SI 413 Fall 2013: Scheme Practicum Exam

This is a **in-class practicum exam** that will count towards your 6-week exam grade. You should save your code *in plain text* in a file called `exam.scm`, put it in a folder on the CS Linux environment, and run `413sub exam 01` to submit.

There are 45 points possible on this exam, spread over 6 problems. All problems are required; you should try to get as many points as you can by writing correct functions.

Read, understand, and follow the following guidelines for this exam. If there are any ambiguities or questions, it is *your responsibility* to resolve them and make sure you understand what is expected.

- **You may** consult the course textbooks, anything posted on or linked directly from the course website, any homework or lab solutions, any graded work you have been handed back, and any notes you have taken in class.
- **You may NOT** consult with any other human, including your classmates and anyone else except the instructor.
- **You may NOT** consult other internet sources not directly reachable from the course website
- **You may NOT** perform any internet searches or post questions or queries online.
- **You may NOT** discuss anything about this class or this exam with anyone in the other section that hasn't taken it yet.
- **You may NOT** use your cell phone.
- **You MUST** contact Dr. Roche if anything is unclear, or if there are any glitches in the submission process.
- **You MUST** follow the same instructions as our Scheme labs. Style counts, although the level of documentation expected in the exam is less than what would be expected for a lab when you have more time. Usually, choosing meaningful names for helper functions and variables will be sufficient.

To use the built-in `printf` function, be sure to include the line

```
(#%require (only scheme/base printf))
```

Also, here is our version of the `filter` function. Feel free to copy into your code:

```
(define (filter pred? L)
  (cond ((null? L) '())
        ((pred? (car L))
         (cons (car L) (filter pred? (cdr L))))
        (else (filter pred? (cdr L)))))
```

Problem 1   5 points

Define a function `(root-info L)` that takes a list of the form `(a b c)`, where `a`, `b` and `c` are numbers, and returns the number of distinct real roots of the polynomial ax^2+bx+c. Recall that b^2-4ac, the *discriminant* of the polynomial, determines the number of real roots. If the discriminant is positive there are two real roots, if it is zero there is one real root, and if it is negative then there are zero real roots.

For full credit, you should use a `let` expression.

Examples:

```
> (root-info '(2 1 2))
0
> (root-info '(2 5 2))
2
> (root-info '(1 -2 1))
1
```

Problem 2   5 points

Write a function `(mostly-positive? x1 x2 x3 ...)` that takes a *variable number of arguments* $x_1, x_2, \ldots, x_n$ and returns true or false depending on whether at least half of the arguments are positive numbers (meaning strictly greater than zero).

For full credit, your function should use things like `lambda`, `filter`, `map`, and `apply` so that there is no explicit recursion.

(See the listing of `filter` on the cover page.)

Example:

```
> (mostly-positive? 4 -2 3)
#t
> (mostly-positive? -1 -2 -20)
#f
> (mostly-positive? -1 1 -1 0 0)
#f
```

Problem 3   10 points

Write a function `(longest-run L)` that takes a list of numbers `L` and computes the length of the longest "run" of equal numbers in a row.

(Hint: Consider writing a helper function `(run-start L)` that computes the length of the longest run at the start of the list.)

Examples:

```
> (longest-run '(10 20 20 20 20 30 40 40))
4
> (longest-run '(10 20 30 40 30 20 10))
1
> (longest-run '(10 10 10 10 10))
5
```

Problem 4   10 points

Computer logins on a certain server are stored as a list of 2-element lists, each of which contains a symbol for the username and a string for the time. They are stored in order, with the earliest login first.

Write a function `(last-login user L)` that takes a symbol for the username and returns the string corresponding to the last entry in `L` that matches that username. If the username is never found, your function should return `#f`

(Hint: I think the *easiest* way to write this function is by making it tail-recursive.)

Examples:

```
> (last-login 'r2d2 '((r2d2 "morning") (c3po "noon") (r2d2 "evening")))
"evening"
> (last-login 'c3po '((r2d2 "morning") (c3po "noon") (r2d2 "evening")))
"noon"
> (last-login 'chewy '((r2d2 "morning") (c3po "noon") (r2d2 "evening")))
#f
```

Problem 5   10 points

A certain computer system must be rebooted every 24 hours, for reasons that will not be specified. The way it works is, every time someone logs on, the current time is compared with the time of the last reboot. If it's been more than 24 hours since the last reboot, the system is restarted; otherwise nothing at all happens.

Write a function `(make-rebooter start)` that returns a new function (`lambda` expression) to represent a rebooting login server. The returned function should take a single argument which will be a number, representing the current login time in seconds since some prior fixed event. Your new function should do nothing if no reboot is necessary, and otherwise it should return the symbol `'reboot` and update the internal time.

Note: the time is always updated from the last reboot, and only one reboot happens at a time.

(In case you've forgotten, there are 60 minutes in an hour, and 60 seconds in a minute. 24 hours before a required reboot.)

Examples:

```
> (define comp (make-rebooter 100000))
> (comp 101000)
> (comp 102000)
> (comp 160000)
> (comp 180000)
> (comp 200000)
reboot
> (comp 300000)
reboot
> (comp 300001)
> (comp 380000)
> (comp 388000)
reboot
```

Problem 6   5 points

Someone has been sneaky trying to avoid reboots, by winding back the clock by a few hours every time they login. Make another function `(better-rebooter start)` that works just like the object above, with the following addition: If the most recent time is strictly less than the last login time, two things happen: a message is displayed saying

```
WARNING: Time went backwards.
```

And then the machine is rebooted as normal, meaning the symbol `'reboot` is returned and the last reboot time is updated accordingly. That is, if time goes backwards, you *always* force a reboot.

Examples:

```
> (define R2 (better-rebooter 0))
> (R2 1000)
> (R2 1000)
> (R2 100000)
reboot
> (R2 1000)
WARNING: Time went backwards.
reboot
> (R2 100000)
reboot
> (R2 500000)
reboot
> (R2 505555)
> (R2 500001)
WARNING: Time went backwards.
reboot
```