Laboratory work nr.1
Regular Grammars

Course: Formal Languages & Finite Automata
Author: Munteanu Ecaterina

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;

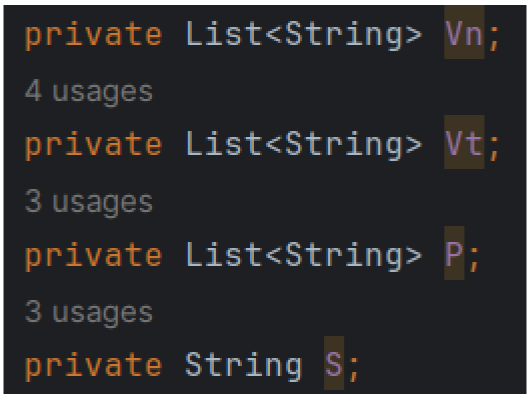
   b. Add one function that would generate 5 valid strings from the language expressed by your given grammar;

   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;

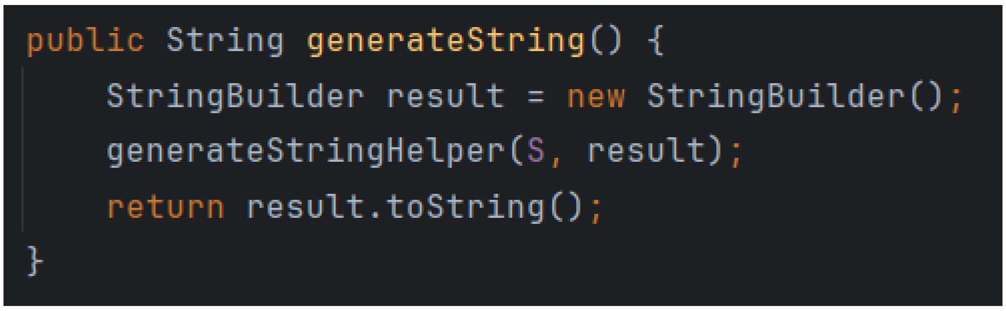
Implementation:

To do this laboratory work, first step was to create the Grammar class (using Java), which we can use as a blueprint for all the Grammars that we can have. The attributes that I used are the following:



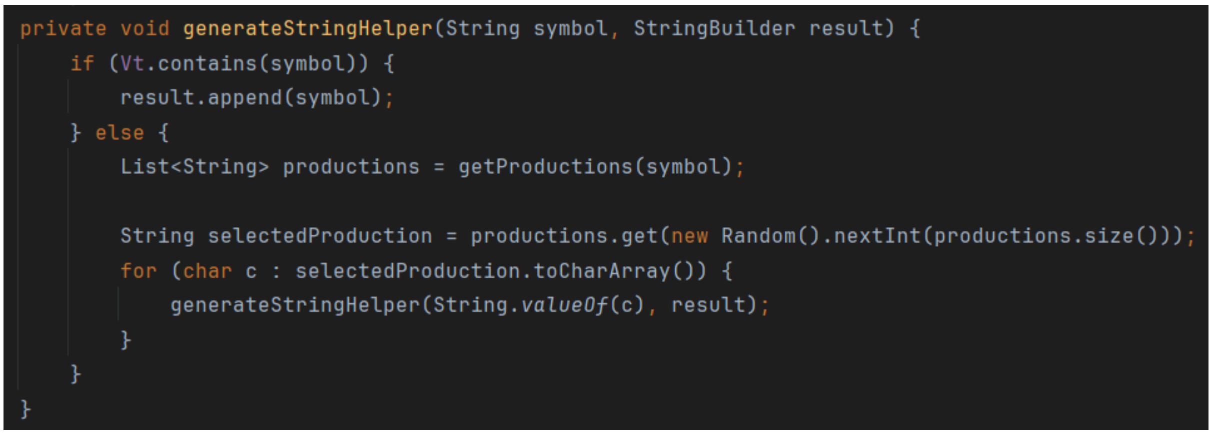
```java
private List<String> Vn;
4 usages
private List<String> Vt;
3 usages
private List<String> P;
3 usages
private String S;
```

It is clear from the naming what each is representing.
The main method of this class is generateString():



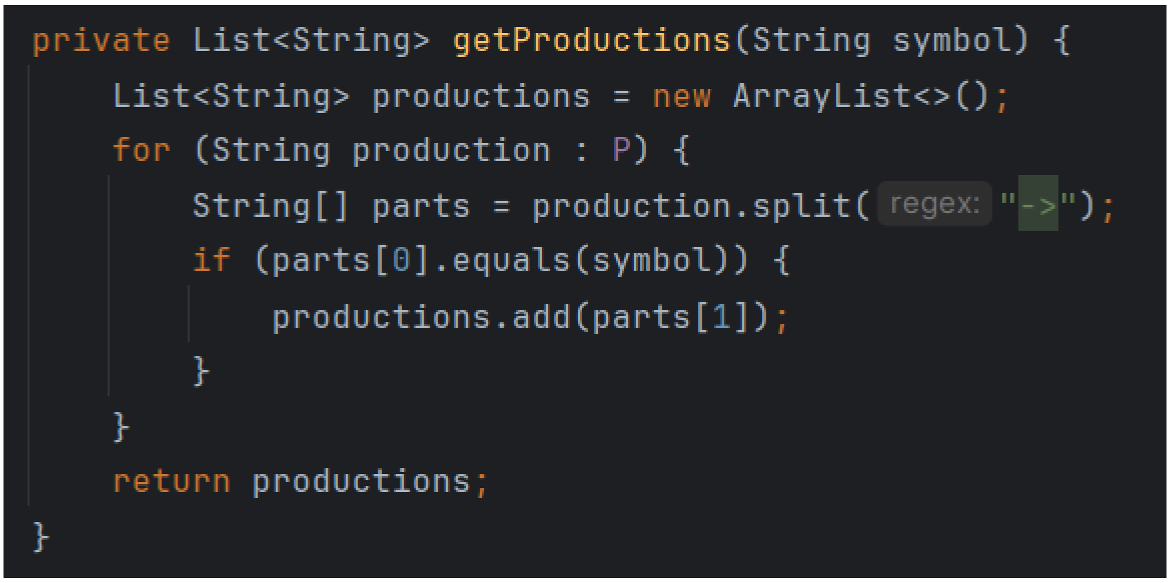
```java
public String generateString() {
    StringBuilder result = new StringBuilder();
    generateStringHelper(S, result);
    return result.toString();
}
```

As we can see, it uses another method - generateStringHelper(), which takes the initial symbol S and a StringBuilder object, in my case called result. This method look like this:



```java
private void generateStringHelper(String symbol, StringBuilder result) {
    if (Vt.contains(symbol)) {
        result.append(symbol);
    } else {
        List<String> productions = getProductions(symbol);

        String selectedProduction = productions.get(new Random().nextInt(productions.size()));
        for (char c : selectedProduction.toCharArray()) {
            generateStringHelper(String.valueOf(c), result);
        }
    }
}
```

If the symbol we transmitted is a terminal one, it just appends it to the result and returns. But if it is a non-terminal symbol, then we find all the productions this symbol can generate (using another method getProductions()) and randomly choose one of them. Getting productions look the following:

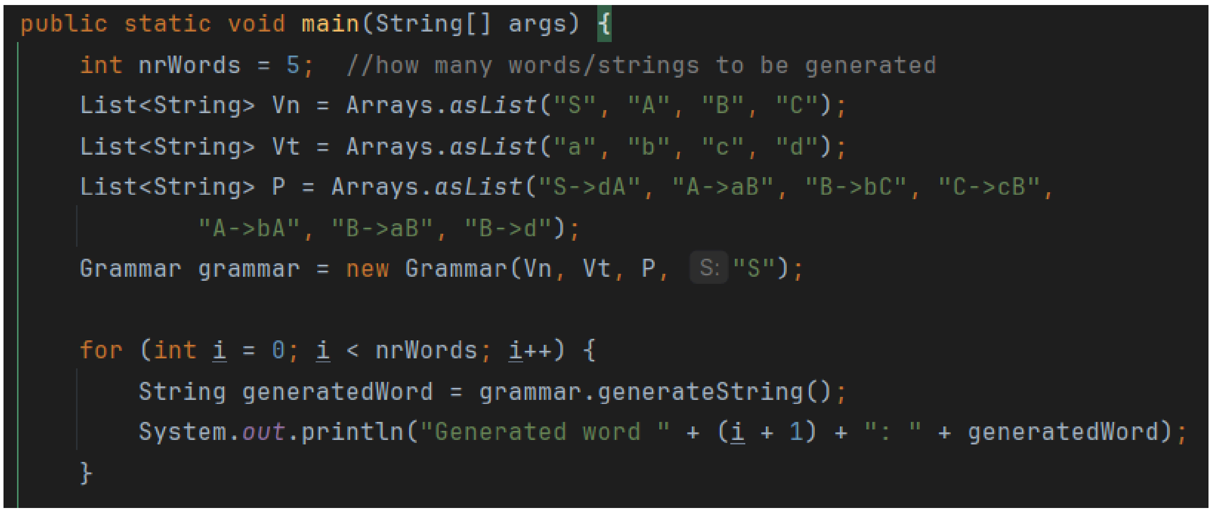


```java
private List<String> getProductions(String symbol) {
    List<String> productions = new ArrayList<>();
    for (String production : P) {
        String[] parts = production.split( regex: "->");
        if (parts[0].equals(symbol)) {
            productions.add(parts[1]);
        }
    }
    return productions;
}
```

It just splits the production into the right and left sides, and checks if the left side is the one we are looking for, and if that’s true, we add this production to our list.

Because generateString() is recursive, it will stop when the right side of the production consists of only terminal symbols for all the productions, thus, generating randomly a grammatically correct string.

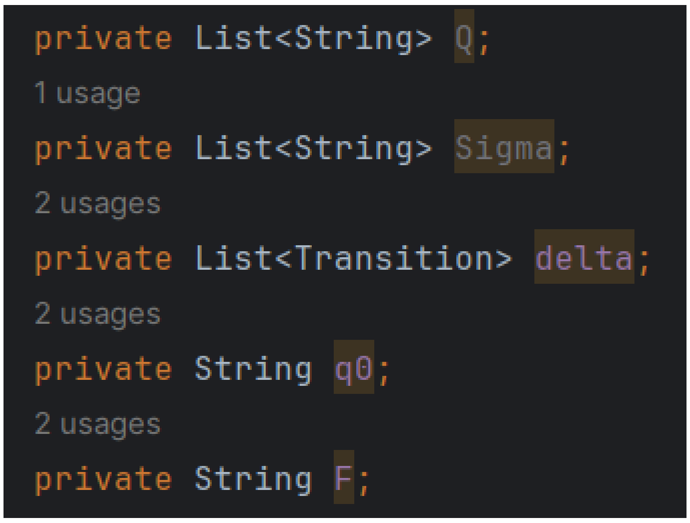
Now, using the class that I created for the moment, I can generate 5 strings in the main class:



```java
public static void main(String[] args) {
    int nrWords = 5;  //how many words/strings to be generated
    List<String> Vn = Arrays.asList("S", "A", "B", "C");
    List<String> Vt = Arrays.asList("a", "b", "c", "d");
    List<String> P = Arrays.asList("S->dA", "A->aB", "B->bC", "C->cB",
            "A->bA", "B->aB", "B->d");
    Grammar grammar = new Grammar(Vn, Vt, P,  S: "S");

    for (int i = 0; i < nrWords; i++) {
        String generatedWord = grammar.generateString();
        System.out.println("Generated word " + (i + 1) + ": " + generatedWord);
    }
```

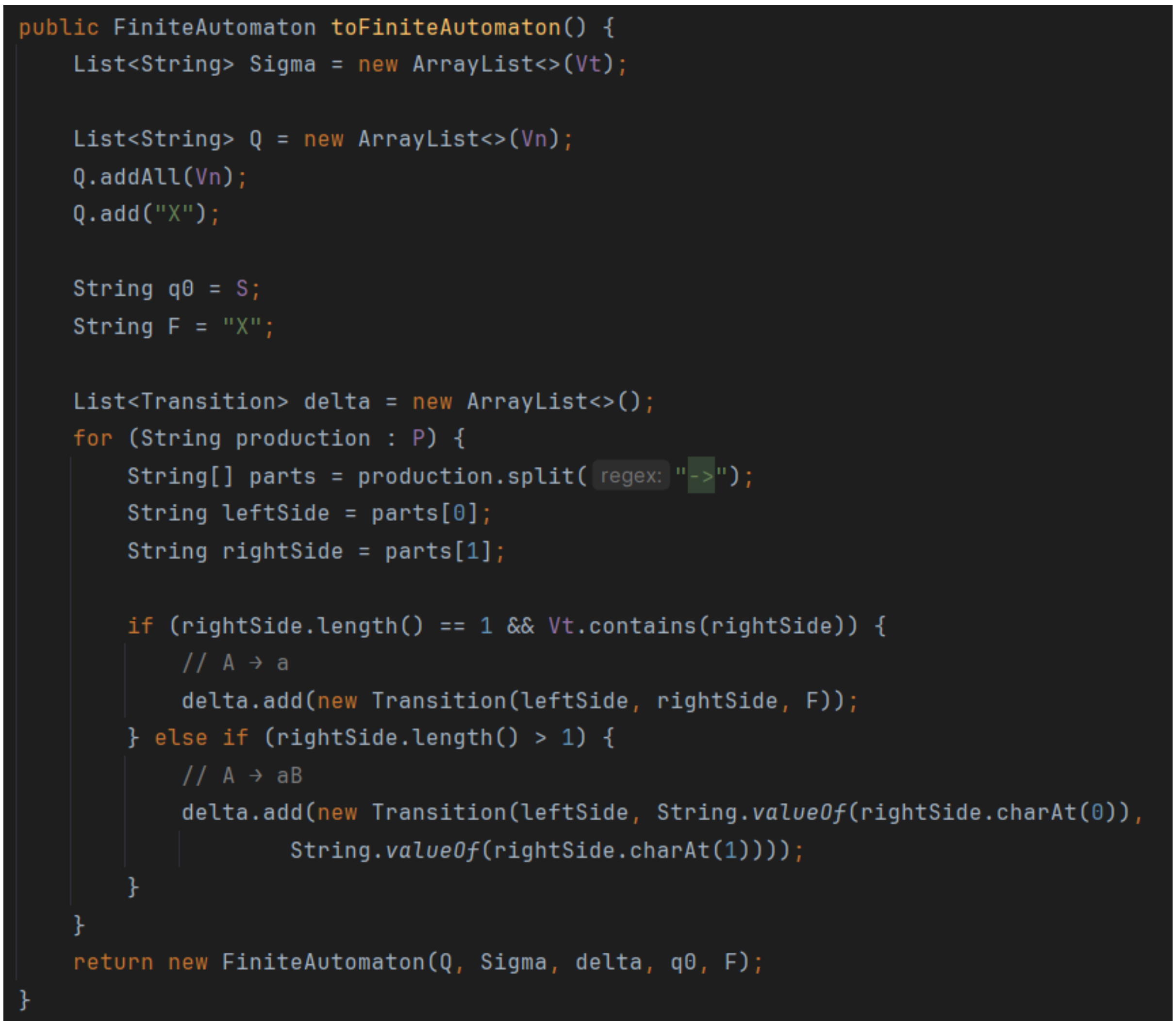
I declare and initiate a Grammar object, transmitting the conditions of my variant, and using a for loop I generate as many words as I need.

The next step is to create a FiniteAutomaton class, with the following attributes:



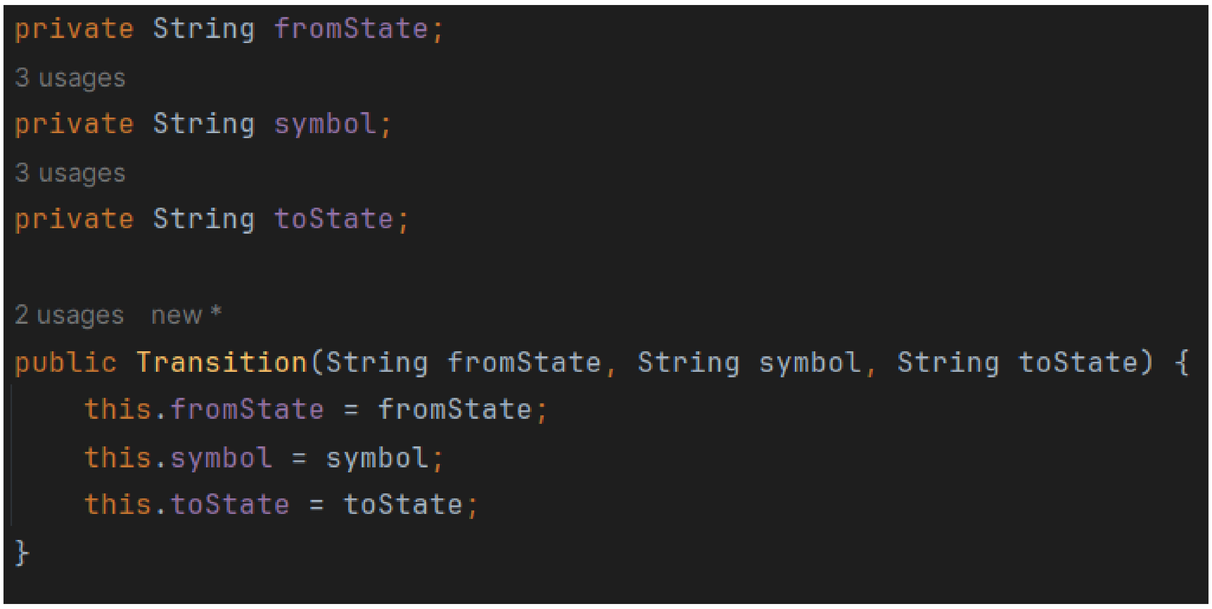
```
private List<String> Q;
1 usage
private List<String> Sigma;
2 usages
private List<Transition> delta;
2 usages
private String q0;
2 usages
private String F;
```

Using this class, we can add a method that will transform a Grammar object into a FiniteAutomaton one into the Grammar class:



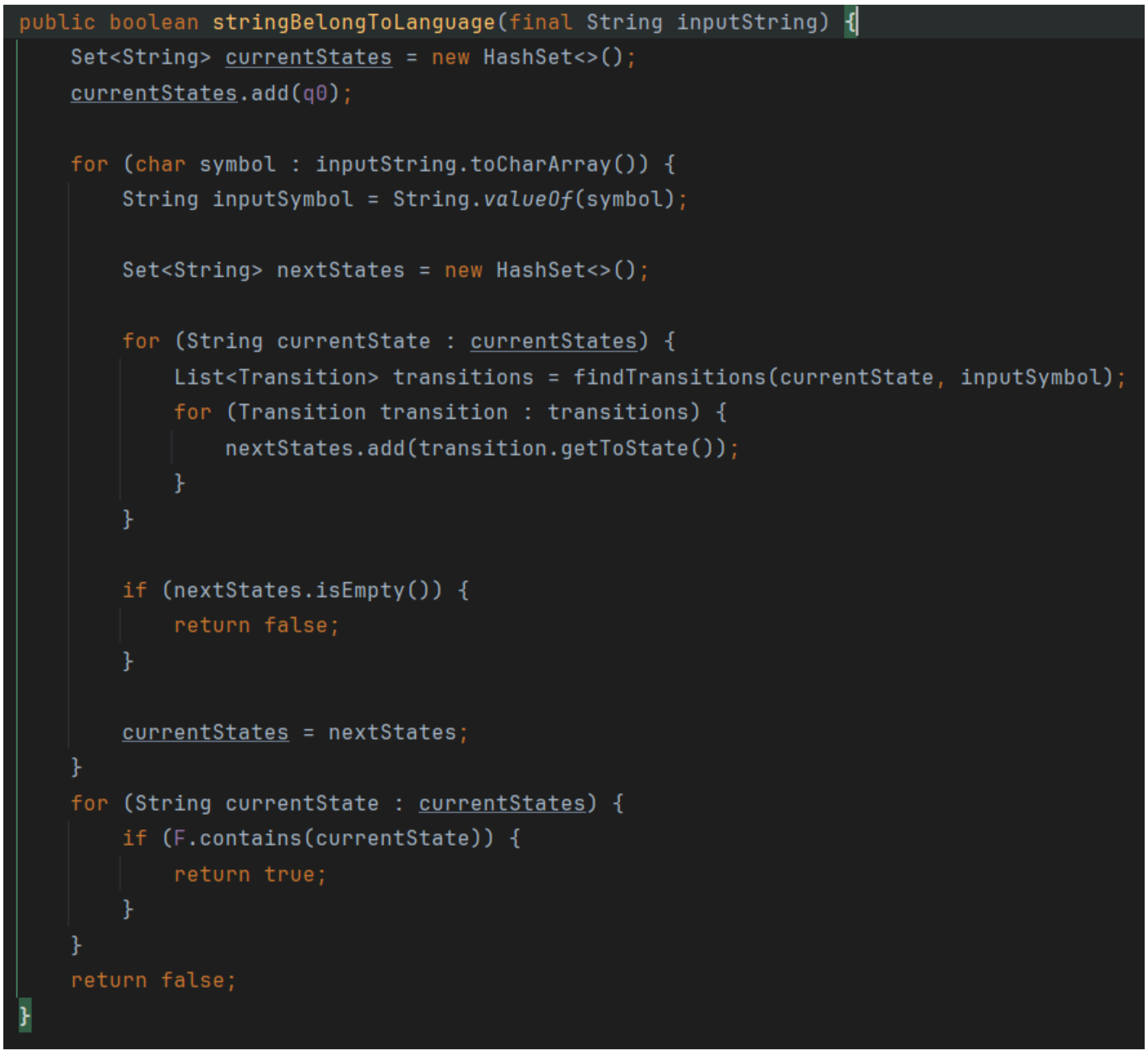
```
public FiniteAutomaton toFiniteAutomaton() {
    List<String> Sigma = new ArrayList<>(Vt);

    List<String> Q = new ArrayList<>(Vn);
    Q.addAll(Vn);
    Q.add("X");

    String q0 = S;
    String F = "X";

    List<Transition> delta = new ArrayList<>();
    for (String production : P) {
        String[] parts = production.split( regex: "->");
        String leftSide = parts[0];
        String rightSide = parts[1];

        if (rightSide.length() == 1 && Vt.contains(rightSide)) {
            // A → a
            delta.add(new Transition(leftSide, rightSide, F));
        } else if (rightSide.length() > 1) {
            // A → aB
            delta.add(new Transition(leftSide, String.valueOf(rightSide.charAt(0)),
                    String.valueOf(rightSide.charAt(1))));
        }
    }
    return new FiniteAutomaton(Q, Sigma, delta, q0, F);
}
```

Using the algorithm from our lecture guide, I created an FiniteAutomaton out of my Grammar. To represent each transition, I decided to create another helping class, with 3 simple attributes:



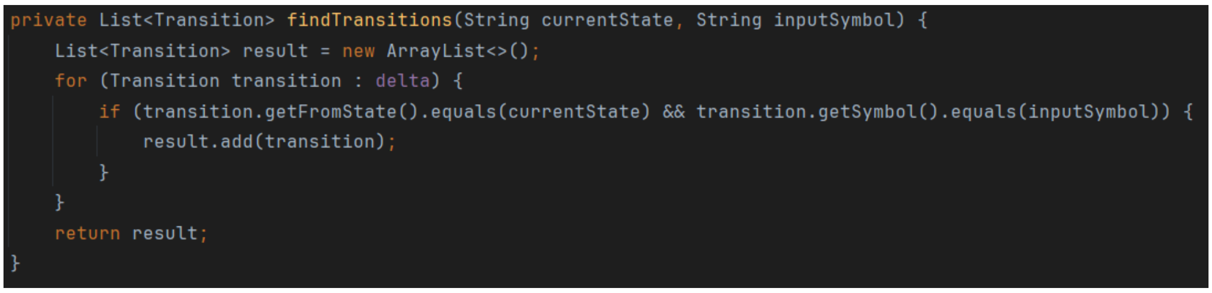
```java
private String fromState;
3 usages
private String symbol;
3 usages
private String toState;

2 usages   new *
public Transition(String fromState, String symbol, String toState) {
    this.fromState = fromState;
    this.symbol = symbol;
    this.toState = toState;
}
```

So, the delta in my FiniteAutomaton is represented by a list of Transition objects.



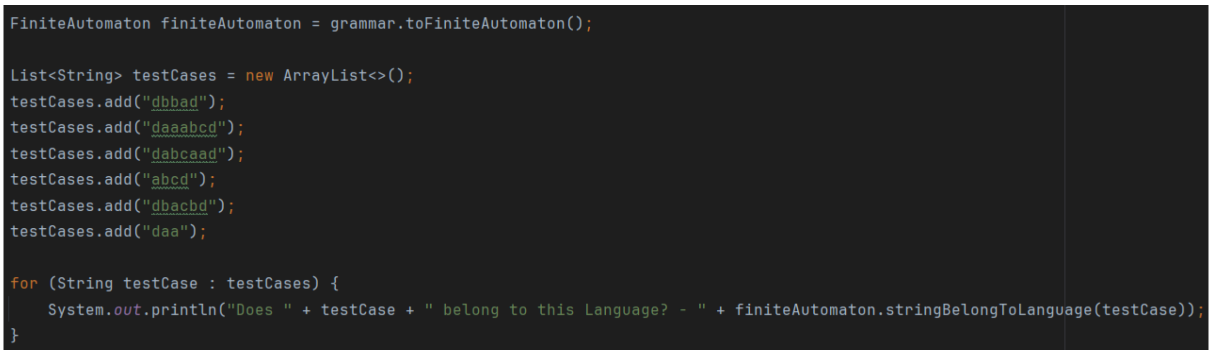
```java
public boolean stringBelongToLanguage(final String inputString) {
    Set<String> currentStates = new HashSet<>();
    currentStates.add(q0);

    for (char symbol : inputString.toCharArray()) {
        String inputSymbol = String.valueOf(symbol);

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

        for (String currentState : currentStates) {
            List<Transition> transitions = findTransitions(currentState, inputSymbol);
            for (Transition transition : transitions) {
                nextStates.add(transition.getToState());
            }
        }

        if (nextStates.isEmpty()) {
            return false;
        }

        currentStates = nextStates;
    }
    for (String currentState : currentStates) {
        if (F.contains(currentState)) {
            return true;
        }
    }
    return false;
}
```

The last step was to create a method that would check whether a string belongs to this FiniteAutomaton or not. For that it takes the symbols of the word one by one, and checks if there are appropriate transitions that contain them, using additional method findTransitions():



```java
private List<Transition> findTransitions(String currentState, String inputSymbol) {
    List<Transition> result = new ArrayList<>();
    for (Transition transition : delta) {
        if (transition.getFromState().equals(currentState) && transition.getSymbol().equals(inputSymbol)) {
            result.add(transition);
        }
    }
    return result;
}
```

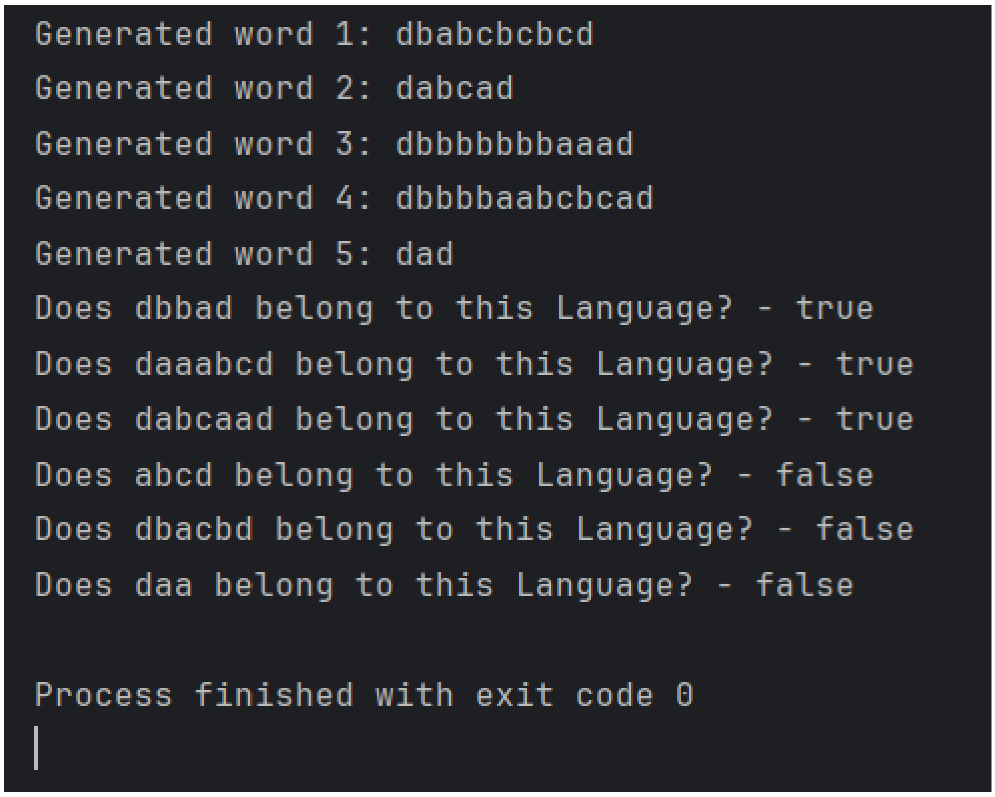
Now, we can use all these to check our test cases:



```java
FiniteAutomaton finiteAutomaton = grammar.toFiniteAutomaton();

List<String> testCases = new ArrayList<>();
testCases.add("dbbad");
testCases.add("daaabcd");
testCases.add("dabcaad");
testCases.add("abcd");
testCases.add("dbacbd");
testCases.add("daa");

for (String testCase : testCases) {
    System.out.println("Does " + testCase + " belong to this Language? - " + finiteAutomaton.stringBelongToLanguage(testCase));
}
```

Results:

The run results are the following:



```
Generated word 1: dbabcbcbcd
Generated word 2: dabcad
Generated word 3: dbbbbbbbaaad
Generated word 4: dbbbbaabcbcad
Generated word 5: dad
Does dbbad belong to this Language? - true
Does daaabcd belong to this Language? - true
Does dabcaad belong to this Language? - true
Does abcd belong to this Language? - false
Does dbacbd belong to this Language? - false
Does daa belong to this Language? - false


Process finished with exit code 0
```

Conclusions:

This laboratory work was an excellent practical exercise in understanding the core concepts of formal languages and understanding their characteristics through specific examples. We learned how we can use all these notions to create new strings, but which have a specific rule that they follow. Also, it helped me learn how to transform a Grammar into a Finite Automaton, and use this to check the correctness of certain words. It is a good base for future laboratory works.