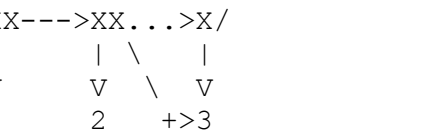
CS 61A Spring 2008 Midterm 3 solutions

1. Box and pointer.

```
> (let ((x (list 1 2 3)))
    (set-cdr! (cdr x) (caddr x))
    x)
```

(1 2 . 3)



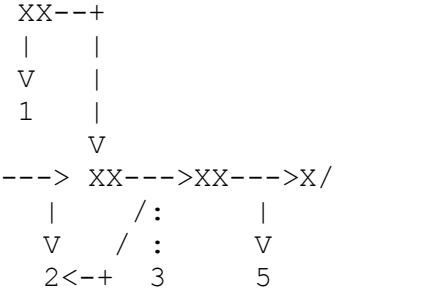
The SET-CDR! says to change the cdr of (cdr x). Since (cdr x) is the second pair in the picture above, we change the right arrow from that pair. The ...> in the picture is the link for (cddr x) before mutation; the new value is the original (caddr x), i.e., the third element of the list.

The resulting structure is not a list, since the new (cddr x) is neither a pair nor the empty list. Thus the printed representation includes a dot, which is not an element of the list, but merely indicates that the 3 is the cdr of the second pair rather than the car of the third. (That third pair is no longer part of the structure.)

Many people thought this would return (1 (2 . 3)), but this would require an extra spine pair pointing to the (2 . 3) pair, not a toplevel improper list.

```
> (define a (list 2 3 5))
> (define b (cons 1 a))
> (set-car! (cdr a) (cadr b))
> b
```

(1 2 2 5)



Before mutation, the value of B is (1 2 3 5). (CADR B) is the second element of B, namely the 2. (CDR A) is the second pair in A's spine, the one whose car is A's second element, 3. So the pointer to 3 is replaced by a pointer to the 2.

Many people thought that (cons 1 a) meant that b's cdr arrow should point either to the letter a or the variable a. Rmember that the arguments to cons
are *evaluated* first, so the arrow should point to the pair!

Some people thought this should return a circular list, but that would require
us to have said (set-car! (cdr a) (cdr b)).

```
> (let ((foo (list 0 3))
      (bar (list 7 10)))
    (set-car! (car foo) (car bar))
    foo)

ERROR: set-car!: wrong type of argument: 0
```

You shouldn't even have to draw a diagram for this one. (CAR FOO) is 0, which isn't a pair, so the SET-CAR! fails.

Scoring: One point per printed representation, one point per diagram, except that ERROR without a diagram got 2 points in the last part.

2. Translate Scheme to OOP.

Here's the beginning of the code you were given:

```
(define make-horse
  (let ((population 0))
    (lambda (color)
      (let ((hunger 0))
      (define (dispatch message)
        ...)))))
```

The procedure DISPATCH represents an instance of the HORSE class; the LAMBDA two lines earlier represents the class itself. So COLOR is an argument to the instantiation of the class, i.e., an instantiation variable.

```
(define-class (horse color)
  (class-vars (population 0))
  (instance-vars (hunger 0))
  (method (eat grass)
    (if (> hunger 0)
     (set! hunger (- hunger grass))
```

```
      (se color '(horse not hungry))))
  (initialize (set! population (+ population 1))))
```

The only part that should have been even slightly obscure is the INITIALIZE. The SET! expression is evaluated whenever the class procedure (the unnamed lambda expression) is invoked, before returning the dispatch procedure that represents the instance. Thus it's an initialization, although the actual representation of initialize methods is a little more complicated because of
the need to set SELF before calling the initializer.

The OOP system provides us with methods for COLOR, POPULATION, and HUNGER, so
you didn't need to write those. We didn't take off points for people who did.

Scoring: One point for COLOR as instantiation variable; one point for each of the four DEFINE-CLASS clauses.

3. What's in the stream?

I find it easiest to draw two alternating rows of dashes when there's an INTERLEAVE call involved, like this:

```
1  2  ___       ___       ___       ___

           ___       ___       ___       ___
```

We fill in the top row of dashes with 2 times the elements of the stream, and the bottom dashes with the elements plus 1. So far we know two elements, so we can fill in the first two dashes on each row:

```
1  2  _2_       _4_       ___       ___

           _2_       _3_       ___       ___
```

Now we know that the next two elements are 2 and 2, so we can fill in the remaining blanks:

```
1  2  _2_       _4_       _4_       _4_

           _2_       _3_       _3_       _3_
```

So the final result is

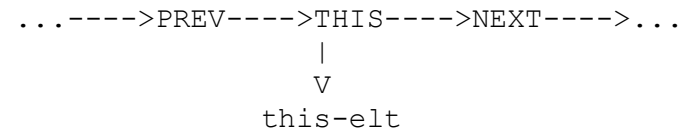
```
   1 2 2 2 4 3 4 3 4 3 ...
```

Scoring: -1 per wrong value.

4. Mutation programming

This problem is about deleting elements from a list, rather than reordering elements, so it has to use SET-CDR!, not SET-CAR!.

If the spine of a list looks like this:



and we want to remove this-elt from the list, we have to (set-cdr! prev next).

```
(define (josephus ring k)
  (if (eq? (cdr ring) ring); only one left
      ring
      (let ((prev ((repeated cdr (- k 2)) ring)))
      (set-cdr! prev (cddr prev))
      (josephus (cdr prev)))))
```

It's important that the base-case test uses EQ?, not EQUAL?, since we are dealing with a circular list, for which EQUAL? will never terminate. If the
list is its own cdr, then it is a one-element circular list, and we're finished. We also can't use LENGTH, which doesn't work for circular lists.

Otherwise, we want to delete the Kth pair, so we need to mutate the pair /before/ that one, the (K-1)th pair, which we find by CDRing the list K-2 times.

The SET-CDR! removes the pair (CDR PREV) from the spine of the list.

Then we make a recursive call, starting with the /new/ (CDR PREV), which used
to be (CDDR PREV) before the deletion.

Several people were curious about the name; you can read more about the problem at http://en.wikipedia.org/wiki/Josephus_problem .

Scoring:

7   correct
6   trivial error
4-5 has the idea
2-3 has an idea
0-1 other

Specific cases:

6 Takes (K-1) cdrs (deletes the pair one ahead).

5 Takes K cdrs (deletes the pair two ahead).
5 Uses EQUAL? or LENGTH in base case test.
5 Treats return value of set-cdr! as a pair.
5 Recurs on PREV instead of (CDR PREV).
4 Recurs on (CDDR PREV) instead of (CDR PREV).
3 SET-CDRs a spine pair to another spine pair.
3 Tries to use set! and almost works.
3 Doesn't change the list: (set-cdr! ring (cdr ring)).
2 Only eliminates one pair (one 'round' of elimination).
2 Uses SET-CAR! to overwrite deleted element with something else.
1 Makes new pairs.
0 No recursion.

5. Concurrency

(a) The question basically is, how many rounds of eating do we need to feed all the TAs? The answer is three rounds, or 15 minutes (since each round takes five minutes).

You know it has to be at least three rounds because three of the TAs require a fork, and there's only one fork. This shows that we can't feed everyone in
fewer than three rounds; to finish the analysis we still have to show that three rounds are enough, but that's easy: we just exhibit a solution in three
rounds.

The simplest such solution comes from noting that three of the TAs require only one utensil each, and they're all different utensils, so they can all eat in parallel:

Round 1: Evan (fork), Albert (spoon), Yaping (knife).
Round 2: Ramesh (fork and spoon).
Round 3: Jerry (fork and knife).

(Of course the three rounds can be in any order.) The other solutions work by pairing a two-utensil TA with the TA who uses the third utensil:

Round 1: Ramesh (fork and spoon), Yaping (knife).
Round 2: Jerry (fork and knife), Albert (spoon).
Round 3: Evan (fork).

or

Round 1: Ramesh (fork and spoon), Yaping (knife).
Round 2: Jerry (fork and knife).
Round 3: Evan (fork), Albert (spoon).

There are several other similar solutions, all taking three rounds or 15 minutes.

Scoring:

3 Correct.
2 15 minutes, but details not quite right.
1 15 minutes, no details.
1 More than 15 minutes, details to match.
0 Less than 15 minutes.

(b) No, no deadlock is possible. Deadlock happens only when two processes both need the same two resources, and ask for them in reverse order. In this problem, there are two TAs who need two resources (utensils), but not the /same/ two resources.

To make deadlock possible, we have to have two TAs who want the same utensils. For example, we could upgrade Yaping's meal to meatloaf, for which
she'd need a fork and knife, the same utensils Jerry needs for his steak. If Jerry asks for the knife first, and Yaping asks for the fork first, then a deadlock is possible.

Scoring:

2 No deadlock, correct scenario to allow deadlock.
1 No deadlock, incorrect or missing scenario.
0 Deadlock in original scenario.

6. Vector programming.

```
(define (diffs in)
  (define (iter out k)
    (if (< k 0)
     out
     (begin
      (vector-set! out k (- (vector-ref in (+ k 1))
                            (vector-ref in k)))
      (iter out (- k 1)))))
  (iter (make-vector (- (vector-length in) 1))
     (- (vector-length in) 2)))
```

As usual with vector programs, we have an iterative helper procedure ITER, which is invoked with the result of a call to MAKE-VECTOR for the result vector, plus an initial value for an index into that vector.

Suppose the input vector is of length 10. Then its elements are numbered from 0 to 9. The result vector will be of length 9, with elements numbered from 0 to 8. This is why we subtract 2 from the length of IN to get the starting value for K.

Element 0 of the result vector is the difference between elements 1 and 0 of

the argument vector. In general, element K of the result comes from elements
K and K+1 of the argument.

Apart from these details about indexing, this should have been a very easy vector program to write.

Although it goes against the spirit of the problem, we accepted some solutions
that used VECTOR-CONS from the lecture notes, since they technically return the correct answer at the end.

Scoring:

7   Correct.
6   Trivial error.
4-5 Has the idea.
2-3 Has an idea.
0-1 Other.

Specifically:

6 Off-by-one error.
6 Forgets to return the result vector
5 Too many or too few elements in result vector.
3 Modifies input vector.
0 Uses lists.

Group problem (environments):

```
> (define y 2)
```

Add binding Y=2 to global frame G.

```
> (define f
    (let ((g (lambda (x) (* x y))))
      (lambda (y) (g y))))
```

The LET expression expands to

```
    ((lambda (g) (lambda (y) (g y)))
     (lambda (x) (* x y)))
```

We start by evaluating the subexpressions of this procedure call, which are
both LAMBDA expressions, so they make procedures:

Procedure P1: params (g), body (lambda (y) ...), env G.
Procedure P2: params (x), body (* x y), env G.

Then we invoke P1 with P2 as argument. Invoking P1 makes a new environment:

Env E1: Binds G to P2, frame extends G.

With E1 as current env, evaluate the body of P1: (lambda (y) (g y)). This makes another procedure:

Procedure P3: params (y), body (g y), env E1.

This procedure is the return value from the LET, so we can finish the DEFINE:

Add binding F=P3 to global environment.

> (f (+ y 1))

Evaluate the subexpressions. The value of F is procedure P3; the value of
(+ Y 1) is 3. Now call P3 with actual argument value 3, creating a new environment:

Env E2: Binds Y to 3, extends E1.

With E2 as the current environment, evaluate the body of P3, namely (G Y). This is a procedure call, so we evaluate the subexpressions. In frame E1 we find that G is P2; in frame E2 we find that Y is 3. So we call P2 with argument 3, creating a new environment:

Env E3: Binds X to 3, extends G.

In that environment we evaluate the body of P2, which is (* X Y). X is 3 from
frame E3; Y is 2 from the global frame. (E3 doesn't extend E2! That would be
dynamic scope.) And * is the multiplication primitive procedure from the global frame. Thus we multiply 3 by 2, giving the final answer of 6.

Scoring:

4 if correct.

3 for two common errors: The first was to make P2's right bubble point to E1
instead of G, and therefore to make E3 extend E1 instead of G. This error comes from not understanding that the value expressions in a LET are evaluated
/before/ the LET's hidden-lambda procedure is called. We didn't take off more
for this because, in this example, it happens not to affect the final answer.

The other 3-point error, not quite as common, was to omit E3 altogether, but

with P2's right bubble correct.

The other errors were quite varied and hard to list coherently; the general scheme is 2 for has the idea, 1 for has an idea, 0 for other.

One expected wrong answer was to use dynamic scope, so E3 would extend E2 instead of extending G. Several groups indeed did that. Of those, most still got the correct 6 as the final result, whereas they should have gotten a final answer of 9 (since both X and Y are 3 in dynamic scope). I guess this means that many of you don't really believe that the environment diagram
you draw actually has any connection to the evaluation of expressions!

One fairly common low-scoring problem was to draw environments out of chronological order, e.g., frame E1, which is created during the definition of F, extends E2, which is created when F is invoked in the last expression. One reason the scoring was so difficult is that I just couldn't "read the minds" of the students who did this sort of thing -- I can't imagine what they were thinking!

Another quite bad mistake was to bind a variable to an expression (e.g., in E2, writing "G -> (lambda (x) (* x y))" or, even worse, "G -> (* x y)". Similarly, some people bound variables to other variables, e.g, an arrow from the letter X in E3 to the letter Y in E2. Variables are bound to values, not to expressions or to other variables!

If you don't like your grade, first check these solutions. If we graded your paper according to the standards shown here, and you think the standards were wrong, too bad -- we're not going to grade you differently from everyone else. If you think your paper was not graded according to these standards, bring it to Brian or your TA. We will regrade the entire exam carefully; we may find errors that we missed the first time around. :-)

If you believe our solutions are incorrect, we'll be happy to discuss it, but you're probably wrong!