# Laboratory work nr.2
# Finite Automata

Course: Formal Languages & Finite Automata
Student:Munteanu Ecaterina

## Objectives:

1. Understand what an automaton is and what it can be used for.

2. Continuing the work in the same repository and the same project, the following need to be added: a. Provide a function in your grammar type/class that could classify the grammar based on Chomsky hierarchy.

   b. For this you can use the variant from the previous lab.

3. According to your variant number (by universal convention it is register ID), get the finite automaton definition and do the following tasks:

   a. Implement conversion of a finite automaton to a regular grammar.

   b. Determine whether your FA is deterministic or non-deterministic.

   c. Implement some functionality that would convert an NDFA to a DFA.

   d. Represent the finite automaton graphically (Optional, and can be considered as a bonus point):

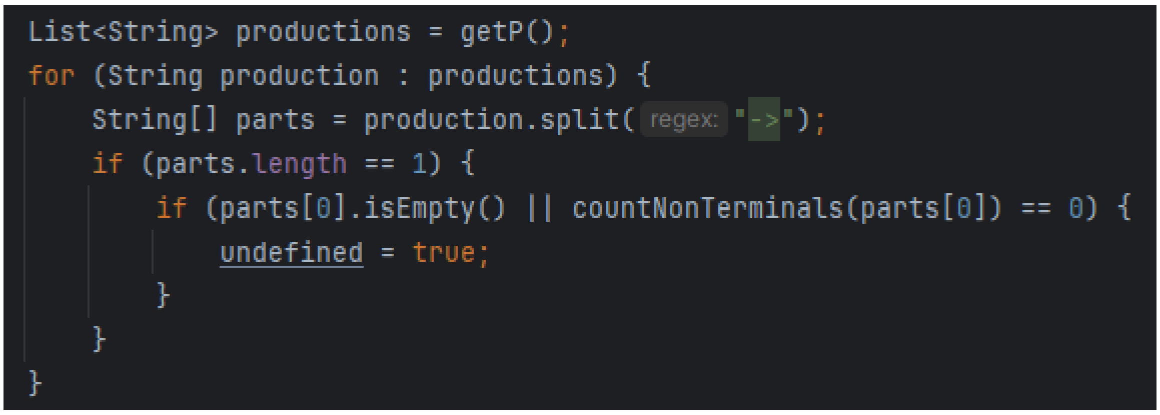
You can use external libraries, tools or APIs to generate the figures/diagrams.

Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

## Implementation:

***Task 2a:***

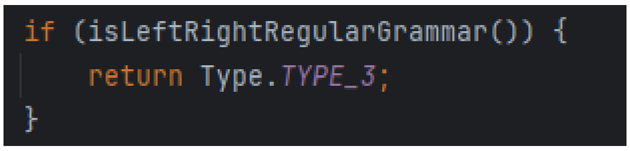
To implement this task I needed to introduce a new method getType in the Grammar class. Firstly it checks if the left side is not an empty string and if it has at least one non-terminal symbol:



```java
List<String> productions = getP();
for (String production : productions) {
    String[] parts = production.split( regex: "->");
    if (parts.length == 1) {
        if (parts[0].isEmpty() || countNonTerminals(parts[0]) == 0) {
            undefined = true;
        }
    }
}
```

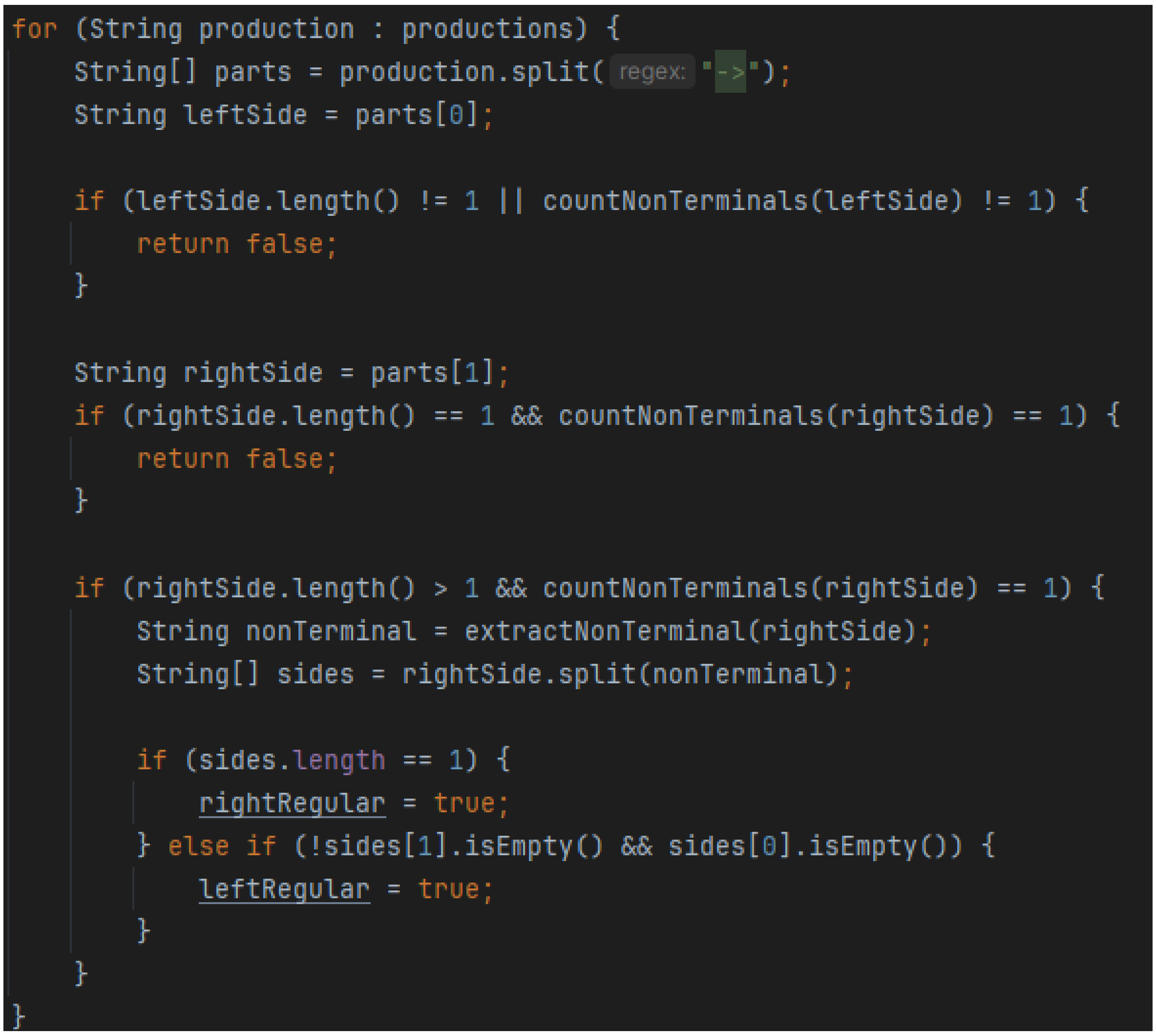
If undefined == true, the type that will be returned is UNDEFINED.

Secondly, my code checks whether the grammar is left or right regular, and if the method returns true, we automatically can return TYPE_3:



```java
if (isLeftRightRegularGrammar()) {
    return Type.TYPE_3;
}
```

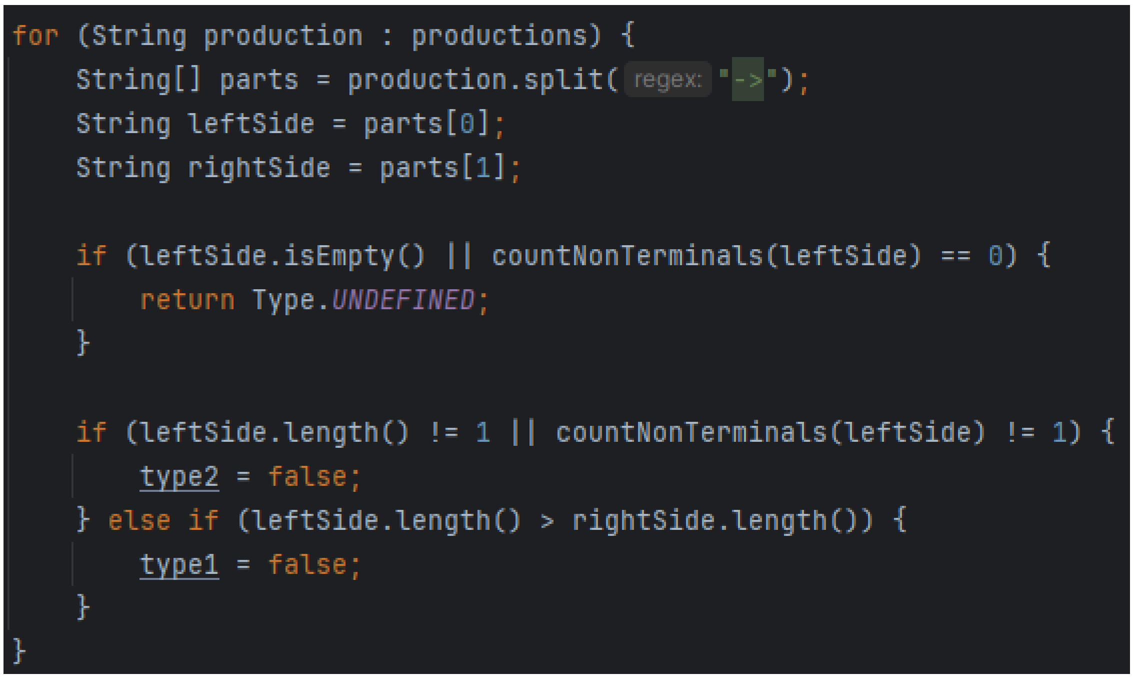
The main part of this method is:



```java
for (String production : productions) {
    String[] parts = production.split( regex: "->");
    String leftSide = parts[0];

    if (leftSide.length() != 1 || countNonTerminals(leftSide) != 1) {
        return false;
    }

    String rightSide = parts[1];
    if (rightSide.length() == 1 && countNonTerminals(rightSide) == 1) {
        return false;
    }

    if (rightSide.length() > 1 && countNonTerminals(rightSide) == 1) {
        String nonTerminal = extractNonTerminal(rightSide);
        String[] sides = rightSide.split(nonTerminal);

        if (sides.length == 1) {
            rightRegular = true;
        } else if (!sides[1].isEmpty() && sides[0].isEmpty()) {
            leftRegular = true;
        }
    }
}
```

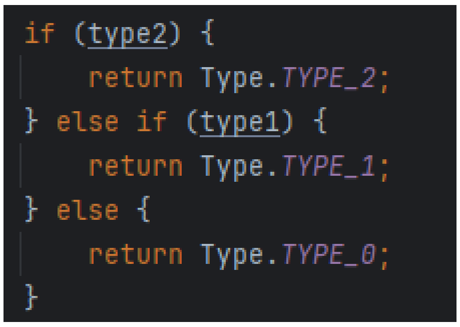
It checks for conditions for each type of grammar, thus returning false when any conditions are not satisfied.

The part of the code is another for, that checks the conditions for the first and second grammar types:



```
for (String production : productions) {
    String[] parts = production.split( regex: "->");
    String leftSide = parts[0];
    String rightSide = parts[1];

    if (leftSide.isEmpty() || countNonTerminals(leftSide) == 0) {
        return Type.UNDEFINED;
    }

    if (leftSide.length() != 1 || countNonTerminals(leftSide) != 1) {
        type2 = false;
    } else if (leftSide.length() > rightSide.length()) {
        type1 = false;
    }
}
```

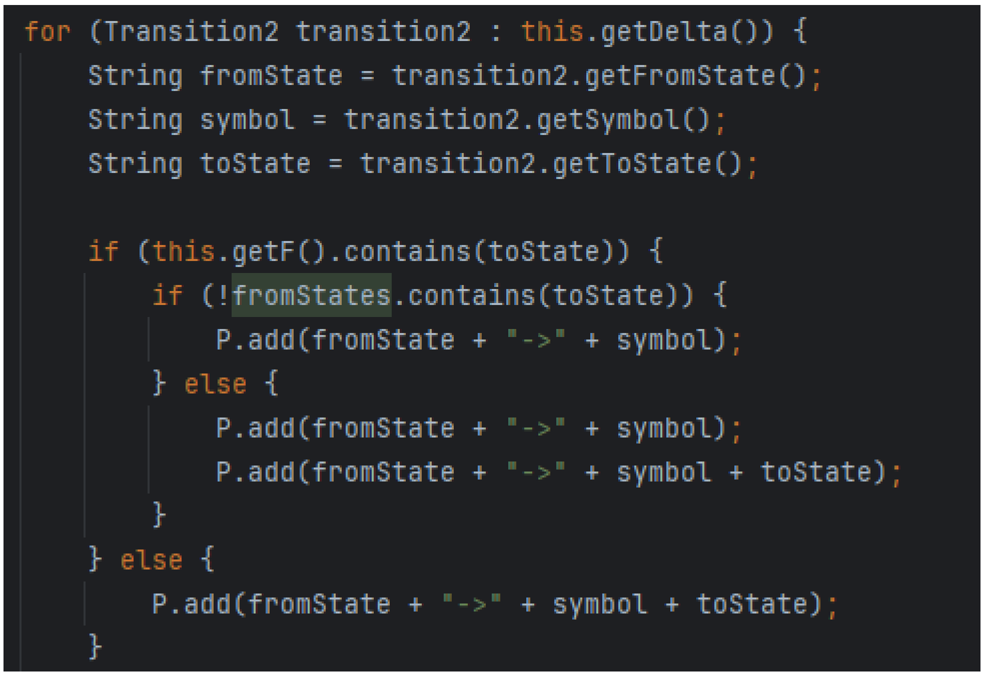
Lastly, when each of the local boolean variables has its value, an if-else chain checks them to return the appropriate response:



```
if (type2) {
    return Type.TYPE_2;
} else if (type1) {
    return Type.TYPE_1;
} else {
    return Type.TYPE_0;
}
```

***Task 3a:***

For this task I also created a method, but in the FiniteAutomaton class. After assigning the state to the non-terminal symbols, the alphabet to the terminal symbols and getting the starting state, I needed to get the information out of the transactions:

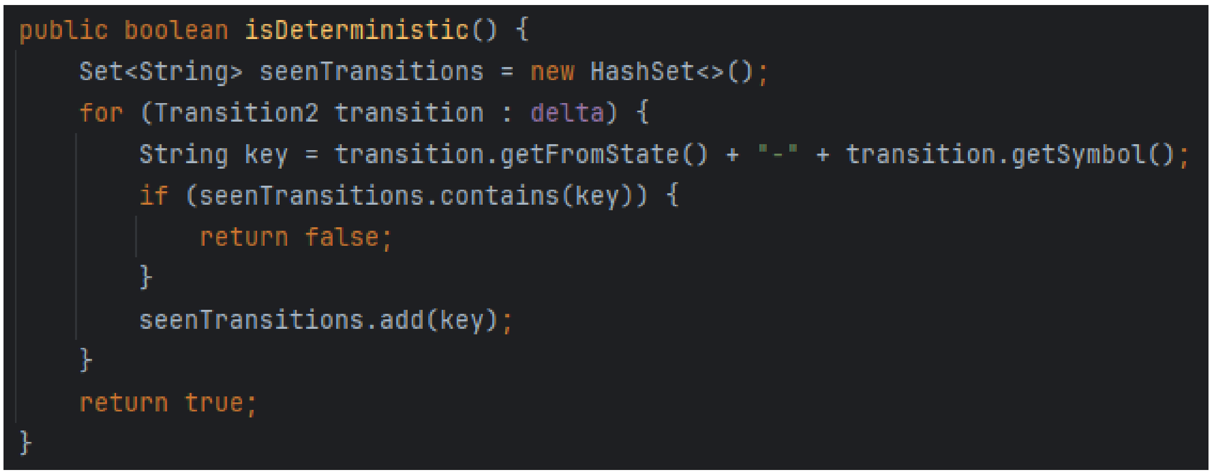


```java
for (Transition2 transition2 : this.getDelta()) {
    String fromState = transition2.getFromState();
    String symbol = transition2.getSymbol();
    String toState = transition2.getToState();

    if (this.getF().contains(toState)) {
        if (!fromStates.contains(toState)) {
            P.add(fromState + "->" + symbol);
        } else {
            P.add(fromState + "->" + symbol);
            P.add(fromState + "->" + symbol + toState);
        }
    } else {
        P.add(fromState + "->" + symbol + toState);
    }
```

In this **for** I extracted the states and the symbols to create the productions for the grammar. If the state on the right part is a final state and it has other transitions further, then I add a production with and without it, but if it doesn't, then only the production without it.

***Task 3b:***

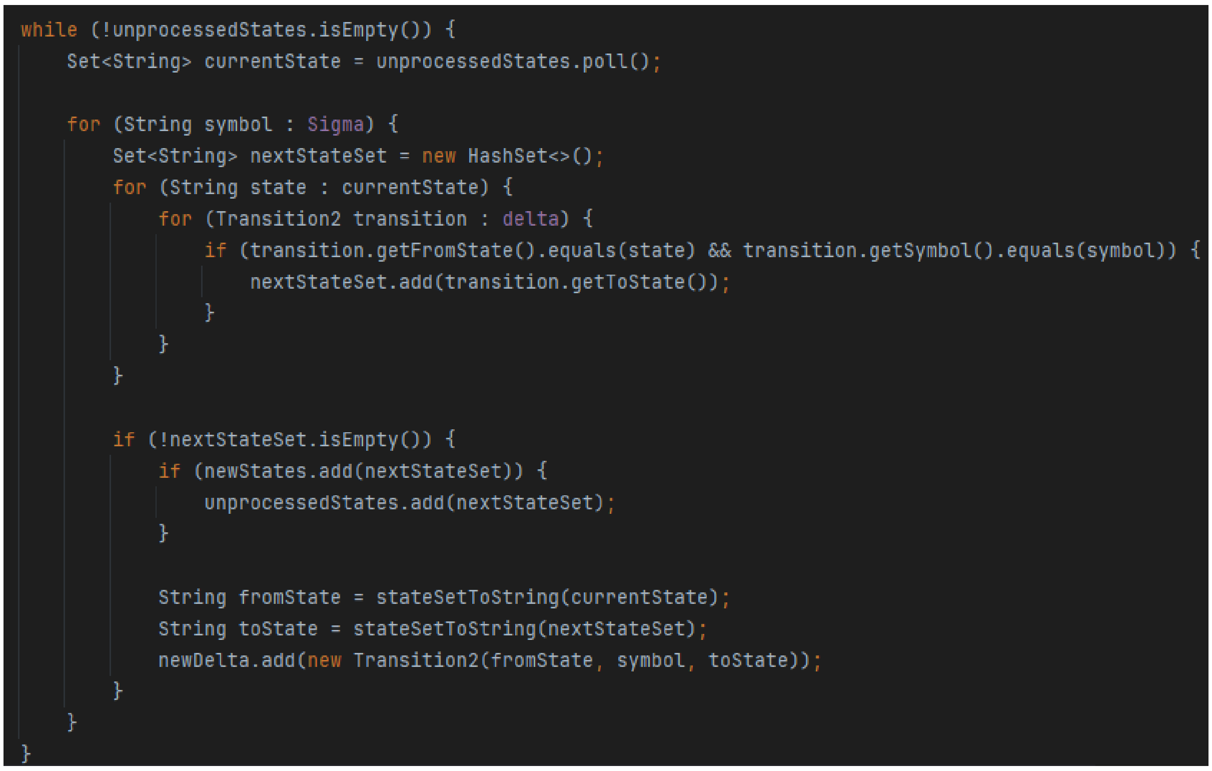
Here the method is pretty straightforward. It checks if any state has multiple transitions with the same symbol, and if it finds one, then it returns **false**:



```java
public boolean isDeterministic() {
    Set<String> seenTransitions = new HashSet<>();
    for (Transition2 transition : delta) {
        String key = transition.getFromState() + "-" + transition.getSymbol();
        if (seenTransitions.contains(key)) {
            return false;
        }
        seenTransitions.add(key);
    }
    return true;
}
```

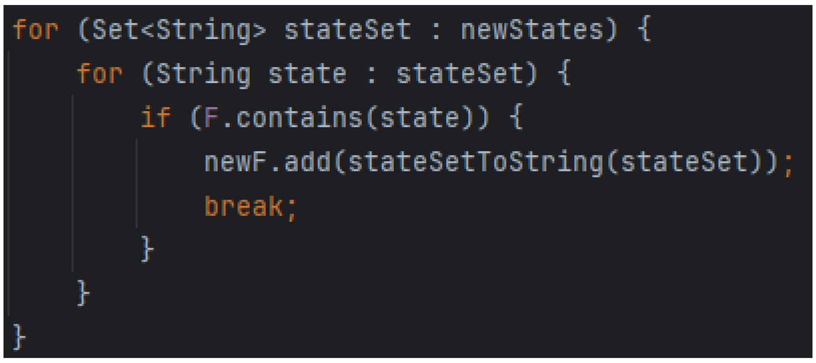
***Task 3c:***

This task is a more voluminous one. The method starts with the initial state set of the NFA and iteratively looks for possible next states for each symbol in the alphabet using a breadth-first search algorithm.



```java
while (!unprocessedStates.isEmpty()) {
    Set<String> currentState = unprocessedStates.poll();

    for (String symbol : Sigma) {
        Set<String> nextStateSet = new HashSet<>();
        for (String state : currentState) {
            for (Transition2 transition : delta) {
                if (transition.getFromState().equals(state) && transition.getSymbol().equals(symbol)) {
                    nextStateSet.add(transition.getToState());
                }
            }
        }

        if (!nextStateSet.isEmpty()) {
            if (newStates.add(nextStateSet)) {
                unprocessedStates.add(nextStateSet);
            }

            String fromState = stateSetToString(currentState);
            String toState = stateSetToString(nextStateSet);
            newDelta.add(new Transition2(fromState, symbol, toState));
        }
    }
}
```

For each symbol, it determines the set of next states reachable from the current state set based on transitions in the NFA. If the set of next states is not empty, it adds it to the set of new states and updates the DFA's transitions. After processing all symbols for a state set, it continues the exploration for newly found sets. It identifies and adds final states to the DFA based on whether they contain a final state of the NFA:



```java
for (Set<String> stateSet : newStates) {
    for (String state : stateSet) {
        if (F.contains(state)) {
            newF.add(stateSetToString(stateSet));
            break;
        }
    }
}
```

The method converts sets of states to strings for the DFA's states, final states, and transitions. Finally, it returns a new FiniteAutomaton representing the equivalent DFA.

## Conclusions:

The code for the laboratory work includes subjects such as Chomsky hierarchy classification, conversion of finite automaton to regular grammar, determination of determinism in finite automata, and the conversion of non-deterministic finite automata to deterministic ones. This code helps the understanding and manipulation of formal languages, contributing to our experience in this laboratory work.