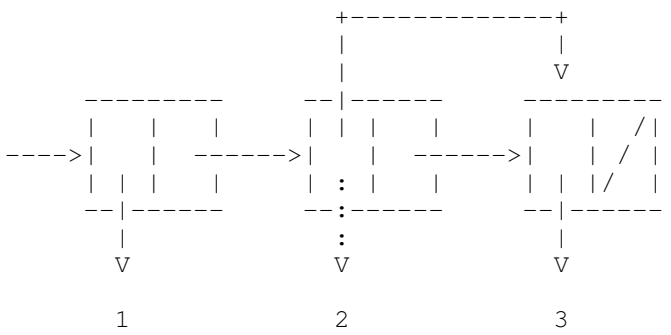
CS 61A Spring 2006 Midterm 3 solutions

1. Box and pointer.

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

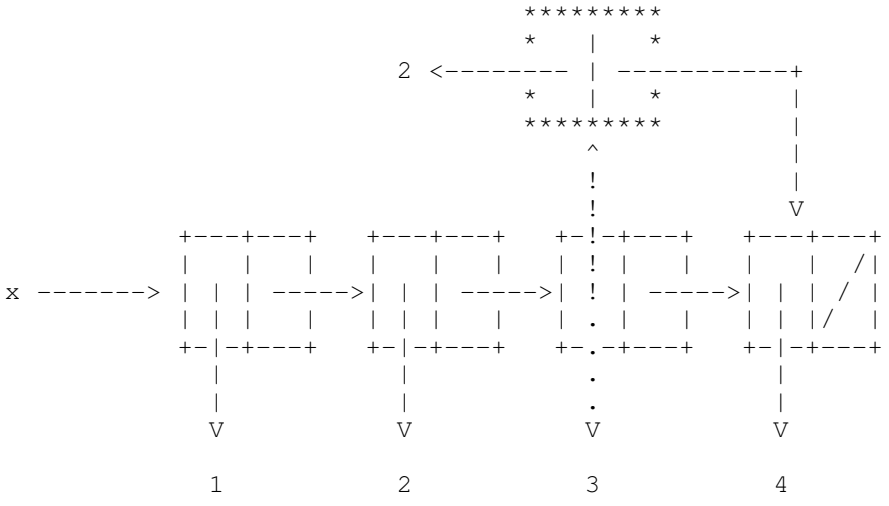
(1 (3) 3)



(Some people had correct diagrams but wanted to print (1 3) or (1 3 3). The backbone of this list still has three pairs, so the list still has three elements! And the car of the second pair is a pair, not the number 3.)

```
> (let ((x (list 1 2 3 4)))
    (set-car! (cddr x) (cons (cadr x) (cdddr x)))
    x)
```

(1 2 (2 4) 4)



[In this problem a new pair is created. In the diagram above, the new pair is the one bordered by asterisks (*). That's because this problem invokes a constructor (CONS) rather than only mutators (SET-CxR!). The car of the new pair points to a number, whereas its cdr points to another pair. A very common error was to make an additional new pair with car 4 and cdr the empty list. But there's only one CONS in this problem, and it only makes one pair.]

Scoring: One point for each diagram, one point for each print form.

2. Concurrency

The crucial point that many people missed is that a serializer only protects against another process that uses the same serializer! Serializers don't magically know what variables you want to protect; all they know is that each serializer should only allow itself to be used by one process at a time.

(a) BYE or BYEBYE

(b) BYE, BYEBYE, HIHI, HIBYE, or BYEHI

To get BYE, first process finishes, then second process runs.

To get BYEBYE, second process finishes, then first process runs.

To get HIHI, first process reads BAZ twice, then second process runs, then first process sets BAZ.

To get HIBYE or BYEHI, the first process reads BAZ once, then the second process runs, then the first process reads BAZ again and sets BAZ. (There are two possible results because Scheme is not guaranteed to evaluate argument subexpressions left-to-right!)

Scoring: Subtract 1/2 point for each missing answer, or each incorrect answer added; round down to an integer.

3. OOP short answer.

The central point of this question is that in the OOP paradigm, objects are used for everything, and the structure of the object classes reflects the structure of the real-world situation being simulated.

(a) parenthood

The class VANILLA has the class SCOOP as its parent, because a vanilla scoop is a particular kind of scoop.

Is a cone a special kind of scoop? No.

Is a scoop a special kind of cone? No.

So the correct answer is "Neither." Most people got this; the most common wrong answer was to say that SCOOP should have CONE as its parent, probably because a scoop can be viewed as *part of* a cone. But the parent/child relationship isn't "a part of"; it's "a kind of."

(b) flavors method

```
(method (flavors)
  (map (LAMBDA (S) (ASK S 'FLAVOR)) scoops))
```

You *could* write the CONE class so that its instance variable SCOOPS would be a list of flavors (i.e., names of flavors) instead of a list of scoop objects. But that wouldn't be following the OOP paradigm. (Not to mention that using the name SCOOPS for a list of flavors would be really confusing!) So if we want to know the flavors in a cone, we have to ask each scoop for its flavor.

A few people said something like (USUAL 'FLAVOR) instead of (ASK S 'FLAVOR), presumably because FLAVOR is a method of the SCOOP class, rather than of the VANILLA or CHOCOLATE class. But even if this could work, the whole idea of inheritance is that you can send the child class (e.g., VANILLA) the same messages you can send to the parent class (SCOOP). You don't have to know whether VANILLA handles the FLAVOR message itself or delegates the message to its parent. And in any case it wouldn't work; USUAL can only be used inside a DEFINE-CLASS, because otherwise it doesn't know what object you're talking about!

(c) adding a scoop

Given a particular ice cream cone, we want to add a particular scoop of vanilla ice cream to it -- not the VANILLA class, and not the word VANILLA.

So the right answer is the third one listed:

```
        (ask my-cone 'add-scoop (instantiate vanilla))
```

The argument to INSTANTIATE is a class, not the name of a class.

Scoring: 1 point for each of (a) and (c); 2 points for (b).
No partial credit except that in part (b) leaving
out the quotation mark before FLAVOR got one point.

4. Local state

The key to this problem is understanding that it's an OOP problem in ordinary Scheme notation. The MAKE-xxx procedures represent classes; the FOOxxx ones represent instances.

All OOP-in-Scheme procedures have the following structure:

```
(define CLASS
  (let (CLASS-VARS)
    (lambda (INSTANTIATION-VARS)        ; this is the class procedure
      (let (INSTANCE-VARS)
        (lambda (message)               ; this is the instance procedure
          ...)))))
```

In this problem the structure is simplified in that there are no instantiation variables and no message -- the instances only know how to do one thing, so they're invoked without an argument.

In MAKE-FOO1, both A and B are class variables. So the sequence of events is

```
        (FOO3-1):  A 1 -> 10,      B 1 -> 11.
        (FOO3-1):  A 10 -> 100,    B 11 -> 111.
        (FOO3-2):  A 100 -> 1000,  B 111 -> 1111.
```

The notation means that in the first call to FOO1-1, A changes from 1 to 10, B changes from 1 to 11. The first two calls will be the same in all four cases, because they're calls to the same instance, so it doesn't matter whether A and B are class or instance variables. But in the last call, to a different instance, the instance variable(s) start from 1 rather than from the value in the previous line. The return value from the last call is the value of B, which is 1111 in this case.

[I actually solved this problem by drawing myself little pictures in which a class variable has one long dash, and an instance variable has two short dashes, one for each instance. So for MAKE-FOO1 the picture is

```
        A _____________

        B _____________
```

For MAKE-FOO2 it's

```
        A _____________

        B _____  _____
```

And so on, Then I put 1 on each dash, and then changed the value on the appropriate dash for each SET!.]

In MAKE-FOO2, A is a class variable; B is an instance variable. So:

```
        (FOO1-1):  A 1 -> 10,      B 1 -> 11.
        (FOO1-1):  A 10 -> 100,    B 11 -> 111.
        (FOO1-2):  A 100 -> 1000,  B 1 -> 1001.
```

In MAKE-FOO3, both are instance variables. So:

```
        (FOO4-1):  A 1 -> 10,      B 1 -> 11.
        (FOO4-1):  A 10 -> 100,    B 11 -> 111.
        (FOO4-2):  A 1 -> 10,      B 1 -> 11.
```

In MAKE-FOO4, A is an instance variable; B is a class variable. So:

```
        (FOO2-1):  A 1 -> 10,      B 1 -> 11.
        (FOO2-1):  A 10 -> 100,    B 11 -> 111.
        (FOO2-2):  A 1 -> 10,      B 111 -> 121.
```

Score: One point each, no part credit. Exception: The answers 11, 121, 1111, 1001 in that order, which come from a systematic reversal of class vs. instance, and would by the standard formula get 0 points, instead gets 2 points (half credit).

5. OOP program

(a) button object:

```
(define-class (button)
  (instance-vars (freq 0))
  (method (set-freq! value)
    (set! freq value)))
```

Note that there is no need to write a FREQ method, since the OOP system automatically provides methods to read the object's variables.

(b) radio object:

```
(define-class (radio)
  (instance-vars (freq 90.7)
                 (buttons (list (instantiate button)
                                (instantiate button)
                                (instantiate button)
                                (instantiate button)
                                (instantiate button)
                                (instantiate button))))
  (method (set-button! num)
    (ask (list-ref buttons num) 'set-freq! freq))
  (method (push num)
    (set! freq (ask (list-ref buttons num) 'freq)))
  (method (up)
    (set! freq (+ freq 0.2)))
  (method (down)
    (set! freq (- freq 0.2))))
```

The most common serious error was to make the variable BUTTONS a list of names of buttons, such as (0 1 2 3 4 5) or (BUT0 BUT1 BUT2 ...). First of all, there is no need for the buttons to have names at all. Second, there is no way to look up a name and then use it as the first argument to SET!, as in

```
        (set! (list-ref buttons num) freq)   ;; WRONG!
```

SET! is a special form whose first argument *is not evaluated* and must be a symbol -- the actual name of a variable, not an expression whose value would be the name. Such solutions got at most one point.

Another fairly common error was to define six global variables containing buttons, so that all radios share the same buttons! These solutions got at most two points.

Some people tried to avoid that last error by using DEFINE as a clause within a DEFINE-CLASS. Sorry, but that doesn't work; you can only use the clauses that DEFINE-CLASS understands: METHOD, INSTANCE-VARS, CLASS-VARS, and so on. In this case, INSTANCE-VARS was what you wanted. If otherwise correct, these solutions got four points.

Some people tried to make RADIO the parent of BUTTON, or even sometimes the other way around. This doesn't make any sense. A radio is not a special kind of button, and a button is not a special kind of radio. These solutions got at most two points.

Finally, several people gave the RADIO class six instantiation

variables to hold the buttons, so that when creating a radio you'd say something like

```
        (instantiate radio (instantiate button)
                           (instantiate button)
                           (instantiate button)
                           ...)
```

This works, but isn't a very realistic simulation. When you buy a radio, you don't also buy six buttons and install them; the radio just comes with six buttons. We gave these solutions three points.

6. List mutation

(a) Create mutators for the binary tree ADT.

```
(define (set-datum! node value)
  (set-car! node value))

(define (set-left-branch! node value)
  (set-car! (cdr node) value))

(define (set-right-branch! node value)
  (set-cdr! (cdr node) value))
```

An alternative for SET-DATUM! is

```
      (define set-datum! set-car!)
```

but otherwise these have to be as seen here. I thought this part of the problem would be trivial for everyone; I was surprised to see so many wrong answers. A common one was

```
(define (set-right-branch! node value)      ;; wrong!
  (set-cdr! (CDDR node) value))
```

but people were also confused about what three procedures you were asked to write. This is an ADT with three fields; to make it mutable, you need mutators for those three fields.

Another very common error was to leave out arguments. A mutator requires two arguments: the structure to modify, and the new value to insert. You can't say things like

```
(define (set-left-branch! node)      ;; wrong
  (set-car! (cdr node)))
```

The call to SET-CAR! with only one argument will fail.

Scoring:

2 correct
1 some small detail wrong
0 misguided

(b) rotation

As in all mutation problems, the easiest solution is to remember every value you expect to need in a LET surrounding the actual mutation:

```
(define (rotate! node)
  (let ((a (left-branch node))
        (b (datum node))
        (c (left-branch (right-branch node)))
        (d (datum (right-branch node)))
        (e (right-branch (right-branch node)))
        (rb (right-branch node)))
    (set-left-branch! rb a)
    (set-datum! rb b)
    (set-right-branch! rb c)
    (set-left-branch! node rb)
```

```
    (set-datum! node d)
    (set-right-branch! node e)))
```

It might not have been obvious that you'd need the variable RB in the LET above, but if not, you could have inserted it after discovering that it's needed for the left branch of the modified node.

In fact not all six of the LET variables are needed; by being very clever about the order of the six mutation instructions in the body of the LET you could work out the minimum number of saved values necessary, but why bother? The solution above is straightforward and clearly works correctly no matter what order you use for the six mutations.

The problem statement said to respect the data abstraction. Notice that in the solution above there are no calls to CAR, CDR, SET-CAR!, or SET-CDR!. Some people violated the abstraction in otherwise correct solutions; in other cases, though, calls to CAR etc. indicated that the person was thinking about pairs instead of thinking about trees, trying to make left-branch mean car and right-branch mean cdr.

The problem statement said not to allocate new pairs. Any call to CONS, LIST, APPEND, or MAKE-TREE violated that requirement.

The problem statement said that the pair at the head of the given tree must still be at the head of the result, because this tree may be a subtree of a higher-up tree. In the example shown, the argument to ROTATE! would be the subtree whose root datum is 15, but that subtree still has to be the right branch of the overall tree (whose root datum is 8). If the entire tree is named THE-TREE, then the call would be

```
        (rotate! (right-branch the-tree))
```

That's why the roles of the two pairs in the node have to be exchanged, so the two DATUMs are exchanged, etc.

On the other hand, some people assumed that the assignment was to rotate the right branch of the given tree, just because that's how the example was set up. In other words, they expected to be called with

```
        (rotate! the-tree)  ;; no
```

and still rotate the 15 node, not the 8 node.

Other errors were based on confusions about how Scheme works, rather than on confusions about what the problem was asking. One such confusion was to have nested calls to mutators:

```
        (set-left-branch! node
                          (set-right-branch! (left-branch node)
                                             (right-branch node)))   ;; wrong
```

or something like that. The mutators have unspecified return values; you shouldn't rely on any particular expectation. (In most such cases, we couldn't tell what return values would have helped, anyway.)

Another, very serious confusion was to think that by saying something like

```
        (let ((new-node node))
          ...)
```

you would have two copies of the original node, one of which you could then mutate without worrying about losing information. If this were true, you wouldn't have been allowed to use LET, since the problem statement says not to allocate any new pairs. But in fact, all this LET does is to make NEW-NODE a new name for the very same node as NODE. Mutating one of the names also changes the value of the other, since they are the same pairs.

Finally, a few people tried to avoid the need for the LET -- that is, avoid the problem of losing track of pieces of the tree -- by using PARALLEL-EXECUTE in an attempt to make all the modifications happen at once. But there is no guarantee that the processes started by PARALLEL-EXECUTE

```
are run in precise lock-step, so that all the processes read the (old) values
and then all the processes make their modifications.

Scoring:

4  OK
3  trivial mistake (one wrong mutation)
2  data abstraction violation, or doesn't save some needed pointers
1  makes new pairs, or really confused about desired mutation
0  even worse


7.  Vector programming

(define (ssort! vec)
  (define (help start)
    (if (= start (length vec))
        vec
        (let ((smallest (subvec-min-start vec start)))
          (let ((temp (vector-ref vec smallest)))
            (vector-set! vec smallest (vector-ref vec start))
            (vector-set! vec start temp)
            (help (+ start 1))))))
  (help 0))

The key point to understand is that you need to walk through the vector,
bringing the smallest element out to position 0, then the next-smallest to
position 1, and so on, until you run out of elements.  Thus we have a helper
procedure with an argument, START, whose value is the position that we're
up to in the vector; it starts at 0 and is increased by 1 on each recursive
invocation.

The next important point is that you can't exchange two elements without
using a temporary variable to remember one of them, hence the LET that
creates the variable TEMP.  There are two nested LETs because the value of
SMALLEST is needed to find the value of TEMP in this solution; I could have
simplified things a little by remembering the other value in the swap instead:

(define (ssort! vec)
  (define (help start)
    (if (= start (length vec))
        vec
        (let ((smallest (subvec-min-start vec start))
              (temp (vector-ref vec start)))
          (vector-set! vec start (vector-ref vec smallest))
          (vector-set! vec smallest temp)
          (help (+ start 1))))))
  (help 0))

Scoring:

6  correct
5  trivial error (e.g. base case off by one)
3  right structure, gets swap wrong
2  serious algorithm confusion
0  allocates new vector, uses lists, or incoherent


Group question: environment diagram.

The first DEFINE is an abbreviation for

        (define bar (lambda (x) (let ...)))

So the implicit LAMBDA expression creates a procedure

        P1: param=x, body=(let ...), env=Global

It's in the global environment because that's always the current
environment for expressions typed at the Scheme prompt, as opposed to
expressions inside a procedure body.  Then in the global environment
we bind BAR to this:
```

```
        G: BAR -> P1
```

That's all that happens when Scheme evaluates the first expression; the stuff in the procedure body is not evaluated at this time.

The second expression is (define foo (bar 4)). First we have to evaluate the procedure call (bar 4). We do this by creating a new environment in which BAR's parameter (x) is bound to the argument value (4), extending the procedure's defining environment:

```
        E1: X -> 4, extends G
```

Using E1 as the current environment, we evaluate the procedure body, which is the expression

```
        (let ((z (lambda (b) ...)) (c x))
          (lambda (x) ...))
```

LET abbreviates a lambda and a procedure call, as follows:

```
        ( (lambda (z c) (lambda (x) ...))  (lambda (b) ...) x )
          -------------------------------  ---------------- -
              implicit procedure               arg for z    c
```

To evaluate a procedure call, we first evaluate all the subexpressions. Two of them are LAMBDA expressions, which create procedures:

```
        P2: params=z,c,  body=(lambda (x) ...),  env=E1
        P3: param=b,     body=(* x b),           env=E1
```

These procedures have E1 as their defining environment because that's the current environment right now.

The third expression, X, is evaluated by looking for the symbol X in E1. We find it; its value is 4.

Now that we've evaluated the subexpressions, we call procedure P2 with arguments P3 and 4. We do this by creating a new environment binding P2's parameters to these arguments and extending P2's defining environment:

```
        E2: Z -> P3,  C -> 4,  extends E1
```

Note that the value of C is 4, not X! Parameters are bound to the actual argument *values*, not to the expressions that give rise to the values. Similarly, the value of Z is P3, not (* X B) or (LAMBDA (B) (* X B)).

With E2 as the current environment, we now evaluate the body of P2. That body is a lambda expression, so it creates a procedure:

```
        P4: param=x,  body=(set! ...)...,  env=E2
```

P4 is the value of the lambda expression, so it's the value returned by the call to P2, so it's the value returned by the LET, so it's the value returned by (bar 4), and so back in the global environment we bind FOO to this value:

```
        G:  BAR -> P1,  FOO -> P4
```

That's it for the second expression. We invoked P2 because it was created by a LET, which both creates and invokes a procedure, but nothing says to invoke P3 or P4 so far.

Now for the third expression, (FOO 3). This is a procedure call; we evaluate the subexpressions (looking up FOO in the global environment and noticing that 3 is self-evaluating). Then we invoke P4, which is the value of FOO, with 3 as its argument. This creates a new environment:

```
        E3: X -> 3,  extends E2
```

With E3 as the current environment, we evaluate the body of P4. This body contains two expressions. The first is

```
        (set! c (z (* c x)))
```

To evaluate this, we must first evaluate (z (* c x)), which is a procedure call. We evaluate the subexpression Z by looking it up in E3, the current environment. We don't find it directly in E3, but E3 extends E2, where Z is bound to P3. The subexpression (* c x) is a procedure call. We evaluate the subexpressions, finding * in G (it's bound to the primitive multiply procedure), C in E2 (where it's bound to 4), and X in E3 (bound to 3). Since the multiply procedure is primitive, we apply it by magic, getting the answer 12 without creating a new environment. This value 12 is the argument to Z (which is actually P3). So we create a new environment for the call to P3:

```
E4: B -> 12,  extends E1
```

The body of P3 is (* x b). We look up these three symbols and find * in the global environment (bound to the primitive multiplication procedure), X in E1 (not in E3, which isn't part of the current environment!) (bound in E1 to 4), and B in E4, bound to 12. By magic we multiply 4 by 12 and get the answer 48.

We did that to evaluate the second argument to (set! c ...). Now that we know the argument value, we can find a binding for C in what's now the current environment, the one in which we saw the SET! expression, namely E3. There's no binding for C in E3's first frame, but we find one in E2, which E3 extends. There we change C's binding from 4 to 48.

The second and last expression in the body of P4 is just C, so we return the value 48 that we just put there.

To summarize, the diagram has five frames (including G) and four procedures:

```
G:  BAR -> P1,  FOO -> P4
E1: X -> 4,                extends G
E2: Z -> P3,  C -> 4,      extends E1
E3: X -> 3,                extends E2
E4: B -> 12,               extends E1

P1: param=x,     body=(let ...),         env=Global
P2: params=z,c,  body=(lambda (x) ...),  env=E1
P3: param=b,     body=(* x b),           env=E1
P4: param=x,     body=(set! ...)...,     env=E2
```

Common errors:

The most common was having P3's right bubble point to E2 instead of E1, probably from thinking that the lambda expression was within the scope of the LET. But the expressions that provide values for the LET variables must be computed before the LET variables are bound! This generally led to having E4 extend E2 instead of E1.

Several people left out E4 altogether.

One mistake I couldn't understand was that a lot of people had FOO bound to E2 instead of P4. An environment can't be the value of a variable! (Environments are not first class in Scheme.)

As mentioned above, too many people bound the E2 variables to expressions [C -> X, Z -> (lambda (b) (* x b))] rather than to the values of those expressions. (In effect you are inventing normal order evaluation, which isn't what Scheme does.)

Some people bound B to 16 instead of 12, thinking that X was 4 (from E1) instead of the correct 3 (from E3). This mistake wasn't necessarily accompanied by any error in the arrangement of E1 and E3 themselves. These people generally got 64 as the answer to (foo 3).

Some people had E1 and E2 backwards (that is, E1 extending E2 which extends G). I can't imagine a mental model that would produce such a diagram, since the value of C in E2 is based on the value of X in E1. Perhaps these groups wanted to evaluate the LET as part of the process of defining BAR (rather than when BAR is invoked), but in that case it's not clear to me where the value of C came from.

A few people had E4 extending E3 (dynamic scope), but not as many as I would have predicted -- it was more common to omit E4 altogether.

Some people added empty frames to the diagram. An empty frame is created by invoking a procedure of no arguments [(lambda () ...)], but there aren't any of those in this problem.

Some people added bindings for Z and C to E1 instead of creating a separate E2. That would be appropriate for internal definitions:

```
(define (bar x)
  (define (z b) (* x b))
  (define c x)
  (lambda (x) ...))
```

But LET creates a new frame because it abbreviates a LAMBDA and an invocation of the resulting procedure.

Scoring: 5 if correct. 4 for a single error (which could involve two pointers if a procedure pointing to the wrong environment led to another environment extending that wrong environment. 2 for multiple errors in an otherwise reasonable environment diagram. 0 for multiple missing frames or seriously incoherent results such as variables whose value is an environment.

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!