Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 2:

Determinism in Finite Automata. Conversion from NDFA to DFA. Chomsky Hierarchy.

Elaborated:

st. gr. FAF-223 Bujilov Dmitrii

Verified:

asist. univ. Dumitru Cretu

Chișinău - 2024

Objectives:

Understand what an automaton is and what it can be used for.

Continuing the work in the same repository and the same project, the following need to be added:

a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.

b. For this you can use the variant from the previous lab.

According to your variant number (by universal convention it is register ID), get the finite automaton definition and do the following tasks:

a. Implement conversion of a finite automaton to a regular grammar.

b. Determine whether your FA is deterministic or non-deterministic.

c. Implement some functionality that would convert an NDFA to a DFA.

d. Represent the finite automaton graphically (Optional, and can be considered as a bonus point):

You can use external libraries, tools or APIs to generate the figures/diagrams.

Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

Please consider that all elements of the task 3 can be done manually, writing a detailed report about how you've done the conversion and what changes have you introduced. In case if you'll be able to write a complete program that will take some finite automata and then convert it to the regular grammar - this will be a good bonus point.

Implementation description

Firstly, I added this method to the FiniteAutomaton class. It determines whether the automaton is deterministic or not:

```
public boolean isDeterministic() {
    for (String state : transitions.keySet()) {
        Map<String, Set<String>> stateTransitions = transitions.get(state);

        for (String symbol : stateTransitions.keySet()) {
            Set<String> nextStates = stateTransitions.get(symbol);
```

```
                if (nextStates.size() > 1) {
                    return false;
                }
            }
        }

        return true;
    }
```

Then I created a method that converts current FiniteAutomaton from NFA to DFA (or returns the same automaton, if it is already a DFA):

```
    public FiniteAutomaton convertToDeterministic() {
        if (isDeterministic()) {
            return this;
        }

        Set<String> newStates = new HashSet<>();
        Map<String, Map<String, Set<String>>> newTransitions = new
HashMap<>();
        Set<String> newAcceptStates = new HashSet<>();

        Queue<Set<String>> stateQueue = new LinkedBlockingQueue<>();
        stateQueue.add(Set.of(startState));

        while (!stateQueue.isEmpty()) {
            Set<String> currentState = stateQueue.poll();
            String stateName = String.join("", currentState);

            if (newStates.contains(stateName)) {
                continue;
```

```
            }

            newStates.add(stateName);

            for (String state : currentState) {
                if (acceptStates.contains(state)) {
                    newAcceptStates.add(stateName);
                    break;
                }
            }

            Map<String, Set<String>> currentStateTransitions = new
HashMap<>();

            for (String symbol : alphabet) {
               for (String state : currentState) {
                   if (transitions.containsKey(state) &&
transitions.get(state).containsKey(symbol)) {
                       Set<String> nextStates =
transitions.get(state).get(symbol);
                       currentStateTransitions.put(symbol,
Set.of(String.join("", nextStates)));
                       stateQueue.add(nextStates);
                   }
               }
            }

            newTransitions.put(stateName, currentStateTransitions);
        }
```

```
        return new FiniteAutomaton(newStates, alphabet, newTransitions,
startState, newAcceptStates);
    }
```

I also created a method that visualizes the current automaton in the FiniteAutomaton class:

```
    public void visualize(String title) {
        new FiniteAutomatonVisualizer(this, title);
    }
```

Here it is the FiniteAutomatonVisualizer class that is responsible for visualizing automatons:

```
import javax.swing.*;
import java.awt.*;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class FiniteAutomatonVisualizer extends JFrame {
    private final Map<String, Point> statePositions;

    private static final Color STATE_COLOR = Color.decode("#9F00FF");

    public FiniteAutomatonVisualizer(FiniteAutomaton automaton, String
title) {
        super(title);
        setSize(600, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        statePositions = new
AutomatonLayout().generateLayout(automaton.getStates());
```

```
        AutomatonPanel automatonPanel = new AutomatonPanel(automaton);
        add(automatonPanel);

        setLocationRelativeTo(null);
        setVisible(true);
    }

    private class AutomatonPanel extends JPanel {
        private final FiniteAutomaton automaton;

        public AutomatonPanel(FiniteAutomaton automaton) {
            this.automaton = automaton;
        }

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

            for (Map.Entry<String, Map<String, Set<String>>> entry :
automaton.getTransitions().entrySet()) {
                String fromState = entry.getKey();
                Map<String, Set<String>> toStatesMap = entry.getValue();

                for (Map.Entry<String, Set<String>> toStateEntry :
toStatesMap.entrySet()) {
                    String delta = toStateEntry.getKey();
                    Set<String> toStates = toStateEntry.getValue();

                    for(String toState : toStates) {
                        drawTransition(g, fromState, toState, delta);
```

```
                }
            }
        }

        drawStartingTransition(g, automaton.getStartState());

        for (Map.Entry<String, Point> entry :
statePositions.entrySet()) {
            drawState(g, entry.getKey(), entry.getValue());
        }

        for(String state : automaton.getAcceptStates()) {
            drawFinal(g, state);
        }
    }


    private void drawState(Graphics g, String state, Point position)
{
        g.setColor(STATE_COLOR);
        g.fillOval(position.x - 20, position.y - 20, 40, 40);
        g.setColor(Color.BLACK);
        g.drawOval(position.x - 20, position.y - 20, 40, 40);
        g.drawString(state, position.x - 5, position.y + 5);
    }

    private void drawTransition(Graphics g, String fromState, String
toState, String delta) {
        Point fromPosition = statePositions.get(fromState);
        Point toPosition = statePositions.get(toState);
```

```
            Point intersection = calculateIntersectionPoint(fromPosition,
toPosition, 10);

            int labelX = (fromPosition.x + toPosition.x) / 2;
            int labelY = (fromPosition.y + toPosition.y) / 2 - 5;

            if (fromState.equals(toState)) {
                drawSelfLoop(g, fromPosition.x, fromPosition.y, delta);
            } else {
                drawArrow(g, fromPosition.x, fromPosition.y,
intersection.x, intersection.y);
            }

            g.drawString(delta, labelX, labelY);
        }

        private void drawFinal(Graphics g, String state) {
            int offset = 15;

            Point position = statePositions.get(state);
            g.setColor(STATE_COLOR);
            g.fillOval(position.x - offset, position.y - offset, 30, 30);
            g.setColor(Color.BLACK);
            g.drawOval(position.x - offset, position.y - offset, 30, 30);
            g.drawString(state, position.x - 5, position.y + 5);
        }

        private void drawStartingTransition(Graphics g, String state) {
            Point toPosition = statePositions.get(state);
            Point fromPosition = new Point(toPosition.x + 80,
toPosition.y);
```

```
            Point intersection = calculateIntersectionPoint(fromPosition,
toPosition, 10);


            drawArrow(g, fromPosition.x, fromPosition.y, intersection.x,
intersection.y);
        }


        private void drawSelfLoop(Graphics g, int x, int y, String
withSymbol) {
            int loopRadius = 20;
            int arrowSize = 15;


            g.drawOval(x - loopRadius, y - loopRadius * 2, loopRadius *
2, loopRadius * 2);


            int arrowX1 = x + (int) (arrowSize * Math.cos(Math.PI / 6));
            int arrowY1 = y - loopRadius * 2 + (int) (arrowSize *
Math.sin(Math.PI / 6));
            int arrowX2 = x + (int) (arrowSize * Math.cos(-Math.PI / 6));
            int arrowY2 = y - loopRadius * 2 + (int) (arrowSize *
Math.sin(-Math.PI / 6));


            g.drawLine(x, y - loopRadius * 2, arrowX1, arrowY1);
            g.drawLine(x, y - loopRadius * 2, arrowX2, arrowY2);


            int labelX = x - g.getFontMetrics().stringWidth(withSymbol) /
2;
            int labelY = y - loopRadius * 2 - 10;
            g.drawString(withSymbol, labelX, labelY);
        }
```

```
        private Point calculateIntersectionPoint(Point from, Point to,
int circleRadius) {

            double dx = to.x - from.x;

            double dy = to.y - from.y;

            double distance = Math.sqrt(dx * dx + dy * dy);


            int intersectionX = from.x + (int) ((dx / distance) *
(distance - circleRadius * 0.7));

            int intersectionY = from.y + (int) ((dy / distance) *
(distance - circleRadius * 0.7));


            return new Point(intersectionX, intersectionY);

        }


        private void drawArrow(Graphics g, int x1, int y1, int x2, int
y2) {

            Graphics2D g2d = (Graphics2D) g.create();


            int arrowLength = 10;


            double angle = Math.atan2(y2 - y1, x2 - x1);


            Point intersection = calculateIntersectionPoint(new Point(x1,
y1), new Point(x2, y2), 20);


            int x3 = (int) (intersection.x - arrowLength * Math.cos(angle
- Math.PI / 6));

            int y3 = (int) (intersection.y - arrowLength * Math.sin(angle
- Math.PI / 6));

            int x4 = (int) (intersection.x - arrowLength * Math.cos(angle
+ Math.PI / 6));

            int y4 = (int) (intersection.y - arrowLength * Math.sin(angle
+ Math.PI / 6));
```

```
            g2d.drawLine(x1, y1, intersection.x, intersection.y);

            Polygon arrowhead = new Polygon();
            arrowhead.addPoint(intersection.x, intersection.y);
            arrowhead.addPoint(x3, y3);
            arrowhead.addPoint(x4, y4);
            g2d.fill(arrowhead);

            g2d.dispose();
        }
    }

    public static class AutomatonLayout {

        public Map<String, Point> generateLayout(Set<String> states) {
            Map<String, Point> statePositions = new HashMap<>();

            int radius = 100;
            int centerX = 300;
            int centerY = 200;

            double angle = 0;
            double angleIncrement = 2 * Math.PI / states.size();

            for (String state : states) {
                if (state != null) {
                    int x = (int) (centerX + radius * Math.cos(angle));
                    int y = (int) (centerY + radius * Math.sin(angle));
```

```
                    statePositions.put(state, new Point(x, y));
                    angle += angleIncrement;
                }
            }

            return statePositions;
        }

    }
}
```

Also I created these methods to convert FiniteAutomaton to a regular grammar (Grammar class):

```
    public Grammar toGrammar(boolean mapStatesToNonTerminals) {
        return new Grammar(startState, states, alphabet,
buildProductions(mapStatesToNonTerminals));
    }

    public Grammar toGrammar() {
        return toGrammar(true);
    }
```

These methods use these methods inside of them to map states to nonterminals (if needed) and to build productions for grammars:

```
    public static String mapStateToNonTerminal(String state) {
        int stateNumber;

        try {
            stateNumber = Integer.parseInt(state.substring(1));
        } catch (NumberFormatException e) {
```

```
            throw new IllegalArgumentException("State must be in the
format qN, where N is a number");
        }

        if (stateNumber < 0) {
            throw new IllegalArgumentException("State number must be non-
negative");
        }

        char letter = (char) ('A' + stateNumber);

        return String.valueOf(letter);
    }

    private Map<String, List<String>> buildProductions(boolean
mapStatesToNonTerminals) {
        Map<String, List<String>> productions = new HashMap<>();

        for (String state : transitions.keySet()) {
            Map<String, Set<String>> stateTransitions =
transitions.get(state);

            for (String symbol : stateTransitions.keySet()) {
                Set<String> nextStates = stateTransitions.get(symbol);

                if (!productions.containsKey(state)) {
                    productions.put(state, new ArrayList<>());
                }

                List<String> productionList = productions.get(state);
```

```
                if (nextStates.isEmpty()) {
                    // If there are no next states, add the symbol as it
is
                    productionList.add(symbol);
                } else {
                    // Add a production for each next state
                    for (String nextState : nextStates) {
                        productionList.add(symbol +
((mapStatesToNonTerminals) ? mapStateToNonTerminal(nextState) :
nextState));
                    }
                }
            }

        if(mapStatesToNonTerminals) {
            Map<String, List<String>> newProductions = new HashMap<>();
            for (String state : productions.keySet()) {
                newProductions.put(mapStateToNonTerminal(state),
productions.get(state));
            }
            return newProductions;
        }

        return productions;
    }
```

Also I implemented a method that defines the Chomsky type of the grammar in Grammar class (based on terminals and nonterminals count):

```
    public void defineChomskyType() {
        if (productions.keySet().stream().allMatch(s -> s.length() == 1
&& countNonTerminals(s) == 1)
```

```
                && productions.values().stream().allMatch(list ->
list.stream().allMatch(l -> countTerminals(l) <= 1) &&
list.stream().allMatch(l -> countNonTerminals(l) <= 1))) {

            System.out.println("The grammar is of type 3 (regular).");

        } else if (productions.keySet().stream().allMatch(s -> s.length()
== 1 && countNonTerminals(s) == 1)

                && productions.values().stream().allMatch(list ->
(list.stream().allMatch(l -> countTerminals(l) >= 0) &&
list.stream().allMatch(l -> countNonTerminals(l) >= 0)))) {

            System.out.println("The grammar is of type 2 (context-
free).");

        } else if (productions.keySet().stream().anyMatch(s -> s.length()
> 1 && countTerminals(s) > 0 && countNonTerminals(s) > 0)) {

            System.out.println("The grammar is of type 0
(unrestricted).");

        } else {

            System.out.println("The grammar is of type 1 (context-
sensitive).");

        }

    }
```

It uses these methods to count terminals and nonterminals:

```
    private int countTerminals(List<String> list) {

        int count = 0;

        for (String s : list) {

            for (char c : s.toCharArray()) {

                if (terminalSymbols.contains(String.valueOf(c))) {

                    count++;

                }

            }

        }

        return count;

    }
```

```
private int countTerminals(String s) {
    int count = 0;
    for (char c : s.toCharArray()) {
        if (terminalSymbols.contains(String.valueOf(c))) {
            count++;
        }
    }
    return count;
}

private int countNonTerminals(List<String> list) {
    int count = 0;
    for (String s : list) {
        for (char c : s.toCharArray()) {
            if (nonTerminalSymbols.contains(String.valueOf(c))) {
                count++;
            }
        }
    }
    return count;
}

private int countNonTerminals(String s) {
    int count = 0;
    for (char c : s.toCharArray()) {
        if (nonTerminalSymbols.contains(String.valueOf(c))) {
            count++;
        }
```

```
        }
        return count;
    }
```

Conclusions and Results

In the course of this laboratory work, substantial progress was made towards achieving the set objectives. The introduction of a function within the grammar type/class to classify the grammar based on the Chomsky hierarchy was a pivotal development. This addition empowers the system with a capacity to categorize grammars, providing a valuable analytical tool for understanding their structural complexities.

A significant milestone was the successful implementation of the conversion process from a finite automaton to a regular grammar. This accomplishment signifies a crucial link between automata and formal language structures, enhancing our ability to analyze and manipulate different types of languages.

Addressing the deterministic or non-deterministic nature of the finite automaton adds depth to our understanding of its behavior. This distinction holds practical significance in various applications, from language recognition to algorithm design, as it informs us about the predictability and precision of the automaton's responses.

The implementation of functionality to convert a non-deterministic finite automaton (NDFA) to a deterministic finite automaton (DFA) represents a substantial step forward. This process not only streamlines the automaton but also contributes to the efficiency of language recognition systems. The ability to automate this conversion process enhances our toolkit for handling diverse automata types.

Delving into the optional graphical representation of the finite automaton added an extra layer of sophistication to the project. The use of external libraries, tools, or APIs for generating visual representations offers a tangible and intuitive way to comprehend the intricacies of the automaton's structure. This visual representation provides a valuable aid in explaining complex concepts and can be particularly beneficial for educational purposes.

In summary, the laboratory work has resulted in a well-rounded achievement, encompassing both theoretical understanding and practical implementation. The developed functionalities contribute to the versatility of the system, allowing for a more thorough exploration of automata and formal language theory. The successful completion of these tasks not only fulfills the objectives but also positions the project as a robust tool for further studies in the field.