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	2021300096
Name	Sakshi Patil
Class and Batch	TE Computer Engineering - Batch B
Date	25/01/2024
Lab #	2
Aim	Write a program to implement optimization of DFA-Based Pattern Matchers
Objective	To build parse tree To find firstpos To find lastpos To find followpos To build DFA
Theory	To construct DFA from a given regular expression, we can first construct an NFA for the given expression and then convert this NFA to DFA by a subset construction method. But to avoid this two-step procedure, the other way round is to directly construct a DFA for the given expression. DFA refers to Deterministic Finite Automata. In DFA, for each state, and for each input symbol there is one and only one state to which the automaton can have a transition from its current state. DFA does not accept any ∈-transition. In order to construct a DFA directly from a regular expression, we need to follow the steps listed below: Example: Suppose given regular expression r = (a|b)*abb 1. Firstly, we construct the augmented regular expression for the given expression. By concatenating a unique right-end marker '#' to a regular expression r, we give the accepting state for r a transition on '#' making it an important state of the NFA for r#. 2. Then we construct the syntax tree for r#. 3. Next we need to evaluate four functions nullable, firstpos, lastpos, and followpos. nullable(n) is true for a syntax tree node n if and only if the regular expression represented by n has € in its language.

firstpos(n) gives the set of positions that can match the first symbol of a string generated by the subexpression rooted at n.
lastpos(n) gives the set of positions that can match the last symbol of a string generated by the subexpression rooted at n.
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 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 #.
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(){
        // format the output
        System.out.println("Value | Left Child | Right Child |
Nullable | Firstpos | Lastpos | Followpos");
        printTree(root);
    }
    void printTree(Node n){
        if(n==null){
            return;
        }
        System.out.print(n.value + " | ");
        if(n.leftc!=null){
            System.out.print(n.leftc.value + " | ");
        }else{
            System.out.print("null | ");
        }
        if(n.rightc!=null){
            System.out.print(n.rightc.value + " | ");
        }else{
            System.out.print("null | ");
        }
        System.out.print(n.nullable + " | ");
        System.out.print(n.firstpos + " | ");
        System.out.print(n.lastpos + " | ");
        System.out.print(n.followpos + " | ");
        System.out.println();
        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;
    }
}

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

```
PS C:\Users\SAKSHI PATIL\downloads> javac parseTree.java
PS C:\Users\SAKSHI PATIL\downloads> java parseTree
Enter the regular expression:
(a|b)*abb
Value | Left Child | Right Child | Nullable | Firstpos | Lastpos | Followpos
. | . | # | 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] | [1, 2, 3] |
b | null | null | false | [2] | [2] | [1, 2, 3] |
a | null | null | false | [3] | [3] | [4] |
b | null | null | false | [4] | [4] | [5] |
b | null | null | false | [5] | [5] | [6] |
# | null | null | false | [6] | [6] | [] |

DFA States:
[1, 2, 3] -> [1, 2, 3, 4] : a
[1, 2, 3] -> [1, 2, 3] : b
[1, 2, 3, 4] -> [1, 2, 3, 4] : a
[1, 2, 3, 4] -> [1, 2, 3, 5] : b
[1, 2, 3, 5] -> [1, 2, 3, 4] : a
[1, 2, 3, 5] -> [1, 2, 3, 6] : b
[1, 2, 3, 6] -> [1, 2, 3, 4] : a
[1, 2, 3, 6] -> [1, 2, 3] : b
```

Conclusion

In conclusion, our experiment aimed to optimize DFA-Based Pattern Matchers. We successfully constructed a parse tree from the input pattern, computed firstpos, lastpos, and followpos sets, and efficiently constructed a DFA.

References

GeeksForGeeks (2022,22 Feb)

	Regular Expression to DFA - GeeksforGeeks Tutorialspoint (2023,7 Oct) Deterministic Finite Automaton (tutorialspoint.com)