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
Variant-3

Elaborated:
st. gr. FAF-223 Pereteatcu Arina

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2024

Theoretical considerations

The most basic machine that can identify patterns is a finite automata (FA). This is employed to describe a Regular Language. It is also employed in the analysis and identification of natural language expressions. An abstract machine with five elements, or tuples, is known as a finite automata or finite state machine. It has a set of states and rules for changing between them, but how it does so is dependent on the input symbol that is applied. The input string can be allowed or refused based on the states and rules. In essence, it's an abstract representation of a digital computer that reads an input string and modifies its internal state in response to the input symbol that's being read at that moment. Each automaton has a language, or a set of strings that it can understand. A finite or infinite collection of strings over a finite number of symbols is the definition of a formal language in computer science. An 'alphabet' is a finite set of symbols. The formal language is made up of the ordered strings that are produced with this alphabet according to the established grammar rules.

Regular Expression is a pattern used to match character combinations in strings.

It can be constructed in 2 ways:

-using a regular expression literal, which consists of a pattern enclosed between slashes "//"

-calling the constructor function of the regular expression object

When the script loads, regular expression literals enable compilation of the regular expression. Using this can enhance performance as long as the regular expression stays consistent. The regular expression can be compiled at runtime by using the constructor function. When you are certain that the regular expression pattern will change, or when you are obtaining the pattern from an external source—such as user input—or when you are unsure about the pattern, utilize the constructor function.

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).

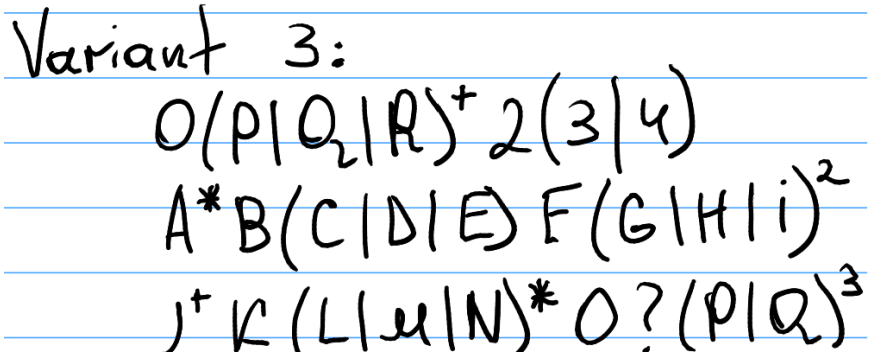
b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

c. Bonus point: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

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

Implementation Description

My laboratory work was done in Python language. I wrote code for the variant 3.



Examples of what must be generated:

{OPP23, OQQQQ24, ...} {AAABCFGG, AAAAAABDFHH, ...} {JJKLOPPP, JKNQQQ, ...}

Code

```
import random

# Function to generate a string that matches the regex pattern: 'O(P|Q|R)^+
2(3|4)'
def generate_string_regex_1():
    generated_string = ''.join('O' + random.choice(['P', 'Q', 'R']) *
random.randint(1, 5) + ' 2' + random.choice(['3', '4']))
    return generated_string

# Function to generate a string that matches the regex pattern: 'A*B(C|D|E)
E(G|H|i)^2'
def generate_string_regex_2():
    generated_string = ''.join('A' * random.randint(0, 5) + 'B' *
random.randint(0, 5) + random.choice(['C', 'D', 'E']) + ' E' +
random.choice(['G', 'H', 'i']) * 2)
    return generated_string

# Function to generate a string that matches the regex pattern:
'J^+K(L|M|N)*O? (P|Q)^3'
def generate_string_regex_3():
    generated_string = ''.join('J' * random.randint(1, 5) + 'K' +
random.choice(['L', 'M', 'N']) * random.randint(0, 5) + 'O' *
random.randint(0, 1) + ' ' + random.choice(['P', 'Q']) * 3)
    return generated_string
```

The function routines offered effectively produces texts that match particular regex patterns. These routines generate a variety of legitimate examples by using random selection and

repetition according to the specified rules of each pattern. These produced strings are helpful test data for validation and regex pattern matching algorithms.

```
def show_processing_sequence(regex_pattern):
    sequence = []
    current_group = ''
    brackets = False

    for char in regex_pattern:
        if char == '[':
            brackets = True
            current_group = ''
        elif char == ']':
            brackets = False
            sequence.append(f"Match one element from '{current_group}' list")
            current_group = ''
        elif char == '*':
            sequence.append("Match zero or more occurrences of the preceding
element")
        elif char == '+':
            sequence.append("Match one or more occurrences of the preceding
element")
        elif char == '{':
            sequence.append("Specify a range of occurrences")
        elif char == '}':
            sequence.append("End of range specification")
        elif char == '(':
            sequence.append("Start of a group")
        elif char == ')':
            sequence.append("End of a group")
        elif char.isalnum() and not(brackets):
            sequence.append(f"Match '{char}'")
        else:
            if brackets:
                current_group += char

    return sequence
```

The *show_processing_sequence* function takes a regex pattern as input and returns a list of explanations describing the processing sequence of that regex pattern. It iterates over each character of the pattern and adds explanation for each character based on the type. If the character is a letter, it adds an explanation that it matches that specific character. If it's a symbol as "|" , it adds an explanation that it matches one of the alternatives.

```
# Examples of generating and explaining regex patterns
regex_patterns = [r'O(P|Q|R)^+ 2(3|4)', r'A*B(C|D|E) E(G|H|i)^2',
r'J^+K(L|M|N)*O? (P|Q)^3']
generators = [generate_string_regex_1, generate_string_regex_2,
generate_string_regex_3]

for regex, generator in zip(regex_patterns, generators):
    generated_str = generator()
    print(f"String matching the regex '{regex}': {generated_str}")
    sequence = show_processing_sequence(regex)
    for step, explanation in enumerate(sequence, start=1):
        print(f"Phase {step}: {explanation}")
    print()
```

The supplied code provides a thorough explanation of the processing sequence for each pattern and effectively produces strings that match predefined regex patterns. Users can learn how these patterns are processed and used in string matching by matching the generated strings with their corresponding regex patterns and processing explanations. This method improves the development and validation procedures for string manipulation activities by making it easier to comprehend and test regex patterns.

Outputs

```
String matching the regex 'O(P|Q|R)^+ 2(3|4)': OPPPPP 23
Phase 1: Match 'O'
Phase 2: Start of a group
Phase 3: Match 'P'
Phase 4: Match 'Q'
Phase 5: Match 'R'
Phase 6: End of a group
Phase 7: Match one or more occurrences of the preceding element
Phase 8: Match '2'
Phase 9: Start of a group
Phase 10: Match '3'
Phase 11: Match '4'
Phase 12: End of a group
```

```
String matching the regex 'A*B(C|D|E) E(G|H|i)^2': AAAAABBBBE EHH
Phase 1: Match 'A'
Phase 2: Match zero or more occurrences of the preceding element
Phase 3: Match 'B'
Phase 4: Start of a group
Phase 5: Match 'C'
Phase 6: Match 'D'
Phase 7: Match 'E'
Phase 8: End of a group
Phase 9: Match 'E'
Phase 10: Start of a group
Phase 11: Match 'G'
Phase 12: Match 'H'
Phase 13: Match 'i'
Phase 14: End of a group
Phase 15: Match '2'
```

```
String matching the regex 'J^+K(L|M|N)*O? (P|Q)^3': JJJJJKLLLLO PPP
Phase 1: Match 'J'
Phase 2: Match one or more occurrences of the preceding element
Phase 3: Match 'K'
Phase 4: Start of a group
Phase 5: Match 'L'
Phase 6: Match 'M'
Phase 7: Match 'N'
Phase 8: End of a group
Phase 9: Match zero or more occurrences of the preceding element
Phase 10: Match 'O'
Phase 11: Start of a group
Phase 12: Match 'P'
Phase 13: Match 'Q'
Phase 14: End of a group
Phase 15: Match '3'
```

The output includes the processing sequence for each pattern as well as strings that match the given regex patterns.

It produces 'OPPPPP 23' for the first text that matches the regex 'O(P|Q|R)^+ 2(3|4)'. According to the processing sequence, it begins with "O," goes through one or more instances of "P," "Q," or "R," then moves on to "2," and lastly either "3" or "4".

The second string yields 'AAAAABBBBE EHH' since it fits the regex 'A*B(C|D|E) E(G|H|i)^2'. The processing order shows how it can handle the following: zero or more instances of 'A', zero or more instances of 'B', either 'C', 'D', or 'E', 'E', and two instances of either 'G', 'H', or 'i'.
The third string yields 'JJJJJKLLLLO PPP' since it matches the regex 'J^+K(L|M|N)*O? (P|Q)^3'. One or more instances of 'J', 'K', zero or more instances of 'L', 'M', or 'N', an optional 'O', a space, and ultimately three instances of either 'P' or 'Q' are included in the processing

sequence.

Overall, this output illustrates how the strings that are produced match the specified regex patterns and provides an explanation of the sequential steps that were taken to accomplish this matching.

Conclusions

To sum up, this laboratory work has given a comprehensive understanding of finite automata and regular expressions, or regex. I've learned a lot about pattern matching algorithms and their useful applications by writing code to produce valid symbol combinations that follow intricate regex patterns and by developing functions that show the processing order of these patterns.

Understanding regular expressions has given us access to a useful tool for manipulating and recognizing patterns inside strings. Text parsing, data validation, and natural language processing are just a few of the domains that profit from the flexible and accurate string processing made possible by the efficient specification of search patterns using constructor functions or regex literals. Furthermore, studying finite automata has improved our understanding of formal language theory and its applications to computing. Finite automata are abstract models of computation that move between states dependent on input symbols, mimicking the behavior of digital computers. This knowledge is essential for creating effective algorithms for problems involving language processing and pattern recognition.

Overall, this lab activity has improved our ability to deal with finite automata and regular expressions while also highlighting their importance in computer science and related fields. Through the accomplishment of the given goals and challenges, we have strengthened our understanding and proficiency with pattern matching algorithms, opening the door for additional research and practical implementation.