Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică
Departamentul Inginerie software și automatică

Laboratory work 4: Formal Languages and Finite Automata

Elaborated:
st. gr. FAF-232 Istrati Ștefan

Verified:
asist. univ. Crețu Dumitru

Chişinău - 2025

Theory:

Regular expressions (regex) are powerful tools for pattern matching and text processing. They are used to define search patterns that can be used to find, match, or manipulate text based on specific criteria. Regex is a compact, expressive, and efficient way to describe patterns, and its use spans a wide range of applications in various domains such as software development, data extraction, text parsing, and validation.

What are Regular Expressions?

A regular expression is a sequence of characters that defines a search pattern. This pattern can be used to match strings of text, perform substitutions, and even validate the structure of text. The concept of regex is built around the idea of defining rules or patterns for strings, which can then be applied to a dataset to identify whether a string conforms to those rules. For example, a regex pattern might be designed to check if an email address is valid. The pattern would ensure that the string contains an "@" symbol, followed by a valid domain name. Similarly, regex can be used to extract specific information from a text document, like finding phone numbers or dates in a large dataset.

Regular expressions are widely used in many programming languages, including Python, JavaScript, Java, Perl, and C#. They are commonly employed in tasks such as:

- **Input Validation**: Ensuring that user input follows a specified format, such as checking if an email address or phone number is in the correct format.
- **Search and Replace**: Searching for a specific pattern in a string or text and replacing it with another string. This is often used in text editors and command-line tools.
- **Data Extraction**: Extracting specific pieces of data from large text files, such as extracting URLs from web pages or dates from logs.
- **Text Parsing**: Analyzing and splitting text into smaller components, often used in natural language processing tasks.

Components of a Regular Expression

A regular expression is made up of various components that specify the pattern to match. Some of the fundamental elements of regular expressions include:

1. **Literal Characters**: These are the basic characters that represent themselves in the regular expression. For example, a matches the character "a", and 1 matches the digit "1".
2. **Metacharacters**: These are characters that have special meanings in regular expressions, such as:

- . (dot): Matches any single character except newline characters.
- ^: Anchors the match to the start of the string.
- $: Anchors the match to the end of the string.
- []: Defines a character class, matching any one of the characters inside the brackets.
- *: Matches zero or more occurrences of the preceding character or group.
- +: Matches one or more occurrences of the preceding character or group.
- ?: Matches zero or one occurrence of the preceding character or group.

3. **Quantifiers**: These specify how many times a pattern should match. Common quantifiers include:

{n}: Matches exactly n occurrences of the preceding character or group.

{n,}: Matches n or more occurrences of the preceding character or group.

{n,m}: Matches between n and m occurrences of the preceding character or group.

4. **Groups and Alternation**: Parentheses () are used to group parts of a regular expression, and the pipe symbol | is used for alternation, meaning "or". For example, (a|b) will match either "a" or "b".
5. **Escaping Special Characters**: Some characters have special meanings in regular expressions. To match these characters literally, you can use a backslash (\). For example, \. will match a literal dot, rather than any character.

Practical Use Cases of Regular Expressions

Regular expressions are highly versatile and are employed in a variety of real-world scenarios:

1. Text Search: Regex is commonly used to search for specific patterns within text. For example, a regex could be used to find all instances of email addresses or phone numbers in a document.
2. Log File Analysis: Regular expressions are often used to analyze log files and extract important information, such as timestamps, error messages, or user activity patterns.
3. Data Validation: Regex can be used to validate user inputs, ensuring that they follow a specific pattern. For instance, validating a user-provided email address or password format before accepting it into a system.
4. Text Processing in Data Science: In data preprocessing tasks, regex is used to clean and extract data from unstructured text. It can be used to extract meaningful information from free-text fields in datasets, such as names, dates, and addresses.

Task: Dynamic Regular Expression String Generation

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:

1. generate_from_regex(pattern)

This function is the main entry point for generating a random string based on a regex pattern. It processes the pattern through three main steps: tokenization, parsing, and evaluation.

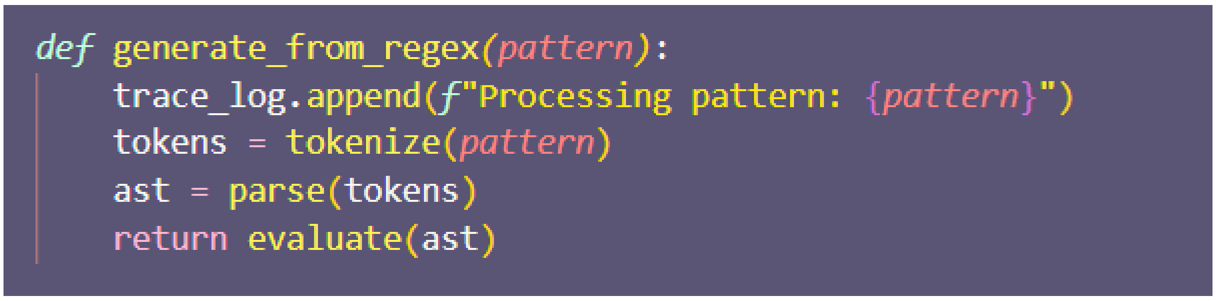
Explanation:

The trace_log.append(f"Processing pattern: {pattern}") line records the pattern being processed.

It calls the tokenize() function to break the pattern into tokens.

It then parses the tokens using the parse() function to build an AST (Abstract Syntax Tree).

Finally, it evaluates the AST using the evaluate() function to generate a string based on the pattern.



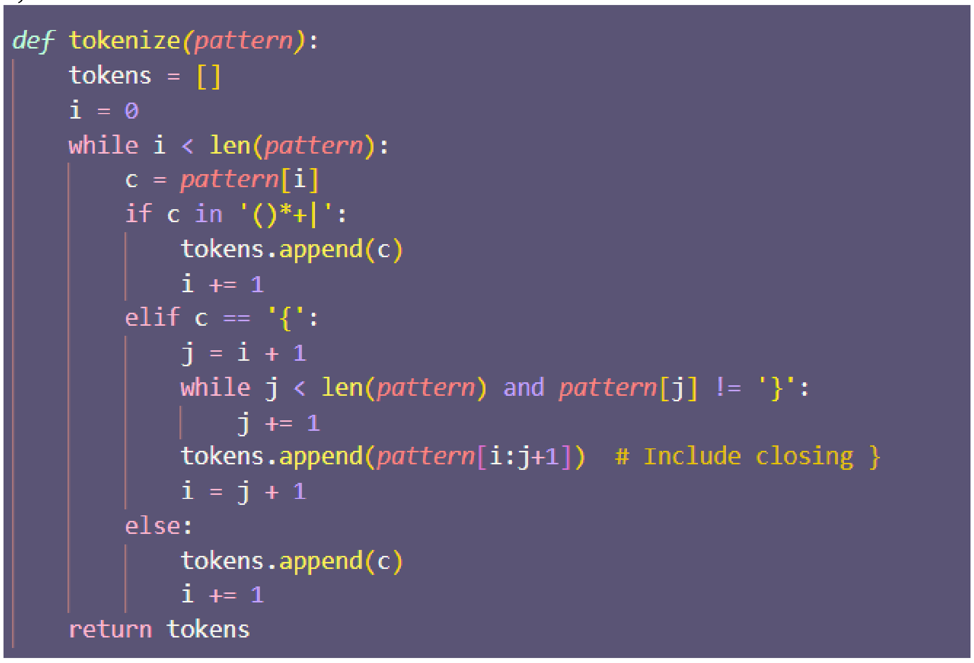
```python
def generate_from_regex(pattern):
    trace_log.append(f"Processing pattern: {pattern}")
    tokens = tokenize(pattern)
    ast = parse(tokens)
    return evaluate(ast)
```

2. tokenize(pattern)

This function is responsible for converting the regex pattern into a list of tokens. It processes each character of the pattern and identifies special characters or sequences like literals, groups, repetitions, and others.

Explanation:

The function iterates over each character (c) of the pattern.
If the character is a special character (e.g., *, +, |, (,)), it is added directly to the tokens list.
If it encounters a {, it identifies a repetition block (e.g., {3}) and extracts the entire block until the closing }.
If none of the above, it treats the character as a literal and adds it to the token list.



```python
def tokenize(pattern):
    tokens = []
    i = 0
    while i < len(pattern):
        c = pattern[i]
        if c in '()*+|':
            tokens.append(c)
            i += 1
        elif c == '{':
            j = i + 1
            while j < len(pattern) and pattern[j] != '}':
                j += 1
            tokens.append(pattern[i:j+1])  # Include closing }
            i = j + 1
        else:
            tokens.append(c)
            i += 1
    return tokens
```

3. parse(tokens)

This function converts the list of tokens into an Abstract Syntax Tree (AST). The AST represents the structure and logic of the regular expression in a more manageable format for evaluation.

Explanation:

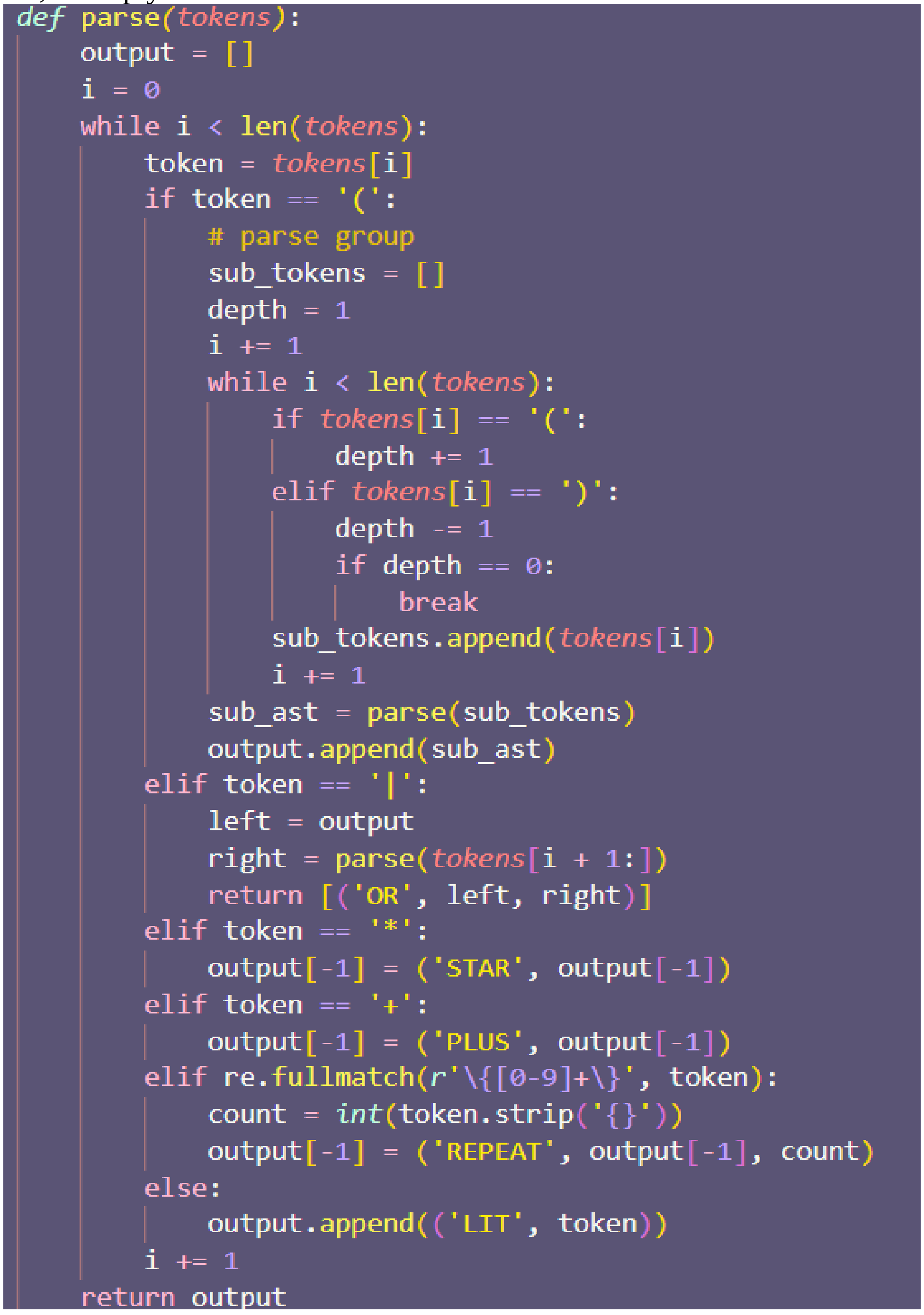
The function processes the tokens one by one.

If it encounters a group (i.e., (), it recursively parses the group and adds it as a sub-tree in the AST.

If it encounters an OR (|), it splits the current tokens into two parts: the left side and the right side of the OR. It returns a node indicating an OR operation.

For special characters like *, +, and {n}, it creates nodes that define the repetition behavior.

If the token is a literal, it simply adds it to the AST as a literal node.



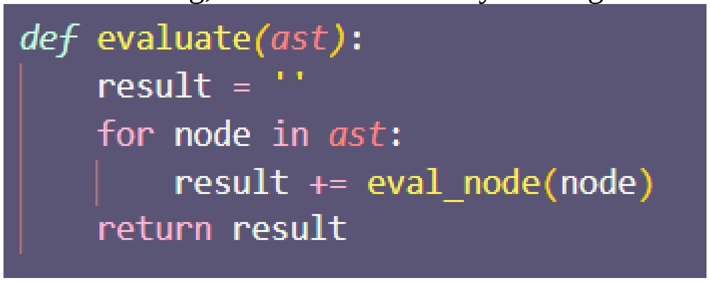
```python
def parse(tokens):
    output = []
    i = 0
    while i < len(tokens):
        token = tokens[i]
        if token == '(':
            # parse group
            sub_tokens = []
            depth = 1
            i += 1
            while i < len(tokens):
                if tokens[i] == '(':
                    depth += 1
                elif tokens[i] == ')':
                    depth -= 1
                    if depth == 0:
                        break
                sub_tokens.append(tokens[i])
                i += 1
            sub_ast = parse(sub_tokens)
            output.append(sub_ast)
        elif token == '|':
            left = output
            right = parse(tokens[i + 1:])
            return [('OR', left, right)]
        elif token == '*':
            output[-1] = ('STAR', output[-1])
        elif token == '+':
            output[-1] = ('PLUS', output[-1])
        elif re.fullmatch(r'\{[0-9]+\}', token):
            count = int(token.strip('{}'))
            output[-1] = ('REPEAT', output[-1], count)
        else:
            output.append(('LIT', token))
        i += 1
    return output
```

4. evaluate(ast)

This function evaluates the Abstract Syntax Tree (AST) generated in the parse() function. It recursively evaluates each node in the AST to generate a string based on the rules defined by the regular expression.

Explanation:

The function processes each node in the AST and calls eval_node() for each node.
It accumulates the result into the result string, which will ultimately be the generated string based on the regex.



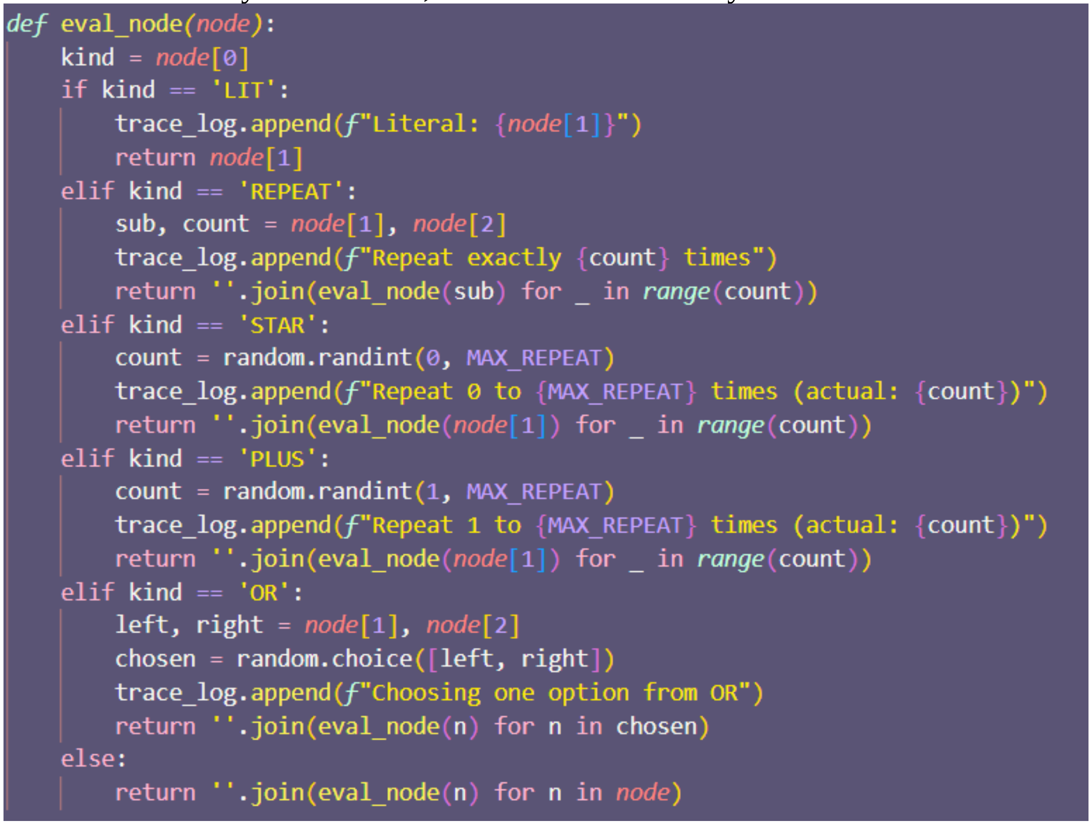
```python
def evaluate(ast):
    result = ''
    for node in ast:
        result += eval_node(node)
    return result
```

5. eval_node(node)

This function is responsible for evaluating individual nodes in the AST, depending on the type of node (literal, repeat, OR, etc.).

Explanation:

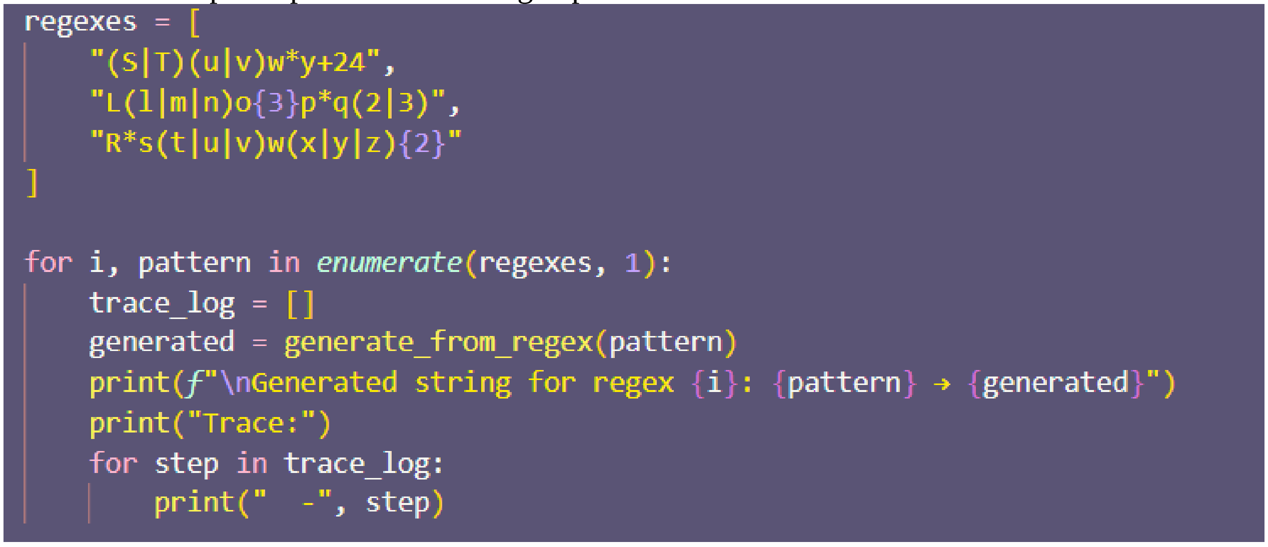
LIT: A literal node simply returns the literal character as a string and logs the action.
REPEAT: A repetition node repeats the sub-node a specified number of times. It logs the number of repetitions.
STAR: A star node (*) means the preceding element can repeat between 0 and MAX_REPEAT times. The number of repetitions is chosen randomly.
PLUS: A plus node (+) means the preceding element must repeat at least once, up to MAX_REPEAT times. The number of repetitions is chosen randomly.
OR: An OR node (|) means that either the left side or the right side can be chosen. The function randomly picks one of the two options and logs the choice.
If the node type doesn't match any of the above, the function recursively evaluates the sub-nodes.



```python
def eval_node(node):
    kind = node[0]
    if kind == 'LIT':
        trace_log.append(f"Literal: {node[1]}")
        return node[1]
    elif kind == 'REPEAT':
        sub, count = node[1], node[2]
        trace_log.append(f"Repeat exactly {count} times")
        return ''.join(eval_node(sub) for _ in range(count))
    elif kind == 'STAR':
        count = random.randint(0, MAX_REPEAT)
        trace_log.append(f"Repeat 0 to {MAX_REPEAT} times (actual: {count})")
        return ''.join(eval_node(node[1]) for _ in range(count))
    elif kind == 'PLUS':
        count = random.randint(1, MAX_REPEAT)
        trace_log.append(f"Repeat 1 to {MAX_REPEAT} times (actual: {count})")
        return ''.join(eval_node(node[1]) for _ in range(count))
    elif kind == 'OR':
        left, right = node[1], node[2]
        chosen = random.choice([left, right])
        trace_log.append(f"Choosing one option from OR")
        return ''.join(eval_node(n) for n in chosen)
    else:
        return ''.join(eval_node(n) for n in node)
```

6. Main Loop

The code includes a loop that processes three regex patterns:



```python
regexes = [
    "(S|T)(u|v)w*y+24",
    "L(l|m|n)o{3}p*q(2|3)",
    "R*s(t|u|v)w(x|y|z){2}"
]

for i, pattern in enumerate(regexes, 1):
    trace_log = []
    generated = generate_from_regex(pattern)
    print(f"\nGenerated string for regex {i}: {pattern} → {generated}")
    print("Trace:")
    for step in trace_log:
        print("  -", step)
```

Conclusion:

Through creating this laboratory, I gained a deeper understanding of how regular expressions can be interpreted and used to generate random strings. I learned how to break down a regex pattern into tokens, parse those tokens into an Abstract Syntax Tree (AST), and evaluate the AST to generate strings that conform to the pattern. Additionally, I gained experience in how to structure code to handle different regex constructs, such as repetition and alternation, in a modular way. Overall, this project enhanced my understanding of both regular expressions and how to implement a flexible string generation system in Python.