Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Inginerie software și automatică

Laboratory work 2:
Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-232 Istrati Ștefan

Verified:
asist. univ. Crețu Dumitru

Chişinău - 2025

Theory:

Finite automata (FA) are abstract mathematical models used to represent and analyze the behavior of systems with discrete states. They are widely utilized in computer science, particularly in the fields of formal language theory, automata theory, and compiler design. Finite automata are essential for the study of regular languages, which can be described by regular expressions and recognized by finite automata. These models form the basis for various practical applications, such as lexical analysis, text search, and pattern recognition.

A finite automaton consists of a finite set of states, a starting state, a set of accepting states, and a set of transitions between these states. The system moves from one state to another based on input symbols, and the transitions are typically represented as a directed graph. Each state represents a possible configuration of the system, while transitions define how the system responds to input symbols.

There are two main types of finite automata: deterministic finite automata (DFA) and nondeterministic finite automata (NDFA). The key difference between the two lies in the transition rules. In a DFA, for each state and input symbol, there is exactly one transition to another state, making the system deterministic. This means that for any given input string, the DFA can only follow one unique path through the states. On the other hand, an NDFA allows multiple transitions for a given state and input symbol, including the possibility of transitioning to more than one state simultaneously. This non-determinism introduces a level of flexibility in the NDFA's operation.

Despite their apparent differences, DFAs and NDFAs are equivalent in terms of the languages they can recognize. Every NDFA can be converted into an equivalent DFA through a process known as determinization. However, the resulting DFA may have exponentially more states than the original NDFA, which can lead to inefficiencies in terms of space and time complexity.

Finite automata are often used to recognize regular languages, which are the simplest class of formal languages in the Chomsky hierarchy. A regular language can be defined by a regular expression, and it can be recognized by a DFA or an NDFA. The relationship between regular expressions and finite automata is a fundamental result in formal language theory, as it provides a means of transforming a regular expression into an automaton and vice versa.

The conversion between finite automata and regular grammars is another important concept. A regular grammar is a formal grammar that generates a regular language, and finite automata can be used to recognize the strings generated by such grammars. The conversion between a finite automaton and a regular grammar involves creating production rules based on the transitions and states of the automaton.

Finite automata are also used in the context of language recognition, where an automaton processes an input string and determines whether it belongs to a particular language. The automaton begins in the starting state and processes each symbol of the input string in sequence. If the automaton reaches an accepting state at the end of the input, the string is accepted; otherwise, it is rejected. This process is crucial in various applications, including the design of compilers and interpreters, where finite automata help in scanning and tokenizing the source code.

In summary, finite automata are powerful and versatile tools for modeling and analyzing systems with discrete states. They provide a formal foundation for understanding regular languages and form the basis for many applications in computer science, especially in areas such as language processing, text analysis, and machine learning.

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:

The FiniteAutomaton class represents a finite automaton, supporting operations such as checking determinism, converting to a DFA, converting to a regular grammar, classifying grammar types, and drawing the automaton as a graph. It is initialized with a set of states, an alphabet, a set of transitions, a start state, and a set of final states.

__init__(self, states, alphabet, transitions, start_state, final_states)

The constructor initializes the finite automaton by accepting five parameters:

- states: a set of states in the automaton.
- alphabet: a set of symbols that the automaton reads.
- transitions: a dictionary mapping each state to a dictionary of symbols, where each symbol points to a list of states.
- start_state: the initial state of the automaton.
- final_states: a set of states that are considered accepting states.

is_deterministic(self)

This function checks if the automaton is deterministic by verifying if there is at most one transition for

each symbol from every state. It returns True if the automaton is deterministic and False otherwise. This function ensures that no state has multiple transitions for the same input symbol, which is the hallmark of non-deterministic finite automata (NDFA).

to_dfa(self)

This function converts the current automaton to a deterministic finite automaton (DFA). If the automaton is already deterministic, it simply returns the current automaton. Otherwise, it performs a subset construction (powerset construction) to generate the DFA. It creates new states corresponding to sets of NDFA states and defines new transitions based on the original automaton's transitions. The resulting DFA is returned as a new FiniteAutomaton object.

to_regular_grammar(self)

This function converts the finite automaton into an equivalent regular grammar. It constructs a grammar by iterating over the transitions of the automaton and creating production rules for each state and symbol combination. If a state is a final state, it adds a production rule for the empty string (ε). The resulting regular grammar is returned as a dictionary where keys are states and values are lists of production rules.

classify_grammar(self, grammar)

This function classifies the given grammar into one of the four types in the Chomsky hierarchy:

- Type-3 (Regular Grammar)
- Type-2 (Context-Free Grammar)
- Type-1 (Context-Sensitive Grammar)
- Type-0 (Unrestricted Grammar)

It checks the structure of the production rules in the grammar, specifically the left-hand side (lhs) and right-hand side (rhs) of each production. Based on the conditions, the function returns a string representing the grammar's type.

draw(self)

This function visualizes the finite automaton as a directed graph. It uses networkx to create a directed graph where states are nodes and transitions are edges. The edges are labeled with the input symbols. The graph layout is computed using a spring layout algorithm, and the automaton is displayed using matplotlib. This function helps visually inspect the structure of the finite automaton.

Main Execution

V- 16

Q = {q0,q1,q2,q3},

∑ = {a,b},

F = {q3},

δ(q0,a) = q1,

δ(q1,b) = q1,

δ(q1,b) = q2,

δ(q2,a) = q2,

δ(q2,b) = q3,

δ(q0,b) = q0. 1.

Initialize the finite Automaton

```
states = {"q0", "q1", "q2", "q3"}
alphabet = {"a", "b"}
transitions = {
    "q0": {"a": ["q1"], "b": ["q0"]},
    "q1": {"b": ["q1", "q2"]},
    "q2": {"a": ["q2"], "b": ["q3"]},
}
start_state = "q0"
final_states = {"q3"}

fa = FiniteAutomaton(states, alphabet, transitions, start_state, final_states)
```

Check if the automaton is deterministic (is_deterministic function)

For the automaton to be deterministic, each state should have at most one transition for each symbol. Let's go through the transitions:

- From q0q0q0, we have:
  - δ(q0,a)=q1
  - δ(q0,b)=q0 (No conflicts here, each symbol maps to one state)
- From q1q1q1, we have:
  - δ(q1,b)=q1 and δ(q1,b)=q2. This is problematic because for symbol b, there are two possible transitions: q1 and q2. Therefore, the automaton is **non-deterministic**.

```
print("Deterministic?", fa.is_deterministic())
```

```
def is_deterministic(self):
    for state, trans in self.transitions.items():
        for symbol, targets in trans.items():
            if len(targets) > 1:
                return False
    return True
```

Bresults:

```
Deterministic? False
```

2. Convert the NDFA to a DFA (to_dfa function)

Since the automaton is non-deterministic, we will use the subset construction method to convert it to a deterministic finite automaton (DFA).

Steps for the subset construction:

- **Initial State**: Start with {q0}, which is the starting state in the NDFA.

- **Transitions**:
    - For symbol aaa, starting from q0, we go to q1 (from δ(q0,a)=q1).
    - For symbol b, starting from q0, we stay in q0(from δ(q0,b)=q0).

So, we create a new state corresponding to {q0 } (let's call it state q0).

- Next, consider q1, which has transitions:
    - For symbol b, we go to q1 and q2 (from δ(q1,b)=q1 and δ(q1,b)=q2\delta(q1, b) = q2δ(q1,b)=q2).

So, we create a new state corresponding to {q1,q2 }.

- Continue this process for the remaining states until all reachable states are explored.

Resulting DFA (simplified):

- **States**: {q0,{q1,q2},q3}
- **Alphabet**: {a,b}
- **Transitions**:
    - From q0, for symbol a, go to {q1,q2 }.
    - From q1,q2 for symbol b, go to q1,q2
    - From {q1,q2}, for symbol b, go to q3.
- **Start State**: q0
- **Final States**: {q3}

The program:

```
def to_dfa(self):
    if self.is_deterministic():
        return self

    new_states = {}
    queue = [frozenset([self.start_state])]
    new_transitions = {}
    new_final_states = set()

    while queue:
        current = queue.pop(0)
        state_name = ','.join(sorted(current))
        new_states[current] = state_name
        new_transitions[state_name] = {}

        for symbol in self.alphabet:
            next_states = set()
            for state in current:
                next_states.update(self.transitions.get(state, {}).get(symbol, []))

            if next_states:
                next_state_name = ','.join(sorted(next_states))
                new_transitions[state_name][symbol] = next_state_name
                if frozenset(next_states) not in new_states:
                    queue.append(frozenset(next_states))

            if any(s in self.final_states for s in next_states):
                new_final_states.add(next_state_name)

    return FiniteAutomaton(set(new_states.values()), self.alphabet, new_transitions, new_states[frozenset([self.start_state])], new_final_states)
```

Results:

```
DFA Transitions: {'q0': {'b': 'q0', 'a': 'q1'}, 'q1': {'b': 'q1,q2'}, 'q1,q2': {'b': 'q1,q2,q3', 'a': 'q2'}, 'q1,q2,q3': {'b': 'q1,q2,q3', 'a': 'q2'}, 'q2': {'b': 'q3', 'a': '
q2'}, 'q3': {}}
```

3. Convert the NDFA to Regular Grammar (to_regular_grammar function)

The regular grammar is constructed by taking each state and its transitions to produce production rules for the grammar.

Steps for constructing the grammar:

For state q0:

- δ(q0,a)=q1 → q0produces aq1.
- δ(q0,b)=q0 → q0produces bq0.

For state q1:

- δ(q1,b)=q1 → q1 produces bq1.
- δ(q1,b)=q2 → q1 produces bq2.

For state q2:

- δ(q2,a)=q2 → q2 produces aq2.
- δ(q2,b)=q3 → q2 produces bq3.

For state q3:

- q3 is a final state, so q3 produces ϵ.

The program:

```
def to_regular_grammar(self):
    grammar = defaultdict(list)
    for state, trans in self.transitions.items():
        for symbol, targets in trans.items():
            for target in targets:
                grammar[state].append(symbol + target)

    for final in self.final_states:
        grammar[final].append('ε')

    return grammar
```

Final result:

```
{'q0': ['aq1', 'bq0'], 'q1': ['bq1', 'bq2'], 'q2': ['aq2', 'bq3'], 'q3': ['ε']})
```

4. Classify the Grammar (classify_grammar function)

Now, let's classify the grammar using the Chomsky hierarchy:

- The grammar is regular because:
  - Each production is of the form A→aB or A→ϵA.
  - This fits the definition of **Type-3 (Regular Grammar)**.

Thus, the grammar is classified as **Type-3 (Regular Grammar)**.

The code:

```
def classify_grammar(self, grammar):
    is_regular = True
    is_context_free = True
    is_context_sensitive = True

    for lhs, rhs_list in grammar.items():
        for rhs in rhs_list:
            if rhs == 'ε':
                continue

            if len(lhs) > 1:
                is_regular = False
                is_context_free = False

            if not any(nonterm.isupper() for nonterm in lhs):
                is_context_free = False
                is_context_sensitive = False

    if is_regular:
        return "Type-3 (Regular Grammar)"
    elif is_context_free:
        return "Type-2 (Context-Free Grammar)"
    elif is_context_sensitive:
        return "Type-1 (Context-Sensitive Grammar)"
    else:
        return "Type-0 (Unrestricted Grammar)"
```

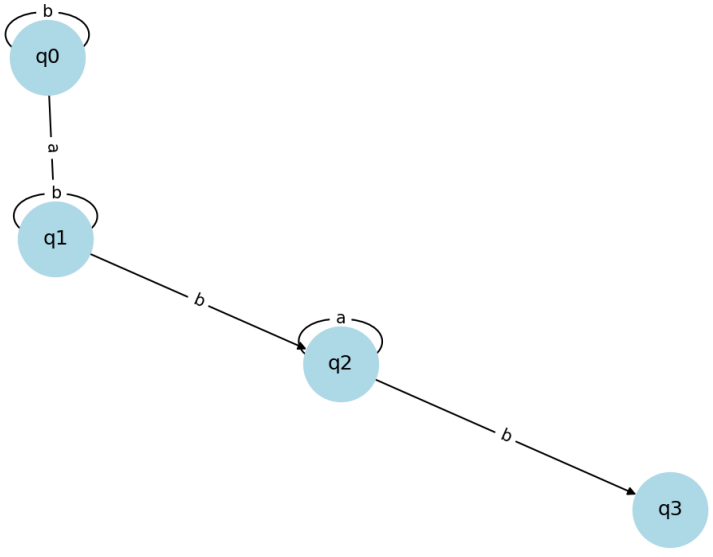
The result

```
Grammar Classification: Type-3 (Regular Grammar)
```

5. Draw the Automaton (draw function)

This function creates a graphical representation of the automaton:

- Nodes represent states q0,q1,q2,q3.
- Directed edges represent the transitions, labeled with the corresponding symbols a and b.
- The final state q3 is distinguished visually.



Conclusion:

In conclusion, the laboratory work on finite automata has provided a deeper understanding of how these abstract mathematical models can be used to represent and analyze systems with discrete states. By implementing a FiniteAutomaton class in Python, we were able to explore key concepts such as determinism, conversion from NDFA to DFA, and the generation of regular grammars. The implemented methods demonstrate how to determine if an automaton is deterministic, convert an NDFA into a DFA, and classify grammars according to the Chomsky hierarchy. Additionally, visualizing the automaton through graphical representation helped solidify the understanding of the structure and flow of state transitions. This hands-on approach not only reinforces theoretical knowledge but also offers practical insights into how finite automata play a vital role in language recognition and the design of computational systems, such as parsers and lexical analyzers.