**Programming Assignment #1 – Scanner**

For your first programming assignment you are being asked to write a scanner (also referred to as a lexical analyzer, tokenizer, lexer, etc.). Essentially, you will be writing a function (or functions if you want to separate logic) that will spit out the next token of a given input stream. Again you will be submitting your assignment to gradescope (the autograder will be available starting next Monday, and the due date is Friday, Sept. 16 at 11:00PM).

**The “Calculator Language” Tokens**

The tokens that your scanner should “recognize” are listed below. These were taken from Example 2.9 of the optional textbook for this course. Pseudocode and a finite automaton for the scanner are available in the 03-scanning lecture slides, which are available on Blackboard.

assign  –> ‘:’ ‘=’  
plus    –> ‘+’  
minus   –> ‘-’  
times   –> ‘*’  
div     –> ‘/’  
lparen  –> ‘(’  
rparen  –> ‘)’  
id      –> letter (letter | digit)*  
           except for ‘read’ and ‘write’  
number  –> digit digit*| digit* (‘.’ digit | digit ‘.’) digit *comment –> ‘/’ ‘*‘ (non-*| non-/)* ‘*’+ ‘/’  
         | // (non-newline)* newline

Note that the list of tokens in the example also includes the following:

letter –> a..z | A..Z  
digit –> 0..9

These, however, are not something that you need to recognize individually, they are just useful for building the ‘id’ and ‘number’ tokens from smaller parts. For example, you’ll read a character at a time from a file and simply use a pair of functions (mentioned below) from the standard library to determine if the character is a letter or digit.

**Provided Code**

I will be providing a main file (scanner\_main.cpp), a header file (calc\_scanner.h), and makefile. The makefile is for you to use if you develop your code on Grok, which would be a good idea. The main function is fairly simple and it does the following:

1. Make sure that a command line argument was provided (it exits with an error if no such file is provided).
2. Opens a file with the given filename (it exits with an error if the file does not open).
3. Finally, it continually calls the get\_net\_token function until that function returns a token that indicates that we've reached the end of the file (the END token).

The provided calc\_scanner header file first defines an [enum](http://en.cppreference.com/w/cpp/language/enum" \t "_blank). This enum enumerates all tokens that our scanner should be able to recognize (for example, we should be able to identify that a ':' followed by a '=' is an ASSIGN token). Next, the header file defines the Token class. The token class has two member varaibles: a TOKEN (which corresponds to the enum) and a std::string. The string variable will be assigned text from the file. In addition to the member variables, the Token struct also has a member function called to\_string, which just returns a string representation of the given Token object.

The final part of the header file is the function declaration for the function that you will be implementing--this is described in the next section.

You do not have to use the makefile that I have provided, but it should make development a bit easier. While developing your code, I'd recommend that you start out simple, just keep all five (scanner\_main.cpp, calc\_scanner.h, calc\_scanner.cpp, makefile, and a test input file) of the needed files in the same directory on Grok. Then, you can compile your program by running the "make" command. In this case, make runs the makefile, which is equivalent to the following compile command: g++ -std=c++14 -Wall scanner\_main.cppcalc\_scanner.cpp. Here is an example showing how you can compile and run your code (I have some extra command-line calls as well):

aclark@grok:~/courses/csc333/programming-assignments/pa1$ ls  
calc\_scanner.cpp  calc\_scanner.h  makefile  scanner\_main.cpp  test.calc  
aclark@grok:~/courses/csc333/programming-assignments/pa1$ make  
g++ -MT scanner\_main.o -MMD -MP -MF .d/scanner\_main.Td -Wall -std=c++14   -c -o scanner\_main.o scanner\_main.cpp  
mv -f .d/scanner\_main.Td .d/scanner\_main.d  
g++ -MT calc\_scanner.o -MMD -MP -MF .d/calc\_scanner.Td -Wall -std=c++14   -c -o calc\_scanner.o calc\_scanner.cpp  
mv -f .d/calc\_scanner.Td .d/calc\_scanner.d  
g++ -Wall -std=c++14    -o scanner scanner\_main.o calc\_scanner.o  
aclark@grok:~/courses/csc333/programming-assignments/pa1$ ./scanner test.calc  
ASSIGN : :=  
PLUS : +  
more-output-that-I-deleted

The make file does some extra stuff that you do not need to worry about at the moment, but if your curious I'd recommend reading the following [post about makefile dependency management](http://make.mad-scientist.net/papers/advanced-auto-dependency-generation/).

**Programming Task**

You task is to implement the **get\_next\_token** function in a file that you created called “calc\_scanner.cpp” (everything else has been provided for you). Your function should be very similar to the pseudocode listed in the book (and in the lecture slides). However, I have added a few extra tokens and changed the requirements a bit.

You’ll see in “calc\_scanner.h” that I have added the following tokens: **COMMENT**, **END**, **TOKEN\_ERROR**. The **COMMENT** token is used to save the commented text–we are doing this instead of trashing the text. The **END** token should be return by your function if the token stream is empty (if we’ve reached the end). Finally, **TOKEN\_ERROR** should be return any time you find an invalid character. You can see an example with these tokens below.

You can add as many helper functions as you’d like, but you will also not be counted off if you put your code in a single function (though, that may make it difficult for you to debug).

The signature for the get\_next\_token function is as follows:

Token get\_next\_token(std::istream &token\_stream)

This is a function that returns a Token object, and takes an input stream (std::istream) as an input. Let's look at these two components.

A Token, which has been defined in the calc\_scanner.h file can be created in a number of ways. The easiest/intuitive way (though, not necessarily the best way) is to do so like this:

// Creating an ASSIGN token  
Token new\_token;  
new\_token.token = TOKEN::ASSIGN;  
new\_token.text = ":=";

The input (token\_stream) is defined as a std::istream for one reason: it gives our function some flexibility. The get\_next\_token function will be able to read tokens from any of the following: std::cin, std::istringstream, std::ifstream, and many more. If we were to require anifstream as an input we would only be able to read from files (not from the standard input or from stringstreams).

**Example Input File**

The text below can be copied into a normal text file and fed to your program for testing purposes. Please note that the given text file has some invalid tokens in it.

:= + - \* / ( ) anID read  
write 10 .2345 2341.1234 // a single line comment  
// another single line comment  
/\**a  
multi-line  
comment \**/

badnumber := 123.456.789

a := 0  
b : 0  
c = 1

When I run my scanner with the above strings I get the following output:

ASSIGN : :=  
PLUS : +  
MINUS : -  
TIMES : \*  
DIV : /  
LPAREN : (  
RPAREN : )  
ID : anID  
READ : read  
WRITE : write  
NUMBER : 10  
NUMBER : .2345  
NUMBER : 2341.1234  
COMMENT : a single line comment  
COMMENT : another single line comment  
COMMENT : a  
multi-line  
comment  
ID : badnumber  
ASSIGN : :=  
TOKEN\_ERROR : Invalid number, too many ‘.’ characters.  
ID : a  
ASSIGN : :=  
NUMBER : 0  
ID : b  
TOKEN\_ERROR : Unexpected character following a colon:  
NUMBER : 0  
ID : c  
TOKEN\_ERROR : Unexpected character: =  
NUMBER : 1  
END : end

**Notes**

* You should develop your function incrementally. Start by figuring out how to identify the simple tokens (‘+’, ‘-‘, etc.), and then move onto the harder ones.
* To get a single character from the input stream you can use the ‘get’ method. For example: token\_stream.get(c); (where ‘c’ is an already defined character)
* You can check to see if the stream is empty with the following code: if (!token\_stream) { /*do something*/ }
* You can check for letters, digits, and spaces with std::isalpha and std::isdigit, and std::isspace, respectively.