Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 3: Lexer & Scanner

Elaborated:	st. gr. FAF-223 Friptu Ludmila
Verified:	asist. univ. Cretu Dumitru

THEORY

1. Lexical Analysis

Lexical tokenization, also known as lexical analysis, is the process of converting a sequence of characters from source code into a sequence of tokens. A token is a string with an assigned and thus identified meaning. It represents a logical unit in the source code, such as keywords, identifiers, operators, literals, and punctuation symbols.

The primary function of lexical tokenization is to simplify the parsing stage by reducing the input character stream into a more manageable sequence of meaningful symbols. It involves scanning the code to identify token boundaries and categorizing each token based on its type, such as whether it is a keyword, an identifier, a constant, an operator, or a punctuation mark.

Lexical tokenization is the first stage in the process of compiling or interpreting code, where the source code is transformed into a format that can be easily understood and processed by a computer. It helps in removing whitespace, comments, and other irrelevant details from the source code, making it easier for subsequent stages of the compilation or interpretation process to analyze and execute the code.

2. How does a lexer work?

A lexer, short for lexical analyzer, is a component of a compiler or interpreter that performs lexical tokenization. The working of a lexer can be outlined in the following steps:

1. ***Input Reading***: The lexer reads the source code as a stream of characters. It processes the code character by character, starting from the beginning.

2. ***Token Recognition***: As the lexer reads through the characters, it identifies patterns that match different types of tokens. This is typically done using regular expressions or a state machine. Each pattern corresponds to a specific type of token, such as a keyword, identifier, literal, operator, or punctuation symbol.

3. ***Tokenization***: When a pattern is recognized, the lexer creates a token. This token is a data structure that typically contains the token's type (e.g., keyword, identifier, etc.), the actual string of characters (lexeme) that make up the token, and possibly additional information such as the token's position in the source code.

4. ***Whitespace and Comments Handling:*** The lexer often ignores whitespace (spaces, tabs, newlines) and comments, as they are usually not significant for the syntax of the program. However, it must recognize them to correctly identify the boundaries between tokens.

5. ***Error Handling:*** If the lexer encounters a sequence of characters that does not match any known token pattern, it generates an error. This error indicates that the source code contains an invalid or unrecognized token, which could be due to a typo, a syntax error, or an unsupported language feature.

6. ***Output Generation:*** The lexer continues processing the source code until it has been completely scanned. The output is a sequence of tokens that are passed to the next stage of the compilation or interpretation process, typically a parser. The parser uses these tokens to construct a syntactic structure of the program.

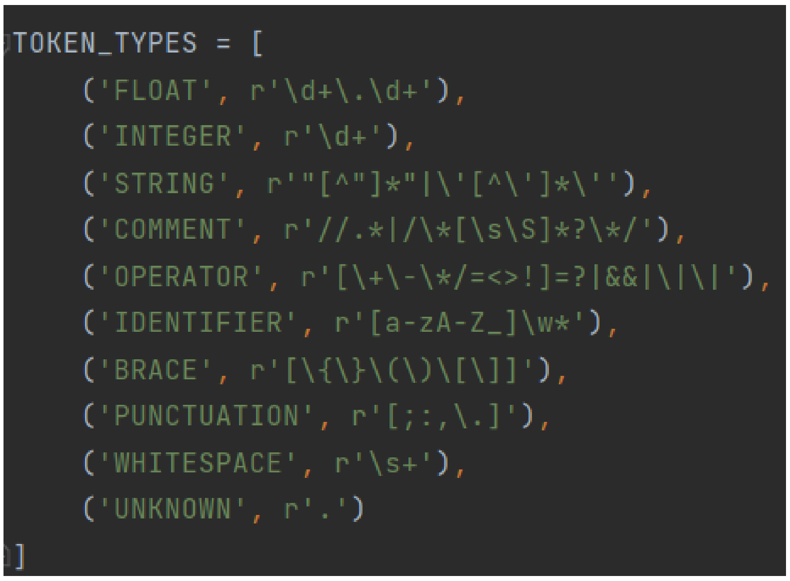
The lexer's design is crucial for the efficiency of the compiler or interpreter, as it is the first phase of the processing pipeline. A well-designed lexer can quickly and accurately convert the source code into tokens, facilitating the subsequent parsing phase.

OBJECTIVES

1. Understand what lexical analysis is.
2. Get familiar with the inner workings of a lexer/scanner/tokenizer.
3. Implement a sample lexer and show how it works.

IMPLEMENTATION

The first thing I implemented in my lexer project is the TOKEN_TYPES list, where I define the different types of tokens that the lexer can recognize, along with the regular expression patterns used to identify each token type in the source code. Each element in the list is a tuple where the first element is the token type, and the second element is the corresponding regular expression pattern.



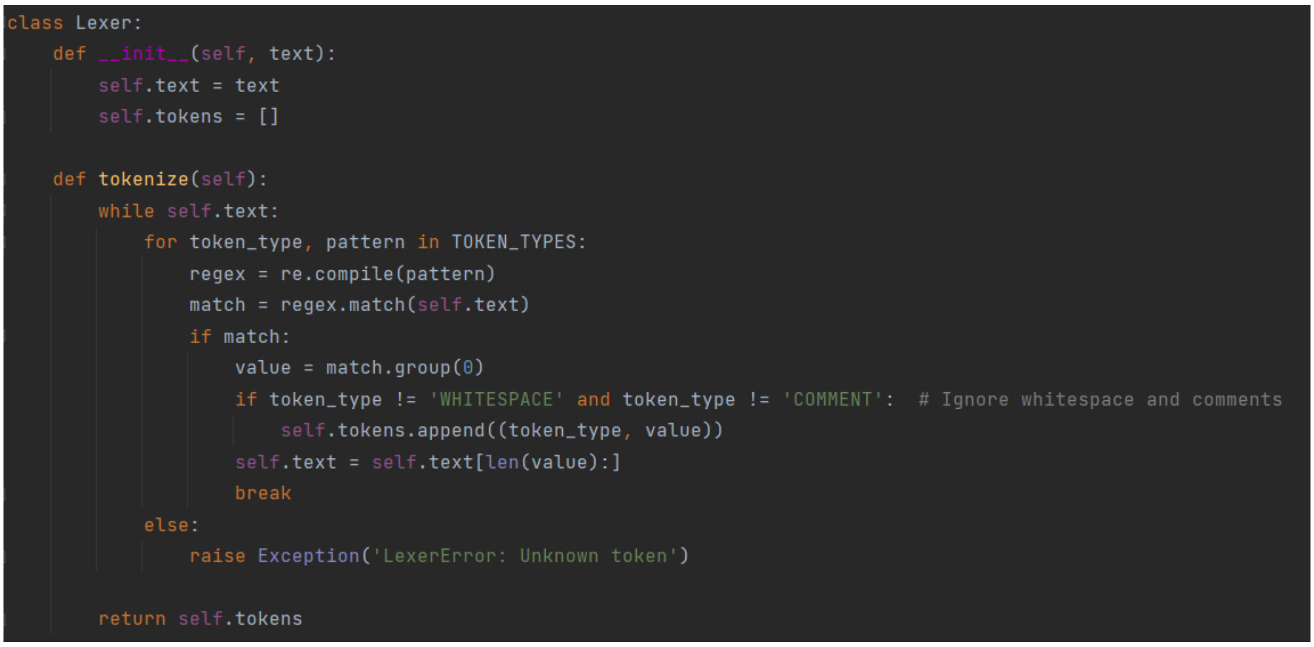
```
TOKEN_TYPES = [
    ('FLOAT', r'\d+\.\d+'),
    ('INTEGER', r'\d+'),
    ('STRING', r'"[^"]*"|\'[^\']*\''),
    ('COMMENT', r'//.*|/\*[\s\S]*?\*/'),
    ('OPERATOR', r'[\+\-\*/=<>!]=?|&&|\|\|'),
    ('IDENTIFIER', r'[a-zA-Z_]\w*'),
    ('BRACE', r'[\{\}\(\)\[\]]'),
    ('PUNCTUATION', r'[;:,\.]'),
    ('WHITESPACE', r'\s+'),
    ('UNKNOWN', r'.')
]
```

Next, I define the Lexer Class, which is responsible for converting a string of source code into a list of tokens based on the patterns defined in TOKEN_TYPES. The constructor method of the Lexer class (*_init__(self, text))*, initializes a new instance of the lexer with the source code to be tokenized . The source code is stored in *self.text*, and *self.tokens* is initialized as an empty list to hold the tokens that will be identified.

The *tokenize(self)* method performs the lexical analysis, converting the source code *(self.text)* into tokens. It works as follows:

- The method uses a while loop that continues as long as *self.text* is not empty.
- Within the loop, it iterates over TOKEN_TYPES to try and match each token type's pattern against the beginning of the remaining text. *for token_type, pattern in TOKEN_TYPE*S: iterates through the defined token types and their corresponding regular expression patterns.
- *regex = re.compile(pattern*): For each token type, it compiles the regular expression pattern into a regex object, which is used to search for a match at the start of the remaining text.

- *match = regex.match(self.text)*: This part attempts to match the compiled regular expression against the start of *self.text*.
- If a match is found, the matched string is extracted using *match.group(0)*. If the token type is neither WHITESPACE nor COMMENT, the token is appended to s*elf.tokens*.
- *self.text = self.text[len(value):]*: This line updates *self.text* by removing the portion that has just been tokenized, advancing the lexer's position in the source code.
- The *break* statement ends the inner for loop once a match is found, and the lexer then proceeds to the next portion of the text. If no match is found for any token type, the else block of the for loop raises an exception indicating an unknown token type.
- The method returns *self.tokens* after the entire text has been tokenized, providing a list of tokens identified in the source code.

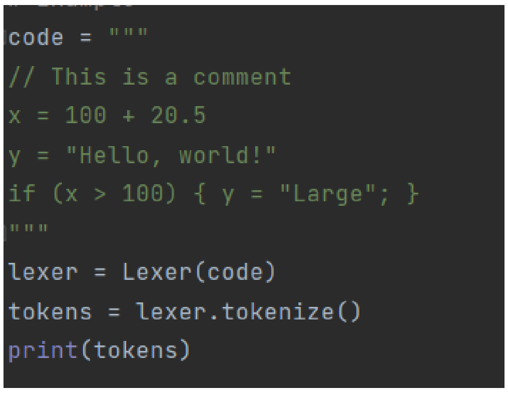


```python
class Lexer:
    def __init__(self, text):
        self.text = text
        self.tokens = []

    def tokenize(self):
        while self.text:
            for token_type, pattern in TOKEN_TYPES:
                regex = re.compile(pattern)
                match = regex.match(self.text)
                if match:
                    value = match.group(0)
                    if token_type != 'WHITESPACE' and token_type != 'COMMENT':  # Ignore whitespace and comments
                        self.tokens.append((token_type, value))
                    self.text = self.text[len(value):]
                    break
            else:
                raise Exception('LexerError: Unknown token')

        return self.tokens
```

RESULTS



```python
code = """
// This is a comment
x = 100 + 20.5
y = "Hello, world!"
if (x > 100) { y = "Large"; }
"""
lexer = Lexer(code)
tokens = lexer.tokenize()
print(tokens)
```

```
[('IDENTIFIER', 'x'), ('OPERATOR', '='), ('INTEGER', '100'),
('OPERATOR', '+'), ('FLOAT', '20.5'), ('IDENTIFIER', 'y'),
('OPERATOR', '='), ('STRING', '"Hello, world!"'),
('IDENTIFIER', 'if'), ('BRACE', '('), ('IDENTIFIER', 'x'),
('OPERATOR', '>'), ('INTEGER', '100'), ('BRACE', ')'),
('BRACE', '{'), ('IDENTIFIER', 'y'), ('OPERATOR', '='),
('STRING', '"Large"'), ('PUNCTUATION', ';'), ('BRACE', '}')]
```

CONCLUSION

In conclusion, implementing the lexer in our laboratory session was essential for understanding lexical analysis in compiler design. Through this hands-on exercise, I developed a Lexer class in Python, which efficiently tokenized source code into structured tokens, reinforcing my grasp of key concepts like pattern matching and regular expressions. This practical experience highlighted the lexer's crucial role in the compiler's workflow and underscored the importance of accuracy and detail in lexical analysis. The challenges faced during implementation provided valuable insights, enhancing our skills in compiler component development and preparing me for more advanced studies in compiler construction and programming language theory.