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

Formal Languages and Finite Automata

Laboratory work 3:
Lexer & Scanner

Elaborated:

Gavriliuc Tudor

Verified:

asist. univ. Cretu Dumitru
prof. univ. Cojuhari Irina

Chisinau 2024

THEORY

Lexical analysis, also known as lexical tokenization, is the process of converting raw text into meaningful lexical tokens. These tokens, categorized by a lexer program, include identifiers, operators, punctuation, and more, depending on the language being processed.

Tokenization Process

Token Identification: Lexical tokens are identified based on predefined rules and patterns defined by the language's lexical grammar. For example, identifiers, keywords, literals, and comments are recognized based on specific rules.

Token Categorization: Each identified lexeme is categorized into token classes, such as identifiers, operators, and literals. These classes represent the fundamental units of syntax within the language.

Output Generation: The output of the lexical analysis process is a stream of tokens, each representing a lexical element from the input text. Tokens serve as input for subsequent stages of language processing.

Lexer Implementation

● Rule-Based Approach: Lexers are typically rule-based programs, often referred to as tokenizers or scanners. They employ finite-state machines or regular expressions to identify and categorize lexemes into tokens. Tokenization Examples

● Common Tokens: Tokens include identifiers, keywords, operators, literals, comments, and whitespace. Each token has a name and, optionally, a value associated with it.

● Tokenization Process: Tokens are identified using various methods, including regular expressions, character sequences, delimiters, or explicit definitions. Lexers handle the tokenization process efficiently, reporting errors for invalid tokens.

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 Python lexer implementation that is offered is a flexible tool for tokenizing input text that is mainly intended for parsing code from different programming languages. The modular architecture of its design enables the efficient management of various lexical analysis components.

Fundamentally, the lexer works in an iterative manner, carefully going over each character in the incoming text. A Position object, which meticulously tracks the current position throughout the text, facilitates this operation. The lexer skillfully recognizes and classifies a wide variety of token types, including as integers, floats, operators, parentheses, strings, characters, identifiers, and keywords, by utilizing the power of regular expressions.

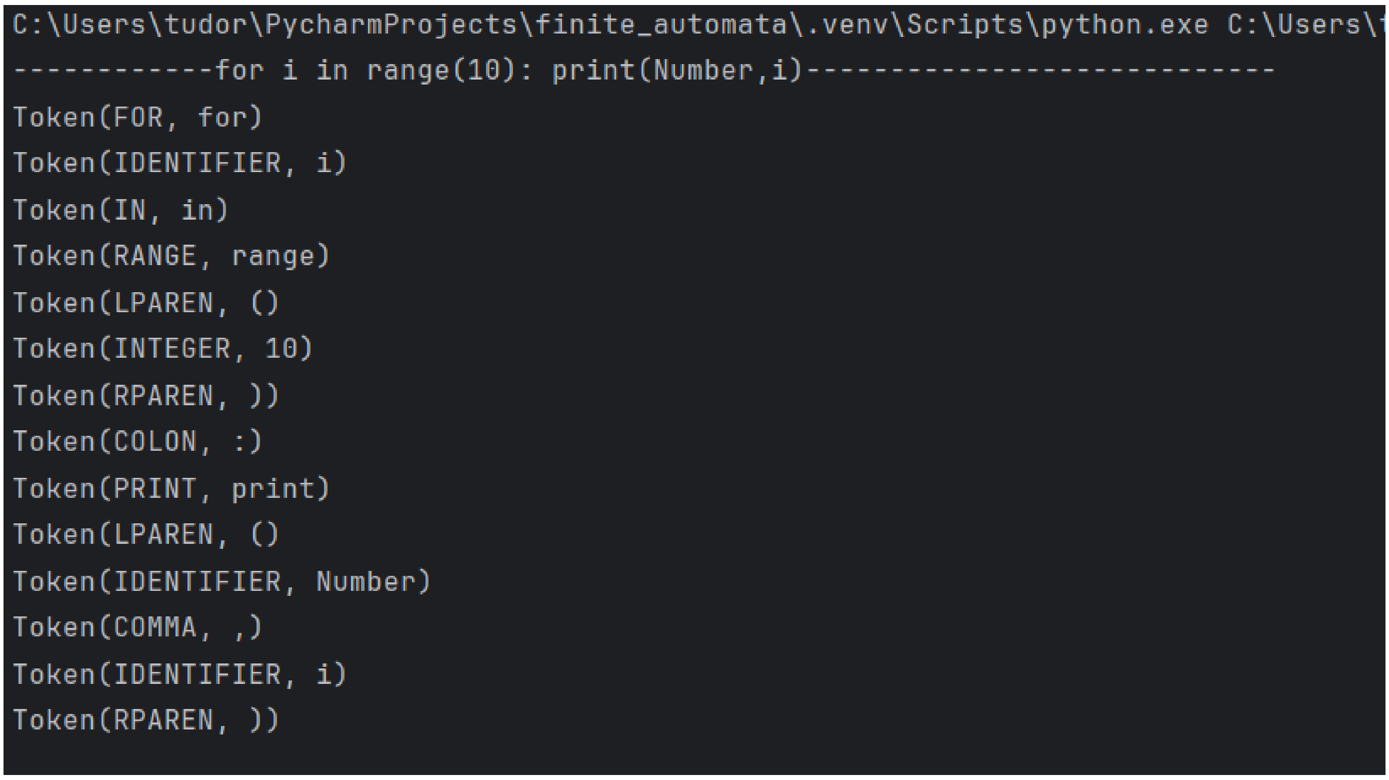
```python
class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

    def __str__(self):
        return f'Token({self.type}, {self.value})'
```

The Token class is a simple data structure designed to represent individual tokens within the lexer. Each instance of Token holds two essential attributes: type and value, where type is a predefined token category (like INTEGER or IDENTIFIER), and value is the actual text segment matched to the token type. The class provides a clear and structured way to encapsulate token information, facilitating further processing in the compilation or interpretation process. The __str__ method is overridden to provide a human-readable string representation of the token, which is helpful for debugging and visualization of the token stream generated by the lexer.

```python
def get_next_token(self):
    while self.pos < len(self.text):
        for pattern, token_type in TOKEN_REGEX:
            regex = re.compile(pattern)
            match = regex.match(self.text, self.pos)
            if match:
                value = match.group(0)
                self.pos = match.end()
                if token_type == SPACE or token_type == NEWLINE:
                    break  # Skip spaces
                elif value in KEY_WORDS:
                    return Token(value.upper(), value)
                else:
                    return Token(token_type, value)
        else:
            self.error()
```

The get_next_token function is a crucial method within the Lexer class that iterates through the input text to identify and generate tokens based on predefined patterns. It utilizes regular expressions to match segments of the text to token types, such as identifiers, numbers, and symbols. When a match is found, the function creates an instance with the matched type and value, then updates the position within the text to continue tokenization. The function gracefully handles spaces and newline characters by skipping them, ensuring that only meaningful tokens are generated. If the function encounters an unrecognized character, it triggers an error mechanism to notify of an invalid token. The process repeats until the end of the input text is reached, at which point the function returns an EOF (end-of-file) token, signaling the completion of the lexical analysis.

Results



```
C:\Users\tudor\PycharmProjects\finite_automata\.venv\Scripts\python.exe C:\Users\
------------for i in range(10): print(Number,i)----------------------------
Token(FOR, for)
Token(IDENTIFIER, i)
Token(IN, in)
Token(RANGE, range)
Token(LPAREN, ()
Token(INTEGER, 10)
Token(RPAREN, ))
Token(COLON, :)
Token(PRINT, print)
Token(LPAREN, ()
Token(IDENTIFIER, Number)
Token(COMMA, ,)
Token(IDENTIFIER, i)
Token(RPAREN, ))
```

A series of tokens produced by using a lexer to parse a piece of code are shown in the output. A recognized syntactic or data structure from the input language, such as keywords (FOR, IN, PRINT), identifiers (i, Number), a number (10), and other symbols ((,),,, :), is represented by each instance of a token. Key components of a basic loop structure in various programming languages are the FOR and IN tokens, which identify the beginning of a loop construct, RANGE, which represents a range function, and PRINT, which represents a print statement. The presence of the LPAREN and RPAREN tokens indicates that function arguments or expression groupings—in this case, the range value 10 and the parameters of the print statement—are enclosed in parenthesis.

Conclusion

In summary, the lexer included in the supplied code shows strong tokenization capabilities for a variety of inputs. The lexer precisely recognizes and classifies lexical tokens, such as identifiers, operators, literals, and keywords, using rigorous lexical analysis and pre-established rules and patterns. Robust handling of several lexical analysis components, including character, string, and numeric literals, is ensured by its rule-based approach and modular design. Furthermore, the lexer handles whitespace and comments effectively, ignoring them when tokenizing the input text. The published findings demonstrate how well the lexer can tokenize text literals, conditional statements, and expressions, clearly breaking out each step in the process.