Ministry of Education, Culture, and Research of the Republic of Moldova
Technical University of Moldova
The Faculty of Computers, Informatics, and Microelectronics

REPORT

Laboratory work no.3
Lexical Analyzer

Executed by:	Cornievschi Bogdan
st. gr. FAF-222	
Verified by:	
univ. assist.	Crețu Dumitru

Chişinău - 2024

Introduction

A lexer, short for lexical analyser, is a critical component in compilers, interpreters, and various text processing applications. Its primary function is to read input text and break it down into a sequence of tokens that represent syntactic units, such as keywords, identifiers, literals, and operators. This project demonstrates the implementation of a simple lexer in Python, designed for a hypothetical programming language with a basic syntax.

Objective

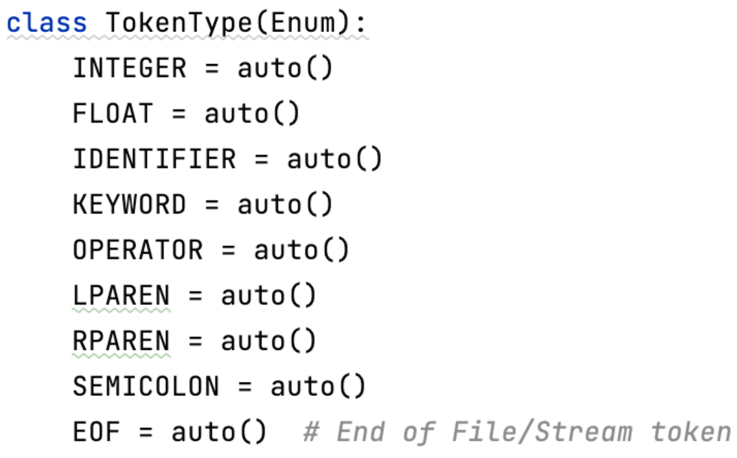
The main objective of this project is to develop a lexer capable of recognising and categorising the fundamental elements of a simplified programming language. These elements include:

- Integer literals (e.g., 123)
- Float literals (e.g., 123.45)
- Identifiers (e.g., variableName)
- Keywords (e.g., if, else, return)
- Operators (e.g., +, -, *, /, <, >)
- Parentheses and semicolons (e.g., (,), ;)

Design and Implementation

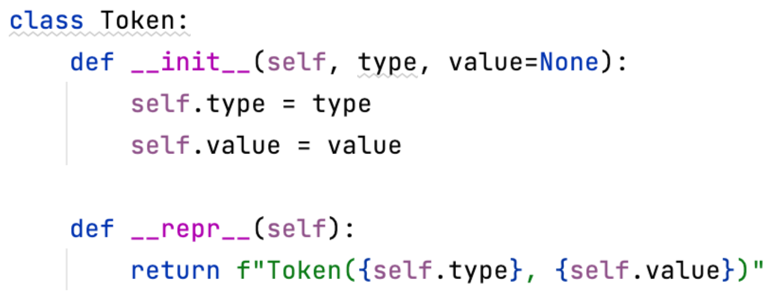
Token Types

The lexer defines several types of tokens using an enumeration (enum) class, allowing for clear and manageable token type management. Each token type is assigned a unique value using Python's auto() function for convenience and maintainability.



```python
class TokenType(Enum):
    INTEGER = auto()
    FLOAT = auto()
    IDENTIFIER = auto()
    KEYWORD = auto()
    OPERATOR = auto()
    LPAREN = auto()
    RPAREN = auto()
    SEMICOLON = auto()
    EOF = auto()  # End of File/Stream token
```

A Token class encapsulates the concept of a token, holding its type and value. This simple representation is essential for the lexer's output, facilitating further processing by parsers or other components.



```python
class Token:
    def __init__(self, type, value=None):
        self.type = type
        self.value = value

    def __repr__(self):
        return f"Token({self.type}, {self.value})"
```

The Lexer class serves as the backbone of the lexical analyser project, encapsulating the logic required to transform raw input text into a stream of tokens. This class is meticulously designed to recognise and categorise the syntactic elements of a hypothetical programming language, identifying keywords, identifiers, literals, operators, and punctuation symbols like parentheses and semicolons. The detailed description of the Lexer class provides insights into its structure, methods, and operational mechanics, highlighting its critical role in the lexical analysis process.

The Lexer class is initialised with a single string of input text, which represents the code or data to be tokenised. This input is stored along with a position indicator (pos), which tracks the lexer's current location within the string. The position indicator is crucial for navigating through the input text character by character.

Operational Mechanics

The lexer operates by iterating over each character in the input text, applying a series of condition checks to determine the appropriate categorisation for each character or sequence of characters. This iterative process continues until the entire text has been processed, effectively breaking it down into tokens.

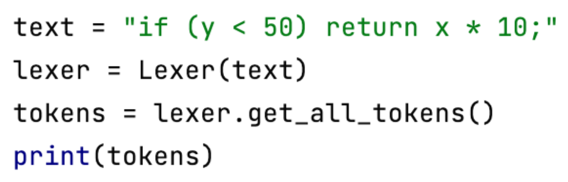
Core Methods

- next_token: This method is the heart of the Lexer class, responsible for identifying the next token in the input text based on the current position. It uses the current character to decide which action to take, such as recognising a digit, letter, operator, or special symbol. Depending on the character, next_token may call other helper methods to process numbers, identifiers, or handle errors. After processing, it advances the position indicator to continue scanning the remaining text.
- skip_whitespace: This method allows the lexer to ignore whitespace characters in the input text, which are irrelevant for tokenisation in most programming languages.
- number: Extracts a number from the input text, handling both integer and floating-point numbers. This method is called when next_token encounters a digit, and it continues to read characters until it completes the number.
- identifier: Similar to number, this method processes a sequence of characters starting with a letter (or underscore), identifying identifiers or keywords. It distinguishes between keywords (reserved words of the programming language) and identifiers (names defined by the user, such as variable names) based on a predefined list of keywords.
- error: Raises an exception when the lexer encounters an invalid character, signaling an error in the input text.

Token Generation

As the Lexer processes the input text, it generates Token instances for each recognised syntactic unit. These tokens are then consumed by a parser or other components of a compiler or interpreter for further processing.

The screenshot from below demonstrates the lexer processing a simple statement, outputting a sequence of tokens representing the syntactic elements of the input text.



```
text = "if (y < 50) return x * 10;"
lexer = Lexer(text)
tokens = lexer.get_all_tokens()
print(tokens)
```

Output example

For the text "if (y < 50) return x * 10;” the result is

```
[Token(TokenType.KEYWORD, if), Token(TokenType.LPAREN, (),
Token(TokenType.IDENTIFIER, y), Token(TokenType.OPERATOR, <),
Token(TokenType.INTEGER, 50), Token(TokenType.RPAREN, )),
Token(TokenType.KEYWORD, return), Token(TokenType.IDENTIFIER, x),
Token(TokenType.OPERATOR, *), Token(TokenType.INTEGER, 10),
Token(TokenType.SEMICOLON, ;), Token(TokenType.EOF, None)]
```

Conclusion

The Lexer class's implementation showcases the fundamental principles of lexical analysis, translating the complexities of programming language syntax into a structured sequence of tokens. By meticulously categorising each character and sequence of characters in the input text, the lexer lays the groundwork for the subsequent phases of compilation or interpretation, such as parsing and semantic analysis. This detailed approach to lexical analysis underscores the importance of the lexer in the broader context of programming language processing and compilation.