Laboratory Work Nr.4 Regular Expressions

Cretu Cristian

April 8, 2024

Theory

Regular expressions, often abbreviated as regex or regexp, are powerful tools used in computer science and programming for pattern matching within strings of text. They provide a concise and flexible means of searching, extracting, and manipulating textual data based on specific patterns. Utilized across various programming languages and text processing utilities, regular expressions enable developers to perform tasks such as validation of input, searching for specific patterns within large datasets, and text manipulation with precision and efficiency. The syntax of regular expressions consists of a combination of literal characters and metacharacters, forming patterns that define the desired matches. With their versatility and widespread adoption, regular expressions serve as an indispensable tool for tasks ranging from simple string manipulation to complex data extraction and transformation. However, mastering regular expressions requires understanding their syntax, metacharacters, and application-specific nuances to leverage their full potential effectively.

Objectives

1. Write and cover what regular expressions are, what they are used for.
2. Take your variant code
3. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown).
4. 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).
5. Bonus point: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

Implementation description

RegEx Operators

I have created custom functions for handling different Regex operators, particularly the $+$, $|$, $*$,$\hat{}$ operators.

For example. here is the OR operator:

```
def or_operator(options: list[str] = None):
    symbol = random.choice(options)
    print(f'Choose {symbol}')
    return symbol
```

Here is the Star Operator

```
def star_operator(symbol: str):
    times = random.randint(0, infty)
    answer = symbol*times
    print(f'Repeat {symbol} {times} times')
    return answer
```

After defining these functions, we have to somehow parse the Regex notation to be able to perform these above mentioned operations on the rules specified. To do that, we search the notation for paranthesis so that we can find groups of symbols to choose from, and other symbols are represented as simple strings.

```
def parse_regex(regex):
    operators = ['+', '*', '?', '^']  # List of operators
    parentheses_stack = []
    parts = []
    current_part = ''
    for char in regex:
        if char == '(':
            if current_part:
                parts.append(current_part)
                current_part = ''
            parentheses_stack.append(len(parts))
        elif char == ')':
            if current_part:
                parts.append(current_part)
                current_part = ''
            start = parentheses_stack.pop()
            parts[start:] = [''.join(parts[start:])]
        elif char in operators:
            if current_part:
                parts.append(current_part)
                current_part = ''
            parts.append(char)
        else:
            current_part += char
    if current_part:
        parts.append(current_part)
```

```
    return parts
```

After that, we can proceed to traverse the parsed regex and perform the operations step by step to obtain the desired string. This function will take a part of the parsed Regex by one, and perform the necessary operations on it, and append them one over one to the final string obtained by the Regex.

```
def compute_regex(parsed_regex: list[str] = None):
    if parsed_regex is None:
        return None
    indices_to_skip = []
    answer = ""
    skipped_simbols = ["(", ")", "*", "+", "?", "^"]
    for i in range(len(parsed_regex)):
        if i in indices_to_skip:
            continue
        part = parsed_regex[i]
        if part not in skipped_simbols:
            if "|" in part:
                current_token = or_operator(part.split("|"))
            else:
                current_token = part
                print(f"Current Token: {current_token}")
            if i+1 < len(parsed_regex):
                next_part = parsed_regex[i+1]
                if next_part == "*":
                    answer += star_operator(current_token)
                elif next_part == "+":
                    answer += plus_operator(current_token)
                elif next_part == "?":
                    answer += question_operator(
current_token)
                elif next_part == "^":
                    power = int(parsed_regex[i+2])
                    answer += n_operator(current_token,
power)
                    indices_to_skip.append(i+2)
                else:
                    answer += current_token
            else:
                answer += current_token
    return answer
```

This function makes the difference between a group of symbols to choose from and simple symbols, and looks at the next symbol in list to see if there is some operation that is needed to perform. Also, for the power operator, it will select as power the part next to the next part and include it into a list of indices to avoid, so that it would not pass over that symbol again and not append it as a single letter.

Execution

The first Regex's execution would look like this

```
Regex: O(P|Q|R)+2(3|4)
Parsed Regex: ['O', 'P|Q|R', '+', '2', '3|4']
Current Token: O
Choose Q from ['P', 'Q', 'R']
Repeat Q 2 times
Current Token: 2
Choose 3 from ['3', '4']
Computed Regex: OQQ23
```

Conclusions

In conclusion, this laboratory work provided valuable insights into regular expressions, emphasizing their role in pattern matching within text. Practical implementation involved creating custom functions for handling regex operators and generating valid symbol combinations. Exploring operators like *, +, —, ?, and ^ enhanced understanding of pattern definition and repetition control. Future enhancements could include supporting additional regex operators, optimizing code for efficiency, and implementing error handling mechanisms. Overall, this experience deepened comprehension of regex principles and their practical application, setting a foundation for further exploration in future projects.