Simple Scanner

Objective

The objective of this lab is to develop a lexical analyzer using Flex that can tokenize various elements of C code, including different types of numbers, operators, keywords, identifiers, string literals, and comments. By the end of this lab, students should be able to:

- Understand the basic usage of Flex to generate a lexer.
- Define patterns for tokenizing different types of numbers, operators, identifiers, keywords, and delimiters.
- Ignore comments and whitespace effectively.
- Print tokens in a readable format.

Prerequisites

- Make sure **g++** is installed. `$g++ --version$`
- Make sure **flex** is installed.`$flex --version$`
- Latest `CSC355_Student` repository

Details

In the **Lab2** directory, you will find *lexer.l* file and *tests* sub-directory, which holds 10 test files (*.txt) and 10 expected output files (*.out). Please see below for the details of *lexer.l* file.

Token Types

You will tokenize the inputs into KEYWORDS, IDENTIFIER, NUMBER, OPERATOR, DELIMETER, and STRING_LITERAL. The token types are predefined for you in the *lexer.l* file. Any input that does not belong to any of the types will be classified as UNKNOWN.

```
/* Token types */
    typedef enum {
    T_KEYWORD,
    T_IDENTIFIER,
    T_NUMBER,
    T_OPERATOR,
    T_DELIMITER,
    T_STRING_LIT,
    UNKNOWN
} TokenType;
```

Token Struct

Token struct is a data structure that represents each token. This structure has two members: *type* and *value*. *type* will hold the type of token (e.g., T_KEYWORD, etc.) *value* will hold the actual string value of the token (e.g., *int*, etc.)

```
/* Token structure */
typedef struct {
    TokenType type;
    char value[256];
} Token;
```

getToken() function

getToken() function will create, read input, complete the token object, and call **printToken()** function to print the token information in a formatted structure.

```
void getToken(TokenType t_type) {
    Token token;
    token.type = t_type;
    strncpy(token.value, yytext, sizeof(token.value) - 1);
    // The null terminator ('\0') is used in C to mark the end of a string.
    token.value[sizeof(token.value) - 1] = '\0';
    printToken(token);
}
```

error() function

Any input not belonging to token types will be treated as an error. Although this is an error, the lexer will not stop. Instead, it will print a message and continue to the next token.

```
void error() {
    char value[256];
    strncpy(value, yytext, sizeof(value) - 1);
    std::cout << "Input: " << value << " is " << "UNKNOWN" << std::endl;
}
```

Tasks

First, in the top comment section, you will write your name slash ('/') Davidson username as an "Author" of the program. For example,

```
/*
 * ....
 *
 *  Author: Terrence Lim/telim
 */
```

My script parses this to identify the program's owner, so please don't forget to write your name and Davidson username delaminated by a slash.
Your task is to complete the *lexer.l* file, compile with *flex* and *g++*, handle all test files. Since our goal is to write a small lexical analyzer, we are going to handle only the following keywords: $"int", "float", "string", "if", "else", "while", "return", and "printf"$. There are two parts in the file you have to complete: *Definitions* and *Rules*.

Definitions

In our lexer, we will only have four definitions: *DIGIT*, *LETTER*, *ID*, and *NUMBER*. For each definition, you will write a regular expression (regex). For example, the regex for *DIGIT* is already given to you, i.e., $DIGIT[0-9]$ means *DIGIT* will match any single decimal character. Please see the following explanation to create your regex:

- *LETTER*: Any single character that is a lowercase letter, an uppercase letter, or an underscore.
- *ID*: Matched one or more characters that are either letters, digits, or underscores. Identifiers must start with a letter.
- *NUMBER*: Match different formats of numeric literals in the input. This includes integers (e.g., 1, 10, 111, etc.), floating-point numbers with optional decimal points (e.g., 1.1, 0.5, etc.), and numbers in scientific notation (e.g., 1e10, 2E10, etc.).

Rules

You will specify the rules to classify matched lexemes. For example, the provided example code: `"int"|"float"|...|"return"|"printf" getToken(T_KEYWORD);` (Please see the code for the complete list) means that if the next read lexeme (a sequence of characters matched by the pattern) is one of the specified keywords (int, float, return, print, etc.), the function getToken will be called with the argument T_KEYWORD.

- You will only handle the following operators: `+,-,*,/,=,<,>,>=,<=,!=,==,&`.
- You will only handle the following delimiters: `;, (, ), {, }, [, ], comma(,)`.
- You will create a rule for handling a single character surrounded by apostrophes. For example, `char x = 'a';`. Here `x` is an identifier while `'a'` is a character token, `T_CHAR_LIT`. Your rule needs to handle any single character except an apostrophe, e.g., `'''` is not allowed, or an empty character, e.g., `''` is allowed. A character cannot have more than one character, e.g., `'ab'` is not allowed.
- You will create a rule for handling a string, e.g., `string str = "this is a string";` for the `T_STRING_LIT` token. `string` is a keyword, `str` is an identifier, and `"this is a string"` is a string (`T_STRING_LIT`). Your rule needs to handle any zero or more characters wrapped with double quotation marks ("").
- You will create rules for all the token types plus handling single-line comment (//), multiple-line comment (/* ... */), and whitespace (space, tab, and newline). We will *ignore* comments and whitespace.
- You will create a rule for invalid identifiers. For this assignment, an invalid identifier starts with a number, e.g., `int 1abc;` (see test file t007.txt).
- All other unrecognized or unhandled patterns will be treated with an **error()** function.

How to Compile and Test Your Code

```
$flex lexer.l
$g++ -o lexer lex.yy.c -lfl
$./lexer test_file.txt
```

- `$flex lexer.l`: This will generate a C source file (*lex.yy.c*) that implements the lexer based on these rules.

- `$g++ lex.yy.c -o lexer_<username> -lfl`: Compiles the generated C source file (*lex.yy.c*) into an executable binary (`lexer_<username>`). Replace `<username>` with your Davidson username. `-lfl` option tells *g++* compiler to link the Flex library.

Expected Output

Given the following test input (*t001.txt*):

```
int a = 1;
int b = 1000000;
float c = 1.0;
float d = 1.1;
```

You are expecting to see the following output:

```
<T_KEYWORD, 'int'>
<T_IDENTIFIER, 'a'>
<T_OPERATOR, '='>
<T_NUMBER, '1'>
<T_DELIMITER, ';'>
<T_KEYWORD, 'int'>
<T_IDENTIFIER, 'b'>
<T_OPERATOR, '='>
<T_NUMBER, '1000000'>
<T_DELIMITER, ';'>
<T_KEYWORD, 'float'>
<T_IDENTIFIER, 'c'>
<T_OPERATOR, '='>
<T_NUMBER, '1.0'>
<T_DELIMITER, ';'>
<T_KEYWORD, 'float'>
<T_IDENTIFIER, 'd'>
<T_OPERATOR, '='>
<T_NUMBER, '1.1'>
<T_DELIMITER, ';'>
```

How to Submit Your Code

The only file you will submit is **lexer.l**. Under the `Labs` directory, create a sub-directory called `Lab2`. You will add your **lexer.l** file. Add, commit, and push your file to GitHub.