**MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

**Laboratory work 2:**

**Regular Grammars & Finite Automata**

**Course: Formal Languages & Finite Automata**

**Author:Ostafi Eugen, FAF-222**

**Chișinău, 2024**

**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.

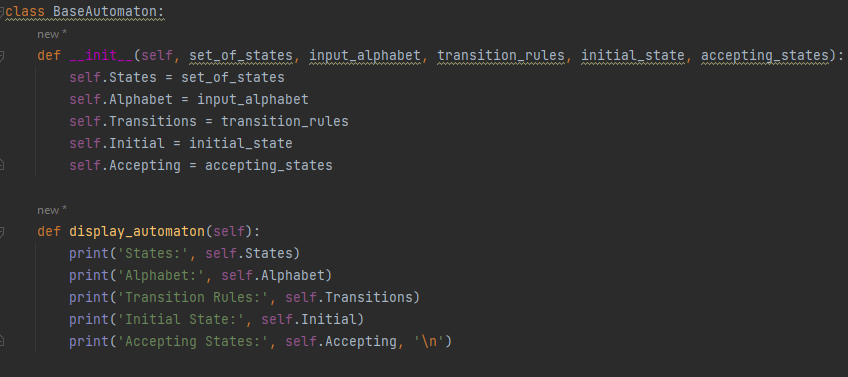
d. Represent the finite automaton graphically (Optional, and can be considered as a **bonus point**):

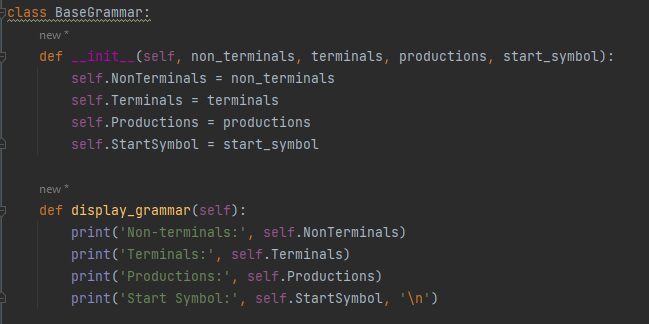
* + You can use external libraries, tools or APIs to generate the figures/diagrams.
  + Your program needs to gather and send the data about the automaton and the lib/tool/API return the visual representation.

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**

1.Classes:





2. Chomsky Hierarchy



The function in the image, classify, is designed to classify a given grammar based on the Chomsky Hierarchy, which is a way to categorize grammars into different types according to their generative power. The Chomsky Hierarchy includes four levels:

Type 3 (Regular Grammar): This is the simplest type of grammar. It can be represented by regular expressions and is used to define regular languages. These grammars allow rules of a specific form, typically one non-terminal followed by one terminal symbol on the right-hand side.

Type 2 (Context-Free Grammar): These grammars are used to define context-free languages, which include most programming languages and the syntax of HTML and XML. A context-free grammar allows rules with a single non-terminal on the left-hand side and a string of terminals and/or non-terminals on the right-hand side.

Type 1 (Context-Sensitive Grammar): These grammars generate context-sensitive languages. They allow rules where the length of the string on the right-hand side is greater than or equal to the length of the string on the left-hand side.

Type 0 (Unrestricted Grammar): This is the most general type of grammar, which allows any production rules. Unrestricted grammars can define all recursively enumerable languages.

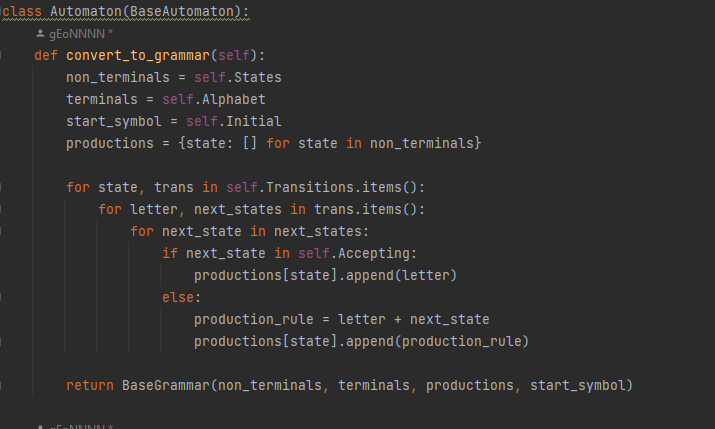
The classify function operates by iterating through the production rules of a grammar and setting flags (is\_regular, is\_context\_free, and is\_context\_sensitive) based on specific criteria that match the definitions of the different types of grammars in the Chomsky Hierarchy:

It first assumes the grammar is of the highest constraint (Type 3, Regular Grammar), and as it inspects the rules, it downgrades this classification if specific conditions are violated (e.g., if a production doesn't match the format of a regular grammar, it then considers if it could be a context-free grammar, and so forth).

The classification is downgraded based on the nature of the production rules: whether they have the right formats, the lengths of the right-hand sides and left-hand sides of the productions, and the use of terminal and non-terminal symbols.

The final output of the function is the type of grammar the given set of rules represents, according to the strictest classification that applies based on the grammar's production rules.

3.To grammar convert



The function begins by setting the non\_terminals of the new grammar to be the states of the automaton, effectively treating each state as a distinct non-terminal symbol in the grammar.

It sets the terminals of the new grammar to be the alphabet of the automaton, which consists of all the input symbols that the automaton recognizes.

The start\_symbol of the new grammar is set to the initial state of the automaton, marking where computations or derivations in the grammar should begin.

An empty set of productions is created for each state in the automaton, setting the foundation for the conversion from state transitions to grammatical productions.

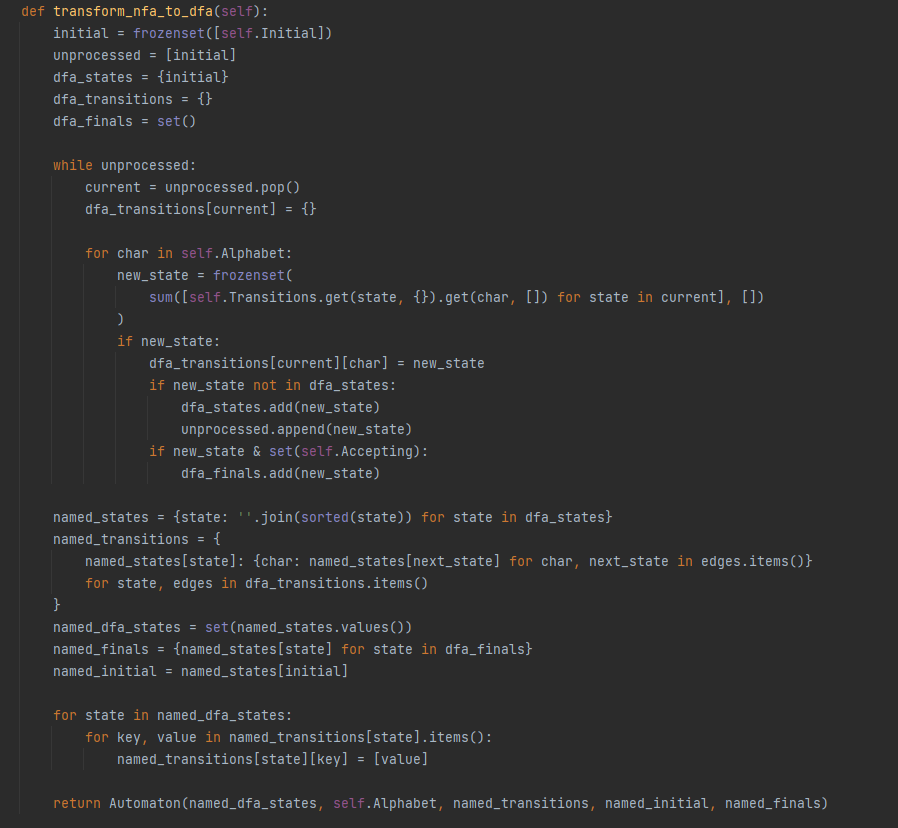
The function iterates through each state and its transitions in the automaton, examining the letter (input symbol) and the next states for each transition.

For each next\_state, if it is an accepting state in the automaton, a production rule is added with the current state as the left-hand side and the letter as the right-hand side. This reflects a transition that leads to an accepting state.

If the next\_state is not accepting, a production rule is formed with the current state as the left-hand side and the letter followed by the next\_state as the right-hand side. This incorporates both the transition symbol and the subsequent state into the grammar.

Finally, the function compiles the non\_terminals, terminals, productions, and start\_symbol into a new BaseGrammar object and returns it. This new grammar represents the same language that was recognized by the original automaton, transformed into the formalism of grammatical productions.

4. NFA to DFA



The function starts by initializing the initial state of the DFA as the closure of the initial state of the NFA. It also initializes a list of unprocessed states, starting with the initial state, and creates empty dictionaries for DFA states and transitions, as well as an empty set for the final states of the DFA.

The main loop continues as long as there are unprocessed states. It takes one unprocessed state and looks at each character in the alphabet. For each character, it calculates the new state, which is the set of states that can be reached from the current state under that character, according to the NFA's transition function.

If the new state (a combination of states from the NFA) has not been seen before, it is added to the list of DFA states and marked for processing. If this new state includes any of the NFA's accepting states, it is added to the set of DFA's accepting states.

After processing all characters from the alphabet for the current state, the function continues with the next unprocessed state until there are no more.

The function then renames the states from sets of NFA states to more readable names (usually combining the names of the NFA states that make up each DFA state) and updates the transition table accordingly.

Finally, it creates and returns a new Automaton object representing the DFA, using the renamed states, the original alphabet, the new transition table, the new initial state, and the new set of accepting states.

5.Graph



The function starts by creating an empty directed graph. It then iterates through all the states of the automaton, adding each as a node to the graph.

Next, it processes the transitions of the automaton. For each state and for each transition from that state (triggered by an input character), it adds a directed edge from the "from" state to the "to" state, labeling the edge with the triggering input character.

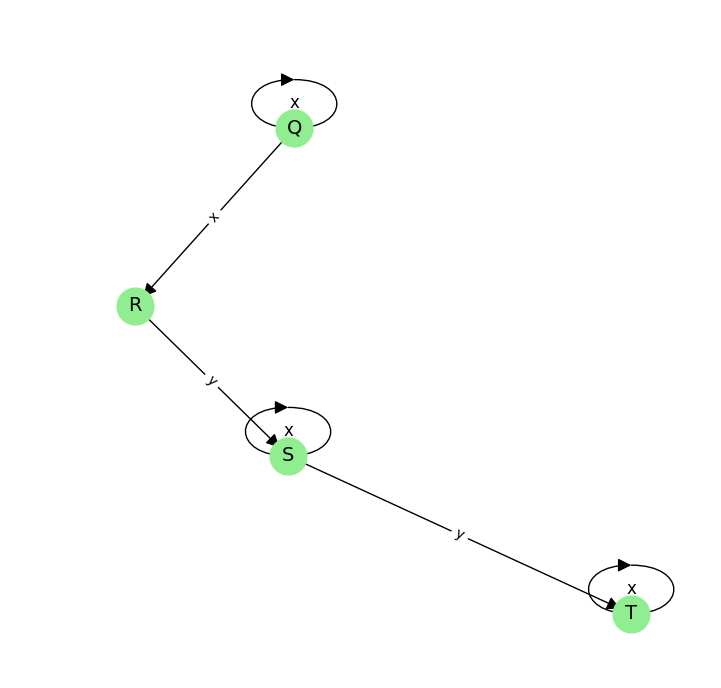
The layout for the graph is determined using a spring layout algorithm, which positions nodes in a way that visually separates them evenly in space, making the graph easier to read.

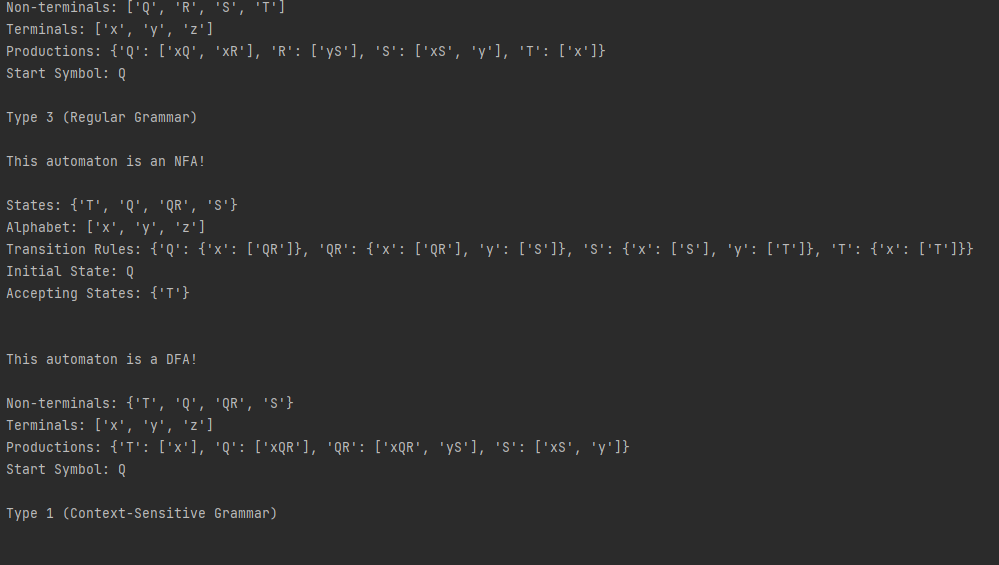
The function then creates a figure for displaying the graph and draws the nodes and edges using specific styling choices, such as node size, node color, arrow style, and arrow size.

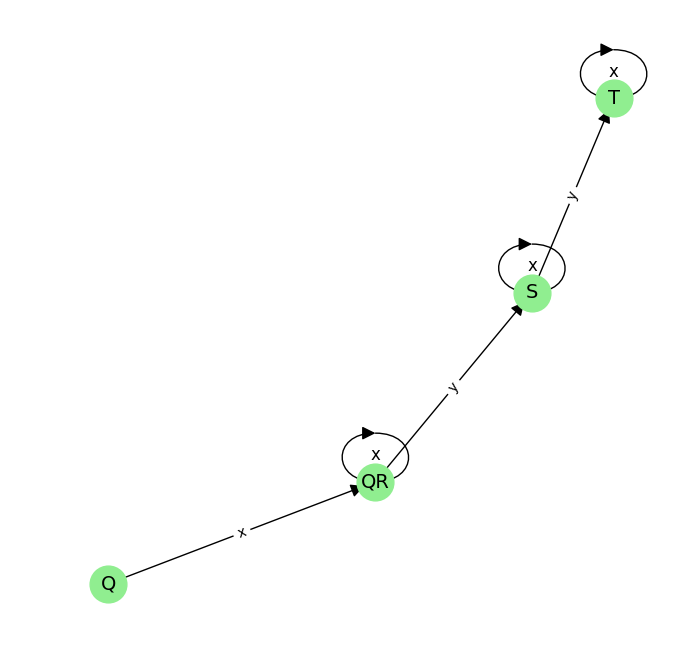
It also draws the labels for the nodes and edges using specified font sizes to ensure they are legible.

For edges that loop from a node back to itself (self-transitions), the function positions the label slightly above the node to avoid visual clutter and to clearly denote the loop.

Finally, the function turns off the axis (so that coordinate markers are not shown) and displays the figure, showing the visual representation of the automaton.

Output:  






**Conclusion**

This are the key components in computational theory and automata studies. They include converting a nondeterministic finite automaton (NFA) to a deterministic finite automaton (DFA), classifying grammars based on the Chomsky Hierarchy, and visualizing finite automatons as directed graphs. These processes underpin the understanding of language recognition, parsing, and automata theory, showcasing the transformation of complex abstract machines into more manageable forms and the classification of languages based on grammar types. Additionally, they facilitate the visual representation of automatons, enhancing comprehension and analysis of computational models and their transitions.