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

Laboratory work nr. 1
Course: Formal languages and finite automata
Topic: Intro to formal languages. Regular grammars. Finite automata

Elaborated:
st. gr. FAF-221 Berco Andrei

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theory

Formal languages are abstract languages used to represent strings of symbols according to certain rules. These rules are defined by a grammar, which dictates the structure and formation of valid strings within the language. Regular grammars are a type of formal grammar that generates regular languages.

A regular grammar consists of a set of production rules that define how strings in the language are formed. These rules typically take the form of substitutions or transformations. Regular grammars are characterized by their simplicity and limited expressive power compared to other types of grammars. They can be described using regular expressions, finite automata, or regular grammars themselves. Finite automata are abstract machines used to recognize and accept strings belonging to a formal language. They consist of a finite set of states, a finite alphabet of input symbols, a transition function that maps states and input symbols to other states, an initial state, and a set of accepting states.

Finite automata come in various forms such as deterministic finite automata (DFA), non-deterministic finite automata (NFA), and finite automata with ε-transitions (ε-NFA).

Objectives:

1. Discover what a language is and what it needs to have in order to be considered a formal one
2. Provide the initial setup for the evolving project that you will work on during this semester
3. Create GitHub repository to deal with storing and updating your project
4. Choose a programming language
5. Store reports separately in a way to make verification of your work simpler
6. Implement a type/class for your grammar
7. Add one function that would generate 5 valid strings from the language expressed by your given grammar
8. Implement functionality to convert an object of type Grammar to one of type Finite Automaton
9. Add a method to the Finite Automaton that checks if an input string can be obtained via state transitions from it

Implementation Description

For implementation I chose to use Python, because it is a familiar language. First, I implemented 2 classes: Grammar and FiniteAutomatom. Below will be the code snippets and short explanation of their functionalities.

First comes class Grammar, with a constructor, method to generate valid strings, method to test string generation and conversion from Grammar to Finite Automatom:

```
class Grammar:
    def __init__(self):
        self.S = 'S'
        self.V_n = ['S','D','R']
        self.V_t = ['a','b','c','d','f']
        self.P = {
            'S':['aS','bD','fR'],
            'D':['cD','dR','d'],
            'R':['bR','f']
        }
```

This is a constructor for class Grammar. Here are declared and assigned with values according to given grammar:

- Start symbol
- Non-terminal symbols
- Terminal symbols
- Production rules

```
def generateString(self):
        import random

        def generateFromSymbol(symbol):
            production = []
            if symbol in self.V_t:
                return symbol
            else:
                production = random.choice(self.P[symbol])
                return ''.join([generateFromSymbol(s) for s in production])

        return generateFromSymbol(self.S)
```

Here is the method to generate strings. Overall, it implies a recursive sub method where the strings are generated symbol by symbol. Starting with the start symbol “S”, we choose at random an available transition from production rules. If it a terminal symbol, we concatenate it to the final

string, if it is a non-terminal, we go again and again. At final, the method returns the generated string.

```
    def toFiniteAutomaton(self):
        return FiniteAutomaton()
```

This is a method from Grammar to create and return an object of type FiniteAutomaton

```
class FiniteAutomaton :
    def __init__(self):
        self.Q = ['S','D','R','X']
        self.Sigma = ['a','b','c','d','f']
        self.Delta = {
            ('S', 'a'): 'S',
            ('S', 'b'): 'D',
            ('S', 'f'): 'R',
            ('D', 'c'): 'D',
            ('D', 'd'): 'R',
            ('R', 'b'): 'R',
            ('D', 'd'): 'X',
            ('R', 'f'): 'X',

        }
        self.q0 = 'S'
        self.F = {'X'}
```

Here is declaration of FiniteAutomaton and its constructor. Here we declare and assign with values according to our grammar:

- Q – set of states
- Sigma – alphabet
- Delta – transition function
- q0 – initial state
- F – final state

```
def stringBelongToLanguage(self,w):
        currentState = self.q0
        leng=0
        for letter in w:
            if (currentState, letter) in self.Delta:
                if (currentState, letter)==('D', 'd') and leng==len(w)-1:
                    currentState = 'X'
                    return currentState in self.F
                elif (currentState, letter)==('D', 'd'):
```

```
                    currentState = 'R'
                else:
                    currentState = self.Delta[(currentState, letter)]
            else:
                return False
            leng+=1
        return currentState in self.F
```

This is the method that checks if an input string is valid according to our grammar. In general, the method works based on checking if the states and transitions are valid according to *Delta*. Because in my variant is represented an NFA (D can change either into cD or d), I added some conditions. I check if there is a transition like (D, d) and the “d” is the final character in a word then function returns true. If not, the variable currentState = “R” (second transition) and method continues checking. If method encounters an invalid transition, it immediately returns false answer.

```
#Test Grammar functionality
grammar = Grammar()
grammar.generate5Words()

finiteAutomatom = grammar.toFiniteAutomaton()
testString1 = 'aaabd'
testString2 = 'afbbbf'
print(f"\nString '{testString1}': \n Validation:
{finiteAutomatom.stringBelongToLanguage(testString1)}\n")
print(f"String '{testString2}': \n Validation:
{finiteAutomatom.stringBelongToLanguage(testString2)}")
```

Here is the main function to test both classes.

Screenshots

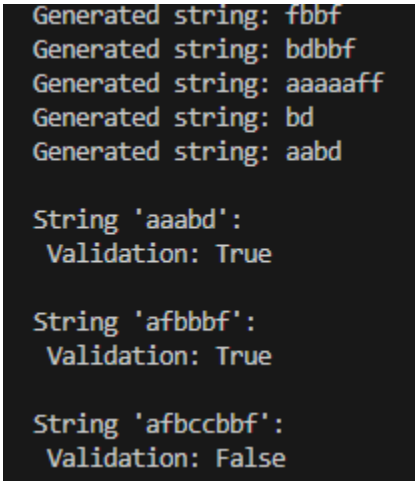

Figure 1: Results of testing

Conclusions

Grammars and finite automata are fundamental concepts in formal language theory, with each capable of representing a class of languages. Grammars describe the syntax of languages through production rules, while finite automata recognize the strings generated by these languages. One of the objectives mentioned earlier was to implement functionality to convert a grammar to a finite automaton.

This conversion process involves analyzing the production rules of the grammar and mapping them to states and transitions in the finite automaton. The resulting finite automaton can then be used to recognize strings generated by the grammar, providing a computational model for the language specified by the grammar. The finite automaton determines whether a given input string is recognized by the language it represents. By simulating state transitions according to the input string, the automaton can determine whether the string is accepted or rejected. Acceptance typically occurs if the automaton ends in an accepting state after processing the entire input string. Finite automata are limited in their computational power compared to grammars. They can recognize regular languages, while grammars can represent more complex language classes such as context-free languages. The provided implementations focus on regular grammars and finite automata, which are foundational concepts in formal language theory and have applications in various areas including compiler design, natural language processing, and string processing algorithms.