CS 61A Lab 13 Solutions

LAB
===

4.27

```
; Type thse in first...
(define count 0)
(define (id x)
  (set! count (+ count 1))
  x)
(define w (id (id 10)))

;;; L-Eval input:
count
;;; L-Eval value:
1

;;; L-Eval input:
w
;;; L-Eval value:
10

;;; L-Eval input:
count
;;; L-Eval value:
2
```

What's going on here? Well, basically COUNT just refers to how many times ID has been called. So before we ask for the value of W, it's only been called once. Why?

DEFINE is a special form, so it behaves exactly like in the regular metacircular evaluator: we evaluate the second argument. This is a call to a compound procedure, so we /delay/ the inner call to (id 10). We can then go ahead and call ID with this newly created promise.

This increments COUNT, but never actually forces the promise, and so that's what W is bound to: a promise to evaluate (id 10) in the global environment.

When we actually ask for the value of W, we /have/ to force the promise, in order to print out a real value to the user. This evaluates the call to (id 10), which makes a promise to evaluate 10 and calls ID.

This increments COUNT again. The value returned is the promise to evaluate 10. This too gets

forced; the value is just 10.

At a sort of high level, this is an "outside-in" sort of evaluation: we evaluated the outside call to ID before the inside one. This is part of normal-order evaluation. By contrast, applicative-order evaluation (like regular Scheme or the original MCE) evaluates things "inside-out"; we'd have to call the inner ID and get a result before we could do anything with the outside one.

4.29

```
(define (square x)
  (* x x))

;;; L-Eval input:
(square (id 10))
;;; L-Eval value:
100

;;; L-Eval input:
count
;;; L-Eval value:
_____
```

This was an interesting question. SQUARE is a compound procedure, so we delay the call to (id 10). Because it's each call to ID that increments COUNT, we're only interested in how many times this promise gets forced: once for each use of X in SQUARE.

In a memoizing evaluator (like this lazy evaluator), the first time the promise is forced, the result (10) will be remembered, and we won't call ID again. So COUNT will end up with 1.

In a non-memoizing evaluator (like the normal-order evaluator Brian showed at the start of the semester), ID will be called both times the promise is forced. The answer will still be 100, but COUNT will end up being 2 instead.

The general question said to consider a program that would run much more slowly without memoization; pretty much any time you do an expensive computation and use the answer more than once, it would be much faster to memoize. It's like using LET to store values so you don't have to do work again.

4.55

```
(supervisor ?x (Bitdiddle Ben))
```

(job ?x (accounting . ?y))

(address ?x (Slumerville . ?y))

The dots are needed because (accounting ?y), for example, would match only entries in which there was a single element after the word "accounting." That is, (accounting ?y) would match (accounting scrivener) but not (accounting chief accountant).

4.62
The base case here involves a 1-element list, not the empty list.

```
(rule (last-pair (?x) (?x)))

(rule (last-pair (?y . ?z) ?x)
      (last-pair ?z ?x))
```