HW #03 – C #2
CSCI-400 Spring 2013

In this assignment will build on your previous assignment to implement a lexer for a simple calculator language that you will later (as in the next assignment) write a parser and interpreter for.

The language is constrained to the following alphabet of 24 characters:

Digits: {**0–9,R,E,.,(,),=,^,*,/,+,–,;,' ','\n'**}

Note that the last two are the space and the newline characters and that the set does NOT include either the comma or the apostrophe. Comments can ideally consist of graphical ASCII characters plus spaces, tabs, and, in the case of block comments, newline characters, but are actually unrestricted on what they can contain (except end-of-line comments can't contain newline characters).

Your program will scan a source code file and produce a token file consisting of token value pairs from the following list of fifteen tokens. Except as noted, the value simply consists of the literal lexeme associated with the token. If the value is shown in square brackets, then it is optional.

<ID> R0 through R9
<INT> value
<FLT> value
<OPAREN> [(]
<CPAREN> [)]
<ASSIGN> [=]
<EXP> [^]
<MUL> [*]
<DIV> [/]
<ADD> [+]
<SUB> [–]
<SEMI> [;]
<BAD> [All characters with certain replacements]
<NEWLINE> [source code line number]
<EOLCMT> [First twenty characters with certain replacements]
<BLKCMT> [First twenty characters with certain replacements]
<END> [number of valid tokens]

VALID TOKENS

The "valid" tokens are those that would be used by the parser to generate code. Most of these tokens are single-character literals and how to recognize them is pretty obvious. Some of these others require some explanation.

HW #03 – C #2
CSCI-400 Spring 2013

The **<ID>** token

This token has possible lexeme values of {**R0**, **R1**, **R2**, …, **R9**} and represents a register (or a variable) in this language. The lexeme value is required for this token.

The **<INT>** token

This token recognizes unsigned integers. The value is required and is the lexeme consisting of a string of digits. Note that a negative integer will lex as a **<SUB>** token followed by an **<INT>** token. Similarly, a '**+**' character preceding an integer should be lexed as an **<ADD>** token.

The **<FLT>** token

This token recognizes unsigned floating point values in either fixed-point or exponential form. The value is required and is the lexeme that is found. In fixed-point form, the lexeme must contain a decimal point and at least one digit, but the decimal point can be located anywhere in the lexeme, including as the first character or as the last character. In exponential form, the mantissa can be either an integer or a fixed-point floating point value as described above. The mantissa is immediately followed by a capital '**E**' (you may choose to recognize a lower case '**e**' as well), followed by a signed integer exponent. Note that the exponent may be negative and the '**–**' character is NOT a **<SUB>** token in this case (although if the mantissa is negative, it will lex as a **<SUB>** token followed by an **<FLT>** token). Similarly, a '**+**' character before the mantissa should result in an **<ADD>** token while one before the exponent should be accepted.

DEBUGGING TOKENS

The following tokens would normally not be exported by the lexer. They are included to help you, and the grader, make better sense of the token file, particularly if problems arise. They are not worth significant points, but implementing them may improve your grade as a result of these benefits

The **<BAD>** token

This token is used whenever the lexer is unable to form a valid token. Any time that the lexer encounters a character that does not extend the present proto-lexeme but for which the present proto-lexeme is not an actual lexeme (i.e., cannot be mapped to a valid token) should be mapped to the **<BAD>** token with the proto-lexeme as its (optional) value. The lexer should then be returned to the Start state and lexing continue starting with the character that did not extend the previous proto-lexeme (note that this character is NOT part of the bad token, it is the first character in the next proto-lexeme). The exception to this is if the lexer is in the Start state when the bad token is generated since this means that the present proto-lexeme is empty and the character that was read cannot even start a lexeme. In this case, the bad token's value will consist

of all characters encountered up to, but not including, the first character that can start a new valid proto-lexeme.

Because the source code file can contain ASCII codes that are not only not in this language's alphabet, but may not be printable or may be control codes that could have undesirable effects if printed to the screen or file, the following rules will apply to characters included in bad lexemes: If the character is a graphical ASCII character other than the pound sign, **#**, it will be included in the token value without modification. For the pound sign and all non-graphical ASCII characters and all non-ASCII characters, a three character sequence will be output that is of the form **#nn** where **nn** is the characters 2-digit value of the byte in hexadecimal.

The **<NEWLINE>** token

Whenever a newline is encountered that is NOT within a block comment, output this token with the number of the line in the source code file that is just starting. The first line, number 1, will not have a token and it will be understood that the first tokens, if not preceded by a **<NEWLINE>** token, are on line 1. Do not put out a token for newline characters that are within a block comment, but to increment the line counter so that the next **<NEWLINE>** token that is output will correspond to the correct line number in the source code.

The **<EOLCMT>** token

This token should be exported whenever an end-of-line comment is encountered. The value, if included, should be the first twenty characters of the comment (or all of them, if shorter than that), not including the **//** sequence that started the comment. The same rules for replacement of non-graphical ASCII characters apply here as for the **<BAD>** token except that spaces and pound signs may optionally be exported as literals. Remember, the trailing newline character is NOT to be considered part of the comment.

The **<BLKCMT>** token

This token should be exported whenever a block comment is encountered. The value, if included, should be the first twenty characters of the comment (or all of them, if shorter than that), not including the **/*** or the ***/** sequences that delimit the comment. The same rules for replacement of non-graphical ASCII characters apply here as for the **<BAD>** token except that spaces and pound signs may optionally be exported as literals.

The **<END>** token

This token should be exported upon detecting the end of the source code file. The value, if included, should be the total number of valid tokens in the file. Valid tokens are those that can be used by the parser to generate code and therefore does not include the **<BAD>** token or any of the debugging tokens, including this one.

HW #03 – C #2
CSCI-400 Spring 2013

THE FUNCTIONS

This section describes the interface specifications that your code must meet. The reasons for these specifications will become evident when we get to Flex and Bison.

Download the HW03.zip file from the course website. This file contains functioning skeleton code for your lexer program. The primary code that you will be focusing on will be in the function yylex() located in the file calc.c. You should not have to modify any of the other functions or do anything with the other files. You may choose to write additional functions to support your yylex() function and you are free to put these function either in the calc.c file or in separate files.

While your lexer must still process the input file one character at a time, you are free to use any functions from any of the C standard libraries for other purposes. In particular, you may be of interest in the ctype and the string libraries.

You should try to figure out how the skeleton program works. It may use features of the C language that you have not encountered before, so you may need to do some research. Feel free to post questions to Piazza about anything you can't figure out in a reasonable amount of time.

GRADING RUBRIC – 80 pts

20 - Good Faith effort.
18 - Single-character valid tokens recognized correctly. (2pts/ea)
10 - ID and INT tokens recognized correctly. (5pts/ea)
10 - FLT tokens recognized correctly.
5 - Debugging tokens recognized correctly. (1pt/ea)
5 - Tokens in output file use labels instead of integers.
10 - Code is reasonably modular and well documented.
2 - NULL pointer checks made were appropriate.
-10 - Improper submission.

SUBMISSION

Zip up all files into a .zip file named

`CS400_UserID_C_02.zip`