Project Two: A Lexer

For this project you will write a lexer (that is, a tokenizer) (in the C programming language) that will parse lines made up of simple math and logical expressions and break them down into meaningful tokens. This project will be modified and used to complete your next C programming project.

NOTE: There are string tokenizer functions (**strtok**, **strtok_r**, and **strsep**) that are part of the C library. **You may not use these functions!** They will not function correctly in the context of this program.

Tasks: You will write a function called **get_token** and a **main** function. The **get_token** function should have the following prototype:

void get_token(char *token);

- The function will have a character pointer as an argument. This pointer will point to a character array (declared in **main**) that **should** contain the last token discovered after the function returns. The **char*** that is passed to the **get_token** function will contain the next token in line when it, **get_token** returns. So if I had the line **12=17**, AFTER the first time I ran **get_token(var)**, **var** would contain the string ``12" and the second time it would contain the string ``=".

- There will be a global character pointer called **line** that will point to the current line of input.

- Your tokenizer should recognize only the following lexemes and their token categories. So your code has to explicitly look for these specific lexemes.
  - + ADD_OP
  - - SUB_OP
  - * MULT_OP
  - / DIV_OP
  - (LEFT_PAREN
  -) RIGHT_PAREN
  - ^ EXPON_OP
  - = ASSIGN_OP
  - < LESS_THEN_OP
  - <= LESS_THEN_OR_EQUAL_OP
  - > GREATER_THEN_OP
  - >= GREATER_THEN_OR_EQUAL_OP
  - == EQUALS_OP
  - ! NOT_OP
  - != NOT_EQUALS_OP
  - ; SEMI_COLON
  - [0-9]+ INT_LITERAL
    - an integer literal (that is, one or more occurrences of any digit); the perl compatible regular expression for this is [0-9]+ (the + at the end means 1 or more occurrences)

- When you see a ``;" that means you have reached the end of a statement.
- A newline only represents the end of a line. Newlines do not end statements! For example, the output with this input:

12=
17--;

is identical to the output with this input:

12=17--;

but, tokens cannot span lines, for example.

12
34

returns two tokens, 12 and 34, not one token 1234.

- There can be whitespace inside or before or after an expression. Whitespace can be blank characters, tab characters, or newline characters. For example, **2 + 5** is a valid expression that has some whitespace inside it.
- Whitespace is one way to delimit between tokens. However, it is possible to start a new token without whitespace between the new token and the previous token. For example,

  - **12=4+7;** has 6 tokens
  - **0!=8** has 3 tokens
  - **===** has 2 tokens

- You can assume that only one statement can be in a single line.
- You need to handle the case of a character sequence that is not a lexeme. You need to output the character sequence and that it is a not a lexeme. You need to then be able to continue to recognize the lexemes that follow that character sequence. For example,

12 = 4 @ + 2;

should generate the error message

===> '@'
Lexical error: not a lexeme

- Your program has to take input from an input file specified as a command line argument and send the output to an output file specified by a command line argument. For example,

  ./tokenizer input.txt output.txt

- Your output file should start each statement with

  Statement #

  followed by the number of the statement starting at one.
- Your output file should include a line of dashes after the last token in a statement. The last token in a statement is a semicolon.

- Your program should read lines of input from a file which the user specifies and print out all the tokens discovered for every statement.
- You can use as many helper functions as you would like. You need to put your function prototypes, #define preprocessor directives, and typedefs in a header file that your #include in your main file. You can put some functions in another file if you wish.
- On the course website are two example input files and the required corresponding output files.
- Your output files must look like these output files.
- These input file might have both blank characters and tab characters. Use the command

  od -c input.txt

  or

  od -c input_errors.txt

  to see the escape characters (including the tab character near the end). Do not try to create these input files yourself. You will very likely not have all the escape characters in the copy on the course website. These input file are just examples; I can use other input files to test your program.

- The example input file **input_errors.txt** contains the following visible characters:

  1 <= 2;

  (2 ^ 2) != 10 +
  7 ;

  2 * 19
  + (5/0);

  1@ === 2;

  2 ** 9 ;

  3 * @

- The output for that input file when the command

  ./**tokenizer input_errors.txt output_errors.txt**

  is run is on the course website. Note that at the start of each line

  Statement # and the statement number

  is printed and after each statement ends a line of dashes is printed. Note that the last sequence of tokens does not have a terminating semi-colon so no line of dashes appears.
- Code which you must use as part of your solution is provided on the course website.
- Your solution will be graded based on how it compiles and works when run on the departmental

Linux server, agora. There should be no warnings or errors when compiled with the **-Wall** flag with gcc.

- When debugging your program by using printf() statements always end the string inside the printf with a newline character, '\n', as in "got to point A\n". The newline character forces the string to be printed to the console instead of being held indefinitely in the output buffer of the input/output runtime.
- You can use regular expressions in your solution. You probably would want to use Perl Compatible Regular Expressions (PCRE). You need in your .c file

#include <pcre.h>

-

and at the command line compile with linking it the library for pcre by

gcc -lpcre filename.c t

to use PCRE. Note that regex.h in the C language refers to POSIX regular expressions which are very different from PCRE. I checked and pcre.h is on agora in /usr/include/pcre.h and the library also exists to be linked in.

Style and Documentation

Your program must follow good programming style as reflected in the programs in the McDowell textbook. The programming style should also be consistent with the Sun's Java Coding Standard (see http://www.oracle.com/technetwork/java/codeconv-138413.html) to the extent possible given that your program is in C instead of Java.

Your program must be fully commented which means

- On agora create a project2 directory that has in it a README text file and the files for your project source code. The README file lists the author names, date, that this is Project 2, explains how to compile and run your program (including showing the exact commands to be entered), a description of the purpose of the program, and any aspects of the program that do not work. You will lose fewer points for parts that do not work that you tell me about.
- every file must have a start-of-file comment at the top with your name, date, and the name of the project and a brief description of what the code in that file does
- every function must have a comment above the header which is like a javadoc method header comment (including describing every parameter and return value).
- Comment other lines of code when you think it will help the readability of your program

Grading and Turn-In

If your program does not compile, the score is zero.

Projects can be turned in late with my normal late policy. That policy is that every day late is three points off (so 5:01 pm is one day late) not including weekends and holidays up to a maximum of thirty late points. The last day to turn in late projects is the Friday of the next to last week of classes which is the 26th of April.

Call your program **tokenizer.c**. If your program is composed of multiple files tar everything up into one file for submission. You must turn in as part of your tar file a readme file that includes your author

names, the data of submission, a description of what is the project files and that you have done, and identify any errors that still exist in your program.

The project is due at 5 pm on Friday, the 20th of March. Submit your files as a gzipped tar file via **handin** on agora. If your tar file is called **project2.tar.gz**, then the command will be:

handin.352.1 2 project2.tar.gz

If you are in the directory containing the project2 directory as a subdirectory, then the command

```
tar -cvzf project2.tar.gz project2
```

will create the project2.tar.gz file in the current directory.