MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Laboratory work 4:
Regular expressions

Course: Formal Languages & Finite Automata

Author: Bostan Victor, FAF-222

Chișinău, 2024

THEORY

Regular expressions (regex) are a powerful tool used in computing for matching patterns within text. They originate from formal language theory and automata theory in computer science and are commonly used for searching, editing, and manipulating text. A regular expression is essentially a sequence of characters that forms a search pattern. It can be used for everything from finding and replacing text in a word processor to validating the format of email addresses or phone numbers in a database.

The syntax of regular expressions allows them to be highly versatile and expressive. Patterns can range from simple, such as finding a single word, to complex, such as identifying email addresses or validating user input. In programming, regular expressions are used in string searching algorithms for "find" or "find and replace" operations, as well as for input validation. They are implemented in various programming languages, including Python, Java, and JavaScript, and are utilized in many contexts, from software development and web development to data analysis and network security.

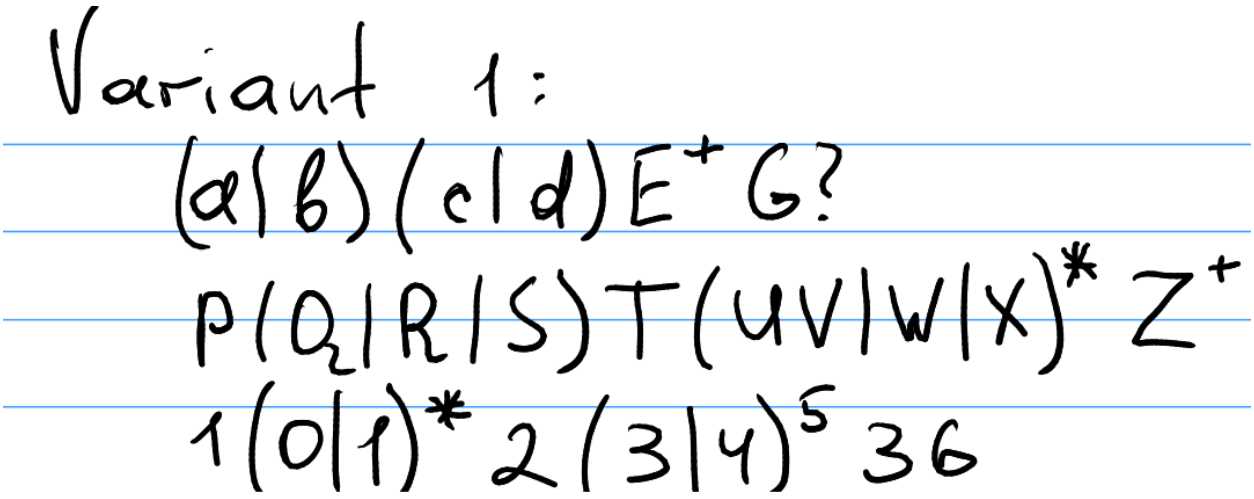
The use of regular expressions enables programmers and data scientists to work more efficiently by providing a concise and flexible means for matching strings of text. For example, they can quickly sift through large datasets to find relevant information or validate user inputs against expected patterns. Despite their power, regular expressions can be complex and may require a learning curve to understand and use effectively. However, mastering them can greatly enhance one's ability to process and analyze text, making them an invaluable tool in the arsenal of modern computing.

OBJECTIVES

1. Write and cover what regular expressions are, what they are used for;
2. 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)

IMPLEMENTATION DESCRIPTION

For this laboratory we are given 3 regular expressions for which we have to generate valid strings. Here is my variant, Variant 1:



Once again, I am going to complete the tasks in Python. The first thing that is needed to be done is to tokenize these regular expressions, which will allow me to iterate through each step of finding a valid string. The idea is the following. Let's take the first expression: **(a|b)(c|d)E$^+$G?**. I split this expression into tokens as following: **(a|b), (c|d), E$^+$, G?**. Each token represents a step in generating the string and after iterating through every step a valid word is generated. In my code, in the list of the regular expression tokens, I represent the tokens as tuples where the first element of the tuple is the number of repetitions a symbol is allowed to have (**1**, **?** – 0 or 1, **5**, + - 1 and more, * - 0 and more), and the second element is the token itself. So, to do this I created a *tokenize* function which takes as input a string written in the format given in the condition (besides the power of a number; so instead of (3|4)^5 I have (3|4)^5), and returns the list of tokens:

```
4   def tokenizer(reg_exp):
5       tokens = []
6       pre_tokens = []
7       triggers = ['*', '+', '?', '^']
8       temp = ''
9
10      # Split regex by brackets
11      reg_exp = reg_exp.replace('(', '%(').replace(')', ')%')
12      reg_exp = reg_exp.strip('%').replace('%%', '%')
13      pre_tokens = reg_exp.split('%')
14
15      # Moves the number of repetitions specifiers from an element to the previous element
16      for i in range(1, len(pre_tokens)):
17          print(i)
18          if pre_tokens[i][0] in triggers:
19              if pre_tokens[i][0] == '^':
20                  pre_tokens[i-1] += pre_tokens[i][0] + pre_tokens[i][1]
21                  pre_tokens[i] = pre_tokens[i][2:]
22              else:
23                  pre_tokens[i-1] += pre_tokens[i][0]
24                  pre_tokens[i] = pre_tokens[i][1:]
25
```

This is the first part of the function where I create the ”pre tokens”. I decided to call them like this since they represent parts of the string that will then help shape the actual tokens in the format I need them to be. What is actually done here, is I split the string by brackets, so a bracket is basically a pre token and everything in between and before or after brackets represent pre tokens as well. In the for loop I make sure that the repetition specifiers like *, + etc. are at the end of the correct element from the list. This is needed to be done because let’s say I have this regular expression: **(a|b)*A;** when I split by brackets the pre tokens will look like this **[‘(a|b)’, ‘*A’],** but should look like this: **[‘(a|b)*’, ‘A’].**

```
    # Checks each pre token to build the tokens
    for pre_token in pre_tokens:
        if pre_token:
            if pre_token[0] == '(':
                if pre_token[len(pre_token)-2] == '^':
                    temp = pre_token[1:-3]
                    tup = (pre_token[len(pre_token)-1], temp.split('|'))
                    tokens.append(tup)
                else:
                    if pre_token[len(pre_token)-1] in triggers:
                        temp = pre_token[1:-2]
                        tup = (pre_token[len(pre_token)-1], temp.split('|'))
                        tokens.append(tup)
                    else:
                        temp = pre_token[1:-1]
                        tup = ('1', temp.split('|'))
                        tokens.append(tup)
            else:
                temp = ''
                skip = False
                for i, char in enumerate(pre_token):
                    if not skip:
                        if char not in triggers:
                            temp += char
                        else:
                            if char == '^':
                                tokens.append((pre_token[i+1], [temp]))
                                temp = ''
                                skip = True
                            else:
                                tokens.append((char, [temp]))
                                temp = ''
                    else:
                        skip = False
                if temp:
                    tokens.append(('1', [temp]))

    return tokens
```

Finally, this is the last step in tokenizing the expression. This may seem complicated but essentially in short what I am doing is I first check if my pre token is a bracket or not. If it is a bracket, I get rid of the actual brackets and split the remaining string as well as figuring out what repetition symbol I have for this specific bracket. If it is not a bracket, I just iterate through the substring looking for the repetition specifiers in order to figure out the individual tokens. And of course, I append each token to the tokens list in tuples form as discussed before. Here is an example of the return of the function in the case of the first regular expression from variant 1:

```
Tokens for (a|b)(c|d)E+G?:
[('1', ['a', 'b']), ('1', ['c', 'd']), ('+', ['E']), ('?', ['G'])]
```

The next step after this is actually starting to generate the strings based on the regular expressions

```python
# Function to generate strings based on provided regular expression patterns
def generate_strings(reg_exp, n, limit):
    strings = []  # List to store the generated strings

    for i in range(n):
        string = ''  # Initialize the string to be built
        # Iterate through each token in the regular expression
        for tup in reg_exp:
            if tup[0] == '1':  # '1' denotes a mandatory symbol
                string += random.choice(tup[1])
            elif tup[0] == '?':  # '?' denotes an optional symbol
                nr = random.randint(0, 1)
                if nr:
                    string += random.choice(tup[1])
            elif tup[0] == '+':  # '+' denotes one or more repetitions of a symbol
                char = random.choice(tup[1])
                nr_of_chars = random.randint(1, limit)
                for j in range(nr_of_chars):
                    string += char
            elif tup[0] == '*':  # For other types, assume '*', denoting zero or more repetitions
                char = random.choice(tup[1])
                nr_of_chars = random.randint(0, limit)
                for j in range(nr_of_chars):
                    string += char
            else:  # The case when we have ^5 for example
                char = random.choice(tup[1])
                for j in range(int(tup[0])):
                    string += char

        strings.append(string)

    return strings
```

As you can see in the image, the *generate_strings* function first initializes an empty list *strings* for the output and generates 5 strings. A string is generated by first iterating through the tuples and for each one checking what is the number of repetitions allowed. If the allowed repetitions is 1 then we just append to the string a random symbol (or the only one if the list has length 1) from the token list. If the allowed repetitions is ?, we first have to generate a random number between 0 and 1 to check if we will append a symbol or not. If a 1 is generated then we append a random symbol from the token list. If the allowed repetitions is +, then we first generate the number of times the symbol is going to be appended, from 1 to limit (as it was in the condition) and append a random symbol from the token that number of times. If the allowed repetitions is *, then it's the same process as with +, but we generate a number between 0 and the limit. Any other case means that we have a fixed number of repetitions so we just append a symbol that number of times Here are some examples on how the function generates the strings (I chose n = 5 for simplicity):

```
5 random strings for Regular Expression (a|b)(c|d)E⁺G?:
['acEEEG', 'bdEE', 'bdEEG', 'bdEEE', 'bcEEEEE']

5 random strings for Regular Expression P(Q|R|S)T(UV|W|X)*Z⁺:
['PRTXXXXXZ', 'PQTUVUVUVUVZZZZ', 'PSTXXXXZZ', 'PSTXXXXXZ', 'PSTXXXXZZZZ']

5 random strings for Regular Expression 1(0,1)*2(3|4)⁵36:
['10024444436', '1000023333336', '10000024444436', '100023333336', '10024444436']
```

```
5 random strings for Regular Expression (a|b)(c|d)E⁺G?:
['bdEEEEE', 'bdEEEEG', 'acEEE', 'adEEEEG', 'bcEEEEG']

5 random strings for Regular Expression P(Q|R|S)T(UV|W|X)*Z⁺:
['PRTXXXXXZZZ', 'PRTUVUVUVUVUVZZ', 'PSTUVUVUVZZZZ', 'PQTXZZZZ', 'PRTUVZ']

5 random strings for Regular Expression 1(0,1)*2(3|4)⁵36:
['1124444436', '1124444436', '111124444436', '10000024444436', '111123333336']
```

```
5 random strings for Regular Expression (a|b)(c|d)E⁺G?:
['bdEEEEEG', 'bcEE', 'adEEEEG', 'bdEEE', 'bcEEEEG']

5 random strings for Regular Expression P(Q|R|S)T(UV|W|X)*Z⁺:
['PRTXXXZZZ', 'PSTWWWZZZ', 'PRTWWWZZZZZ', 'PQTUVUVZZZ', 'PRTWWWZZ']

5 random strings for Regular Expression 1(0,1)*2(3|4)⁵36:
['1124444436', '123333336', '111123333336', '1124444436', '124444436']
```

It is easy to check that every string that was generated belongs to their respective regular expressions. Now, all that remains to be done is the bonus task. For the bonus task we had to create a function that provides the sequence of the processing of the string generation. So, I created a function which generates a random string based on the regular expression given as input, and also provides a step-by-step explenation on how it is done:

```python
# Function to generate a string with a step-by-step explanation
def generate_string_with_explanation(exp, reg_exp, n, limit):
    string = ''
    step = 1  # Step counter for explanation
    print(f'\nGenerating a string for Regular Expression {exp}:\n')
    print(f'Step-by-step explenation:')

    for tup in reg_exp:
        l = len(tup[1])  # Get the number of options for the current component
        if tup[0] == '1':
            string += random.choice(tup[1])
            if l == 1:
                print(f'Step {step}. Append 1 instance of the symbol "{tup[1][0]}". String = {string}')
                step += 1
            else:
                print(f'Step {step}. Append 1 instance of one of these symbols - {", ".join(tup[1])}. String = {string}')
                step += 1
        elif tup[0] == '?':
            nr = random.randint(0, 1)
            print(f'Step {step}. Generate a random number between 0 and 1 to determine if a symbol is going to be appended. Generated number = {nr}')
            step += 1
            if nr:
                string += random.choice(tup[1])
                if l == 1:
                    print(f'Step {step}. Append 1 instance of the symbol "{tup[1][0]}". String = {string}')
                    step += 1
                else:
                    print(f'Step {step}. Append 1 instance of one of these symbols - {", ".join(tup[1])}. String = {string}')
                    step += 1
            else:
                print(f'Step {step}. Nothing is being appended. String = {string}')
                step += 1
```

```python
        elif tup[0] == '+':
            char = random.choice(tup[1])
            nr_of_chars = random.randint(1, limit)
            print(f"Step {step}. Generate how many symbols will be appended between 1 and the limit, which is {limit}. In this case we append
            {nr_of_chars} symbols")
            step += 1
            for j in range(nr_of_chars):
                string += char
            if l == 1:
                print(f'Step {step}. Append {nr_of_chars} instances of the symbol "{tup[1][0]}". String = {string}')
                step += 1
            else:
                print(f'Step {step}. Append {nr_of_chars} instances of one of these symbols - {", ".join(tup[1])}. String = {string}')
                step += 1
        elif tup[0] == '*':
            char = random.choice(tup[1])
            nr_of_chars = random.randint(0, limit)
            print(f'Step {step}. Generate how many symbols will be appended between 0 and the limit, which is {limit}. In this case we append
            {nr_of_chars} symbols')
            step += 1
            for j in range(nr_of_chars):
                string += char
            if l == 1:
                print(f'Step {step}. Append {nr_of_chars} instances of the symbol "{tup[1][0]}". String = {string}')
                step += 1
            else:
                print(f'Step {step}. Append {nr_of_chars} instances of one of these symbols - {", ".join(tup[1])}. String = {string}')
                step += 1
        else:
            char = random.choice(tup[1])
            for j in range(int(tup[0])):
                string += char
            if l == 1:
                print(f'Step {step}. Append {int(tup[0])} instances of the symbol "{tup[1][0]}". String = {string}')
                step += 1
            else:
                print(f'Step {step}. Append {int(tup[0])} instances of one of these symbols - {", ".join(tup[1])}. String = {string}')
                step += 1
```

As you can see, I used the same method of generating strings as in the previous function. I just added *print* functions along the way to showcase each step of doing it. Here is an example of output:

```
Generating a string for Regular Expression P(Q|R|S)T(UV|W|X)*Z*:

Step-by-step explenation:
Step 1. Append 1 instance of the symbol "P". String = P
Step 2. Append 1 instance of one of these symbols - Q, R, S. String = PR
Step 3. Append 1 instance of the symbol "T". String = PRT
Step 4. Generate how many symbols will be appended between 0 and the limit, which is 5. In this case we append 4 symbols
Step 5. Append 4 instances of one of these symbols - UV, W, X. String = PRTXXXX
Step 6. Generate how many symbols will be appended between 1 and the limit, which is 5. In this case we append 2 symbols
Step 7. Append 2 instances of the symbol "Z". String = PRTXXXXZZ

The resulting string is: PRTXXXXZZ
```

CONCLUSION

In conclusion, this laboratory work provided a practical understanding of regular expressions and their application in string generation and processing. By implementing functions to generate strings based on predefined patterns, I explored the versatility and power of regular expressions in automating text analysis and manipulation tasks. Through the step-by-step generation and explanation process, I deepened my comprehension of how different components of regular expressions affect string outcomes, enhancing my ability to construct and interpret regex patterns effectively. This hands-on experience reinforces the importance of regular expressions as a fundamental tool in software development and data processing. Now, as it comes to the difficulties met in this lab, there actually wheren’t any, as it is a pretty straight forward task.