Ministry of Education and Research of the Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics, and Microelectronics

Laboratory work 1: Intro to formal languages. Regular grammars. Finite Automata.

Course:	Formal Languages & Finite Automata

Elaborated:
st. gr. FAF-223 Maria Colța

Verified:
asist. univ. Dumitru Crețu

Chișinău 2024

TABLE OF CONTENTS

THEORY

Formal Language

A formal language is considered to be a format used to transmit information from a sender to the one that receives it. The usual components of a language are:

- The alphabet: Set of valid characters;
- The vocabulary: Set of valid words;
- The grammar: Set of rules over the language.

Grammar

A regular grammar is a specific type of formal grammar that generates regular language. Usually a grammar is defined by:

- V_t: A finite set of terminal symbols
- V_n: A finite set of non-terminal symbols
- S: Start symbol
- P: Finite set of production rules

Finite Automaton

Finite automata theory focuses on algorithms for processing symbol strings and their set membership, defined by automaton rules. Key points include:

- Finite Set of States: Includes start states and accept states;
- Transitions: Triggered by input symbols between states;
- Types: Deterministic (DFA) and Nondeterministic (NFA):
- DFA: One state at a time, each input leads to one next state;
- NFA: Multiple states at once, multiple next states possible;
- Applications: Pattern matching, lexical analysis, network protocols.

OBJECTIVES

Tasks

- Discover what a language is and what it needs to have in order to be considered a formal one.
- Provide the initial setup for the evolving project that I will work on during this semester.
- According to the grammar definition:
    - Implement a type/class for the grammar;
    - Generate 5 valid strings from the language expressed by the given grammar;
    - Implement some functionality that would convert an object of type Grammar to one of type Finite Automaton;
    - For the Finite Automaton, add a method to check if the input string can be obtained via the state transition from it.

IMPLEMENTATION

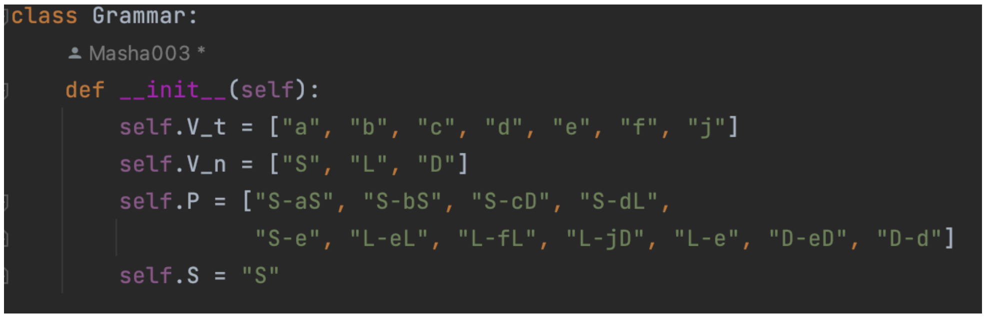
For these tasks, I was given the following grammar:

```
Variant 4:
VN={S, L, D},
VT={a, b, c, d, e, f, j},
P={
    S → aS
    S → bS
    S → cD
    S → dL
    S → e
    L → eL
    L → fL
    L → jD
    L → e
    D → eD
    D → d
}
```

- **Implement a type/class for the grammar;**

I implemented two classes: `Grammar` and `Finite Automaton`.

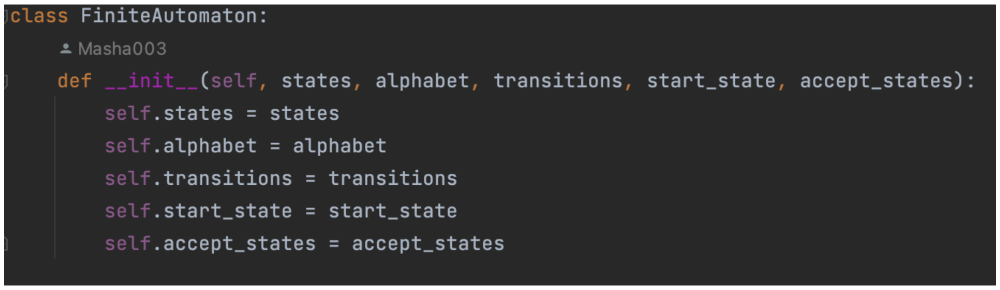
The given grammar was encapsulated within the `Grammar` class, which is responsible for generating strings that belong to the language defined by the grammar. I declared the terminal and non-terminal symbols, as well as the rules for the production rules and the start symbol.



```
class Grammar:
    Masha003 *
    def __init__(self):
        self.V_t = ["a", "b", "c", "d", "e", "f", "j"]
        self.V_n = ["S", "L", "D"]
        self.P = ["S-aS", "S-bS", "S-cD", "S-dL",
                  "S-e", "L-eL", "L-fL", "L-jD", "L-e", "D-eD", "D-d"]
        self.S = "S"
```

Figure 1. Grammar class definition

`FiniteAutomaton` class was implemented to simulate a finite automaton that can recognize a subset of the language described by the grammar. The FA is defined by its states, alphabet, transition functions, start state, and accept states.

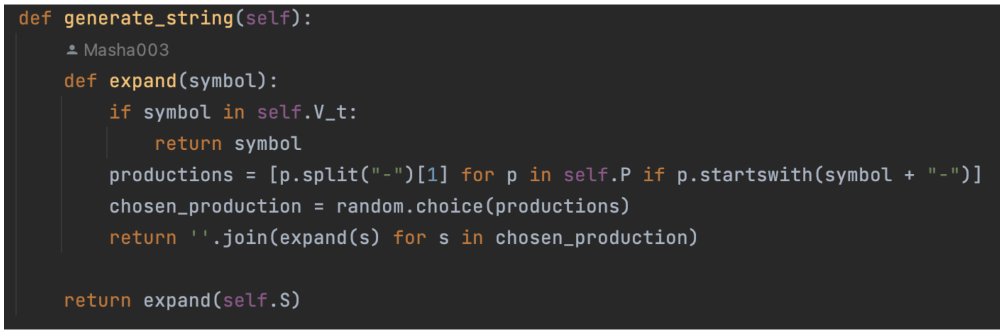


```python
class FiniteAutomaton:
    Masha003
    def __init__(self, states, alphabet, transitions, start_state, accept_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.start_state = start_state
        self.accept_states = accept_states
```

Figure 2. FiniteAutomaton class definition

- **Generate 5 valid strings from the language expressed by the given grammar;**

The string generation process is made using the `generate_string` method in the `Grammar` class. The generation process relies on recursively expanding non-terminal symbols based on the production rules until a string consisting solely of terminal symbols is produced.

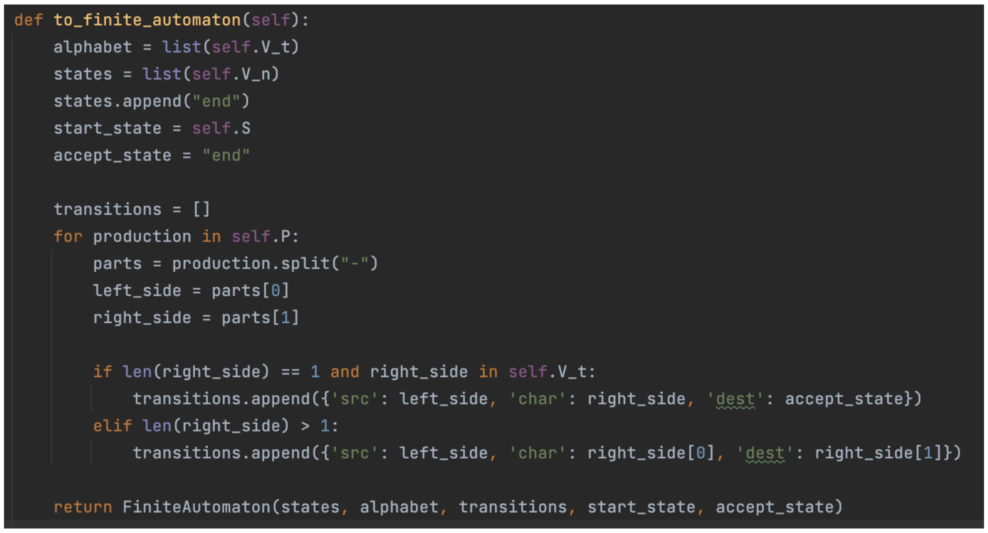


```python
def generate_string(self):
    Masha003
    def expand(symbol):
        if symbol in self.V_t:
            return symbol
        productions = [p.split("-")[1] for p in self.P if p.startswith(symbol + "-")]
        chosen_production = random.choice(productions)
        return ''.join(expand(s) for s in chosen_production)

    return expand(self.S)
```

Figure 3. `generate_string` method in `Grammar` class

- **Implement some functionality that would convert and object of type Grammar to one of type Finite Automaton;**

The `to_finite_automaton` method transforms a context-free grammar into a finite automaton by creating states from non-terminal symbols and an "end" state, and defining transitions based on the grammar's production rules. It interprets each production rule to either transition to an "end" state if the rule produces a terminal symbol, or move between states based on the first character of the rule's output, thus constructing a simplified automaton representation of the grammar.

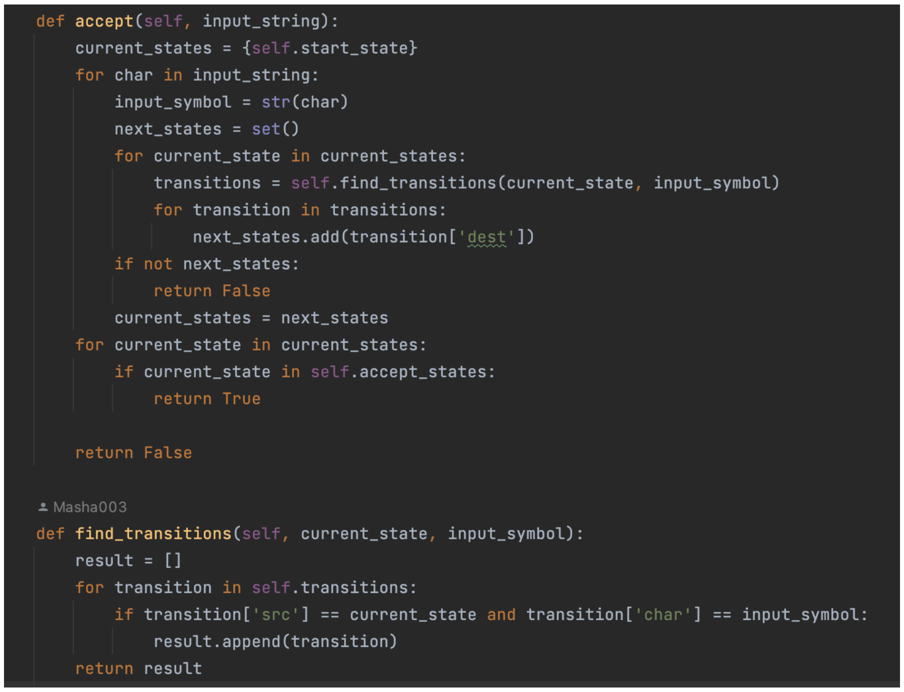


```python
def to_finite_automaton(self):
    alphabet = list(self.V_t)
    states = list(self.V_n)
    states.append("end")
    start_state = self.S
    accept_state = "end"

    transitions = []
    for production in self.P:
        parts = production.split("-")
        left_side = parts[0]
        right_side = parts[1]

        if len(right_side) == 1 and right_side in self.V_t:
            transitions.append({'src': left_side, 'char': right_side, 'dest': accept_state})
        elif len(right_side) > 1:
            transitions.append({'src': left_side, 'char': right_side[0], 'dest': right_side[1]})

    return FiniteAutomaton(states, alphabet, transitions, start_state, accept_state)
```

Figure 4. `to_finite_automathon` method in `Grammar` class

- **For the Finite Automaton, add a method to check if the input string can be obtained via the state transition from it.**

In this class, I implemented the `accept` method, which checks whether a given string is recognized by the automaton by simulating state transitions based on the input string and determining if the automaton can end in an accept state. It uses the find_transitions method to retrieve all the transitions available from a given `current_state` for a given `input_symbol`.

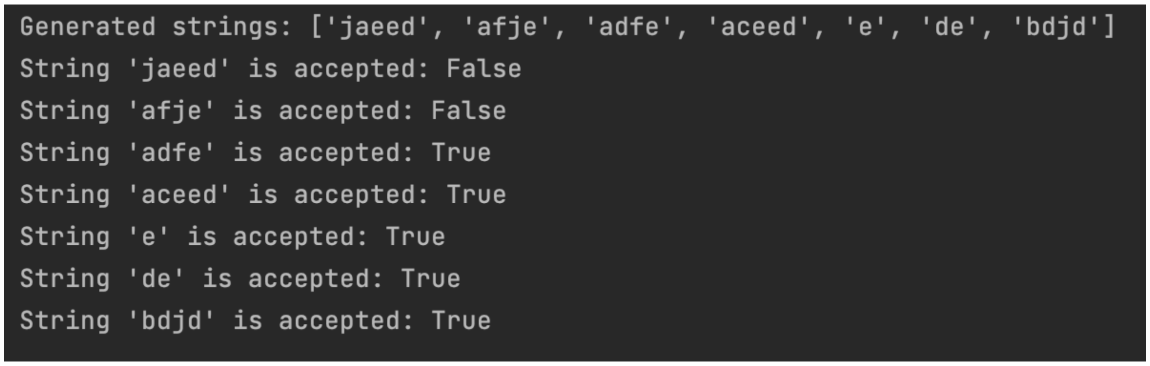


```python
def accept(self, input_string):
    current_states = {self.start_state}
    for char in input_string:
        input_symbol = str(char)
        next_states = set()
        for current_state in current_states:
            transitions = self.find_transitions(current_state, input_symbol)
            for transition in transitions:
                next_states.add(transition['dest'])
        if not next_states:
            return False
        current_states = next_states
    for current_state in current_states:
        if current_state in self.accept_states:
            return True

    return False

Masha003
def find_transitions(self, current_state, input_symbol):
    result = []
    for transition in self.transitions:
        if transition['src'] == current_state and transition['char'] == input_symbol:
            result.append(transition)
    return result
```

Figure 5. `accept` and `find_transitions` methods in `FiniteAutomaton` class

RESULTS

After running the code, we get the following result:



```
Generated strings: ['jaeed', 'afje', 'adfe', 'aceed', 'e', 'de', 'bdjd']
String 'jaeed' is accepted: False
String 'afje' is accepted: False
String 'adfe' is accepted: True
String 'aceed' is accepted: True
String 'e' is accepted: True
String 'de' is accepted: True
String 'bdjd' is accepted: True
```

Figure 1. The result of the program

Note: The first two strings were hardcoded to test that the Automaton doesn't accept the string if it doesn't follow the rules :).

CONCLUSION

This laboratory work helped me understand formal languages and automata, which are the foundations for understanding patterns in text. I focused on two classes: Grammar, where I learned to apply rules for generating string, and FiniteAutomaton, which identifies strings that fit within the given language framework.

The "Grammar" class is responsible for generating diverse text strings, as well as converting to FiniteAutomaton class. There I applied predefined production rules, to manipulate symbols and letters. For example, a transition like "S-aS", and generating sequences like "aaabbS" or "abcaaS" by progressively replacing symbols according to defined patterns. This hands-on exploration allowed me to understand the power of rule-based generation and comprehend how seemingly simple instructions can result in complex language variations.

Based on the constructed grammars, the "FiniteAutomaton" class checks if generated strings adhered to the defined rules. It checks each symbol and transition, accepting only those strings that followed the prescribed language structure. This process involved translating grammar rules into a network of states and transitions within the automaton, each symbol acting as a key. Through this conversion, I gained more knowledge about different formal language mechanisms and their ability to precisely model specific language constraints.

All in all, this lab was a deep dive into the theory and real-world application of how computers understand language. It showed me just how important these concepts are in everything from creating compilers to figuring out complex algorithms.