For the remainder of the semester we will be building a small program that interprets a small language. The language will have constants, a small number of keywords, and some operators.

The remainder of the semester will be broken into three pieces:

Program 2 - Lexical analyzer
Program 3 - Parser
Program 4 - Interpreter

For the lexical analyzer, you will be provided with a description of the lexical syntax of the language. You will produce a lexical analysis function and a program to test it.

The lexical analyzer function must have the following calling signature:

```
Token getNextToken(istream *in, int *linenumber);
```

The first argument to getNextToken is a pointer to an istream that the function should read from. The second argument to getNextToken is a pointer to an integer that contains the current line number. getNextToken will update this integer every time it reads a new line. getNextToken returns a Token. A Token is a class that contains a TokenType, a string for the lexeme, and the line number that the token was found on.

A header file, tokens.h, will be provided for you. It contains a declaration for the Token class, and a declaration for all of the TokenType values. You MUST use the header file that is provided. You may NOT change it.

The lexical rules of the language are as follows:

1. The language has identifiers, which are defined to be a letter followed by zero or more letters or numbers. This will be the TokenType IDENT.
2. The language has integer constants, which are defined to be one or more digits. This will be the TokenType ICONST.
3. The language has string constants, which are a double-quoted sequence of characters, all on the same line. This will be the TokenType SCONST.
4. The language has reserved the keywords print, if, then, true, false. They will be TokenTypes PRINT IF THEN TRUE FALSE.
5. The language has several operators. They are + - * / ( ) ; = == != > >= < <= && || which will be TokenTypes PLUS MINUS STAR SLASH LPAREN RPAREN SC ASSIGN EQ NEQ GT GEQ LT LEQ LOGICAND LOGICOR
6. A comment is all characters from a # to the end of the line; it is ignored and is not returned as a token. NOTE that a # in the middle of an SCONST is NOT a comment!
7. Whitespace between tokens can be used for readability. It serves to delimit tokens.
8. The newline should be recognized so that lines can be counted.
9. An error will be denoted by the ERR token.
10. End of file will be denoted by the DONE token.

Note that any error detected by the lexical analyzer should result in the ERR token, with the lexeme value equal to the string recognized when the error was detected.

Note also that both ERR and DONE are unrecoverable. Once the getNextToken function returns a Token for either of these token types, you shouldn't call getNextToken again.

The assignment is to write the lexical analyzer function and some test code around it.

It is a good idea to implement the lexical analyzer in one source file, and the main test program in another source file.

The test code is a main() program that takes several command line arguments:

-v (optional) if present, every token is printed when it is seen
-sum (optional) if present, summary information is printed
-allids (optional) if present, a list of the lexemes for all identifiers should be printed in alphabetical order
filename (optional) if present, read from the filename; otherwise read from standard in

Note that no other flags (arguments that begin with a dash) are permitted. If an unrecognized flag is present, the program should print "INVALID FLAG {arg}", where {arg} is whatever flag was given, and it should stop running.

At most one filename can be provided, and it must be the last command line argument. If more than one filename is provided, the program should print "TOO MANY FILE NAMES" and it should stop running.

If the program cannot open a filename that is given, the program should print "UNABLE TO OPEN {arg}", where {arg} is the filename given, and it should stop running.

The program should repeatedly call the lexical analyzer function until it returns DONE or ERR. If it returns DONE, the program proceeds to handling the -mci and -sum options, if any, and then exits. If it returns ERR, the program should print "Error on line N ({lexeme})", where N is the line number in the token and lexeme is the lexeme from the token, and it should stop running.

If the -v option is present, the program should print each token as it is read and recognized, one token per line. The output format for the token is the token name in all capital letters (for example, the token LPAREN should be printed out as the string LPAREN. In the case of token IDENT, ICONST, and SCONST, the token name should be followed by a space and the lexeme in parens. For example, if the identifier "hello" is recognized, the -v output for it would be ID (hello)

If the -sum option is present the program should, after seeing the DONE token and processing the -allids option, print out the following report:

Total lines: L
Total tokens: N
Total identifiers: I
Total strings: X

Where L is the number of input lines, N is the number of tokens (not counting DONE), I is the number of IDENT tokens, and X is a count of the number of SCONST tokens.

If N is zero, no further lines are printed.

If the -allids option is present, the program should, after seeing the DONE token, print out the string IDENTIFIERS: followed by lexemes for all of the identifiers, in alphabetical order, separated by commas. If there are no identifiers, then nothing is printed.

PART 1:

- Compiles
- Argument error cases
- Files that cannot be opened
- Too many filenames
- Zero length file

PART 2:

- Recognizes string with a newline in it as an error
- Recognizes string with a # in it as a string, not a comment
- Recognizes single character token types
- Supports -v mode

PART 3

- Recognizes identifiers
- Recognizes integer constants
- Supports -v mode