**Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.**

**Course: Formal Languages & Finite Automata**

**Author: Vladlen Mîțu**

**Theory:**

A **finite automaton** is a mathematical model used to represent processes with a defined start and end. It works similarly to a state machine, where transitions occur between states based on input symbols. If an automaton allows multiple possible transitions for the same input, it is **non-deterministic**; otherwise, it is **deterministic**. Non-deterministic automata can be converted into deterministic ones using specific algorithms, ensuring a more predictable system behavior.

**Chomsky Hierarchy**

1. Type 3 - Regular Languages (Finite Automata, simple patterns)

2. Type 2 - Context-Free (Pushdown Automata, programming languages)

3. Type 1 - Context-Sensitive (more complex grammars)

4. Type 0 - Recursively Enumerable (Turing Machines, most powerful)

**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.

1. 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.

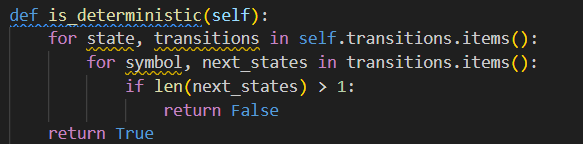
d. Represent the finite automaton graphically (Optional, and can be considered as a ***bonus point***):

* + 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.

Please consider that all elements of the task 3 can be done manually, writing a detailed report about how you've done the conversion and what changes have you introduced. In case if you'll be able to write a complete program that will take some finite automata and then convert it to the regular grammar - this will be **a good bonus point**.

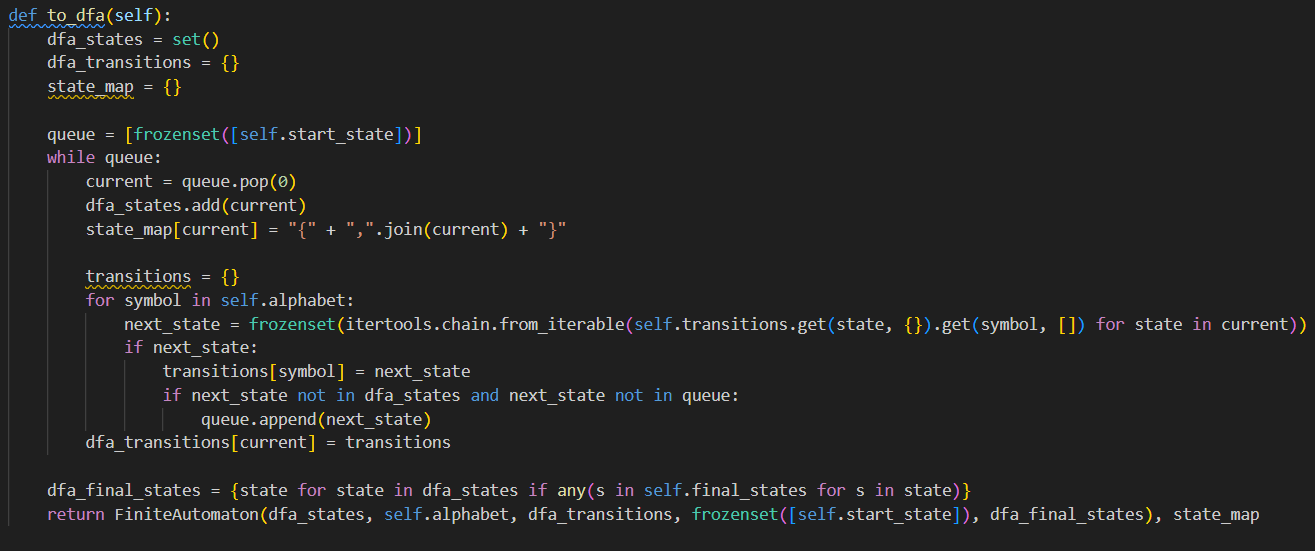
**Implementation description:**

The is\_deterministic method in the FiniteAutomaton class checks if the finite automaton is deterministic. A deterministic finite automaton (DFA) has at most one transition for each symbol from any given state.



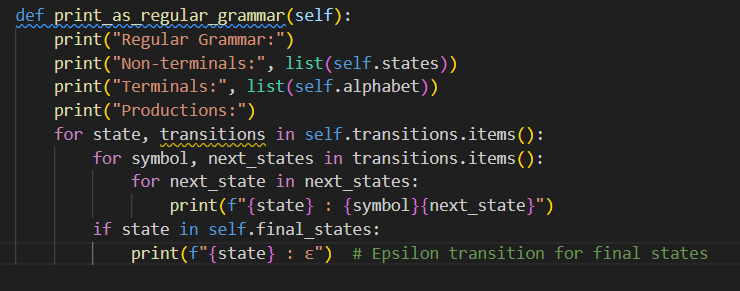
The to\_dfa method converts a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA). Here's a concise description:

1. Initializes sets for DFA states and transitions, and a state mapping dictionary.
2. Uses a queue to process states, starting with the initial state of the NFA.
3. For each state in the queue:
   * Adds the state to DFA states and maps it to a string representation.
   * Computes transitions for each symbol by combining possible transitions from constituent NFA states.
   * Adds new states to the queue if they haven't been processed.
4. Identifies DFA final states as those containing any NFA final states.
5. Returns a new FiniteAutomaton instance representing the DFA and the state mapping.



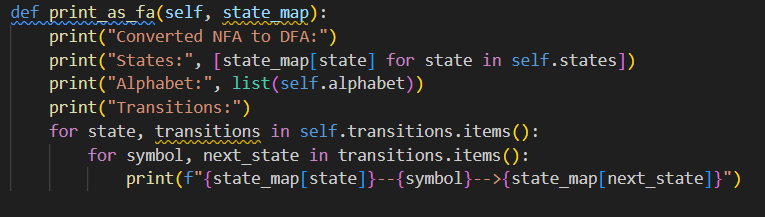
The print\_as\_regular\_grammar method in the FiniteAutomaton class prints the finite automaton as a regular grammar. Here's a breakdown of what the method does:

1. Prints the header "Regular Grammar:".
2. Prints the non-terminals, which are the states of the automaton.
3. Prints the terminals, which are the symbols in the automaton's alphabet.
4. Prints the productions:
   * Iterates over each state and its corresponding transitions.
   * For each state and symbol, iterates over the next states and prints a production rule in the format state : symbol next\_state.
   * If the state is a final state, prints an epsilon transition (state : ε).



The print\_as\_fa method in the FiniteAutomaton class prints the finite automaton as a DFA using a provided state mapping. Here's a breakdown of what the method does:

1. Prints the header "Converted NFA to DFA:".
2. Prints the states of the DFA using the state\_map to convert NFA state sets to their string representations.
3. Prints the alphabet of the automaton.
4. Prints the transitions:
   * Iterates over each state and its corresponding transitions.
   * For each state and symbol, prints a transition rule in the format state\_map[state]--symbol-->state\_map[next\_state], using the state\_map to convert state sets to their string representations.



**Conclusion:**

This code defines a FiniteAutomaton class that represents a finite automaton, which can be either deterministic (DFA) or non-deterministic (NFA). The class includes methods to:

1. **Check if the automaton is deterministic** (is\_deterministic).
2. **Convert an NFA to a DFA** (to\_dfa).
3. **Print the automaton as a regular grammar** (print\_as\_regular\_grammar).
4. **Print the DFA representation of the automaton** (print\_as\_fa).

The code also includes an example of defining an NFA with specific states, alphabet, transitions, start state, and final states. It prints the NFA as a regular grammar, checks if the NFA is deterministic, converts the NFA to a DFA if necessary, and prints the DFA.

This implementation demonstrates the conversion process from NFA to DFA and provides a clear way to visualize the automaton both as a regular grammar and as a DFA.

**References:**

1. Else Course FAF.LFA21.1
2. Finite Automata <https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/>

<https://stackoverflow.com/questions/35272592/how-are-finite-automata-implemented-in-code>