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.

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 given information about a radio show: name, total running time in minutes, and a list of ads to run during the show, where for each ad we were given its name, the running time in minutes and the profit it generates in dollars.

Here is the program we produced:

```
;; Data definitions

;; A Radio Show (RS) is make-rs String Number [Listof Ad]
(define-struct rs (name minutes ads))

;; An Ad is (make-ad String Number Number)
(define-struct ad (name minutes profit))

;; Examples of data:

(define ipod-ad (make-ad "ipod" 2 100))
(define ms-ad (make-ad "ms" 1 500))
(define xbox-ad (make-ad "xbox" 2 300))

(define news-ads (list ipod-ad ms-ad ipod-ad xbox-ad))
(define game-ads (list ipod-ad ms-ad ipod-ad ms-ad xbox-ad ipod-ad))
(define bad-ads (list ipod-ad ms-ad ms-ad ipod-ad xbox-ad ipod-ad))

(define news (make-rs "news" 60 news-ads))
(define game (make-rs "game" 120 game-ads))



;; compute the total time for all ads in the given list
;; total-time: [Listof Ad] -> Number
(define (total-time adlist)
  (cond
    [(empty? adlist) 0]
```

```
    [else (+ (ad-minutes (first adlist))
             (total-time (rest adlist)))]))

(check-expect (total-time news-ads) 7)
(check-expect (total-time game-ads) 10)


;; how much time is there for the show itself
;; show-time: RS -> Number
(define (show-time an-rs)
  (- (rs-minutes an-rs) (total-time (rs-ads an-rs))))

(check-expect (show-time news) 53)
(check-expect (show-time game) 110)
```

1. Convert the *total-time* function to *total-time-acc* that uses the accumulator.

2. Compute the total profit for the show — using both the function we get by following the design recipe and one that uses the accumulator. (If you are running out of time, skip this, do it at home, and go on to the last problem.)

3. The show producer wants to make sure that the list of ads does not repreat the same ad twice without having a different ad in between. So, a list of ads (*list ipod-ad ms-ad ipod-ad xbox-ad*) is OK, but the list of ads (*list ipod-ad ms-ad ms-ad ipod-ad xbox-ad ipod-ad*) is not acceptable, because two *ms* ads are run without a break in between.

   Design the function *no-repeat* that produces *true* if the list of ads is acceptable and produces *false* otherwise.

   Do you need an accumulator here? Why? Can you write the function without one?

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