MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică
Departamentul Inginerie Software și Automatică

Zagorodniuc Anastasia FAF-223

Report

Intro to formal languages. Regular grammar. Finite Automata.

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

Objectives

- 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 you will work on during this semester. You can deal with each laboratory work as a separate task or project to demonstrate your understanding of the given themes, but you also can deal with labs as stages of making your own big solution, your own project. Do the

- Implement a type/class for your grammar

- Add one function that would generate 5 valid strings from the language expressed by your given grammar

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

- For the Finite Automaton, please add a method that checks if an input string can be obtained via the state transition from it

Code

Points A and B: carried out

A basic grammar-based string generator with recursive techniques was implemented in this code. Beginning with a simple symbol, it follows a set of guidelines to substitute other symbols or letters until it reaches the end of a string. In order to do this, it creates strings by recursively expanding non-terminal symbols in accordance with production rules that are chosen at random. It does this by using a map to link non-terminal symbols with their production rules. It repeats this five times to illustrate various scenarios.

```
using System;

using System.Collections.Generic;



class Grammar

{

    private readonly Dictionary<char, List<string>> rules = new
Dictionary<char, List<string>>();

    private readonly Random random = new Random();

    private readonly HashSet<string> generatedStrings = new
HashSet<string>();
```

```
public Grammar()
{
    InitializeRules();
}

private void InitializeRules()
{
    rules['S'] = new List<string> { "aB" };
    rules['B'] = new List<string> { "bS", "aC", "c" };
    rules['C'] = new List<string> { "bD" };
    rules['D'] = new List<string> { "c", "aC" };
}

public string GenerateString()
{
    string generatedString;
    do
    {
        generatedString = Expand('S');
    } while (generatedStrings.Contains(generatedString));

    generatedStrings.Add(generatedString);
    return generatedString;
}

private string Expand(char symbol)
{
        var result = "";

        if (rules.ContainsKey(symbol))

        {

            var possibleProductions = rules[symbol];

            var production =
possibleProductions[random.Next(possibleProductions.Count)];

            foreach (var nextSymbol in production)

            {

                result += Expand(nextSymbol);

            }

        }

        else

        {

            result += symbol;

        }

        return result;

    }

}
```

Point D: implemented

A finite automaton that analyzes character strings to see if they adhere to a predetermined pattern as specified by the automaton's rules was created in this software. It employs a series of stages, or states, and modifies them in response to the letters it encounters. Based on the final destination, it determines if the string is acceptable or not.

```
class FiniteAutomaton

{

    private readonly HashSet<string> states = new HashSet<string>
{ "S", "B", "C", "D" };

    private readonly HashSet<char> alphabet = new HashSet<char>
{ 'a', 'b', 'c' };
```

```
    private readonly Dictionary<Tuple<string, char>, string>
transitions = new Dictionary<Tuple<string, char>, string>

    {

        { Tuple.Create("S", 'a'), "B" },

        { Tuple.Create("B", 'b'), "S" },

        { Tuple.Create("B", 'a'), "C" },

        { Tuple.Create("B", 'c'), "c" },

        { Tuple.Create("C", 'b'), "D" },

        { Tuple.Create("D", 'c'), "c" },

        { Tuple.Create("D", 'a'), "C" }

    };

    private string currentState = "S";

    private readonly HashSet<string> acceptingStates = new
HashSet<string> { "c" };


    public bool Transition(char symbol)

    {

        var key = Tuple.Create(currentState, symbol);


        if (transitions.ContainsKey(key))

        {

            currentState = transitions[key];

            return true;

        }

        return false;

    }


    public bool IsStringAccepted(string inputString)

    {
        foreach (var symbol in inputString)
        {
            if (!alphabet.Contains(symbol) || !Transition(symbol))
            {
                return false;
            }
        }
        return acceptingStates.Contains(currentState);
    }
}



using System;

class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Enter lab:");
        var labInput = Console.ReadLine();

        switch (labInput)
        {
            case "lab1":
                RunLab1();
                break;
            default:
                Console.WriteLine("Coming soon");
                break;
        }
    }

    private static void RunLab1()
    {
        var grammar = new Grammar();
        for (var i = 0; i < 5; i++)
        {
            Console.WriteLine(grammar.GenerateString());
        }

        var fa = new FiniteAutomaton();
        Console.WriteLine("Enter a string to check:");
        var inputString = Console.ReadLine();

        var isValid = fa.IsStringAccepted(inputString);
        if (isValid)
        {
            Console.WriteLine("The string is accepted by the
automaton.");
        }
        else
        {
            Console.WriteLine("The string is not accepted by the
automaton.");
        }
        }

}
```

Conclusion:

The implemented solution provides a foundation for working with formal languages, starting with grammars and transitioning to finite automata. The **Grammar** class facilitates the generation of valid strings, and the conversion to a **FiniteAutomaton** allows for checking whether an input string is accepted by the defined automaton. This project lays the groundwork for further exploration and development in the field of formal languages and automata.