CMSC 10500-1: Homework 7

(due on Friday July 16th)

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 -> 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).

   Here are a few possible solutions..

```
(define (andmap1 pred lox)
  (foldr (lambda (x y) (and x y)) true (map pred lox)))

(define (andmap2 pred lox)
  (= (length lox) (length (filter pred lox))))

(define (andmap3 pred lox)
  (empty? (filter (lambda (x) (not (pred x))) lox)))

(define (ormap1 pred lox)
  (foldr (lambda (x y) (or x y)) false (map pred lox)))

(define (ormap2 pred lox)
  (not (empty? (filter pred lox))))

(define (ormap3 pred lox)
  (not (andmap2 (lambda (x) (not (pred x))) lox)))
```

   The `andmap2` in `ormap3` can be replaced with any version of `andmap`. Similarly one can define an `andmap4` in terms of any of `ormap` implementations (except that defined in terms of andmap).

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`.

   The key observation is that `(cons num lst)` appends the lst to the list containing one element namely `(list num)`. In order to concatenate `(list p q r)` with `lst2`, we need `(cons p (cons q (cons r lst2)))`, i.e. we need to fold using `cons` from the right using `lst2` as the base. This is an example where the base value plays an important role.

```
(define (concat lst1 lst2)
    (foldr cons lst2 lst1))
```

3. **(3 pts)** Write a scheme function `flatmap` which consumes a function of the form `X -> 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.

   This is simple. Just apply map to get a list of lists, and concatenate all them together by folding using append.

```
;; 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)
    (foldr append empty (map 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))


;; cost: item-qty item-price
;;   cost of the item if the items match, else 0
(define (cost iq ip)
  (cond
    [(symbol=? (item-price-item ip) (item-qty-item iq))
```

```
          (* (item-price-price ip) (item-qty-qty iq))]
        [else 0]
      )
    )

    ;; item-cost : item-qty list-of-item-price -> number
    ;;    returns the price of the item
    (define (item-cost iq loip)
        ...)

    ;; base-cost: list-of-item-qty list-of-item-price -> number
    ;; computes the cost of all items without tax
    (define (base-cost loiq loip)
       ...)

    ;; total-cost: list-of-item-qty list-of-item-price -> number
    ;; computes the cost of all items with tax
    (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.

The function `cost` returns the cost of the item if the items are the same, else returns zero. Thus the cost of an item, is just the sum of the cost function, for each item in the price list. This gives...

```
    ;; item-cost : item-qty list-of-item-price -> number
    ;;    returns the price of the item
    (define (item-cost iq loip)
        (foldr + 0 (map (lambda (ip) (cost iq ip)) loip)))
```

Similarly the `base-cost` is just item-cost summed over the shopping cart.

```
    ;; base-cost : list-of-item-qty list-of-item-price -> number
    ;;    returns the total price of the cart w/o tax
    (define (item-cost loiq loip)
        (foldr + 0 (map (lambda (iq) (item-cost iq loip)) loiq)))
```

If we combine everything the total-cost function can be written as:

```
(define (total-cost loiq loip)
    (* 1.0875
       (foldr +
              0
              (map (lambda (iq)
                   (foldr + 0
                          (map (lambda (ip) (cost iq ip))))
                   )
              )
       )
    )
)
```

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.

Since the list can be arbitrarily deep, we must use recursion. However since the length of the list is arbitrary, the number of times we need to call ourselves recursively is also not known in advance. Also it is enough to call ourselves on sublists (not numbers in the lists). If we want to use a higher order function like `map` it will call the function on every element. So our function should be able to handle simple numbers apart from lists.

Once we have this, consider the second example. The input list contains a list of 4 items, viz. `4`, `5`, `(list 4 6 7 (list 78 34) (list 5))` and `(list )`. Assuming we already have the correct answer to these inputs by a recursive call, we have `(list 4)`, `(list 5)`, `(list 4 6 7 78 34 5)` and `(list )`. It only remains to see that the solution for the main problem, is just the concatenation of all these lists. Thus we get the following solution

```
;; data: deeplist-of-numbers -> list-of-numbers
;;   return a list of number found in this deeplist.
(define (data lst)
        (cond
          [(number? lst) (list lst)]
          [(empty? lst) empty]
          [else (foldr append empty (map data lst))]
        )
    )
```

Note that the else clause can be replaced with `(flatmap data lst)`, where `flatmap` is the function of question 3.