Make Up Class 4/10/24 (by Zoom)

HW 7 - #2

Recursive list-ref

pre: ℓ is a list, and j is an index into that list

post: (list-ref ℓ j) returns the index j elt of ℓ

idea: defer cdr - we'll have a wrapper, but the recursive function will assemble a stack of cdrs.

For example:

(car (cdr (cdr (cdr '(1 2 3 4 5)))) =
car '(4 5)) = 4 (index 3 elt)

```
(define (rec ℓ j)
  (cond ((zero? j) (car ℓ))
        (else (cdr (rec ?? ...)))))
```

Looks wrong. So let's try moving car to the wrapper:

```
(define (rec ℓ j)
  (cond ((zero? j) ℓ)
        (else (cdr (rec ℓ (- j 1))))))
```

{ want to return (cdr (cdr ... (cdr ℓ) ...))
j cdrs

What are we recursing on? Clearly not ℓ!

So - j.

D&C: to apply cdr j times to l,
we apply cdr to $(\text{cdr}^{j-1}\ l)$.

We need a wrapper:

```
(define (my-listref l j)
  (car (rec l j)))
```

Thus we return the index j element of l.

Check: has the 0-based indexing been correctly worked in?

Yes - by induction on j. { work this out!

[Notice that rec itself does not cdr down l in the usual sense -
rec returns the result of applying cdr j times to l.]

Let's take another look at this problem. A problem from A&S ch 1 asked us to write a higher order function which inputs 2 composable functions f and g and which returns the composed function

lambda → $\lambda x . f(g(x))$ { math notation

For example

(compose square cube)

would be called

((compose square cube) 2)

giving 64

```
; pre: f and g are composable
(define (compose f g)
  (lambda (x) (f (g x))))
; post: returns λx. f(g(x)).
```

And a related problem is (iirc)

Write a function repeated which works

((repeated square 2) 5) = 625

```
(define (repeated f n)
  (cond ((zero? n) (lambda (x) x))
        (else (compose f (repeated f (- n 1))))))
```

Pf by induction on n ✓ { Note that (repeated f n) is a function.

You should rework your solutions to HW 7 using repeated! It might not apply to all of those problems, but it certainly applies to a number of them.

Let's reconsider the recursive list-ref ---

we had

```
(define (rec l j)
  (cond ((zero? j) l)
        (else (cdr (rec l (- j 1))))))
```

```
(define (alt-rec l j)
  (repeated cdr j))
```

This returns the function

$\lambda x.(cdr^{j} \; x)$ ← sloppy but intuitive notation

which is not what rec did. Can we change the wrapper to make this compute list-ref?

How about you all working this out?