Project Report

Project Title: C++ Lexical Analyzer in Python

Group Members:
Mohid Ali (2023794)
Abdul Hannan(2023008)
Sahibzada Muhammad Asad Shayan (2023629)

CS-224
Instructor: Sajid Ali

Section: BCS

Faculty of Computer Sciences and Engineering
Ghulam Ishaq Khan Institute ofEngineering Sciences and technology 2025

Table of Contents

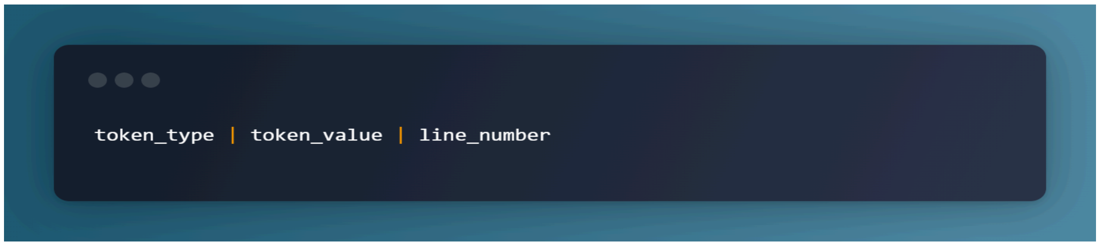
1. **Introduction**
2. **Project Description**
3. **Goals and Objectives**
4. **Key Components**
5. **Code Snippets**
6. **Sample Input and Output**
7. **Challenges and Learnings**
8. **Conclusion**

1. Introduction

This project is developed as part of the **CS224: Formal Languages and Automata Theory** course under the guidance of **Mr. Muhammad Sajid Ali**. It follows up on a warm-up assignment where a lexical analyzer was built using Flex. The current task is to **reimplement that analyzer in Python** using the built-in **re** module without using any external libraries. The analyzer reads C++ code, identifies valid tokens, and outputs them in a structured format.

2. Project Description

The lexical analyzer takes **C++ source files** as input, scans them using regular expressions, and extracts all valid tokens such as keywords, identifiers, numbers, strings, separators, operators, comments, and preprocessor directives. These tokens are written to output files in the format:



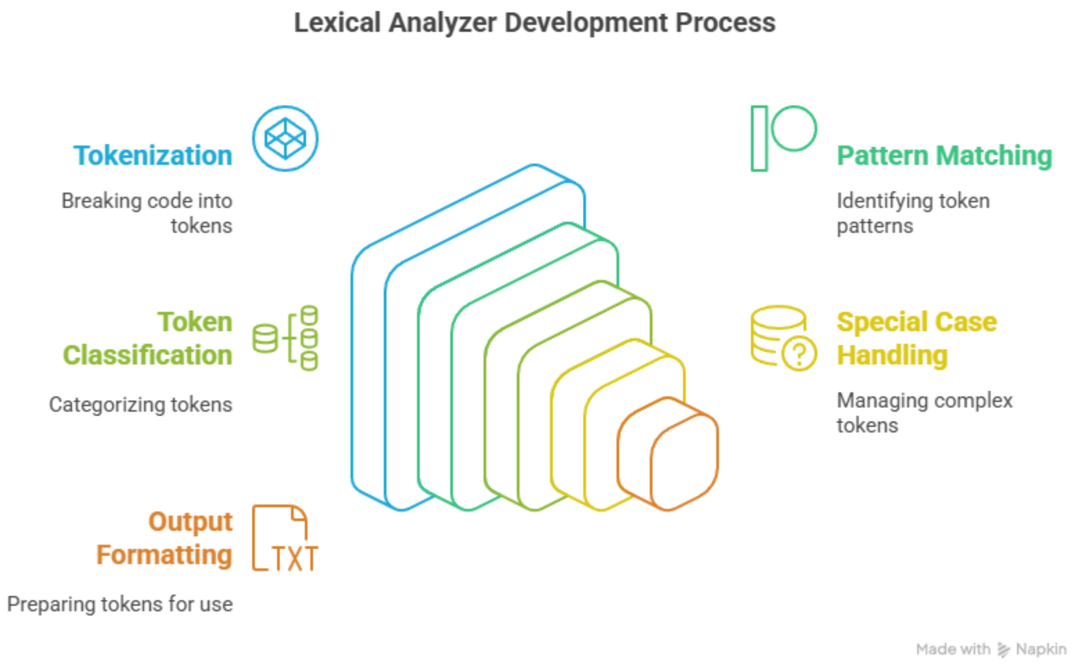
```
token_type | token_value | line_number
```

The project directory includes:

- Multiple input files (`input.cpp`, `input1.cpp`, `input2.cpp`)
- Python lexer implementation (`lexer.py`)
- Outputs (`output.txt`, `output1.txt`, `output2.txt`, `output8.txt`)
- Original Flex code (`main.l`), compiled code (`lexer.exe`), and the generated C file (`lex.yy.c`)

3. Goals and Objectives

- Rebuild a functional lexical analyzer in Python mimicking Flex behavior.
- Understand how tokenizing tools work internally.
- Use only standard Python libraries (**re, sys**) for pattern matching.
- Produce clearly labeled tokenized output files.
- Extend analyzer capabilities to include comments.

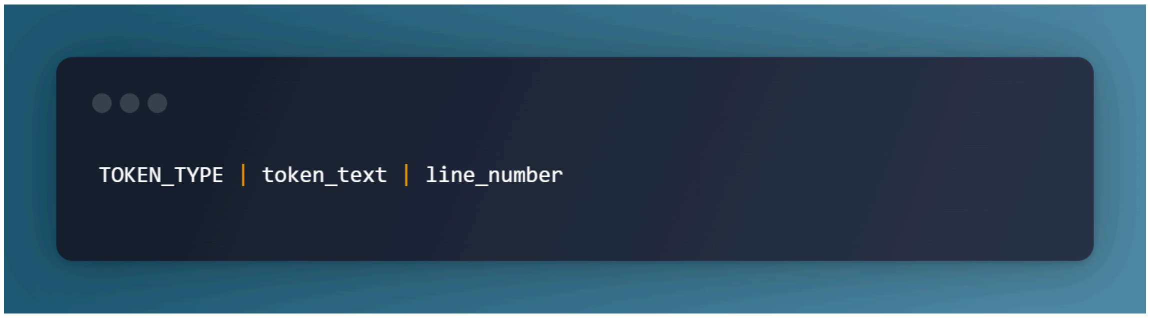


4. Key Components

a. Token Types Handled:

- Keywords (e.g., `int`, `float`, `return`)
- Identifiers
- Integer and Float Literals
- Character and String Literals
- Operators (**+**, `==`, **&&**, etc.)
- Separators (`{`, `}`, `(`, `)`, `;`, etc.)
- Comments (both `//` and `/* */`)
- Preprocessor directives (`#include`, `#define`, etc.)
- Newlines (for tracking line numbers)

b. Output Format: Each token is output as:

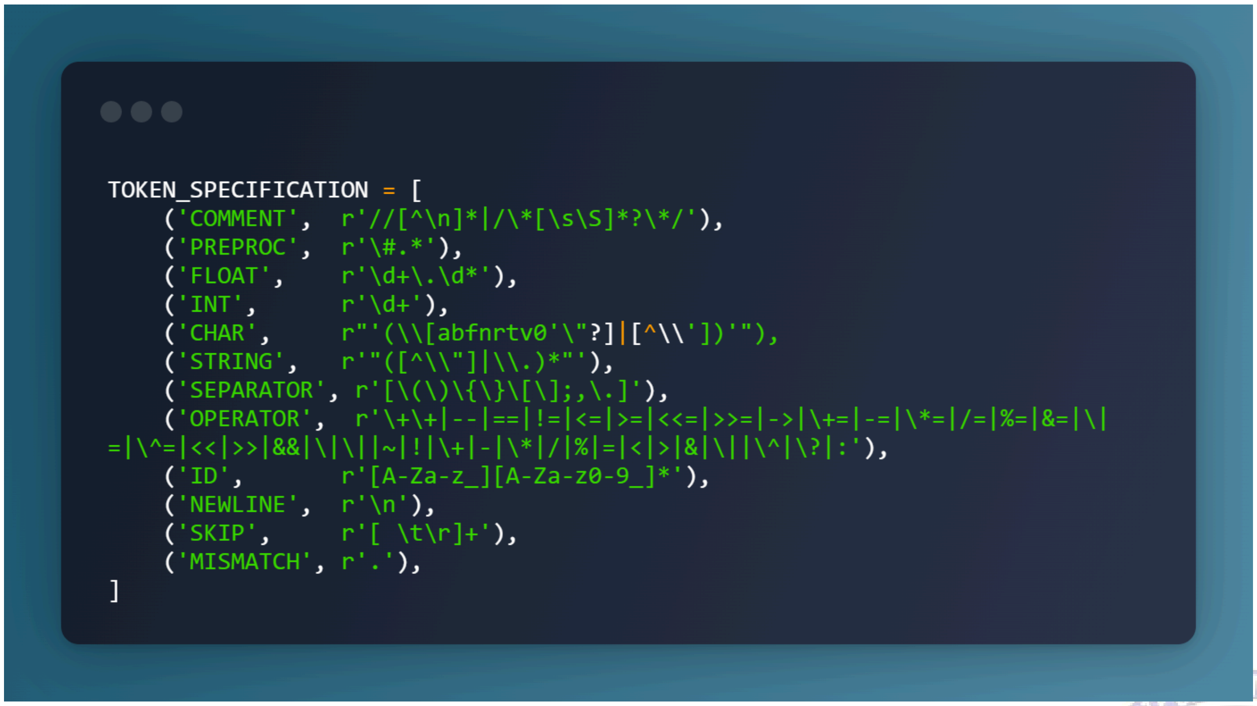


```
TOKEN_TYPE | token_text | line_number
```

c. Input/Output Handling: The script reads from a C++ file and writes tokens to a `.txt` output file.

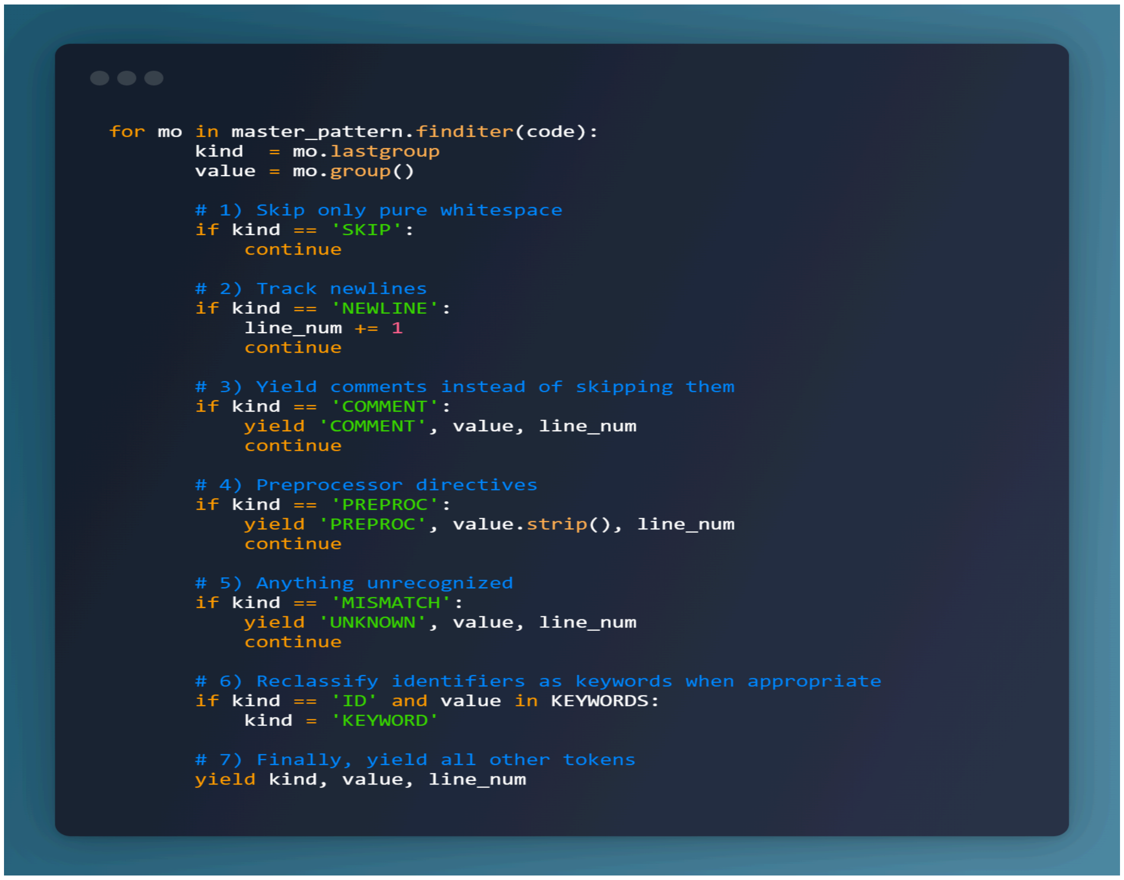
5. <u>Code Snippets</u>

➢ **Token Specification (from lexer.py):**



```
TOKEN_SPECIFICATION = [
    ('COMMENT',  r'//[^\n]*|/\*[\s\S]*?\*/'),
    ('PREPROC',  r'\#.*'),
    ('FLOAT',    r'\d+\.\d*'),
    ('INT',      r'\d+'),
    ('CHAR',     r"'(\\[abfnrtv0'\"?]|[^\\'])'"),
    ('STRING',   r'"([^\\"]|\\.)*"'),
    ('SEPARATOR', r'[\(\)\{\}\[\];,\.]'),
    ('OPERATOR',  r'\+\+|--|==|!=|<=|>=|<<=|>>=|->|\+=|-=|\*=|/=|%=|&=|\|
=|\^=|<<|>>|&&|\|\||~|!|\+|-|\*|/|%|=|<|>|&|\||\^|\?|:'),
    ('ID',       r'[A-Za-z_][A-Za-z0-9_]*'),
    ('NEWLINE',  r'\n'),
    ('SKIP',     r'[ \t\r]+'),
    ('MISMATCH', r'.'),
]
```

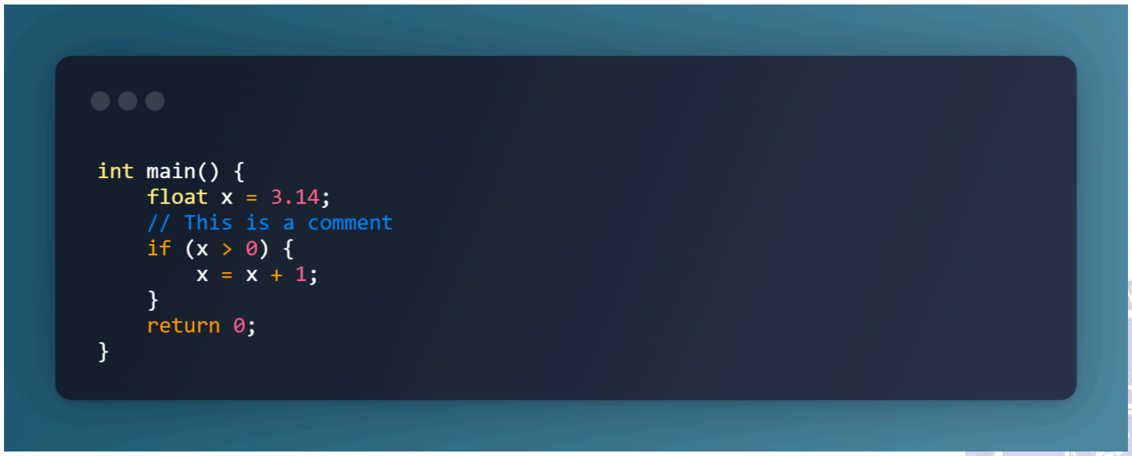
➢ **Tokenizing Loop:**



```
for mo in master_pattern.finditer(code):
        kind  = mo.lastgroup
        value = mo.group()

        # 1) Skip only pure whitespace
        if kind == 'SKIP':
            continue

        # 2) Track newlines
        if kind == 'NEWLINE':
            line_num += 1
            continue

        # 3) Yield comments instead of skipping them
        if kind == 'COMMENT':
            yield 'COMMENT', value, line_num
            continue

        # 4) Preprocessor directives
        if kind == 'PREPROC':
            yield 'PREPROC', value.strip(), line_num
            continue

        # 5) Anything unrecognized
        if kind == 'MISMATCH':
            yield 'UNKNOWN', value, line_num
            continue

        # 6) Reclassify identifiers as keywords when appropriate
        if kind == 'ID' and value in KEYWORDS:
            kind = 'KEYWORD'

        # 7) Finally, yield all other tokens
        yield kind, value, line_num
```

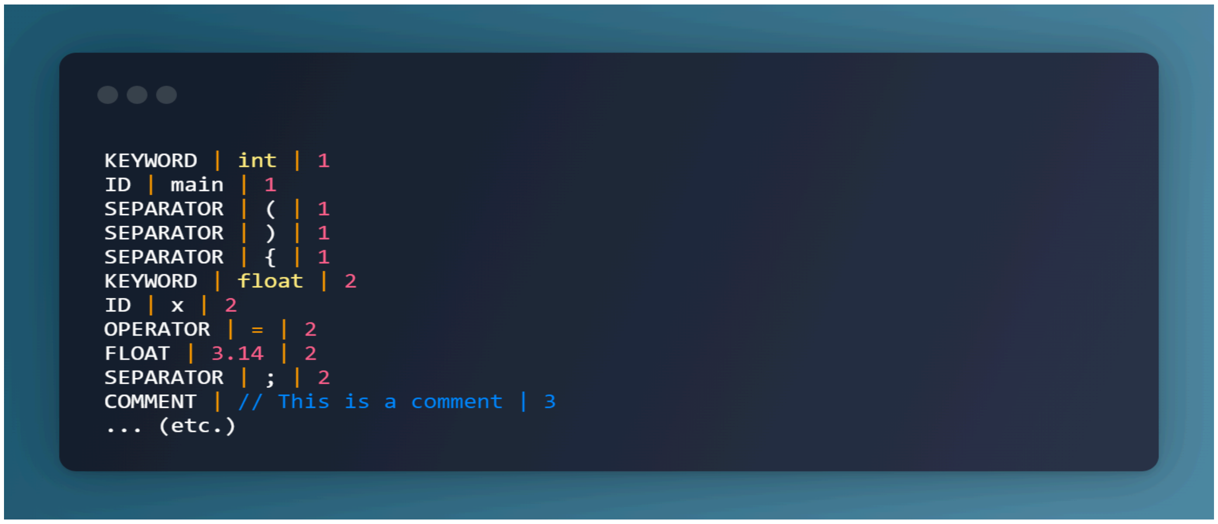
6. Sample Input and Output

➢ **Sample Input (input.cpp):**



```
int main() {
    float x = 3.14;
    // This is a comment
    if (x > 0) {
        x = x + 1;
    }
    return 0;
}
```

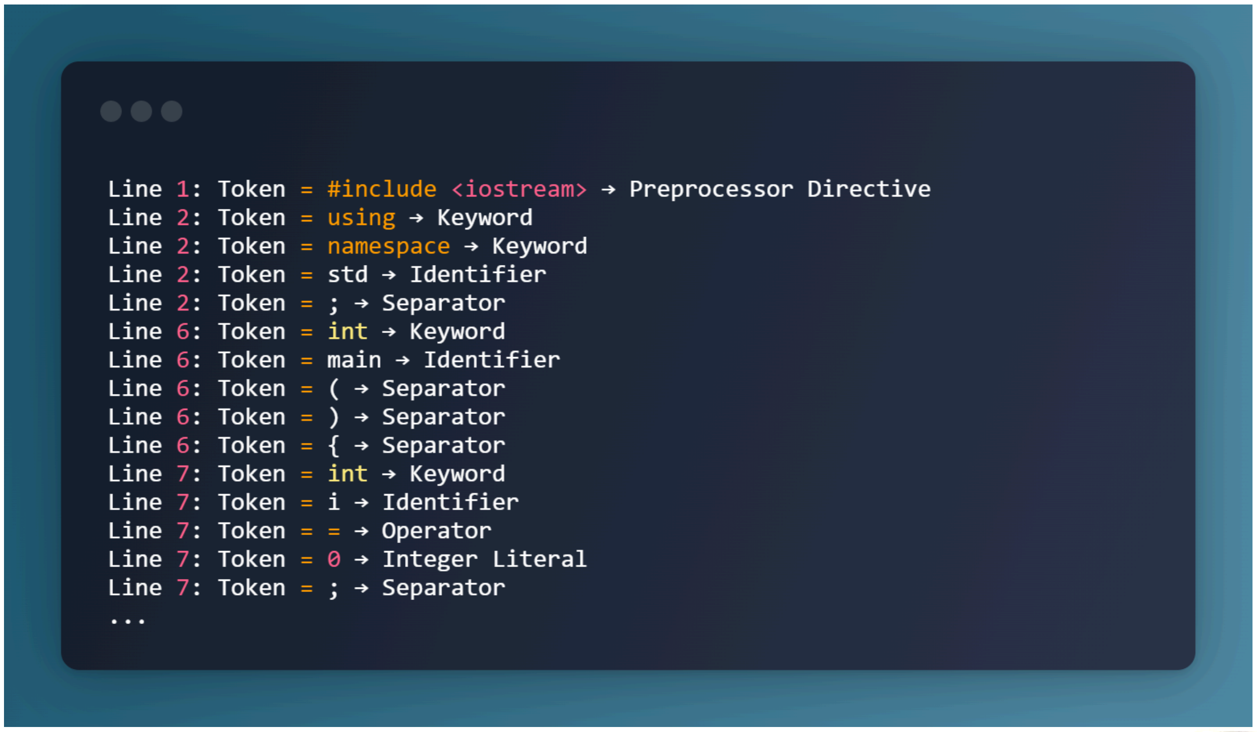
➢ **Sample Output (output.txt):**



```
KEYWORD | int | 1
ID | main | 1
SEPARATOR | ( | 1
SEPARATOR | ) | 1
SEPARATOR | { | 1
KEYWORD | float | 2
ID | x | 2
OPERATOR | = | 2
FLOAT | 3.14 | 2
SEPARATOR | ; | 2
COMMENT | // This is a comment | 3
... (etc.)
```

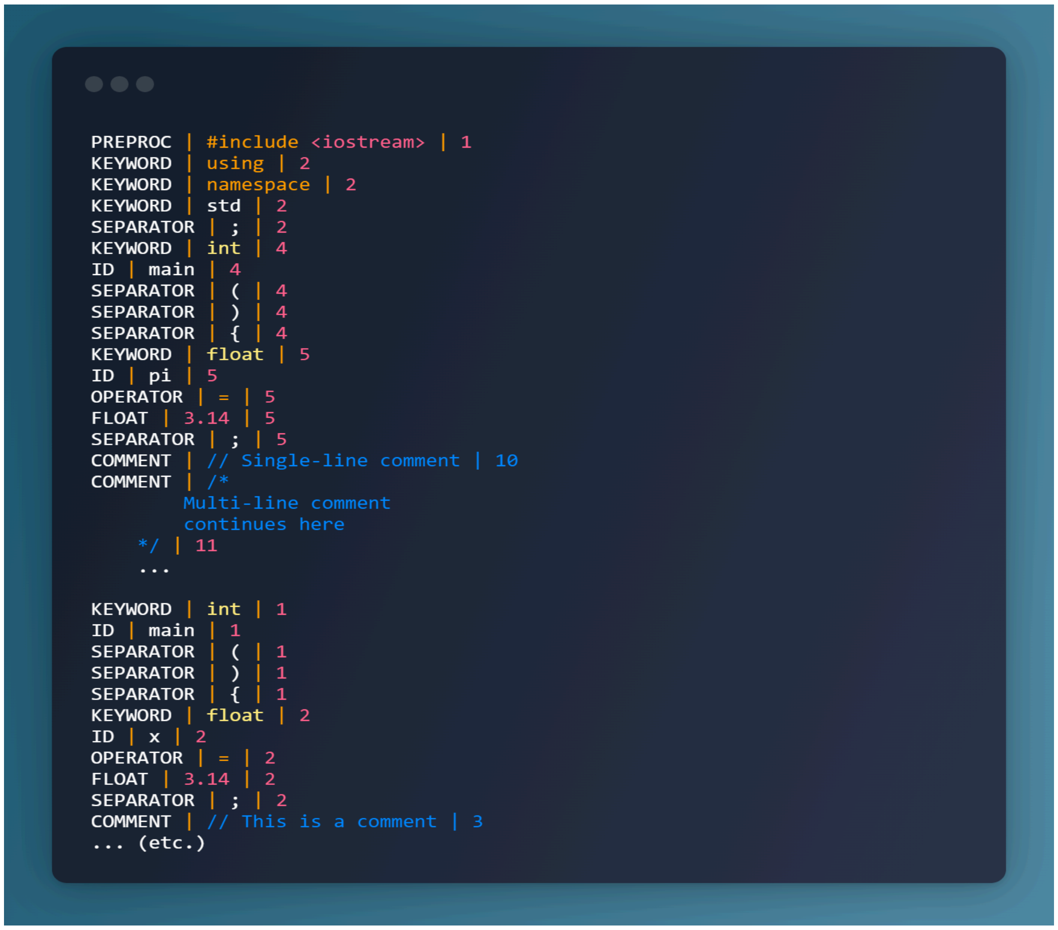
➢ **Additional Outputs:**

➢ **Output1.txt:**



```
Line 1: Token = #include <iostream> → Preprocessor Directive
Line 2: Token = using → Keyword
Line 2: Token = namespace → Keyword
Line 2: Token = std → Identifier
Line 2: Token = ; → Separator
Line 6: Token = int → Keyword
Line 6: Token = main → Identifier
Line 6: Token = ( → Separator
Line 6: Token = ) → Separator
Line 6: Token = { → Separator
Line 7: Token = int → Keyword
Line 7: Token = i → Identifier
Line 7: Token = = → Operator
Line 7: Token = 0 → Integer Literal
Line 7: Token = ; → Separator
...
```

➢ **Output8.txt:**



```
PREPROC | #include <iostream> | 1
KEYWORD | using | 2
KEYWORD | namespace | 2
KEYWORD | std | 2
SEPARATOR | ; | 2
KEYWORD | int | 4
ID | main | 4
SEPARATOR | ( | 4
SEPARATOR | ) | 4
SEPARATOR | { | 4
KEYWORD | float | 5
ID | pi | 5
OPERATOR | = | 5
FLOAT | 3.14 | 5
SEPARATOR | ; | 5
COMMENT | // Single-line comment | 10
COMMENT | /*
        Multi-line comment
        continues here
    */ | 11
    ...

KEYWORD | int | 1
ID | main | 1
SEPARATOR | ( | 1
SEPARATOR | ) | 1
SEPARATOR | { | 1
KEYWORD | float | 2
ID | x | 2
OPERATOR | = | 2
FLOAT | 3.14 | 2
SEPARATOR | ; | 2
COMMENT | // This is a comment | 3
... (etc.)
```

7. Challenges and Learnings

Challenges:

- Handling escaped characters inside string and char literals.
- Writing regex patterns that support multi-line comments.
- Ensuring operators like **++, +=, ==** are matched before + or =.
- Managing line numbers correctly for tokens on the same line.

Learnings:

- Stronger understanding of how regular expressions can be composed.
- Improved grasp of how lexical analyzers work under the hood.
- Exposure to Python's **re** module and tokenization mechanics.

8. Conclusion

This project successfully **recreated a C++ lexical analyzer in Python** without using third-party libraries. It mirrors the functionality of the Flex-based analyzer from the warm-up task and extends it to handle comments explicitly. By working through regular expressions and manual token extraction, we gained a deep understanding of how tools like **Flex tokenize** source code internally.

The project output is clean, modular, and extensible, making it a solid base for future compiler or interpreter components.