Continuation-Passing Style

Remember accumulator-passing style? Another 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-k
      (lambda (vec   k)
              (cond
                   [(null? vec) (k 0)]
                   [else (sum-k (cdr vec) (lambda (y) (k (+ y (car vec)))))])))
```

At the top level the continuation is (lambda (x) x). So
        (sum-k '(1 2 3 4) (lambda (x) x)) is 10.

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

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.

Note that the top-level continuation, which we give at the start of the computation, is usually (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 rev-k
        (lambda (lat k)
                (cond
                        [(null? lat) (k null)]
                        [else (rev-k (cdr lat) (lambda (y) (k (append y (list (car lat))))))])))
```

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

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

```
(define sum
        (lambda (vec)
                (cond
                        [(null? vec) 0]
                        [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 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)))))])))
```

```
Now (sum-k '(1 2 3)  (lambda (x) x)) is 6
        (sum-k '(1 2 bob) (lambda (x) x)) 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).