Ministry of Education, Culture, and Research of the Republic of Moldova
Technical University of Moldova
The Faculty of Computers, Informatics, and Microelectronics

REPORT

Laboratory work no.2
Language and Automata Theory

Executed by: Cornievschi Bogdan
st. gr. FAF-222

Verified by:
univ. assist. Crețu Dumitru

Chişinău - 2024

Objectives

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.

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.

Variant

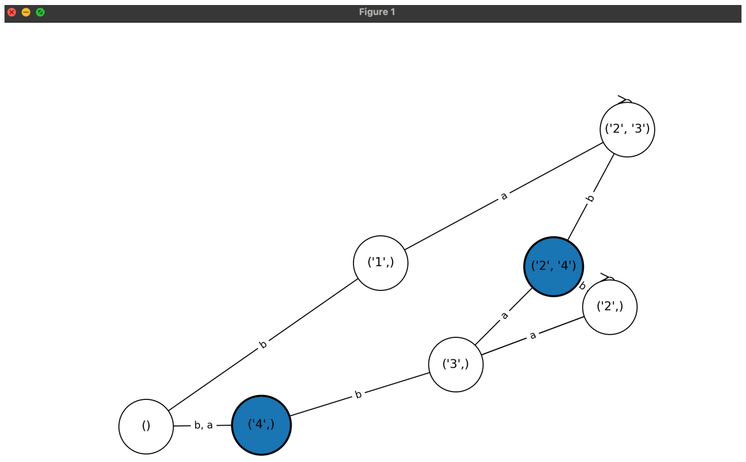
Variant 4
$Q = \{q0,q1,q2,q3\}$,
$\Sigma = \{a,b\}$,
$F = \{q3\}$,
$\delta(q0,a) = q1$,
$\delta(q0,a) = q2$,
$\delta(q1,b) = q1$,
$\delta(q1,a) = q2$,
$\delta(q2,a) = q1$,
$\delta(q2,b) = q3$.

Solution

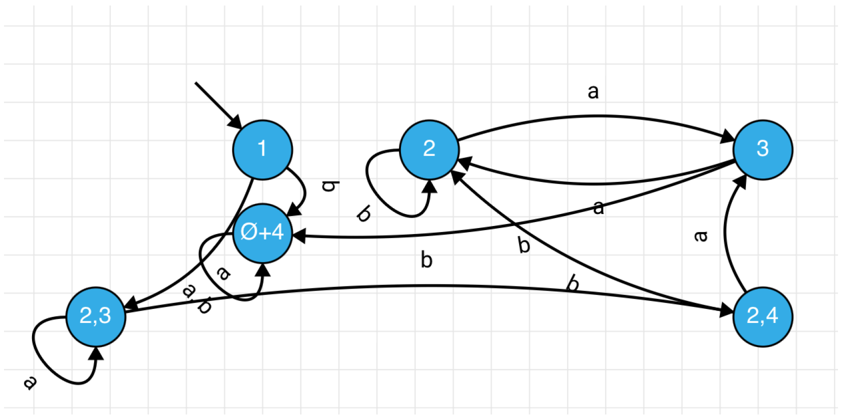
```python
import networkx as nx

import matplotlib.pyplot as plt


class Automaton:

    def __init__(self, states, alphabet, final_states, transitions):

        self.states = states

        self.alphabet = alphabet

        self.final_states = final_states

        self.transitions = transitions


    def is_dfa(self):

        for state in self.states:

            for symbol in self.alphabet:

                transitions = self.transitions.get((state, symbol), None)

                if transitions is None or len(transitions) != 1:

                    return False

        return True


    def convert_to_dfa(self):

        if self.is_dfa():

            return self  # It's already a DFA, no conversion needed.


        new_states = []

        new_transitions = {}

        new_final_states = []


        initial_state = ('1',)  # Assuming 'q0' is the initial state
        new_states.append(initial_state)
        queue = [initial_state]

        while queue:
            current = queue.pop(0)
            for symbol in self.alphabet:
                next_state = sum([self.transitions.get((state, symbol), []) for
state in current], [])
                next_state = tuple(sorted(set(next_state)))  # Removing
duplicates & sorting for consistency
                if next_state not in new_states and next_state:
                    new_states.append(next_state)
                    queue.append(next_state)

                new_transitions[(current, symbol)] = next_state

                if any(state in self.final_states for state in next_state):
                    new_final_states.append(next_state)

        return Automaton(new_states, self.alphabet, new_final_states,
new_transitions)

    def draw(self):
        # Create a new directed graph
        G = nx.DiGraph()

        # Add nodes with labels
        for state in self.states:
            G.add_node(state, label=str(state))
        # Prepare a dictionary to keep track of transitions for combining labels
        edge_labels = {}

        # Add edges with labels
        for (src, symbol), dst in self.transitions.items():
            # If the transition is already in the dictionary, append the symbol
to the label
            if (src, dst) in edge_labels:
                edge_labels[(src, dst)] += ', ' + symbol
            else:
                edge_labels[(src, dst)] = symbol
                G.add_edge(src, dst)

        # Graph layout
        pos = nx.spring_layout(G)
        plt.figure(figsize=(12, 8))

        # Draw nodes and edges
        nx.draw_networkx_nodes(G, pos, node_size=3000, node_color='white',
edgecolors='black')
        nx.draw_networkx_edges(G, pos, arrowstyle='->', arrowsize=20)

        # Draw node and edge labels
        nx.draw_networkx_labels(G, pos, labels={node: node for node in
G.nodes()})
        nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

        # Highlight the initial state with an incoming arrow
        initial_state = '1'  # Replace with the actual initial state
        if initial_state in pos:
            initial_pos = (pos[initial_state][0] - 0.1, pos[initial_state][1])
            G.add_node('start', pos=initial_pos, label='')
            G.add_edge('start', initial_state, label='')

        # Double-circle the final states
        for final_state in self.final_states:
            nx.draw_networkx_nodes(G, pos, nodelist=[final_state],
node_shape='o', node_size=3500, edgecolors='black',
                                   linewidths=2)

        plt.axis('off')
        plt.show()

    def __repr__(self):
        return f'States: {self.states}\nAlphabet: {self.alphabet}\nFinal States:
{self.final_states}\nTransitions: {self.transitions}'


# Example usage:
states = {'1', '2', '3', '4'}
alphabet = {'a', 'b'}
final_states = {'4'}
transitions = {
    ('1', 'a'): ['2', '3'],
    ('2', 'b'): ['2'],
    ('2', 'a'): ['3'],
    ('3', 'a'): ['2'],
    ('3', 'b'): ['4'],
}
```

```
automaton = Automaton(states, alphabet, final_states, transitions)

print("The automaton is a DFA." if automaton.is_dfa() else "The automaton is an
NFA.")


if not automaton.is_dfa():

    dfa = automaton.convert_to_dfa()

    print("Converted to DFA:")

    print(dfa)

    dfa.draw()
```

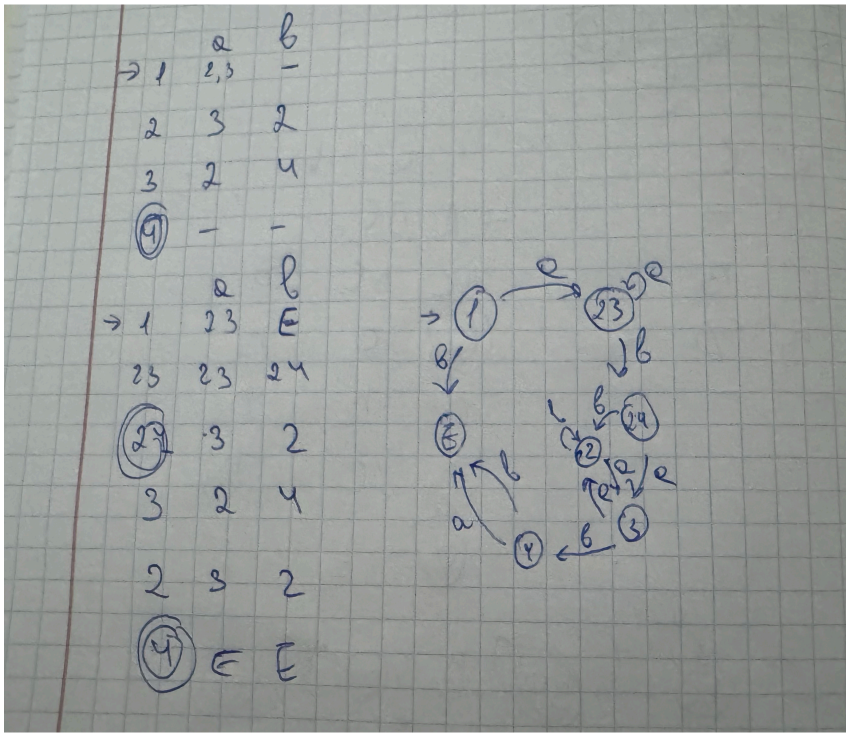
The result



1. The result obtained after executing the code for my variant



2. The DFA obtained after constructing NFA from my variant with the project joeylemon.github.io



	a	b
→1	2,3	–
2	3	2
3	2	4
4	–	–

	a	b
→1	23	E
23	23	24
24	3	2
3	2	4
2	3	2
4	E	E

3. The DFA from NFA obtained by the manual subset construction method

As we can see, my result is nearly the same as obtained by the project and by implementing subset construction method, the drawbacks of my graph is that it doesn't show the direction of the edges, and the input that leaves you in the same state is not shown too.

Conclusion

The laboratory work was a comprehensive exercise in applying theoretical concepts from automata theory to practical problems. It reinforced the importance of understanding the foundational models of computation, such as finite automata, and their applications in computer science. By implementing conversions between different forms of automata and translating automata into regular grammars, we not only deepened our understanding of these models but also acquired practical skills that are applicable in various computational tasks. The optional visualization component further enriched this learning experience by offering a visual perspective on abstract concepts, making them more accessible and understandable.

Overall, this laboratory work underscores the intersection of theory and practice in computer science education, preparing students with the knowledge and skills needed to tackle complex problems in the field. Through tasks that ranged from theoretical analysis to practical implementation and visualization, we have gained a more holistic understanding of automata theory and its relevance to both computational and linguistic applications.