Part 1 - Due March 25
Part 2 - Due April 1

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:

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

For Assignment 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:

```
Tok getTok(istream& in, int& linenumber);
```

The first argument to getTok is a reference to an istream that the function should read from. The second argument to getTok is a reference to an integer that contains the current line number. getTok should update this integer every time it reads a newline. getTok returns a Tok. A Tok is a class that contains a Token, a string for the lexeme, and the line number that the Tok was found on.

A header file, lex.h, will be provided for you. It contains a declaration for the Tok 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 IDENT.
2. The language has integer constants, which are defined to be one or more digits with an optional leading sign. This will be the Token ICONST.
3. The language has string constants, which are a double-quoted sequence of characters, all on the same line. This will be the Token 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, println, repeat, begin, end. They will be Tokens PRINT PRINTLN REPEAT BEGIN END.
6. The language has several operators. They are + - * = / ( ) which will be Tokens PLUS MINUS STAR SLASH EQ 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 a SCONST is NOT a comment!
9. Whitespace between tokens can be used for readability, and can serve as one way 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 getTok function returns a Tok for either of these tokens, you shouldn't call getTok again.

Tokens may be separated by spaces, but in most cases are not required to be. For example, the input characters "3+7" and the input characters "3 + 7" will both result in the sequence of tokens ICONST PLUS ICONST.

The input characters "Hello" "World", and the input characters "Hello""World" will both result in the token sequence SCONST SCONST.

Telling the difference between 3 - 2 and 3 - -2 is easy because of the spaces delimiting the tokens. In both cases it's clear that the resulting tokens are ICONST MINUS ICONST (the second ICONST is -2, or negative 2). However, you must ALSO write code to tell the difference between 3-2 and 3--2, with no spaces between tokens. The solution for this case is to interpret the first minus of two consecutive minuses as a MINUS token, and the second minus as the beginning of an ICONST.

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 flags:

- -v (optional) if present, every token is printed when it is seen
- -iconsts (optional) if present, print out all the unique integer constants in alphabetical order
- -sconsts (optional) if present, prints out all the unique integer constants in numeric order
- -ids (optional) if present, print out all of the unique identifiers in alphabetical order

An optional filename argument may be passed to main. If present, your program should open and read from that 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. There can be at most one file name specified on the command line.

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.

If getTok 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.

The program should repeatedly call getTok until it returns DONE or ERR. If it returns DONE, the program prints summary information, then handles -sconst, -iconst and -ids, in that order.

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.

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, SCONST, 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 -sconsts 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 SCONSTs in the input, then nothing is printed.

The -iconsts option should cause the program to 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 ICONSTs 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 IDENTs in the input, then nothing is 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 two parts are listed below.

PART 1 - Due March 25

- Compiles
- Argument error cases
- Empty input files
- -v for IDENT and keywords
- Identifiers, keywords, and the -ids option

PART 2 - Due April 1

- Everything else

FINAL NOTE: read the entire assignment carefully and thoroughly. Think First, then code!

It is a VERY good idea to draw yourself a DFA of the patterns that the assignment will need to recognize.