Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work nr. 4
Course: Formal languages and finite automata
Topic: Regular expressions

Elaborated:
st. gr. FAF-221 Revenco Victor

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theory

Regular expressions, often abbreviated as regex or regexp, are sequences of characters that define a search pattern. They are widely used in computer science and programming for manipulating and searching text data. Regular expressions provide a powerful and flexible means of matching patterns within strings of text.

What Are Regular Expressions Used For?

Pattern Matching: Regular expressions are primarily used for matching patterns within text. This could include simple patterns like finding all occurrences of a specific word, or complex patterns like identifying email addresses or validating phone numbers.

Text Manipulation: Regular expressions can be used to modify text by replacing certain patterns with other strings. This is commonly used in text editing applications, data processing tasks, and scripting languages to manipulate textual data efficiently.

Input Validation: Regular expressions are often employed for input validation in software applications. For instance, they can be used to ensure that user input matches a specific format, such as validating email addresses, URLs, or passwords.

Search and Replace: Regular expressions enable advanced search and replace functionality in text editors and programming languages. They allow you to search for patterns within text and replace them with other patterns, providing a powerful tool for text manipulation and transformation.

Parsing: Regular expressions can be used for parsing structured data from text. This includes extracting information from log files, parsing HTML or XML documents, and extracting data from structured text formats.

Lexical Analysis: Regular expressions are fundamental in lexical analysis, which is the process of converting a sequence of characters into a sequence of tokens (e.g., keywords, identifiers, literals) for processing by a compiler or interpreter.

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 lab I chose python because working with Java was a bit too hard on this one. The whole code consists of a big function called generateString. It takes a regex as input at passes through each char in the regex to form a string.

```
elif regex[i] == "(" and regex[regex.index(")", i) + 1] == "+":
    times = random.randint(1, 5)
    for _ in range(times):
        char = random.choice(options(regex[i + 1:regex.index(")", i)]))
        string += char
        print(f"Adding {char} to string - {string}")
    i = regex.index(")", i) + 1
```

in this example we have the case where a char is follow by “+” meaning from 1 occurrence to 5 max. The nr of occurences are randomly chosen. Using a for we add to the string a char. In case we have something similar to (A|B)+, we use the options function which with the random.choice chooses a char (either A or B). Almost all elifs are similar only thing that changes being the nr of occurences, for example if it’s a fixed nr of times, or * or ? and so on. If you want to place in the regex something to a power, the char must be in between parantheses and the power should be in between {}.

```
else:
    string += regex[i]
    print(f"Adding {regex[i]} to string - {string}")
    i += 1
```

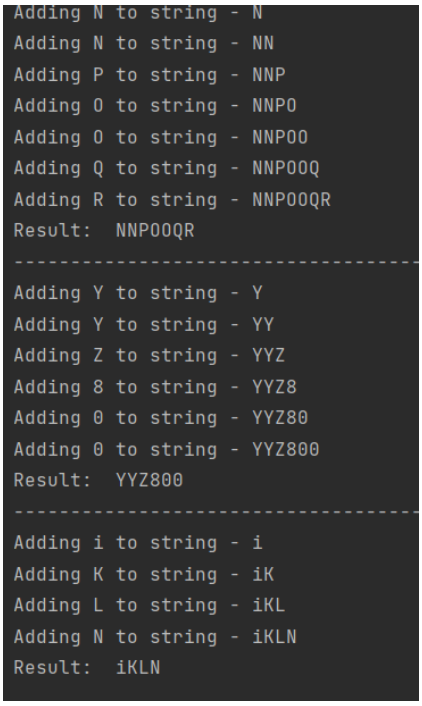
This part of the code is the final else clause in the generateString function. It handles the case where the current character in the regular expression regex is neither a special character ('(', ')', '|', '+', '*', '?', '{') nor part of a quantifier. In this case, the character is simply added to the string that is being generated, and then the index i is incremented to move to the next character in the regex string.

```
def options(sequence):
    return sequence.split("|")
```

This part of the code splits chars that are in between () and split by |. In the code above then its

used to randomly choose from the output.

Screenshots



Conclusions

To sum up, this lab shows how to use Python code to practically comprehend and apply the fundamental components of regular expressions. It offers insight into how regular expressions can be used for text creation and modification jobs by setting rules and using reasoning to generate strings based on those rules.

The solution can be used to generate strings with one or more occurrences, zero or more

occurrences, fixed occurrences, or zero or one occurrence based on predefined choices, among other cases. It also manages the intricacies of nested expressions and randomizes the quantity of occurrences inside predetermined regions.

Although the functionality of regular expressions is streamlined in this implementation, it nevertheless provides an interactive means of understanding the ideas underlying regular expressions and their useful applications in text processing and modification. Potential future improvements could involve extending the functionality to encompass more intricate regular expression capabilities and streamlining the code to improve readability and efficiency.