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

Laboratory work 3:
Course: Formal Languages and Finite Automata
Topic: Lexer & Scanner

Elaborated:
st. gr. FAF-221 Chichioi Iuliana

Verified:
asist. univ. Cretu Dumitru

Chişinău - 2023

TABLE OF CONTENTS

THEORY......3
Tokenization Process......3
Lexer Implementation......3
Tokenization Examples......3
OBJECTIVES......3
IMPLEMENTATION......3
RESULTS......5

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 provided lexer implementation in Python serves as a versatile tool for tokenizing input text, tailored primarily for parsing code in various programming languages. Its architecture is built upon a modular design, allowing for the effective handling of different aspects of lexical analysis.

At its core, the lexer operates through an iterative process, meticulously scanning the input text character by character. This process is facilitated by a Position object, which diligently tracks the current position within the text. By leveraging the power of regular expressions, the lexer adeptly identifies and categorizes a diverse range of token types, including integers, floats, operators, parentheses, strings, characters, identifiers, and keywords.

```
import re
 # Regular expressions for token patterns
TOKEN_REGEX = r'(?P<TOKEN>{})'
TOKEN_PATTERNS = {
      'INT': r'\d+',
      'FLOAT': r'\d+\.\d+',
      'OPERATOR': r'[\+\-\*/]',
      # Other token patterns...
 }
```

The make_tokens() method serves as the core of the tokenization process in the lexer, orchestrating the generation of tokens from the input text. It relies on a set of helper methods, each tailored to handle specific token types, ensuring accurate parsing and token creation. make_number() is responsible for identifying and creating tokens representing numeric literals, such as integers and floats. make_string() handles the parsing of string literals, including the proper handling of escape characters. make_char() specifically deals with character literals, ensuring their correct representation as tokens. Finally, make_identifier() identifies and categorizes identifiers, distinguishing them from keywords and generating corresponding tokens. By leveraging these helper methods, the lexer efficiently navigates through the input text, generating tokens in accordance with predefined patterns and rules.

```
class Lexer:
      def __init__(self, text):
            self.text = text
            self.pos = Position(-1, 0, -1, text)
            self.current_char = None
      def make_tokens(self):
            tokens = []
            errors = []
            while self.current_char is not None:
                  if re.match(r'\d', self.current_char):
                        tokens.append(self.make_number())
                  elif self.current_char == '"':
                        tokens.append(self.make_string())
                  # Other token handling...
                  else:
                        pos_start = self.pos.copy()
                        char = self.current_char
                        self.advance()
                        errors.append(IllegalCharError(pos_start, self.pos, "'" + char
                  + "'"))
      return tokens, errors
```

A notable feature of the lexer is its capability to handle comments—both single-line and multi-line—without generating tokens for them. Additionally, it effectively manages whitespace characters, disregarding them unless they form part of a string literal. This robust handling of comments and whitespace contributes to the lexer's overall efficiency and accuracy.

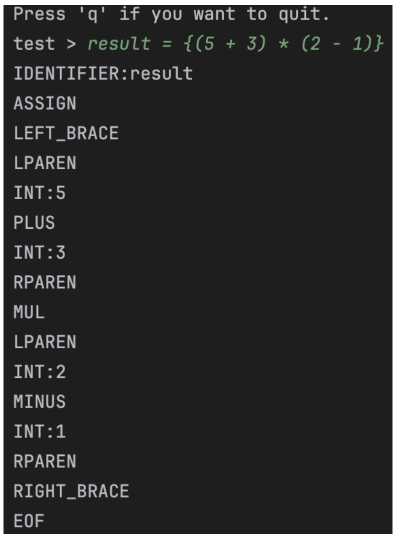
```
class Lexer:
      def skip_comment(self):
            # Skip single-line or multi-line comments
            ...
      def make_tokens(self):
            ...
            while self.current_char is not None:
                  if self.current_char == '/':
                        if self.peek() == '/':
                              self.skip_comment()
                        ...
                  ...
```

Upon completion of tokenization, the lexer produces a comprehensive list of tokens, encapsulating the lexical structure of the input text. Each token, represented by a Token object, contains vital information about its type and, if applicable, its corresponding value. Furthermore, any encountered lexical errors, such as illegal characters, are diligently captured and reported by the lexer, ensuring robustness and accuracy in the tokenization process.

In essence, the lexer operates based on predefined rules tailored to the target programming language or input format. Its modular design fosters flexibility and extensibility, making it well-suited for parsing a wide range of programming languages or textual formats. Ultimately, the lexer stands as an indispensable component in the intricate process of parsing and analyzing code or text.

RESULTS

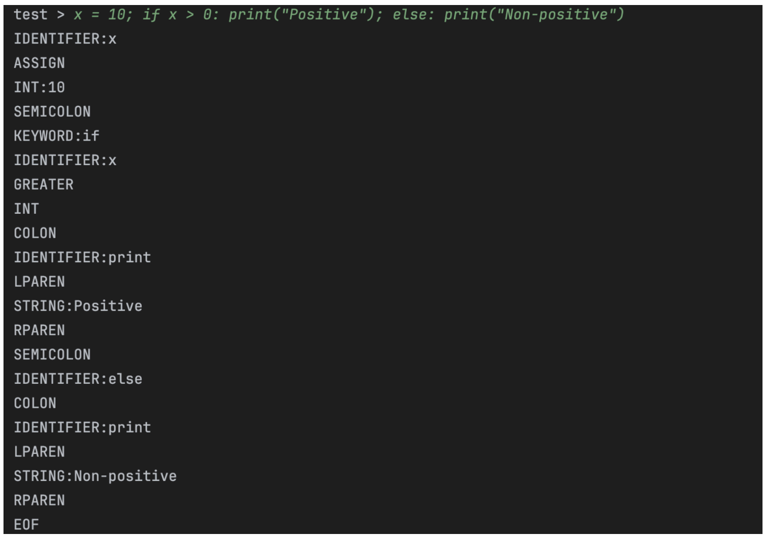
The reported results showcase the functionality of the lexer implemented in the provided code. In the evaluation of expressions, such as result = {(5 + 3) * (2 - 1)}, the lexer accurately identifies tokens like identifiers (result), mathematical operators (+, -, *), and braces, providing a clear breakdown of each component involved in the expression.



```
Press 'q' if you want to quit.
test > result = {(5 + 3) * (2 - 1)}
IDENTIFIER:result
ASSIGN
LEFT_BRACE
LPAREN
INT:5
PLUS
INT:3
RPAREN
MUL
LPAREN
INT:2
MINUS
INT:1
RPAREN
RIGHT_BRACE
EOF
```

Figure 1 first_input

Additionally, when processing conditional statements like x = 10; if x > 0: print("Positive"); else: print("Non-positive"), the lexer distinguishes between keywords (if), identifiers (x), comparison operators (>), and string literals ("Positive", "Non-positive"), enabling precise tokenization of the input.



```
test > x = 10; if x > 0: print("Positive"); else: print("Non-positive")
IDENTIFIER:x
ASSIGN
INT:10
SEMICOLON
KEYWORD:if
IDENTIFIER:x
GREATER
INT
COLON
IDENTIFIER:print
LPAREN
STRING:Positive
RPAREN
SEMICOLON
IDENTIFIER:else
COLON
IDENTIFIER:print
LPAREN
STRING:Non-positive
RPAREN
EOF
```

Figure 2 second_input

Overall, the lexer demonstrates robust functionality in tokenizing various inputs, accurately capturing the structure and content of expressions, statements, and literals while appropriately handling error scenarios, also skip comments.

CONCLUSION

In conclusion, the lexer implemented in the provided code demonstrates effective tokenization capabilities for a wide range of inputs. Through meticulous lexical analysis, the lexer accurately identifies and categorizes lexical tokens, including identifiers, operators, literals, and keywords, based on predefined rules and patterns. Its modular design and rule-based approach ensure robust handling of various aspects of lexical analysis, such as numeric literals, string literals, and character literals. Additionally, the lexer efficiently handles comments and whitespace, disregarding them when tokenizing the input text. The reported results showcase the lexer's functionality in accurately tokenizing expressions, conditional statements, and text literals, providing a clear breakdown of each component involved. Overall, the lexer serves as a versatile tool for parsing and analyzing code or text, contributing to the intricate process of language processing and comprehension.