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)))
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: `(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:

```
(define fact-k (lambda (n k)
    (if (= 0 n) (k 1)
        (fact-k (- n 1) (lambda (y) (k (* n y))))))))
```

At the top level we call this with

```
(fact-k n (lambda (y) y) )
```
E.g.

\[
\text{fact-k}\ 3\ (\lambda\ (y)\ y) \ \\
\text{k0}
\]

\[
\text{fact-k}\ 2\ (\lambda\ (z)\ (\text{k0\ (\ast\ 3\ z)\ } )) \ \\
\text{k1}
\]

\[
\text{fact-k}\ 1\ (\lambda\ (x)\ (\text{k1\ (\ast\ 2\ x)\ } )) \ \\
\text{k2}
\]

\[
\text{fact-k}\ 0\ (\lambda\ (t)\ (\text{k2\ (\ast\ 1\ t)\ } )) \ \\
\text{k3}
\]

\[
(\text{k3\ 1})
\]

\[
(\text{k2\ 1})
\]

\[
(\text{k1\ 2})
\]

\[
(\text{k0\ 6})
\]

\[
6
\]
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

\[
(\text{define prod (lambda (vec)}
   \quad (\text{if (null? vec)}
     \quad 1
     \quad (\text{if (= (car vec) 0)}
       \quad 0
       \quad (* (\text{car vec}) (\text{prod (cdr vec)})))))))
\]

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

(define prod2-k (lambda (vec k escape)
    (if (null? vec)
        (k 1)
        (if (= 0 (car vec))
            (escape 0)
            (prod2-k (cdr vec) k (lambda(z) (k (* (car vec) z))) escape))))

We might define a standard escape routine:
(define myExit (lambda (y) (printf "Exiting with ~s\n" y)))
Now (prod2-k '(4 3 2 1 0) (lambda (y) y) 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:

```
(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 \((\text{count } x \ L)\) which counts the number of occurrences of \(x\) in the arbitrary list \(L\)?

\[
(\text{define count-k} \ \\
(\lambda (x \ L \ k) \ \\
(\text{cond} \ \\
\quad [(\text{null? } L) \ (k \ 0)] \ \\
\quad [(\text{atom?} \ (\text{car} \ L)) \ (\text{if} \ (\text{eq?} \ (\text{car} \ L) \ x) \ \\
\quad \quad (\text{count-k} \ x \ (\text{cdr} \ L) \ (\lambda (t) \ (k \ (+ \ t \ 1)))) \ \\
\quad \quad (\text{count-k} \ x \ (\text{cdr} \ L) \ (\lambda (t) \ (k \ t)))] \ \\
\quad [\text{else} \ (\text{count-k} \ x \ (\text{car} \ L) \ \\
\quad \quad (\lambda (t) \ (\text{count-k} \ x \ (\text{cdr} \ L) \ \\
\quad \quad \quad (\lambda (s) \ (k \ (+ \ s \ t)))))])])
\]