1 Understanding Loops

Last semester we had the following problem:

```
;; add all numbers in the following list:

(define JANUS
  (list #i31
        #i2e+34
        #i-1.2345678901235e+80
        #i2749
        #i-2939234
        #i-2e+33
        #i3.2e+270
        #i17
        #i-2.4e+270
        #i4.2344294738446e+170
        #i1
        #i-8e+269
        #i0
        #i99))
```

We produced two solutions for this problem:

```
(define (sum-right alist)
  (foldr + 0 alist))

(define (sum-left alist)
  (foldl + 0 alist))
```

but unfortunately, these did not produce the same result:

```
> (sum-left JANUS)
#i99.0
> (sum-right JANUS)
#i-1.2345678901235e+80
```

Do you remember why?
Maybe seeing the definition of `foldl` and `foldr` will help:

```
; foldr : (X Y  ->  Y) Y (listof X)  ->  Y
; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base alox) ...)

; foldl : (X Y  ->  Y) Y (listof X)  ->  Y
; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
(define (foldl f base alox) ...)
```

The numbers in the list `(list 3 5 8 2)` can be added in one of the following ways:

```
(+ 3 (+ 5 (+ 8 (+ 2 0))))

(+ 2 (+ 8 (+ 5 (+ 3 0))))
```

1. Design the function `sum-reg` using the standard *Design Recipe* and explain in what order are the numbers added.

2. Now convert the function to one that uses an accumulator `sum-acc`. What is the order of the additions now?

3. Now match the two functions `sum-reg` and `sum-acc` with the functions `sum-left` and `sum-right` defined earlier.

1.1 Designing Programs with Accumulators

We recognize the need for accumulator when the intermediate computation within the recursive function we are trying to design requires that we remember some information encountered earlier in the computation. The examples of computing the sum or a product of all numbers qualifies only if we specify the order in which the operation should be performed.

Next we think of the meaning of the accumulator. The following two hints may help. First, the accumulator value has the same type as the expected result. Next we determine what should the function compute when there is only one piece of information that we need to remember. This value becomes the value of the first accumulator, and is the value produced when the problem is small enough so that the recursive function invocation never happens. We call this the *base value*.

At this point we should write a comment that explains the meaning of the accumulator. Additionally, we should specify the invariant that the accumulator needs to satisfy. (For explanation of how to specify the invariant, please read the relevant pages in the HtDP text.)

The template for the whole function then becomes:

```
;; produce a value of the type Y from the given list of X
;; rec-fcn: [Listof X] -> Y
(define (rec-fcn lox)
  (rec-fcn-acc lox base-acc-value))

;; recur with updated accumulator,
;; unless the end of list is reached
;; rec-fcn-acc: [Listof X] Y -> Y
(define (rec-fcn-acc lox acc)
  (cond
    ;; at the end produce the accumulated value
    [(empty? lox) acc]
```

```
    ;; otherwise invoke rec-fcn-acc with updated accumulator
    ;; and the rest of the list
    [(cons? lox)  (rec-fcn-acc (rest lox)
                               (update (first lox) acc))]))
```

Identify the parts of this template in your solution to the addition problem. What is the contract for the `update` function? Can we add the `update` function as an argument to the `rec-fcn-acc` function? Try it and compare it with the definition of `foldl` and `foldr`.

1.2 Exploring the commutativity and associativity of other computations.

Next we design two functions that compute the factorial of a given number. We recall that we can define factorial of 5 as one of the following two values:

```
5! = 1 . 2 . 3 . 4 . 5
5! = 5 . 4 . 3 . 2 . 1
```

Design the functions `fac-L->R` and `fac-R->L` that compute a factorial of the given number.

For help consult HtDP, exercise 31.3.2 and the text before this exercise. Run the exercise 31.3.2 and verify the book's assertions about the times needed to compute the two results.

1.3 Homework partners

We stop now to set up the homework partner teams.

1.4 The need for accumulators

In the lectures we had the following problem. We were computing the discount price of a list of books the customer wants to buy.

Here is the program we produced:

```
;; Lecture 1
;; CS U213 Fall 2008
;; bookstore4.ss

;; A Book is (make-book String Author Num Symbol)
;; There are three kinds of books:
;; fiction, nonfiction, textbook
```

```
;; represented by symbols 'F 'N 'T
(define-struct book (title author price kind))

;; An Author is (make-author String Num)
(define-struct author (name yob))

;; Examples of authors
(define eh (make-author "Hemingway" 1900))
(define ebw (make-author "White" 1920))
(define mf (make-author "MF" 1970))

;; Examples of books
(define oms (make-book "Old Man and the Sea" eh 10 'F))
(define eos (make-book "Elements of Style" ebw 20 'N))
(define htdp (make-book "HtDP" mf 60 'T))

;; the sale price of the book depends on the daily discounts
;; these may differ depending on the kind of book
;; suppose today we have the folloiwng discounts:
;; there is 30% discount on fiction books
;; there is 20% discount on nonfiction books
;; textbooks sell at full price


;; tests for the function book-sale-price
(check-expect (book-sale-price oms) 7)
(check-expect (book-sale-price eos) 16)
(check-expect (book-sale-price htdp) 60)

;; compute the sale prics of the given book
;; based on today's discounts
;; book-sale-price: Book -> Number
(define (book-sale-price abook)
;  ... (book-title abook) ...  String
;  ... (book-author abook) ... Author
;  ... (book-price abook) ...  Num
;  ... (book-kind abook) ...   Symbol
  (cond
    [(symbol=? (book-kind abook) 'F)
     (- (book-price abook) (* 0.3 (book-price abook)))]
    [(symbol=? (book-kind abook) 'N)
     (- (book-price abook) (* 0.2 (book-price abook)))]
    [(symbol=? (book-kind abook) 'T)
     (book-price abook)]))
```

```
;; tests for the function before1950?
(check-expect (before1950? oms) true)
(check-expect (before1950? eos) true)
(check-expect (before1950? htdp) false)

;; was the author of the book born before 1950?
;; before1950?: Book -> Boolean
(define (before1950? abook)
;  ... (book-title abook) ...  String
;  ... (book-author abook) ... Author
;  ... (book-price abook) ...  Num
;  ... (book-kind abook) ...   Symbol

;  ... (author-before1950? (book-author abook)) ...  Boolean

;  ... (author-name (book-author abook)) ... String
;  ... (author-yob  (book-author abook)) ... Num
  (author-before1950? (book-author abook)))

;; tests for the function author-before1950?
(check-expect (author-before1950? eh) true)
(check-expect (author-before1950? ebw) true)
(check-expect (author-before1950? mf) false)

;; was this author born before 1950?
;; author-before1950?: Author -> Boolean
(define (author-before1950? an-author)
;  ... (author-name an-author) ... String
;  ... (author-yob  an-author) ... Num
  (< (author-yob  an-author) 1950))

;; tests for the function all-before1950
(check-expect (all-books-before1950 empty)
              empty)
(check-expect (all-books-before1950 (list oms htdp eos))
              (list oms eos))
(check-expect (all-books-before1950 (list oms eos))
              (list oms eos))
(check-expect (all-books-before1950 (list htdp))
              empty)

;; produce from this list of all books written
;; by authors born before 1950
;; all-books-before1950: [Listof Book] -> [Listof Book]
```

```
(define (all-books-before1950 alob)
  (filter before1950? alob))

;; tests for the function total-sale-price
(check-expect (total-sale-price empty) 0)
(check-expect (total-sale-price (list oms htdp eos)) 83)
(check-expect (total-sale-price (list oms eos)) 23)
(check-expect (total-sale-price (list htdp)) 60)

;; compute the total sale price of all books in a list
;; we can compute make a list of all sale prices
;; and then add all items in the list
;; total-sale-price: [Listof Book] -> Num
(define (total-sale-price alob)
  (foldl + 0 (map book-sale-price alob)))
```

The last function is not very efficient - we first consume the list of books and produce a list of discount prices, then we go over the list of discount prices and compute the total sale price. If we were in a bookstore, a clerk could easily add the newly computed sale price to the *running total*.

1. Define the `total-sale-price-acc` function that uses an accumulator to keep track of the total sale price up till now. Do not use any of the *Scheme* loops. Run the same tests as we have defined for the original version.

2. Design the function `average-sale-price` that computes the average price per book - by following the *Design Recipe* step by step. You may have to define several functions. Do not use any of the *Scheme* loops.

3. Refactor the function `average-sale-price` (designing a new function `average-sale-price-acc` that traverses over the list of books only once. Run the same tests as we have defined for the function `average-sale-price`.

Save all your work — the next lab will build on the work you have done here!