Ministerul Educaţiei şi Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work nr. 2

Course: Formal languages and finite automata

Topic: Determinism in Finite Automata. Conversion from NFA to DFA. Chomsky Hierarchy.

Elaborated:
st. gr. FAF-221 Semichin A.

Verified:

asist. univ. Cretu Dumitru

Chişinău - 2024

Theory:

In the realm of formal languages and automata theory, Finite Automata (FA) are pivotal in modeling computational processes. They consist of a finite set of states, a set of input symbols, transition rules defining state changes upon reading symbols, an initial state, and a set of final (accepting) states. Non-deterministic Finite Automata (NFA) allow multiple transitions from a single state with the same symbol, whereas Deterministic Finite Automata (DFA) have unique transitions per state-symbol pair.

The conversion of an NFA to a DFA is essential for streamlining automata-based algorithms and simplifying automata models, as it ensures unique state transitions for every input symbol.

Objectives:

- Discuss the provided Java implementation for an NFA.
- Highlight the process of converting an NFA to a DFA.
- Explain the extraction of regular grammar from the NFA.

Implementation Description:

Provided Java Code Overview:

The provided Java code implements the NFA and DFA concepts using object-oriented programming. Key components include:

FiniteAutomaton Class: Represents the NFA with functionalities such as checking determinism, converting to DFA, and extracting regular grammars.

```java
// Class to represent a Finite Automaton
class FiniteAutomaton {
    private Set<String> states;
    private Set<String> inputSymbols;
    private Map<String, Map<String, Set<String>>> transitions;
    private String initialState;
    private Set<String> finalStates;

    // Constructor
    public FiniteAutomaton(Set<String> states, Set<String> inputSymbols, Map<String, Map<String,
Set<String>>> transitions,
                           String initialState, Set<String> finalStates) {
        this.states = states;
        this.inputSymbols = inputSymbols;
        this.transitions = transitions;
        this.initialState = initialState;
        this.finalStates = finalStates;
    }
```

Main Class: Instantiates an NFA, checks its determinism, converts it to a DFA if non-deterministic, and then extracts and prints its regular grammar.

```java
// Checking if the automaton is deterministic
        boolean isDeterministic = fa.isDeterministic();
        if (isDeterministic) {
            System.out.println("The Finite Automaton is deterministic.");
        } else {
            System.out.println("The Finite Automaton is not deterministic.");

            // Converting NFA to DFA
            FiniteAutomaton dfa = fa.convertToDFA();
            System.out.println("Equivalent DFA:");
            printDFA(dfa);
        }
```

```java
// Defining NFA
        Set<String> states = new HashSet<>(Arrays.asList("q0", "q1", "q2", "q3", "q4"));
        Set<String> inputSymbols = new HashSet<>(Arrays.asList("a", "b"));
        Map<String, Map<String, Set<String>>> transitions = new HashMap<>();
        transitions.put("q0", Map.of("a", Set.of("q1")));
        transitions.put("q1", Map.of("b", Set.of("q1", "q2")));
        transitions.put("q2", Map.of("a", Set.of("q4"), "b", Set.of("q3")));
        transitions.put("q3", Map.of("a", Set.of("q1")));
        transitions.put("q4", Collections.emptyMap()); // Empty map for q4 to denote no transitions
        String initialState = "q0";
        Set<String> finalStates = new HashSet<>(Arrays.asList("q4"));
```

Conversion from NFA to DFA:
The conversion process involves the following steps:

Epsilon Closure: Computes the epsilon closure for a given set of states to account for epsilon transitions.

```java
// Method to compute epsilon closure for given NFA states
    private Set<String> epsilonClosure(Set<String> states) {
        Set<String> closure = new HashSet<>(states);
        Stack<String> stack = new Stack<>();
        stack.addAll(states);

        while (!stack.isEmpty()) {
            String currentState = stack.pop();
            if (transitions.containsKey(currentState) && transitions.get(currentState).containsKey("")) {
                for (String nextState : transitions.get(currentState).get("")) {
                    if (!closure.contains(nextState)) {
                        closure.add(nextState);
                        stack.push(nextState);
                    }
                }
            }
        }

        return closure;
    }
```

Subset Construction: Utilizes a queue-based approach to build states of the DFA. For each state in the DFA, compute transitions for every input symbol by merging transitions of the corresponding NFA states.

```java
// Method to get the next states given the current state and input symbol
    public Set<String> getTransition(String state, String symbol) {
        if (transitions.containsKey(state) && transitions.get(state).containsKey(symbol)) {
            return transitions.get(state).get(symbol);
        }
        return Collections.emptySet(); // Return empty set if transition is undefined
    }
```

State Representation: DFA states are represented as keys (strings) derived from the set of NFA states they correspond to.

```java
// Method to convert a set of states to a key for use in Map
    private String getStateKey(Set<String> stateSet) {
        return String.join(",", stateSet);
    }

    // Method to get the next states given the current state and input symbol
    public Set<String> getTransition(String state, String symbol) {
        if (transitions.containsKey(state) && transitions.get(state).containsKey(symbol)) {
            return transitions.get(state).get(symbol);
        }
        return Collections.emptySet(); // Return empty set if transition is undefined
    }
```

Regular Grammar Extraction:

The regular grammar is extracted by iterating over each state of the NFA. For each state:

Construct production rules by examining transitions for each input symbol.
If the state is a final state, include an empty production.

```java
public Map<String, List<String>> toRegularGrammar() {
        Map<String, List<String>> regularGrammar = new HashMap<>();
        for (String state : states) {
            List<String> productions = new ArrayList<>();
            for (String symbol : inputSymbols) {
                if (transitions.containsKey(state) && transitions.get(state).containsKey(symbol)) {
                    Set<String> nextStates = transitions.get(state).get(symbol);
                    for (String nextState : nextStates) {
                        productions.add(symbol + nextState);
                    }
                }
            }
            if (finalStates.contains(state)) {
                productions.add("");
            }
            regularGrammar.put(state, productions);
        }
        return regularGrammar;
    }
```

Conclusion:

The implemented Java code effectively demonstrates the conversion of an NFA to a DFA and the extraction of a regular grammar from the NFA. This process is crucial for automata theory, aiding in the understanding and analysis of formal languages and their computational properties. The provided code serves as a practical example of automata manipulation and can be adapted for various language-processing tasks and algorithms.

In summary, this report provides insights into the concepts of NFA-to-DFA conversion and regular grammar extraction, showcasing their implementation within the context of a Java-based automata framework.