

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

**Ciumacenco Pavel FAF-232**

**Report**

*Laboratory work nr.4*

***Course: Formal Languages & Finite Automata***

Checked by:

**Cretu Dumitru,** *university prof.*

FCIM, UTM

**Chișinău – 2025**

**Theory:**

### Introduction to Regular Expression

A **regular expression** (commonly abbreviated as **regex** or **regexp**) is a sequence of characters that defines a search pattern. These patterns are primarily used for string matching and manipulation operations, such as finding, extracting, validating, or replacing substrings within a larger text. Regular expressions provide a powerful, flexible, and concise way to process and analyze textual data, and they are supported across many programming languages including Python, JavaScript, Java, C#, Perl, and more.

The concept of regular expressions originates from formal language theory and automata, forming the foundation for defining **regular languages** — one of the simplest classes of languages in the **Chomsky hierarchy**. A regular expression is a declarative way to express finite automata behavior, and as such, it plays an essential role in compiler design, syntax validation, tokenization, lexical analysis, and data parsing.

### Syntax and Components of Regular Expressions

Regular expressions consist of **literal characters** and **special symbols** that allow pattern generalization. Below is a breakdown of some key components of regex syntax:

#### a. Literal Characters

Literal characters match exactly what they represent:

* "a" matches the character 'a'.
* "dog" matches the word 'dog'.

#### b. Metacharacters

These are special symbols used to control how regex behaves:

* . — Matches any character except a newline.
* ^ — Anchors the match at the beginning of a line.
* $ — Anchors the match at the end of a line.
* \ — Escapes a metacharacter (e.g., \. matches a literal period).

#### c. Character Classes

* [abc] — Matches 'a', 'b', or 'c'.
* [a-z] — Matches any lowercase letter from 'a' to 'z'.
* [^0-9] — Matches any character *except* digits.

#### d. Quantifiers

* \* — Matches 0 or more repetitions.
* + — Matches 1 or more repetitions.
* ? — Matches 0 or 1 occurrence (optional).
* {n} — Matches exactly n times.
* {n,} — Matches n or more times.
* {n,m} — Matches between n and m times.

#### e. Grouping and Alternation

* (abc) — Groups characters together.
* (a|b|c) — Matches one of 'a', 'b', or 'c'.

### Applications of Regular Expressions

Regular expressions are extensively used in software development, system administration, and data processing tasks. Some common real-world applications include:

* **Form Validation**: Ensuring user input matches a required pattern, such as email addresses or phone numbers.
* **Search and Replace**: Efficiently locating and modifying strings in documents, codebases, or logs.
* **Lexical Analysis**: Tokenizing programming code into meaningful elements (tokens).
* **Log Parsing**: Extracting structured data from log files.
* **Data Scraping**: Retrieving information from web pages and documents.
* **Security Audits**: Detecting sensitive patterns like API keys, IP addresses, or credentials.

### Implementation in this project

In this project, we built a **dynamic regular expression evaluator and generator**. Rather than hardcoding outputs for specific patterns, the goal was to build a **recursive interpreter** that can process any valid regular expression structure similar to those found in Variant 1 of the task image.

#### The Main Features Include:

* **Support for core regex elements**: including groups (), alternation |, quantifiers \*, +, ?.
* **Controlled repetition**: To prevent infinite output from constructs like \* or +, a repeat cap of 5 was introduced.
* **Randomized generation**: Each pattern element has multiple valid outputs; the program chooses randomly for variety.
* **Traceable steps**: An optional debug mode logs step-by-step processing of the regex, showing which branches were chosen and how characters were repeated.

### Challenges and Considerations

While the regex syntax is powerful, implementing a parser and evaluator from scratch presents challenges:

* **Recursive Group Handling**: Nested groups (e.g., ((a|b)c)+) require careful tracking of parentheses and scopes.
* **Greedy vs. Lazy Quantifiers**: Standard regex engines can be greedy (matching as much as possible). Our generator needed to balance randomness and constraint.
* **Ambiguity in Generation**: Unlike matching, where regex answers "does this fit?", generation answers "what fits this?"—which has many correct answers. Ensuring variety and correctness simultaneously is non-trivial.
* **Infinite Expansions**: Without repeat limits, quantifiers like \* can lead to very long or infinite outputs.

**Objectives:**

1. Write and cover what regular expressions are, what they are used for;
2. Below you will find 3 complex regular expressions per each variant. Take a variant depending on your number in the list of students and do the following:  
   a. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown). Be careful that idea is to interpret the given regular expressions dinamycally, not to hardcode the way it will generate valid strings. You give a set of regexes as input and get valid word as an output  
   b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);  
   c. Bonus point: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

Write a good report covering all performed actions and faced difficulties.

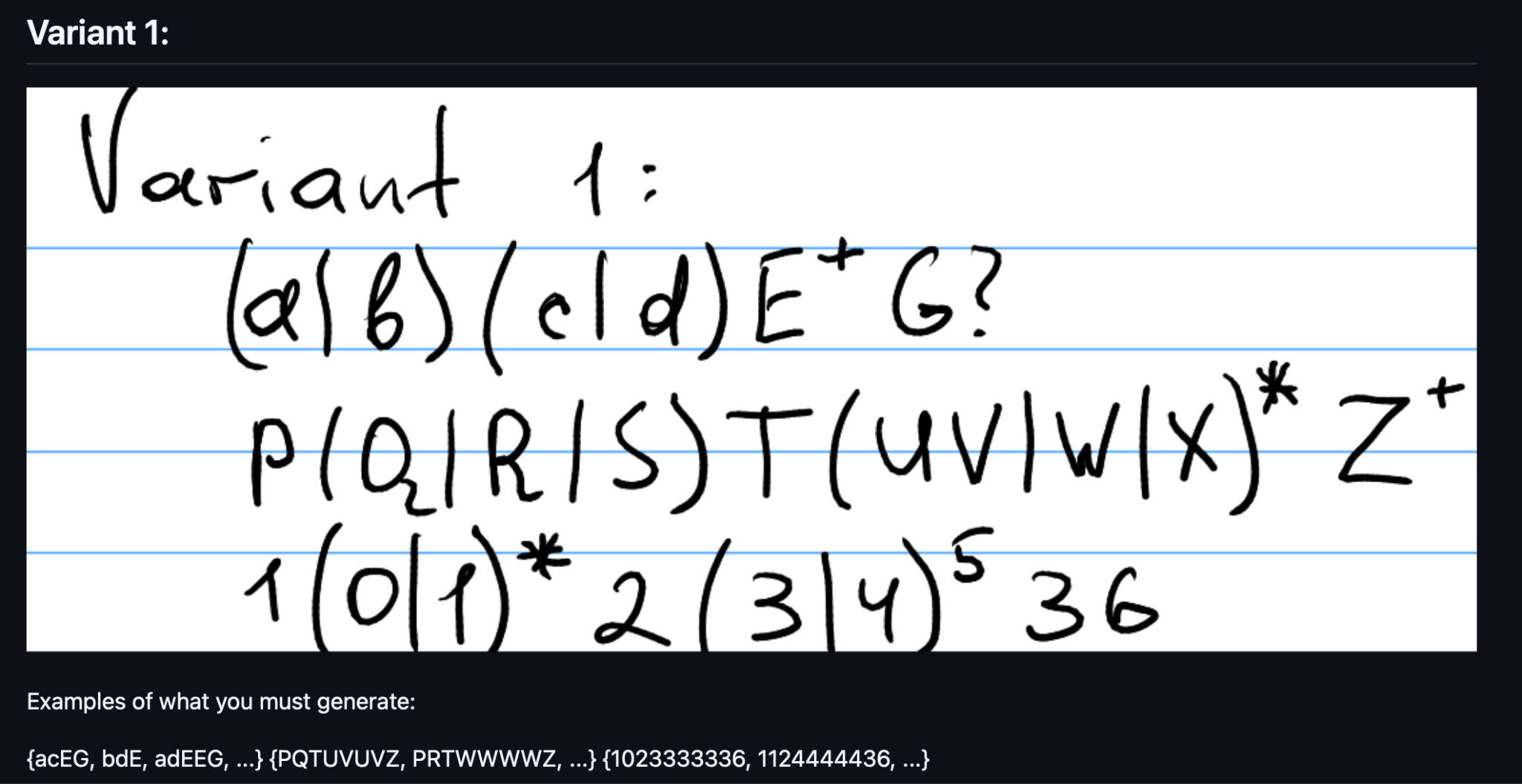


Figure 1: Variant based on list order 9

**Implementation Description:  
  
  
 1. Imports and Config**

**import random**

**import re**

**MAX\_REPEAT = 5**

**Explanation:**We import Python’s random module to make random choices and repetitions, and re (though unused now) is standard for regex operations.  
MAX\_REPEAT is a safety limit for \* and + so the generator doesn’t create infinitely long outputs.

**def generate\_from\_regex(regex, trace=False):**

**pos = 0**

**trace\_steps = []**

**def log(step):**

**if trace:**

**trace\_steps.append(step)**

**Explanation:**This defines the main function that takes a regex string and a flag to enable tracing.  
We use pos to track the current index in the regex, and trace\_steps collects log messages if tracing is on.

**def parse\_expression():**

**nonlocal pos**

**result = ""**

**...**

**Explanation:**parse\_expression is the core recursive function that builds the string by reading characters from left to right.  
It handles different regex elements like groups, quantifiers (\*, +, ?), and simple characters.

**while pos < len(regex):**

**char = regex[pos]**

**if char == '(':**

**...**

**elif char == ')':**

**...**

**elif char == '|':**

**...**

**elif char in "\*+?":**

**...**

**else:**

**result += char**

**pos += 1**

**Explanation:**This loop reads the regex one character at a time.  
It checks for special symbols like (, |, \*, +, and ?, and acts accordingly — building the final string in result.

**elif char in "\*+?":**

**prev = result[-1]**

**...**

**Explanation:**When a quantifier is found, it applies to the previous character in result.  
We randomly decide how many times to repeat (or whether to include) that character and update the result string.

**def parse\_group():**

**nonlocal pos**

**options = []**

**group = ""**

**...**

**chosen = random.choice(options)**

**log(f"Group ({'|'.join(options)}) → '{chosen}'")**

**return chosen**

**Explanation:**This function handles parsing of grouped expressions with alternation, like (a|b|c).  
It splits the content inside () by |, randomly picks one option, and returns it.

**final\_result = parse\_expression()**

**return (final\_result, trace\_steps) if trace else final\_result**

**Explanation:**We execute the main parsing and return either just the final string, or the string plus the trace log.  
This lets the caller choose between a clean or debug-heavy output.

**regexes = [**

**"(a|b)(c|d)E+G?",**

**"P(Q|R|S)T(uv|w|x)\*Z+",**

**"1(0|1)\*2(3|4)5"**

**]**

**for r in regexes:**

**result, steps = generate\_from\_regex(r, trace=True)**

**...**

**Explanation:**We provide sample regexes from the image and call our generator on each.  
If tracing is on, we print the steps explaining how the result was built.

**4.Conclusions.Screenshots.Results.**

Understanding and implementing regular expression evaluation from scratch is more than an exercise in string processing — it provides insight into the **foundations of computer science**. From lexical analyzers in compilers to state machines and natural language processing, regular expressions are fundamental.

By implementing a simplified regex engine with generation capabilities, we also gain a deeper appreciation of:

* Language recognizers vs. language generators
* Determinism and randomness in parsing
* The balance between strict syntax and flexible application

Regular expressions are an indispensable tool in both theoretical and applied computer science. Their concise syntax and immense expressive power make them a go-to solution for numerous text-based tasks. Through this project, we not only explored the syntax and application of regex but also constructed a dynamic generator that interprets regex and produces valid strings accordingly — opening doors to further research in test generation, grammar simulation, and interactive compilers.

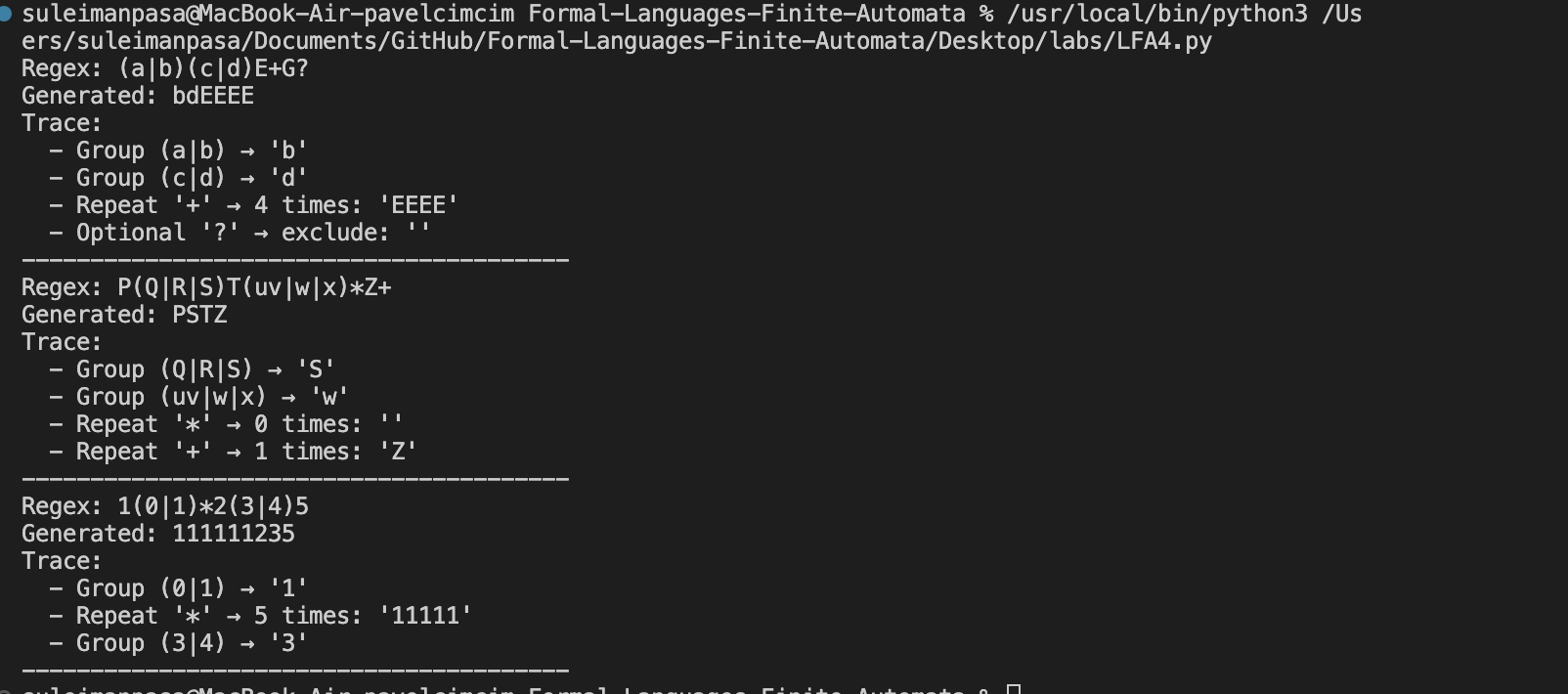


Figure 10: Results Picture