Continuation Passing Style
Consider our function dynamics, represented by a table such as the following:

<table>
<thead>
<tr>
<th>Direction</th>
<th>Expression</th>
<th>Continuation</th>
<th>Result of Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>(fact 3)</td>
<td></td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact 2)</td>
<td>(* 3 □)</td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact 1)</td>
<td>(* 3 (*2 □))</td>
<td>?</td>
</tr>
<tr>
<td>IN</td>
<td>(fact 0)</td>
<td>(* 3 (* 2 (* 1 □)))</td>
<td>?</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 0)</td>
<td>(* 3 (* 2 (* 1 1)))</td>
<td>1</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 1)</td>
<td>(* 3 (* 2 1))</td>
<td>1</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 2)</td>
<td>(* 3 2)</td>
<td>2</td>
</tr>
<tr>
<td>OUT</td>
<td>(fact 3)</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

We could represent any of these continuations by a function of one argument, such as `(lambda (y) (* 3 (* 2 y)))`
We talked more informally about CPS on February 20.
This leads to the following style of coding in which we explicitly build up continuations. Here are the rules:

- 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: 
  \( \text{(lambda (y) y)} \)
- 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.
For example, here is the CPS version of the factorial function:

\[
(\text{define fact-k (lambda (n k)}
\quad (\text{if (= 0 n) (k 1)}
\quad \quad (\text{fact-k (- n 1) (lambda (y) (k (* n y)))))})
\]

At the top level we call this with

\[
(\text{fact-k n (lambda (y) y) })
\]
E.g.

\begin{verbatim}
(fact-k 3 (lambda (y) y) )
  k0
(fact-k 2 (lambda (z) (k0 (* 3 z) ) ) )
  k1
(fact-k 1 (lambda (x) (k1 (* 2 x) ) ) )
  k2
(fact-k 0 (lambda (t) (k2 (* 1 t) ) ) )
  k3

(k3 1)
(k2 1)
(k1 2)
(k0 6)
6
\end{verbatim}
Now, why would we ever do this?

Consider the following function, which multiplies together the values in a vector (flat list of numbers):

```
(define prod (lambda (vec)
    (if (null? vec) 1 (* (car vec) (prod (cdr vec))))))
```

```
(prod '(5 4 3 0 2 1 0)) ultimately produces
(* 5 (* 4 (* 3 (* 0 (* 2 (* 1 0))))))
```

This is quite stupid. It could be worse -- we could be doing a much more expensive operation than multiplication, and the list could be a lot longer.
We could change prod to

```
(define prod (lambda (vec)
    (if (null? vec)
        1
        (if (= (car vec) 0)
            0
            (* (car vec) (prod (cdr vec)))))
)
```

but this doesn't help; with (prod '(4 3 2 1 0)) we still end up computing

```
(* 4 (* 3 (* 2 (* 1 0))))
```
Here is a CPS version

```
(define prod-k (lambda (vec k)
    (if (null? vec)
        (k 1)
        (if (= (car vec) 0)
            (k 0)
            (prod-k (cdr vec)
                (lambda (y) (if (= 0 y)
                    0
                    (k (* (car vec) y))))))))
```

Now on `(prod-k '(4 3 2 1 0) (lambda (y) y))` we replace 5 products with a comparison to 0.
Here is an even better solution

\[
\text{(define prod2-k (lambda (vec\ k\ escape)
  (if (null? vec)
    (k 1)
    (if (= 0 (car vec))
      (escape 0)
      (prod2-k (cdr vec)
        (lambda (z) (k (* (car vec) z)))
        escape)))))
\]

We might define a standard escape routine:

\[
\text{(define myExit (lambda (y) (printf "Exiting with ~s\n" y)))}
\]
Now \((\text{prod2-k } '(4 \ 3 \ 2 \ 1 \ 0) \ (\lambda \ (y) \ y) \ \text{myExit})\) prints "Exiting with 0"
Note that CPS doesn't always give an efficient program. Here is the CPS version of the Fibonacci function:

```
(define fib-k (lambda (n k)
    (if (= 0 n)
        (k 0)
        (if (= 1 n)
            (k 1)
            (fib-k (- n 1) (lambda (y) (fib-k (- n 2) (lambda (z) (k (+ y z))))))))))
```
We could add an escape continuation to this:

```scheme
(define fib-ke (lambda (n k esc)
    (cond
     [(= 0 n) (k 0)]
     [(= 1 n) (k 1)]
     [(= 5 n) (esc 5)]
     [else (fib-ke (- n 1)
              (lambda (x) (fib-ke (- n 2) (lambda (y) (k (+ x y))) esc))
                esc))])))```
Can we find CPS versions of append and reverse?
(define append-k (lambda (L1 L2 k)
    (cond
        [(null? L1) (k L2)]
        [else (append-k (cdr L1)
            L2
            (lambda (y) (k (cons (car L1) y))))])))

(define reverse-k (lambda (lat k)
    (cond
        [(null? lat) (k null)]
        [else (reverse-k (cdr lat)
            (lambda (x) (k (append-k
                x
                (list (car lat))
                (lambda (z) z))))))])))
Note that we could easily add an escape continuation to these. For example:

```
(define reverse-ke
    (lambda (L k escape)
        (cond
            [(null? L) (k null)]
            [(eq? 0 (car L)) (escape 0)]
            [else (reverse-ke
                        (cdr L)
                        (lambda (x)
                            (k (append-ke
                                x
                                (list (car L))
                                (lambda (y) y))))
                            escape)]))
```

What about (rember x L) ; removes first instance of x from L

or

(rember* x L) ; removes all instances of x from L
(define rember-k (lambda (x L k)
    (cond
        [(null? L) (k null)]
        [(eq? x (car L)) (k (cdr L))]
        [else (rember-k x (cdr L) (lambda (y) (k (cons (car L) y))))])))

(define rember*-k (lambda (x L k) ;; removes all occurrences of x from L
    (cond
        [(null? L) (k null)]
        [(eq? x (car L)) (rember*-k x (cdr L) (lambda (y) (k y)))]
        [else (rember*-k x (cdr L) (lambda (y) (k (cons (car L) y)))))])
)