

# CS51 Assignment 7: Object-Oriented Programming

Due: Tue, April 14th, 2009, 11:59PM

*Total Points: 44 (including 5 style points)*

## 1 Mutable Lists

*(7 points total)*

### Exercise 1.

*7 points*

In lecture, you learned about creating mutable lists using `mcons`. A mutable list is *not* a regular scheme list, and therefore cannot be passed to functions expecting a regular list. These functions must be re-written for mutable lists.

As you may have noticed, a call to `set!` doesn't seem to evaluate to anything, since nothing is printed by interpreter.

```
> (define foo 1)
> (set! foo 2)
> ; Nothing was printed!
```

However, `set!` *does* actually evaluate to a value, `<#void>`, which the interpreter simply chooses not to print. You can make functions you write return `<#void>` by calling the `void` function. For example,

```
> (define foo 1)
> (void)
> (equal? (void) (set! foo 2))
#t
```

(a) [5 points] Write `map!`, which performs a map operation on a mutable list. Calling `(map! f lst)` will modify the contents of `lst` such that each element `n` is replaced with `f(n)`. `map!` should return `<#void>`

```
> (define my-list (mcons 1 (mcons 2 (mcons 3 empty))))
> my-list
{ 1 2 3 }
> (map! (lambda (x) (* x 2)) my-list)
> my-list
{ 2 4 6 }
```

(b) [2 points] Write at least 3 tests for `map!`. We've provided an example of one way to write a test for a mutating function, but you do need to follow this example exactly.

## 2 Encapsulation using structs

(15 points total)

### Exercise 2.

15 points

You've seen how to encapsulate data and functions in a struct. Mutable data allows you to update local variables within a struct, which can make this encapsulation more natural.

In this assignment, we are asking you to write a `btree-set`. A `btree-set` is struct that allows you to add numbers and check for membership. This is accomplished with the functions `btree-set-insert` and `btree-set-search`, respectively.

Internally, a `btree-set` is implemented using a binary tree; we've provided you with binary tree operations.

(a) [12 points] Fill in the missing code in `new-btree-set`. The function should return a `btree-set` struct with the `insert` and `search` fields assigned to the appropriate insert and search functions.

(b) [3 points] Write at least 3 tests for `btree-set`. To assist you, we've provided the function `build-set`, which turns a list of numbers into a `btree-set`.

## 3 Classes

(16 points total)

### Exercise 3.

16 points

(a) [8 points] Now, rewrite the encapsulated binary-tree-set above as a class, `set%`. It should be *very* similar to the code you wrote above.

(b) [4 points] A useful benefit of defining our set as a class is that it is very easy to extend its functionality. Write a subclass of `set%`, `filtered-set%`, whose initializer takes a predicate, that only allows numbers that satisfy the predicate to be inserted. (When given a number to insert that does not satisfy the predicate, your implementation should just silently leave the set unchanged.)

(c) [4 points] Write a subclass of `set%`, `sized-set%`, that provides a way to get the number of elements in the set. (Your implementation of `sized-set%` should have a method, `get-size%`, that runs in constant time; insertion and search should still work in time linear in the depth of the tree. You need not *prove* these properties, however.)

## 4 Finishing Up

*(1 point total)*

### Exercise 4.

*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.