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).

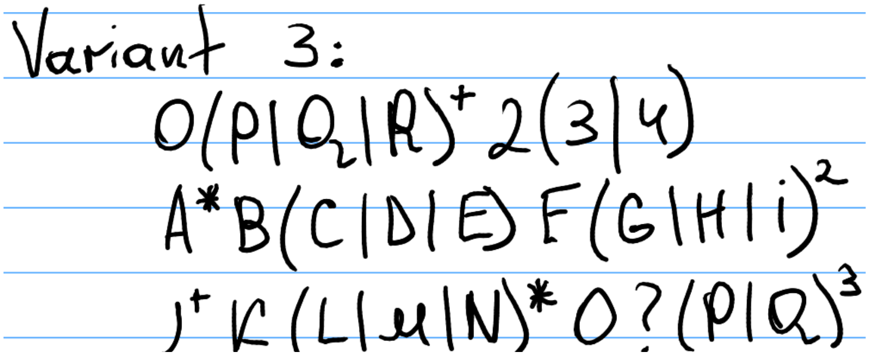
b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

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.



Variant 3:

$O(P|Q|R)^+ 2(3|4)$

$A^* B(C|D|E) F(G|H|i)^2$

$J^+ K(L|M|N)^* O?(P|Q)^3$

Examples of what must be generated:

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

Code

```
import re import
random
 def generate_strings_for_regex(regex_list):
generated_strings_list = []     for regex_list_item in
regex_list:         regex = random.choice(regex_list_item)
print("Processing regex:", regex)         try:
            # Compile the regular expression pattern             pattern =
re.compile(regex)
            # Initialize an empty set to store already generated strings for this regex
            generated_set = set()
            # Initialize an empty list to store valid strings for this regex             generated_strings = []
            # Generate strings until we reach the limit             attempts = 0
while len(generated_strings) < 3 and attempts < 100:
generated_string, steps = generate_string(regex)
                # Check if the generated string matches the regular expression and is not already
generated
                if pattern.fullmatch(generated_string) and generated_string not in generated_set:
generated_set.add(generated_string)
                    generated_strings.append((generated_string, steps))                 attempts += 1
            generated_strings_list.append(generated_strings)         except re.error:
            print("Invalid regex:", regex)     return generated_strings_list
```

The “import re” imports regular expression module in Python, it provides a range of functions and features for working with regular expressions, such as to search for patterns within strings, compiles regular expression pattern into a regex object, replaces occurences of a pattern in a string within another string. This module can also define groups within patterns using parentheses (), supports modifiers that affect how pattern is interpreted, supports character classes for word characters and whitespaces, also quantifiers * , +, ? .

The “import random” imports random module which provides generation of random strings.

The function “generate_strings_for_regex” takes a list of regular expressions as input, it has a loop iterating over each item in the regex_list.

The “while len” loop continues until either 3 valid strings are generated or 100 attempts are made. The if condition checks if the generated string matches the regular expression pattern and is not already generated.

```
  def generate_string(regex):     string = ""     steps = []     i = 0     while i < len(regex):         if
regex[i] == "(" and regex.find(")", i) == len(regex) - 1 or
(regex[i] == "(" and regex[regex.find(")", i) + 1] not in ["*", "+",                     "?", "{"]):
         char = random.choice(options(re.findall(r'     ',                    \((.*?)\)     regex[i:])[0]))
         string += char
         steps.append(f"Adding {char} to string - {string}")           i = regex.find(")", i)
elif regex[i] == "(" and regex[regex.find(")", i) + 1] == "+":
         times = random.randint(1, 5)            for _ in range(times):
            char = random.choice(options(re.findall(r' ', regex[i:])[0]))       \((.*?)\)
            string += char

def options(sequence):     return sequence.split("|")
```

The function “generate_string(regex)” takes a regular expression as input and generates random string that matches the given regular expression.

The code contains while loop to iterate through each character of the regex.

The if condition checks if the current character is an “(“ or *, +, ?, {.

Then a random character is selected from the random choice and the selected character is appended.

The "elif regex” checks if the current character is “(“ and “+”.

Then a random nr of repetitions between 1 and 5. And starts a loop to repeat the process of the randomly determined number of times.

Continuing in a similar fashion, the code updates the generated string and steps based on the handling of more circumstances, including *,?, {}, and single characters. The resulting string and the processing step list are finally returned.

```
regex_list = [
   ["O(P|Q|R)+ 2(3|4)", "O(P|Q|R)+ 2(3|4)", "O(P|Q|R)+ 2(3|4)"],
   ["A*B(C|D|E) F(G|H|i){2}", "A*B(C|D|E) F(G|H|i){2}", "A*B(C|D|E)
F(G|H|i){2}"],
   ["J^+K(L|M|N)*O? (P|Q){3}", "J^+K(L|M|N)*O? (P|Q){3}", "J^+K(L|M|N)*O?
(P|Q){3}"]
]

generated_strings_list = generate_strings_for_regex(regex_list) print("Generated strings for each
regex:") for i, regex_strings in enumerate(generated_strings_list):
   print(f"For regex list {i + 1}:")     for string, steps in regex_strings:         print(f"  Generated
string: {string}")
       print("  Processing steps:")         for step in steps:             print(f"    - {step}")
```

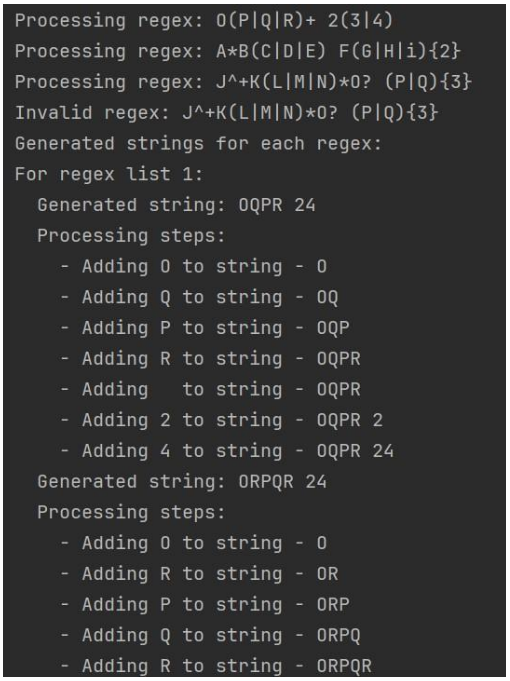
This code generates strings for each regular expression provided in the “regex_list” using the “generate_strings_for_regex” function, and then prints out the generated strings along with their processing steps.

The “def options” is a function that splits the input “sequence” at each occurrence of the “|” and returns a list of the resulting substrings.

The loop iterates over the “generated_strings_list” , which contains the generated strings for each regular expression.

In summary, this code prints the created strings and the steps required to generate them, for every regular expression in the regex_list.

Outputs



```
Processing regex: O(P|Q|R)+ 2(3|4)
Processing regex: A*B(C|D|E) F(G|H|i){2}
Processing regex: J^+K(L|M|N)*O? (P|Q){3}
Invalid regex: J^+K(L|M|N)*O? (P|Q){3}
Generated strings for each regex:
For regex list 1:
  Generated string: OQPR 24
  Processing steps:
    - Adding O to string - O
    - Adding Q to string - OQ
    - Adding P to string - OQP
    - Adding R to string - OQPR
    - Adding   to string - OQPR
    - Adding 2 to string - OQPR 2
    - Adding 4 to string - OQPR 24
  Generated string: ORPQR 24
  Processing steps:
    - Adding O to string - O
    - Adding R to string - OR
    - Adding P to string - ORP
    - Adding Q to string - ORPQ
    - Adding R to string - ORPQR
```

With respect to every regular expression in the supplied regex_list, the output shows the produced strings together with the associated processing stages. A header at the start of each segment states which regex is being processed at that moment. Every erroneous regex is indicated as such. To help in understanding the underlying logic of the string creation process, the created strings for each valid regex are displayed together with the specific steps involved in their generation. This dissection makes it easier to see how each regex affects the creation process overall and provides a clear understanding of how it works.

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.