MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Grebennicova Ecaterina FAF-223

Report

Regular expressions

of Formal Languages &
Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

1. Theory

Regular expressions, often abbreviated as regex, are a powerful tool used in programming to search, match, and manipulate text. They allow you to define a search pattern using a sequence of characters. These patterns can specify a set of strings that match a particular syntactic rule, making them incredibly useful for validating text formats (like email addresses or phone numbers), searching within texts, replacing parts of texts, and splitting strings based on specific patterns.

2. Objectives

- Write a code that will generate valid combinations of symbols conform given regular expressions
- In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times

3. Implementation Description

In this Java program we generate a random string that matches a specific pattern described by a custom regular expression-like format and then print it.

The code:

```
      System.out.println("Generating strings for regex 1:");
      System.out.println(generateFromRegex("(a|b)(c|d)E+G"));
```

This code snippet represents the main method, where inside we call another function which responds to string generation for the first regex and later print it on the screen.

Using a for loop we iterate over each character of the input regex string. Depending on the character encountered, different actions are taken, as determined by a switch statement.

```
    for (int i = 0; i < regex.length(); i++) {
   char c = regex.charAt(i);
   switch (c) {
       case '(':
           int closingParethesis = regex.indexOf(')', i);
           boolean asterisk = closingParethesis + 1 < regex.length() &&
regex.charAt(closingParethesis + 1) == '*';
           String[] options = regex.substring(i + 1,
closingParethesis).split("\\|");

           if (asterisk) {
               if (random.nextBoolean()) {
                   int option = random.nextInt(options.length);
                   lastAppended = options[option];
                   result.append(lastAppended);

                   int repetitions = random.nextInt(5);
                   for (int j = 0; j < repetitions; j++) {
                       result.append(lastAppended);
                   }
               }
           } else {
               lastAppended = options[random.nextInt(options.length)];
               result.append(lastAppended);
           }

           i = closingParethesis;
           if (asterisk) {
               i++;
           }
           break;
```

Case "(":

When an opening parenthesis "(" is encountered, it indicates the start of a group. The method finds the corresponding closing parenthesis ")" to identify the entire group. The content between the parentheses is split into options based on the "|" delimiter, representing different choices within the group. After we check if the closing parenthesis is immediately followed by an asterisk "*". The asterisk signifies that the preceding element or the group can be repeated zero or more times. But before that if the closing parenthesis is followed by an asterisk, the method uses random.nextBoolean() to make a random decision on whether to append an option from the group or skip it. If the decision is to append, it randomly selects one of the options. After selecting an option, the method decides randomly on the number of times (0 to 4) to repeat the chosen option. It appends the selected option to the result. But if the group is followed by an asterisk and the method decides to process the group, the index "i" is incremented an additional time to skip over the asterisk character, ensuring it's not processed again in the next loop iteration. If there is not an asterisk following the group, one option from the group is chosen randomly and appended to result without considering repetition.

```
      case '+':
         int newRepetitions = 1 + random.nextInt(4);
         for (int j = 0; j < newRepetitions; j++) {
             result.append(lastAppended);
         }
         break;
```
Case “+“:

It appends the selected option before for a number of times randomly chosen and appends them all to the string created.

```
      case '^':
         int dot = regex.indexOf('.', i);
         if (dot != -1) {
             String numStr = regex.substring(i + 1, dot);
             int repeatCount;
             try {
                 repeatCount = Integer.parseInt(numStr);
             } catch (NumberFormatException e) {
                 repeatCount = 1;
             }
             for (int j = 0; j < repeatCount - 1; j++) {
                 result.append(lastAppended);
             }
             i = dot;
         } else {
             lastAppended = Character.toString(c);
             result.append(c);
         }
         break;
```
Case “^”:

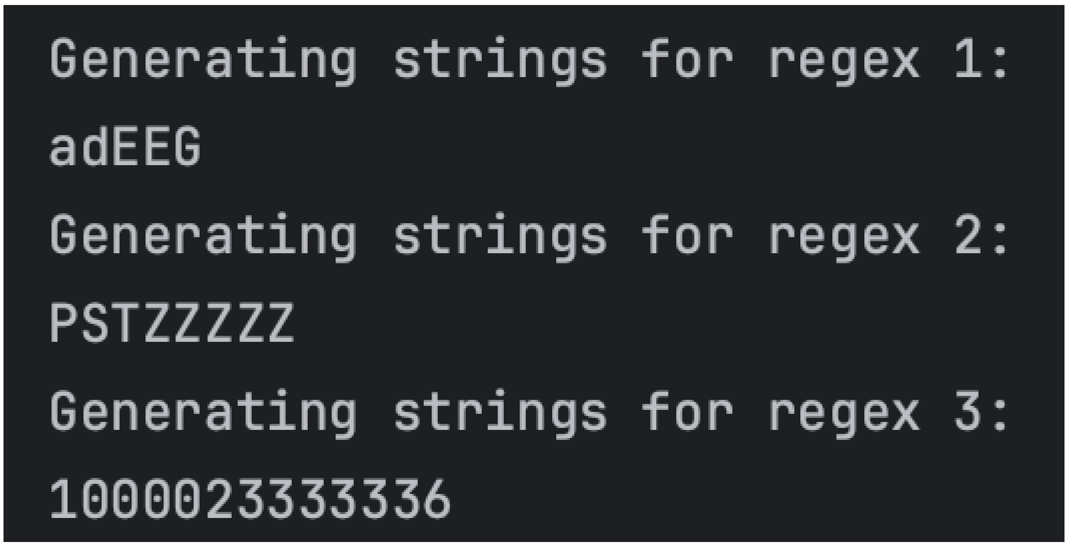
The “^” character is used to indicate a specific number of repetitions for the previously appended character or group. The number of repetitions is specified directly after the “^” and before a dot “.”. We search for the index of the first “.” following the ^. This dot marks the end of the repetition count. If a dot is found, the substring between the “^” and the “.” is extracted as a string. This substring is supposed to represent the repetition count. Converting it to integer we determine how many times the last appended sequence should be repeated. If the conversion fails due to an invalid format (NumberFormatException), repeatCount defaults to 1. A for-loop repeats the last appended sequence a number of times. If no dot is found following a “^”, this implies there isn't a valid repetition count. In such cases, ^ is treated as a literal character to be appended to the result.

```
      default:
         // Directly append literal characters and fixed sequences
         lastAppended = Character.toString(c);
         result.append(c);
         break;
```

Case default:

Any characters in the regex string that does not have a special meaning ((, ^, etc.) are considered literal characters to be directly appended to the result string and simply appended to the result.

Output:



```
Generating strings for regex 1:
adEEG
Generating strings for regex 2:
PSTZZZZZ
Generating strings for regex 3:
1000023333336
```

Bonus point:

For bonus point, we had to implement a method that takes a regular expression as a string and produces a step-by-step textual description of how the regex is processed.

```
      description.append("Processing sequence for regex:
").append(regex).append("\n");

      for (int i = 0; i < regex.length(); i++) {
         char c = regex.charAt(i);
         switch (c) {
             case '(':
                 int closingParenthesis = regex.indexOf(')', i);
                 boolean asterisk = closingParenthesis + 1 < regex.length() &&
regex.charAt(closingParenthesis + 1) == '*';
                 description.append(stepCounter++).append(". Found a group
```

```java
'(").append(regex.substring(i + 1, closingParenthesis)).append(")'.\n");
                 if (asterisk) {
                     description.append(stepCounter++).append(". This group is
followed by '*', indicating it can be chosen zero or more times.\n");
                 } else {
                     description.append(stepCounter++).append(". This group will be
chosen exactly once.\n");
                 }
                 i = closingParenthesis;
                 break;
             case '+':
                 description.append(stepCounter++).append(". Found a '+', indicating
the previous character/group will be repeated one to four times.\n");
                 break;
             case '*':
                 break;
             case '^':
                 int dot = regex.indexOf('.', i);
                 if (dot != -1) {
                     String numStr = regex.substring(i + 1, dot);
                     int repeatCount = 1;
                     try {
                         repeatCount = Integer.parseInt(numStr);
                     } catch (NumberFormatException e) {
                         repeatCount = 1;
                     }
                     description.append(stepCounter++).append(". Found a '^',
specifying to repeat the previous character/group ").append(repeatCount).append("
times.\n");
                     i = dot;
                 } else {
                     description.append(stepCounter++).append(". Found a '^' but no
following numerical repetition specification, treating as literal.\n");
                 }
                 break;
             default:
                 description.append(stepCounter++).append(". Found a literal
character '").append(c).append("'.\n");
                 break;
         }
     }

     description.append(stepCounter).append(". End of processing.\n");
     System.out.println(description);
```

We use a StringBuilder to build this description efficiently. First of all we append to the description a text that will initiate us what regex we are going to process.

We iterate over each character in the regex string using a for loop. Within the loop, it checks the current character to determine the appropriate processing action based on a `switch` statement. Based on the cases and iterations it marks down the steps.

- Literal character: If he sees a literal character he prints it on the screen

- Opening parenthesis: If he sees an opening parenthesis he goes through all characters/groups till the closing parenthesis. He prints the group found within the parenthesis calling to choose a character.

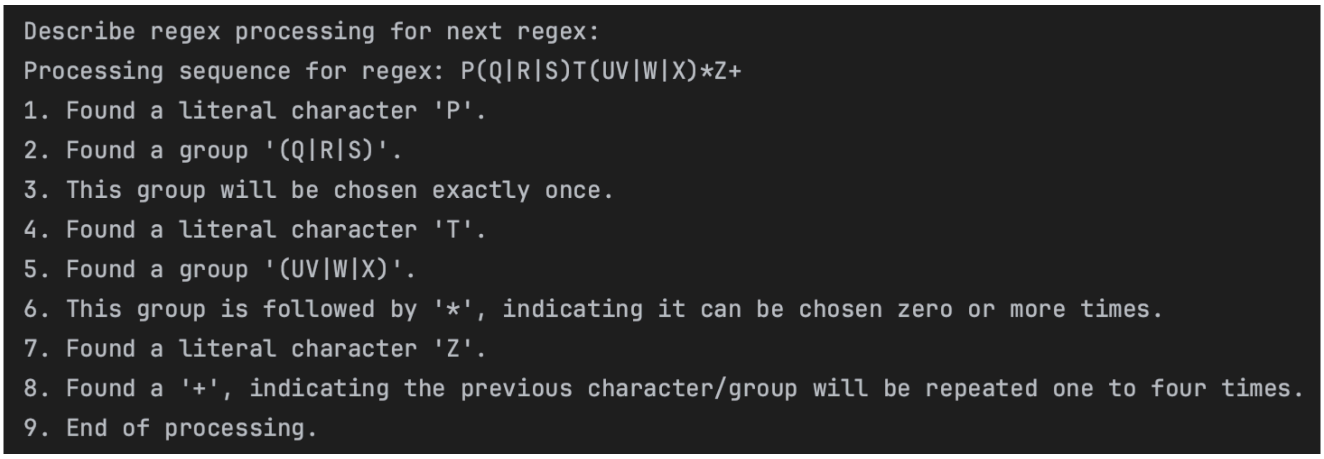
- *: If after the closing parenthesis comes an asterisk he prints that the the group can be written zero or more times.

- ^: If after the closing parenthesis comes the circumflex sign, it writes that the previous chosen character has to be repeated a specified number of times.

- +: If after the literal character comes a plus, it writes that the character will be repeated one to four times.

After iterating through all characters it marks that the end of processing has come.

Output:



```
Describe regex processing for next regex:
Processing sequence for regex: P(Q|R|S)T(UV|W|X)*Z+
1. Found a literal character 'P'.
2. Found a group '(Q|R|S)'.
3. This group will be chosen exactly once.
4. Found a literal character 'T'.
5. Found a group '(UV|W|X)'.
6. This group is followed by '*', indicating it can be chosen zero or more times.
7. Found a literal character 'Z'.
8. Found a '+', indicating the previous character/group will be repeated one to four times.
9. End of processing.
```

4. Conclusion

In conclusion, I could say that this lab was not a difficult one, but requested a lot of work and understanding of the regex and its functionality. The `RegexGenerator` class provides a unique approach to interpreting and generating strings based on a custom regex-like pattern. It supports basic regex operations such as grouping `()`, alternatives `|`, a custom repetition mechanism using `^`, and traditional quantifiers like `*` and `+`, with some modifications to their standard meanings. The generation process involves selecting random options within groups, repeating characters or groups based on specific rules, and handling literal characters. The `describeRegexProcessing` method offers description about the interpretation of the regex pattern, providing a step-by-step breakdown of its structure and logic. Overall, the code was successfully completed.