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

**Course: Formal Languages & Finite Automata**

**Author: Obrijan Filip**

**Theory:**

  A finite automaton is a mechanism used to represent processes of different kinds. It can be compared to a state machine as they both have similar structures and purpose as well. The word finite signifies the fact that an automaton comes with a starting and a set of final states. In other words, for process modeled by an automaton has a beginning and an ending.

    Based on the structure of an automaton, there are cases in which with one transition multiple states can be reached which causes non determinism to appear. In general, when talking about systems theory the word determinism characterizes how predictable a system is. If there are random variables involved, the system becomes stochastic or non deterministic.

    That being said, the automata can be classified as non-/deterministic, and there is in fact a possibility to reach determinism by following algorithms which modify the structure of the automaton.

**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**.

Variant 22

Q = {q0,q1,q2},

∑ = {a,b},

F = {q2},

δ(q0,a) = q0,

δ(q1,b) = q1,

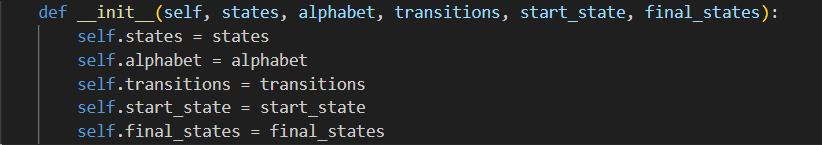
δ(q1,b) = q2,

δ(q0,b) = q1,

δ(q1,a) = q0,

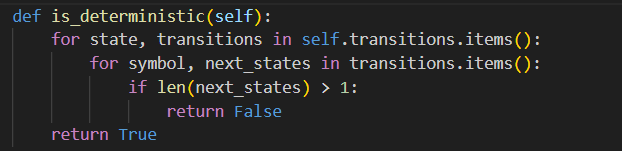
δ(q2,b) = q1.

**Implementation description:**

****

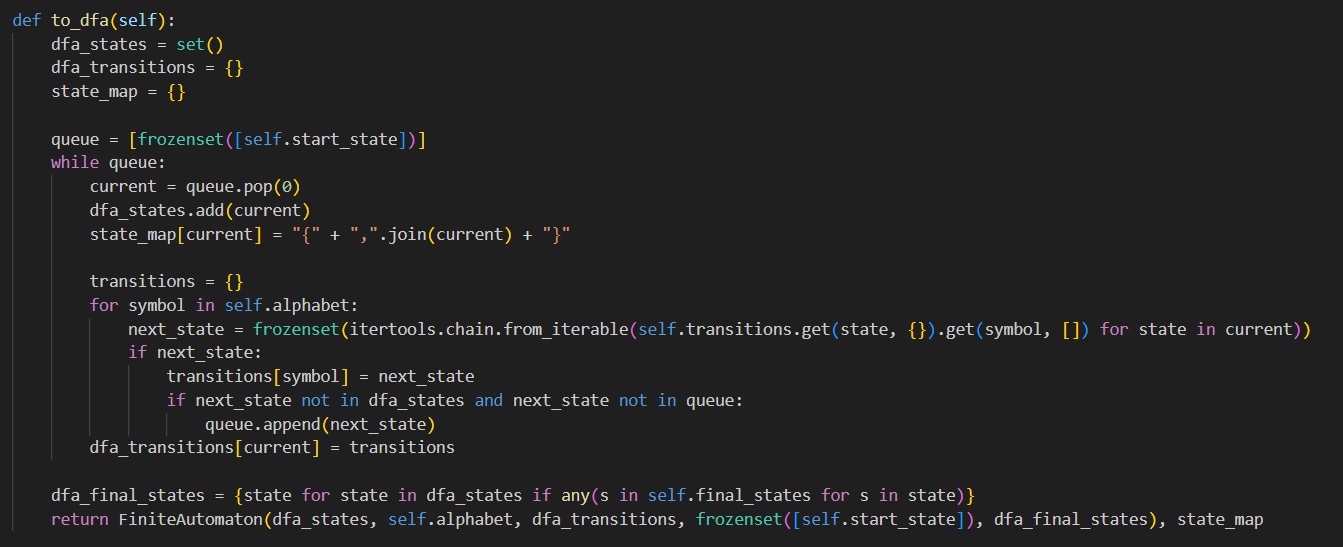
**What It Does:**

* This is the **constructor method** for the FiniteAutomaton class.
* It initializes the automaton with:
  + **states** → The set of all states in the automaton.
  + **alphabet** → The set of input symbols (e.g., {a, b}).
  + **transitions** → A dictionary defining state transitions.
  + **start\_state** → The initial state.
  + **final\_states** → The set of final (accepting) states.
* Stores these as instance attributes to be used in other methods.



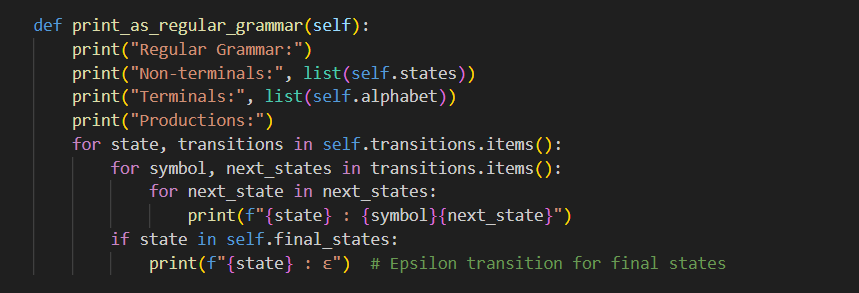
**What It Does:**

* Checks whether the automaton is **deterministic** (DFA) or **non-deterministic** (NFA).
* It iterates through the transitions dictionary:
  + If any state has **more than one possible next state for the same input symbol**, it means the automaton is an **NFA**, and the function returns False.
  + If every symbol leads to at most **one** state, it returns True (i.e., it is a DFA).
* **Example:**
  + **NFA:** { "q1": { "b": {"q2", "q3"} } } (two possible next states → non-deterministic)
  + **DFA:** { "q1": { "b": {"q2"} } } (only one next state → deterministic)



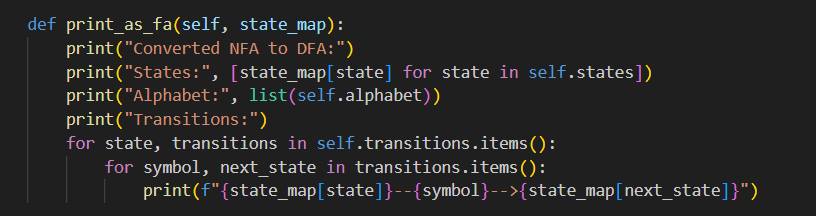
**What It Does:**

* Converts a **Non-deterministic Finite Automaton (NFA)** to a **Deterministic Finite Automaton (DFA)**.
* Uses **subset construction (powerset construction)** to handle multiple possible transitions.
* **Steps:**
  1. **Initialize**:
     + dfa\_states: A set to track new DFA states.
     + dfa\_transitions: A dictionary to store the DFA transition table.
     + state\_map: A dictionary mapping DFA states to readable names.
     + queue: A list to process new DFA states (starting with {start\_state}).
  2. **Process each DFA state:**
     + Extracts all possible transitions for each symbol.
     + Groups multiple NFA states into a single DFA state.
     + Adds new states to the queue if they haven't been seen before.
  3. **Identify final states:** Any DFA state that contains at least one NFA final state is marked as a **DFA final state**.
  4. **Returns:**
     + A new FiniteAutomaton object representing the DFA.
     + A state\_map for translating DFA states into readable names.



**What It Does:**

* Converts the automaton into an **equivalent regular grammar**.
* **Outputs:**
  + **Non-terminals** → The states.
  + **Terminals** → The alphabet symbols.
  + **Productions** (rules):
    - {state} : {symbol}{next\_state} for each transition.
    - If a state is final, it gets an **epsilon transition (ε)**, allowing it to end a string.



**What It Does:**

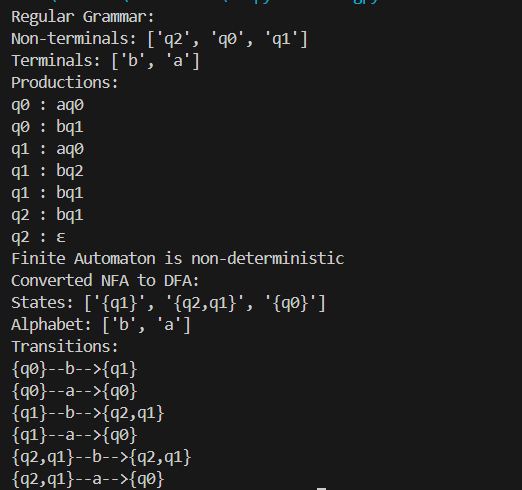
* Prints the **converted DFA** with readable state names.
* **Outputs:**
  + **States** in the DFA.
  + **Alphabet** used.
  + **Transitions** in a readable format:

{q0}--a-->{q0}

{q0}--b-->{q1}

* Uses state\_map to convert **frozensets (DFA states)** into human-readable names.
* Helps verify whether the **NFA-to-DFA conversion was correct**.

**Output example:**

****

**Conclusion:**

In this laboratory work, we analyzed and implemented a **Finite Automaton** class that can handle both **Non-deterministic Finite Automata (NFA)** and **Deterministic Finite Automata (DFA)**. This implementation provided a practical approach to understanding the fundamental principles of automata theory, including **state transitions, determinism, and the conversion between NFA and DFA**.

We began by defining an **NFA** based on the given transition rules, ensuring that it correctly followed the structure of finite automata. We then verified whether the automaton was **deterministic** using the is\_deterministic function. This function systematically examined the transitions for each state and input symbol to determine if multiple transitions existed for the same input. Since the automaton was **non-deterministic**, we proceeded with its conversion into a **DFA** using the to\_dfa function. This conversion utilized the **subset construction algorithm**, which systematically groups sets of NFA states into new DFA states, ensuring that the resulting DFA adheres to deterministic principles.

After obtaining the **DFA**, we displayed its equivalent **regular grammar representation**, which helps in understanding how the automaton can be expressed in terms of formal grammar rules. The transition table of the newly generated **DFA** was also printed to provide a clear view of how states transition from one to another when processing different input symbols. By mapping the original NFA states to corresponding DFA states, we ensured a structured and comprehensive conversion process.

This laboratory work successfully demonstrated the **core theoretical and practical aspects** of finite automata. Understanding **state transitions** is crucial in recognizing how an automaton processes input strings, while identifying **determinism** helps differentiate between NFAs and DFAs. The **NFA-to-DFA conversion** showcased an important technique used in compiler design, lexical analysis, and various applications in formal language theory. Additionally, the incorporation of **regular grammars** highlighted their role in defining **formal languages**, further bridging the connection between automata and language theory.

Overall, this implementation serves as a fundamental step toward mastering automata theory concepts, reinforcing the importance of **automated computation models** in theoretical computer science and practical applications such as **pattern recognition, text processing, and programming language design**.

**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>