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:

```
Lex 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 should update this integer every time it reads a newline. getNextToken returns a Lex. A lex is a class that contains a Token, a string for the lexeme, and the line number that the Token was found on.

A header file, lex.h, will be provided for you. It contains a declaration for the Lex class, and a declaration for all of the Token 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 Token ID.
2. The language has integer constants, which are defined to be one or more digits. This will be the Token INT.
3. The language has string constants, which are a double-quoted sequence of characters, all on the same line. This will be the Token STR.
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, let, if, loop, begin, end. They will be Tokens PRINT LET IF LOOP BEGIN END.
6. The language has several operators. They are + - * / ! ( ) which will be Tokens PLUS MINUS STAR SLASH BANG LPAREN RPAREN
7. The language recognizes a semicolon as the token SC
8. A comment is all characters from // to the end of the line; it is ignored and is not returned as a token. NOTE that a // in the middle of an STR is NOT a comment!

9. Whitespace between tokens can be used for readability, and it serves to delimit tokens.
10. An error will be denoted by the ERR token.
11. 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 Lex for either of these tokens, 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
- -consts (optional) if present, print out all the string constants in alphabetical order followed by all the integer constants in numeric 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

The flag arguments (arguments that begin with a dash) may appear in any order, and may appear multiple times.

No other flags 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 -consts 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, INT, STR, and ERR, 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 -consts option should cause the program to print STRINGS: on a line by itself, followed by every unique string constant found, one string per line, in alphabetical order. If there are no STRs in the input, then nothing is printed. Then the program should print INTEGERS: on a line by itself, followed by every unique integer constant found, one integer per line, in numeric order. If there are no INTs 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:

Lines: L
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.

When you log into Vocareum, your workspace will be populated with a copy of lex.h, a zip file of all test cases and “.correct” files, and a shell script called runcase. DO NOT edit or delete any of these files. The zip file also contains a script called StudentTest, which you may execute to run individual test cases, same as “runcase” on Vocareum.

If you type “runcase” on Vocareum, or if you run “StudentTest” on your own system, the script prints a list of all test cases, showing what is run for each case.

For Part 1, you need only create a main that checks arguments and files. For Part 2, you should implement recognizing operators, identifiers and comments, and should support the -v flag. For Part 3, you must implement everything.

The test cases required for the three parts are listed below.

PART 1 - Due Oct 16

- Compiles
- badarg1, badarg2
- badfile, toomanyfile

PART 2 - Due Oct 23

- All part1 cases
- emptyfile, spacefile
- comments
- items1
- mix5
- err2
- err5

PART 3 - Due Oct 30

- All part2 cases
- All remaining cases