```
#|
Lecture Topic: Continuous Passing Style (CPS)
Date: 02162010


The following notes were copied from Adam Foltzer

This lecture information will not be found in a book.

Given (f (g (h i) j) k) what can be done first? => (h i)
Why? It must be evaluated before (g(h i)j).

How about (f (g (h i) (j l)))? => either (h i) or (j l) scheme doesn't care

How to take control?
Start with the following expression:
(hi
  (lambda(hi)
    ...))

1. Assume that hi is the result of applying (h i)
2. drop in everything else to replace the ...

so...  (f (g (h i) (j l))) becomes

(hi
  (lamda (hi)
    (f (g hi (j l)))

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

Write rember* in CPS

Direct style
|#

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

#|

To convert to CPS style use the following rules
Rule 1:
when ever we see a lambda in the code and we want to use CPS style then
A. add an argument
B. process the body

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

Start by added the argument


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

```
Rule 2:
Don't sweat the small stuff!!!!
-small stuff is stuff we know will terminate right away
-don't sweat it if you know it will be evaluated
-if it might be evaluated, instead pass it to k

Ex. (null? ls)
-we know it will be evaluated
-we know it is small stuff
-we don't worry about it

So what do you do with the '() that is returned as the answer to (null? ls)?
=> pass it to k

|#

(define rember8
  (lambda(ls k)
    (cond
      [(null? ls) (k '()) ] ;;small stuff
      [(= (car ls) 8) (k (cdr ls))] ;;small stuff
      [else (cons (car ls) (rember8 (cdr ls)))] ;; not small stuff
      )))

#|
[else (cons (car ls) (rember8 (cdr ls)))]  is not small stuff so
need to build a new continuation
there is still small stuff in the body so we pass it to k
the else line is shown on the following
|#

;rember8 that is CPSed
(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))))]
      )))

#|
how do you invoke it?  => we need a k and ls to pass in as follows
(rember8 '() k) => '()

k could be the identify function
(lambda (x)
  x)

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

What properties can be observed about this program?
1. All non-small stuff calls are tail calls
EX surround tail calls with *

(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, & cons count?

```
1. they are small stuff (if we combine small stuff together in small things
                            the combination remains small)
2. All arguments = small stuff
   lambda        = small stuff
```

This is essentially a 'C' program. Just convert the continuation to data structures. "This was done with closure"

How about we trace (rember8 '(1 2 8 3 4 6 7 8 5) (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 8 we apply (k (cdr ls))
k = k3
ls = '(8 3 4 6 7 8 5)
'(8 3 4 6 7 8 5)     |  (lambda (x)
                          (k2 (cons 2 x))) = k3
(k3 '(3 4 6 7 8 5)) =>
(k2 (cons 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)
Done
```

remove all 8's

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

Let's CPS this thing

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

**multirember8 takes two arguments
=> so you need to make another 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))))])))
```

what does (lambda (x) (k x)) do?

=> it takes the whatever is passed "x" and passes it to k

```
Eta reduction: (lambda (x)
                 (M x))
```

M if x is not free in M & M is going to terminate

M = any arbitrary expression that satisfies the above rule (does not have to be a single variable like k)

so when every you see a tail call you dont even need 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))))])))


|#
```