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

| **UID** | 2021300101 |
| --- | --- |
| **Name** | Adwait Purao |
| **Class and Batch** | TE Computer Engineering - Batch \_\_ |
| **Date** | 25/01/2024 |
| **Lab #** | 2 |
| **Aim** | Write a program to implement optimization of DFA-Based Pattern Matchers |
| **Objective** | To build a parse tree  To find firstpos  To find lastpos  To find followpos  To build DFA |
| **Theory** | **Parsing and Automata:**  In the world of programming languages, parsing plays a crucial role in understanding the structure and meaning of code. Let's delve into some key concepts involved in this process:  **Building a Parse Tree:**  A parse tree, often resembling an upside-down tree, is a graphical representation of a program's grammatical structure [1]. It depicts how tokens (the basic building blocks of code) are combined to form expressions and statements, reflecting the language's syntax. Here's the process of building a parse tree:  **Start with the initial symbol:** This represents the entire program and is often denoted by "S" or the grammar's starting point.  **Match tokens:** Use production rules from the grammar to match the first token against the left-hand side of a rule. If there's a match, replace it with the right-hand side.  **Repeat:** Recursively apply step 2 for each newly exposed non-terminal symbol (a symbol representing a group of rules), continuing until all tokens are consumed and a terminal symbol (representing a single token) remains at each leaf node.  **Finding Firstpos:**  Firstpos, short for "first position sets," is a crucial concept in LL parsing [2]. It defines the set of terminal symbols that can begin any valid sentence derived from a non-terminal symbol. To find firstpos:  **For each non-terminal:**   * Check if it directly expands to a terminal symbol. If so, add that symbol to its firstpos set. * Iterate through each production rule for the non-terminal. * For each rule:   + Find the firstpos of the first symbol on the right-hand side.   + If "ε" (empty string) is in the firstpos and the symbol is nullable (can be replaced by ε), add all symbols after it to the current non-terminal's firstpos.   + Continue until encountering a non-nullable symbol or reaching the end of the rule.   **Finding Lastpos:**  Lastpos, or "last position sets," are analogous to firstpos but refer to the set of terminal symbols that can end a valid sentence derived from a non-terminal [2]. To find lastpos:  **For each non-terminal:**   * Reverse each production rule for the non-terminal. * Find the firstpos of the reversed rule (now acting as the beginning). * Remove "ε" from the resulting set.   **Finding Followpos:**  Followpos, or "follow sets," are crucial in both LL and LR parsing [2]. They define the set of terminal symbols that can follow a specific non-terminal in a valid sentence. To find followpos:  **For each non-terminal:**   * Add the end-of-input symbol ($) to its followpos if it can appear at the end of a sentence. * For each production rule where the non-terminal appears:   + Find the firstpos of the symbols following the non-terminal in the rule.   + Remove "ε" from the resulting set.   + If "ε" is in the firstpos and the symbol following the non-terminal is nullable, add the followpos of the non-terminal itself to the current set.   **Building a Deterministic Finite Automaton (DFA):**  A DFA is a powerful tool for recognizing regular languages [3]. It comprises states, transitions between states, and an initial and final state. To build a DFA:   * Start with the initial state. * For each state:   + For each symbol in the alphabet:     - Create a new state or use an existing one.     - Add a transition labeled with the symbol from the current state to the new/existing state.   + Mark the final state(s) based on the language's definition.   Remember, these are just foundational concepts. Each topic deserves deeper exploration, and many resources are available to delve further. |
| **Implementation / Code** | import java.util.\*;  class Node {  char value;  Node leftc;  Node rightc;  int posNumber;  Set<Integer> firstpos;  Set<Integer> lastpos;  Set<Integer> followpos;  boolean nullable;  Node(char value) {  this.value = value;  firstpos = new HashSet<Integer>();  lastpos = new HashSet<Integer>();  followpos = new HashSet<Integer>();  posNumber = 0;  }  }  class State {  ArrayList<Integer> value;  boolean marked;  State() {  value = new ArrayList<Integer>();  }  }  class Transition {  State from;  State to;  char value;  Transition(State from, State to, char value) {  this.from = from;  this.to = to;  this.value = value;  }  }  class Tree {  Node root;  int count = 0;  Set<Character> alphabet;  ArrayList<Node> leaves;  ArrayList<State> Dstates;  ArrayList<Transition> Dtrans;  Tree() {  root = null;  leaves = new ArrayList<Node>();  alphabet = new HashSet<Character>();  Dstates = new ArrayList<State>();  Dtrans = new ArrayList<Transition>();  }  void parseRegex(String regex) {  Stack<Character> st = new Stack<>();  for (int i = 0; i < regex.length(); i++) {  if (regex.charAt(i) == '(') {  int j = i + 1;  while (regex.charAt(j) != ')') {  st.push(regex.charAt(j));  if (Character.isLetter(regex.charAt(j))) {  count++;  alphabet.add(regex.charAt(j));  }  j++;  }  j++;  char c1 = st.pop();  char c2 = st.pop();  char c3 = st.pop();  Node n1 = new Node(c1);  Node n2 = new Node(c2);  Node n3 = new Node(c3);  n2.leftc = n3;  n2.rightc = n1;  i = j;  root = n2;  }  if (regex.charAt(i) == '\*') {  Node temp = new Node('\*');  temp.leftc = root;  root = temp;  }  if (Character.isLetter(regex.charAt(i))) {  count++;  alphabet.add(regex.charAt(i));  if (root != null) {  if (root.value != '.') {  Node temp = new Node('.');  temp.leftc = root;  temp.rightc = new Node(regex.charAt(i));  root = temp;  } else {  if (root.rightc != null) {  Node temp = new Node('.');  temp.leftc = root;  temp.rightc = new Node(regex.charAt(i));  root = temp;  } else {  root.rightc = new Node(regex.charAt(i));  }  }  } else {  Node temp = new Node('.');  temp.leftc = new Node(regex.charAt(i));  }  }  }  Node temp = new Node('.');  temp.rightc = new Node('#');  temp.leftc = root;  root = temp;  count++;  }  void printTree() {  System.out.println("--------------------------------------------------------------------------------------------");  System.out.printf("| %-5s | %-14s | %-15s | %-9s | %-12s | %-8s | %-12s |\n",  "Value", "Left Child", "Right Child", "Nullable", "Firstpos", "Lastpos", "Followpos");  System.out.println("--------------------------------------------------------------------------------------------");  printTree(root);  System.out.println("--------------------------------------------------------------------------------------------");  }  void printTree(Node n) {  if (n == null) {  return;  }  System.out.printf("| %-5s | %-14s | %-15s | %-9s | %-12s | %-8s | %-12s |\n",  n.value, (n.leftc != null) ? n.leftc.value : "null",  (n.rightc != null) ? n.rightc.value : "null", n.nullable,  n.firstpos, n.lastpos, n.followpos);  printTree(n.leftc);  printTree(n.rightc);  }  void numberLeaves(Node n) {  if (isLeaf(n)) {  n.posNumber = count;  n.firstpos.add(count);  n.lastpos.add(count);  leaves.add(0, n);  count--;  return;  } else if (n.value == '\*') {  numberLeaves(n.leftc);  } else {  numberLeaves(n.rightc);  numberLeaves(n.leftc);  }  }  void assignNullable(Node n) {  if (n.value == '|') {  n.nullable = n.leftc.nullable || n.rightc.nullable;  assignNullable(n.leftc);  assignNullable(n.rightc);  } else if (n.value == '.') {  n.nullable = n.leftc.nullable && n.rightc.nullable;  assignNullable(n.leftc);  assignNullable(n.rightc);  } else if (n.value == '\*') {  n.nullable = true;  assignNullable(n.leftc);  } else {  n.nullable = false;  }  }  void assignFirstLastPos(Node n) {  if (n.value == '|') {  assignFirstLastPos(n.leftc);  assignFirstLastPos(n.rightc);  Set<Integer> temp1 = new HashSet<Integer>();  temp1.addAll(n.leftc.firstpos);  temp1.addAll(n.rightc.firstpos);  n.firstpos.addAll(temp1);  Set<Integer> temp2 = new HashSet<Integer>();  temp2.addAll(n.leftc.lastpos);  temp2.addAll(n.rightc.lastpos);  n.lastpos.addAll(temp2);  } else if (n.value == '.') {  assignFirstLastPos(n.leftc);  assignFirstLastPos(n.rightc);  if (n.leftc.nullable) {  Set<Integer> temp1 = new HashSet<Integer>();  temp1.addAll(n.leftc.firstpos);  temp1.addAll(n.rightc.firstpos);  n.firstpos.addAll(temp1);  } else {  n.firstpos.addAll(n.leftc.firstpos);  }  if (n.rightc.nullable) {  Set<Integer> temp1 = new HashSet<Integer>();  temp1.addAll(n.leftc.lastpos);  temp1.addAll(n.rightc.lastpos);  n.lastpos.addAll(temp1);  } else {  n.lastpos.addAll(n.rightc.lastpos);  }  } else if (n.value == '\*') {  assignFirstLastPos(n.leftc);  n.firstpos.addAll(n.leftc.firstpos);  n.lastpos.addAll(n.leftc.lastpos);  } else {  return;  }  }  void calculateFollowPos(Node n) {  if (n.value == '.') {  Iterator<Integer> it = n.leftc.lastpos.iterator();  while (it.hasNext()) {  int i = it.next();  Set<Integer> temp = new HashSet<Integer>();  temp.addAll(n.rightc.firstpos);  temp.addAll(leaves.get(i - 1).followpos);  leaves.get(i - 1).followpos.addAll(temp);  }  } else if (n.value == '\*') {  Iterator<Integer> it = n.lastpos.iterator();  while (it.hasNext()) {  int i = it.next();  Set<Integer> temp = new HashSet<Integer>();  temp.addAll(n.firstpos);  temp.addAll(leaves.get(i - 1).followpos);  leaves.get(i - 1).followpos.addAll(temp);  }  }  }  void assignFollowPos(Node n) {  if (n == null) {  return;  } else {  calculateFollowPos(n);  assignFollowPos(n.leftc);  assignFollowPos(n.rightc);  }  }  void constructDstates() { State s0 = new State();  s0.value.addAll(root.firstpos); Dstates.add(s0);  Queue<State> queue = new LinkedList<>(); queue.add(s0);  // Set to keep track of processed states  Set<Set<Integer>> processedStates = new HashSet<>();  processedStates.add(new HashSet<>(s0.value)); // Convert ArrayList<Integer> to Set<Integer>  while (!queue.isEmpty()) {  State currentState = queue.poll();  for (char a : alphabet) { Set<Integer> U = new HashSet<>();  for (int p : currentState.value) { Node node = leaves.get(p - 1); if (node.value == a) {  U.addAll(node.followpos);  }  }  if (!processedStates.contains(U)) {  State newState = new State();  newState.value.addAll(U);  Dstates.add(newState);  queue.add(newState);  processedStates.add(U);  }  State newState = getStateByValue(Dstates, U);  Dtrans.add(new Transition(currentState, newState, a));  }  }  }  void printDFA(){  System.out.println('\n' + "DFA States: "); for (Transition t : Dtrans) {  System.out.println(t.from.value + " -> " + t.to.value + ": " + t.value);  }  }  boolean containsState(ArrayList<State> states, Set<Integer> value) {  for (State state : states) {  if (state.value.equals(value)) {  return true;  }  }  return false;  }  State getStateByValue(ArrayList<State> states, Set<Integer> value)  {  for (State state : states) {  if (state.value.size() != value.size()) {  continue; // If sizes are different, sets cannot be equal  }  boolean equalSets = true;  for (int pos : state.value) { if (!value.contains(pos)) {  equalSets = false; break;  }  }  if (equalSets) { return state;  }  }  return null;  }  boolean isLeaf(Node n) {  return n.leftc == null && n.rightc == null;  }  State getUnmarkedState() {  for (int i = 0; i < Dstates.size(); i++) {  if (!Dstates.get(i).marked) {  return Dstates.get(i);  }  }  return null;  }  boolean checkAllMarked() {  for (int i = 0; i < Dstates.size(); i++) {  if (!Dstates.get(i).marked) {  return false;  }  }  return true;  }  }  public class parseTree {  public static void main(String args[]) {  Tree t = new Tree();  Scanner sc = new Scanner(System.in);  System.out.println("Enter the regular expression: ");  String regex = sc.nextLine();  t.parseRegex(regex);  t.numberLeaves(t.root);  t.assignNullable(t.root);  t.assignFirstLastPos(t.root);  t.assignFollowPos(t.root);  t.constructDstates();  t.printTree();  t.printDFA();  sc.close();  }  } |
| **Output** |  |
| **Conclusion** | In summary, our experiment focused on enhancing DFA-Based Pattern Matchers. We achieved our goal by creating a parse tree from the given pattern, calculating firstpos, lastpos, and followpos sets, and proficiently constructing a Deterministic Finite Automaton (DFA). This approach significantly improved the efficiency of pattern matching in our study. |
| **References** | References:  [1] Aho, A. V., Sethi, R., & Ullman, J. D. (2007). Compilers: principles, techniques, and tools (2nd ed.). Addison-Wesley.  [2] Appel, A. W. (2014). Modern compiler implementation in Java. Cambridge University Press.  [3] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). Introduction to automata theory, languages, and computation (2nd ed.). Addison-Wesley. |