Yaaaaaay! Midterm 2, Let's Review!!!!

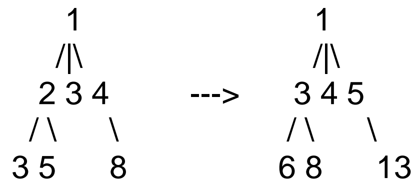
1: Deep List Warm Up

Write the procedure count-pairs that takes an argument list. This list can be a flat list or a deep-list, but is always a proper list. Count-pairs returns the number of pairs in the list.

```
> (count-pairs  '(1 2 3))

> 3

> (count-pairs  '((1 2) (3 4) ((5)) 6))

> 10
```

2: Tree Revenge!!

Write a procedure sumpath that takes a Tree (with datum and children) of numbers as its argument, and returns a tree of the same shape in which each datum is the sum of all the data between the original datum and the root. For example, in the case below the value 5 in the original tree becomes 8 (1 + 2 + 5) in the returned tree:



3: Generichu!! I Choose You!!

So the other day, the TA's were supposed to be having one of their weekly meetings to do cool CS stuff……But instead were goofing off and having heated Pokemon battles. The problem was that we kept getting errors when our Pokemon fought because we all represented our chosen Pokemon differently. Let's fix this!!!

Let's look at Eric, Phill, and Kevin. So we each have only one Pokemon. Let's start out simple. A Pokemon is stored by what Pokemon it is, and what level it is. Eric has a Charmander, and represents it as a pair. (Charmander . 12) Phill has a Squirttle, and represents it as a list (Squirttle 14). Now each Pokemon is also tagged with its trainer's name by the following procedures

```
(define attach-tag cons)

(define type-tag car)

(define contents cdr)
```

What about Kevin? Well, Kevin thinks he's too good for Pokemon, so whenever you see a Pokemon tagged with Kevin, you just get back, "This is dumb, I'm gonna go watch My Little Pony…"

Write generic operating procedures get-Pokemon, and get-Level, that take in any tagged Pokemon and behave accordingly. Do this first using conventional style, then using DDP.

Lastly, we want to get a little trickier, add a new generic-operation is-Super-Effective? that takes in a tagged-Pokemon and a type, and returns true or false based on whether or not the tagged-Pokemon is super effective against the given type. Add this as a DDP generic operator.

Note: Phill's Squirttle is super effective against fire, Eric's Charmander is super-effective against grass. Kevin is still watching My Little Pony. ☺

4: SQUIRREL!! BALL!! LOVE!! (That's right, everyone loves dogs ☺)

OOP Below The Line

Using normal scheme (not OOP above the line syntax) write a procedure make-dog that creates a dog-object. The dog should have an instantiation name variable, an owner class variable. This owner can just be the word 'master, since to every single dog that's all their owner is. A hunger instance-variable represented as a number that starts at zero. Dogs should accept two main messages fetch and eat that provide the following behavior. Fetch takes a number argument n, as long as a dog's hunger isn't above 10, the dog will fetch n times, which is equivalent to having its hunger increase by n. If a dog's is too hungry, he will complain that he/she needs to eat. When a dog eats, his/her hunger goes back to zero. A dog should also respond to name, hunger, and owner methods as well. Here is some examples:

```
>  (define dog-1 (make-dog 'fluffy))

>  (define dog-2 (make-dog 'sparky))

>  (dog-1 'name)

>  fluffy

>  (dog-2 'name)

>  sparky

>  ((dog-1 'fetch) 8)

>  ((dog-2 'fetch) 3)

>  (dog-1 'hunger)

>  8

>  (dog-2 'hunger)

>  3
```

> ((dog-1 ‘fetch) 3)

> ((dog-1 ‘fetch) 1)

> error: Dog is too hungry to fetch

> (dog-1 ‘eat)

> (dog-1 ‘hunger)

> 0

> (dog-2 ‘hunger)

> 3

> (dog-1 ‘owner)

> master

> (dog-2 ‘owner)

> master

5: Now this is getting Fun…Environment Fun

Env-Diagrams

Draw the Environment-Diagram for your definition of make-dog, and for the execution of the code sequence above

(Yes, this is long and tedious, but if you can do this, you will destroy all future env diagram questions, so take your time and do it right…)

6: Sorry, I think you're language is below my level….

Translate the following class definition using OOP above the line syntax to ordinary scheme below the line OOP syntax. Call the new definition make-person.

```
(define-class (person name)
    (instance-vars (salary 100) (money 0))
   (method (work) (set! money (+ money salary)))
   (method (new-salary amount) (set! salary amount)) )
```

If you have got through 4, 5, and 6, you are now a pro at OOP below the line and env diagrams!! Congrats!!

7: If I were a mutant my power would be…….

Mutation!!!

Fill in the blanks below to make STk behave as shown. You may not create any new pairs.
Solutions that do so will receive no points. In other words, you may not fill in the blanks with append, cons, list, or anything else that creates pairs.

```
; Assume x, y, and z have already been defined.

> (define a (append (list x y) z))
a

> (define three (list 3))
three

> a
((1 2) (1 2) 1 2)

> (set-cdr! (cdr z) _____________ )
okay

> a
((1 2) (1 2) 1 2 3)

> (set-cdr! ____________ ______________)
okay

> a
((1 2 3) (1 2 3) 1 2 3)

> (set-car! ____________ ____________ )
okay

> (set-car! _____________ ____________ )
okay

> a
((4 2 6) (4 2 6) 1 2 6)
```

8: SCHEME WAS NUMBER ONE!! (anybody?...)

How many times is eval-1 and apply-1 each called in the process of evaluating the following expression:

Scheme-I : (+ ((lambda (x) (* x 2)) ((lambda (y) (+ y 3)) 4)) 5)

19

(Dark Tower Anyone?)

The scheme-I code is provided here for your reference:

```
(define (scheme-1)
  (display "Scheme-1: ")
  (flush)
  (print (eval-1 (read)))
  (scheme-1))

(define (eval-1 exp)
  (cond ((constant? exp) exp)
        ((symbol? exp) (eval exp)) ; use underlying Scheme's EVAL
        ((quote-exp? exp) (cadr exp))
        ((if-exp? exp)
         (if (eval-1 (cadr exp))
             (eval-1 (caddr exp))
             (eval-1 (cadddr exp))))
       ((lambda-exp? exp) exp)
       ((pair? exp) (apply-1 (eval-1 (car exp)) ; eval the operator
                             (map eval-1 (cdr exp))))
       (else (error "bad expr: " exp))))


(define (apply-1 proc args)
  (cond ((procedure? proc) ; use underlying Scheme's APPLY
         (apply proc args))
        ((lambda-exp? proc)
         (eval-1 (substitute (caddr proc) ; the body
                             (cadr proc) ; the formal parameters
                             args ; the actual arguments
                             '()))) ; bound-vars, see below
       (else (error "bad proc: " proc))))
```