Writing "Accumulator-Style" Functions

(Note: These steps can be followed when you use accumulators to produce a solution that is tail recursive. These steps can often also be used for context-based accumulators, but not always).

Designing accumulator-style functions is not difficult - just follow these steps:

Step 1: Introduce an accumulator as a new parameter

The idea behind accumulator-style functions is that the answer is built up and saved in an "accumulator" as the recursion progresses. This accumulator needs to be passed on from one activation of the function to the next, so it makes sense to pass it as an argument. Since the accumulator is storing the answer to the function, the data type of the accumulator needs to be the same as the data type of the result of the function. Here's an example; if your original function is defined like this:

```
;; sum:  ListOfNumber -> Number
;; consumes a list of numbers and produces the sum of the
numbers in the list
(define (sum alon)
  ...)
```

then a definition for a function that accomplishes the same thing "accumulator-style" looks like this:

```
;; sum-accum:  ListOfNumber Number -> Number
;; consumes a list of numbers and an accumulator and
produces the sum of
;; the numbers in the list, building up the answer in the
accumulator
(define (sum-accum alon sum-so-far)
  ...)
```

Step 2: Return the accumulator in the empty? case

The accumulator is the place where the answer is being built up. So by the time the entire list has been processed, the final result should be sitting in the accumulator.

```
;; sum-accum:  ListOfNumber Number -> Number
;; consumes a list of numbers and an accumulator and
produces the sum of
;; the numbers in the list, building up the answer in the
accumulator
(define (sum-accum alon sum-so-far)
  (cond [(empty? alon)  sum-so-far]
        ...))
```

Step 3: Make a recursive call on the rest of the list, updating the accumulator as needed

In the cons? case, make a recursive call on the rest of the list, as we've done all along. When you make the recursive call, you need to pass along the accumulator, updated as appropriate for the function being defined. If the accumulator is a number, updating often means adding, multiplying, etc. If the accumulator is a list, updating might mean cons'ing an item onto the list. This is the only part of designing accumulator-style functions where you need to do a lot of thinking.

```
;; sum-accum:  ListOfNumber Number -> Number
;; consumes a list of numbers and an accumulator and
produces the sum of
;; the numbers in the list, building up the answer in the
accumulator
(define (sum-accum alon sum-so-far)
  (cond [(empty? alon)  sum-so-far]
        [(cons? alon) (sum-accum (rest alon) (+ (first
alon) sum-so-far))]))
```

Step 4: Write a function that calls the accumulator-style function, initializing the accumulator

It's in this step that the accumulator is initialized.

```
;; sum:  ListOfNumber -> Number
;; consumes a list of number and produces the sum of the
numbers in the list
(define (sum alon)
  (sum-accum alon 0))
```

That's it! Notice that our new version of the function `sum` has exactly the same signature and purpose as the original `sum`. The user of the function doesn't need to be burdened with the knowledge that an accumulator is being used to solve the problem.