**Regular expressions**

**Course: Formal Languages & Finite Automata**

**Author: Vladlen Mîțu**

**Theory:**

Regular expressions (regex) are patterns used to match and manipulate text. They are commonly used in search operations, input validation, and text processing. A regex consists of literals (e.g., abc) and metacharacters like \*, +, ?, {n}, |, and () to define complex patterns.

* **\* (Star)**: Matches zero or more occurrences.
* **+ (Plus)**: Matches one or more occurrences.
* **? (Question mark)**: Matches zero or one occurrence.
* **{n}**: Matches exactly n occurrences.
* **| (OR operator)**: Matches either pattern.
* **() (Grouping)**: Groups expressions for precedence or capturing.

Regex is widely used in programming languages like Python, Java, and JavaScript, often implemented through libraries like Python’s re module.

**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). Be careful that idea is to interpret the given regular expressions dinamycally, not to hardcode the way it will generate valid strings. You give a set of regexes as input and get valid word as an output

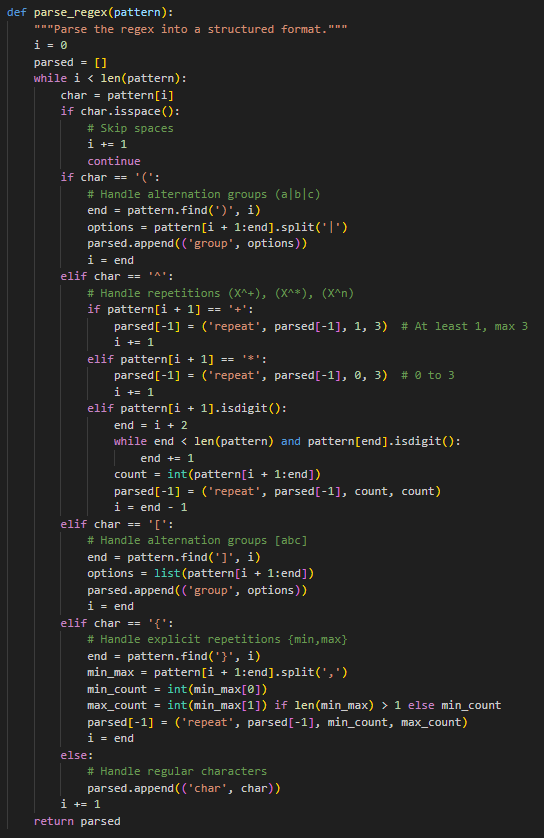
b. In case you have an example, where symbol may be written undefined number of times, take a limit of 5 times (to evade generation of extremely long combinations);

c. Bonus point: write a function that will show sequence of processing regular expression (like, what you do first, second and so on)

**Implementation description:**

The parse\_regex(pattern) function is designed to convert a custom regex-like pattern into a structured format that can be used for string generation. Here's a brief explanation:

1. **Input:**
   * Takes a string pattern representing the regex-like expression (e.g., (S|T)(U|V)W^\*Y^+24).
2. **Output:**
   * Returns a list of tokens, where each token is a tuple:
     + ('char', value): A single character.
     + ('group', options): A group of options (e.g., ['S', 'T']).
     + ('repeat', sub\_token, min\_count, max\_count): A repetition of a sub-token with specified bounds.
3. **Parsing Logic:**
   * Iterates through the pattern character by character.
   * Groups: Handles alternation groups like (S|T) or [abc] by identifying the start (( or [) and end () or ]) and splitting the options.
   * Repetitions:
     + ^+: Matches one or more repetitions.
     + ^\*: Matches zero or more repetitions.
     + ^n: Matches exactly n repetitions.
     + {min,max}: Matches a range of repetitions.
   * Characters: Treats any other character as a literal and adds it to the parsed structure.
   * Spaces: Skips spaces to ensure they are not treated as part of the regex.

****

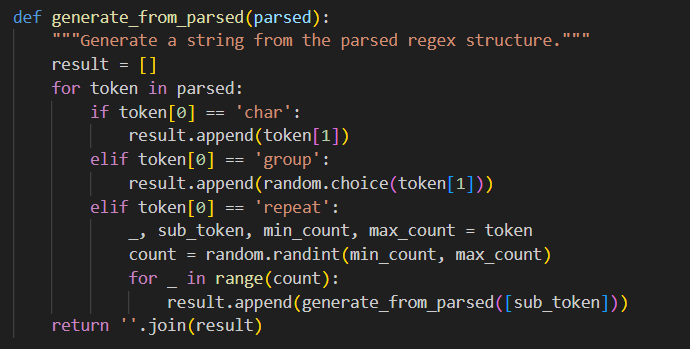
The generate\_from\_parsed(parsed) function generates a string based on the structured format produced by parse\_regex.

**Purpose**

It interprets the parsed tokens (e.g., characters, groups, repetitions) and recursively generates a valid string that matches the given regex pattern.

**How It Works**

1. **Input**:
   * Takes a list of parsed tokens (e.g., [('char', 'S'), ('group', ['U', 'V']), ('repeat', ('char', 'W'), 0, 3)]).
2. **Output**:
   * Returns a string generated by interpreting the tokens (e.g., "SUWW").
3. **Logic**:
   * Iterates through each token in the parsed list:
     + char: Appends the literal character to the result.
     + **group**: Randomly selects one option from the group.
     + **repeat**: Recursively generates the repeated sub-token for a random count within the specified range.
   * Combines all parts into a single string.
4. **Example**:



**generate\_from\_regex(pattern, num\_samples)**

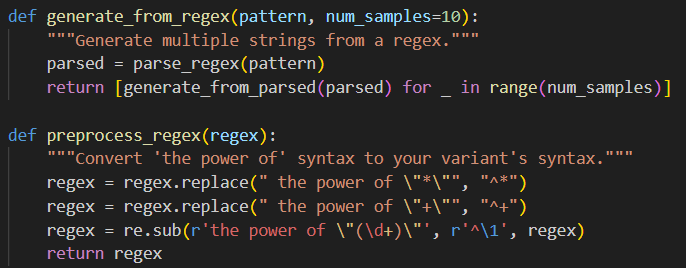
* **Purpose**: Generates multiple strings that match a given regex-like pattern.
* **How It Works**:
  1. Calls parse\_regex to convert the pattern into a structured format (tokens).
  2. Uses generate\_from\_parsed to generate strings based on the parsed tokens.
  3. Repeats the process num\_samples times to produce multiple strings.

**preprocess\_regex(regex)**

**Purpose**: Converts human-readable syntax (e.g., "the power of") into the custom regex-like syntax (e.g., ^\*, ^+, ^n).

**How It Works:**

1. Replaces "the power of \"\*\"" with ^\*.
2. Replaces "the power of \"+\"" with ^+.
3. Converts "the power of \"n\"" into ^n using regex substitution.



**Conclusion:**

The program successfully parses custom regex-like patterns into a structured format using parse\_regex, preprocesses human-readable syntax with preprocess\_regex, and generates valid strings matching the patterns with generate\_from\_parsed. It handles alternation, repetitions, and groups effectively, ensuring flexibility and correctness. The implementation is modular, making it easy to extend or adapt for additional regex features.

**References:**

1. Else Course FAF.LFA21.1
2. Lexer implementation: https://en.wikipedia.org/wiki/Lexical\_analysis
3. Lexical analysis : https://en.wikipedia.org/wiki/Lexical\_analysis