Ministry of Education, Culture and Research of the Republic of Moldova
Technical University of Moldova
Department of Software and Automation Engineering

REPORT

Laboratory work *No. 4*

Formal Languages & Finite Automata

Elaborated by:
Gorcea Alexandrina, FAF-223
Verified by: Dumitru Crețu, univ. assist.

Chișinău, 2024

Theoretical considerations:

Regular expressions, fundamental in formal language theory, offer expressive pattern-matching capabilities in computing. Their concise syntax enables the definition of intricate patterns, vital for tasks like input validation and text parsing. Varying in syntax among programming languages, regexes may exhibit differences in efficiency and supported features. Understanding greedy versus lazy matching, character classes, and anchors ensures effective pattern definition.

Despite their power, regexes require cautious handling due to potential inefficiencies and debugging complexities. Balancing expressiveness with performance considerations, regexes remain indispensable tools for string manipulation, embodying the intersection of theoretical rigor and practical utility in computer science.

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).

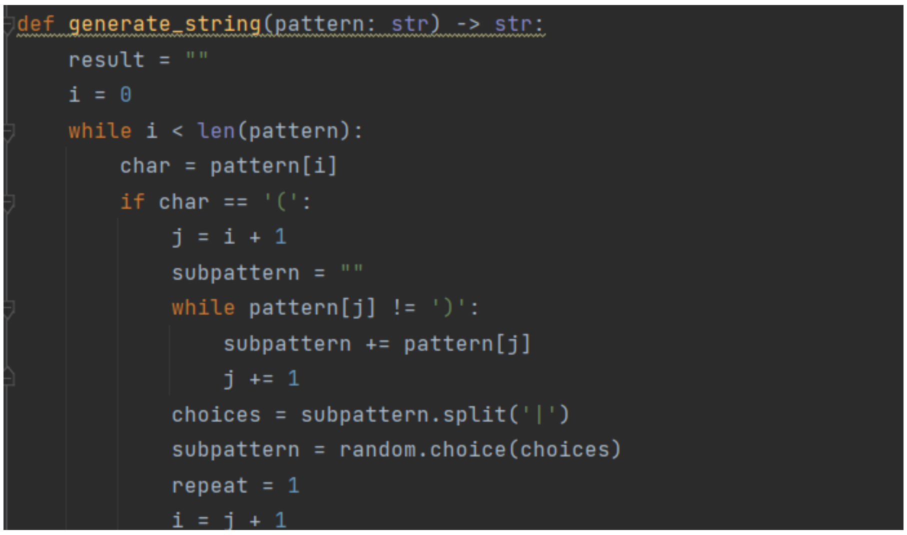
b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

c. **Bonus point**: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

Implementation description:

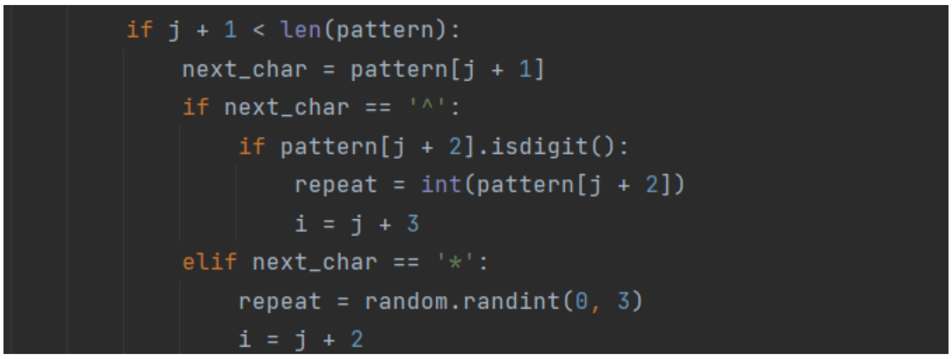
My code provides a flexible solution for generating random strings based on predefined patterns using regular expressions. The script comprises a `generate_string` function responsible for parsing patterns and generating strings accordingly. Through iterative traversal of the input pattern, the function intelligently handles special characters such as '|' for alternation, '*' for arbitrary repetitions, and '^' for fixed repetitions. Random choices are made for alternatives, and random repetitions are determined within specified ranges, ensuring variability in generated strings. The `main` function serves as a demonstration platform, showcasing the script's capability to generate diverse and randomized string outputs for a variety of example patterns. Further analysis and elaboration on the script's inner workings, performance considerations, and practical applications will be provided in subsequent sections of the report.

The following code defines a function generate_string that creates a string based on a given pattern. It iterates through the pattern, handling parentheses by randomly choosing one of their contents. It tracks the position in the pattern, accumulating characters until it finds a closing parenthesis. Then, it selects a random choice from the options inside the parentheses. The process continues until the end of the pattern. This mechanism allows for dynamic generation of strings based on variable patterns, essential for generating diverse outputs.



```python
def generate_string(pattern: str) -> str:
    result = ""
    i = 0
    while i < len(pattern):
        char = pattern[i]
        if char == '(':
            j = i + 1
            subpattern = ""
            while pattern[j] != ')':
                subpattern += pattern[j]
                j += 1
            choices = subpattern.split('|')
            subpattern = random.choice(choices)
            repeat = 1
            i = j + 1
```

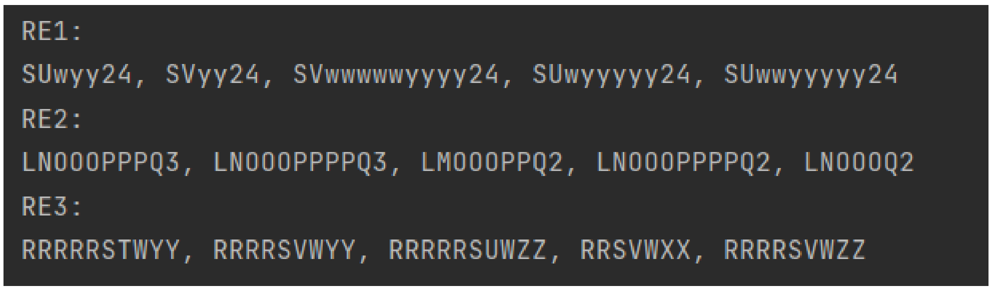
The next section of the code handles special characters '*', '+', '?', and '^' that appear directly after a character in the pattern sequence. It ensures proper handling of these characters to determine the repetition behavior of the preceding character.



```python
            if j + 1 < len(pattern):
                next_char = pattern[j + 1]
                if next_char == '^':
                    if pattern[j + 2].isdigit():
                        repeat = int(pattern[j + 2])
                        i = j + 3
                elif next_char == '*':
                    repeat = random.randint(0, 3)
                    i = j + 2
```

At the end, we have the `main` function, which orchestrates the execution of the script, demonstrating the `generate_string` function's functionality by generating and printing random strings based on example patterns. By iterating through the example patterns and generating random strings, the `main` function showcases the versatility and utility of the implemented string generation mechanism.

Results (for variant 4):



```
RE1:
SUwyy24, SVyy24, SVwwwwwyyyy24, SUwyyyyy24, SUwwyyyyy24
RE2:
LNOOOPPPQ3, LNOOOPPPPQ3, LMOOOPPQ2, LNOOOPPPPQ2, LNOOOQ2
RE3:
RRRRRSTWYY, RRRRSVWYY, RRRRRSUWZZ, RRSVWXX, RRRRSVWZZ
```

Conclusion:

In conclusion, the implemented Python script offers a robust solution for generating random strings based on user-defined patterns using regular expressions. Through careful parsing and handling of special characters, the script facilitates the creation of diverse and dynamic string outputs.

By leveraging the power of regular expressions, it provides a flexible approach to pattern matching and string manipulation tasks. The script's modular structure allows for easy integration into various applications requiring randomized string generation, from data generation to testing scenarios. With further enhancements and optimizations, such as additional special character support and performance improvements, the script could serve as a valuable tool for developers across different domains.

Overall, it underscores the importance of regular expressions in computational tasks and highlights their effectiveness in addressing diverse string generation requirements.