

CS51 Assignment 9: Mutability and OOP

Due: Tuesday, April 28th, 2009, 11:59PM

Total Points: 48 (including 5 style points)

(Note: There will be an optional contest to use CS51 techniques or algorithms to predict the outcomes of Red Sox games. The details will be finalized this weekend. Check <http://cs51.seas.harvard.edu/redsox.php> for updates.)

1 Mutable Data

(18 points total)

1.1 Lists

Exercise 1.

12 points

We've defined the mutable lists `foo` and `bar` as follows:

```
(define foo (mcons 'a (mcons 'b (mcons 'c empty))))  
(define bar (mcdr foo))
```

Fill in the values of `foo` and `bar` after applying the following functions. Assume that before each function is run, the variables are reset to the above values. NOTE: while you can obviously test your answers in DrScheme, you should first try to figure them out yourself—you won't have DrScheme available on the final.

(a) [2 points]

```
> (define (f1 lst) (set-mcar! (mcdr lst) 'x))  
> (f1 foo)
```

(b) [2 points]

```
> (f1 bar)
```

(c) [2 points]

```
> (define (f2) (set-mcar! (mcdr foo) 'x))  
> (f2)
```

(d) [2 points]

```
> (define (f3 lst)
  (let ([copy lst])
    (set-mcar! (mcdr copy) 'x)))
> (f3 bar)
```

(e) [2 points]

```
> (define (f4)
  (let ([copy foo])
    (set-mcar! (mcdr copy) 'x)))
> (f4)
```

(f) [2 points]

```
>(define (f5)
  (let ([copy foo])
    (set! copy 'x)))
> (f5)
```

1.2 Game, set!, Match

We've defined the variables `goo` and `gar` as follows:

```
> (define goo 'a)
> (define gar goo)
```

Exercise 2.

6 points

Fill in the values of `goo` and `gar` after applying the following functions. Assume that before each function is run, the variables are reset to the above values. NOTE: Same as above—try to do these without running them.

(a) [2 points]

```
> (define (g1 var) (set! var 'x))
> (g1 goo)
```

(b) [2 points]

```
> (define (g2) (set! goo 'x))
> (g2)
```

(c) [2 points]

```
> (define (g3)
  (let ([copy goo])
    (set! copy 'x)))
> (g3)
```

2 John Scheme, again?

(20 points total)

You have just joined the Prime Shipping Company, which ships integers in a one-dimensional world. Prime Shipping began with very simple shipping model – load everything onto a `stack-truck%`, which, as you might expect, operated like a last-in-first-out stack. It makes stops around the one-dimensional world, and at each stop, unloads a number of items (from the top of the stack) and picks up items (pushing them on top of the stack). Before long, Prime Shipping wanted other options for shipping, and so their programmer, the now-infamous John Scheme, wrote `queue-ship%` and `pqueue-plane%` implementations (the priority queue will remove the minimum element each time). There's one problem: John made three separate classes, and now Prime Shipping wants to have `Queue-Trucks`, `Stack-Planes`, and the like. John suggests that you simply write the classes as they are needed, but you want to make it easier to add more combinations in the future – if you write a new data structure for storage, you want to be able to use it with all existing modes of transportation.

2.1 Your Task

Exercise 3.

20 points

(a) [10 points] We've given you John Scheme's implementation of `stack-truck%`, `queue-ship%`, and `pqueue-plane%`. Each class has a different type of hard-coded storage container, a different capacity, a different speed, and makes a different sound as it travels. Describe a class hierarchy that allows you to arbitrarily combine modes of transportation with storage containers. Ultimately, you should try to minimize the amount of replicated code. List all the classes you'll have, what classes they will inherit from, and what data they will store internally. Be sure to look over the existing code so you know exactly what functionality you'll need to have! The only requirement is that you must be able to construct objects that respond to the `get-cargo-size` and `load-and-unload` messages and behave appropriately for a chosen mode of transportation and storage container.

(b) [10 points] Implement the classes you described above. Test your version by passing your objects to the `run-route` function and making sure they behave equivalently to John Scheme's versions. Feel free to reuse parts of the code from the existing versions, but you should replace the parts that you think are poorly written or designed.

3 Nowhere to go but down

(4 points total)

Exercise 4.

4 points

(a) [4 points] Least squares is a good error function for hillclimbing due to a few key properties. Describe two functions that are *bad* for hillclimbing (i.e. where hillclimbing is unlikely to find the correct minimum). You do not need to provide formulas; a simple description of the graph will suffice. (Think about rolling downhill).

4 Finishing Up

(1 point total)

Exercise 5.

1 point

(a) [1 point]

Please tell us how much time you spent on this problem set. You may use this space to add any thoughts or questions you want to send to your TF as well, if you so desire.