Computer Engineering 175
Phase I: Lexical Analysis

Polonius: "What do you read, my lord?"
Hamlet: "Words, words, words."
Shakespeare, *Hamlet*, Act II

1 Overview

In this assignment, you will write a lexical analyzer for the Simple C language. This assignment is worth 10% of your course grade. Your program is due at 11:59 pm, Sunday, January 13th.

2 Lexical Structure

The following points summarize the lexical rules for Simple C:

- spaces, tabs, newlines, and other whitespace are ignored; it is formally defined by the regular expression `(␣|\t|\n|\f|\v|\r)`$^+$
- comments are surrounded by `/*` and `*/` and may not include a `*/`
- an identifier consists of a letter or underscore followed by optional letters, underscores, and digits; a letter is one of `a`, `b`, ..., `z`, `A`, `B`, ..., `Z`; a digit is one of `0`, `1`, ..., `9`; it is formally defined by `[_a–zA–Z][_a–zA–Z0–9]`*
- keywords are `auto`, `break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `extern`, `float`, `for`, `goto`, `if`, `int`, `long`, `register`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `union`, `unsigned`, `void`, `volatile`, and `while`; keywords may not be used as identifiers
- an integer consists of one or more digits; it is defined by the regular expression `[0–9]`$^+$
- a real number consists of one or more digits, a decimal point, one or more digits, and an optional exponent, which itself consists of the letter `e` in either lower or upper case followed by an optional sign and then one or more digits; it is defined by `[0–9]`$^+$`.[0–9]`$^+$`([eE][+–]`$^?$`[0–9]`$^+$`)`$^?$
- a string is enclosed in double quotes (`"` ... `"`) and may not contain a double quote or a newline; it is defined by the regular expression `"[^\n"]`*`"`
- operators are `=`, `||`, `&&`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `+`, `-`, `*`, `/`, `%`, `&`, `!`, `++`, `--`, `.`, `->`, `(`, `)`, `[`, `]`, `{`, `}`, `;`, `:`, and `,`
- any other character is illegal

3 Assignment

You will write a simple lexical analyzer for Simple C by reading the ***standard input*** (`std::cin`) one character at a time. Although a lexical analyzer can be viewed as generating a stream of tokens, it is typically written as a function returning tokens to a calling function (i.e., a parser). A simple main program that calls your lexical analyzer will be provided. This program expects your analyzer function to be declared as `int lexan(string &lexbuf)`. Your function will store the matched text in `lexbuf` and return an integer token value. For example, if the keyword `if` is matched, `lexbuf` should contain `if` and your function should return `KEYWORD`. As another example, if the string `"hello"` is matched, `lexbuf` should contain `"hello"` and your function should return `STRING`.

Your program will only be given ***lexically correct*** programs as input. However, it is strongly advised that you test your program against lexically incorrect programs (e.g., unterminated strings and comments, invalid real literals, invalid operators, etc.) as a way of finding errors in your implementation.

4 Hints

Most of the constructs are easily recognized by their first character. For example, integers and reals start with a digit, and keywords and identifiers start with a letter or underscore. You will find it easiest to use `cin.get()`, and if you wish, `cin.peek()` or `cin.putback()`. Also, you will find the functions such as `isdigit()` defined in `<cctype>` very helpful.

Although the constructs are specified as regular expressions, you will not find it very helpful to use a regular expression library as part of your implementation. Such libraries are designed to match strings held in buffers against regular expressions, not to read an input file and tokenize it.