MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work n.4

of Formal Languages & Finite Automata

Checked by:

Cretu Dumitru, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

1. Theory:

Regular expression - (shortened as regex or regexp), sometimes referred to as rational expression, is a sequence of characters that specifies a match pattern in text. Usually such patterns are used by string-searching algorithms for "find" or "find and replace" operations on strings, or for input validation. Regular expression techniques are developed in theoretical computer science and formal language theory. The concept of regular expressions began in the 1950s, when the American mathematician Stephen Cole Kleene formalized the concept of a regular language. They came into common use with Unix text-processing utilities. Different syntaxes for writing regular expressions have existed since the 1980s, one being the POSIX standard and another, widely used, being the Perl syntax. Regular expressions are used in search engines, in search and replace dialogs of word processors and text editors, in text processing utilities such as sed and AWK, and in lexical analysis. Regular expressions are supported in many programming languages. Library implementations are often called an "engine", and many of these are available for reuse.

2. Purpose of the task work:

- Write and cover what regular expressions are, what they are used for.
- 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 conforms to 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:

For the program to be more clear and understandable, first off all we create a Regular Expression class at which we will make calls. This class has no attributes, it will also have static methods and methods that will return the generated string for the specific regular expression

```
class Regex:
    def __init__(self):
        pass
```

Then, here comes the main piece of gold of our code, a regular expression for all the regular expressions. Even if it sounds quite paradoxical, the idea is that we compute a regular expression to determine different regular expressions. A regular expression can have symbols: a,b,c,0,1; a possible choice of symbols (a|b|0|cde); and conditions for how many times we will print the respective symbols: *, +, ?, {5}.

So, for a specific regex we have to underline its symbols and conditions, to do this we use the findall method of re module in python, to find all the matches. The method returns a list of all non-overlapping matches in the string. If one or more capturing groups are present in the pattern, return a list of groups; this will be a list of tuples if the pattern has more than one group. Empty matches are included in the result.

For the method to work we compute “The regular expression for all regular expressions”, and

then we call the findall method in pair with the current regular expression

```
def get_matches(self, regex: str) -> list:
        matches = re.findall(r'([*?+]|\{\d+\})|(\(.*?\)|[A-Za-z0-9])', regex)
        return matches
```

Here, `r'([*?+]|\{\d+\})|(\(.*?\)|[A-Za-z0-9])'` represents our regular expression for all the other regular expressions, and for example the expression (a|b)(c|d)E+G?, the get_matches function will return [('', '(a|b)'), ('', '(c|d)'), ('', 'E'), ('+', ''), ('', 'G'), ('?', '')].

The string at index 1 of the set represents the symbol or possible variants of symbols, and string at index 0 of the set represents the condition of how many times to print the following symbol/symbols.

Now that we have divided our regex into matches, we have to generate a random string which validates the condition of the regex. To do this we just compute a code that checks if the symbols have condition or not.

```
def generate_string(self, regex: str) -> str:
        generated_string = ""
        matches = self.get_matches(regex)

        for i in range(len(matches)):
            expression = matches[i][1]
            if i == len(matches)-1:
                condition = ''
            else:
                condition = matches[i + 1][0]
            # print(expresion, condition)

            if expression and condition:
                    generated_string += self.generate_substring(expresion, condition)

            elif expression:
                    generated_string += self.generate_substring(expresion, '')

        return generated_string
```

After we check if the expression has condition or not, we call the generate_substring method, which generates a substring based on the expression and condition and after we append this substring to our main string.

```
def generate_substring(self, expression: str, condition: str) -> str:
        substring = ''

        if expression.startswith('(') and expression.endswith(')'):
            elements = expression[1:-1].split('|')
        else:
            elements = expression

        if condition.startswith('{') and condition.endswith('}'):
            count = int(condition[1:-1])
            for _ in range(count):
                substring += random.choice(elements)
        elif condition == '*':
            count = random.randint(0, LIMIT)
```

```
            for _ in range(count):
                element = random.choice(elements)
                substring += element
        elif condition == '+':
            count = random.randint(1, LIMIT)
            for _ in range(count):
                element = random.choice(elements)
                substring += element
        elif condition == '?':
            if random.choice([True, False]):
                substring = random.choice(elements)
        else:
            substring = random.choice(elements)

        return substring
```

The generate_substring method works in the following way. It checks if there is a condition or not, if not it just returns the element or a random element of the elements in the expression. Else if there is an condition, it check what condition it is (*, +, ?, {number}) and based on the condition generates the substring, then returns the substring.

Also, for the bonus point we created a function that shows us step by step what we do first, second and so on when we process a regular expression.

```
def process_regex(self, regex: str) -> None:
        step = 1
        matches = self.get_matches(regex)
        # print(matches)
                print(f"\nSequence of processing the following regular
expression: {regex}")
        for i in range(len(matches)):
            expression = matches[i][1]
            if i == len(matches)-1:
                condition = ''
            else:
                condition = matches[i + 1][0]

            if expression and condition:
                print("Step " + str(step) + ":")
                print(f'Looking at {expression}{condition}')
                step += 1
            elif expression:
                print("Step " + str(step) + ":")
                print(f'Looking at {expression}')
                step += 1
```

The function gets again the possible matches of the regex, calling the get_matches function. After, with the same logic it prints the symbol or symbols with their condition or not, if it does not exist.

4. Program execution:

For a more clear way to test the program, we stored all the regexes in a text file (‘input.txt’), and all the generated strings are written in another text file (‘output.txt’). and in the terminal we just print the sequence processing of an specific regex wich is selected in main.py

input.txt:

```
(a|b)(c|d)E+G?
P(Q|R|S)T(UV|W|X)*Z+
```

```
1(0|1)*2(3|4){5}36
M?N{2}(O|P){3}Q*R+
(X|Y|Z){3}8+(9|0)
(H|i)(J|K)L*N?
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}
(S|T)(U|V)W*Y+24
L(M|N)O{3}P*Q(2|3)
R*S(T|U|V)W(X|Y|Z){2}
```

output.txt:

```
Generated strings for regex '(a|b)(c|d)E+G?' :
bcEEEEEG
adEEEEEEEE
bdEEEEEEEEEG
acEEEG
bcEEEE

Generated strings for regex 'P(Q|R|S)T(UV|W|X)*Z+' :
PQTXXZZZZ
PSTXWUVWUVZZZZ
PSTUVZZ
PRTWWUVXWXZZZ
PRTXWUVUVXWZZZZZZZ

Generated strings for regex '1(0|1)*2(3|4){5}36' :
11101023444336
100001024433436
10010023444436
10001024334336
1100023334336

Generated strings for regex 'M?N{2}(O|P){3}Q*R+' :
NNPPPQQQQQQQRRRRRRRRR
NNOOPQQQQQQQQQQRRRRRRRR
NNOPPQQQQQRRRRRRR
MNNOPOQQQQQQQQQRRRRRRRRRR
NNPOPQQQQQQQQRR

Generated strings for regex '(X|Y|Z){3}8+(9|0)' :
YYY8888888880
YXY88888888889
ZYY88888888880
YYY8888888889
ZXX8888880

Generated strings for regex '(H|i)(J|K)L*N?' :
iKLLL
iKLLLLLLLLLLN
HKLLL
HJLLLLLLLLL
iJLLLLN

Generated strings for regex 'O(P|Q|R)+2(3|4)' :
OQRPQP24
OPQPRRRPQR23
OPQPQQQ24
OQQR24
ORPRQPQ24

Generated strings for regex 'A*B(C|D|E)F(G|H|i){2}' :
AAABEFiG
AAAAAAABEFHH
ABCFiG
```

```
AAAAABCFiG
AAAAAAAAAABEFGi

Generated strings for regex 'J+K(L|M|N)*O?(P|Q){3}' :
JJKLLLLMNQPP
JJJJJJJJJJKNMOQPP
JJJJJJJJJKMMNLNNLQPP
JJJJKMMMLNMPPQ
JJJKMLLMLPQQ

Generated strings for regex '(S|T)(U|V)W*Y+24' :
TUWWWYYY24
SUYYYYY24
SUY24
SUWWWWWWWY24
TUWWWWYYYYYY24

Generated strings for regex 'L(M|N)O{3}P*Q(2|3)' :
LMOOOPPPPPPPPQ3
LMOOOPPPPPPPPPQ3
LMOOOPPPQ3
LMOOOPPPPQ2
LNOOOPPPPPPPQ3

Generated strings for regex 'R*S(T|U|V)W(X|Y|Z){2}' :
RRRRRRRRRSTWZX
RRRRRRRRRSUWYY
SVWZX
RRRRRRRRSUWYY
RRRSVWZY
```

The sequence processing of the 9th regex from the input.txt:

```
Sequence of processing the following regular expression: J+K(L|M|N)*O?(P|Q){3}

Step 1:
Looking at J+
Step 2:
Looking at K
Step 3:
Looking at (L|M|N)*
Step 4:
Looking at O?
Step 5:
Looking at (P|Q){3}
```

5. Conclusion:

In conclusion, the successful implementation of the regular expression class in this university laboratory work marks a significant achievement in our understanding and application of fundamental concepts of formal languages and usage of regular expressions. Through meticulous design, diligent coding, and rigorous testing, we have created a program that generates how many strings we want based on a specific regular expression, and also we computed a program that shows a sectional step by step processing of a regular expression, which also helps to our understanding of regular expressions and how they work.