Lab 3: Building a Scanner

Due: March 28, 2025

Group of 3 students

Objective

This lab will challenge students to build a lexical scanner (tokenizer) using Flex. The scanner must:

- Process a complete C source file as input.
- Recognize and categorize tokens.
- Output in a structured format, so an automated grading script can verify it.

AI tools can generate a working solution for this assignment, but relying on them without understanding the process will leave you unprepared for exams, interviews, and real-world programming challenges. The real value of this lab is in learning how tokenization works, experimenting with regex, and debugging your own Flex scanner. If you write the code yourself, you'll gain practical skills that AI alone cannot teach, like problem-solving, debugging, and pattern recognition. I strongly encourage you to attempt the assignment yourself, discuss with your peers, and not to worry about points. There will be bonus works to recover points if you lose some.

Scanner Requirements

Your scanner must tokenize and classify the following set of categories:

Category	**Examples**	**Additional Complexity**
Keywords	`int, return, void, while, typedef, static, struct`	Detect more reserved words used in C.
Identifiers	`variable_name, functionName, _myVar`	Ensure valid names, prevent keyword misclassification.
Constants	`123, 3.14, 0x1A3F, 042, 'A'`	Support hex (0x...), octal (0...), and char literals ('c').
Operators	`+, -, *, /, =, +=, ==, &&, ||, !, >>, <<, &`	Detect multi-character operators like +=, ==, >>, <<.
Punctuations	`;, {, }, (, ), [, ], ,`	Include brackets [ ] and commas.
Preprocessor Directives	`#include <stdio.h>, #define MAX 100, #ifdef DEBUG`	Expand to handle #ifdef, #ifndef, and #undef.
Strings	`"Hello, world!"`	Handle escaped characters (\", \n, \\ inside strings).

Comments	`// Single-line comment, /* Multi-line comment */`	Ensure comments are fully ignored (not in token output).
Whitespace & Formatting	`space, tab, newline`	Ignore formatting characters.
Special (Unrecognized Tokens)	`@, $, ?, <=>`	Detect invalid symbols or unknown operators.

Expected Input & Output

Example Input File (input.c)

```
#include <stdio.h>

typedef struct {
    int x;
    float y;
} Point;

void printPoint(Point p) {
    printf("Point: (%d, %.2f)", p.x, p.y);
}

int main() {
    Point p1 = {10, 20.5};
    printPoint(p1);
    return 0;
}
```

Expected Scanner Output (output.txt)

```
< PREPROCESSOR, #include<stdio.h> >
< KEYWORD, typedef >
< KEYWORD, struct >
< PUNCTUATION, { >
< KEYWORD, int >
< IDENTIFIER, x >
< PUNCTUATION, ; >
< KEYWORD, float >
< IDENTIFIER, y >
< PUNCTUATION, ; >
< PUNCTUATION, } >
< IDENTIFIER, Point >
< PUNCTUATION, ; >
< KEYWORD, void >
< IDENTIFIER, printPoint >
< PUNCTUATION, ( >
< IDENTIFIER, Point >
< IDENTIFIER, p >
< PUNCTUATION, ) >
< PUNCTUATION, { >
< IDENTIFIER, printf >
< PUNCTUATION, ( >
< STRING, "Point: (%d, %.2f)" >
< PUNCTUATION, , >
< IDENTIFIER, p >
< PUNCTUATION, . >
< IDENTIFIER, x >
< PUNCTUATION, , >
< IDENTIFIER, p >
< PUNCTUATION, . >
< IDENTIFIER, y >
< PUNCTUATION, ) >
< PUNCTUATION, ; >
< PUNCTUATION, } >
< KEYWORD, int >
< IDENTIFIER, main >
< PUNCTUATION, ( >
< PUNCTUATION, ) >
< PUNCTUATION, { >
< IDENTIFIER, Point >
< IDENTIFIER, p1 >
< OPERATOR, = >
< PUNCTUATION, { >
< CONSTANT, 10 >
< PUNCTUATION, , >
< CONSTANT, 20.5 >
< PUNCTUATION, } >
< PUNCTUATION, ; >
< IDENTIFIER, printPoint >
< PUNCTUATION, ( >
< IDENTIFIER, p1 >
< PUNCTUATION, ) >
< PUNCTUATION, ; >
< KEYWORD, return >
< CONSTANT, 0 >
< PUNCTUATION, ; >
< PUNCTUATION, } >
```

Submission Requirements

Submit a single zip file named as:
ID1_ID2_ID3_Name1_Name2_Name3_Lab3.zip

Contents of the Zip File:

- tokenizer.l → The Flex file containing all regex patterns for tokenization.
- Makefile → A Makefile with `make` command to automate compilation.
- report.pdf → Includes:
    - Name & IDs of all three students.
    - Screenshots of tokenized output.
    - Explanation of Regular Expressions used.
    - Any extra features added (e.g., handling error cases).