MINISTERUL EDUCAŢIEI, CULTURII ŞI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Departamentul Inginerie Software și Automatică

CRUDU ALEXANDRA

FAF-233

Report

*Laboratory work №***1**

of Formal Languages Finite Automata

Checked by:

Cretu D. *, university assistant*

DISA, FCIM, UTM

Chișinău – 2025

Purpose of the laboratory work

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 following:

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)

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;

Technical implementation

```
import Foundation

class Grammar {
    private let nonTerminals: Set<Character>
    private let terminals: Set<Character>
    private let rules: [Character: [String]]
    private let startSymbol: Character

    init(nonTerminals: Set<Character>,
         terminals: Set<Character>,
         rules: [Character: [String]],
         startSymbol: Character) {
        self.nonTerminals = nonTerminals
```

```
        self.terminals = terminals
        self.rules = rules
        self.startSymbol = startSymbol
    }


    private func generateString(symbol: Character,
      length: Int, maxLength: Int) -> String {
        if length > maxLength {
            return ""
        }

        // If it's a terminal, just return it as a
          string.
        if terminals.contains(symbol) {
            return String(symbol)
        }

        // If there's no production for this symbol,
            return empty.
        guard let productions = rules[symbol], !
          productions.isEmpty else {
            return ""
        }

        // Randomly pick a production from the
          available ones.
        let chosenProduction = productions.
          randomElement() ?? ""

        var result = ""
        for ch in chosenProduction {
            result += generateString(symbol: ch,
              length: length + 1, maxLength:
              maxLength)
        }
        return result
```

```
}

/// Generates a specified number of strings from
   the start symbol.
func generateStrings(count: Int) -> [String] {
    var generatedStrings = [String]()
    for _ in 0..<count {
        let str = generateString(symbol:
          startSymbol, length: 0, maxLength: 15)
        generatedStrings.append(str)
    }
    return generatedStrings
}


func convertToFA() -> FiniteAutomaton {
    var transitions: [Character: [Character:
      Character]] = [:]

    // States: all non-terminals + a dead state
      (X)
    var states = nonTerminals
    let deadState: Character = "X"
    states.insert(deadState)

    let initialState = startSymbol
    var acceptStates = Set<Character>()

    // Initialize the transition maps
    for nt in nonTerminals {
        transitions[nt] = [:]
    }
    transitions[deadState] = [:]

    // Build transitions from the grammar rules
    for (nt, prods) in rules {
        for production in prods {
```

```
                // If production is a single
                   terminal, transition to deadState
                   and mark it as accepting.
                if production.count == 1 {
                    if let terminal = production.
                       first {
                        transitions[nt]?[terminal] =
                            deadState
                        acceptStates.insert(
                           deadState)
                    }
                } else if production.count >= 2 {
                    // If production is (terminal +
                       nonTerminal)
                    let terminal = production.first!
                    let nextState = production.
                       dropFirst().first! // next
                       symbol
                    transitions[nt]?[terminal] =
                       nextState
                }
            }
        }

        return FiniteAutomaton(states: states,
                               alphabet: terminals,
                               transitions:
                                  transitions,
                               initialState:
                                  initialState,
                               acceptStates:
                                  acceptStates)
    }
}

class FiniteAutomaton {
    private let states: Set<Character>
```

```
private let alphabet: Set<Character>
private let transitions: [Character: [Character:
    Character]]
private let initialState: Character
private let acceptStates: Set<Character>

init(states: Set<Character>,
     alphabet: Set<Character>,
     transitions: [Character: [Character:
       Character]],
     initialState: Character,
     acceptStates: Set<Character>) {
    self.states = states
    self.alphabet = alphabet
    self.transitions = transitions
    self.initialState = initialState
    self.acceptStates = acceptStates
}

/// Checks whether the given input string is
  accepted by this FA.
func accepts(input: String) -> Bool {
    var currentState = initialState

    for symbol in input {
        // If the symbol is not in the alphabet
          or there's no valid transition, reject
          .
        guard alphabet.contains(symbol),
              let stateTransitions = transitions
                [currentState],
              let nextState = stateTransitions[
                symbol]
        else {
            return false
        }
        currentState = nextState
```

```
        }

        // Check if the last state is an accept
           state.
        return acceptStates.contains(currentState)
    }
}

// -------------------------
// MARK: - Main execution
// -------------------------


func main() {


    // -------------------------
    // Variant 10 grammar:
    //
    //   VN = { S, B, L }
    //   VT = { a, b, c }
    //   P = {
    //       S -> aB
    //       B -> bB
    //       B -> cL
    //       L -> cL
    //       L -> aS
    //       L -> b
    //   }
    //
    let nonTerminals: Set<Character> = ["S", "B", "L
      "]
    let terminals: Set<Character> = ["a", "b", "c"]
    let rules: [Character: [String]] = [
        "S": ["aB"],
        "B": ["bB", "cL"],
        "L": ["cL", "aS", "b"]
```

```
    ]

    // Create the grammar with start symbol 'S'
    let grammar = Grammar(nonTerminals: nonTerminals
      ,
                          terminals: terminals,
                          rules: rules,
                          startSymbol: "S")

    // Generate some random strings
    let generatedStrings = grammar.generateStrings(
      count: 5)
    print("Generated Strings:")
    for str in generatedStrings {
        print(str)
    }

    // Convert to a Finite Automaton
    let fa = grammar.convertToFA()

    // Test the FA with some strings
    print("\nTesting with sample strings:")
    let testStrings = [
        "ab",
        "abb",
        "acb",
        "b",
        "abc"
    ]
    for testStr in testStrings {
        print("String: \(testStr) - Accepted: \(fa.
          accepts(input: testStr))")
    }
}


main()
```

Results

```
Generated Strings:
acb
abccaacb
abcaacaacaabccb
acb
acccb

Testing with sample strings:
String: ab - Accepted: false
String: abb - Accepted: false
String: acb - Accepted: true
String: b - Accepted: false
String: abc - Accepted: false
Program ended with exit code: 0
```

Figure 1: Results

Result Description

Result Description The implemented Swift program successfully demonstrates the key concepts of formal languages and finite automata by achieving the following:

0.1 Grammar Implementation String Generation:

The program defines a regular grammar (Variant 10) with non-terminals S, B, L, terminals a, b, c, and production rules:

1. S $\rightarrow$ aB
2. B $\rightarrow$ bB — cL
3. L $\rightarrow$ cL — aS — b

Using a recursive string generation algorithm, the code produces 5 random valid strings derived from this grammar. This confirms that the grammar correctly generates strings according to its production rules.

0.2 Finite Automaton Conversion Testing:

The grammar is converted into a finite automaton where non-terminals represent states, and the production rules define the transition function. The FA includes an additional dead state for productions that end in a terminal. Sample strings are then tested against the FA to verify their acceptance. The testing results illustrate that the FA correctly recognizes strings that belong to the language defined by the grammar.

Conclusion

This laboratory work effectively bridges theoretical concepts with practical application. By implementing both a grammar and its corresponding finite automaton in Swift, the project demonstrates how formal languages can be structured and analyzed computationally. The successful generation of valid strings and the accurate determination of string membership via the FA validate the approach and confirm that the underlying principles of regular grammars and finite automata are correctly applied. This exercise lays a solid foundation for further exploration in formal language theory and automata, reinforcing the importance of these concepts in computer science and related fields.

Bibliography

1. lfa: https://github.com/crudualexandra/LFA$_L ab.git$