Regular expressions

Course: Formal Languages & Finite Automata

Author: Moraru Gabriel

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.

```python
def parse_regex(pattern):
    i = 0
    parsed = []
    while i < len(pattern):
        char = pattern[i]
        if char.isspace():
            i += 1
            continue
        if char == '(':
            end = pattern.find(')', i)
            options = pattern[i + 1:end].split('|')
            parsed.append(('group', options))
            i = end
        elif char == '^':
            if pattern[i + 1] == '+':
                parsed[-1] = ('repeat', parsed[-1], 1, 3)
                i += 1
            elif pattern[i + 1] == '*':
                parsed[-1] = ('repeat', parsed[-1], 0, 3)
                i += 1
            elif pattern[i + 1].isdigit():
                end = i + 2
                while end < len(pattern) and pattern[end].isdigit():
                    end += 1
                count = int(pattern[i + 1:end])
                parsed[-1] = ('repeat', parsed[-1], count, count)
                i = end - 1
        elif char == '?':
            parsed[-1] = ('repeat', parsed[-1], 0, 1)
        else:
            parsed.append(('char', char))
        i += 1
    return parsed
```

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**:

```
def generate_from_parsed(parsed):
    result = []
    for token in parsed:
        if token[0] == 'char':
            result.append(token[1])
        elif token[0] == 'group':
            result.append(random.choice(token[1]))
        elif token[0] == 'repeat':
            _, sub_token, min_count, max_count = token
            count = random.randint(min_count, max_count)
            for _ in range(count):
                result.append(generate_from_parsed([sub_token]))
    return ''.join(result)
```

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.

```
def main():
    # Separated parts of the full regex
    part1 = "O(P|Q|R)^+2(3|4)"
    part2 = "A^*B(C|D|E)F(G|H|I)^2"
    part3 = "J^+K(L|M|N)^*O?(P|Q)^3"

    print("\n--- Part 1 (Prefix) ---")
    prefix_samples = generate_from_regex(part1, 10)
    print(prefix_samples)

    print("\n--- Part 2 (Middle) ---")
    middle_samples = generate_from_regex(part2, 10)
    print(middle_samples)

    print("\n--- Part 3 (Suffix) ---")
    suffix_samples = generate_from_regex(part3, 10)
    print(suffix_samples)
```

Conclusion:

The program effectively transforms custom regex-like patterns into a structured representation using parse_regex, simplifies human-readable syntax through preprocess_regex, and generates valid matching strings via generate_from_parsed. It robustly supports alternation, repetition, and grouping, ensuring both flexibility and accuracy. Thanks to its modular design, the implementation is easy to maintain, extend, or adapt for future enhancements or added regex capabilities.

References:

1. Else Course FAF.LFA21.1