CS3104 Compiler Lab, Autumn 2025
Assignment 3

Date: 22 August 2025

Instructions: Write your name and roll number on top of your source code as comments. Submit two files: your Lex file (`Assignment3.l`) and your C/C++ file (`procltx.c` or `procltx.cpp`). Submit your code at: `http://10.22.10.100/~abyaym/CS3104/submission/`

Building upon your experience from Assignments 1 and 2, you will now develop a **Lexical Analyzer using Lex/Flex** for your calculator language. This assignment introduces you to formal lexical analysis tools and helps you understand how tokens are recognized and processed in a systematic way.

Problem Statement

Your task is to build a lexical analyzer using Lex/Flex that can tokenize the calculator language from Assignment 2, recognize different types of tokens (numbers, variables, operators, keywords), handle whitespace and comments, generate a token stream with appropriate token types, and provide meaningful error messages for invalid tokens.

Token Specification

Your lexical analyzer should recognize the following tokens:

- **IDENTIFIER:** Variable names (single letters a-z, A-Z)
- **NUMBER:** Positive integers (one or more digits)
- **ASSIGN:** Assignment operator (`=`)
- **PLUS, MINUS, MULTIPLY, DIVIDE, MODULO:** Arithmetic operators (`+`, `-`, `*`, `/`, `%`)
- **LPAREN, RPAREN:** Parentheses (`(`, `)`)
- **QUIT, EXIT:** Exit keywords (`quit`, `exit`)
- **NEWLINE:** End of statement (`\n`)
- **WHITESPACE:** Spaces and tabs (to be ignored)
- **COMMENT:** C-style comments (`/* ... */` and `// ...`)

Input Format and Example Output

```
Input:
x = 10 + y * 2  // This is a comment
z = (a - b) % 5
/* Multi-line
   comment here */
quit

Expected Token Stream:
IDENTIFIER(x) ASSIGN NUMBER(10) PLUS IDENTIFIER(y) MULTIPLY NUMBER(2) NEWLINE
IDENTIFIER(z) ASSIGN LPAREN IDENTIFIER(a) MINUS IDENTIFIER(b) RPAREN MODULO
    NUMBER(5) NEWLINE
QUIT
```

Listing 1: Sample Input and Expected Token Output

Core Requirements

- **Lex File (Assignment3.l):** Define regular expressions for all token types, handle comments and whitespace appropriately, provide token codes or symbolic names, and implement error handling for invalid characters.
- **C/C++ Integration File (procltx.c/procltx.cpp):** Process tokens generated by Lex, display token information in a readable format, maintain a simple token counter, and demonstrate integration between Lex and C/C++ code.
- **Token Display:** For each recognized token, display the token type and lexeme (actual text), handle special cases like keywords vs identifiers, and show line numbers for better debugging.

Sample Implementation Structure

```
%{
#include <stdio.h>
#include <stdlib.h>
int line_num = 1;
int token_count = 0;
%}

%%
[a-zA-Z]        { /* Process IDENTIFIER */ }
[0-9]+          { /* Process NUMBER */ }
"="             { /* Process ASSIGN */ }
"+"             { /* Process PLUS */ }
/* Add more patterns here */
%%

int main() {
    yylex();
    return 0;
}
```

Listing 2: Basic Lex File Structure (Assignment3.l)

Practice Problems

Complete the following exercises to test your lexical analyzer:

1. **Basic Tokenization:** Test with simple expressions like `a = 5 + 3`. Verify that each token is correctly identified and that the token count is accurate.
2. **Complex Expression:** Process the expression `result = (x + y * 2) / (z - 1) % 10` and ensure proper handling of all operators and parentheses.
3. **Comments and Whitespace:** Test with input containing both single-line and multi-line comments:

```
// Variable assignments
x = 10  /* this is x */
y = 20 + /* inline comment */ 5
/* Multi-line comment
   spanning several lines */
z = x * y

```

Verify that comments are properly ignored and don't affect tokenization.

4. **Nested Comments and Edge Cases:** Handle complex nested comment scenarios and edge cases:

```
a = 5 /* outer comment /* inner comment */ still outer */ + 3
b = 10 // comment with /* symbols inside single-line comment
c = /* comment at start */ 20 /* comment at end */
// /* Mixed comment styles */
d = 15 /* comment with // inside */ * 2

```

   Ensure your lexer correctly handles nested comments or reports appropriate errors if your implementation doesn't support nesting.

5. **Large Number Handling and Boundary Cases:** Test with various number formats and boundary conditions:

```
bigNum = 999999999999999999
leadingZeros = 0000012345
singleDigit = 0
maxInt = 2147483647
/* Test expression with very large numbers */
huge = 123456789 * 987654321 + 1111111111

```

   Verify proper tokenization of large numbers and handle potential overflow scenarios gracefully.

Advanced Features (Optional)

- **Line Number Tracking:** Maintain and display line numbers for each token.
- **Symbol Table Integration:** Extend your tokenizer to work with a simple symbol table.
- **Enhanced Error Reporting:** Provide detailed error messages with line and column numbers.
- **Token Statistics:** Generate statistics about token frequency and types.

Implementation Guidelines

- Use Lex/Flex regular expressions efficiently and avoid conflicts between patterns.
- Handle edge cases like empty input or files with only comments.
- Ensure your C/C++ code properly integrates with the generated lexer.
- Test with various input combinations including erroneous input.
- Document your token definitions clearly in comments.

Error Handling Requirements

Your lexical analyzer should handle the following error conditions:

- **Invalid Characters:** Report unknown symbols with line numbers.
- **Unterminated Comments:** Detect and report unclosed multi-line comments.
- **Malformed Numbers:** Handle cases like numbers with invalid characters.

Compilation Instructions

To compile and run your lexical analyzer:

```
# For C implementation:
$ flex Assignment3.l
$ gcc lex.yy.c procltx.c -o lexer -lfl
$ ./lexer < input.txt

# For C++ implementation:
$ flex Assignment3.l
$ g++ lex.yy.c procltx.cpp -o lexer -lfl
$ ./lexer < input.txt
```

This assignment introduces you to the power of formal lexical analysis tools. The techniques you learn here will be essential for building more sophisticated language processors and understanding how modern compilers work. The systematic approach of Lex/Flex will help you appreciate the advantages of tool-based development over manual parsing.

Good luck and enjoy exploring lexical analysis!