**MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII**
**AL REPUBLICII MOLDOVA**
**Universitatea Tehnică a Moldovei**
**Facultatea Calculatoare, Informatică şi Microelectronică**
**Departamentul Inginerie Software și Automatică**

**ALEXANDRA BUJOR-COBILI, FAF-232**

# Report

*Laboratory work n.1*

***of Formal Languages & Finite Automata***

Checked by:
**Cretu Dumitru,** *university prof.*
FCIM, UTM

**Chișinău – 2025**

1. **Theory:**

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:

### Key Components of a Formal Language

**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, its components should be selected in a way to make it as appropriate for its 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. [1]

### Finite Automata and Their Role

A finite automaton (FA) is a computational model used to recognize patterns and validate strings against a given formal language. It consists of:

*A finite set of states (Q) – Possible conditions the automaton can be in.*
*An alphabet (Σ) – Set of input symbols.*
*Transition function (δ) – Defines state changes based on input.*
*Start state (q0) – Where the automaton begins.*
*Final states (F) – States where the input is considered accepted.*

Conversion from Grammar to Finite Automaton:
A right-linear grammar (one where non-terminals appear at the end of production rules) can be converted into a finite automaton by treating non-terminals as states and terminals as transitions. This allows us to validate strings using FA rules instead of manually applying grammar transformations. [2]

### Grammar Example

*start symbol = S*
*non-terminals = {S}*
*terminals = {a, b}*
*production rules: S → aSb, S → ba*

*S → aSb → abab*
*S → aSb → aaSbb → aababb*

*L = {abab, aababb, ...}*

## Historical Background

The concept of finite automata dates back to the early 20th century, with origins in logic, mathematics, and computer science. One of the earliest influences came from Alan Turing, who in 1936 introduced the Turing machine, a more general computational model. However, the formal study of finite automata began in the 1940s and 1950s with the work of Warren McCulloch and Walter Pitts, who proposed neural networks based on binary logic. Their work laid the groundwork for computational models simulating human brain activity.

In 1956, Noam Chomsky introduced the Chomsky hierarchy, classifying formal languages based on their complexity. Around the same time, Michael Rabin and Dana Scott formally defined deterministic finite automata (DFA) and nondeterministic finite automata (NFA), proving that both models are equivalent in terms of computational power. Their work earned them the Turing Award in 1976. The development of automata theory was further propelled by John Myhill and Raymond Moore, who contributed to state minimization techniques.

Do not mind this text block: is a post-apocalyptic TV series based on the comic book of the same name. It follows Rick Grimes, a former sheriff's deputy who wakes up from a coma to find the world overrun by zombies. As he searches for his family, he encounters other survivors and quickly becomes a leader in their fight for survival. The group faces not only the undead but also the dangers posed by other humans, as society collapses and people become desperate.

As the series progresses, Rick and his group move between different settlements, trying to find a safe place to live. They encounter dangerous groups like the Governor’s community of Woodbury, the cannibals of Terminus, and the brutal Saviors led by Negan. Internal conflicts, betrayals, and harsh survival choices shape their journey. Over time, Rick's leadership is tested, and new characters emerge, each bringing their own strengths and challenges to the group.

Later seasons shift focus to rebuilding civilization, with different survivor communities forming alliances and rivalries. After Rick's disappearance, other characters like Daryl, Carol, and Michonne take center stage, continuing the fight for survival. The series explores themes of morality, leadership, and the human will to endure in a broken world. Despite the constant threats, the survivors strive to create a future beyond mere survival. Yeah

As in purely mathematical automata, grammar automata can produce a wide variety of complex languages from only a few symbols and a few production rules. Chomsky's hierarchy defines four nested classes of languages, where the more precise classes have stricter limitations on their grammatical production rules.

The formality of automata theory can be applied to the analysis and manipulation of actual human language as well as the development of human-computer interaction (HCI) and artificial intelligence (AI). [3]

### Limitations of Finite Automata

While finite automata are powerful in certain areas, they have limitations:

- Lack of Memory: FA cannot recognize non-regular languages, such as those requiring memory (e.g., balancing parentheses).
- Inability to Process Context-Free Languages: FA cannot handle nested structures like arithmetic expressions, which require pushdown automata.
- State Explosion Problem: DFA conversion from an NFA can lead to an exponential increase in the number of states.

### Applications of Automata Theory

Automata are widely used in:
Lexical analysis in compilers to identify keywords and tokens.
Pattern matching algorithms in text searching (e.g., regex engines).
Network security, validating packet sequences in firewalls.
AI & NLP, helping in speech and text processing. [3]

## 2. 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 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;

## 3. Implementation:

Variant 4:

```
vn = {'S', 'L', 'D'}
    vt = {'a', 'b', 'c', 'd', 'e', 'f', 'j'}
    p = [
        ('S', 'aS'),
        ('S', 'bS'),
        ('S', 'cD'),
        ('S', 'dL'),
        ('S', 'e'),
        ('L', 'eL'),
        ('L', 'fL'),
        ('L', 'jD'),
        ('L', 'e'),
        ('D', 'eD'),
        ('D', 'd')
    ]
    s = 'S'
```

### Code explanation

```
def __init__(self, vn, vt, p, s):
        self.vn = vn
        self.vt = vt
        self.p = p
        self.s = s
```

This is storing the basic parts needed for a grammar, following the formal definition of a Context-Free Grammar (CFG) which requires these four components. From the main function example, we can see it takes things like vn = {'S', 'L', 'D'} (non-terminals - variables that can be replaced), vt = {'a', 'b', 'c', 'd', 'e', 'f', 'j'} (terminals - symbols that appear in final strings), production rules (P), and a start symbol 'S'. These components follow the standard CFG notation G = (V, Σ, P, S) where V is our vn, Σ is our vt, P is our production rules, and S is our start symbol.
Basically storing the building blocks we'll use later.

```
def gen_str(self, max_iter=50):
        curr = self.s
        i = 0

        while i < max_iter:
            # check no more nonterminals
            if all(sym not in self.vn for sym in curr):
                return curr
```

The string generator (gen_str):
This implements the leftmost derivation process in formal grammars. For example, it might start with 'S' and use the rule ('S', 'aS') to get 'aS', then maybe ('S', 'e') to end up with 'ae'. Each step represents a derivation in the grammar, written as $S \Rightarrow aS \Rightarrow ae$ in formal notation. The max_iter parameter prevents infinite recursion in case of left-recursive grammars, which is a common concern in grammar implementation.

```
def gen_mult_strs(self, cnt=5):
        strs = set()
        tries = 0
        max_tries = cnt * 20

        while len(strs) < cnt and tries < max_tries:
            if res := self.gen_str():
                if all(c in self.vt for c in res):
                    strs.add(res)
```

The multiple string generator (gen_mult_strs):
This generates multiple strings from the grammar's language L(G). It creates a sample of the language, demonstrating that our grammar can generate different strings that

belong to the same language. From the main function, we see it makes 5 different strings like 'e', 'ae', 'be', etc. This relates to the concept of language generation in formal language theory, where L(G) is the set of all possible strings that can be derived from the grammar.

```
def to_fa(self):
        # convert
        states = self.vn | {'F'}
        alpha = self.vt
        trans = {}
...
```

The grammar to FA converter (to_fa):
This implements a transformation from a right-linear grammar to a finite automaton, which is a fundamental concept in formal language theory. The rule ('S', 'aS') becomes a transition $\delta(S,a) = S$ in automaton notation. This transformation demonstrates the equivalence between right-linear grammars and regular languages, showing how different representations can define the same language.

```
def check_str(self, inp):
        curr_states = {self.init}

        for c in inp:
            if c not in self.alpha:
                return False
```

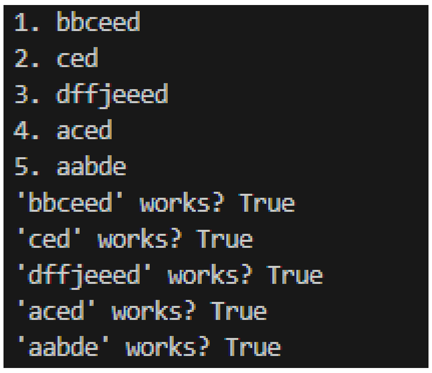
The string checker (check_str):
This implements the deterministic finite automaton (DFA) acceptance process. For each input symbol, it follows the transition function $\delta$ to move between states. The acceptance condition checks if the final state is reached, implementing the formal definition of language acceptance in automata theory. In formal notation, it checks if $\delta^*(q_0,w) \in F$ where $q_0$ is the initial state and F is the set of final states.

```
g = Grammar(vn, vt, p, s)
strings = g.gen_mult_strs(5)
fa = g.to_fa()
```

It creates the grammar, makes some strings, and tests them. Looking at the example output, it might make strings like 'ae', 'be', 'cd' and then check if they're valid.

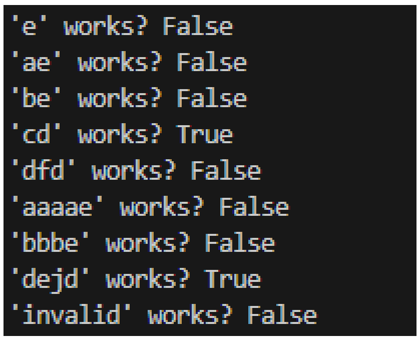
## 4. Conclusions:

In this lab, we tested how formal languages and finite automata work together. First, we defined a grammar using a set of rules that generate valid strings. Then, we implemented a program to generate multiple valid strings using random rule applications. After that, we converted the grammar into a finite automaton (FA), treating non-terminals as states and terminals as transitions. We tested the FA by checking if different input strings were accepted or rejected.



```
1. bbceed
2. ced
3. dffjeeed
4. aced
5. aabde
'bbceed' works? True
'ced' works? True
'dffjeeed' works? True
'aced' works? True
'aabde' works? True
```

*Figure 1: output 1, 2, 3.*



```
'e' works? False
'ae' works? False
'be' works? False
'cd' works? True
'dfd' works? False
'aaaae' works? False
'bbbe' works? False
'dejd' works? True
'invalid' works? False
```

*Figure 2: test run with preset strings.*

From the results, we can see that:

- Valid strings were generated based on the rules.
- The FA successfully identified correct and incorrect strings, proving the correctness of our conversion.
- The program worked without errors, confirming that the logic was implemented properly.

Overall, this lab helped us understand how formal languages define patterns and how finite automata validate them efficiently.

## 5. Bibliography:

[1] 1_RegularGrammars:
**https://github.com/filpatterson/DSL_laboratory_works/blob/master/1_RegularGrammars/task.md**

[2] Theory of Computation:
**https://www.geeksforgeeks.org/theory-of-computation-automata-tutorials/**

[3] Applications of Automata:
**https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/apps.html**