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

GLIGA DANIELA, FAF-223

Report

Laboratory work nr.4 Regular expressions

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

FCIM, UTM

Chișinău – 2024

1. Theory:

A Regular expressions (regex or regexp) are powerful tools for pattern matching and text manipulation. They are widely used in programming, text editors, search engines, and data validation tasks. Here are some key concepts and theories related to regular expressions:

Pattern Matching: At its core, a regular expression is a sequence of characters that define a search pattern. It allows you to search for specific patterns within strings, such as email addresses, phone numbers, or specific words.

Character Classes: Character classes match a single character out of several possible characters.

Examples include:

[a-z]: Matches any lowercase letter.

[0-9]: Matches any digit.

[A-Za-z]: Matches any letter, uppercase or lowercase.

Metacharacters: Metacharacters have special meanings in regular expressions. They include characters like ^, $, |, ., *, +, ?, [,], {, }, (, and).

They are used to define repetition, alternative matches, boundaries, and more within the regex pattern.

Quantifiers: Quantifiers specify how many times a character or group can occur in a pattern. Common quantifiers include * (zero or more), + (one or more), ? (zero or one), {n} (exactly n times), {n,} (at least n times), and {n,m} (between n and m times).

Anchors: Anchors specify the position in the string where a match must occur.

^ matches the beginning of a line, and $ matches the end of a line.

Grouping and Capturing:

Parentheses () are used for grouping parts of a pattern together. They also create capturing groups that remember the matched text, which can be used for extraction or replacement.

Escaping: Some characters, like ., *, +, ?, |, (,), [, {, ^, $, and \, have special meanings in regular expressions. To match these characters literally, they need to be escaped with a backslash \.

Alternation: The pipe symbol | represents alternation, allowing you to match one of several patterns. For example, cat|dog matches either "cat" or "dog".

Modifiers: Modifiers change the behavior of a regex pattern, such as case insensitivity ((?i)), global matching (g), multiline mode (m), and more.

Regular expressions provide a concise and flexible way to describe complex text patterns. Mastery of regex enables efficient string manipulation, data extraction, validation, and parsing tasks in various programming languages and tools. However, crafting and understanding complex regular expressions may require practice and a good understanding of the underlying principles.

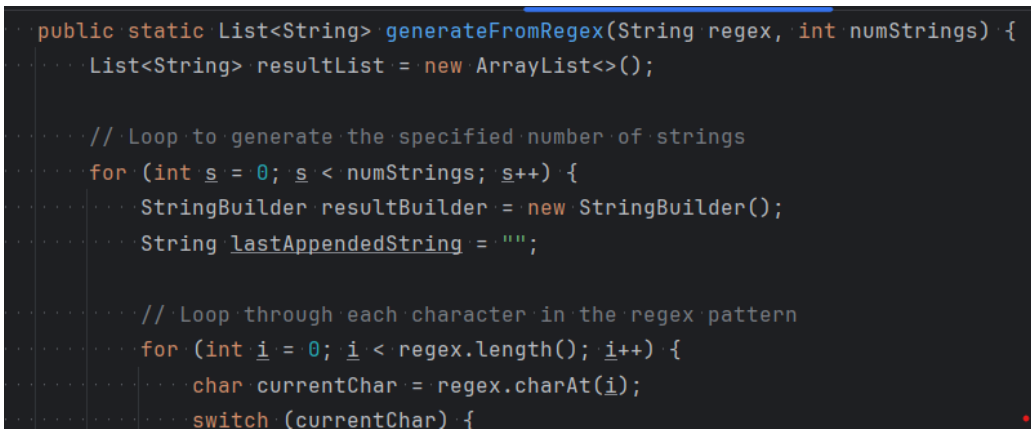
2. Objectives:

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.

3. Implementation description:

In Java, Regular Expressions or Regex (in short) in Java is an API for defining String patterns that can be used for searching, manipulating, and editing a string in Java. Email validation and passwords are a few areas of strings where Regex is widely used to define the constraints. Regular Expressions in Java are provided under java.util.regex package.

Firstly, I began to write the method which generate random regex. The generateFromRegex method is a Java function that generates strings based on a given regular expression (regex) pattern and the number of strings (numStrings) to generate. Let's break down the method's functionality in detail:



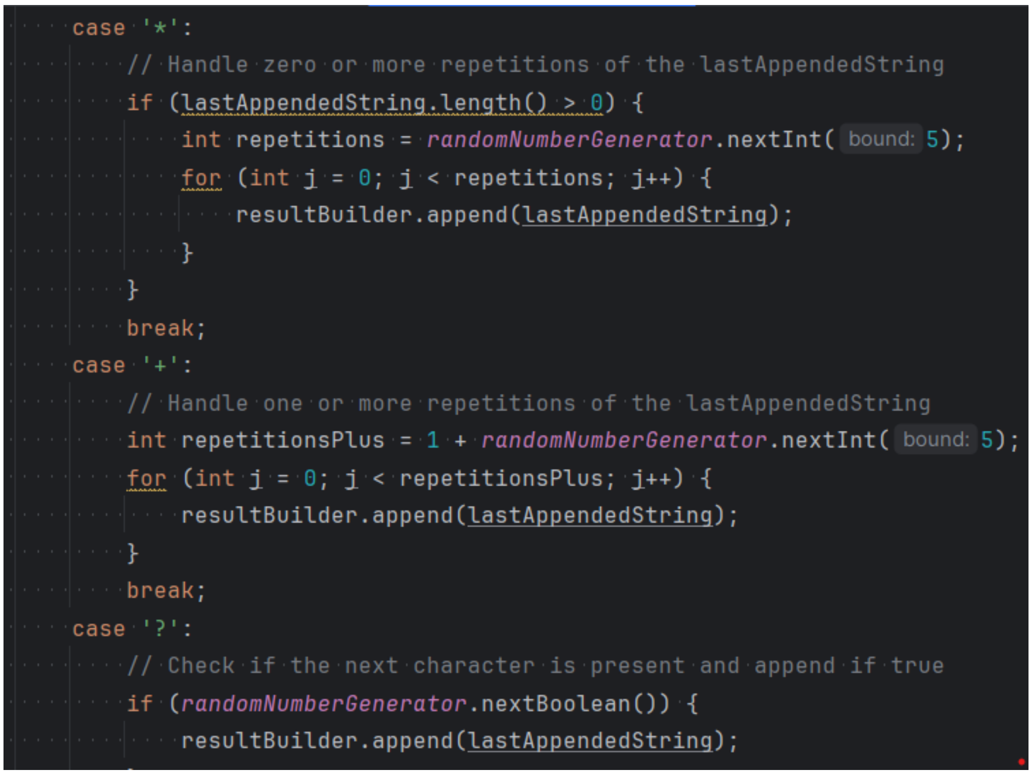
```java
public static List<String> generateFromRegex(String regex, int numStrings) {
    List<String> resultList = new ArrayList<>();

    // Loop to generate the specified number of strings
    for (int s = 0; s < numStrings; s++) {
        StringBuilder resultBuilder = new StringBuilder();
        String lastAppendedString = "";

        // Loop through each character in the regex pattern
        for (int i = 0; i < regex.length(); i++) {
            char currentChar = regex.charAt(i);
            switch (currentChar) {
```

The generateFromRegex method is designed to generate strings based on a given regular expression (regex) pattern and the specified number of strings to generate (numStrings). It starts by initializing an empty ArrayList called resultList to store the generated strings. Inside the method, there is a loop that runs numStrings times, ensuring the desired number of strings is generated.

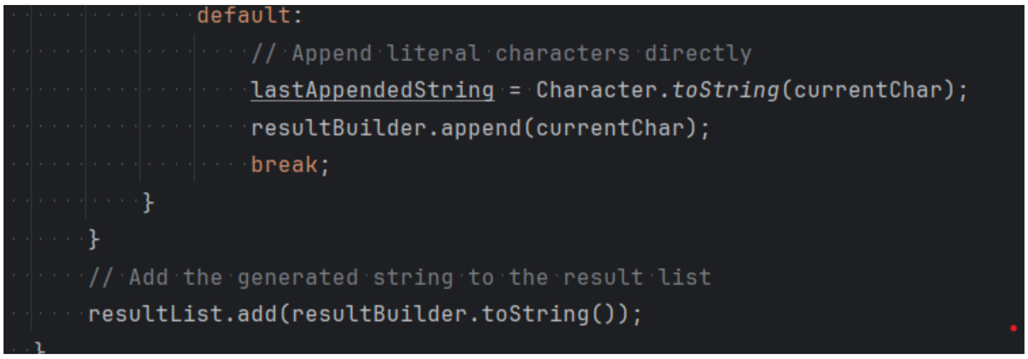
For each iteration, a new StringBuilder named resultBuilder is created to construct the current generated string. Additionally, there's a variable lastAppendedString that keeps track of the last string segment appended to resultBuilder, which is crucial for handling repetitions in the regex pattern.



```
    case '*':
        // Handle zero or more repetitions of the lastAppendedString
        if (lastAppendedString.length() > 0) {
            int repetitions = randomNumberGenerator.nextInt( bound: 5);
            for (int j = 0; j < repetitions; j++) {
                resultBuilder.append(lastAppendedString);
            }
        }
        break;
    case '+':
        // Handle one or more repetitions of the lastAppendedString
        int repetitionsPlus = 1 + randomNumberGenerator.nextInt( bound: 5);
        for (int j = 0; j < repetitionsPlus; j++) {
            resultBuilder.append(lastAppendedString);
        }
        break;
    case '?':
        // Check if the next character is present and append if true
        if (randomNumberGenerator.nextBoolean()) {
            resultBuilder.append(lastAppendedString);
```

The method then iterates through each character in the regex pattern using a loop, switching on the current character to handle different cases such as groups enclosed in parentheses, zero or more repetitions (*), one or more repetitions (+), optional characters (?), and custom repetition counts specified after the ^ character.

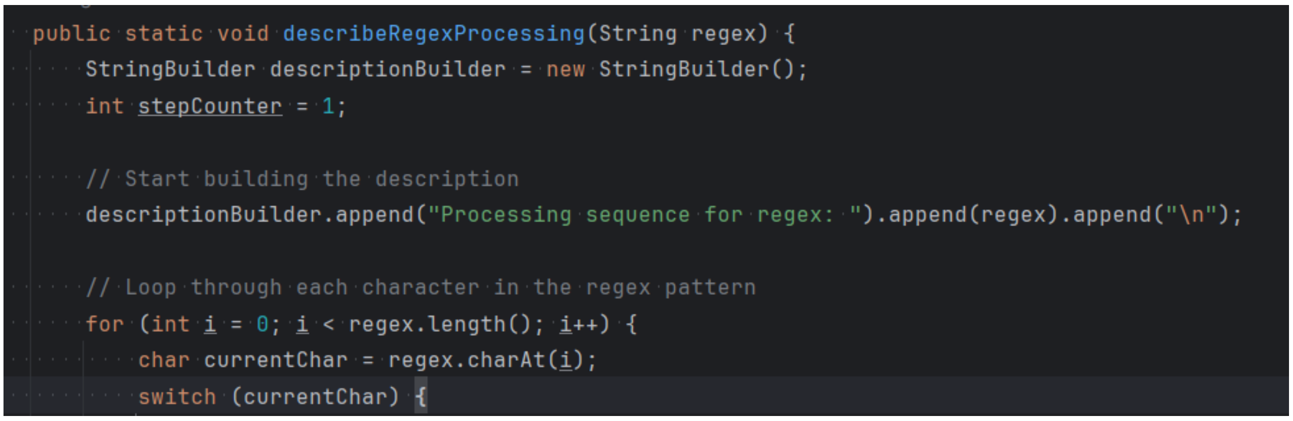
The randomness in repetitions and group selections is introduced using a Random object (randomNumberGenerator), ensuring varied outputs for each run of the method. Finally, each generated string is added to the resultList, and once all iterations are complete, the method returns the resultList containing the generated strings based on the provided regex pattern.



```
            default:
                // Append literal characters directly
                lastAppendedString = Character.toString(currentChar);
                resultBuilder.append(currentChar);
                break;
        }
    }
    // Add the generated string to the result list
    resultList.add(resultBuilder.toString());
```

This method encapsulates the logic for regex-based string generation, encompassing key regex concepts such as groups, repetitions, and optional characters, making it a versatile tool for text generation tasks based on complex patterns.

For the bonus point I have implemented a describeRegexProcessing method.

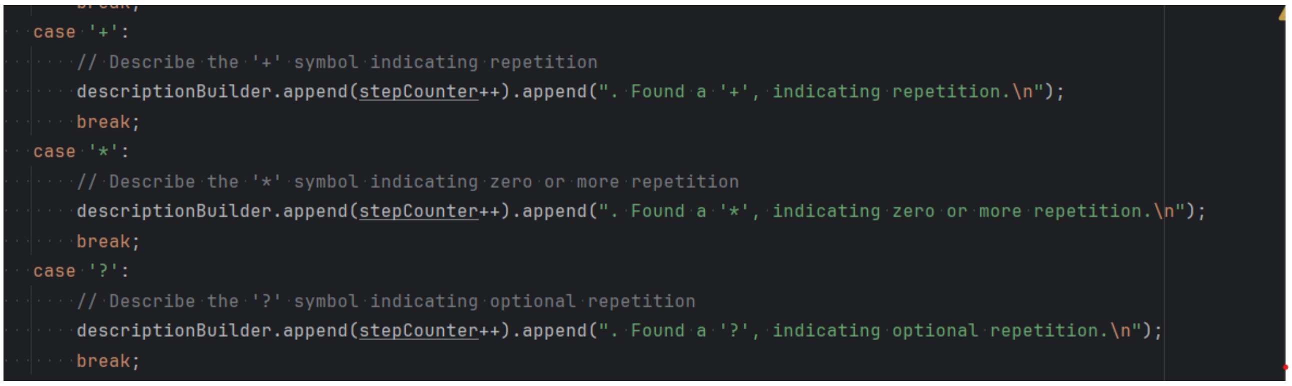


```
public static void describeRegexProcessing(String regex) {
    StringBuilder descriptionBuilder = new StringBuilder();
    int stepCounter = 1;

    // Start building the description
    descriptionBuilder.append("Processing sequence for regex: ").append(regex).append("\n");

    // Loop through each character in the regex pattern
    for (int i = 0; i < regex.length(); i++) {
        char currentChar = regex.charAt(i);
        switch (currentChar) {
```

The describeRegexProcessing method is designed to provide a detailed description of the processing steps involved in parsing a given regular expression (regex) pattern. It starts by initializing a StringBuilder named descriptionBuilder to construct the description of regex processing steps. Additionally, it initializes a stepCounter variable to keep track of the processing steps.



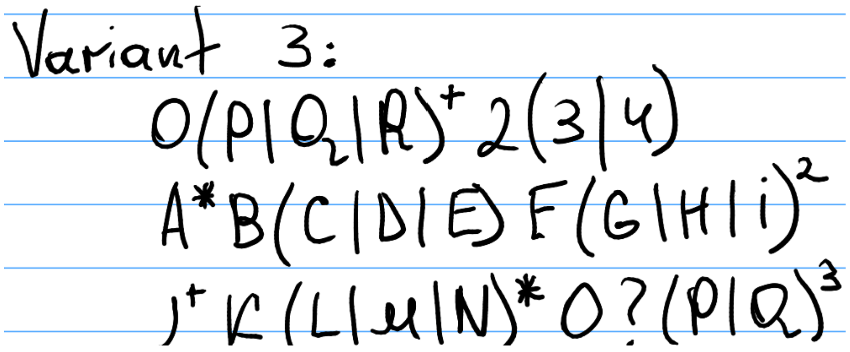
```
case '+':
    // Describe the '+' symbol indicating repetition
    descriptionBuilder.append(stepCounter++).append(". Found a '+', indicating repetition.\n");
    break;
case '*':
    // Describe the '*' symbol indicating zero or more repetition
    descriptionBuilder.append(stepCounter++).append(". Found a '*', indicating zero or more repetition.\n");
    break;
case '?':
    // Describe the '?' symbol indicating optional repetition
    descriptionBuilder.append(stepCounter++).append(". Found a '?', indicating optional repetition.\n");
    break;
```

The description building process begins with appending an initial message indicating the start of the processing sequence for the given regex pattern, including the pattern itself. Next, the method iterates through each character in the regex pattern using a loop, switching on the current character (currentChar) to handle different cases encountered in the regex pattern.

For each character encountered, the method provides a descriptive message based on the character's significance in regex syntax. It describes groups found in parentheses, symbols indicating repetition (+, *, ?), and the ^ symbol indicating a repetition count.

4. Screen printing of program output:

For checking if the program generates the proper result I have checked via the regular expressions from my variant:



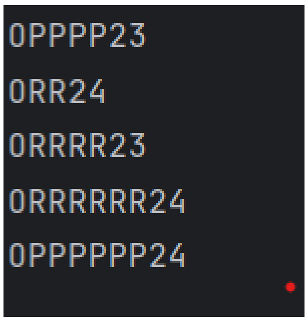
Variant 3:

$O(P|Q|R)^{+}2(3|4)$

$A^{*}B(C|D|E)F(G|H|i)^{2}$

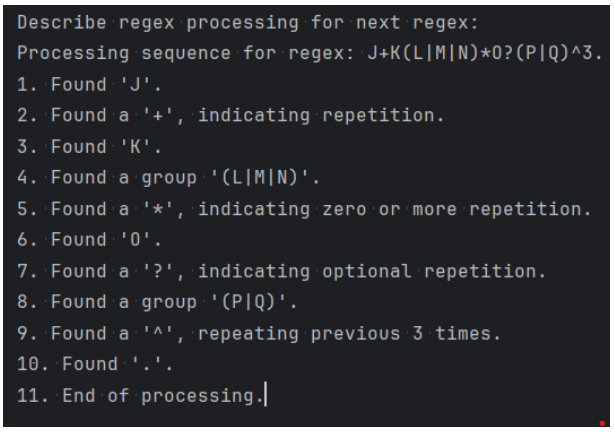
$J^{+}K(L|M|N)^{*}O?(P|Q)^{3}$

After pressing run, I have obtained the following output:



```
OPPPP23
ORR24
ORRRR23
ORRRRRR24
OPPPPPP24
```

The output includes the generated strings for each regex pattern, providing insights into how regex patterns can influence string generation based on defined rules and repetitions. Same for each expression.



```
Describe regex processing for next regex:
Processing sequence for regex: J+K(L|M|N)*O?(P|Q)^3.
1. Found 'J'.
2. Found a '+', indicating repetition.
3. Found 'K'.
4. Found a group '(L|M|N)'.
5. Found a '*', indicating zero or more repetition.
6. Found 'O'.
7. Found a '?', indicating optional repetition.
8. Found a group '(P|Q)'.
9. Found a '^', repeating previous 3 times.
10. Found '.'.
11. End of processing.
```

Additionally, the description of regex processing steps is printed, offering a structured understanding of how the program interprets and processes the provided regex pattern.

5. Conclusions:

In conclusion, the laboratory work on regular expressions generation has provided a hands-on experience in understanding and applying regex patterns for string generation tasks. Through practical exercises, we gained insights into constructing regex patterns to match specific text formats and sequences, thereby automating the generation of structured data. The use of metacharacters, quantifiers, and grouping mechanisms allowed us to create versatile patterns for capturing complex text patterns efficiently. Additionally, exploring repetition symbols and optional elements enhanced our ability to tailor regex patterns for varied string generation requirements. Overall, this laboratory work deepened our understanding of regex concepts and their practical applications in text processing and data manipulation tasks.