CS 258 Spring 2017
Assignment 1
Command Language Interpreter

January 10, 2017

1 Purpose

In this assignment you will construct a main program and subroutine package that will form the basis of all future assignments. This assignment builds a *command language interpreter* (CLI) or "shell", although it is much simpler than most shells we use in practice. This project is worth 50 points.

2 Due Date

You must have turned your assignment in by midnight on January 26, 2017.

3 Part I: Command Language Interpreter (CLI)

3.1 Command Format

Your CLI must interpret commands either entered from a keyboard or read in from a file. The commands will look like:

```
COMMAND parm1, parm2, parm3
```

where COMMMAND is a character string designating the action to perform (typically the function to call), and the `parm1`, `parm2`, ..., `parmN` are parameters to be passed to that function. The parameters will usually be numeric, but some commands will take strings as inputs.

The parameters may be separated by commas or whitespace. Whitespace means any sequence of spaces and tab characters. I suggest you read about the C string routine `strtok`, which will help with this. If it is helpful, you may assume no command will have more than 32 parameters.

The parser should be case-insensitive to the *commands*, but case sensitive to the *parameters*. This means that READ, read, and ReAd are equivalent to the same command, but that "read Junk.cli" is not the same as "read junk.cli".

Comments may appear in a data file signaled by the "#" character. Ignore any text on a line after this character.

For this assignment first implement 3 dummy commands:

Command	Parameters	Description
Move	x, y, z	Move the pen to (x, y, z)
Draw	x, y, z	Draw with the pen to (x, y, z)
Color	v, h, s	Set the color to v, h, s.

Make the CLI just gather the parameters and print them out for now. If insufficient or incorrect parameters are given, do something intelligent. In particular, assume that unspecified values in the *Move* and *Draw* commands are set to 0. Thus, for example "Move 0,1" will be interpreted as "Move 0,1,0."

Also implement a "`Read`" command:

Command	Parameters	Description
Read	`filename`	Read commands from the file `filename`

The "*Read*" command should open the specified file and read input from it, line by line, until an EOF (end-of-file) is reached, then resume reading commands from the file the *Read* came from, or to the CLI prompt if the read command was typed in. Thus, *Reads* may be nested; that is, a file may *Read* another file which in turn reads another file, and so on, up to a reasonable number, say 6. You may also, however, be able to structure your program so that the *Read* is implemented via recursion.

3.2 Suggestions

The internals of the system are completely up to you. You should think about modularity and expandability, because you will be expanding on this assignment for the rest of the semester. For example, in future assignments you will implement commands such as "zoom" and "triangle." These commands take different numbers and types of parameters ("triangle" will take 12 parameters) and perform some graphical function. Below are some suggestions on how you might approach this project. These reflect *my* way of structuring a program. They are not the only way to do it and not even necessarily the best. *If my way of doing things confuses you, ignore it and figure out your own way.* What is important is to structure your program in a way that you find comprehensible, expandable, and debuggable.

3.2.1 Program Structure

A CLI type program consists of three types of routines.

1. The main program. The same main program should work for any of the different programs you write. The only thing that changes are the following two sections.
2. The "Dispatcher." The main program calls this and it calls the appropriate routines in the modules. The purpose of the dispatcher is to link the module entry points to the main program.
3. The Modules. These do the work. The set of modules that you link together determines what the program does.

3.2.2 Modules

You will write one or more of these each project. They are meant to be relatively self-contained, with well-defined interaction between them. A module will have entry points for

- Initialization.
- Command interpretation.
- Various access routines.
- Termination.

Each module will typically manage some small or local database and provide access routines to perform some functions on this database. The command interpretation routine is central to connecting the module into the CLI structure.

A module typically recognizes commands to set up or display values in its database, or call some action routine to do something, e.g., display a picture on the screen.

3.2.3 Dispatcher

The dispatcher has the same initialization, termination, and command interpretation routines as a module, and in fact looks very much like one. The dispatcher's command routine typically calls the command routines for all the modules you need for the current program. The initialization and Termination routines do likewise. If you like you can fold the functionality of the initialization, termination and command interpretation routines into one function. You may want to put some local commands into the dispatcher for debugging purposes. It is this routine, really, that determines the "personality" of the program.

3.2.4 The main CLI package

For the purpose of this course one main module should serve for all. It manages a database looking something like

- The current active file handle
- A stack of file handles from nested Reads
- The current input line
- A pointer to the last character parsed from above.

The code portion consists of

- The main program itself
- a `CLI_StringPar` function

Main Program. This does the following:

1. Initialize itself and all modules. One aspect of initialization is to initialize the graphics window. I will supply code for you to do this. Your part will be written on top of the code I supply you.
2. Read the next command. It it's an EOF pop the file handle from any nested Reads. If the stack is empty terminate all modules and exit.
3. If a command line has been found, break off the command part, and pass it to the dispatcher. If it succeeds, goto 2.
4. If it's a Read command, push the current handle, open the new file, and go to 2.
5. Else, it's an invalid command. Purge the file stack and go to 2.

As mentioned above, you may be able to implement READ via recursion.

CLI_StringPar. This routine will be called by all the modules to retrieve their parameters. It scans for the comma or whitespace and returns the string containing the next command to the caller. If there is no command it returns a failure indicator.

3.2.5 Errors

Spend some thought and effort doing error checking, The CLI is an interactive program and it will be very useful to get meaningful error messages. The CLI should NOT die on incorrect input (e.g., non-numeric characters in a numeric parameter); it should print out the offending line, pointing out the problem, and ignore the command. Generate test data to exercise your program both with and without syntax errors.

3.2.6 Additional Note

You'll probably need a `CLI_FloatPar` command, that converts a string parameter to a float. You can encapsulate this in a module in the following way. The `CLI_FloatPar` command has the same calling sequence as `CLI_StringPar` but will return a float (or a failure indicator if no parameter is given). The routine will just call `CLI_StringPar` and do an `atof` on it. Some commented example scripts that you should be able to process in your first project using the *Read* command will be distributed on Blackboard.

4 What to Turn In

Turn in the source of your program, along with instructions (or a Makefile) to compile it. Also, turn in a few scripts for the CLI which exercise it and the dummy commands.