Higher order functions

The higher order procedures provided by scheme can be be found in Page 313 of your text book. You may have to set your Scheme language to “Intermediate Student with Lambda”.

Note the definition of foldr and foldl. They consume a function of the form $X \ Y \rightarrow \ Y$ and a list-of-$X$ and produce a list-of-$Y$. The simpler examples, have $X$ and $Y$ as the same.

1. (4 pts) Implement andmap and ormap using the other higher order functions (and no recursion).

2. (4 pts) Write a scheme function concat which consumes two lists and produces one list whose elements contain those of the first list followed by those of the second list. Your implementation should not use recursion or append.

3. (3 pts) Write a scheme function flatmap which consumes a function of the form $X \rightarrow$ list-of-$Y$ and a list-of-$X$, and produces a list-of-$Y$ obtained by joining together all the lists produced the function when applied on each member of the list. Do not use recursion.

```
;; flatmap: (X -> list-of-Y) list-of-X -> list-of-Y
;; to construct a list obtained by combining all the
;; lists obtained by applying the function to each member
;; of the list.
(define (flatmap fn lox)
  ...)
```

4. (4 pts) Recall the shopping cart problem of the midterm. Consider the following scheme fragment

```
;; item is a symbol, price is a number
(define-struct item-price (item price))

;; item = symbol, qty = integer > 0
(define-struct item-qty (item qty))
```
(define (cost iq ip)
  (cond
   [(symbol=? (item-price-item ip) (item-qty-item iq))
    (* (item-price-price ip) (item-qty-qty iq))]
   [else 0]
  )
)

(define (item-cost iq loip)
...
)

(define (base-cost loiq loip)
...
)

(define (base-cost loiq loip)
  (* 1.0875 (base-cost loiq loip)))

The function cost consumes an item-qty and an item-price and produces the cost of the item (taking the quantity into account) if the item's match or 0 if they do not. Complete the definition of item-cost and base-cost, using only higher order functions, and local or lambda constructs.

5. (5 pts) Write a scheme function data which takes a data structure constructed using numbers and lists (could be arbitrarily deeply nested), and produces a simple list containing all the numbers found in this list.

   > (data 3)
   (list 3)
   > (data (list 4 5 (list 4 6 7 (list 78 34) (list 5)) (list )))
   (list 4 5 4 6 7 78 34 5)

The only type of data your function needs to handle is lists and numbers. Since they could be arbitrarily nested one cannot give a simple description of the type of data your function consumes. However, you function always produces a list of numbers.