Topic: Regular expressions

Course: Formal Languages & Finite Automata

Author: Durbailo Daniel

Group: FAF-222

Variant: 1

Chisinau 2024

1.Theory

Regular languages and regular expressions are fundamental concepts in computer science, particularly in the field of formal language theory and automata theory. Here's some theory about regular languages and regular expressions framed in the narrative style you provided:

In the vast landscape of computer science, nestled within the realms of formal language theory, there exists a captivating concept known as regular languages. These languages, with their structured simplicity, serve as the bedrock of computational theory, offering a concise yet powerful framework for expressing patterns in strings.

At the heart of regular languages lies the notion of regular expressions—a language of patterns, a lexicon of symbols that bestows upon the programmer the ability to articulate intricate textual compositions with elegance and precision. Like the brushstrokes of a painter on a canvas, regular expressions weave a tapestry of characters, delineating the contours of recognizable patterns within the vast expanse of textual data.

With their arsenal of metacharacters and quantifiers, regular expressions offer a lexicon of syntactic constructs that transcend the mundane constraints of literal interpretation. Anchors tether the expression to the beginning or end of a line, while character classes and ranges provide a palette from which to select specific characters or ranges thereof. Alternation grants the freedom to choose between divergent paths, while repetition operators bestow upon the expression the power of iteration, enabling the recognition of patterns repeated ad infinitum.

Guided by the principles of Kleene's theorem, which postulates the equivalence between finite automata and regular expressions, the regular expression engine embarks on a journey through the labyrinth of characters, navigating the intricate maze of patterns with grace and precision. With each step, it traverses through states, each representing a momentary glimpse into the structure of the text, each transition a subtle revelation of the underlying pattern.

In the realm of lexical analysis, regular expressions serve as the guiding light, the cornerstone upon which lexers are built. Through their judicious application, lexers transform the raw stream of characters into a structured narrative—a symphony of tokens, each imbued with meaning and purpose, each a testament to the power of pattern recognition and formal language theory.

And thus, in the grand tapestry of computational discourse, regular languages and regular expressions stand as pillars of elegance and utility, forever entwined in the annals of computer science, forever revered as the guardians of structure and coherence in the realm of textual data.

Variant 1:

$(a|b)(c|d)E^{+}G?$

$P(Q|R|S)T(U|V|W|X)^{*}Z^{+}$

$1(0|1)^{*}2(3|4)^{5}36$

Variant 2:

$M?N^{2}(O|P)^{3}Q^{*}R^{+}$

$(X|Y|Z)^{3}8^{+}(9|0)$

$(H|i)(J|K)L^{*}N?$

2.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);

3.Implementation:

1.Description:

For the following laboratory work I made a program that generates sets of valid strings based on some regular expressions.

```
def generate_sequences_from_regex(regex, limit=5):
    sequences = ['']
    i = 0
    while i < len(regex):
        if regex[i] == '(':
            j = i
            while regex[j] != ')':
                j += 1
            group = regex[i + 1:j].split('|')
            i = j + 1
            if i < len(regex) and regex[i] in '*+?>':
                quantifier = regex[i]
                i += 1
                if quantifier == '*':
                    group_seqs = ['']
                    for n in range(1, limit + 1):
                        for prod in itertools.product(group, repeat=n):
                            group_seqs.append(''.join(prod))
                    sequences = [s + g for s in sequences for g in group_seqs]
                elif quantifier == '+':
                    group_seqs = []
                    for n in range(1, limit + 1):
                        for prod in itertools.product(group, repeat=n):
                            group_seqs.append(''.join(prod))
```

Figure 1. *The logic for some symbols like '(' , ')', '|', '*'*

```
            sequences = [s + g for s in sequences for g in group_seqs]
        elif quantifier == '?':
            group_seqs = [''] + group
            sequences = [s + g for s in sequences for g in group_seqs]
        elif quantifier == '>':
            x = regex[i:].find('<')
            if x == -1:
                raise ValueError("No matching '<' for power expression.")
            power = int(regex[i:i + x])
            group_seqs = [c * power for c in group]
            sequences = [s + g for s in sequences for g in group_seqs]
            i += x + 1
    else:
        sequences = [s + g for s in sequences for g in group]
else:
    if i < len(regex) - 1 and regex[i + 1:i + 2] in '*+?>':
        char = regex[i]
        quantifier = regex[i + 1]
        i += 2
        if quantifier == '*':
            sequences = [s + char * n for s in sequences for n in range(limit + 1)]
        elif quantifier == '+':
            sequences = [s + char * n for s in sequences for n in range(1, limit + 1)]
        elif quantifier == '?':
```

Figure 2. *The logic for some symbols like ‘?‘ , ‘*+’, ‘?’*

Explanation:

This function, `generate_sequences_from_regex`, takes a regular expression (regex) and an optional limit parameter to generate sequences based on the given regex. It starts by initializing an empty list called `sequences` to store the generated sequences. Another variable, `i`, is set to track the current position within the regex.

As it progresses through the regex, the function examines each character one by one. When it encounters an opening parenthesis '(', it identifies a group within the regex. It then locates the closing parenthesis ')' to extract the characters within that group. If there are multiple options within the group separated by '|', it splits them accordingly.

After parsing the group, the function checks if there's a quantifier ('*', '+', '?', '>') following it. These quantifiers indicate how many times the group should be repeated in the generated sequences. Depending on the quantifier, the function generates sequences accordingly. For example, '*' generates sequences with any number of repetitions up to a specified limit, '+' generates sequences with one or more repetitions, '?' generates sequences with zero or one repetition, and '>' multiplies the characters in the group by a certain power before adding them to the sequences.

If the function encounters a character outside of a group, it checks if there's a quantifier following it. If there is, it generates sequences based on that quantifier. If not, it simply adds the character to all existing sequences.

Once the entire regex is parsed, the function returns the list of generated sequences.

In summary, this function reads a regular expression, understands the groups and quantifiers, and generates sequences based on the rules defined in the regex, combining characters accordingly.

```
def describe_regex_processing(regex):
    description = []
    i = 0
    while i < len(regex):
        if regex[i] == '(':
            j = i
            while regex[j] != ')':
                j += 1
            group = regex[i + 1:j]
            description.append(f"\nProcess group '{group}'")
            i = j + 1
            if i < len(regex) and regex[i] in '*+?':
                quantifier = regex[i]
                description.append(f"with quantifier '{quantifier}'")
                i += 1
            elif i < len(regex) and regex[i] == '>':
                end_power = i + regex[i:].find('<')
                if end_power == -1 or i == len(regex) - 1:
                    raise ValueError("No matching '<' for power expression.")
                power = regex[i+1:end_power]
                description.append(f"with power quantifier repeating previous group/char {power} times")
                i = end_power + 1
        elif regex[i] in '*+?':
```

Figure 3. *The function that describes the process of obtaining the strings*

Explanation:

The `describe_regex_processing` function takes a regular expression (`regex`) as input and generates a text-based description of how it processes the regex.

It begins by initializing an empty list called `description` to store the textual description of the regex processing. Another variable, `i`, is set to track the current position within the regex.

As it traverses the regex, the function examines each character one by one. If it encounters an opening parenthesis '(', it identifies a group within the regex. It then locates the closing parenthesis ')' to extract the characters within that group. The characters within the group are added to the description, indicating that a group is being processed. If there's a quantifier ('*', '+', '?') immediately after the group, it adds that information to the description as well. If the quantifier is a power quantifier ('>'), it finds the end of the power expression and adds that information to the description.

If the function encounters a character outside of a group, it checks if it has a quantifier ('*', '+', '?'). If it does, it adds information about processing that character with the quantifier to the description. If the quantifier is a power quantifier ('>'), it adds information about repeating the previous character a certain number of times based on the power expression.

Once the entire regex is processed, the function joins all the descriptions into a single string and returns it.

In summary, the `describe_regex_processing` function provides a textual overview of how each part of the given regex is handled and processed.

4.Conclusions/Screenshots/Results:

On the screenshots below you can see the results that I obtained for the first 2 variants:

Variant 1:

For the expression No.1 **“(a|b)(c|d)E+G?”** I get:

```
First 10 sequences: ['acE', 'acEG', 'acEE', 'acEEG', 'acEEE', 'acEEEG', 'acEEEE', 'acEEEEG', 'acEEEEE', 'acEEEEEG']

Regex description:
Process group 'a|b'
Process group 'c|d'
Include character 'E'
Process character 'E' with quantifier '+'
Include character 'G'
Process character 'G' with quantifier '?'
```

Figure 4. *The output for the expression* **“(a|b)(c|d)E+G?”**

For the expression No.2 **"(P|Q|R|S)T(U|V|W|X)*Z+"** I get:

```
First 10 sequences: ['PTZ', 'PTZZ', 'PTZZZ', 'PTZZZZ', 'PTZZZZZ', 'PTUZ', 'PTUZZ', 'PTUZZZ', 'PTUZZZZ', 'PTUZZZZZ']

Regex description:
Process group 'P|Q|R|S'
Include character 'T'
Process group 'U|V|W|X' with quantifier '*'
Include character 'Z'
Process character 'Z' with quantifier '+'
```

Figure 5. *The output for the expression* **"(P|Q|R|S)T(U|V|W|X)*Z+"**

For the expression No.3 **"1(0|1)*2(3|4)>5<36"** I get:

```
First 10 sequences: ['123333336', '124444436', '1023333336', '1024444436', '1123333336', '1124444436',

Regex description:
Include character '1'
Process group '0|1' with quantifier '*'
Include character '2'
Process group '3|4' with power quantifier repeating previous group/char 5 times
Include character '3'
Include character '6'
```

Figure 6. *The output for the expression* **"1(0|1)*2(3|4)>5<36"**

Variant 2:

For the expression No.4 **"M?N>2<(01|P)>3<Q*R+"** I get:

```
First 10 sequences: ['NN010101R', 'NN010101RR', 'NN010101RRR', 'NN010101RRRR', 'NN010101RRRRR', 'NN010101QR']

Regex description:
Include character 'M'
Process character 'M' with quantifier '?'
Include character 'N' with power quantifier repeating previous char 2 times
Process group '01|P' with power quantifier repeating previous group/char 3 times
Include character 'Q'
```

Figure 7. *The output for the expression* **"M?N>2<(01|P)>3<Q*R+"**

For the expression No.5 **"(X|Y|Z)>3<8+(9|0)"** I get:

```
First 10 sequences: ['XXX89', 'XXX80', 'XXX889', 'XXX880', 'XXX8889', 'XXX8880', 'XXX88889', 'XXX88880', 'XXX888889', 'X

Regex description:
Process group 'X|Y|Z' with power quantifier repeating previous group/char 3 times
Include character '8'
Process character '8' with quantifier '+'
Process group '9|0'
```

Figure 8. *The output for the expression* **"(X|Y|Z)>3<8+(9|0)"**

For the expression No.6 **"(H|I)(J|K)L*N?"** I get:

```
First 10 sequences: ['HJ', 'HJN', 'HJL', 'HJLN', 'HJLL', 'HJLLN', 'HJLLL', 'HJLLLN', 'HJLLLL', 'HJLLLLN']

Regex description:
Process group 'H|I'
Process group 'J|K'
Include character 'L'
Process character 'L' with quantifier '*'
Include character 'N'
Process character 'N' with quantifier '?'
```

Figure 9. *The output for the expression* **"(H|I)(J|K)L*N?"**

5.Conclusions:

In this lab, I delved into the practical implementation of regular expressions, witnessing firsthand how they breathe life into text processing tasks. Crafting a sequence generator from the ground up provided a tangible insight into translating theoretical concepts into tangible applications.

Navigating through the intricacies of patterns, quantifiers, and various regex components underscored the importance of meticulous attention to detail when constructing tools for language processing. Each element plays a crucial role, emphasizing the significance of thorough testing to ensure seamless functionality.

Concluding this lab, I find it immensely valuable. It has honed my skills and fostered a deeper appreciation for the integration of computer science principles in software development endeavors.

References:

1. https://www.geeksforgeeks.org/write-regular-expressions/
2. https://coderpad.io/blog/development/the-complete-guide-to-regular-expressions-regex/
3. https://docs.python.org/3/library/re.html
4. https://cran.r-project.org/web/packages/stringr/vignettes/regular-expressions.html