**Determinism in Finite Automata.**

**Course: Formal Languages & Finite Automata**

**Author: Vladlen Mîțu**

**Theory:**

In the case of a **DFA**, there's one transition for every input from every state, thus indicating a single path. On the other hand, an **NFA** allows for multiple transitions and ε-transitions, and both will recognize the same languages. Because DFAs do not allow for these multiple transitions, the method used to convert an NFA to a DFA is known as subset construction

1. The sets of NFA states are treated as a single DFA state.

2. Transitions are defined with respect to all possible moves available from the NFA.

3. DFA final states are those which include an NFA final state.

**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.

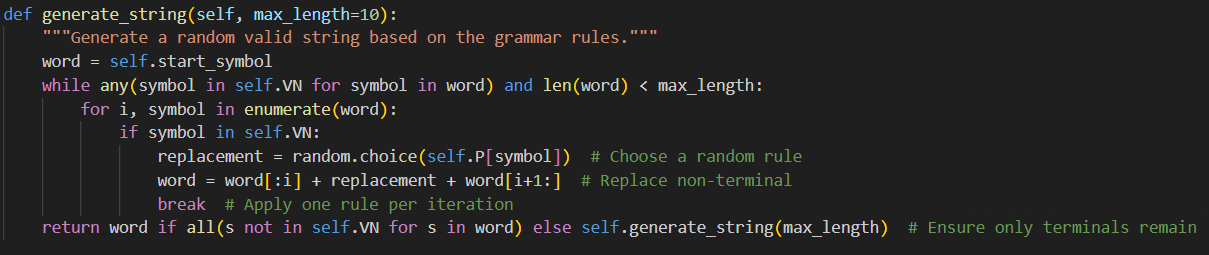
c. Implement some functionality that would convert an NDFA to a DFA.

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

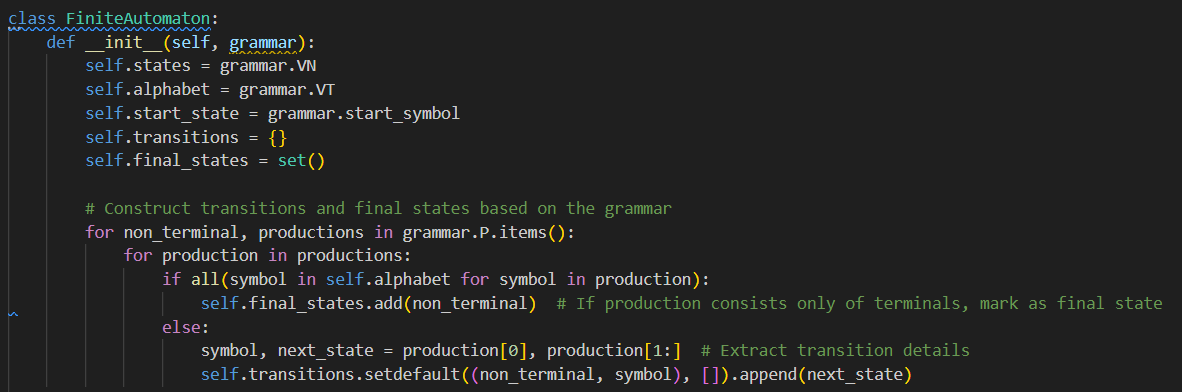
* + You can use external libraries, tools or APIs to generate the figures/diagrams.
  + Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

**Implementation description:**

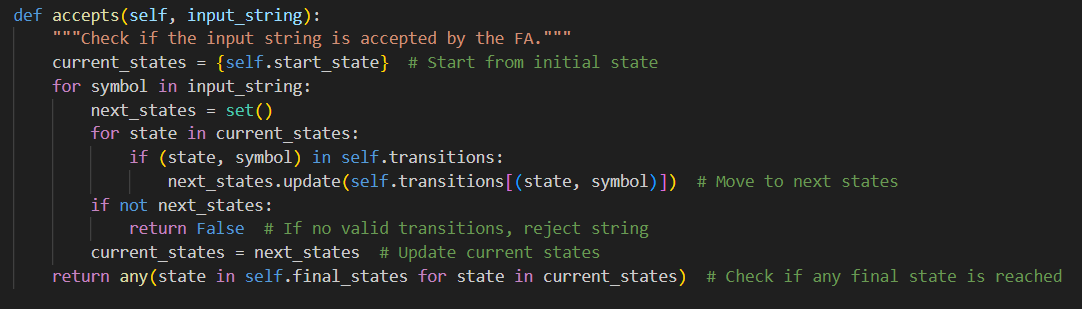
This function determines whether a given input string is **accepted** by the **Finite Automaton (FA)** by simulating state transitions. It starts from the **initial state** and processes each symbol, checking for valid transitions. If at any point no valid transition exists, the string is **rejected**. If the string is fully processed and the automaton ends in a **final state**, it is **accepted**. This approach ensures that only valid strings are recognized, making it useful for **lexical analysis, pattern matching, and formal language processing**.



This class constructs a **Finite Automaton (FA)** from a given grammar by defining its **states, alphabet, transitions, and final states** based on the provided production rules. It systematically converts the grammar into an automaton structure, ensuring that each **non-terminal** is properly mapped to its corresponding transitions. If a non-terminal produces only terminal symbols, it is marked as a **final state**, indicating that the automaton can successfully terminate in that state. For other cases, transitions are created to represent valid state changes according to the grammar's rules. This transformation allows the automaton to process input strings and determine whether they belong to the language defined by the grammar.



This function determines whether a given input string is **accepted** by the **Finite Automaton (FA)** by simulating state transitions. It starts from the **initial state** and processes each symbol, moving to the next state if a valid transition exists. If at any point no valid transition is found, the string is **rejected**. If the entire string is processed and the automaton ends in a **final state**, the string is **accepted**. This method ensures that only valid strings are recognized, making it useful in **compiler design, pattern matching, and formal language processing**.



**Conclusion:**

This code effectively constructs a **Finite Automaton (FA)** from a given grammar, defining states, transitions, and final states based on production rules. It generates valid strings by applying random derivations from the grammar until only terminal symbols remain. Additionally, it implements a method to check if a given input string is accepted by the automaton by simulating state transitions and ensuring it reaches a final state. By combining **formal grammar** and **automata theory**, the code demonstrates how a language can be processed and validated systematically.

**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>