Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Course: Formal Languages & Finite Automata

Laboratory work 4: Regular Expressions.

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Dumitru Crețu

Chişinău – 2024

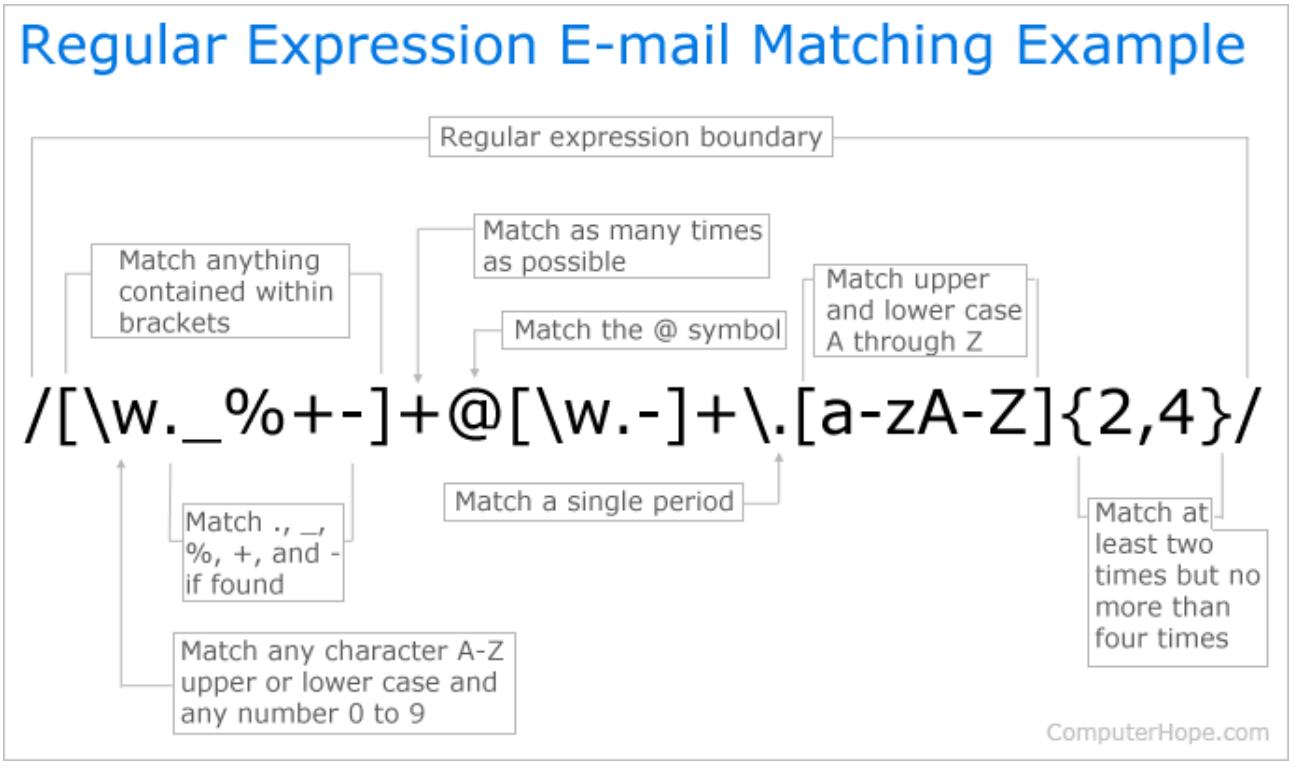
Theory

Regular Expressions

Short for regular expression, a regex is a string of text that lets you create patterns that help match, locate, and manage text. Perl is a great example of a programming language that utilizes regular expressions. However, it's only one of the many places you can find regular expressions. Regular expressions can also be used from the command line and in text editors to find text within a file.

When first trying to understand regular expressions, it seems as if it's a different language. However, mastering regular expressions can save you thousands of hours if you work with text or need to parse large amounts of data.

Regex can be used to search, extract, validate, or transform text based on specified patterns. This pattern can be as basic as searching for a comma (,) in a text or as complex as searching for a valid email address in a text. Below is an example of a regular expression with each of its components labeled.



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

Like my previous labs, I used Python to solve the problems. Starting off, I had the 3rd variant, which had these regular expressions:

- O(P|Q|R)+2(3|4)
- A*B(C|D|E)E(G|H|i)^2
- J+K(L|M|N)*O?(P|Q)^3

I initially started off by making a function that would parse these 3 specifics regular expressions and generate a word based on them (using 5 as the limit for the number of “infinite” iterations).

```
def generate_words_from_regex(nr, limit):
    string = ""
    counter = 0
    if nr == 1:
        string += "O"
        rand = random.random()
        while rand <= 0.25:
            rand = random.random()
        while rand > 0.25 and counter < limit:
            counter += 1
            if rand < 0.5:
                string += "P"
            elif rand < 0.75:
                string += "Q"
            else:
                string += "R"
            rand = random.random()
```

This functions simply takes 2 parameters and outputs a generated string. As my variant had 3 different regex expressions, choosing 1, and a limit of 5 would yield a string that would follow the rules of the first regex, with a maximum limit of repeating characters that can be appended an undefined number of times of 5.

For the second part/bonus point, I made a function that would take any regex and parse it, generating a valid string. We start off with a while loop, where we check the current regex character to see if there's a parenthesis or a regular character.

```
def parse_regular_expression(regex):
    string = ""
    i = 0
    sequence = []
    while i < len(regex):
        current = ""
        if regex[i] == '(':
            sequence.append('(')
            while i < len(regex) and regex[i] != ')':
                current += regex[i]
                i += 1
            if i < len(regex):
                current += regex[i]
            else:
                return "Error: ')' is missing"
            sequence.append(')')
```

If there's no parenthesis and we're not in an expression, it means we currently have a character which we must append to the string. However, we first check if the next character is a special character, meaning we'd have to allow for special behavior instead of appending the character to the string. After we're out of the loop, we return the newly created string along with the sequence of operations to the user.

```
elif regex[i] not in "*+?^":
    if i+1 < len(regex):
        if regex[i+1] not in "*+?^":
            string += regex[i]
            sequence.append(regex[i])
            i += 1
        elif regex[i+1] == "*":
            sequence.append("*")
            rand = random.random()
            while rand > 0.333:
                string += regex[i]
                sequence.append(regex[i])
                rand = random.random()
            i += 1
```

The second functions operates by analyzing for parenthesis first, and then for a "character", which can be as simple as a letter or any symbol that's not in "*+|?^". For characters that can repeat an undefined amount of times, I used a random.random(), with a 1/3 probability of allowing for repeat characters. For the '+' operations, in order to ensure that the character appeared at least once in the string, we made a while loop that would continue until the random number was greater than 0.333, as to ensure that we enter the subsequent loop and append the character to the string.

Results

Input:

```
# O(P|Q|R)+2(3|4)
# A*B(C|D|E)E(G|H|i)^2
# J+K(L|M|N)*O?(P|Q)^3
print(generate_words_from_regex(1, 5))
print(parse_regular_expression("O(P|Q|R)+2(3|4)"))
```

Output:

```
OR24
('OQRPRRQQP24', ['O', '(', ')', '+', 'Q', 'R', 'P', 'R', 'R', 'Q', 'Q', 'P', '2', '(', ')', '4'])
```

Conclusions

In this laboratory work, I learnt about how to implement my own regular expression parser. By making use of special symbols – "()|*+?^", we can create complex patterns that could be used to generate/parse various pieces of text for specific bits of information, which would otherwise take a long time to scan, thus saving resources and time.

While working on this, I didn't encounter any challenges/problems, it was relatively straightforward. After I finished the first part which encompassed my variant's regular expressions, I worked on an "universal" regex parser, which would take any regex that follows the rules defined in the laboratory to generate legal words. Another important part is that I understood how regex works under the hood. I've used it a lot in the past, but I've never really researched about how it works, sifting through the input text to find the patterns and return all sequences which are valid for the pattern.

To conclude, regex is a very important tool, and learning about its implementation taught me a lot about how it works, what it does exactly, and what it should output. Additionally, this knowledge could be useful in creating a domain-specific regex for my ELSD classes, allowing users to make their own patterns.

References:

1. What is a Regex (Regular Expression)? [online]. [accessed 01.04.2024]. Available: https://www.computerhope.com/jargon/r/regex.htm
2. Understanding Regular Expressions (Regex). [online]. [accessed 01.04.2024]. Available: https://medium.com/@victoriousjvictor/understanding-regular-expressions-regex-e1c048f5aa6c