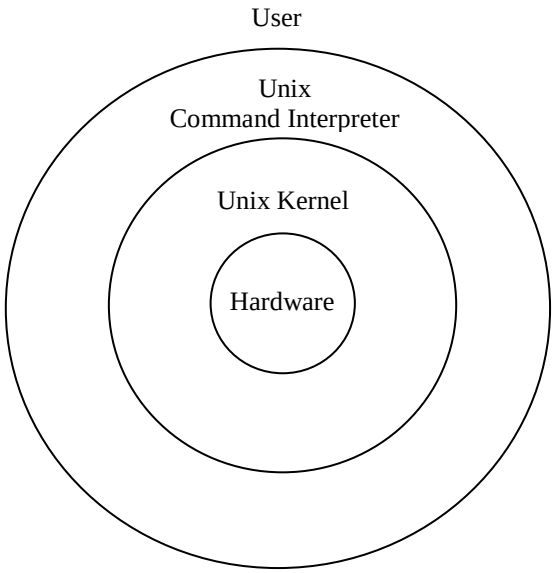
CMPS 11
Intermediate Programming
Lab Assignment 3

In this lab assignment you will create a shell script to compile and run your Lawn.java program, redirecting its input and output from and to files. Recall from lab2 that a shell script is a text file containing instructions to be executed by a Unix command interpreter. If necessary re-read the sections of the lab2 project description pertaining to shell scripts and input-output redirection.

Why is a Unix command interpreter called a shell? A computer is sometimes thought of as consisting of a series of concentric circles with the hardware in the center and the outer layers representing software interfaces.



The computer hardware consists of the CPU, memory, and I/O devices. The kernel serves as a bridge between software applications and the actual data processing that takes place at the hardware level. In the Unix operating system the main application is the command interpreter which acts as an interface between the kernel and user. This outermost layer or "shell" carries out user commands typed at the prompt `%`. There are many versions of the Unix shell. Some of the most common are

sh	Bourne Shell
csh	C Shell
tcsh	Tenex C Shell
ksh	Korn Shell
bash	Bourne Again Shell
zsh	Z Shell

Each shell has slightly different commands and features. The Unix commands you are already familiar with work in all shells, so the differences are only in the advanced commands. Go to

http://www.faqs.org/faqs/unix-faq/shell/shell-differences/

to see the pros/cons and histories of various shells.

To see which shell you are running, do the Unix command `ps -p $$`. Most likely it will be csh, the C Shell. To run a different shell just type its name. For example do `ksh` then `tcsh`. Notice that each shell gives you a slightly different prompt. Type `exit` twice to get back to your original shell.

Each shell is also a complete programming language with conditional and iterative control structures, variables, I/O commands, etc. A shell script is just a program in one of these languages. Typically one specifies the particular shell to be used on the first line of the script as follows.

```
#! /bin/bash
```

We will be using the bash shell in this assignment so all examples will begin with the above line. The character sequence `#!` is known in unix circles as "shebang". Subsequent lines that begin with the `#` character are comments and are ignored by the shell. Traditionally the first program you write in any language just prints the message "Hello, World!" to standard output. Here is the hello world program as a bash script.

```
#! /bin/bash
#--------------------------------------
#  example1
#--------------------------------------
echo "Hello, World!"
```

Create a subdirectory of `cs11` called `lab3`. Create a text file in that directory called `example1` containing the above lines. Make it executable and run it by doing

```
% chmod 700 example1
% example1
```

Notice that just like any Java program, we start our script with a comment block. A shell script can contain any commands we might run at the command line. Here's another example.

```
#! /bin/bash
#--------------------------------------
#  example2
#--------------------------------------
NAME="your_name"
echo -n "Hello $NAME. It is now"
date +"%l:%M %p %Z, %A %B %d, %Y."
```

Replace `your_name` in this script with your own name then run it. There's a lot going on here so lets take it one line at a time.

```
NAME="your_name"
```

assigns the variable `NAME` to be the string `"your_name"`. Unlike in Java, spaces matter in shell scripts. In particular

```
NAME = "your_name"
```

will not work (try it and see the error message you get). Thus to define a variable we just do `VAR=VALUE`. To refer to the value stored in a variable however, we must place a `$` before its name. This helps to explain the next line.

```
echo -n "Hello $NAME. It is now"
```

If you do `man echo` you will see that the `-n` option omits the newline character at the end of the printed line. The `date` command prints out the date and time in a standard format. That format can be altered as desired. Try `date` by itself, then do `man date` to see all the formatting options. The line

```
date +"%l:%M %p %Z, %A %B %d, %Y."
```

prints the date in a slightly more friendly format, as you can see when you run the script. Now consider another example.

```
#! /bin/bash
#--------------------------------------
#  example3
#--------------------------------------
ls -l
example1 > junk-out
example2 >> junk-out
javac example1 >& comp-errs
```

Try to predict what the effect of this script will be. If necessary type the commands individually and observe their effects. The first line `ls -l` is the only line whose output goes to stdout. Recall from lab2 the meanings of the redirect operators `>`, `>>` and `>&`. The next line `example1 > junk-out` runs example1 and sends its output to a new file called `junk-out`. The line `example2 >> junk-out` appends the output of `example2` to the same file. The next line runs the javac compiler on the file `example1`. Since `example1` is not a Java source file, one expects to get only error messages from the compiler. In fact the errors you get from javac do not go to stdout, but to stderr. The line `javac example1 >& comp-errs` places those errors in the file `comp-errs`. Run the above script and observe these effects.

A short introduction to writing bash scripts can be found at

http://www.panix.com/~elflord/unix/bash-tute.html

You can see more advanced examples at

http://www.hlevkin.com/Shell_progr/hellobash.htm

Chapter 13 of the recommended text *Your Unix* by Sumitabha Das (mentioned in our course syllabus) contains a very thorough introduction to programming with sh, the Bourne Shell.

What to turn in

Copy your program Lawn.java from pa1 into the lab3 directory. Write a bash script called `RunLawn` that performs the following operations.

1. Print the line `compiling Lawn.java` to stdout.
2. Compile the program `Lawn.java`, sending any errors or warnings to the file `Lawn-errs`.
3. Print the line `running Lawn.class` to stdout.
4. Run the class file `Lawn.class`, reading input from a file called `Lawn-in` and sending output to a file called `Lawn-out`.
5. Print the line `deleting the file Lawn.class` to stdout.
6. Remove the file `Lawn.class` from the current directory.

Begin by asking yourself how you would perform each of the above steps at the Unix command line. Obviously item (2) requires the javac compiler, and item (4) requires the java command, which invokes the Java Virtual Machine (JVM). To test your script prepare an input file called `Lawn-in` containing the five numbers you would type in response to the user prompts in `Lawn.java`, which ask for lengths, widths, and mowing rate. Run your script and see if the expected output is stored in `Lawn-out`. Notice that the user prompts are intermixed with program output making the file `Lawn-out` a little hard to read. Edit your source file `Lawn.java` so as to eliminate those user prompts. A transcript of one possible test run of your script would appear as follows.

```
% more Lawn-in
100 200
50 75
5
% RunLawn
compiling Lawn.java
running Lawn.class
deleting Lawn.class
% more Lawn-errs
% more Lawn-out
The lawn area is 16250.0 square feet.
The mowing time is 0 hours 54 minutes 10 seconds.
%
```

Note that `Lawn-errs` is empty since there were no syntax errors. Delete a semi-colon in `Lawn.java` to introduce a syntax error and run your script again. Check that the error messages are directed to the file `Lawn-errs`. Correct the syntax error before you turn the project in.

Submit the files `RunLawn` and your edited version of `Lawn.java` to the assignment name lab3. This project is somewhat shorter than lab2, but still requires some time to get it right, so start early and ask questions in lab sessions, office hours and on Piazza.