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 2 DFA. Chomsky Hierarchy.

Elaborated:

st. gr. FAF-223 Cravcenco Dmitrii

Verified:

asist. univ. Dumitru Cretu

Chișinău - 2024

Objectives:

1. Understand what an automaton is and what it can be used for.
2. 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.
3. 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

Implementation description

Firstly I created a method that could classify the grammar based on Chomsky hierarchy. In this method I firstly check if grammar is of the class 0 , to do so, I have to check if there exists at least one transition from terminal state. Then I also check if from state has more than 1 non-terminal symbol, this means that grammar is not of the second type -> it is of the first type. After checking for 0 type, I check for type III. I use regex to find out if all the transitions belong to right-hand or left-hand rule. If grammar has all the transitions according to one of the rules then it is of the type III. After this I check if grammar belongs to the second type also using regex and if it doesn't belong -> type I.

```
public String getChomskyType() {

        Set<String> states = rules.keySet();
        boolean notSecondType = false;

        // check for 0 TYPE
        for (String fromState : states) {

            char[] chars = fromState.toCharArray();

            if (fromState.length() > 1)
                notSecondType = true;

            for (char c : chars) {
                if (terminals.contains(c)) {
                    return "TYPE 0";
                }
            }
        }

        if (notSecondType)
            return "TYPE I";


        // check for III TYPE
        boolean leftHandRule = true, rightHandRule = true;

        Pattern type3RightHandRule = Pattern.compile("^[a-z][A-Z]$");
        Pattern type3LeftHandRule = Pattern.compile("^[A-Z][a-z]$");


        boolean currentValue = true;
        for (String fromState : states) {

            List<String> toState = rules.get(fromState);
            for (String s : toState) {

                currentValue = false;
```

```
                // check A-> aB
                if (!Pattern.matches(type3RightHandRule.pattern(), s) &&
s.length() > 1) {
                    rightHandRule = false;
                    currentValue = true;
                }
                // check A-> Ba
                if (!Pattern.matches(type3LeftHandRule.pattern(), s) &&
s.length() > 1) {
                    leftHandRule = false;
                    currentValue = true;
                }

                // check A->a
                if (s.length() == 1) {
                    currentValue = true;
                }

                if (!currentValue) {
                    break;
                }
            }
        }

        if ((rightHandRule && !leftHandRule) || (leftHandRule &&
!rightHandRule))
            if (currentValue)
                return "TYPE III";


        Pattern secondTypePattern = Pattern.compile("([a-z]+)?([A-Z]+)?([a-
z]+)?");
        boolean secondType = true;

        for (String fromState : states) {

            List<String> toState = rules.get(fromState);

            for (String s : toState) {
                if (!Pattern.matches(secondTypePattern.pattern(), s))
                    secondType = false;
            }
        }

        if (secondType)
            return "TYPE II";

        return "TYPE I";
    }
```

Then I implemented a method to generate convert FA to regular grammar. I had to define starting symbol, set of non-terminals, terminals and establish rules according to transitions from FA.

```
public Grammar(FiniteAutomaton fa) {

        startSymbol = fa.getStartStateQ0();
        nonTerminals = fa.getStatesQ();
        terminals = fa.getAlphabetSigma();
        rules = new HashMap<>();

        for (FiniteAutomaton.Transition t : fa.getTransitions()) {

            String toState = t.getToState() == null ? "" :
t.getToState().toString();
            if (rules.containsKey(t.getFromState().toString())) {

rules.get(t.getFromState().toString()).add(t.getWithSymbol().toString() +
toState);
            } else {
                rules.put(t.getFromState().toString(), new
ArrayList<>(List.of(t.getWithSymbol().toString() + toState)));
            }
        }
    }
```

To determine wether FA is deterministic I created the following method. It checks if there are multiple transitions from one state to another with the same 'with' symbol.

```
public boolean isDeterministic() {

        Set<String> fromAndWithSymbols = new HashSet<>();

        for (Transition transition: transitions) {

            String currentFromAndWith = transition.getFromState().toString() +
transition.getWithSymbol().toString();
            if (fromAndWithSymbols.contains(currentFromAndWith)) {
                return false;
            }
            fromAndWithSymbols.add(currentFromAndWith);
        }

        return true;
    }
```

To convert a NDFA to DFA I build the following method. There are a few main steps that I followed to convert NDFA to DFA: 1. Check if deterministic -> if yes return the same FA. 2. Create starting state for the DFA. 3. Add it to the set of new states. 4. Research if there are transitions in new states (if there are -> add new states to the set of states that should be checked). 5. While there are new states to check -> repeat 4. 6. Add new states and transitions to the DFA, return it.

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

        FiniteAutomaton dfa = new FiniteAutomaton();
        dfa.setStartStateQ0(startStateQ0);

        Set<String> newStates = new HashSet<>();
        newStates.add(startStateQ0);

        Set<Transition> newTransitions = new HashSet<>();

        Queue<String> statesToCheck = new LinkedList<>();
        statesToCheck.add(startStateQ0);

        do {

            String currentState = statesToCheck.poll();

            if (currentState == null) {
                break;
            }

            List<String> currentStates = List.of(currentState.split(" "));
            Map<String, Set<String>> withSymbolIntoState = new HashMap<>();

            for (String state : currentStates) {
                for (Transition transition : transitions) {
                    if (transition.getFromState().equals(state)) {
                        String withSymbol = transition.getWithSymbol();
                        String toState = transition.getToState();

                        if (withSymbolIntoState.containsKey(withSymbol)) {
                            withSymbolIntoState.get(withSymbol).add(toState);
                        } else {
                            withSymbolIntoState.put(withSymbol, new
HashSet<>(List.of(toState)));
                        }
                    }
                }
            }

            for (String symbol : withSymbolIntoState.keySet()) {
                Set<String> statesForSymbol = withSymbolIntoState.get(symbol);

                if (statesForSymbol != null && !statesForSymbol.isEmpty()) {
                    String newState = statesForSymbol.stream()
                            .filter(Objects::nonNull)

.sorted(Comparator.nullsLast(Comparator.naturalOrder()))
```

```
                            .collect(Collectors.joining(" "));

                    if (!newState.isEmpty() && !newStates.contains(newState)) {
                        newStates.add(newState);
                        newTransitions.add(new Transition(currentState, symbol,
newState));
                        statesToCheck.add(newState);
                    }
                }
            }

        } while (!newStates.isEmpty());

        System.out.println("New transitions: " + newTransitions);
        dfa.setStatesQ(newStates);
        dfa.setTransitions(newTransitions);
        return dfa;
      }
```

To represent a FA graphically I used JavaX. As example of the functionality from the drawing class I provide you with two functions that draw state and transition.

```
         private void drawState(Graphics g, String state, Point position) {
            if (state.contains("FINAL")) {
                g.setColor(Color.RED);
            } else if (state.contains("S")) {
                g.setColor(Color.GREEN);
            } else {
                g.setColor(Color.ORANGE);
            }
            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, FiniteAutomaton.Transition
transition) {
            Point fromPosition =
statePositions.get(transition.getFromState().toString());
            Point toPosition;

            if (transition.getToState() != null) {
                toPosition =
statePositions.get(transition.getToState().toString());
            } else {
                // For transitions to the final state (empty string)
                toPosition = new Point(fromPosition.x + 80, fromPosition.y);
            }

            // Calculate intersection point on the state's circle
            Point intersection = calculateIntersectionPoint(fromPosition,
```

```
toPosition, 10);

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

            // Draw line with arrowhead or loop
            if (transition.getFromState().equals(transition.getToState())) {
                drawSelfLoop(g, fromPosition.x, fromPosition.y,
transition.getWithSymbol().toString());
            } else {
                drawArrow(g, fromPosition.x, fromPosition.y, intersection.x,
intersection.y);
            }

            g.drawString(transition.getWithSymbol().toString(), labelX, labelY);
        }
```

Conclusions and Results

In this laboratory, I successfully implemented transition from NFA to DFA method, also created visualization and conversion from FA to Grammar. Here's a summary of my key accomplishments:

- Chomsky Hierarchy Classification

I implemented a function that returns the type of current grammar based on Chomsky hierarchy. The function checks for the presence of terminal symbols in the grammar, the number of non-terminal symbols in the grammar, and the structure of the grammar's transition rules.

- Checking for Determinism in Finite Automaton

I developed a method to check if a finite automaton is deterministic. The method checks for multiple transitions from one state to another with the same 'with' symbol.

- Conversion from Finite Automaton to Regular Grammar

I developed a method to convert a finite automaton to a regular grammar. The method considers the transition rules defined by the automaton and generates the grammar accordingly.

- Conversion from NFA to DFA

I developed a method to convert a non-deterministic finite automaton to a deterministic finite automaton. The method uses a queue to keep track of the states that need to be checked and iteratively adds new states and transitions to the DFA.

- Graphical Representation of Finite Automaton

I created a graphical representation of the finite automaton using JavaX. The representation includes the states and transitions of the automaton.