**CS61A Notes — Week 08b: Review**

**QUESTION 1.**

```
(define (combiner a b)
  (if (< (length a) (length b)) a b))

(define (fast-grad tree)
  (if (null? (children tree))
      (list (datum tree))
      (cons (datum tree)
            (accumulate combiner (map fast-grad (children tree))))))
```

**QUESTION 2.**

```
(rule (is-connected ?here ?end (?end))
      (connected ?here ?end))

(rule (is-connected ?here ?end (?next . ?there))
      (and (connected ?here ?next)
           (is-connected ?next ?end ?there)))
```

**QUESTION 3.**

1, 4, 16, 64, 256 (even powers of 2)

**QUESTION 4.**

a)

```
(define (average-maker tagged-grades)
  (for-each
    (lambda (grade)
      (let ((total (get (type-tag grade) 'total))
            (app (get (type-tag records) 'appearances)))
        (put (type-tag grade) 'total
             (+ (content grade)
                (if total total 0)))
        (put (type-tag grade) 'appearances
             (+ 1 (if app app 0)))))
    tagged-grades)
```

b)

```
(define (get-average type)
  (let ((total (get type 'total))
        (app (get type 'appearances)))
    (if total (/ total app) 0)))
```

**QUESTION 5.**

```
*** ERROR: Unbound variable: if
```

Remember what 'if' is in the MCE. It's a variable, so it will do a variable lookup, but will there be a variable 'if' in the MCE environment? Nope it isn't!

**QUESTION 6.**

```
(let ((a 3) (b a))
  (+ a 4))
```

ERROR, in the MCE and analyzing evaluators. VALUE in the lazy evaluator.

The problem with this expression is that the variable A is used to assign a value to B, but it is only bound in the body of the LET. This can be seen easily if we de-sugar the LET into:

```
( (lambda (a b) (+ a 4))
  3 a )
```

In regular Scheme (the MCE), we evaluate the subexpressions before we call the procedure, and we get an error because A is unbound. Clearly nothing changes if we evaluate the expression in the analyzing evaluator, because the analyzing evaluator is exactly the same as the vanilla MCE above the line, albeit a little faster. The lazy evaluator, however, does not have any problems with this expression because the arguments to the procedure, 3 and A, are delayed. The value of B is never needed, so it is never computed.

```
(let ((a 3) (b a))
  (+ a b))
```

This produces an ERROR in all of the evaluators listed.

Why does this fail in lazy, whereas the earlier LET worked? Consider the situation when we have just called the procedure corresponding to the LET. A is bound to a thunk that, when forced, evaluates the expression `3' in the global environment. B is bound to a thunk that evaluates the expression `A' in the global environment. We evaluate (+ A B), and both thunks are forced because + is a primitive. But notice that forcing B causes the expression `A' to be evaluated in the *global* environment, where A isn't bound. Note that each thunk carries with it both an expression and an environment in which the expression was supposed to be evaluated. This allows the lazy evaluator to follow the same scoping rules as an applicative order evaluator.

**QUESTION 7.**

```
> b
(1 2 7 8 5)
> c
(1 2 (3 8 5) 6)
```

**QUESTION 9.**

This change makes ACTUAL-VALUE non-recursive. This means it won't correctly handle "thunks of thunks" or "thunks of thunks of thunks" etc. Things will break down when you have more than one level of thunkification.

```
(bar 3) => 3
```

BAR sees a thunk of 3 which when ACTUAL-VALUEd returns 3 — one level of thunkification is cool.

```
        ((lambda (x) (* x x)) ((lambda (x) x) 5)) => error: wrong args to *
```

* sees a thunk of ((lambda (x) x) 5) which when ACTUAL-VALUEd returns a thunk of 5 -- two levels of thunkification.

```
        (foo 6 7) => error: wrong args to * in BAR
```

This time the * in BAR is given a thunk of A which when ACTUAL-VALUEd returns a thunk of 6 -- two levels.

```
        (let ((a +) (b 4))
          (+ b b))           => 8
```

+ sees a thunk of 4 which when ACTUAL-VALUEd returns 4.

```
        (identity (identity 6))  => a thunk is printed instead of 6
```

The driver loop sees a thunk of (identity 6) which when ACTUAL-VALUEd returns a thunk of 6 -- two levels.

**QUESTION 10.**

```
(define useless-square
   (let ((val #f))
      (lambda (num)
         (if (not val)
           (begin (set! val (* num num)) val)
           val))))
```

**QUESTION 11.**

```
(define (flatten ls)
  (cond ((null? ls) ls)
        ((not (pair? (car ls)))
           (append (list (car ls)) (flatten (cdr ls))))
        (else
           (append (flatten (car ls)) (flatten (cdr ls))))))

(rule (pair? (?car . ?cdr)))

(rule (flatten () ()))

(rule (flatten (?car . ?cdr) ?res)
      (and (pair? ?car)
           (flatten ?car ?fcar) (flatten ?cdr ?fcdr)
           (append ?fcar ?fcdr ?res)))

(rule (flatten (?car . ?cdr) ?res)
      (and (not (pair? ?car)) (flatten ?cdr ?fcdr)
           (append (?car) ?fcdr ?res)))
```