Ministerul Educaţiei şi Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 1:

Course: Formal Languages and Finite Automata
Topic: Intro to formal languages. Regular grammars. Finite Automata.

Elaborated:
st. gr. FAF-221 Chichioi Iuliana

Verified:

asist. univ. Cretu Dumitru

Chişinău - 2023

TABLE OF CONTENTS

THEORY

A **formal language** is a set of strings over a given alphabet, defined by specific rules or patterns. These rules are typically described using a formal system called a **grammar**, which consists of a set of production rules that dictate how strings are formed.

- **Alphabet**: The set of symbols from which strings are formed.
- **Terminal Symbols (VT)**: Symbols from the alphabet used directly in strings.
- **Non-terminal Symbols (VN)**: Symbols representing rules or patterns for generating strings.
- **Production Rules (P)**: Rules that specify how strings can be formed using non-terminal symbols.

A **finite automaton** is a mathematical model used to recognize or generate strings belonging to a formal language. There are several types of finite automata, including:

Deterministic Finite Automaton (DFA):
- Consists of a finite set of states, an alphabet, a transition function, an initial state, and a set of accepting states.
- The transition function maps each state and input symbol to a unique next state.
- Accepts a string if, after reading the entire string, it reaches an accepting state.

Nondeterministic Finite Automaton (NFA):
- Similar to a DFA but allows multiple possible transitions from a given state for a given input symbol.
- Accepts a string if there exists at least one path through the automaton that leads to an accepting state.

Nondeterministic Finite Automaton with ε-transitions (ε-NFA):
- Extends the NFA by allowing transitions on an empty string (ε).
- This allows transitions without consuming any input symbols.
- Can be converted to a DFA using the ε-closure and subset construction algorithms.

Nondeterministic Finite Automaton with ε-closures (ε-NFA):
- An extension of the ε-NFA that explicitly represents ε-transitions as part of the transition function.
- Simplifies the conversion process to a DFA by including ε-transitions in the transition diagram.

OBJECTIVES

1.Understanding Formal Languages: The primary objective is to understand what constitutes a formal language and the elements required for its definition, including alphabets, production rules, and grammar.

2.Project Setup: Set up the initial environment for the project by creating a GitHub repository and choosing a suitable programming language. The focus should be on selecting a language that simplifies the task without extensive setup requirements.

3.Modular Reports: Organize project reports separately to facilitate verification and understanding of the work done. Each report should focus on a specific aspect or functionality of the project.

4.Grammar Implementation: Implement a class to represent the grammar of the formal language defined by the given rules. This class should encapsulate the non-terminal symbols, terminal symbols, and production rules.

5.String Generation: Develop a function to generate five valid strings from the language defined by the

grammar. These strings should adhere to the production rules specified in the grammar.

6.Finite Automaton Conversion: Implement functionality to convert an object representing the grammar into an equivalent finite automaton. This conversion should preserve the language defined by the grammar.

7.Language Recognition: Add a method to the finite automaton class to determine whether a given input string belongs to the language represented by the finite automaton. This method should simulate state transitions based on the input string.

IMPLEMENTATION

For the implementation, **Python** was chosen as the programming language due to its familiarity and ease of use. Two main classes were implemented: `Grammar` and `FiniteAutomaton`. Below are the code snippets with explanations of their functionalities.

Grammar Class:

Constructor: Initializes the grammar with the start symbol, non-terminal symbols, terminal symbols, and production rules.

```
import random

1 usage  new *
class Grammar:
    new *
    def __init__(self):
        self.V_n = {'S', 'F', 'L'}
        self.V_t = {'a', 'b', 'c', 'd'}
        self.P = {
            'S': ['bS', 'aF', 'd'],
            'F': ['cF', 'dF', 'aL', 'b'],
            'L': ['aL', 'c']
        }

```

Figure 1 class Grammar

Generate String: Generates a valid string from the grammar based on the production rules. It recursively selects production choices until reaching the maximum length or a terminal symbol.

```
def generate_string(self, start_symbol='S', max_length=10):
    if max_length == 0:
        return ''
    if start_symbol not in self.V_n:
        return start_symbol
    production_choices = self.P[start_symbol]
    chosen_production = random.choice(production_choices)
    generated_string = ''
    for symbol in chosen_production:
        generated_string += self.generate_string(symbol, max_length - 1)
    return generated_string
```

Figure 2 def generate_string

Convert to Finite Automaton: Converts the grammar to an equivalent finite automaton. It creates states, alphabet, and transitions based on the grammar rules.

```
def to_finite_automaton(self):
    states = self.V_n.union(self.V_t)
    alphabet = self.V_t
    transitions = {}
    for non_terminal, productions in self.P.items():
        for production in productions:
            if len(production) == 2:
                if (non_terminal, production[0]) in transitions:
                    transitions[(non_terminal, production[0])].append(production[1])
                else:
                    transitions[(non_terminal, production[0])] = [production[1]]
            elif len(production) == 1:
                if (non_terminal, production) in transitions:
                    transitions[(non_terminal, production)].append(non_terminal)
                else:
                    transitions[(non_terminal, production)] = [non_terminal]
    initial_state = 'S'
    accepting_states = {'S', 'F', 'L'}  # Assuming all non-terminals are accepting states
    return FiniteAutomaton(states, alphabet, transitions, initial_state, accepting_states)
```

Figure 3 def to_finite_automaton

FiniteAutomaton Class:

Constructor: Initializes the finite automaton with states, alphabet, transitions, initial state, and accepting states.

```
class FiniteAutomaton:
    new *
    def __init__(self, states, alphabet, transitions, initial_state, accepting_states):
        self.states = states
        self.alphabet = alphabet
        self.transitions = transitions
        self.initial_state = initial_state
        self.accepting_states = accepting_states
```

Figure 4 class FiniteAutomaton

String Belongs to Language: Checks if an input string belongs to the language represented by the finite automaton. It simulates state transitions based on the input string and determines if it reaches an accepting state.

```
def string_belongs_to_language(self, input_string):
    current_states = {self.initial_state}
    for symbol in input_string:
        next_states = set()
        for state in current_states:
            if (state, symbol) in self.transitions:
                next_states.update(self.transitions[(state, symbol)])
        current_states = next_states
    return any(state in self.accepting_states for state in current_states)
```

Variable "transitions"
Inferred type: Any

Figure 5 def string_belongs_to_language

The `generate_string method` recursively generates strings based on the grammar's production

rules. It selects random choices for non-terminal symbols until reaching a terminal symbol or the maximum length. In the `to_finite_automaton` method, the grammar is converted to a finite automaton by creating states, transitions, and accepting states based on the grammar rules. The `string_belongs_to_language` method simulates state transitions of the finite automaton based on the input string. If the automaton reaches an accepting state, the input string is considered to belong to the language.

RESULTS

The main function tests the functionality of both classes by generating strings from the grammar and checking if they belong to the language represented by the finite automaton. The results are printed to verify the correctness of the implementation.

```python
# Test Grammar functionality
grammar = Grammar()

# Generate 5 valid strings from the grammar
print("Generated strings:")
for _ in range(5):
    generated_string = grammar.generate_string()
    print(generated_string)

# Convert Grammar to Finite Automaton
finite_automaton = grammar.to_finite_automaton()

# Test strings with the Finite Automaton
test_strings = ['abcc', 'bdab', 'cddd', 'abcb', 'ad']
for string in test_strings:
    if finite_automaton.string_belongs_to_language(string):
        print(f"'{string}' belongs to the language.")
    else:
        print(f"'{string}' does not belong to the language.")
```

Figure 6 test

SCREENSHOTS

```
Generated strings:
bd
aaaac
d
d
adaac
'abcc' belongs to the language.
'bdab' belongs to the language.
'cddd' does not belong to the language.
'abcb' belongs to the language.
'ad' belongs to the language.
```

Figure 7 results

CONCLUSION

In conclusion, the implementation effectively demonstrates the conversion of a context-free grammar to an equivalent finite automaton and the recognition of strings belonging to the language defined by the grammar. By utilizing Python as the programming language, the implementation ensures readability, flexibility, and ease of maintenance.

Through the Grammar class, the essential components of the grammar, including start symbol, non-terminal symbols, terminal symbols, and production rules, are defined and utilized for generating valid strings. The generation process employs a recursive approach, selecting random production choices until a terminal symbol is reached or the maximum length is exceeded.

The FiniteAutomaton class encapsulates the finite automaton's components, such as states, alphabet, transitions, initial state, and accepting states. The conversion process from the grammar to the finite automaton involves creating states and transitions based on the grammar's production rules.

Furthermore, the implementation includes functionality to test whether an input string belongs to the language represented by the finite automaton. By simulating state transitions based on the input string, the implementation determines whether the automaton reaches an accepting state, indicating the string's validity in the language.

Overall, the implementation achieves the stated objectives of understanding formal languages, implementing a grammar-to-finite-automaton conversion, and testing string recognition. Through modular design and clear documentation, the codebase facilitates understanding, maintenance, and further development of language processing functionalities.