Midterm 2 Review Solutions

1: Count-Pairs

```
(define (count-pairs deep-ls)
  (cond ((null? deep-ls) 0)
        ((atom? deep-ls) 0)
        ((pair? (car deep-ls))
         (+ 1 (count-pairs (car deep-ls))
              (count-pairs (cdr deep-ls))))
        (else (+ 1 (count-pairs (cdr deep-ls))))))
```

2: Trees

```
(define (sumpath tree)
   (define (help tree sum)
      (make-tree (+ (datum tree) sum)
                 (map (lambda (child) (help child (+ (datum tree) sum)))
                 (children tree))))
   (help tree 0))
```

3:

Conventional:

```
(define (get-Pokemon record)
  (let ((tag (type-tag record)))
    (cond ((equal? tag ‘eric) (car (contents record)))
          ((equal? tag ‘Phill) (car (contents record)))
          ((equal? tag ‘Kevin) (error “Kevin only likes My Little Pony”)))))

(define (get-Level record)
  (let ((tag (type-tag record)))
    (cond ((equal? tag ‘eric) (cdr (contents record)))
          ((equal? tag ‘Phill) (cadr (contents record)))
          ((equal? tag ‘Kevin (error “Kevin only.....”)))))
```

DDP:

```
(define (get-Pokemon record)
  ((get (type-tag record) ‘Pokemon) (contents record)))
(put ‘eric ‘Pokemon car)
(put ‘phill ‘Pokemon car)
(put ‘kevin ‘Pokemon (error “....”))

(define (get-Level record)
   ((get (type-tag record) ‘level) (contents record)))
(put ‘eric ‘level cdr)
(put ‘phill ‘level cadr)
```

```
(put ‘kevin ‘level (error “....”))


(define (is-Super-Effective? record type)
  ((get (type-tag record) ‘is-Super-Effective?) type))
(put ‘eric ‘is-Super-Effective? (lambda (type) (equal? type ‘grass)))
(put ‘phill ‘is-Super-Effective? (lambda (type) (equal? type ‘fire)))
(put ‘kevin ‘is-Super-Effective? (lambda (type) (error “Ponies Rule!!”)))


4:
(define make-dog
  (let ((owner 'master))
    (lambda (name)
      (let ((hunger 0))
          (lambda (m)
            (cond ((equal? m 'fetch)
                   (lambda (n)
                     (if (> hunger 10)
                         (error "Dog needs to Eat")
                         (set! hunger (+ hunger n)))))
                  ((equal? m 'eat)
                   (set! hunger 0))
                  ((equal? m 'hunger) hunger)
                  ((equal? m 'owner) owner)
                  ((equal? m 'name) name)
                  (else (error "Bad Message")))))))
```

5: Use envdraw to check your solution

6:

```
(define make-person
  (lambda (name)
    (let ((salary 100) (money 0))
      (lambda (m)
          (cond ((eq? m 'work)
                 (set! money (+ money salary)))
                ((eq? m 'new-salary)
                 (lambda (amount) (set! salary amount)))
                ((eq? m 'name) name)
                ((eq? m 'salary) salary)
                ((eq? m 'money) money)
                (else (error "Bad Message")))))))
```

7:

Solutions to Blanks In Order

The trick here was to realize that in order for any of this to work at all, that x and y had to be eq?

```
>  (eq? x y)
>  #t
```

```
(set-cdr! (cdr z) three)
(set-cdr! (cdr x) three)
```

(set-car! x 4)

(set-car! three 6)

8:

Eval-1 is called 16 times

Apply-1 is called 5 times

(Disclaimer: I double checked the above solution multiple times, but it is tough to trace, there is still a chance it might be wrong, let us know if you think it is, but I am fairly certain it is correct)