Scanner Design and Implementation Report

- By Ammar Ahmad, 2020071

Introduction

The purpose of this report is to document the design, implementation details, and test cases for the MiniLang scanner. The MiniLang scanner is designed to tokenize MiniLang source code according to the language's specifications.

Scanner Design

The scanner is implemented in Python and consists of three main components: TokenType, Token, and Scanner classes.

- **TokenType**: Enumerates the different types of tokens recognized by the scanner, such as INTEGER, BOOLEAN, OPERATOR, KEYWORD, IDENTIFIER, LITERAL, COMMENT.
- **Extra Tokens**: *LEFT_BRACE, RIGHT_BRACE, LEFT_PAREN, and RIGHT_PAREN* – added on my own.
- **Token**: Represents an individual token with its token type and lexeme.
- **Scanner**: Responsible for tokenizing the MiniLang source code. It reads the source code from a file, tokenizes each line, and stores the tokens in a list.

Implementation Details

- The scanner utilizes regular expressions to match token patterns for integers, booleans, operators, keywords, identifiers, literals, comments, and special characters (braces and parentheses).
- It iterates through each line of the source code, tokenizes the line using the specified patterns, and appends the tokens to the token list.
- Error handling is implemented to raise a ValueError if an invalid token is encountered.

Test Cases

Test cases are essential to ensure the correctness and robustness of the scanner. Below are some example test cases:

1. **Basic Syntax Test**: Verify that the scanner correctly tokenizes basic MiniLang syntax elements such as variables, arithmetic operations, if-else statements, and print statements.
2. **Error Handling Test**: Test cases with invalid tokens or syntax errors to ensure proper error handling.
3. **Complex Expression Test**: Test complex expressions involving multiple operators, parentheses, and nested if-else statements.
4. **Large Input Test**: Test the scanner's performance and memory usage with large input files.

How to Run the Program

To run the MiniLang scanner program:

1. Save the provided Python code in a file, e.g., **scanner.py**.

2. Create a text file containing MiniLang source code, e.g., **input.txt**.
3. Execute the Python script:

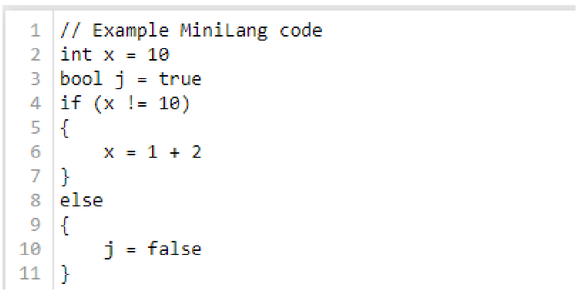
   python scanner.py

4. This program will tokenize the source code in **input.txt** and print the resulting tokens along with their types & lexemes.

Conclusion

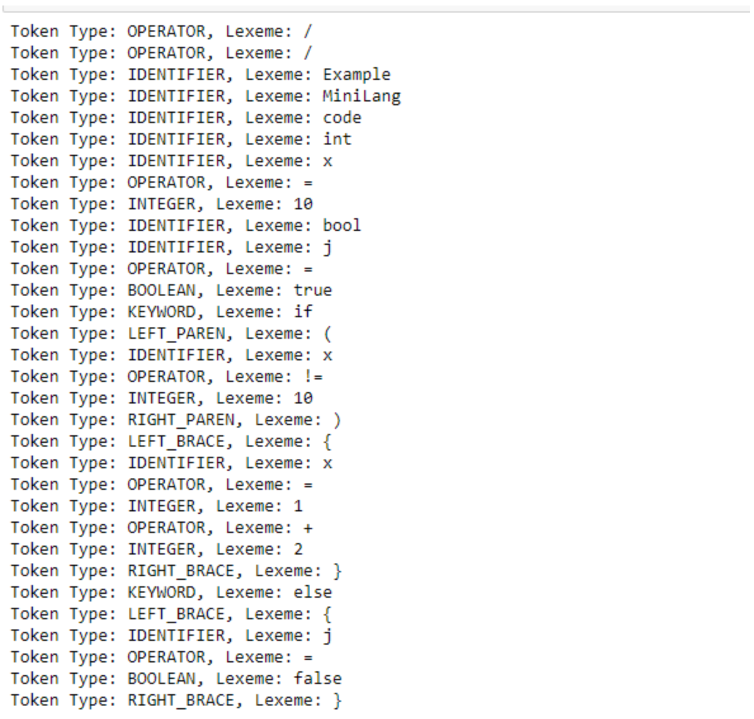
The MiniLang scanner is a crucial component of the compiler design process, responsible for tokenizing source code and preparing it for subsequent stages. By following the specified design and implementation details, the scanner effectively handles MiniLang syntax and provides accurate tokenization.

Screen Shots



```
// Example MiniLang code
int x = 10
bool j = true
if (x != 10)
{
    x = 1 + 2
}
else
{
    j = false
}
```

Figure 1: input.txt



```
Token Type: OPERATOR, Lexeme: /
Token Type: OPERATOR, Lexeme: /
Token Type: IDENTIFIER, Lexeme: Example
Token Type: IDENTIFIER, Lexeme: MiniLang
Token Type: IDENTIFIER, Lexeme: code
Token Type: IDENTIFIER, Lexeme: int
Token Type: IDENTIFIER, Lexeme: x
Token Type: OPERATOR, Lexeme: =
Token Type: INTEGER, Lexeme: 10
Token Type: IDENTIFIER, Lexeme: bool
Token Type: IDENTIFIER, Lexeme: j
Token Type: OPERATOR, Lexeme: =
Token Type: BOOLEAN, Lexeme: true
Token Type: KEYWORD, Lexeme: if
Token Type: LEFT_PAREN, Lexeme: (
Token Type: IDENTIFIER, Lexeme: x
Token Type: OPERATOR, Lexeme: !=
Token Type: INTEGER, Lexeme: 10
Token Type: RIGHT_PAREN, Lexeme: )
Token Type: LEFT_BRACE, Lexeme: {
Token Type: IDENTIFIER, Lexeme: x
Token Type: OPERATOR, Lexeme: =
Token Type: INTEGER, Lexeme: 1
Token Type: OPERATOR, Lexeme: +
Token Type: INTEGER, Lexeme: 2
Token Type: RIGHT_BRACE, Lexeme: }
Token Type: KEYWORD, Lexeme: else
Token Type: LEFT_BRACE, Lexeme: {
Token Type: IDENTIFIER, Lexeme: j
Token Type: OPERATOR, Lexeme: =
Token Type: BOOLEAN, Lexeme: false
Token Type: RIGHT_BRACE, Lexeme: }
```

Figure 2: output