Accumulator-Passing Style
One of the major design goals of the Scheme language was to make it efficient. One key aspect of this is that Scheme internally converts all tail-recursions into loops. This needs some explanation.
First, a function is *tail-recursive* if the last thing it does is recurse (and return the result of the recursion). For example, here are two versions of the factorial function:

(define fact1 (lambda (n)
    (cond
      [(= 0 n) 1]
      [else (* n (fact1 (- n 1)))])))

(define fact2
  (letrec ([fact-a (lambda (n acc)
        (cond
          [(= 0 n) acc]
          [else (fact-a (- n 1) (* n acc)))]))]
    (lambda (n) (fact-a n 1))))
(define fact1 (lambda (n)
    (cond
        [(= 0 n) 1]
        [else (* n (fact1 (- n 1)))])))

fact1 is not tail recursive: in the else line of the cond expression we compute (fact1 (- n 1)) and then multiply this result by n.
(define fact2
  (letrec ([fact-a (lambda (n acc)
                  (cond
                    [ (= 0 n) acc]
                    [else (fact-a (- n 1) (* n acc))]]))])
  (lambda (n) (fact-a n 1))))

fact2 is tail recursive. (fact2 n) just returns (fact-a n 1), and if n > 1 fact-a just returns the result of its recursion: (fact-a (- n 1) (* n acc)). For example, (fact2 4) returns

(fact-a 4 1)
= (fact-a 3 4)
= (fact-a 2 12)
= (fact-a 1 24)
= (fact-a 0 24)
= 24
You can see how a tail-recursion could be turned into a loop: we just need variables that represent the functions arguments. These get updated each time around the loop until the base case is reached, and the base-case tells us what to return.
There are two strategies for trying to write tail-recursions. One of these is *Accumulator-passing style*, which adds an extra parameter $acc$ onto the function. We accumulate the answer in this accumulator. Since the natural expression of most functions doesn't include this parameter, we usually write the tail-recursion as a helper function. fact2 illustrates this:

```scheme
(define fact2
  (letrec ([fact-a (lambda (n acc)
                    (cond
                      [(= 0 n) acc]
                      [else (fact-a (- n 1) (* n acc))]))])
    (lambda (n) (fact-a n 1))))
```
Here are some examples of this:

; (sum vec) adds together the elements of vec:
(define sum
  (letrec ([sum-a (lambda (vec acc)
                            (cond
                                [(null? vec) acc]
                                [else (sum-a (cdr vec) (+ (car vec) acc))])])]
    (lambda (vec) (sum-a vec 0))))
; (reverse lat) reverses its argument, as you might expect:
(define reverse
  (letrec ([reverse-a (lambda (lat acc)
                    (cond
                      [(null? lat) acc]
                      [else (reverse-a (cdr lat) (cons (car lat) acc))]])])
    (lambda (lat) (reverse-a lat null))))
Sometimes this isn't so easy. Here's a version of rember:

```scheme
(define rember
    (letrec ([rember-a (lambda (x lat acc)
            (cond
                [(null? lat) (h acc null)]
                [(eq? x (car lat)) (h acc (cdr lat))]
                [else (rember-a x (cdr lat) (cons (car lat) acc))]])]
        [h (lambda (lat1 lat2)  ; h reverses lat1 onto lat2
            (cond
                [(null? lat1) lat2]
                [else (h (cdr lat1) (cons (car lat1) lat2))])])]
    (lambda (x lat) (rember-a x lat null)))
```
The other strategy for producing tail recursions is *Continuation-passing style*. This uses a concept called a *continuation* which we will discuss at the end of the semester.