```
#|

CPS Lecture

No published books do this subject justice (including Dan's!)

Which part of (f (g (h i) j) k) can be done first? (h i), since it
must be evaluated before (g (h i) j) can be applied.

What about (f (g (h i) (j k)))? Scheme doesn't specify the order in
which arguments are evaluated so it could be either (h i) or (j k).

So, let's take control. (h i (lambda (hi) ...)) We assume that hi is
the result of applying (h i). Then, we drop in everything else that
has to be done to replace the ...:

(f (g (h i) (j l)))

becomes

(h i (lambda (hi) (f (g hi (j l)))))

(lambda (hi) (f (g hi (j l)))) is a continuation. hi only appears once
in the body of the continuation, because hi is intended to replace (h
i) and only (h i).

Let's write rember in CPS. First, the direct style:

(define rember8
  (lambda (ls)
    (cond
      [(null? ls) '()]
      [(= (car ls) 8) (cdr ls)]
      [else (cons (car ls) (rember8 (cdr ls)))])))

First rule: whenever we see a lambda in the code we want to CPS, we
have to add an argument, and then process the body:

(lambda (x ...) ...) => (lambda (x ... k) ...^)

Let's start by adding a k to the outer lambda:

(define rember8
  (lambda (ls k)
    (cond
      [(null? ls) '()]
      [(= (car ls) 8) (cdr ls)]
      [else (cons (car ls) (rember8 (cdr ls)))])))

Now, to handle the rest of the program, we have to introduce a new rule.

***

Second rule: "Don't sweat the small stuff!"

Small stuff is stuff we know will terminate right away.

Don't sweat the small stuff if we know it will be evaluated.

Don't sweat the small stuff if it *might* be evaluated, but instead
pass it to k.

***

A good example of the first is (null? ls) in the first cond line. We
know it will be evaluated, and we know it's small stuff, so we don't
have to worry about it.

What about the '() that's returned as an answer? The other part of the
second rule is that if some small stuff *might* be evaluated, we just
pass it to k.

After applying the second rule to the first cond line, we get:

(define rember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]
      [(= (car ls) 8) (cdr ls)]
      [else (cons (car ls) (rember8 (cdr ls)))])))

The second cond line also has small stuff in both the test and the
return value, so we can treat it just like the first line.

(define rember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]
      [(= (car ls) 8) (k (cdr ls))]
      [else (cons (car ls) (rember8 (cdr ls)))])))

The else case, however, does *not* have small stuff as a return value,
so we have to build a new continuation:

(define rember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]
      [(= (car ls) 8) (k (cdr ls))]
      [else (rember8 (cdr ls) (lambda (x) (cons (car ls) x)))])))

We're not quite done, though, since we now have small stuff in the
body of the continuation. So, we just pass it to k:

(define rember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]
      [(= (car ls) 8) (k (cdr ls))]
      [else (rember8 (cdr ls) (lambda (x) (k (cons (car ls) x))))])))

This is now completely CPSed, but how do we invoke it? After all, we
need a k to pass in. Since (rember8 '() k) should be '(), k can be the
identity function (lambda (x) x):

> (rember8 '(1 2 8 3 4 6 7 8 5) (lambda (x) x))

What properties can we observe about this program?

First, all non-small stuff calls are tail calls. Here's the program
with the tail calls surrounded by asterisks:

(define rember8
  (lambda (ls k)
    (cond
      [(null? ls) (*k* '())]
      [(= (car ls) 8) (*k* (cdr ls))]
      [else (*rember8* (cdr ls) (lambda (x) (*k* (cons (car ls) x))))])))

Why don't null?, =, car, cdr, and cons count? Because they're just
small stuff, and when we combine small stuff together in small ways,
the combination remains small.

Second, all arguments are small stuff. Yep, even the lambda in the
else line, because lambda is *always* small stuff.

Notice that this is essentially a C program. All we have to do is
convert the continuations to data structures (remember how we did the
same thing with closures).

Let's trace (rember8  (lambda (x) x))

ls | k

'(1 2 8 3 4 6 7 8 5) | (lambda (x) x) = id
'(2 8 3 4 6 7 8 5)   | (lambda (x) (id (cons 1 x))) = k2
'(8 3 4 6 7 8 5)     | (lambda (x) (k2 (cons 2 x))) = k3

Once we hit the 8, we apply (k (cdr ls)) where k is k3 and ls is '(8 3
4 6 7 8 5)

(k3 '(3 4 6 7 8 5)) = (k2 (cons 2 '(3 4 6 7 8 5)))
(k2 '(2 3 4 6 7 8 5)) = (id (cons 1 '(2 3 4 6 7 8 5)))
(id '(1 2 3 4 6 7 8 5)) = '(1 2 3 4 6 7 8 5)

And we're done.

Let's try a more complicated program, multirember8. Instead of just
removing the first 8, it'll remove all of the 8s.

(define multirember8
  (lambda (ls)
    (cond
      [(null? ls) '()]
      [(= (car ls) 8) (multirember8 (cdr ls))]
      [else (cons (car ls) (multirember8 (cdr ls)))])))

Now, let's start CPSing by going back to our CPSed rember8:

(define multirember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]
      [(= (car ls) 8) (multirember8 (cdr ls))] ;; uh-oh!
      [else (multirember8 (cdr ls) (lambda (x) (k (cons (car ls) x))))])))

What do we need to do for the second line? Since multirember8 takes
two arguments, we need to now pass it a continuation.

(define multirember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]
      [(= (car ls) 8) (multirember8 (cdr ls) (lambda (x) (k x)))]
      [else (multirember8 (cdr ls) (lambda (x) (k (cons (car ls) x))))])))

But what's (lambda (x) (k x)) doing? It's taking whatever is passed to
it, and passing it to k. This whole expression, therefore, is
equivalent to k.

Eta reduction: (lambda (x) (M x)) = M if x is not free in M and M is
guaranteed to terminate. M is any arbitrary expression that satisfies
these rules; it doesn't have to be only a single variable like k.

So, whenever you see a tail call, you don't even have to think about
eta. Just pass k to it.

(define multirember8
  (lambda (ls k)
    (cond
      [(null? ls) (k '())]
      [(= (car ls) 8) (multirember8 (cdr ls) k)]
      [else (multirember8 (cdr ls) (lambda (x) (k (cons (car ls) x))))])))

|#
```