MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

GLIGA DANIELA, FAF-223

Report

Laboratory work nr.1 Regular Grammars

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

FCIM, UTM

Chișinău – 2024

1. Theory:

A regular grammar is a type of formal grammar characterized by a set of production rules, where each production rule has the form $A \to xB$ or $A \to x$, where A is a non-terminal symbol, x is a terminal symbol or epsilon (the empty string), and B is a non-terminal symbol. Regular grammars are also known as right-linear grammars or Type 3 grammars in the Chomsky hierarchy.

Regular grammars generate regular languages, which are the simplest class of formal languages. Regular languages can be recognized by finite automata, making them amenable to efficient parsing and processing algorithms.

In a regular grammar, each production rule generates either a single terminal symbol or a terminal symbol followed by a single non-terminal symbol. This restriction ensures that productions are simple and linear, contributing to the simplicity of regular languages.

Regular grammars are closely related to regular expressions, which are concise textual representations of regular languages. Regular expressions provide a compact and expressive way to define patterns within strings, and they can be directly converted to equivalent regular grammars and vice versa.

Regular grammars belong to the lowest level of the Chomsky hierarchy, which categorizes formal grammars based on their generative power. Understanding regular grammars provides a foundational understanding of more complex grammar classes such as context-free grammars and context-sensitive grammars.

While regular languages are suitable for representing many practical problems, they have limitations in expressing certain types of languages that require more complex nested structures or context-sensitivity. Recognizing these limitations is crucial for selecting appropriate formalisms for modeling real-world phenomena.

2. 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. You can deal with each laboratory work as a separate task or project to demonstrate your understanding of the given themes, but you also can deal with labs as stages of making your own big solution, your own project. Do the following:

a. Create GitHub repository to deal with storing and updating your project;

b. Choose a programming language. Pick one that will be easiest for dealing with your tasks, you need to learn how to solve the problem itself, not everything around the problem (like setting up the project, launching it correctly and etc.);

c. Store reports separately in a way to make verification of your work simpler (duh)

3. According to your variant number, get the grammar definition and do the following:

a. Implement a type/class for your grammar;

b. Add one function that would generate 5 valid strings from the language expressed by your given grammar;

c. Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;

d. For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it;

Variant 12:

VN={S, F, D},

VT={a, b, c},

P={

S → aF

F → bF

F → cD

S → bS

D → cS

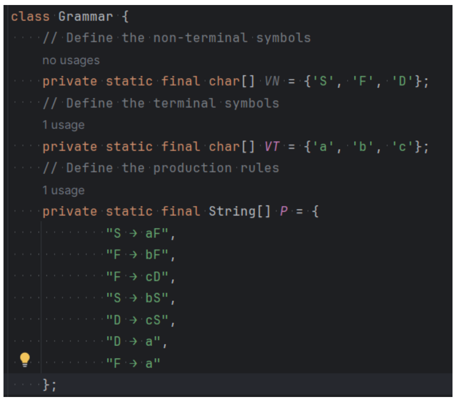
D → a

F → a

}

3. Implementation description:

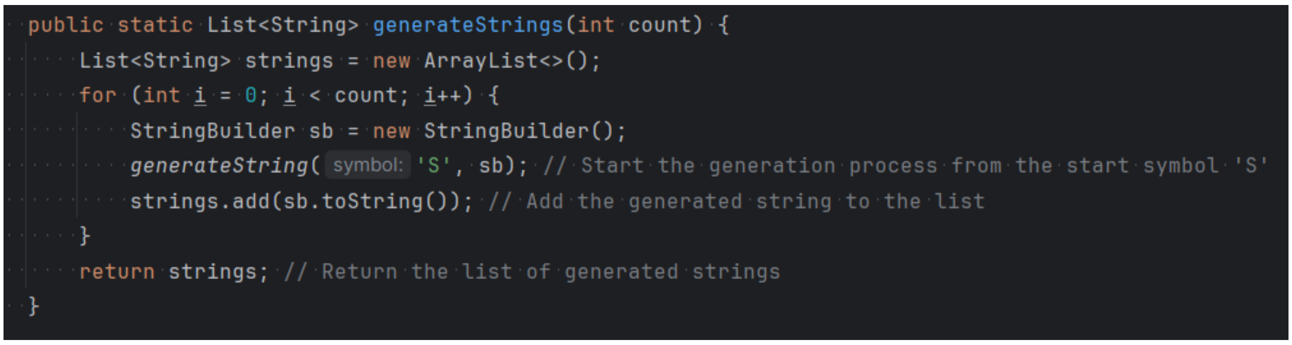
Grammar Class: This class represents a context-free grammar. It contains static arrays to define non-terminal symbols (VN), terminal symbols (VT), and production rules (P). It also includes methods to generate valid strings based on the grammar rules and to check if a symbol is terminal.



```java
class Grammar {
    // Define the non-terminal symbols
    no usages
    private static final char[] VN = {'S', 'F', 'D'};
    // Define the terminal symbols
    1 usage
    private static final char[] VT = {'a', 'b', 'c'};
    // Define the production rules
    1 usage
    private static final String[] P = {
            "S → aF",
            "F → bF",
            "F → cD",
            "S → bS",
            "D → cS",
            "D → a",
            "F → a"
    };
```

Figure 1. Grammar Class

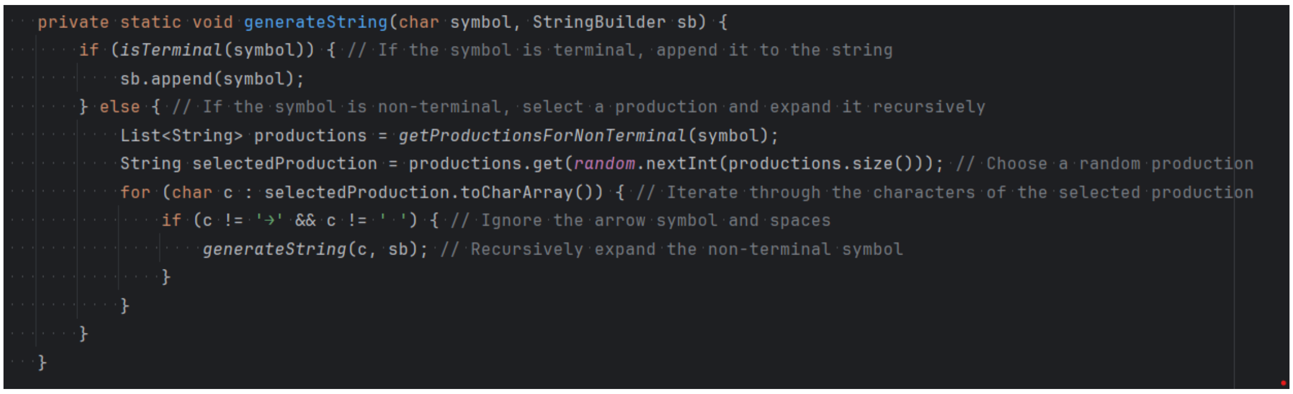
generateStrings(int count): This method generates a specified number of valid strings by iteratively calling the generateString method, starting from the start symbol 'S' and adding the generated strings to a list.



```java
public static List<String> generateStrings(int count) {
    List<String> strings = new ArrayList<>();
    for (int i = 0; i < count; i++) {
        StringBuilder sb = new StringBuilder();
        generateString( symbol: 'S', sb); // Start the generation process from the start symbol 'S'
        strings.add(sb.toString()); // Add the generated string to the list
    }
    return strings; // Return the list of generated strings
}
```

Figure 2. generateStrings Method

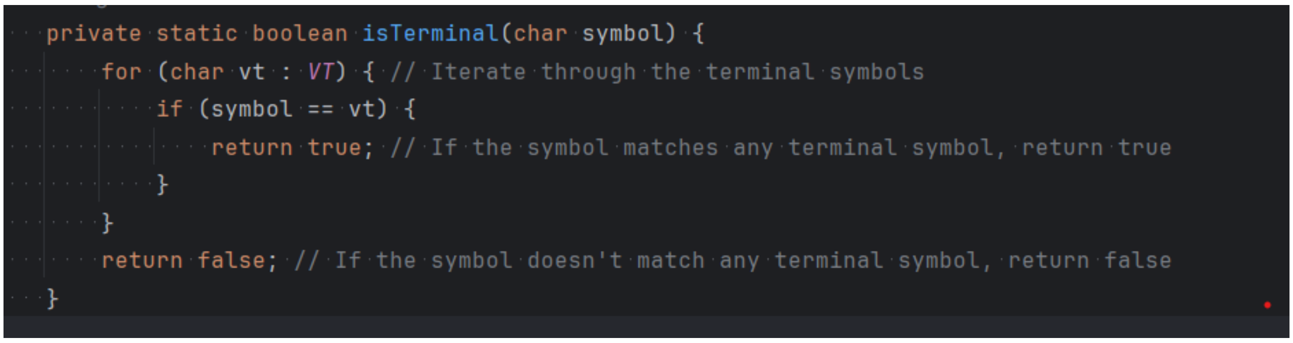
generateString(char symbol, StringBuilder sb): This recursive method generates a string following the grammar rules. It expands non-terminal symbols by randomly selecting a production rule and recursively expanding each symbol until only terminal symbols remain.



```java
private static void generateString(char symbol, StringBuilder sb) {
    if (isTerminal(symbol)) { // If the symbol is terminal, append it to the string
        sb.append(symbol);
    } else { // If the symbol is non-terminal, select a production and expand it recursively
        List<String> productions = getProductionsForNonTerminal(symbol);
        String selectedProduction = productions.get(random.nextInt(productions.size())); // Choose a random production
        for (char c : selectedProduction.toCharArray()) { // Iterate through the characters of the selected production
            if (c != '→' && c != ' ') { // Ignore the arrow symbol and spaces
                generateString(c, sb); // Recursively expand the non-terminal symbol
            }
        }
    }
}
```

Figure 3. generateString Method

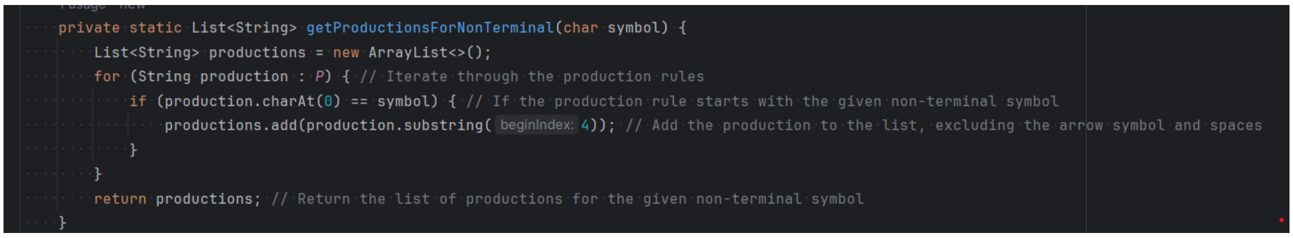
isTerminal(char symbol): This method checks if a given symbol is terminal by iterating through the terminal symbols array and comparing each symbol with the given one.



```java
private static boolean isTerminal(char symbol) {
    for (char vt : VT) { // Iterate through the terminal symbols
        if (symbol == vt) {
            return true; // If the symbol matches any terminal symbol, return true
        }
    }
    return false; // If the symbol doesn't match any terminal symbol, return false
}
```

Figure 4. isTerminal Method

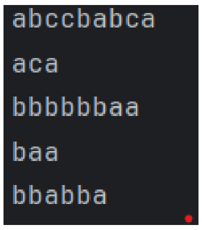
getProductionsForNonTerminal(char symbol): This method retrieves all production rules associated with a given non-terminal symbol by iterating through the production rules array and selecting those that start with the given symbol. It returns a list of productions for the given non-terminal symbol.



```java
private static List<String> getProductionsForNonTerminal(char symbol) {
    List<String> productions = new ArrayList<>();
    for (String production : P) { // Iterate through the production rules
        if (production.charAt(0) == symbol) { // If the production rule starts with the given non-terminal symbol
            productions.add(production.substring( beginIndex: 4)); // Add the production to the list, excluding the arrow symbol and spaces
        }
    }
    return productions; // Return the list of productions for the given non-terminal symbol
}
```

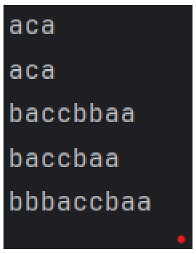
Figure 5. getProductionForNonTerminal Method

4. Screen printing of program output:



```
abccbabca
aca
bbbbbbaa
baa
bbabba
```

Figure 6: Output (random) for points a,b



```
aca
aca
baccbbaa
baccbaa
bbbaccbaa
```

Figure 7: Second output for points a,b

5. Conclusions:

In the first laboratory work on formal languages and finite automata, we delved into fundamental concepts essential for understanding the structure and behavior of languages. Through the study of context-free grammars and their associated automata, we gained insights into how languages can be formally defined and processed by machines.

We began by exploring context-free grammars, which provide a structured way to generate strings in a language through production rules. By defining terminal and non-terminal symbols along with production rules, we can systematically generate valid strings to the language's syntax rules.

By implementing a context-free grammar and using it to generate valid strings, we gained practical experience in applying theoretical concepts to real-world scenarios. Through recursive algorithms and random selection of production rules, we observed how complex strings can be generated from simple grammatical rules.

Moreover, this laboratory work provided a foundation for further exploration into advanced topics such as parsing algorithms, formal language theory, and the applications of formal languages in different fields.

Overall, this laboratory work served as an introduction to formal languages and finite automata, equipping us with essential knowledge and skills to understand and manipulate languages in computational contexts.

6. Bibliography:

1. Hopcroft, J. E., & Ullman, J. D. (1979, January 1). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Publishing Company.
2. G. (2022, April 26). *Regular grammar (Model regular grammars)*. GeeksforGeeks. Regular grammar (Model regular grammars) - GeeksforGeeks