rules.  - Firstpos Calculation: Traverse the Parse Tree, calculating `firstpos` sets for each node based on its type (terminal or non-terminal) and the `firstpos` sets of its children.  - Lastpos Calculation: Traverse the Parse Tree (possibly in reverse), calculating `lastpos` sets for each node by considering its type and the `lastpos` sets of its children.  - Followpos Calculation: Use information from `firstpos` and `lastpos` to compute `followpos` for non-terminal nodes. Account for concatenation and closure operations.  Output:  - Display the constructed Parse Tree to visually represent the parsing structure.  - Provide the `firstpos`, `lastpos`, and `followpos` sets for each node, enabling further analysis of the parsing process.  Applications:  Efficient parsing and analysis of regular expressions and grammars, crucial in the fields of compiler construction, syntax analysis, and language processing, facilitating the generation of abstract syntax trees and subsequent code generation or interpretation. |
| **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 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();  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));  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.drawRect(startX, startY, stateWidth, stateHeight);  g.drawString(state, startX + stateWidth / 3, startY + stateHeight / 2);  startX += 100;  }  // Draw transitions  startX = 75;  startY += stateHeight;  for (char symbol : alphabet) {  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** |  |
| **Conclusion** |  |
| **References** |  |