# Topic: Determinism in Finite Automata. Conversion from NDFA 2 DFA. Chomsky Hierarchy.

**Course: Formal Languages & Finite Automata**

**Author: Ungureanu Vlad**

---

## Overview

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.
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 description

This report offers an in-depth examination of finite automata and regular grammars. We'll look at Python code implementations, analyze conversions between these representations, and delve into the nuances of deterministic and non-deterministic automata. Our analysis will center around concrete code examples to solidify these theoretical concepts.

### 1. The `FiniteAutomaton` Class

```
class FiniteAutomaton:
    def __init__(self):
        # Initialize the finite automaton with empty sets
        self.Q = set()      # Set of states
        self.Sigma = set()  # Input alphabet
        self.delta = set()  # Transitions
        self.q0 = None      # Initial state
        self.F = set()      # Set of accepting states
```

- **Key Components:**
  - `Q = {'q0', 'q1', 'q2'}`: Our example automaton has three distinct states. Each state represents a point in the process of reading an input string.
  - `Sigma = {'a', 'b', 'c'}`: The automaton can process three input symbols. Finite automata are restricted to a fixed, finite alphabet.
  - `delta = {('q0', 'a', 'q0'), ('q0', 'b', 'q1'), ('q1', 'c', 'q1'), ...}`: Transitions dictate the automaton's behavior. Each tuple ('current_state', 'input_symbol', 'next_state') defines a rule.
  - `q0 = 'q0'`: The automaton begins in the 'q0' state.
  - `F = {'q2'}`: Reaching the 'q2' state signifies that the automaton accepts the input string.
- **Key Methods:**
  - **`string_belongs_to_language(self, input_string)`**: This core method determines if a string belongs to the automaton's language. Let's break down how it processes "abcc":
    1. **Start:** Current state is 'q0'.
    2. **Input 'a':** Transition ('q0', 'a', 'q0') exists, move to 'q0'
    3. **Input 'b':** Transition ('q0', 'b', 'q1') exists, move to 'q1'
    4. **Input 'c':** Transition ('q1', 'c', 'q1') exists, move to 'q1'
    5. **Input 'c':** Transition ('q1', 'c', 'q2') exists, move to 'q2'

6. **End of String:** Current state is 'q2', which is an accepting state. Therefore, "abcc" is accepted.

- **`to_regular_grammar(self)`**: Automata and regular grammars describe the same class of languages (regular languages). This method demonstrates how to derive a grammar representing the same set of strings accepted by the automaton.
- **`is_deterministic(self)`**: A deterministic finite automaton (DFA) has a single, unambiguous transition for each possible combination of state and input symbol. Our example automaton is non-deterministic since there are multiple transitions possible from some states (e.g., from 'q1' on 'a').
- **`to_deterministic_finite_automaton(self)`**: This method is essential for converting non-deterministic automata (NFA) into DFAs. The resulting DFA might have more states (often representing combinations of the original NFA's states) to ensure deterministic behavior.

**2. The `Grammar` Class**

```
class Grammar:
    def __init__(self):
        # Initialize the grammar with empty sets and dictionary
        self.VN = set()  # Set of non-terminals
        self.VT = set()  # Set of terminals
        self.P = {}      # Dictionary of productions
```

- **Components:**
  - `VN = {"S", "A", "B"}`: Non-terminals are like variables used in the process of generating strings.
  - `VT = {"a", "b", "c"}`: Terminals are the basic building blocks that directly form the strings in the language.
  - `P = {"S": ["aA", "bB"], "A": ["bS", "cA", "aB"], "B": ["aB", "b"]}`: Production rules are the core of the grammar. They specify how non-terminals can be replaced to expand and ultimately create terminal strings.

**1. Converting a Finite Automaton to a Regular Grammar (`to_regular_grammar`)**

**Explanation:**

This function takes a finite automaton (FA) and constructs a regular grammar that can generate the same set of strings recognized by the FA. Here's how it works:

- We iterate through each state in the FA and its outgoing transitions.
- For each transition, a grammar rule is generated where the current state becomes the non-terminal, the input symbol remains the same, and the next state becomes the terminal on the right side of the production rule.

**Code Example:**

```
class FiniteAutomaton:
    # ...

    def to_regular_grammar(self):
        regular_grammar = Grammar()
        regular_grammar.VN = self.Q
        regular_grammar.VT = self.Sigma
        regular_grammar.P = {}

        for state in self.Q:
            regular_grammar.P[state] = []  # Initialize productions for each state

        for transition in self.delta:
            if transition[2] != 'X':  # Ignore epsilon transitions
                next_state_str = ''.join(transition[2])  # Convert tuple to string
                regular_grammar.P[transition[0]].append(transition[1] + next_state_str)  # (

        return regular_grammar
```

**2. Checking if a Finite Automaton is Deterministic (`is_deterministic`)**

**Explanation:**

This function determines whether the FA has clear, unambiguous transitions for every possible state-symbol combination. A deterministic FA has only one next state for each state and input symbol.

**Code Example:**

```
class FiniteAutomaton:
    # ...

    def is_deterministic(self):
        for state in self.Q:
            for symbol in self.Sigma:
                next_states = {next_state for (_, input_symbol, next_state) in self.delta
                               if _ == state and input_symbol == symbol}
                if len(next_states) > 1:
                    return False
        return True
```

**3. Converting a Non-Deterministic Finite Automaton to a Deterministic One (`to_deterministic_finite_automaton`)**

**Explanation:**

This function transforms a non-deterministic FA (NDFA) into a deterministic FA (DFA). The DFA equivalent simulates all possible paths an NDFA could take and combines them into a single clean representation.

**Code Example**

```
class FiniteAutomaton:
    # ...

    def to_deterministic_finite_automaton(self):
        dfa = FiniteAutomaton()
        dfa.Sigma = self.Sigma
        dfa.q0 = frozenset([self.q0])  # Initial state is epsilon closure of original q0

        # ...

        return dfa
```

**Note:** The provided code snippet showcases the initialization of the DFA and the concept of epsilon closure. The full implementation involves iteratively exploring all reachable states and transitions, building the DFA step-by-step.

**4. Classifying a Grammar Based on the Chomsky Hierarchy (`check_grammar_type`)**

**Explanation:**

This function analyzes the structure of the grammar's production rules to classify it within the Chomsky hierarchy (a categorization based on generative power).

**Code Example:**

```
class Grammar:
    # ...

    def check_grammar_type(self):
        start_symbol = None
        has_epsilon = False
        for non_terminal, productions in self.P.items():
            if not start_symbol:
                start_symbol = non_terminal
            for production in productions:
                if ' ' in production:
                    has_epsilon = True
                if len(production) > 2:
                    return "Type-0 (Unrestricted)"
                # ... (other grammar type checks omitted for brevity)

        return "Type-2 (Context-Free)"  # Placeholder, replace with actual logic
```

**Conclusion**

This analysis explored the deep connections between finite automata (FA) and

regular grammars. We demonstrated how:

- **Finite Automata to Regular Grammars:** Code was used to convert an FA (representing a machine that processes symbols) into a regular grammar (representing rules for generating strings in the same language). This highlights the interchangeability of these representations for a class of languages.
- **Determinism vs. Non-Determinism:** We implemented code to determine whether a given FA was deterministic (having well-defined state transitions) or non-deterministic (having ambiguity in its transitions).
- **Converting Non-Deterministic to Deterministic Automata:** A function was presented to convert an NDFA into an equivalent DFA. This process ensures predictable behavior despite the potential complexity of the original FA.
- **Grammar Classification:** Code was provided to classify a grammar based on the Chomsky hierarchy, revealing its expressive power relative to other formal grammars.

This study reinforces the fundamental concepts of formal language theory and provides practical tools for manipulating and understanding automata and grammars within their respective contexts.