Ministerul Educaţiei și 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 Călugăreanu Ana

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.

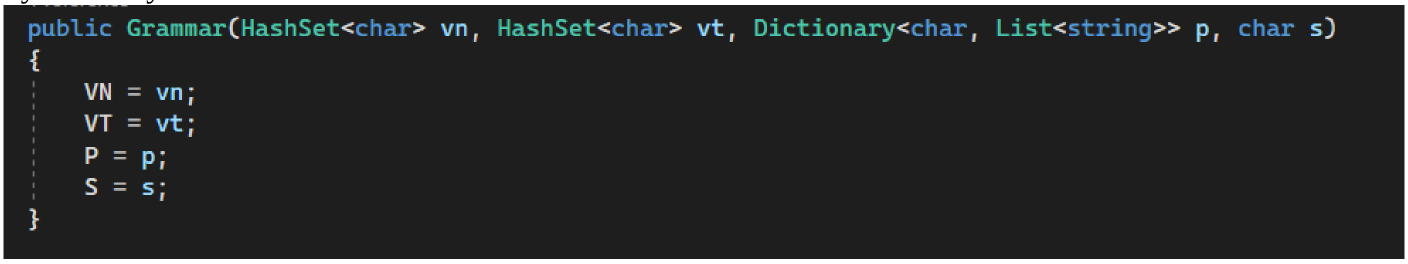
7. **Language Recognition:** Add a method to the finite automaton class to determine whether a given input string belongs to the language represented by the finite automaton. This method should simulate state transitions based on the input string.

IMPLEMENTATION

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

Grammar Class:

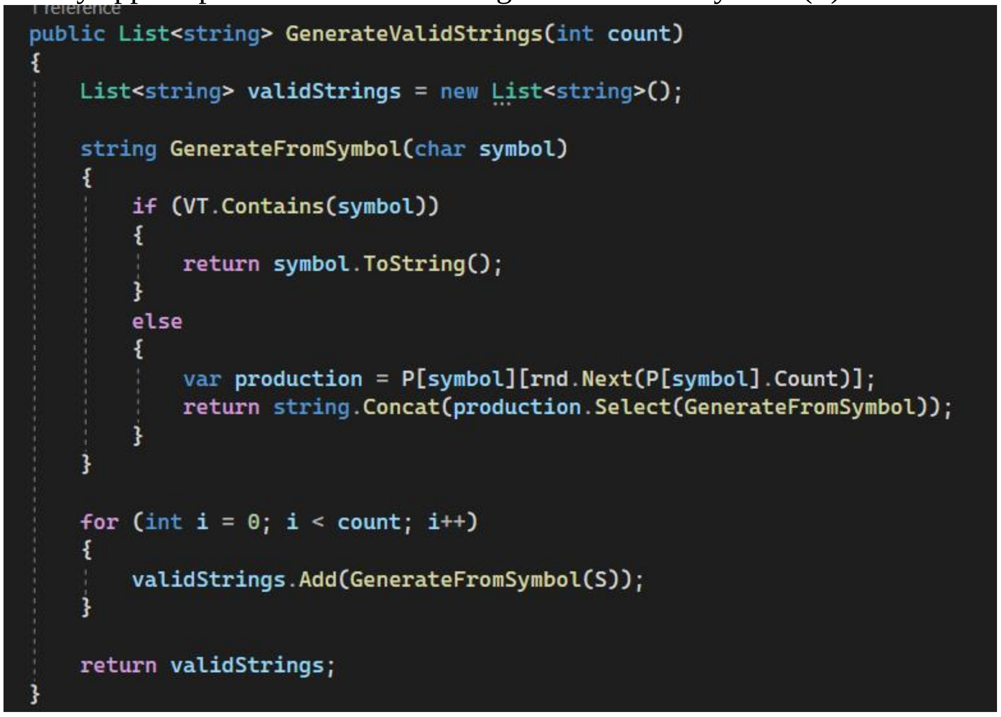
Constructor: Initializes a new instance of the Grammar class with specified non-terminals (VN), terminals (VT), production rules (P), and the start symbol (S). This setup allows the grammar to be defined dynamically at runtime.



```
public Grammar(HashSet<char> vn, HashSet<char> vt, Dictionary<char, List<string>> p, char s)
{
    VN = vn;
    VT = vt;
    P = p;
    S = s;
}
```

Figure 1 class Grammar

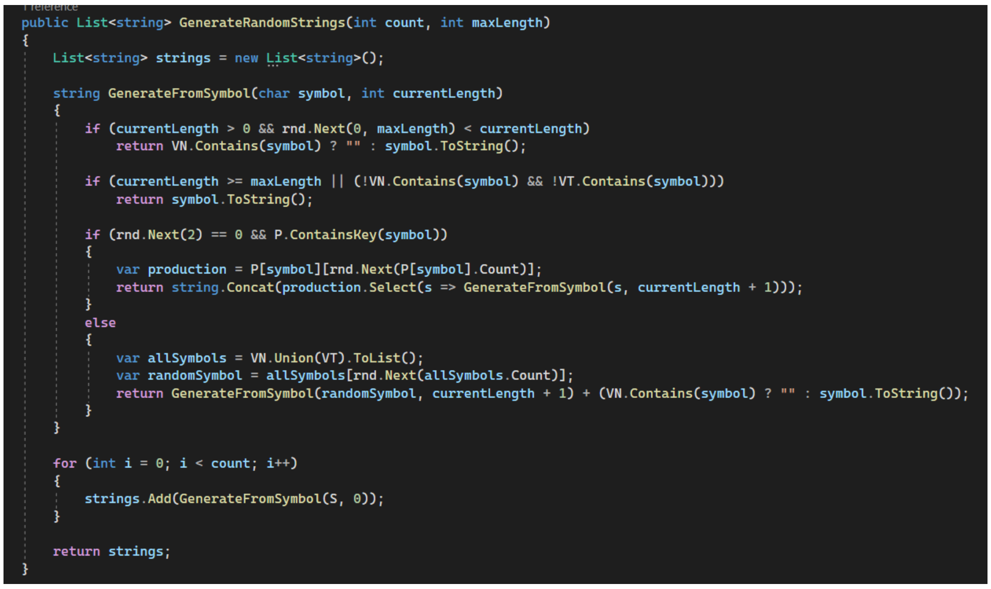
Generate String: Generates a list of count strings that are valid according to the grammar's production rules. It recursively applies production rules starting from the start symbol (S) to construct each string.



```
public List<string> GenerateValidStrings(int count)
{
    List<string> validStrings = new List<string>();

    string GenerateFromSymbol(char symbol)
    {
        if (VT.Contains(symbol))
        {
            return symbol.ToString();
        }
        else
        {
            var production = P[symbol][rnd.Next(P[symbol].Count)];
            return string.Concat(production.Select(GenerateFromSymbol));
        }
    }

    for (int i = 0; i < count; i++)
    {
        validStrings.Add(GenerateFromSymbol(S));
    }

    return validStrings;
}
```

Figure 2 Generate ValidStrings Method

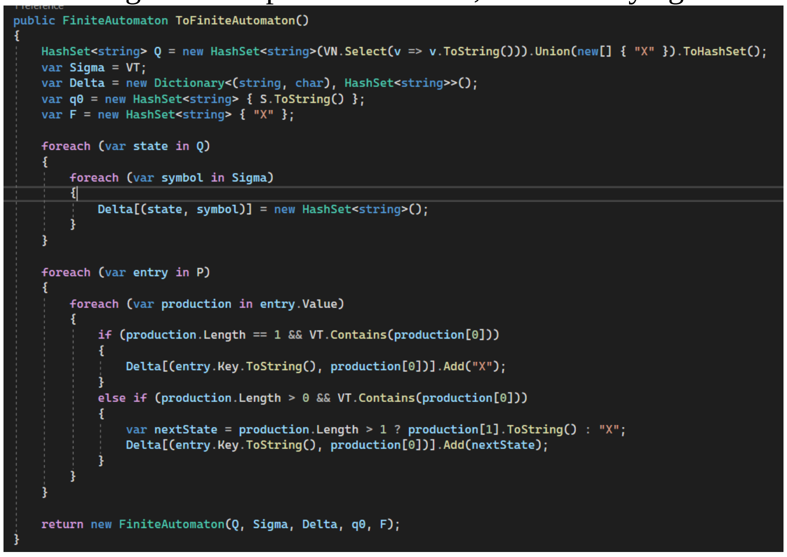
GenerateRandomStrings Method: Produces a list of strings of varying lengths, up to maxLength, by randomly selecting production rules to follow or inserting random symbols, potentially creating strings not strictly valid according to the grammar.



```csharp
1 reference
public List<string> GenerateRandomStrings(int count, int maxLength)
{
    List<string> strings = new List<string>();

    string GenerateFromSymbol(char symbol, int currentLength)
    {
        if (currentLength > 0 && rnd.Next(0, maxLength) < currentLength)
            return VN.Contains(symbol) ? "" : symbol.ToString();

        if (currentLength >= maxLength || (!VN.Contains(symbol) && !VT.Contains(symbol)))
            return symbol.ToString();

        if (rnd.Next(2) == 0 && P.ContainsKey(symbol))
        {
            var production = P[symbol][rnd.Next(P[symbol].Count)];
            return string.Concat(production.Select(s => GenerateFromSymbol(s, currentLength + 1)));
        }
        else
        {
            var allSymbols = VN.Union(VT).ToList();
            var randomSymbol = allSymbols[rnd.Next(allSymbols.Count)];
            return GenerateFromSymbol(randomSymbol, currentLength + 1) + (VN.Contains(symbol) ? "" : symbol.ToString());
        }
    }

    for (int i = 0; i < count; i++)
    {
        strings.Add(GenerateFromSymbol(S, 0));
    }

    return strings;
}
```

Figure 3 Generate RandomStrings Method

ToFiniteAutomaton Method: Converts the grammar into a finite automaton (FiniteAutomaton) representation. This involves creating states for each non-terminal and an additional "accept" state ("X"), defining transitions based on the grammar's production rules, and identifying the start and accept states.

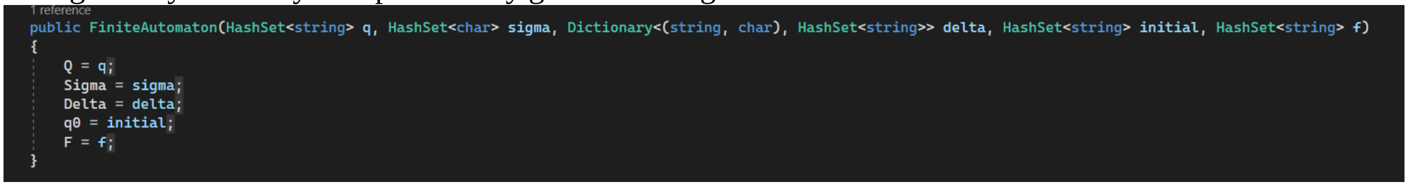


```csharp
1 reference
public FiniteAutomaton ToFiniteAutomaton()
{
    HashSet<string> Q = new HashSet<string>(VN.Select(v => v.ToString())).Union(new[] { "X" }).ToHashSet();
    var Sigma = VT;
    var Delta = new Dictionary<(string, char), HashSet<string>>();
    var q0 = new HashSet<string> { S.ToString() };
    var F = new HashSet<string> { "X" };

    foreach (var state in Q)
    {
        foreach (var symbol in Sigma)
        {
            Delta[(state, symbol)] = new HashSet<string>();
        }
    }

    foreach (var entry in P)
    {
        foreach (var production in entry.Value)
        {
            if (production.Length == 1 && VT.Contains(production[0]))
            {
                Delta[(entry.Key.ToString(), production[0])].Add("X");
            }
            else if (production.Length > 0 && VT.Contains(production[0]))
            {
                var nextState = production.Length > 1 ? production[1].ToString() : "X";
                Delta[(entry.Key.ToString(), production[0])].Add(nextState);
            }
        }
    }

    return new FiniteAutomaton(Q, Sigma, Delta, q0, F);
}
```

Figure 4 ToFiniteAutomaton Method

FiniteAutomaton Class:

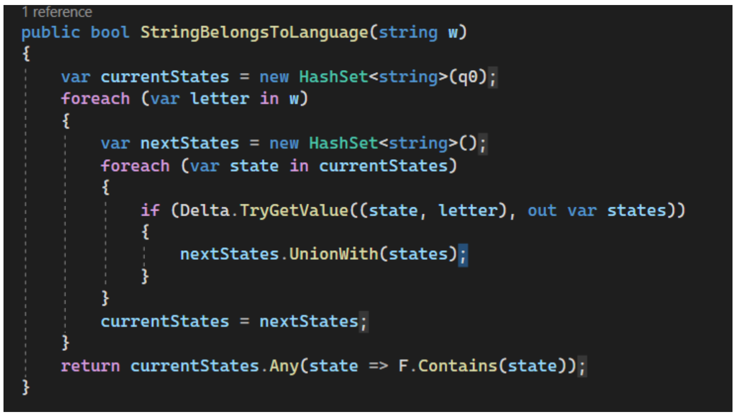
Constructor: Initializes a new instance of the FiniteAutomaton class with specified states (Q), alphabet (Sigma), transition rules (Delta), initial states (q0), and accepting states (F). This allows the automaton to be configured dynamically to represent any given formal grammar.



```csharp
1 reference
public FiniteAutomaton(HashSet<string> q, HashSet<char> sigma, Dictionary<(string, char), HashSet<string>> delta, HashSet<string> initial, HashSet<string> f)
{
    Q = q;
    Sigma = sigma;
    Delta = delta;
    q0 = initial;
    F = f;
}
```

Figure 5 class FiniteAutomaton

StringBelongsToLanguage Method: Determines whether a given string (w) belongs to the language defined by the finite automaton. It simulates the automaton's transitions based on the input string, tracking the current states and checking if any of the final states are accepting states after processing the entire string.

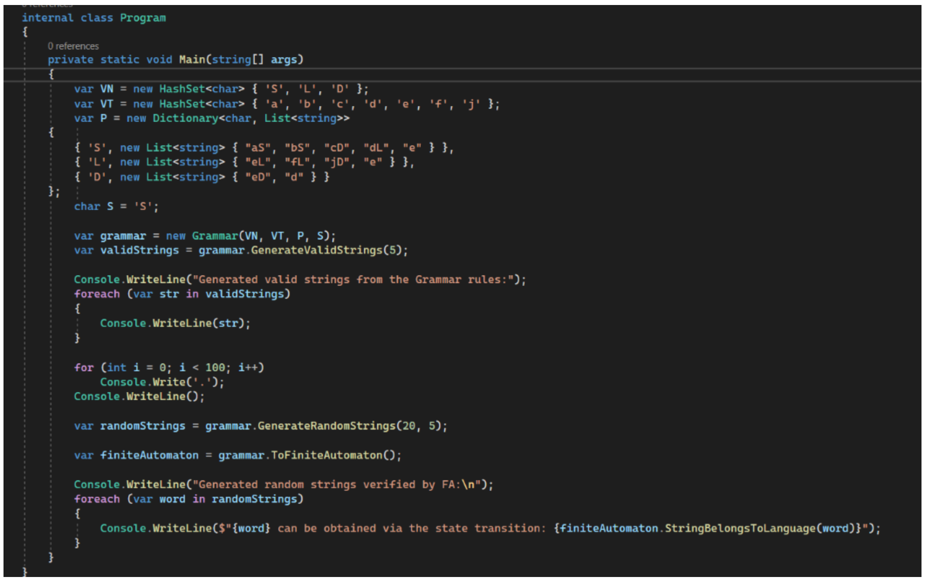


```
1 reference
public bool StringBelongsToLanguage(string w)
{
    var currentStates = new HashSet<string>(q0);
    foreach (var letter in w)
    {
        var nextStates = new HashSet<string>();
        foreach (var state in currentStates)
        {
            if (Delta.TryGetValue((state, letter), out var states))
            {
                nextStates.UnionWith(states);
            }
        }
        currentStates = nextStates;
    }
    return currentStates.Any(state => F.Contains(state));
}
```

Figure 6 StringBelongsToLanguage Method

RESULTS

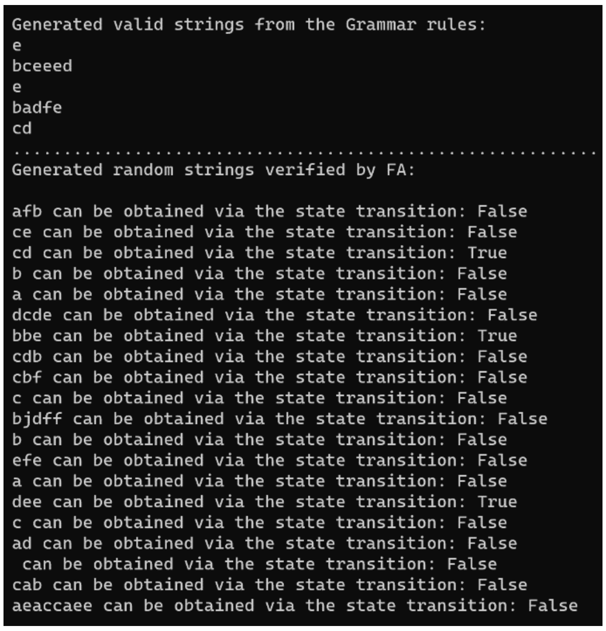
In this code, I defined a formal grammar with specific non-terminals, terminals, and production rules, then used this grammar to generate a set of valid strings. I also created a set of random strings to test the versatility and applicability of the grammar. Finally, I converted the grammar into a finite automaton and used this automaton to determine which of the randomly generated strings adhere to the grammar I defined, demonstrating the practical use of automata theory for string validation.



```
internal class Program
{
    0 references
    private static void Main(string[] args)
    {
        var VN = new HashSet<char> { 'S', 'L', 'D' };
        var VT = new HashSet<char> { 'a', 'b', 'c', 'd', 'e', 'f', 'j' };
        var P = new Dictionary<char, List<string>>
    {
        { 'S', new List<string> { "aS", "bS", "cD", "dL", "e" } },
        { 'L', new List<string> { "eL", "fL", "jD", "e" } },
        { 'D', new List<string> { "eD", "d" } }
    };
        char S = 'S';

        var grammar = new Grammar(VN, VT, P, S);
        var validStrings = grammar.GenerateValidStrings(5);

        Console.WriteLine("Generated valid strings from the Grammar rules:");
        foreach (var str in validStrings)
        {
            Console.WriteLine(str);
        }

        for (int i = 0; i < 100; i++)
            Console.Write('.');
        Console.WriteLine();

        var randomStrings = grammar.GenerateRandomStrings(20, 5);

        var finiteAutomaton = grammar.ToFiniteAutomaton();

        Console.WriteLine("Generated random strings verified by FA:\n");
        foreach (var word in randomStrings)
        {
            Console.WriteLine($"{word} can be obtained via the state transition: {finiteAutomaton.StringBelongsToLanguage(word)}");
        }
    }
}
```

Figure 7 Program class

SCREENSHOTS



```
Generated valid strings from the Grammar rules:
e
bceeed
e
badfe
cd
.........................................................
Generated random strings verified by FA:

afb can be obtained via the state transition: False
ce can be obtained via the state transition: False
cd can be obtained via the state transition: True
b can be obtained via the state transition: False
a can be obtained via the state transition: False
dcde can be obtained via the state transition: False
bbe can be obtained via the state transition: True
cdb can be obtained via the state transition: False
cbf can be obtained via the state transition: False
c can be obtained via the state transition: False
bjdff can be obtained via the state transition: False
b can be obtained via the state transition: False
efe can be obtained via the state transition: False
a can be obtained via the state transition: False
dee can be obtained via the state transition: True
c can be obtained via the state transition: False
ad can be obtained via the state transition: False
 can be obtained via the state transition: False
cab can be obtained via the state transition: False
aeaccaee can be obtained via the state transition: False
```

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.