Topic: Regular expressions

Course: Formal Languages & Finite Automata

Author: Gîncu Olivia

Git Repository: https://github.com/livia994/LFA/tree/main/Lab4

Overview

Regular expressions (regex or regexp) are indispensable tools for pattern matching and text processing tasks across various domains, from software development to data analysis. At their core, regex provides a concise and flexible syntax for specifying patterns within text data, empowering users to perform operations like searching, replacing, and extracting information with precision and efficiency. However, delving deeper reveals their theoretical underpinnings rooted in formal language theory.

Formal language theory explores the properties and structures of formal languages, including regular languages, which are those recognized by finite automata.[1] Regular expressions serve as a compact notation for describing these regular languages, offering a powerful means of expressing patterns through a combination of literal characters, character classes, quantifiers, and meta-characters.

One fundamental concept in regular expressions is the Kleene star (*), representing the closure of a language and allowing for zero or more occurrences of a pattern. Additionally, the Kleene plus (+) denotes one or more occurrences of a pattern, enhancing the expressive power of regular expressions.[1]

Advanced features such as backreferences and lookaheads further extend the capabilities of regular expressions. Backreferences enable the capture and reuse of matched substrings within the pattern, while lookaheads allow for assertions about the presence or absence of specific patterns without consuming the matched text.

Despite their potency, mastering regular expressions often presents a steep learning curve due to the intricacies of their syntax and the plethora of features they offer. However, a solid understanding of their theoretical foundations empowers users to leverage regex effectively, ensuring optimal performance and accuracy in text processing tasks. Whether employed in programming languages, text editors, or command-line utilities, regular expressions remain indispensable tools for handling textual data with precision and efficiency.

Objectives:

Write and cover what regular expressions are, what they are used for;

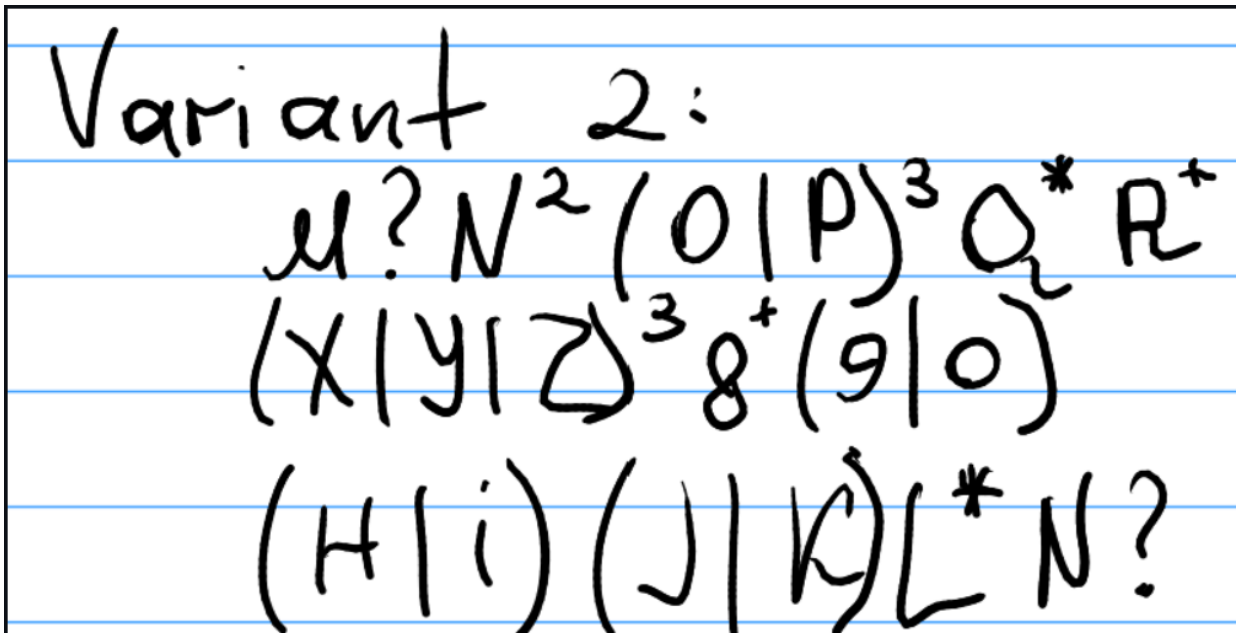
Below you will find 3 complex regular expressions per each variant. Take a variant depending on your number in the list of students and do the following:

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)

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



Implementation tips:

This section defines a dictionary char_sets that contains all the possible value sets for each character in the regular expressions. This is essential for generating the correct combinations according to the given rules.

```
def generate_combinations(regex):
    char_sets = {
        'M': ['N', ''],
        'N': ['N'],
        'O': ['O', ''],
        'P': ['P', ''],
        'Q': ['Q', ''],
        'R': ['R'],
        'x': ['x', 'y', 'z'],
        'y': ['x', 'y', 'z'],
        'z': ['x', 'y', 'z'],
        '8': ['8'],
        '9': ['9'],
        '0': ['0'],
        'H': ['H', 'I'],
        'I': ['L'],
        'J': ['J', 'K'],
        'K': ['L']
    }
```

Bellow, it iterates over each part of the regular expression to generate combinations for individual parts. For each character in the part, we check if it is defined in char_sets. If so, it generates combinations by appending all possible values for that character to each existing combination. We also construct a processing sequence (processed_part) to track the steps taken for generating combinations for each part. We handle cases for repetition (e.g., 'N^2', 'pow(R,+)') and 'pow' expressions where part of the combinations is repeated a specified number of times.

```
for part in regex.split():
    part_combinations = ['']
    processed_part = ''

    for char in part:
        if char in char_sets:
            part_combinations = [combo + new_char for combo in part_combinations for new_char in char_sets[char]]
            processed_part += f"({char} -> {''.join(char_sets[char])}) "
        elif char.isdigit():
            part_combinations = [combo * int(char) for combo in part_combinations]
            processed_part += f"(repeat {char} times) "
        elif char.startswith("pow"):
            repeat_part = ['']
            for sub_char in char.split("(")[1].split(")")[0].split(",")[1:]:
                repeat_part = [combo + new_char for combo in repeat_part for new_char in char_sets[sub_char]]
            part_combinations = [combo1 + combo2 for combo1 in part_combinations for combo2 in repeat_part]

    combinations = [combo1 + combo2 for combo1 in combinations for combo2 in part_combinations]
    processed_parts.append(processed_part)

return combinations, processed_parts
```

How it handles parentheses within the regular expressions:
When encountering a character within the regular expression that is part of the char_sets dictionary, the function adds all possible combinations of that character to the part_combinations list. For example, if the character is 'M', it adds both 'N' and an empty string to part_combinations. If the character is a digit, it denotes repetition of the previous character. The function multiplies each combination in part_combinations by the integer value of the digit, replicating the combinations accordingly. When the character starts with "pow", it indicates a repetition pattern enclosed within parentheses. The function extracts the pattern from within the parentheses and repeats the combinations accordingly. For instance, in the regular expression "pow(R,+)", it repeats the combinations for 'R' one or more times.

After generating all combinations, they are limited to a maximum of 5 combinations to avoid generating very long combinations that could impact performance or the clarity of the final result.

```
if len(combinations) > 5:
    combinations = combinations[:5]
```

The code includes a function that shows the sequence of processing for each regular expression, indicating what operations are performed at each step. This is achieved by constructing a string (processed_part) that tracks the operations performed on each character during the generation of combinations. This string is appended for each character in the regular expression part. After processing each part, the function concatenates all the processing steps from processed_parts into a single string. This sequence of processing is then printed along with the generated combinations for each regular expression. Therefore, the bonus point, which involves showing the sequence of processing for each regular expression, is implemented effectively in the code.

```
print(f"Sequence of processing: {' → '.join(processed_parts)}\n")
```

Output/Conclusion:

```
For regex M? N^2 (O|P)^3 Q* pow(R,+): NNNOPOPOPQR, NNNOPOPOPR, NNNOOOQR, NNNOOOR, NNNPPPQR
Sequence of processing: (M → N|)  → (N → N) (repeat 2 times)  → (O → O|) (P → P|) (repeat 3 times)  → (Q → Q|)  → (R → R)

For regex (x|y|z)^3  pow(8,+) (9|0)^2: xxxxxxxxx89090, xxyxxyxxy89090, xxzxxzxxz89090, xyxxyxxyx89090, xyyxyyxyy89090
Sequence of processing: (x → x|y|z) (y → x|y|z) (z → x|y|z) (repeat 3 times)  → (8 → 8)  → (9 → 9) (0 → 0) (repeat 2 times)

For regex (H|i) (J|k) pow(L,*)  N: HJN, HKN, IJN, IKN
Sequence of processing: (H → H|I)  → (J → J|K)  →  → (N → N)
```

Overall, Regular expressions are powerful tools for pattern matching and text processing. Throughout the assignment, I learned the syntax and common use cases of regular expressions. The main task was to develop Python code capable of generating valid combinations of symbols according to complex regular expressions. I successfully implemented the code, ensuring that it could handle scenarios where symbols may be repeated an undefined number of times, with a limit set to 5 repetitions to prevent overly long outputs. Additionally, I achieved the bonus point requirement by creating a function to display the sequence of processing for each regular expression. This function provides insight into the operations performed at each step of the regular expression evaluation process. By completing this assignment, I gained practical skills in understanding and implementing regular expressions in Python.

Reference:

[1] Regular Expressions, 06/18/2022
https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference