Accumulator-Passing Style
Continuation-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:

```lisp
(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
(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))))

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:

```
(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:

```
(define rember
  (letrec ([rember-a (lambda (x lat acc)
      (cond
        [(null? lat) (reverse acc)]
        [(eq? x (car lat)) (h acc (cdr lat))]
        [else (rember-a x (cdr lat) (cons (car lat) acc))])]]
    [h (lambda (acc lat1) ; h reverses acc onto lat1
      (cond
        [(null? acc) lat1]
        [else (h (cdr acc) (cons (car acc) lat1))])])]
  (lambda (x lat) (rember-a x lat null))))
```
The other strategy for producing tail recursions is *Continuation-passing style*. The continuation of an expression is what we do with the result of that expression. For example, the continuation of the expression (+ 3 4) in (* 2 (+ 3 4)) is that we multiply it by 2. We represent continuations in Scheme as functions of 1 variable, where the variable represents the result of the expression. The continuation of (+ 3 4) in the expression (* 2 (+ 3 4) ) is (lambda (y) (* 2 y)).
In continuation-passing style, the recursive functions carry their continuations around as an extra parameter. Here is an example that sums the elements of a vector:

```
(define sum
  (letrec ([sum-k (lambda (vec k)
                (cond
                  [(null? vec) (k 0)]
                  [else (sum-k (cdr vec) (lambda (y) (k (+ y (car vec))))))])])
    (lambda (vec) (sum-k vec (lambda (x) x)))))
```
(define sum
  (letrec ([sum-k (lambda (vec k)
      (cond
        [(null? vec) (k 0)]
        [else (sum-k (cdr vec) (lambda (y) (k (+ y (car vec))))))])]
    (lambda (vec) (sum-k vec (lambda (x) x)))))

The interesting line is the else condition of the cond expression. We do a tail-recursive call on the cdr of vec. That much isn't surprising. The new continuation is lambda (y), where y represents the answer to (sum-k (cdr vec)...) We take y and add (car vec) to it; this gets us the sum of vec. We then apply k, the incoming continuation to this, because k tells us what to do with the answer.
(define sum
  (letrec ([sum-k (lambda (vec k)
      (cond
        [(null? vec) (k 0)]
        [else (sum-k (cdr vec) (lambda (y) (k (+ y (car vec))))))]]))
    (lambda (vec) (sum-k vec (lambda (x) x))))

Note that the top-level continuation, which we give at the start of the computation, is always (lambda (x) x): This means "return the answer".
Here are rules for writing continuation-passing style functions:

• Continuations are represented by functions of one argument. You can think of this argument as the result of the recursive call.
• At the top level the continuation is always the identity: (lambda (y) y)
• Every recursive function gets an additional argument, k, which is the continuation for a call to this function.
• The continuation parameter must be applied to any answer produced by the function -- instead of returning x we return (k x)
• All recursive calls are tail-recursive. Context gained during evaluation of the function is incorporated in the new continuation passed in the recursive call.
Here is a reverse function done in continuation-passing style:

(define reverse
  (letrec ([rev-k (lambda (lat k)
                    (cond
                      ((null? lat) (k null))
                      [else (rev-k (cdr lat) (lambda (y) (k (append y (list (car lat)))))))
                      ])
          )
   (lambda (lat) (rev-k lat (lambda (x) x))))
)

The continuation for the recursive call is (lambda (y)... so y will be the reversal of (cdr lat), append y onto the list whose only element is (car lat) -- this gets us the reversal of lat -- and apply the incoming continuation to this result.
One more example. The append function joins together two lists: (append '(1 2 3) '(4 5 6)) is (1 2 3 4 5 6).

(define append
  (letrec ([append-k (lambda (lat1 lat2 k)
          (cond
            [(null? lat1) (k lat2)]
            [else (append-k (cdr lat1) lat2 (lambda (y) (k(cons (car lat1)y))))])])
       (lambda (lat1 lat2) (append-k lat1 lat2 (lambda (x) x)))))

The continuation for the recursive call says "Let y be the result of appending (cdr lat1) onto lat2.  cons (car lat1) onto y, and apply the incoming continuation to the result."
Programming with explicit continuations gives you a lot of control. Here is one example of this. Consider a simple recursive function that sums the elements of a vector

\[
\text{(define sum} \\
\text{\quad (lambda (vec) \} \\
\text{\quad \quad (cond \} \\
\text{\quad \quad \quad [(null? vec) 0] \} \\
\text{\quad \quad \quad [else (+ (car vec) (sum (cdr vec)))]])))}
\]

Suppose we want to modify this to return 'error if we get to an element of vec that isn't a number.
The following doesn't work:

```
(define sum1
  (lambda (vec)
    (cond
      [(null? vec) 0]
      [(not (number? (car vec))) 'error]
      [else (+ (car vec) (sum1 (cdr vec)))])))
```

If we call this with a bad "vector": (sum1 '(1 2 bob), then when we recurse to (sum1 '(2 bob)) we want to add 2 to (sum1 '(bob)). However, (sum1 '(bob)) is 'error, and we can't add 2 to 'error, so our program crashes
However, since continuation-passing style uses tail-recursion, we can pass the 'error symbol back to the top. The following does work:

(define sum2
  (letrec ([sum-k (lambda (vec k)
                    (cond
                      [(null? vec) (k 0)]
                      [(not (number? (car vec))) 'error]
                      [else (sum-k (cdr vec) (lambda (y) (k (+ y (car vec))))))]
                    (lambda (vec) (sum-k vec (lambda (x) x))))]

Now (sum2 '(1 2 3)) is 6 and (sum2 '(1 2 bob)) is 'error
This is even better done by giving an explicit error continuation, which tells us what to do with an error:

(define sum3
  (letrec ([sum-k (lambda (vec k err)
        (cond
          [(null? vec) (k 0)]
          [(not (number? (car vec))) (err (car vec))]
          [else (sum-k (cdr vec)
                        (lambda (y) (k (+ y (car vec))))
                        err)])]]
       (lambda (vec) (sum-k vec
                        (lambda (x) x)
                        (lambda (x) (list 'bad-element x))))))

)
Now (sum3 '(1 2 3)) is 6 and (sum3 '(1 2 bob)) is (bad-element bob).