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.

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:

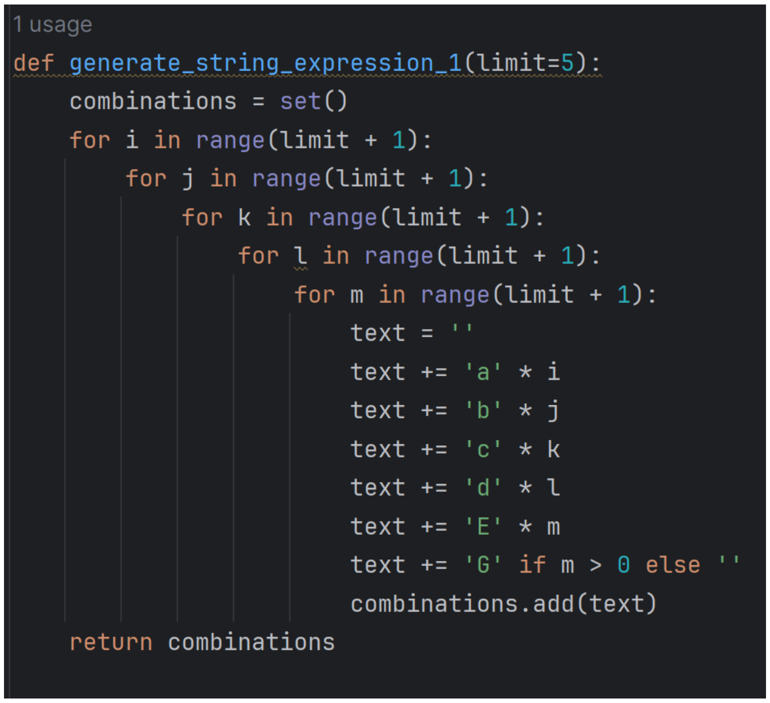
   a. Write a code that will generate valid combinations of symbols conform given regular expressions (examples will be shown).

   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.



```
1 usage
def generate_string_expression_1(limit=5):
    combinations = set()
    for i in range(limit + 1):
        for j in range(limit + 1):
            for k in range(limit + 1):
                for l in range(limit + 1):
                    for m in range(limit + 1):
                        text = ''
                        text += 'a' * i
                        text += 'b' * j
                        text += 'c' * k
                        text += 'd' * l
                        text += 'E' * m
                        text += 'G' if m > 0 else ''
                        combinations.add(text)
    return combinations
```

Figure 1. The logic for the first regular expression

Explanation: Function Definition: This code defines a Python function named generate_string_expression_1 that takes an optional argument limit. This argument specifies the maximum length of each segment of characters in the generated strings.

Looping through Ranges: The function uses nested for loops to generate combinations of characters. These loops iterate over the range from 0 to the limit value (inclusive) for each of the variables i, j, k, l, and m.

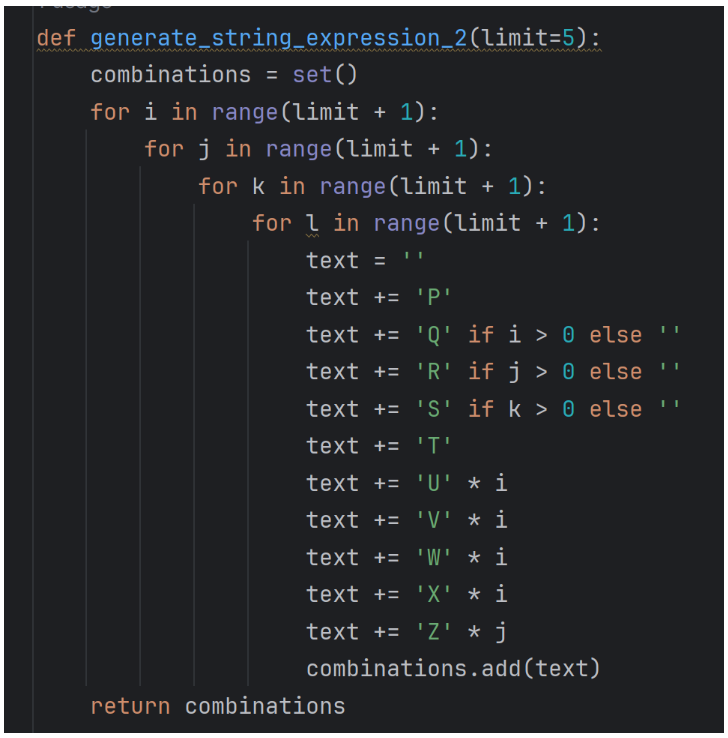
Building the Text: Within the nested loops, the function builds strings (text) by concatenating characters based on the current values of i, j, k, l, and m. Each character is repeated a certain number of times, corresponding to the current value of the loop variable.

Conditions: The logic includes a conditional statement (if) inside the innermost loop. If the current value of m is greater than 0, the character 'G' is added to the string. This condition ensures that the character 'G' is included only if the value of m is greater than 0.

Adding to Set: The generated string (text) is added to a set named combinations. Using a set ensures that each generated string is unique, as sets do not allow duplicate elements.

Return Value: Finally, the function returns the set combinations, containing all unique combinations of characters generated within the specified limits.

In simple terms, this code generates all possible combinations of strings consisting of the characters 'a', 'b', 'c', 'd', 'E', and optionally 'G', within the specified length limits for each character segment. It ensures that the character 'G' is included only if there are repetitions of the character 'E'.



```python
def generate_string_expression_2(limit=5):
    combinations = set()
    for i in range(limit + 1):
        for j in range(limit + 1):
            for k in range(limit + 1):
                for l in range(limit + 1):
                    text = ''
                    text += 'P'
                    text += 'Q' if i > 0 else ''
                    text += 'R' if j > 0 else ''
                    text += 'S' if k > 0 else ''
                    text += 'T'
                    text += 'U' * i
                    text += 'V' * i
                    text += 'W' * i
                    text += 'X' * i
                    text += 'Z' * j
                    combinations.add(text)
    return combinations
```

Figure 2. The logic for the second regular expression

1. Function Definition: This code defines another Python function named `generate_string_expression_2` that takes an optional argument `limit`, just like the previous function.

2. Looping through Ranges: Similar to the previous function, this function also uses nested `for` loops to generate combinations of characters. The loops iterate over the range from 0 to the `limit` value (inclusive) for each of the variables `i`, `j`, `k`, and `l`.

3. Building the Text: Within the nested loops, the function builds strings (`text`) by concatenating characters based on the current values of `i`, `j`, `k`, and `l`. However, in this case, the building process is a bit more complex.

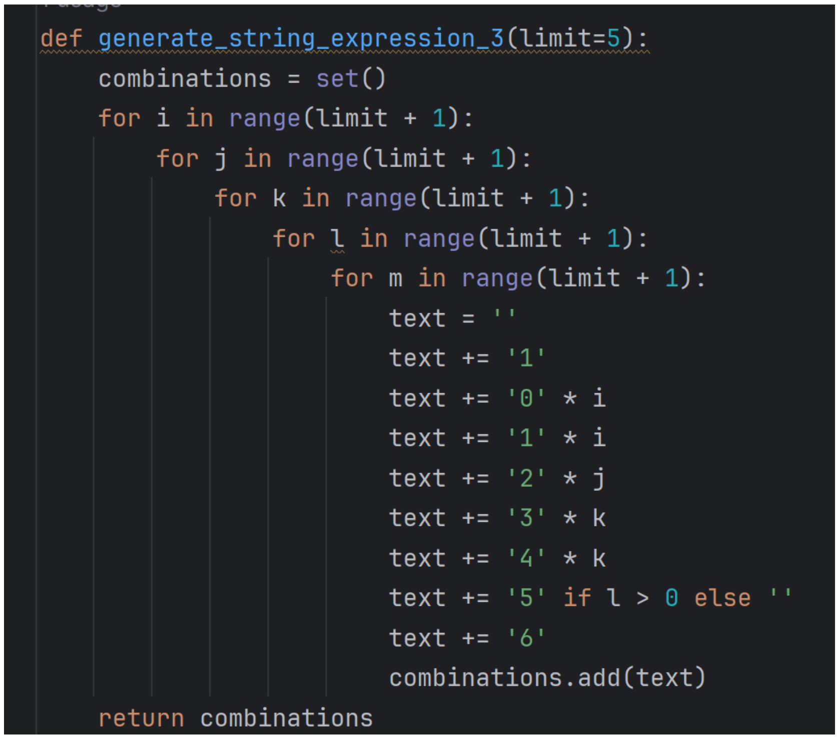
4. Adding Characters: The string `text` starts with the character `'P'`. Then, based on the values of `i`, `j`, `k`, and `l`, additional characters are added to the string. If `i` is greater than 0, the character `'Q'` is added; if `j` is greater than 0, the character `'R'` is added; if `k` is greater than 0, the character `'S'` is added. After that, the character `'T'` is always added to the string.

5. Repetition: Following the addition of specific characters, the string includes repetitions of characters `'U'`, `'V'`, `'W'`, `'X'`, and `'Z'` based on the value of `i` and `j`. For example, if `i` is 3, then `'U'`, `'V'`, `'W'`, and `'X'` will each appear three times in the string.

6. Adding to Set: Similar to the previous function, the generated string (`text`) is added to a set named `combinations`. This ensures uniqueness of strings.

7. Return Value: Finally, the function returns the set `combinations`, containing all unique combinations of characters generated within the specified limits.

In simple terms, this code generates all possible combinations of strings consisting of characters `'P'`, `'Q'`, `'R'`, `'S'`, `'T'`, `'U'`, `'V'`, `'W'`, `'X'`, and `'Z'`, within the specified length limits for each character segment. The presence of certain characters like `'Q'`, `'R'`, and `'S'` depends on whether the corresponding loop variables are greater than 0.



```
def generate_string_expression_3(limit=5):
    combinations = set()
    for i in range(limit + 1):
        for j in range(limit + 1):
            for k in range(limit + 1):
                for l in range(limit + 1):
                    for m in range(limit + 1):
                        text = ''
                        text += '1'
                        text += '0' * i
                        text += '1' * i
                        text += '2' * j
                        text += '3' * k
                        text += '4' * k
                        text += '5' if l > 0 else ''
                        text += '6'
                        combinations.add(text)
    return combinations
```

Figure 3. The logic for the third regular expression

1. Function Definition: Similar to the previous functions, this code defines a Python function named `generate_string_expression_3`. It also takes an optional argument `limit`, indicating the maximum length of each segment of characters in the generated strings.

2. Looping through Ranges: Like before, nested `for` loops are used to iterate over ranges from 0 to the `limit` value (inclusive) for each of the variables `i`, `j`, `k`, `l`, and `m`.

3. Building the Text: Within the nested loops, the function constructs strings (`text`) by concatenating characters based on the current values of `i`, `j`, `k`, `l`, and `m`.

4. Adding Characters: The string `text` begins with the character `'1'`. Then, based on the values of `i`, `j`, `k`, and `l`, additional characters are added.

- The character `'0'` is repeated `i` times.

- The character `'1'` is repeated `i` times.

- The character `'2'` is repeated `j` times.

- The character `'3'` is repeated `k` times.

  - The character `'4'` is also repeated `k` times.
  - If `l` is greater than 0, the character `'5'` is added once.
  - Finally, the character `'6'` is appended to the string.

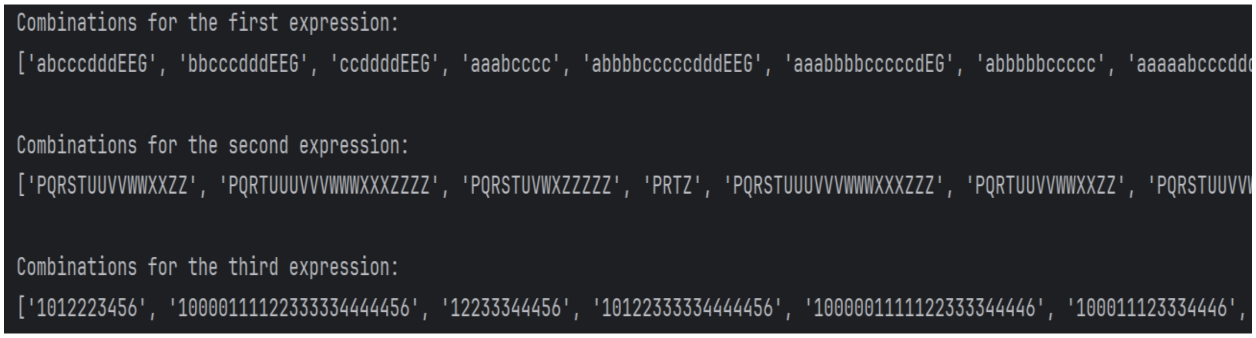
5. Adding to Set: As with the previous functions, the generated string (`text`) is added to a set named `combinations`. This ensures that only unique strings are retained.

6. Return Value: The function returns the set `combinations`, which contains all unique combinations of characters generated within the specified limits.

In simpler terms, this code generates all possible combinations of strings consisting of characters `'0'`, `'1'`, `'2'`, `'3'`, `'4'`, `'5'`, and `'6'`, with specific repetitions of each character based on the values of loop variables `i`, `j`, `k`, `l`, and `m`, within the specified length limits for each character segment. Additionally, the character `'5'` is included in the string only if `l` is greater than 0.

4.Conclusions/Screenshots/Results:

Here you can see the output of my program, 3 sets of outputs, one per each regular expression:



```
Combinations for the first expression:
['abcccdddEEG', 'bbcccdddEEG', 'ccddddEEG', 'aaabcccc', 'abbbbcccccdddEEG', 'aaabbbbcccccdEG', 'abbbbbccccc', 'aaaaabcccddd

Combinations for the second expression:
['PQRSTUUVVWWXXZZ', 'PQRTUUUVVVWWWXXXZZZZ', 'PQRSTUVWXZZZZZ', 'PRTZ', 'PQRSTUUUVVVWWWXXXZZZ', 'PQRTUUVVWWXXZZ', 'PQRSTUUVVW

Combinations for the third expression:
['1012223456', '1000011112233333444456', '12233344456', '10122333334444456', '1000001111122333344446', '100011123334446',
```

5.Conclusions:

Throughout this lab session, I delved into the realm of lexical analysis, a cornerstone in the compilation process. My exploration centered on the intricate world of regular expressions, which serve as the bedrock for establishing syntactic rules in tokenizing programming languages and structured texts.

By creating a lexer from scratch, I translated theoretical concepts into tangible applications, bridging the abstract principles of computer science with practical software development. This hands-on experience not only deepened my understanding of regular expressions but also highlighted their versatility in lexical analysis.

Designing the lexer posed challenges, requiring meticulous attention to detail and comprehensive testing to ensure accurate parsing of complex code into meaningful tokens. This process underscored the importance of rigorous testing and iterative refinement in compiler design.

In summary, this lab session was instrumental in reinforcing my comprehension of lexical analysis and its indispensable role in language processing. It equipped me with valuable knowledge and skills that will undoubtedly be beneficial in my future endeavors in computer science and programming.

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