

**Laboratory work nr. 1**  
**Course: Formal languages and finite automata**  
**Topic: Intro to formal languages.**  
**Regular grammars. Finite automata**

Elaborated:  
st. gr. FAF-221

Revenco Victor

Verified:  
asist. univ.

Cretu Dumitru

# Theory

Formal languages are abstract languages used to represent strings of symbols according to certain rules. These rules are defined by a grammar, which dictates the structure and formation of valid strings within the language. Regular grammars are a type of formal grammar that generates regular languages.

A regular grammar consists of a set of production rules that define how strings in the language are formed. These rules typically take the form of substitutions or transformations. Regular grammars are characterized by their simplicity and limited expressive power compared to other types of grammars. They can be described using regular expressions, finite automata, or regular grammars themselves. Finite automata are abstract machines used to recognize and accept strings belonging to a formal language. They consist of a finite set of states, a finite alphabet of input symbols, a transition function that maps states and input symbols to other states, an initial state, and a set of accepting states.

Finite automata come in various forms such as deterministic finite automata (DFA), non-deterministic finite automata (NFA), and finite automata with  $\epsilon$ -transitions ( $\epsilon$ -NFA).

## Objectives:

1. Discover what a language is and what it needs to have in order to be considered a formal one
2. Provide the initial setup for the evolving project that you will work on during this semester
3. Create GitHub repository to deal with storing and updating your project
4. Choose a programming language
5. Store reports separately in a way to make verification of your work simpler
6. Implement a type/class for your grammar
7. Add one function that would generate 5 valid strings from the language expressed by your given grammar
8. Implement functionality to convert an object of type Grammar to one of type Finite Automaton
9. Add a method to the Finite Automaton that checks if an input string can be obtained via state transitions from it

# Implementation Description

For this laboratory I chose Java. I was variant 26. First I started with the Grammar class.

```
import java.util.*;

public class Grammar {
    private Set<String> nonTerminals;
    private Set<String> terminals;
    private Map<String, List<String>> productions;

    public Grammar(Set<String> nonTerminals, Set<String>
terminals, Map<String, List<String>> productions) {
        this.nonTerminals = nonTerminals;
        this.terminals = terminals;
        this.productions = productions;
    }

    public List<String> generateStrings(String startSymbol, int
numStrings) {
        Set<String> Strings = new HashSet<>();
        while (Strings.size() < numStrings) {
            StringBuilder sb = new StringBuilder();
            generateString(startSymbol, sb);
            Strings.add(sb.toString());
        }
        return new ArrayList<>(Strings);
    }

    private void generateString(String symbol, StringBuilder sb)
{
        if (!productions.containsKey(symbol)) {
            sb.append(symbol);
            return;
        }
        List<String> choices = productions.get(symbol);
        String choice = choices.get(new
Random().nextInt(choices.size()));
        for (char c : choice.toCharArray()) {
            if (Character.isUpperCase(c)) {
                generateString(String.valueOf(c), sb);
            } else {
                sb.append(c);
            }
        }
    }
}
```

```

    public FiniteAutomaton toFiniteAutomaton() {
        Set<String> states = nonTerminals;
        Set<String> alphabet = terminals;
        Map<String, Map<String, String>> transitions = new
HashMap<>();
        transitions.put("S", Collections.singletonMap("d",
"A"));
        transitions.put("A", new HashMap<String, String>() {{
            put("a", "B");
            put("b", "A");
        }});
        transitions.put("B", new HashMap<String, String>() {{
            put("b", "C");
            put("d", "B");
        }});
        transitions.put("C", new HashMap<String, String>() {{
            put("c", "B");
            put("a", "A");
        }});
        FiniteAutomaton finiteAutomaton = new
FiniteAutomaton(states, alphabet, "S", new
HashSet<>(Arrays.asList("A", "B")), transitions);

        return finiteAutomaton;
    }
}

```

First we declare the attributes of the grammar (non-terminal/terminal values and productions) which were given for each variant. Then we have the constructor itself which we will use in the main function to create an instance of grammar. The next 2 functions are used to produce 5 strings using the grammar. Using a while function we create the necessary number of strings. For each iteration we create a new StringBuilder and we call the second function. The second function starts with the nonterminal value S and goes through the production choosing values until a string is completed. After that it returns the word, which is added to the array list of strings. The last function is used to transform from grammar to finite automata. First its given the alphabet and the states, after which all the transitions. At the end a new object of the type FiniteAutomaton is returned.

```

import java.util.*;

public class FiniteAutomaton {
    private Set<String> states;
    private Set<String> alphabet;
    private String startState;
    private Set<String> finalStates;
    private Map<String, Map<String, String>> transitions;

    public FiniteAutomaton(Set<String> states, Set<String>
alphabet, String startState, Set<String> finalStates,
Map<String, Map<String, String>> transitions) {
        this.states = states;
        this.alphabet = alphabet;
        this.startState = startState;
        this.finalStates = finalStates;
        this.transitions = transitions;
    }

    public boolean accepts(String word) {
        String currentState = startState;
        for (char symbol : word.toCharArray()) {
            String symbolStr = String.valueOf(symbol);
            if (!alphabet.contains(symbolStr)) {
                return false;
            }
            if (transitions.containsKey(currentState) &&
transitions.get(currentState).containsKey(symbolStr)) {
                currentState =
transitions.get(currentState).get(symbolStr);
            } else {
                return false;
            }
        }
        return finalStates.contains(currentState);
    }
}

```

This is the class for FiniteAutomaton. As attributes it's given states, alphabet, startState, finalStates and transitions. After it follows the constructor. The function "accepts" is used to check if a given string can be passed by the transitions. It starts with the startState("S"). Then after splitting the word into an array of char, it iterates through each character in the word. If the alphabet does not contain the character, then it returns false. If it does, then it checks that the transitions are in a correct order. If we get to the end, then it returns true.

```

import java.util.*;

public class Main {
    public static void main (String[] args) {
        Set<String> nonTerminals = new
HashSet<>(Arrays.asList("S", "A", "B", "C"));
        Set<String> terminals = new HashSet<>(Arrays.asList("a",
"b", "c", "d"));
        Map<String, List<String>> productions = new HashMap<>();
        productions.put("S", Collections.singletonList("dA"));
        productions.put("A", Arrays.asList("aB", "b"));
        productions.put("B", Arrays.asList("bC", "d"));
        productions.put("C", Arrays.asList("cB", "aA"));
        Grammar grammar = new Grammar(nonTerminals, terminals,
productions);

        List<String> strings = grammar.generateStrings("S", 5);
        for (String str : strings) {
            System.out.println(str);
        }

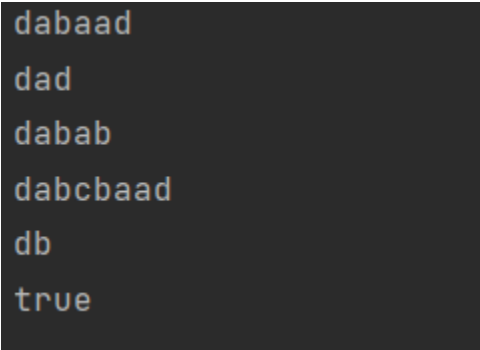
        FiniteAutomaton finiteAutomaton =
grammar.toFiniteAutomaton();

        String checkWord = "dabaabaabcbaabab";
        System.out.println(finiteAutomaton.accepts(checkWord));
    }
}

```

This is the main class with the main function. First of all we insert the non-terminal & terminal values + the productions. After that we create an instance of Grammar using “new Grammar” and giving those values. Then a list is created with the generated strings, in our case being 5 strings. “S” is given as the start state. We then print those strings. After that we create an instance of FiniteAutomaton using the function of transformation from grammar to finite automata. We get a string, in our case “dabaabaabcbaabab” and use the “accepts” function from FiniteAutomaton class to check if it passes. For this particular string it returns true.

## Screenshots



```
dabaad
dad
dabab
dabcbaad
db
true
```

## Conclusions

In this laboratory work, we implemented a grammar and converted it into a finite automaton using Java. We used a set of non-terminals, a set of terminals, and a set of production rules to define the grammar. We then converted this grammar into a finite automaton by defining states, transitions between states based on input symbols, an initial state, and a set of final states.

One of the key challenges was to correctly represent the grammar rules and the transitions in the finite automaton. We used nested data structures such as `Map<String, List<String>>` for grammar productions and `Map<String, Map<String, String>>` for state transitions to efficiently represent these concepts.

Overall, this laboratory work helped us understand the relationship between context-free grammars and finite automata, and how to implement them in Java. It also provided us with practical experience in working with formal languages and automata theory concepts.