# CS 61A         Midterm 3 solutions            Spring 2009

1.  Box and pointer.

```
(define x (list 1 2))
(define y (list 3 4))
(define z (append x y))
(set-car! (cdr x) 5)
(set-car! y 6)
z
```

Result: (1 2 6 4)

```
              5              6
              ^              ^
              |              |
X ----> XX--->X/    Y --+--> XX--->X/
        |     :        /     :     |
        V     :       /      :     V
        1     2   ----       3     4
        ^     ^  /
        |     | /
Z ----> XX--->XX
```

Mostly this is a question about understanding how APPEND works,
namely, /not/ by mutation (that's APPEND!), but by /copying/ the
spine pairs of all but the last argument -- in this case,
copying
X but not copying Y.

```
(define x (list 1 2))
(define y (list 3 4))
(define z (cons x y))
(set-car! (cdr x) 5)
(set-car! y 6)
z
```

Result: ((1 5) 6 4)

```
Z ----> XX-------------------+
        |                    |
        |                    |
        V                    V
X ----> XX--->X/    Y -----> XX--->X/
        |     :\             :\    |
        V     : \            : \   V
        1     2 V            3 V   4
                5              6
```

This box-and-pointer diagram should have been really easy; the
only
slight difficulty is knowing not to say ((1 5) . (6 4)) as the

printed representation.

```
(define x (list 1 2))
(define y (list 3 4))
(define z (cons x y))
(set! x 5)
(set! y 6)
z
```

Result: ((1 2) 3 4)

```
Z ----> XX-------------------+
        |                    |
        |                    |
        V                    V
X ....> XX--->X/    Y .....> XX--->X/
 \      |     |      \       |     |
  \     V     V       \      V     V
   V    1     2        V     3     4
   5                   6
```

Here the crucial understanding is that changing the value of a /variable/ doesn't change the contents of any pairs, and that Z points to what the values of X and Y were at the time of the CONS, not to the names X and Y.

Scoring:  One point per diagram, one point per printed representation.

2.  OOP to Scheme.

```
(define make-foo
  (let ((a 3))         ; This LET happens when we define the class
    (lambda (b)        ; This is MAKE-FOO, which makes an instance
      (let ((c 4))     ; This LET happens when we make an instance
        (define (d e)  ; This is a method
          (+ a e))
        (define (f g)  ; This is the instance's dispatch procedure
          (if (eq? g 'h)
              d
              (error "huh?")))
        f))))          ; MAKE-FOO returns the dispatch procedure!
```

| | |
|---|---|
| Class variable: | A |
| Instance variable: | C |
| Instantiation variable: | B |
| Message: | G and H |
| Method: | D |
| Dispatch procedure: | F |
| Method argument: | E |
| Instance: | F |

G is a variable whose value will be the message that the user sends; H is a

literal (quoted) symbol, so it's a particular message that the object
accepts.
So "message" was the question with two answers.

F is the dispatch procedure; it takes a message as argument and returns
a
method.  But it's also the instance itself, since an instance is
represented
by its dispatch procedure.  So "F" was the answer with two questions.

We accepted B as an answer to "instance variable" on the theory that an
instantiation variable is a kind of instance variable.  But you should
understand the difference: An instance variable gets its initial value
from
the class definition; in this case, every instance of the FOO class has
a
variable C whose initial value is 4.  An instantiation variable gets its
initial value from an argument to INSTANTIATE (in our OOP language) or
to
the class instantiation procedure MAKE-FOO (in the below-the-line
implementation).

Scoring:  Start with 6, subtract 2/3 point per missing or incorrect
answer,
round.  (Don't ask "up or down"; we /rounded/.)

3.  Mutation.

(a) The answer to the question in the hint is a little complicated.
From
the point of view of the big list, you want to change individual
elements;
you want to say something like

```
        (set-car! ls (cons (caar ls) (cadar ls))      ; wrong
```

for each spine pair.  But you can't do that, because you can't use CONS
in
this problem.  Instead, you have to change the structure of the small
two-element lists; you'll change the first (of two) spine pair of each
sublist into a dotted pair of the two elements.  So this problem calls
for
SET-CDR!, which rearranges the structure, rather than SET-CAR!, which
just
replaces elements.

```
(define (twos->assoc ls)
  (if (null? ls)
      '()
      (begin (set-cdr! (car ls) (cadar ls))
             (twos->assoc (cdr ls))
             ls)))
```

Don't forget that the procedure is supposed to return the list as its
value!

If you find things like (CADAR LS) confusing, use data abstraction!
Define

```
        (define second-element cadr)
        (define first-element car)
```

and then say (SECOND-ELEMENT (FIRST-ELEMENT LS)).

Instead of an explicit recursion, you can also solve this problem using
the
higher order procedure FOR-EACH:

```
(define (twos->assoc ls)
  (for-each (lambda (element)
              (set-cdr! element (cadr element)))
            ls)
  ls)
```

But it would be an error to use MAP instead of FOR-EACH, since MAP
allocates
a new list (that is, the spine pairs of the result) instead of changing
the
input list.

Scoring:

5  correct.
4  trivial error (e.g., base case, or doesn't return LS).
3  has the idea (e.g., args to set-cdr! a little wrong).
2  uses MAP instead of FOR-EACH; otherwise perfect.
1  has an idea (e.g., uses set-car!).
0  allocates pairs instead of mutation, or incoherent.

(b) In this problem it's quite clear that we are rearranging elements,
not
just replacing values, since in the example a three-element list
becomes a
six-element list.

What the underlined, boldface sentence in the problem is telling you is
that
the list you return doesn't have to start with the same pair as the
original
input list.  It's clearly easiest if you just return (CAR LS) after
doing
some mutation, because that sublist already has the right first two
elements.

```
(define (twos->flat ls)
  (if (null? ls)
      '()
      (begin (set-cdr! (cdar ls) (cadr ls))
             (twos->flat (cdr ls))
             (car ls))))
```

It's tempting to say (twos->flat (cddar ls)) as the recursive call,
since

the pair (cdr ls) itself will not be part of the returned list value,
but
the domain of TWOS->FLAT is lists of two-element lists, and (cddar ls)
isn't one of those.  It's always helpful to think carefully about the
domain and range of every procedure /before/ you start writing it!

An alternative to the SET-CDR! expression would be

```
        (append! (car ls) (cadr ls))
```

which does exactly the same thing.  (But it has to be APPEND!, not
APPEND,
because the latter allocates new pairs.)

This one can't be done with FOR-EACH, because you need access to the
next element of the big list as well as the current element.

Scoring:  Same as part (a), but no 2-point solutions (because no FOR-
EACH
solution possible).

4.  Vectors.

```
(define (foo vec)
  (define (help result index sum len)
    (if (= index len)
        (begin (vector-set! result index sum)
               result)
        (begin (vector-set! result index (vector-ref vec index))
               (help result
                     (+ index 1)
                     (+ sum (vector-ref vec index))
                     len))))
  (help (make-vector (+ (vector-length vec) 1))
        0
        0
        (vector-length vec)))
```

The result vector has one more element than the argument vector, because
of the (MAKE-VECTOR (+ (VECTOR-LENGTH VEC) 1)) in the initial call to
HELP.
The first (VECTOR-LENGTH VEC) elements of the result are the same as the
corresponding elements of the argument, because of the call

```
        (VECTOR-SET! RESULT INDEX (VECTOR-REF VEC INDEX))
```

in which the same value INDEX is used to index both vectors.  The last
element, in the base case, is set to SUM, which is the sum of all the
other
elements.  (We didn't even give it a different name to confuse you!)  So
the result is

```
#(5 6 7 8 26)
```

Scoring:  2 points for being five elements long; 1 point for the correct
values in the first four elements, 1 point for the correct value in the
last
element.

5.  Streams.

The hint tells us to compute STREAM-AVERAGE in terms of STREAM-SUMS, which
is easy:

```
(define (stream-average s n)
  (stream-map (lambda (element) (/ element n))
              (stream-sums s n)))
```

This procedure works for both finite and infinite streams; it's the job of
STREAM-MAP to worry about the distinction.

Now to write STREAM-SUMS.  It helps to realize that we need a helper procedure
that will add up the first N elements of the stream:

```
(define (stream-sums s n)      ; first version
  (define (add-head s n)
    (if (= n 0)
        0
        (+ (stream-car s)
           (add-head (stream-cdr s) (- n 1)))))
  (if (stream-null? s) ;; see below
      the-empty-stream
      (cons-stream (add-head s n) (stream-sums (stream-cdr s) n))))
```

This version doesn't really do the right thing for finite streams; for the
example stream shown, it'd return (321 4320 54300 54000 50000) instead of
the desired (321 4320 54300).  But the underlined boldface sentence says
you don't have to handle finite streams, so we'd accept this -- and we'd
even accept this with the base case test for STREAM-SUMS left out:

```
(define (stream-sums s n)      ; second version
  (define (add-head s n)
    (if (= n 0)
        0
        (+ (stream-car s)
           (add-head (stream-cdr s) (- n 1)))))
  (cons-stream (add-head s n) (stream-sums (stream-cdr s) n)))
```

It's tempting, but wrong, to try to do the work of ADD-HEAD in STREAM-SUMS
itself:

```
(define (stream-sums s n)      ; WRONG WRONG WRONG
  (+ (stream-car s)
     (stream-sums (stream-cdr s) (- n 1))))
```

(Or the above with a base case added, still wrong.)  The range of STREAM-SUMS
is streams; the range of ADD-HEAD is numbers.  On the other hand, it's
possible to dispense with STREAM-SUMS altogether and just use ADD-HEAD:

```
(define (stream-average s n)
  (cons-stream (/ (add-head s n) n)
               (stream-average (stream-cdr s) n)))

(define (add-head s n)
  (if (= n 0)
      0
      (+ (stream-car s)
         (add-head (stream-cdr s) (- n 1)))))
```

Some people who used this approach used the name STREAM-SUMS for what I've
been calling ADD-HEAD.  That's a possible understanding of the hint, but it's
a stretch; "stream-sumS" is a bad name for a procedure that computes only one
sum!  Clearly the name suggests that the procedure should return a stream, not
a number.  Call it STREAM-SUM if you like, but not STREAM-SUMS.

Another, perhaps more "streamly" approach, uses higher order functions to
do all the adding of all the elements "at once":

```
(define (stream-sums s n)      ; coolest version
  (cond ((stream-null? s) the-empty-stream)
        ((= n 1) s)
        (else (stream-map +
                          (stream-sums (stream-cdr s) (- n 1))
                          s))))
```

The call to STREAM-MAP effectively replaces the ADD-HEAD mechanism, but it
returns a stream of sums, not just one sum.

This version requires understanding a subtlety: The arguments to STREAM-MAP
have to be in this order, because STREAM-MAP's base case test is on the first
argument stream (which is the second argument, after the function).  So for
finite streams, the /smaller/ stream has to come first.

The first COND clause, testing (STREAM-NULL? S), is an error check; the
argument stream really should have at least N elements!

Scoring:  There were a lot of ways to go wrong, but we used the following
general scale:

```
8  correct.
7  trivial mistake.
5  has the idea.
2  has an idea.
0  other.
```

Having the idea required, at least, having the domains and ranges of your procedures consistent, and making an attempt to add N elements, even if not quite right.  Some common 5-point solutions were forgetting to divide by N, or assuming that N=3.  (Using the N argument for getting the sums but then dividing by 3 for the average got 7 points.)

Group problem (environment diagram):

```
(define x 10)
(define y 12)
```

These make bindings in the global environment; nobody had trouble with those expressions.

```
(define (bar) (lambda (y) (+ x y))
```

Note the parentheses around BAR!  This expression defines BAR as a procedure that takes no arguments, whose body is a lambda expression.  Some groups ignored the parentheses around BAR and treated this as if it were the same as (define (bar y) (+ x y)).  That was a misreading of the problem, but much worse were the groups that made a binding for the name "(BAR)" -- treating the parentheses as if they were letters of the alphabet!  Only symbols can be variable names, not lists.

```
(define foo
  (lambda (x)
    (let ((y (* 2 x))
          (val (bar)))
      (+ (val x) 4))))
```

This is equivalent to (define (foo x) (let ...)); it creates a procedure with parameter X and binds the name FOO to that procedure.  The procedure body (i.e., the LET expression) is not evaluated at this time.

```
(foo 1)
```

This is a procedure call, so we evaluate the subexpressions; we look up FOO in the global environment, finding the procedure we just made, and 1 is self-evaluating.  The second step in a procedure call is to make a new frame in which the parameters (just X in this case) are bound to the actual argument

values (1).  Call this frame E1.  Thirdly, we extend the environment in which
FOO was created (the global environment) with E1.  Finally, we evaluate the
body (the LET expression) with E1 as the current environment.

A LET expression abbreviates a LAMBDA and a procedure call:

```
        ((lambda (y val) (+ (val x) 4))
         (* 2 x)
         (bar))
```

The first step is to evaluate all three of these subexpressions!  This
happens /before/ we can make a new frame, bind Y and VAL, etc.

The first subexpression is a LAMBDA expression, which creates a procedure
with parameters Y and VAL, whose right bubble points to the current
environment, namely E1.

The second subexpression is a procedure call: we look up "*", finding it
in the global environment; 2 is self-evaluating, and X is bound in E1, so
the value of X is 1 in this expression, not 10.  Therefore (* 2 X)
returns 2.

The third subexpression is a call to procedure BAR.  Too many groups made
the variable VAL point to procedure BAR itself, as if the expression were

```
        (let ((y (* 2 x))
              (VAL BAR))       ; Note the difference: BAR instead of
(BAR)
          (+ ...))
```

BAR doesn't take any arguments, so the only subexpression we have to evaluate
is the symbol BAR itself; the procedure bound to BAR was created in the global
environment.  So we now make a frame in which the formal parameters (of which
there aren't any!) are bound to the argument values (ditto) -- an empty frame.
Call it E2.  Use it to extend the global environment (not E1!), because that's
where BAR's right bubble points.

With E2 as the current environment, we evaluate the body of BAR, namely

```
        (LAMBDA (Y) (+ X Y))
```

This creates a procedure whose right bubble points to E2.  That procedure
is returned as the value of the expression (BAR), so in a moment it will be
bound to the name VAL.

But first we have to make a new frame to hold the bindings of Y (to 2) and

VAL (to the procedure we just made).  Call this frame E3.  Put the
bindings
in it, and use it to extend E1.  Why E1?  Because that's where the right
bubble of the LET procedure, the one with parameters Y and VAL, points.

By the way, hardly anyone got E2 and E3 named in the right order; even
most of
the groups that got full credit thought E2 should be the Y/VAL frame
and E3
the empty one.  But the names of the frames don't affect the result, as
long
as the frames extend the right environments.

With E3 as the current environment, we can now evaluate the LET body:

```
        (+ (VAL X) 4)
```

This is a procedure call, to the global addition procedure.  Before we
can
call +, we have to evaluate the argument subexpression (VAL X).  A few
groups
wanted to make a frame for the call to +, with "(VAL X)" as a formal
parameter!  But + is a primitive procedure, and so it has no formal
parameters and no frame created for it.

(VAL X) is also a procedure call.  We now have three calls in progress:
the
original call to FOO, the call to the unnamed LET procedure, and this
new
call to VAL.  (The call to BAR has finished.)  We find bindings for VAL
(in E3) and X (in E1, the next frame in the environment, where it's
bound
to 1).  So we make a new frame, E4, in which VAL's formal parameter Y is
bound to 1, the actual argument value.  (We do /not/ bind Y to the
symbol X!
That would be normal order evaluation, not applicative order.)

Frame E4 is used to extend the environment in which VAL was created,
namely
the empty frame E2 (which in turn extends the global environment).
With E4 as
the current environment, we evaluate the body of VAL, which is (+ X Y).
We
find + in the global environment, X in the global environment (where
its value
is 10), and Y in frame E4 (where its value is 1).  (Notice that the
value of X
used here is different from the value of X in the expression (VAL X),
which
was seen in a different environment!)  So the value of (+ X Y) is 10+1
or 11.

That's the value returned by the call (VAL X), so we can finish
evaluating
(+ (VAL X) 4), whose value is 15.  That's the value returned by the
LET, so
it's the value returned by (FOO 1).

Summary:

We have these frames:

```
G:  X=10, Y=12, BAR=P1, FOO=P2.
E1: X=1, extends G.
E2: empty, extends G.
E3: Y=2, VAL=P4, extends E1.
E4: Y=1, extends E2.
```

And we have these procedures:

```
P1: No parameters, body is (LAMBDA (Y) (+ X Y)), right bubble is G.
P2: Parameter X, body is (LET ...), right bubble is G.
P3: Parameters Y and VAL, body is (+ (VAL X) 4), right bubble is E1.
P4: Parameter Y, body is (+ X Y), right bubble is E2.
```

Scoring:

There were a few really unusual solutions that didn't fit the following categories; they were mostly scored 0 or 1.

```
5  Correct (maybe except the names of E2 and E3).
4  Almost correct, mostly follows lexical scope, one or two things that
   look just careless rather than misguided.
3  Dynamic scope (E4 extends E3, and/or E2 extends E1) and the final
   answer is 6.
2  Dynamic scope, but the final answer is still 15, which suggests that
you
   don't think the environment diagram actually /means/ anything!
2  VAL is bound to the same procedure as BAR.
1  Normal order (e.g., Y is bound to X in E4, or VAL is bound to (BAR)
   in E3), or bindings of non-symbols (e.g., (BAR) in the global frame,
or
   (VAL X) in some extra frame).
0  Only two frames, only two procedures, or just incoherent.
```