

Topic: Regular expressions

Course: Formal Languages & Finite Automata

Author: Grosu Renat

Theory:

Regular expressions are a crucial theoretical concept used extensively in computer science, particularly in the fields of text processing, compilers, and data validation. They provide a concise and powerful language for specifying patterns in strings, which allows for string matching, searching, and manipulation tasks. The theory behind regular expressions is rooted in formal language theory and automata theory, two fundamental areas of computer science that study the abstract representation of computational processes and languages.

Basic Theory:

1. **Pattern Matching:** Regular expressions define search patterns for text, which are used to identify strings that match a particular structure or content. They are utilized in various programming languages and tools to perform complex text searches and replacements.
2. **Formal Language Representation:** Regular expressions represent regular languages, which are sets of strings described by the regular expressions' patterns. Regular languages are the simplest class of languages in the Chomsky hierarchy, and they can be recognized by finite automata.
3. **Finite Automata:** In the theoretical model, regular expressions can be compiled into finite automata—either deterministic (DFA) or non-deterministic (NFA). These automata process input strings by moving through a series of states according to transition rules defined by the regular

expression.

4. **Expressions and Operators:** Regular expressions consist of characters, operators, and constructs that define patterns. Common operators include:
 - Concatenation: Sequence of characters to match.
 - Alternation (`|`): Match either of the expressions.
 - Kleene Star (`*`): Match zero or more occurrences.
 - Plus (`+`): Match one or more occurrences.
 - Question Mark (`?`): Match zero or one occurrence.
 - Ranges (`[a-z]`): Match any character within the specified range.
5. **Capturing Groups:** Regular expressions can define groups with parentheses `()`, which can capture the matching strings for extraction or back-referencing.
6. **Greedy and Lazy Matching:** By default, quantifiers in regular expressions are greedy, meaning they match as much text as possible. A lazy (or non-greedy) match, indicated by appending `?` to a quantifier, matches as little text as necessary.
7. **Backtracking:** Many regex engines use a technique called backtracking to find matches. If a pattern fails to match further down the string, the engine backtracks to previous match positions and tries different paths through the pattern.
8. **Lookahead and Lookbehind:** These are zero-length assertions that specify a pattern must or must not follow (lookahead) or precede (lookbehind) a certain point in the string, without including it in the match.

Applications:

- **Text Editors and Utilities:** Regular expressions are commonly used in text editors, search utilities, and command-line tools for tasks like find-and-replace operations and filtering text.
- **Data Validation:** They are used to validate the format of strings, such as email addresses, phone numbers, and other data in user input forms.
- **Programming Languages:** Most programming languages support regular expressions through libraries or built-in features, allowing developers to integrate pattern matching directly into their code.
- **Computational Linguistics:** In natural language processing, regular expressions are used for tasks like tokenization, stemming, and syntactic pattern recognition.

In summary, regular expressions are a powerful tool derived from theoretical computer science that provides a versatile means of text analysis and manipulation. They are foundational to many applications in software development and data processing.

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)

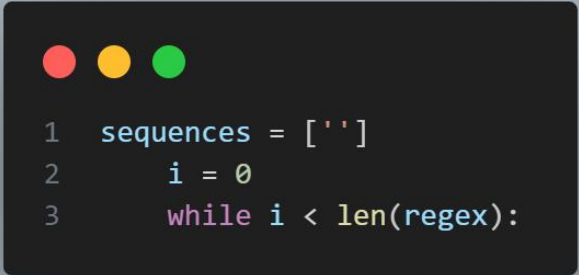
Write a good report covering all performed actions and faced difficulties.

Implementation description:

Imports and Setup:


The script initiates by importing the `itertools` module which is fundamental for creating efficient iterators that aid in the complex pattern generation process used in the sequence generator function.

Sequence Generation Function:




```
1 sequences = []
2     i = 0
3     while i < len(regex):
```

Here we initialize the sequences list, which will hold the final sequences. `i` is used as an index to iterate through the regex pattern.



```
1  if regex[i] == '(':
2      j = i
3      while regex[j] != ')':
4          j += 1
5      group = regex[i + 1:j].split('|')
6      i = j + 1
```

These lines handle groups within the regex. When an opening parenthesis (is encountered, it finds the corresponding closing parenthesis) and creates a list called `group` by splitting the text inside the parentheses using the pipe |, which in regex denotes alternation.



```
1  if i < len(regex) and regex[i] in '*+?>':
2      quantifier = regex[i]
3      i += 1
```

After processing a group, the code checks if the next character is a quantifier (*, +, ?, >) and assigns it to the variable `quantifier`, then increments `i` to move past the quantifier.



```
1  if quantifier == '*':
2      group_seqs = ['']
3      for n in range(1, limit + 1):
4          for prod in itertools.product(group, repeat=n):
5              group_seqs.append(''.join(prod))
6      sequences = [s + g for s in sequences for g in group_seqs]
```

If the quantifier is *, the function creates sequences by repeating the elements of the group from 0 up to the limit number of times using `itertools.product` to generate combinations of group elements.



```
1  elif quantifier == '+':
2      group_seqs = []
3      for n in range(1, limit + 1):
4          for prod in itertools.product(group, repeat=n):
5              group_seqs.append(''.join(prod))
6      sequences = [s + g for s in sequences for g in group_seqs]
```


For the + quantifier, it performs a similar operation as * but ensures that the group is included at least once (i.e., repeating from 1 to limit times).



```
1  elif quantifier == '?':
2      group_seqs = [''] + group
3      sequences = [s + g for s in sequences for g in group_seqs]
```


For the ? quantifier, it adds sequences with and without the group,

indicating that the group is optional.



```
1 elif quantifier == '>':
2     x = regex[i:].find('<')
3     if x == -1:
4         raise ValueError("No matching '<' for power expression.")
5     power = int(regex[i:i + x])
6     group_seqs = [c * power for c in group]
7     sequences = [s + g for s in sequences for g in group_seqs]
8     i += x + 1
```

When encountering $>$, it signifies the start of a power expression, which expects a corresponding $<$. The function extracts the number between $><$, multiplies each element of the group by this number, and generates the sequences accordingly.



```
1 else:
2     sequences = [s + g for s in sequences for g in group]
```

If no quantifier follows the group, it simply adds each element of the group to the sequences as is.

```

1  else:
2      if i < len(regex) - 1 and regex[i + 1:i + 2] in '*+?>':
3          char = regex[i]
4          quantifier = regex[i + 1]
5          i += 2
6          if quantifier == '*':
7              sequences = [s + char * n for s in sequences for n in range(limit + 1)]
8          elif quantifier == '+':
9              sequences = [s + char * n for s in sequences for n in range(1, limit + 1)]
10         elif quantifier == '?':
11             sequences = [s + char * n for s in sequences for n in range(2)]
12         elif quantifier == '>':
13             x = regex[i:].find('<')
14             if x == -1:
15                 raise ValueError("No matching '<' for power expression.")
16             power = int(regex[i:i + x])
17             sequences = [s + char * power for s in sequences]
18             i += 2

```

This else block handles characters outside groups. If the next character is a quantifier, it records the current character as char and the quantifier, then moves the index past them. Rest of the code handles the quantifiers for individual characters similar to how it's done for the groups, including the custom power quantifier denoted by ><.

```


1      else:
2          sequences = [s + regex[i] for s in sequences]
3          i += 1
4  return sequences

```

If the character is not followed by a quantifier, it is added to each of the existing sequences. Finally, the function returns the list of generated sequences.


Regex processing description function:

This function generates a description of how a regex pattern is processed.



```
1 description = []
2 i = 0
3 while i < len(regex):
```

Here, we initialize an empty list called description that will contain parts of the final description. i is used to iterate over the regex string.



```
1 if regex[i] == '(':
2     j = i
3     while regex[j] != ')':
4         j += 1
5     group = regex[i + 1:j]
6     description.append(f"\nProcess group '{group}'")
7     i = j + 1
```

These lines detect a group within parentheses (and). The content inside the parentheses is treated as a group of characters from which one should be selected. The group is then added to the description list with a note that it will be processed as a group.



```
1 if i < len(regex) and regex[i] in '*+?':
2     quantifier = regex[i]
3     description.append(f"with quantifier '{quantifier}'")
4     i += 1
```

After finding a group, this segment checks for a quantifier (*, +, or ?) that applies to that group and appends a description of the quantifier.



```
1 elif i < len(regex) and regex[i] == '>':
2     end_power = i + regex[i:].find('<')
3     if end_power == -1 or i == len(regex) - 1:
4         raise ValueError("No matching '<' for power expression.")
5     power = regex[i+1:end_power]
6     description.append(f"with power quantifier repeating previous group/char {power} times")
7     i = end_power + 1
```

If a > is encountered, it indicates the start of a power quantifier, which requires finding the matching <. The function then describes the power quantifier as repeating the previous group or character a specified number of times.



```
1 elif regex[i] in '*+?':
2     description.append(f"\nProcess character '{regex[i-1]}' with quantifier '{regex[i]}'")
3     i += 1
```

This section deals with quantifiers that follow individual characters outside of groups, appending the character along with its quantifier to the description.

```
1 elif regex[i] == '>':
2     end_power = i + regex[i:].find('<')
3     if end_power == -1 or i == len(regex) - 1:
4         raise ValueError("No matching '<' for power expression.")
5     power = regex[i+1:end_power]
6     description.append(f"with power quantifier repeating previous char {power} times")
7     i = end_power + 1
```

Similarly, this part handles the power quantifier for individual characters. If there's a > without a preceding group, it calculates how many times the character before > should be repeated.

```
1     else:
2         description.append(f"\nInclude character '{regex[i]}'")
3         i += 1
4 return ' '.join(description)
```

For any other character, the function simply notes that the character should be included in the sequence. Finally, all parts of the description are joined into a single string and returned.

Conclusions/Screenshots/Results:

My variant:

Variant 1:

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

$P(Q|R|S)T(UV|W|X)^*Z^+$

$1(0|1)^*2(3|4)^536$

These are the outputs of the functions for the examples from the 1st variant:

Regular expression 1:

```
First 10 sequences: ['acE', 'acEG', 'acEE', 'acEEG', 'acEEE', 'acEEEG', 'acEEEE', 'acEEEEG', 'acEEEEEE', '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 '?'
```

Regular expression 2:

```
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 '+'
```

Regular expression 3:

```
First 10 sequences: ['123333336', '124444436', '1023333336', '1024444436', '1123333336', '1124444436',  
'10023333336', '10024444436', '10123333336', '10124444436']
```

```
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'
```

Through this lab work, I dove into the practical side of regular expressions and really saw how they bring text processing to life. Getting to build a sequence generator from scratch was a hands-on way to see how theory turns into real-world applications.

Dealing with patterns, quantifiers, and all the details of regex taught me a lot about how careful you have to be when you're building tools for language processing. Every little bit counts, and testing each piece is key to making sure everything works right.

Wrapping up, this lab has been super valuable. It's sharpened my skills and given me a deeper appreciation for how we use computer science concepts in software development.

References:

<https://introcs.cs.princeton.edu/java/51language/>

<https://www.geeksforgeeks.org/introduction-of-finite-automata/>