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 Berco Andrei

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 implementation I chose to use Python, because it is a familiar language.

The implementation consists of a function which takes the given rule and travers it symbol by symbol and checking if it is a special one or no. In this function are covered some base cases, like 1 or more occurrences or a fixed number. Also in each case is printed the step, like how the string which is generated is modified

Some examples of covered cases in code are:

1. 1 or more occurrences from options

```
        elif rule[i] == "(" and rule[rule.index(")", i) + 1] == "+":
            times = random.randint(1, 5)
            for _ in range(times):
                char =  choice(options(rule[i + 1:rule.index(")", i)]))
                string +=  char
                print(f"One or more occurrences from options: Adding {char}
to string => {string}")
            i = rule.index(")", i) + 1
```

The code checks if the current character is an opening parenthesis "(" and if the next character after the closing parenthesis ")" is a plus sign "+". This condition suggests that there should be one or more occurrences of characters/options enclosed within the parentheses.

It generates a random integer between 1 and 5, inclusive, indicating the number of times the characters/options within the parentheses will be repeated.

A loop is initiated that iterates a number of times equal to the randomly chosen value (stored in the variable times). This ensures that the characters/options within the parentheses are added to the output string the specified number of times.

Within the loop, a random option is chosen from the characters/options enclosed within the parentheses, and it is added to the output string.

2. Fixed occurrences from options

```
        elif rule[i] == "(" and rule[rule.index(")", i) + 1] == "{":
            for _ in range(int(rule[rule.index("{", i) + 1])):
                char = choice(options(rule[i+1:rule.index(")", i)]))
                string += char
                print(f"Fixed occurrences from options: Adding {char} to
```

```
string => {string}")
            i = rule.index("}", i) + 1
```

This code block checks if the current character is an opening parenthesis "(" and if the next character after the closing parenthesis ")" is an opening brace "{". This condition indicates that there's a specified fixed number of occurrences for characters/options enclosed within the parentheses. It retrieves the fixed number of occurrences specified within the braces by converting the substring following the opening brace "{" to an integer. A loop is initiated that iterates a number of times equal to the fixed number of occurrences obtained in the previous step. This ensures that the characters/options within the parentheses are added to the output string the specified number of times. Within the loop, a random option is chosen from the characters/options enclosed within the parentheses, and it is added to the output string. After the loop completes, the index "i" is updated to the position of the closing brace "}". This ensures that the main loop continues parsing the rule string from the correct position after processing the fixed occurrences.

Other cases are 0 or more occurrences, 0 or 1 occurrence or just one. The code for these cases can be found on GitHub, because it is pretty similar with the above one.

Next comes the main function with the rules from 3rd variant:

```
rule1 = "O(P|Q|R)+2(3|4)"
print('Final string: ',generateString(rule1))
print('-'*70)

rule2 = "A*B(C|D|E)F(G|H|i){"+"2}"
print('Final string: ',generateString(rule2))
print('-'*70)

rule3 = "J+K(L|M|N)*O?(P|Q){"+"3}"
print('Final string: ',generateString(rule3))
```

rule1 = "O(P|Q|R)+2(3|4)": This line defines the first rule as a string where "O" is followed by one or more occurrences of either "P", "Q", or "R", followed by the literal "2" and one occurrence of either "3" or "4".

print('Final string: ',generateString(rule1)): This line calls the generateString function with the first

rule rule1 as an argument and prints the resulting string along with the label "Final string".

rule2 = "A*B(C|D|E)F(G|H|i){"+"2}": Here, the second rule is defined as a string where "A" occurs zero or more times, followed by one occurrence of either "B", "C", or "D", followed by "F", followed by one occurrence of either "G", "H", or "i" repeated exactly twice.

print('Final string: ',generateString(rule2)): This line calls the generateString function with the second rule rule2 as an argument and prints the resulting string.

rule3 = "J+K(L|M|N)*O?(P|Q){"+"3}": This line defines the third rule as a string where "J" occurs one or more times, followed by "K", followed by zero or more occurrences of either "L", "M", or "N", followed by zero or one occurrence of "O", followed by either "P" or "Q" repeated exactly three times.

print('Final string: ',generateString(rule3)): Finally, this line calls the generateString function with the third rule rule3 as an argument and prints the resulting string.

Screenshots

```
Adding O to string => O
One or more occurrences from options: Adding P to string => OP
Adding 2 to string => OP2
Just one occurrence from options: Adding 4 to string => OP24
Final string:  OP24
--------------------------------------------------------------------------
Adding A to string => A
Adding B to string => AB
Just one occurrence from options: Adding D to string => ABD
Adding F to string => ABDF
Fixed occurrences from options: Adding i to string => ABDFi
Fixed occurrences from options: Adding i to string => ABDFii
Final string:  ABDFii
--------------------------------------------------------------------------
Adding J to string => J
Adding K to string => JK
Zero or more occurrences from options: Adding M to string => JKM
Zero or more occurrences from options: Adding L to string => JKML
Zero or more occurrences from options: Adding M to string => JKMLM
Zero or more occurrences from options: Adding L to string => JKMLML
Zero or one occurrence: Adding O to string => JKMLMLO
Fixed occurrences from options: Adding P to string => JKMLMLOP
Fixed occurrences from options: Adding Q to string => JKMLMLOPQ
Fixed occurrences from options: Adding P to string => JKMLMLOPQP
Final string:  JKMLMLOPQP
```

Figure 1. Output for all 3 rules

Conclusions

In conclusion, this lab demonstrates a practical approach to understanding and implementing basic elements of regular expressions through Python code. By defining rules and applying logic for generating strings based on those rules, it provides insight into how regular expressions can be utilized for text generation and manipulation tasks.

The implementation covers various scenarios such as generating strings with one or more occurrences, zero or more occurrences, fixed occurrences, or zero or one occurrence from specified options. Additionally, it handles the complexities of nested expressions and randomizing the number of occurrences within defined ranges.

While this implementation provides a simplified version of regular expression functionality, it offers a hands-on way to comprehend the concepts behind regular expressions and their practical applications in text processing and manipulation. Further enhancements could include expanding the functionality to cover more complex regular expression features and optimizing the code for efficiency and readability.