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).

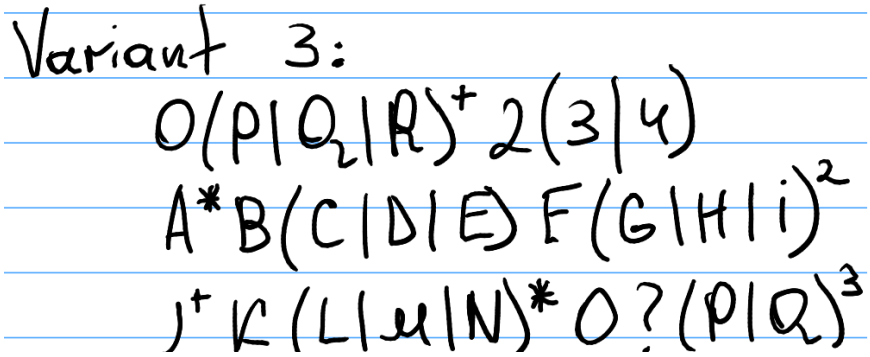
b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

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

```python
import random
# Function to generate a string that matches the given regex pattern
def generate_string_regex(regex_pattern):
    generated_string = ''
    for char in regex_pattern:
        if char.isalpha():
            generated_string += char
        elif char == '|':
            generated_string += random.choice(char)
        elif char == '(' or char == ')' or char == '?' or char == '^' or char == '+' or char == '{' or char == '}' or char == '*':
            continue
        elif char.isdigit():
            for _ in range(int(char)):
                generated_string += random.choice('0123456789')
    return generated_string
```

The *generate_string_regex* function takes a regex pattern as input and generates a string. It iterates over each character of the regex and generates the string based on the character's type.

```
# Function to explain the regex pattern processing sequence
1 usage
def show_processing_sequence(regex_pattern):
    sequence = []
    for char in regex_pattern:
        if char.isalpha():
            sequence.append(f"Match '{char}'")
        elif char == '|':
            sequence.append("Match one of the alternatives")
        elif char == '(':
            sequence.append("Start of a group")
        elif char == ')':
            sequence.append("End of a group")
        elif char == '?':
            sequence.append("Match zero or one occurrence of the preceding element")
        elif char == '^':
            sequence.append("Start of the line")
        elif char == '+':
            sequence.append("Match one or more occurrences of the preceding element")
        elif char == '*':
            sequence.append("Match zero or more occurrences of the preceding element")
        elif char == '{':
            sequence.append("Start of a range")
        elif char == '}':
            sequence.append("End of a range")
    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.

```
# Function to generate and explain regex patterns
1 usage
def generate_and_explain(regex_patterns):
    for regex in regex_patterns:
        generated_str = generate_string_regex(regex)
        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 function "generate_and_explain" iterates over each regex and prints both the generated string and the processing sequence explanations. It takes a list of regex patterns as input, uses "generate_string_regex" to generate a string, and "show_processing_sequence" to explain the processing sequence of patterns.

```
# List of 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'
]

# Generate and explain regex patterns
generate_and_explain(regex_patterns)
```

The list *"regex_patterns"* contains a list of example regex and the function *"generate_and_explain"* function is called.

Outputs

```
String matching the regex 'O(P|Q|R)^+ 2(3|4)': OP|Q|R78802|4596
Phase 1: Match 'O'
Phase 2: Start of a group
Phase 3: Match 'P'
Phase 4: Match one of the alternatives
Phase 5: Match 'Q'
Phase 6: Match one of the alternatives
Phase 7: Match 'R'
Phase 8: End of a group
Phase 9: Start of the line
Phase 10: Match one or more occurrences of the preceding element
Phase 11: Start of a group
Phase 12: Match one of the alternatives
Phase 13: End of a group
```

```
String matching the regex 'A*B(C|D|E) E(G|H|i)^2': ABC|D|EEG|H|i03
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 one of the alternatives
Phase 7: Match 'D'
Phase 8: Match one of the alternatives
Phase 9: Match 'E'
Phase 10: End of a group
Phase 11: Match 'E'
Phase 12: Start of a group
Phase 13: Match 'G'
Phase 14: Match one of the alternatives
Phase 15: Match 'H'
Phase 16: Match one of the alternatives
Phase 17: Match 'i'
Phase 18: End of a group
Phase 19: Start of the line
```

```
String matching the regex 'J^+K(L|M|N)*O? (P|Q)^3': JKL|M|NOP|Q917
Phase 1: Match 'J'
Phase 2: Start of the line
Phase 3: Match one or more occurrences of the preceding element
Phase 4: Match 'K'
Phase 5: Start of a group
Phase 6: Match 'L'
Phase 7: Match one of the alternatives
Phase 8: Match 'M'
Phase 9: Match one of the alternatives
Phase 10: Match 'N'
Phase 11: End of a group
Phase 12: Match zero or more occurrences of the preceding element
Phase 13: Match 'O'
Phase 14: Match zero or one occurrence of the preceding element
Phase 15: Start of a group
Phase 16: Match 'P'
Phase 17: Match one of the alternatives
Phase 18: Match 'Q'
Phase 19: End of a group
Phase 20: Start of the line
```

The generated strings that fit the supplied regex patterns are shown in this output, along with the order in which each regex pattern was processed. The regex pattern for which the string is formed is shown in the first line. A string that follows the pattern follows next. Subsequently, every line represents a stage in the processing order:

Phases 1-3: For the first regex pattern, the string starts with 'O', followed by a group that matches 'P', 'Q', or 'R'. Then, '^+' indicates the start of the line and one or more occurrences of the preceding element, which is the group (P|Q|R).

Phases 4-13: For the second regex pattern, the string starts with zero or more occurrences of 'A', followed by 'B'. Then, a group matches 'C', 'D', or 'E'. After that, a space is followed by 'E', and another group matches 'G', 'H', or 'i'. '^' indicates the start of the line.

Phases 14-20: For the third regex pattern, the string starts with one or more occurrences of 'J', followed by 'K'. Then, there's an optional group that matches 'L', 'M', or 'N'. After that, 'O' is optional. Finally, there's a group matching 'P' or 'Q'. '^' indicates the start of the line again.

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.