MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall 2007

Recitation 7 — 9/26/2007 Solutions
List Manipulations

List Functions

```
(define (length lst)
   (if (null? lst)
       0
       (+ 1 (length (cdr lst)))))

(define (map proc lst)
   (if (null? lst)
       '()
       (cons (proc (car lst))
             (map proc (cdr lst)))))

(define (filter pred lst)
   (if (null? lst)
       '()
       (if (pred (car lst))
           (cons (car lst) (filter pred (cdr lst)))
           (filter pred (cdr lst)))))

;also known as accumulate, foldr
(define (fold-right op init lst)
   (if (null? lst)
       init
       (op (car lst)
           (fold-right op init (cdr lst)))))

(define (list-ref lst n)
  (if (= n 0)
      (car lst)
      (list-ref (cdr lst) (- n 1))))

(define (append lst1 lst2)
  (if (null? lst1)
      lst2
      (cons (car lst1)
            (append (cdr lst1) lst2))))
```

Problems

1. Write a function `occurrences` that takes a number and a list and counts the number of times the number appears in the list. Write two versions – one that uses `filter`, and one that uses `fold-right`. For example,

```
(occurrences 1 (list 1 2 1 1 3))  ==>  3

(define (occurrences elm lst)
   (length (filter (lambda (x) (= x elm)) lst)))

(define (occurrences elm lst)
  (fold-right
   (lambda (a b)
           (+ (if (= a elm) 1 0)
              b))
   0
   lst))
```

2. Define `length` using a higher order list procedure.

```
(define (length lst)
  (fold-right
   (lambda (a b) (+ b 1))
   0
   lst))
```

3. Define `ls` to be a list of *procedures*:

```
(define (square x) (* x x))
(define (double x) (* x 2))
(define (inc x) (+ x 1))
(define ls (list square double inc))
```

   Now say we want a function `apply-procs` that behaves as follows:

```
(apply-procs ls 4)
=> ((square 4) (double 4) (inc 4)) = (16 8 5)
(apply-procs ls 3)
=> ((square 3) (double 3) (inc 3)) = (9 6 4)
```

   Write a definition for `apply-procs` using map.

```
(define (apply-procs proclst val)
  (map (lambda (p) (p val)) proclst))
```

4. Suppose `x` is bound to the list `(1 2 3 4 5 6 7)`. Using map, filter, and/or fold-right, write an expression involving `x` that returns:

   (a) `(1 4 9 16 25 36 49)`
   ```
   (map square x)
   ```

   (b) `(1 3 5 7)`
   ```
   (filter odd? x)
   ```

   (c) `((1 1) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7))`
   ```
   (map (lambda (x) (list x x)) x)
   ```

   (d) `((2) ((4) ((6) ())))`
   ```
   (fold-right (lambda (a b) (list (list a) b))
   '() (filter even? x))
   ```

   (e) The maximum element of `x`: 7
   ```
   (fold-right (lambda (a b) (if (> a b) a b)) 0 x)
   ```

   (f) list of last element of `x`: (7)
   ```
   (fold-right (lambda (e r) (if (null? r) (list e) r)) '() x))
   ```

   (g) The list in reverse order: `(7 6 5 4 3 2 1)`
   ```
   (fold-right (lambda (a b) (append b (list a))) '() x)
   ```

   (h) Bonus: reverse a list in less than $\Theta(n^2)$ time
   ```
   (fold-left (lambda (a b) (cons b a)) '() x)
   ```