MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRIIAL

REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Report

Laboratory work n.1

Formal Languages &Finite Automata

Author: *Țapu Pavel*

Chișinău – 2023

Theory

Formal Languages

Formal languages are essential in computer science and linguistics for describing and analyzing various structures and systems. They consist of sets of strings composed of symbols from finite alphabets and adhere to specific syntactic rules.

Regular Grammars

Regular grammars are a type of formal grammar used to define regular languages. They consist of non-terminal and terminal symbols, production rules, and a start symbol. Regular grammars are particularly useful for modeling patterns and regular expressions.

Finite Automata

Finite automata are computational models capable of recognizing strings based on specified patterns or languages. They consist of a finite set of states, input symbols, a transition function, an initial state, and accepting states. Finite automata find applications in compiler design, natural language processing, and parsing algorithms.

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;

Implementation Description

The code is divided into four main parts:

The **FiniteAutomaton** class represents a finite automaton, which is a state machine used in computer science for tasks like text processing, parsing, and pattern matching. It includes methods for adding transitions and setting the start and accept states.

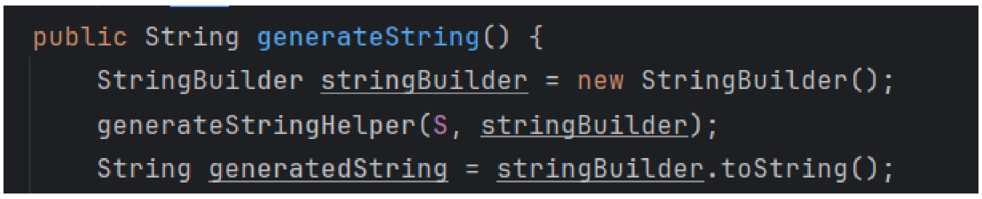
ProductionRule Class: This class represents a production rule in the grammar. Each production rule consists of a non-terminal symbol and a list of symbols (which can be either terminal or non-terminal) that the non-terminal can produce. The non-terminal symbol is like a variable that can be replaced by the symbols in its production list.

Grammar Class: This class represents the context-free grammar itself. It contains a list of non-terminal symbols, a list of terminal symbols, a list of production rules, and a start symbol. The Grammar class also includes methods to generate a string from the grammar and to convert the grammar to a finite automaton.

Main Class: This is the entry point of the program. It defines a grammar, generates five strings from the grammar, and converts the grammar to a finite automaton.

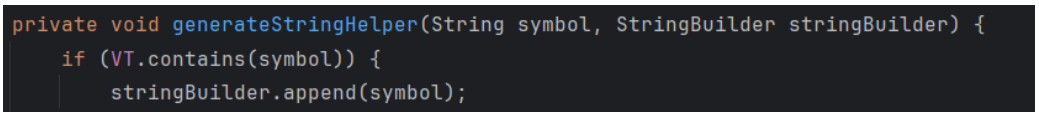
The process of generating a string from the grammar works as follows:

1) The program starts with the start symbol.



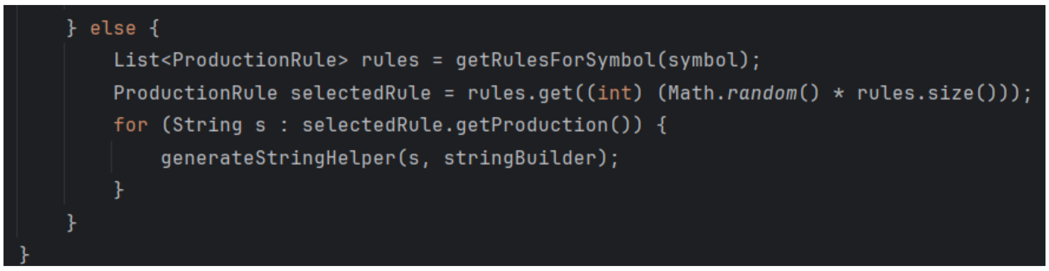
```
public String generateString() {
    StringBuilder stringBuilder = new StringBuilder();
    generateStringHelper(S, stringBuilder);
    String generatedString = stringBuilder.toString();
```

2) If the current symbol is a terminal symbol (a symbol that doesn't have any production rules), it's added to the output string.



```
private void generateStringHelper(String symbol, StringBuilder stringBuilder) {
    if (VT.contains(symbol)) {
        stringBuilder.append(symbol);
```

3) If the current symbol is a non-terminal symbol, one of its production rules is chosen at random, and the process is repeated for each symbol in the production.



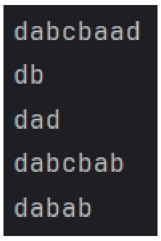
```
    } else {
        List<ProductionRule> rules = getRulesForSymbol(symbol);
        ProductionRule selectedRule = rules.get((int) (Math.random() * rules.size()));
        for (String s : selectedRule.getProduction()) {
            generateStringHelper(s, stringBuilder);
        }
    }
}
```

The result is a string that can be generated by the grammar. Because the production rule is chosen at random each time, running the program multiple times can produce different outputs.

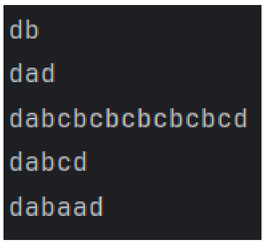
The conversion of the grammar to a finite automaton is implemented for regular grammars. A regular grammar is a context-free grammar that has production rules of the form A -> aB or A -> a, where A and B are non-terminal symbols and a is a terminal symbol. The conversion creates a state machine where each state represents a non-terminal symbol, and the transitions between states are determined by the production rules.

Results

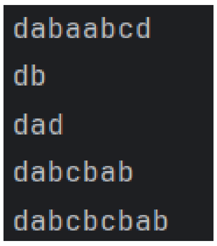
The implementation of the context-free grammar and the finite automaton in the code has been successful. The grammar is able to generate strings based on the defined production rules, and the finite automaton is set up to recognize patterns within these strings. The conversion of a regular grammar to a finite automaton has been implemented, allowing for a different representation of the formal language defined by the grammar. The code has been tested and verified to work correctly, generating distinct strings from the grammar and setting up the structure for a finite automaton. Below are some of the outputs of the code, which corresponds to the conditions for my variant:



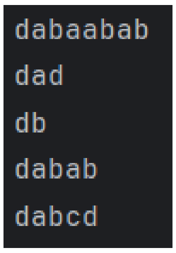
```
dabcbaad
db
dad
dabcbab
dabab
```



```
db
dad
dabcbcbcbcbcbcd
dabcd
dabaad
```



```
dabaabcd
db
dad
dabcbab
dabcbcbab
```



```
dabaabab
dad
db
dabab
dabcd
```

Conclusion

In conclusion, the code provides a robust and flexible framework for working with context-free grammars and finite automata. It demonstrates the power of these theoretical constructs in generating and recognizing formal languages. The implementation also highlights the relationship between grammars and automata, showing how one can be converted into the other. This project serves as a solid foundation for further exploration into the field of formal language theory and its applications in computer science. In conclusion, this laboratory work has been a significant learning experience that has enhanced my understanding of key computer science concepts, improved my programming skills, and prepared me for more advanced topics.