9 Understanding Loops

9.1 Problem

1. Design the function `sum-reg` that computes the sum of all numbers in a list of numbers 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. Use the following sequence of numbers to add, using each of the two functions you have designed. What do yoou observe? Can you explain the difference? Use the Stepper to observe how the sum is evaluated in each case.

```
;; 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))
```

9.2 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?

9.3 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?