CS3104 Compiler Lab, Autumn 2025
Assignment 4

Date: 29 August 2025

Instructions: Write your name and roll number on top of your source code as comments. Submit two files: your Lex file (`Assignment4.l`) and your C/C++ file (`SimpleLexer.c` or `SimpleLexer.cpp`).
Submit your code at: `http://10.22.10.100/~abyaym/CS3104/submission/`

Building upon your Flex expertise from Assignment 3, you will now develop a lexical analyzer for a **Simple Programming Language**. This assignment introduces you to handling the complexity of a real programming language with multiple data types, control structures, and advanced token recognition patterns.

Problem Statement
Design and implement a lexical analyzer using Flex for a simple programming language called **SimpleLang**. Your lexer must recognize all language constructs, handle different number formats and string literals, support comments and basic preprocessor directives, and provide comprehensive error handling with token statistics.

SimpleLang Language Specification
Your lexical analyzer must handle the following language elements:

- **Keywords:** `int`, `float`, `string`, `if`, `else`, `while`, `function`, `return`, `true`, `false`
- **Identifiers:** Multi-character names starting with letter, followed by letters or digits (e.g., `variable`, `temp`, `counter123`)
- **Numbers:**
  - Integers: `123`, `456`, `0`
  - Floats: `3.14`, `2.5`, `0.001`
  - Hexadecimal: `0xFF`, `0x1A2B`
- **String Literals:** Double-quoted strings with basic escape sequences: `"Hello World"`, `"Line 1\nLine 2"`, `"Quote: \"text\""`
- **Operators:**
  - Arithmetic: `+`, `-`, `*`, `/`
  - Assignment: `=`
  - Comparison: `==`, `!=`, `<`, `>`
  - Logical: `&&`, `||`, `!`
- **Delimiters:** `(`, `)`, `{`, `}`, `;`, `,`
- **Comments:** Single-line (`//`) and multi-line (`/* */`)
- **Preprocessor Directives:** Basic recognition of `#include`

Sample SimpleLang Program

```
#include <stdio.h>
#define MAX_SIZE 100

// Function to calculate sum
function int sum(int a, int b) {
    return a + b;
}

/* Main program */
function int main() {
    int x = 10;
    float y = 3.14;
    string message = "Hello, SimpleLang!";
    bool flag = true;

    if (x > 5) {
        int result = sum(x, 20);
        printf("%s Result: %d\n", message, result);
    }

    // Test different number formats
    int hex_val = 0xFF;
    float sci_val = 1.23e5;

    return 0;
}
```

Listing 1: Example SimpleLang Program Input

Expected Token Output Format

For the sample program above, your lexer should produce output similar to:

```
Line 1: PREPROCESSOR(#include) OPERATOR(<) IDENTIFIER(stdio.h) OPERATOR(>)
Line 2: PREPROCESSOR(#define) IDENTIFIER(MAX_SIZE) INTEGER(100)
Line 4: COMMENT_SINGLE(// Function to calculate sum)
Line 5: KEYWORD(function) KEYWORD(int) IDENTIFIER(sum) DELIMITER(()
        KEYWORD(int) IDENTIFIER(a) DELIMITER(,) KEYWORD(int) IDENTIFIER(b)
        DELIMITER()) DELIMITER({)
Line 6: KEYWORD(return) IDENTIFIER(a) OPERATOR(+) IDENTIFIER(b) DELIMITER(;)
Line 12: KEYWORD(int) IDENTIFIER(x) OPERATOR(=) INTEGER(10) DELIMITER(;)
Line 13: KEYWORD(float) IDENTIFIER(y) OPERATOR(=) FLOAT(3.14) DELIMITER(;)
Line 14: KEYWORD(string) IDENTIFIER(message) OPERATOR(=)
         STRING("Hello, SimpleLang!") DELIMITER(;)
Line 22: KEYWORD(int) IDENTIFIER(hex_val) OPERATOR(=) HEX_INTEGER(0xFF)
         DELIMITER(;)
Line 23: KEYWORD(float) IDENTIFIER(sci_val) OPERATOR(=) FLOAT(1.23e5)
         DELIMITER(;)
```

Listing 2: Expected Token Stream (Partial)

Core Requirements

- **Token Classification:** Implement proper token categorization with subtypes (INTEGER, FLOAT, HEX_INTEGER, STRING, etc.)
- **String Processing:** Handle basic escape sequences in strings (`\n`, `\t`, `\"`, `\\`)
- **Number Recognition:** Support integers, floats, scientific notation, and hexadecimal numbers
- **Comment Handling:** Process both single-line and multi-line comments properly

- **Error Recovery:** Continue processing after encountering invalid tokens and report errors with line numbers
- **Statistics Generation:** Provide token counts, keyword frequency, and basic program analysis

Practice Problems
Test your lexical analyzer with the following scenarios:

1. **Number Format Testing:** Process various numeric literals:

```
int decimal = 12345;
float pi = 3.14159;
int hex_value = 0xABCD;
float small = 0.001;

```

2. **String Literal Processing:** Handle strings with escape sequences:

```
string simple = "Hello World";
string with_newline = "Line 1\nLine 2";
string with_quote = "He said \"Hello\" to me";
string empty = "";

```

3. **Comment and Operator Testing:** Test comments and operators together:

```
// Single line comment
int x = 5; // End of line comment
/* Multi-line comment */
if (x == 5 && y != 10) {
    result = x + y * 2;
}

```

4. **Function Definition:** Process complete function with control structures:

```
function int calculate(int x, float y) {
    if (x > 0) {
        while (x < 10) {
            x = x + 1;
        }
        return x;
    } else {
        return 0;
    }
}

```

5. **Error Handling:** Test lexer's error recovery capabilities:

```
int valid_var = 123;
int invalid@name = 456;        // Invalid character
string bad_string = "unterminated string;
float bad_number = 12.34.56;   // Malformed number

```

Output Requirements
Your lexical analyzer should provide:

- **Token Information:** Token type, lexeme, and line number for each token
- **Error Reporting:** Clear error messages with line numbers for invalid tokens
- **Statistics Summary:** Total token count, keyword frequency, and identifier list
- **Token Distribution:** Breakdown of token types found in the input

Implementation Guidelines

- Use Flex states for handling strings, comments, and preprocessor directives
- Implement proper pattern ordering to avoid conflicts (longer patterns first)
- Handle whitespace appropriately (ignore but track line numbers)
- Provide meaningful error messages for common mistakes
- Use appropriate data structures for collecting statistics
- Test thoroughly with edge cases and malformed input

Compilation and Testing

```
# Compilation
$ flex Assignment4.l
$ gcc lex.yy.c SimpleLexer.c -o simplelexer -lfl
# OR for C++
$ g++ lex.yy.c SimpleLexer.cpp -o simplelexer -lfl

# Testing
$ ./simplelexer < program.sl        # File input
$ ./simplelexer                     # Interactive mode
```

This assignment challenges you to handle the complexity of a real programming language lexer. The skills you develop here in pattern recognition, state management, and error handling will be essential for understanding how modern compilers process source code. Take your time to design clean, efficient patterns and thorough error handling.

Good luck building your SimpleLang lexer!