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 patternmatching 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)

Regular Expression *Variant 4* Explanation:

- (S|T): Matches either 'S' or 'T'.
- (U|V): Matches either 'U' or 'V'.
- w*: Matches zero or more occurrences of the character 'w'.
- y+: Matches one or more occurrences of the character 'y'.
- 24: Matches the literal string '24'.
- L: Matches the literal character 'L'.
- (M|N): Matches either 'M' or 'N'.
- O^3: Matches the literal string 'O' followed by exactly three occurrences of the character 'P'.
- P*: Matches zero or more occurrences of the character 'P'.
- Q(2|3): Matches either 'Q2' or 'Q3'.
- R*: Matches zero or more occurrences of the character 'R'.
- S: Matches the literal character 'S'.
- (T|U|V): Matches either 'T', 'U', or 'V'.
- W(X|Y|Z)^2: Matches 'W' followed by exactly two occurrences of either 'X', 'Y', or 'Z'.

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 named generate_string(pattern: str) -> str. This function serves as the core of the string generation process. It parses the input pattern character by character, handling special cases such as parentheses, alternatives, and repetition indicators. For instance, when encountering an opening parenthesis '(', it dynamically selects one of the options within the parentheses separated by '|'. This allows for flexible string generation based on multiple choices defined within the pattern.

Furthermore, the script intelligently handles special characters like '', '+', '?', and '^' to determine repetitions. For example, when '' is encountered after a character, it randomly determines the number of times that character should be repeated, providing variability in the generated strings. Similarly, '^' indicates a specific number of repetitions, offering precise control over the string's structure.

```
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. It begins by ensuring the existence of a character following the current position in the pattern, thus preventing out-of-bounds errors. Subsequently, it identifies the next character and initiates different behaviors based on its value.

When encountering the '^' character, signifying a specific number of repetitions, the script extracts the repetition count if the subsequent character is a digit. This enables precise control over the number of repetitions for the preceding subpattern. Conversely, '*' indicates zero or more repetitions, prompting the script to assign a random repetition count between 0 and 3.

Similarly, the '+' and '?' characters denote one or more repetitions and zero or one repetition, respectively. The script adapts accordingly by assigning random repetition counts within predefined ranges. These adaptations ensure variability in the generated strings, enhancing their dynamism and usefulness in various applications.

Upon determining the repetition count based on the encountered special character, the script appends the subpattern multiplied by the repetition count to the result string. This step consolidates the modifications introduced by the special characters, contributing to the final output string's structure and content. By dynamically adjusting repetition counts based on the presence of '^', '*', '+', and '?' characters, the script enhances the flexibility and variability of the generated strings. This approach underscores the significance of robust character handling mechanisms in string manipulation tasks, facilitating the creation of diverse and tailored string outputs in Python applications.

```
        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
            elif next_char == '+':
                repeat = random.randint(1, 3)
                i = j + 2
            elif next_char == '?':
                repeat = random.randint(0, 1)
                i = j + 2
        result += subpattern * repeat
        continue
```

The section of code from bellow commences by assessing the existence of a character following the current position 'i' in the pattern, a crucial step to prevent index out-of-range errors and ensure the script's robustness. Upon confirming the next character's presence, denoted as 'next_char', the script proceeds to interpret its value and adapt its behavior accordingly.

Each special character triggers specific actions within the script:

- '^' character: Indicates a specific number of repetitions. When detected, the script multiplies the current character by the integer value specified by the character following '^' in the pattern. This feature enables precise control over the repetition count for the character.
- '*' character: Represents zero or more repetitions. In response, the script appends a random number of repetitions (between 0 and 5) of the current character to the result string. This randomness introduces variability into the generated strings, enhancing their dynamism.
- '+' character: Denotes one or more repetitions. The script includes a random number of repetitions (between 1 and 5) of the current character in the result string. This functionality ensures the generation of strings with varying lengths and structures, adding to their versatility.
- '?' character: Signifies zero or one repetition. If encountered, the script includes either 0 or 1 repetition of the current character in the result string randomly. This flexibility allows for the generation of strings with optional elements, accommodating different usage scenarios.

Following the processing of special characters, the script appends the current character to the result string and updates the index 'i' accordingly to proceed with the next character in the pattern. This iterative process continues until the entire pattern is traversed, resulting in the generation of the final string output. By intelligently interpreting special characters like '^', '*', '+', and '?', the script enhances its flexibility and adaptability, enabling the generation of customized strings with diverse structures and content.

```
        next_char = pattern[i + 1] if i + 1 < len(pattern) else ""
        if next_char == '^':
            result += char * int(pattern[i+2])
            i += 3
            continue
        elif next_char == '*':
            result += char * random.randint(0, 5)
            i += 2
            continue
        elif next_char == '+':
            result += char * random.randint(1, 5)
            i += 2
            continue
        elif next_char == '?':
            result += char * random.randint(0, 1)
            i += 2
            continue
        result += char
        i += 1
    return result
```

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.

```
def main():
    pattern1 = "(S|T)(U|V)w*y+24"
    pattern2 = "L(M|N)O^3P*Q(2|3)"
    pattern3 = "R*S(T|U|V)W(X|Y|Z)^2"

    print("RE1:")
    generated_strings = [generate_string(pattern1) for _ in range(5)]
    print(", ".join(generated_strings))

    print("RE2:")
    generated_strings = [generate_string(pattern2) for _ in range(5)]
    print(", ".join(generated_strings))

    print("RE3:")
    generated_strings = [generate_string(pattern3) for _ in range(5)]
    print(", ".join(generated_strings))

if __name__ == "__main__":
    main()
```

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.