MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII

AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

Departamentul Inginerie Software și Automatică

Mihalachi Mihai FAF-221

Report

Laboratory work no.4

of Formal Languages & Finite Automata

Chișinău – 2024

Theory

A regular expression, often abbreviated as "regex" or "regexp," is a sequence of characters that forms a search pattern. This pattern is used mainly for string matching within text or data. Regular expressions are incredibly powerful tools for searching, manipulating, and validating strings of text according to certain patterns or rules.

Regular expressions consist of normal characters like letters, numbers, and symbols, along with special characters known as metacharacters. Metacharacters have special meanings within regular expressions and allow you to specify rules such as repetition, alternatives, character classes, etc.

For example, the regular expression ^([a-zA-Z0-9_.+-]+)@([a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+)$ is commonly used to validate email addresses.

Regular expressions are supported by most programming languages and text processing tools, such as Python, Perl, Java, JavaScript, and many others. They are widely used in tasks such as data validation, text search and replace, web scraping, and parsing structured data.

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)

Write a good report covering all performed actions and faced difficulties.

Implementation description:

1. Since my number in the list of students is 20, I got to do the 4th variant, but the methods I created for my variant also works for the other ones (I included a check for ‘?’ even though I don’t have it in my variant). Also, to separate the power number from the numbers that are part of the string, I used ^{digit} to represent powers.

2. To generate a string using the given regular expression pattern I use a while to iterate through the characters in the pattern. The first thing I check is parentheses, if there is an opening parenthesis I search for the whole substring that’s inside the parenthesis. After that since there can only be or operation(‘|’) in parentheses I take each option and place them into an array by splitting by checking the ‘|’. Then I randomly choose one of the choices and take it as a variable, because I will then check if there exists an operation that is done with the parentheses. If there exists then the outputted variable is subjected to the operation, otherwise it’s just appended to the result string. Now after the parentheses we check if the character has an operation after it. If it has, then we subject the character to the said operation, if not then we append it to the result string. Finally we return the final string result. Here is the code for this method:

```
def generate_string(pattern: str) -> str:
    result = ""
    i = 0
    while i < len(pattern):
        char = pattern[i]
        if char == '(':
            j = i + 1
            subpattern = ""
            while pattern[j] != ')':
                subpattern += pattern[j]
                j += 1
            choices = subpattern.split('|')
            subpattern = random.choice(choices)
            repeat = 1
            i = j + 1
            if j + 1 < len(pattern):
                match pattern[j + 1]:
                    case '^':
                        if pattern[j + 2].isdigit():
                            repeat = int(pattern[j + 2])
                            i = j + 3
                    case '*':
                        repeat = random.randint(0, 3)
                        i = j + 2
                    case '+':
                        repeat = random.randint(1, 3)
                        i = j + 2
                    case '?':
                        repeat = random.randint(0, 1)
                        i = j + 2
            result += subpattern * repeat
            continue
        next_char = pattern[i + 1] if i + 1 < len(pattern) else ""
        match next_char:
            case '^':
                result += char * int(pattern[i+2])
                i += 3
                continue
            case '*':
                result += char * random.randint(0, 5)
                i += 2
                continue
            case '+':
                result += char * random.randint(1, 5)
                i += 2
                continue
            case '?':
                result += char * random.randint(0, 1)
                i += 2
                continue
            case _:
                result += char
                i += 1
    return result
```

3. For the bonus point I use almost the same premise as for the generating strings method where you check for operations, but instead of appending it to a result string, I store the operations in an array. There is one case where I delete an operation from the array and replace it with its bigger operation(e.g (U|V|W) with (U|V|W)^2). This is happening only to the parentheses operations, because later I use the array to show the appending operations by appending the array elements, and to avoid duplicates (for storing both (U|V|W) and (U|V|W)^2), I just show the 1st operation then it gets replaced in the array with the other one. It will first show the parentheses operations that have an operation outside parentheses, then each operation separately, then each operation with one another consecutively. Here is the code for this method:

```
def show_sequence(pattern: str):
  operations = []
  operation_number = 1
  i = 0
  while i < len(pattern):
    char = pattern[i]
    if char == '(':
        j = i + 1
        subpattern = '('
        while pattern[j] != ')':
            subpattern += pattern[j]
            j += 1
        subpattern += ')'
        operations.append(subpattern)
        if j + 1 >= len(pattern):
            break
        if pattern[j + 1] in "*+?":
            print(f"{operation_number}: {operations[-1]}")
            operations[-1] = subpattern + pattern[j+1]
            operation_number += 1
            i = j + 2
        elif pattern[j + 1] == "^":
            print(f"{operation_number}: {operations[-1]}")
            operations[-1] = subpattern + pattern[j + 1] + pattern[j + 2]
            operation_number += 1
            i = j + 3
        else:
            i = j + 1
        continue
        if i + 1 >= len(pattern):
            operations.append(char)
            i += 1
            continue
        if pattern[i + 1] in "*+?":
            operations.append(char + pattern[i + 1])
            i += 2
        elif pattern[i + 1] == "^":
            operations.append(char + pattern[i + 1] + pattern[i + 2])
            i += 3
        else:
            operations.append(char)
            i += 1
    for i in range(len(operations)):
        if len(operations[i]) > 1:
            print(f"{operation_number}: {operations[i]}")
            operation_number += 1
    operation = operations[0]
    for n in range(1, len(operations)):
        operation += operations[n]
        print(f"{operation_number}: {operation}")
        operation_number += 1
```

Conclusion

In conclusion, the laboratory work focused on constructing a custom regular expression for a specific problem presents an invaluable learning opportunity. This task deepens comprehension of regular expression syntax and metacharacters, fostering a solid understanding of their functionality and applicability in real-world scenarios.

Engaging in this exercise cultivates essential problem-solving skills, requiring students to analyze problem requirements, decompose them into manageable components, and construct regex patterns that effectively fulfill the defined criteria. The iterative nature of crafting regex solutions emphasizes the importance of testing, refining, and iterating upon initial approaches until desired outcomes are achieved.