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

Laboratory work 1:

Intro to formal languages. Regular grammars. Finite Automata.

Elaborated:
st. gr. FAF-223

Ciornii Alexandr

Verified:
asist. Univ.

Cretu Dumitru

Chișinău – 2024

Overview

A formal language can be considered to be the media or the format used to convey information from a sender entity 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/constraints over the lang.

Now these components can be established in an infinite amount of configurations, which actually means that whenever a language is being created, it's components should be selected in a way to make it as appropriate for it's use case as possible. Of course sometimes it is a matter of preference, that's why we ended up with lots of natural/programming/markup languages which might accomplish the same thing.

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. 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 following:

3.a. Create GitHub repository to deal with storing and updating your project;

b. Choose a programming language. Pick one that will be easiest for dealing with your tasks, you need to learn how to solve the problem itself, not everything around the problem (like setting up the project, launching it correctly and etc.);

c. Store reports separately in a way to make verification of your work simpler (duh)

4.According to your variant number, get the grammar definition and do the following:

a. Implement a type/class for your grammar;

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

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

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

Variant 3:

VN={S, D, R},

VT={a, b, c, d, f},

P={

S → aS

S → bD

S → fR

D → cD

D → dR

R → bR

R → f

D → d

}

Implementation in C# language:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

class GrammarToNFAConverter
{



   static void Main()
   {
      // Define a context-free grammar
      Grammar grammar = new Grammar();
      grammar.AddRule("S", "aS");
      grammar.AddRule("S", "bD");
      grammar.AddRule("S", "fR");
      grammar.AddRule("D", "cD");
      grammar.AddRule("D", "dR");
      grammar.AddRule("R", "bR");
      grammar.AddRule("R", "f");
      grammar.AddRule("D", "d");

      var nfa = new NFA();
      // States
      nfa.AddState(0, isAccepting: false); // Initial state
      nfa.AddState(1, isAccepting: false); // Accepting state
      nfa.AddState(2, isAccepting: false); // Accepting state
      nfa.AddState(3, isAccepting: true); // Accepting state

      // Transitions
      nfa.AddTransition(0, 'a', 0);
      nfa.AddTransition(0, 'b', 1);
      nfa.AddTransition(0, 'f', 2);
      nfa.AddTransition(1, 'c', 1);
```

```
        nfa.AddTransition(1, 'd', 2);
        nfa.AddTransition(1, 'd', 3);
        nfa.AddTransition(2, 'b', 2);
        nfa.AddTransition(2, 'f', 3);



        // Check if a string is accepted by the NFA
        for (int i = 0; i < 5; i++)
        {
            string test = grammar.GenerateString();
            Console.WriteLine(test);
            bool isAccepted = nfa.Accepts(test);
            Console.WriteLine($"String '{test}' is {(isAccepted ? "accepted" : "rejected")} by the NFA.");
        }
    }

}




class GrammarCringe
{
    private Dictionary<char, List<string>> rules = new Dictionary<char, List<string>>();
    private char startSymbol;

    public void AddRule(char nonTerminal, string production)
    {
        if (!rules.ContainsKey(nonTerminal))
        {
            rules[nonTerminal] = new List<string>();
        }
        rules[nonTerminal].Add(production);
    }

    public void SetStartSymbol(char startSymbol)
    {
        this.startSymbol = startSymbol;
    }

    public char GetStartSymbol()
    {
        return startSymbol;
    }

    public Dictionary<char, List<string>> GetRules()
    {
        return rules;
    }

    public IEnumerable<char> GetNonTerminals()
    {
        return rules.Keys;
    }

    public IEnumerable<char> GetTerminals()
    {
        return rules.Values.SelectMany(p => p).SelectMany(p => p).Where(c => !char.IsUpper(c)).Distinct();
    }
}
class Grammar
{
    private Dictionary<string, HashSet<string>> productionRules;
    private Random random;

    public Grammar()
    {
```

```
        productionRules = new Dictionary<string, HashSet<string>>();
        random = new Random();
    }

    public void AddRule(string nonTerminal, string production)
    {
        if (!productionRules.ContainsKey(nonTerminal))
        {
            productionRules[nonTerminal] = new HashSet<string>();
        }
        productionRules[nonTerminal].Add(production);
    }

    public string GenerateString()
    {
        return GenerateString("S");
    }

    private string GenerateString(string symbol)
    {
        StringBuilder result = new StringBuilder();

        if (productionRules.ContainsKey(symbol))
        {
            var possibleProductions = new List<string>(productionRules[symbol]);
            int selectedProductionIndex = random.Next(possibleProductions.Count);
            string selectedProduction = possibleProductions[selectedProductionIndex];

            foreach (char c in selectedProduction)
            {
                if (char.IsUpper(c)) // Non-terminal symbol
                {
                    result.Append(GenerateString(c.ToString()));
                }
                else
                {
                    result.Append(c);
                }
            }
        }

        return result.ToString();
    }
}
class NFATransition
{
    public char Symbol { get; set; }
    public int TargetState { get; set; }
}

class NFAState
{
    public int StateId { get; set; }
    public List<NFATransition> Transitions { get; } = new List<NFATransition>();
    public bool IsAccepting { get; set; }
}

class NFA
{
    private List<NFAState> states = new List<NFAState>(10);

    public void AddState(int stateId, bool isAccepting = false)
    {
        states.Add(new NFAState { StateId = stateId, IsAccepting = isAccepting });
    }

    public void AddTransition(int sourceState, char symbol, int targetState)
```

```
    {
        var transition = new NFATransition { Symbol = symbol, TargetState = targetState };
        states[sourceState].Transitions.Add(transition);
    }

    public bool Accepts(string input)
    {
        var currentStates = new HashSet<int> { 0 }; // Start with the initial state
        foreach (char symbol in input)
        {
            var nextStates = new HashSet<int>();
            foreach (var state in currentStates)
            {
                foreach (var transition in states[state].Transitions)
                {
                    if (transition.Symbol == symbol)
                    {
                        nextStates.Add(transition.TargetState);
                    }
                }
            }
            currentStates = nextStates;
        }

        // Check if any of the current states is an accepting state
        return currentStates.Any(state => states[state].IsAccepting);
    }
}
```

Results:

```
abcd
String 'abcd' is accepted by the NFA.
bdbf
String 'bdbf' is accepted by the NFA.
ff
String 'ff' is accepted by the NFA.
abdf
String 'abdf' is accepted by the NFA.
bcd
String 'bcd' is accepted by the NFA.
```

```
ff
String 'ff' is accepted by the NFA.
aaaaff
String 'aaaaff' is accepted by the NFA.
fbf
String 'fbf' is accepted by the NFA.
ff
String 'ff' is accepted by the NFA.
aff
String 'aff' is accepted by the NFA.
```

```
fbf
String 'fbf' is accepted by the NFA.
bdbbf
String 'bdbbf' is accepted by the NFA.
ff
String 'ff' is accepted by the NFA.
aaafbbf
String 'aaafbbf' is accepted by the NFA.
abd
String 'abd' is accepted by the NFA.
```

Conclusion:

The program demonstrates the conversion of a given context-free grammar to a non-deterministic finite automaton (NFA) in C#. Here are some key points about the program:

Context-Free Grammar Definition:

The context-free grammar is defined using non-terminals (VN), terminals (VT), and production rules (P).

The grammar rules are represented using the Grammar class, allowing the specification of non-terminals, terminals, start symbols, and production rules.

NFA Conversion:

The ConvertToNFA function in the GrammarToNFAConverter class takes a context-free grammar as input and generates an NFA.

States and transitions are created based on the grammar rules, and the resulting NFA is constructed.

NFA Representation:

The NFA is represented using the NFA class, which includes methods to add states, set the initial state, add transitions, and check whether a given input string is accepted.

Transitions are stored in a dictionary, and epsilon transitions are represented by the constant string "ε".

Example Usage:

The example provided in the Main method demonstrates the conversion and usage of the program for a specific context-free grammar.

The program then displays the transition table of the generated NFA and checks whether the input string "afbd" is accepted by the NFA.

Extensibility:

The program can be easily extended or modified to handle different context-free grammars by updating the grammar rules in the Grammar class.

Simplicity and Limitations:

The program is kept simple for educational purposes and may not handle all possible edge cases or complex grammars.

For more sophisticated applications, additional features and error handling may be required.

Overall, this program serves as a basic example of how one can convert a context-free grammar to an NFA in C#, and it can be a starting point for further exploration and enhancements based on specific needs.