MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică
Departamentul Inginerie Software și Automatică

Report

Zagorodniuc Anastasia FAF-223

Regular expressions

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

Chișinău – 2023

1. Theory

Regular expressions, commonly known as regex, are potent tools employed in programming for text search, matching, and manipulation. They enable the definition of search patterns using character sequences. These patterns can specify sets of strings adhering to specific syntactic rules, thereby proving invaluable for validating text formats such as email addresses or phone numbers, searching within texts, replacing text portions, and segmenting strings based on defined 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 C# 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:

```
Console.WriteLine("Generating strings for regex 1:");
Console.WriteLine(RegexGenerator.GenerateFromRegex("(S|T)(U|V)W*Y+24"));
```

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.

Case “(“:

Encountering an opening parenthesis signifies the commencement of a group. The method identifies the corresponding closing parenthesis to delineate the entire group. Content between parentheses is split into options delimited by “|”, representing different choices within the group. Subsequently, it's checked if the closing parenthesis is immediately followed by an asterisk “*”. The asterisk denotes that the preceding element or group can repeat zero or more times. If the closing parenthesis is followed by an asterisk, the method employs random.nextBoolean() to decide whether to append an option from the group or skip it. Upon choosing to append, a random option is selected, and the method randomly determines the number of times (0 to 4) to repeat the chosen option before appending it to the result. If the group is followed by an asterisk and the method decides to process it, the index “i” is incremented an additional time to bypass the asterisk character in the subsequent loop iteration. If no asterisk follows the group, one option from the group is randomly chosen and appended to the result without considering repetition.

```
case '(':
    // Find the matching ')'
    int closingParenthesis = regex.IndexOf(')', i);
    // Check if followed by '*'.
    bool asterisk = closingParenthesis + 1 < regex.Length && regex[closingParenthesis + 1] == '*';
    // Split options within the group.
    string[] options = regex.Substring(i + 1, closingParenthesis - i - 1).Split('|');

    // case '*'
    if (asterisk)
    {
        // If followed by '*', decide to append an option and possibly repeat it.
        if (random.Next(2) == 0)
        {
            int option = random.Next(options.Length);
            lastAppended = options[option];
            result.Append(lastAppended);

            // Decide on repeating the chosen option 0 to 4 more times
            int repetitions = random.Next(5);
            for (int j = 0; j < repetitions; j++)
            {
                result.Append(lastAppended);
            }
        }
    }
    else
    {
        // If not followed by '*', select and append a random option.
        lastAppended = options[random.Next(options.Length)];
        result.Append(lastAppended);
    }

    // Skip to ')'
    i = closingParenthesis;
    if (asterisk)
    {
        // Skip also the '*' character
        i++;
    }
    break;
```

Case “+“:

This appends the selected option a randomly chosen number of times and concatenates them to the resultant string.

```
case '+':
    // Append 1 to 4 repetitions of the previous character/group
    int newRepetitions = 1 + random.Next(4);
    for (int j = 0; j < newRepetitions; j++)
    {
        result.Append(lastAppended);
    }
    break;
```

Case “^”:

The “^” character indicates a specific number of repetitions for the previously appended character or group. The repetition count is specified after the “^” and before a dot “.”. The method searches for the index of the first “.” following the “^”. If found, the substring between “^” and “.” is extracted as the repetition count, converted to an integer to determine the number of times the last appended sequence should repeat. In cases of conversion failure due to an invalid format, the repeat count defaults to 1. A for-loop then repeats the last appended sequence as many times as specified. If no dot follows a “^”, indicating an absence of a valid repetition count, “^” is treated as a literal character to append to the result.

```
case '^':
    // Find the next '.' to get the repetition count
    int dot = regex.IndexOf('.', i);
    if (dot != -1)
    {
        // Extract the number between '^' and '.' for repetition count
        string numStr = regex.Substring(i + 1, dot - i - 1);
        int repeatCount;
        if (int.TryParse(numStr, out repeatCount))
        {
            // Repeat the previous character/group as specified. Subtract 1 because it already exists once
            for (int j = 0; j < repeatCount - 1; j++)
            {
                result.Append(lastAppended);
            }
            // Move the index to the position of the dot
            i = dot;
        }
        else
        {
            lastAppended = c.ToString();
            // If no '.' is found, treat '^' as a literal
            result.Append(c);
        }
    }
    break;
```

Case default:

Any characters in the regex string lacking special meaning (e.g., “(”, “^”, etc.) are deemed literal characters and directly appended to the result string.

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

Output:

```
Generating strings for regex 1:
SVW*YY24
Generating strings for regex 2:
LNO3P*Q2
Generating strings for regex 3:
R*SVWZ2
```

Bonus point:

```
var description = new StringBuilder();
int stepCounter = 1;

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

for (int i = 0; i < regex.Length; i++)
{
    char c = regex[i];
    switch (c)
    {
        case '(':
            int closingParenthesis = regex.IndexOf(')', i);
            bool asterisk = closingParenthesis + 1 < regex.Length && regex[closingParenthesis + 1] == '*';
            description.Append(stepCounter++).Append(". Found a group '(").Append(regex.Substring(i + 1, closingParenthesis - i - 1)).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 '^':
            int dot = regex.IndexOf('.', i);
            if (dot != -1)
            {
                string numStr = regex.Substring(i + 1, dot - i - 1);
                int repeatCount = 1;
                if (int.TryParse(numStr, out repeatCount))
                {
                    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");
Console.WriteLine(description);
```

For bonus points, a method was implemented to provide a step-by-step textual description of how the regex is processed. Utilizing a StringBuilder for efficient description building, the method iterates over each character in the regex string, determining the appropriate processing action based on a switch statement. It then logs each step of the process, including handling literal characters, groups delineated by parentheses, asterisks indicating zero or more repetitions, circumflex signs specifying repetition counts, and plus signs indicating repetition of characters.Output:

```
Describe regex processing for next regex:
Processing sequence for regex: R*S(T|U|V)W(X|Y|Z)^2
1. Found a literal character 'R'.
2. Found a literal character '*'.
3. Found a literal character 'S'.
4. Found a group '(T|U|V)'.
5. This group will be chosen exactly once.
6. Found a literal character 'W'.
7. Found a group '(X|Y|Z)'.
8. This group will be chosen exactly once.
9. Found a literal character '2'.
10. End of processing.
```

4. Conclusion

In conclusion, while not overly challenging, this lab necessitated significant effort and comprehension of regex and its functionalities. The RegexGenerator class presents a distinct approach to interpreting and generating strings based on a custom regex-like pattern. It supports

fundamental regex operations such as grouping (), alternatives |, custom repetition mechanisms using ^, and traditional quantifiers like * and +, albeit with some modifications to their standard interpretations. The generation process involves selecting random options within groups, repeating characters or groups based on specific rules, and handling literal characters. Additionally, the describeRegexProcessing method offers a detailed breakdown of the regex pattern's interpretation, facilitating step-by-step comprehension of its structure and logic. Overall, the code was successfully executed.