Programming Project #1

Objective: Demonstrate your understanding of lexical analysis by writing a program that performs partial lexical analysis of standard C source code as described below.

Requirements:

1. **Development:** Your program must be written in either C or Java and you may use any development tool: Visual Studio, Eclipse, Cygwin gcc/make, etc. However, your program must be portable by using standard ANSI C(C89) or Java SE 13 – compatible with Windows, Linux, and Mac OS. Use warning level 4 for C.
2. **Execution:** Your program must be executable from a command prompt with the input and output file names as command-line arguments. If your program is called "lexscan", the terminal command "`lexscan mycode.c lexlog.txt`" or "`java –jar lexscan.jar mycode.c lexlog.txt`" (from either a Windows DOS or Linux command line prompt) would perform the required lexical analysis using the file `mycode.c` as the input source code and produce the required output in the ASCII text file `lexlog.txt`. Suitable error messages must be displayed if command line arguments are missing or data files cannot be properly opened. The only other console output should be a final summary as described below when program arguments and files are good.
3. **Program Coding Requirements:**
   a. Program must be designed and implemented such that main() opens data files and produces all of the required output. In order to do so, it must repeatedly call your lexical scan function which performs all input operations and sends back each token-lexeme and related information needed by main(), in a C struct or Java object of your design. Basically, main() contains a very simple loop that calls the lexical scan function to obtain the next token and then display its data until an "EOF" token is returned. Main() would also be responsible for counting tokens and displaying the final summary.

   b. C lexemes cannot span multiple lines because newline chars are whitespace that terminate everything except multi-line comments. String and char literals not properly terminated by their closing quotes are reported as errors by compilers. However, for this project, the input file will be syntactically correct C code so all string and char literals will be properly terminated by their closing quote. Another limitation to consider is the maximum length of any lexeme. Programmers are not likely to create obnoxiously long identifiers so, realistically, the longest lexemes would be string literals with an ISO required minimum of 509 chars and actual maximums depend on compiler implementations (Visual Studio 2017 was 2048 and I did not go looking for an update). For this project, use a maximum input line size of 128 which would include any leading tabs or spaces and all other chars up to but not including the newline char. That means the largest possible lexeme would be a variable name with 128 chars or a 126-char string literal surrounded by its quotes, filling a single line with no leading spaces or tab chars. Comment lines could also fill 128 chars but multi-line comments are not combined to make one long comment. Integers and floats could also be very long but will not exceed 128 chars. Use a programmer-defined constant with a value of 128 upon which variables, structures, or objects are constructed as needed.

   c. Location of each lexeme must be reported as a pair of line and column numbers that identify the position in the code file where the lexeme begins. A tab character in the input stream moves the column position to the position immediately following the next multiple of 4 (typical behavior in most IDEs but almost all can change that setting). Example: a tab character in columns 5-8 will move the position indicator to column 9. You can actually see the cursor's col position in most IDEs in a status line at the bottom of the editing pane. The source code file will always end with a newline char followed by an empty line.

4. **Lexical Analysis:** Your program must recognize and report the location, token class (output TOKEN IDs shown in parentheses below), and actual lexeme text for the following token groups:

   a. Preprocessor directives (PPDIR) – any line that begins with a "#". Remainder of line is not processed for any other tokens. Technically, rules vary between compilers but we will require the # to appear in the first column and will not check directive names for validity.

   b. Identifiers (IDENT) & Keywords (KEYWD) – everything that begins with an alphabetic character (ignore possibility of underscores as first character of an identifier but underscores are allowed internally). You must implement a mechanism to recognize the standard C keywords. See http://en.wikibooks.org/wiki/C_Programming for an online list of the 32 keywords (ANSI C/C89 and ISO C/C90). Technically, maximum size of an identifier can vary between compilers with a minimum of 31 significant chars but we will ignore size limitations except as discussed in paragraph 3.b. Note, the maximum in Visual Studio 2017 is actually 247!

   c. Integer literals (INTGR) – simple decimal literals only (ignore possible negative sign, type suffix, and hexadecimal and octal formats). Technically, the maximum unsigned 64-bit integer would be 20 decimal digits, or 16 hex digits plus 2-char prefix 0x, or 22 octal digits plus 1-char prefix 0 but it should be the compiler, not the scanner, that enforces such limitations based on target architecture.

   d. Floating point literals (FLOAT) – simple decimal literals only (ignore possible negative sign, type suffix, and scientific notation). Like with integers, it should be the compiler that enforces limitations.

   e. String literals (STRNG) – C string literals and nothing inside the string is processed for tokens. Do NOT convert special escape sequences but they may exist. Overall limitation for lines and strings was covered in paragraph 3.b.

   f. Character literals (CHAR) – C ASCII char literals – do NOT convert escape sequences to their actual representation but they may exist. Do not enforce length limitation.

   g. Comments (CMMNT) – C block style only (not single-line // comments) and nothing inside the comment is processed for tokens. Can be multi-line but consider each line part of same comment but reported separately. Continuation lines will start in column 1 because any char is just more of the comment. The multi-line comments will need some finesse in your logic.

5. **Output** – Your program must report the location of all lexemes from the token groups described above:

   a. Each occurrence of a lexeme is reported on a separate line of output in the format:
      "`(###, ###) → token → Lexeme`"
      where *token* is one of the TOKEN IDs described above and *Lexeme* is the actual source text. The first ### is the line number and the second ### is the column number, separated by a comma and a single space. The numbers must be right-aligned in 3-character fields and the " → " between the items represents a single tab character. Note: the quotation marks do not appear in the output. However, string lexemes must be surrounded by " ", char lexemes must be surrounded by ' ', and comments must be surrounded by their beginning and terminating symbols. If a newline character is encountered while processing a comment, the output of the comment will generate a new line of output with a new line number with column position 1 and the comment token indicator. See sample input and output on the Moodle assignment page.

Output continues on next page…

5. **Output:** (continued)

   b. Output must also contain a final summary statement as follows, printed to output file and displayed on the console screen:

```
Processed  ####  lines
           ####  PPDIRs
           ####  IDENTs
           ####  KEYWDs
           ####  INTGRs
           ####  FLOATs
           ####  STRNGs
           ####  CHARs
           ####  CMMNTs
```

Deliverables: Upload source code to Moodle – use a zip file if you create a project with multiple code files.

Due Date: Monday, Mar 4th (before midnight)