For the remainder of the semester we will be building programs that interpret 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 Program 2, 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 reference to an istream that the function should read from. The second argument to getNextToken is a reference to an integer that contains the current line number. getNextToken will update this integer every time it reads a newline. 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 ID.
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. A string constant can include escape sequences: a backslash followed by a character. The sequence \n should be interpreted as a newline. The sequence \\ should be interpreted as a backslash. All other escapes should simply be interpreted as the character after the backslash.
5. The language has reserved the keywords print, set, if, loop, begin, end. They will be TokenTypes PRINT SET IF LOOP BEGIN END.
6. The language has several operators. They are + - * / ( ) which will be TokenTypes PLUS MINUS STAR SLASH LPAREN RPAREN
7. The language recognizes a semicolon as the token SC
8. The language recognizes a newline as the token NL

9. 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!
10. Whitespace between tokens can be used for readability. It serves to delimit tokens.
11. An error will be denoted by the ERR token.
12. 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
-strings (optional) if present, print out all the string constants in alphabetical order
-ids (optional) if present, print out all of the identifiers 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 "UNRECOGNIZED 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 "ONLY ONE FILE NAME ALLOWED" and it should stop running.

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

The program should repeatedly call getNextToken until it returns DONE or ERR. If it returns DONE, the program proceeds to handling the -strings and -ids options, in that order. It should then print summary information and exit.

If getNextToken returns ERR, the program should print "Error on line N ({lexeme})", where N is the line number for 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 ID, 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).

The -strings option should cause the program to print STRINGS: on a line by itself, followed by every string constant found, one string per line, in alphabetical order. If there are no SCONSTs in the input, then nothing is printed.

The -ids option should cause the program to print IDENTIFIERS: followed by a comma-separated list of every identifier found, in alphabetical order. If there are no IDs in the input, then nothing is printed.

The summary information is as follows:

Total lines: L
Total tokens: N

Where L is the number of input lines and N is the number of tokens (not counting DONE).

If L is zero, no further lines are printed.

PART 1 - Due Feb 27

- Compiles
- Argument error cases
- Files that cannot be opened
- Too many filenames
- Properly handles a zero length file
- Recognizes keywords and identifiers
- Summary information
- -v mode

PART 2 - Due Mar 6

- Recognizes all remaining tokens
- 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 -strings and -ids