CS 61A   Midterm 3 solutions   Fall 2009

1. Box and pointer/what will Scheme print?

(a)

```
> (let ((ls '(1 2 3 4)))
    (for-each (lambda (x) (set! x (+ x 1))) ls)
    ls)
```

Result: (1 2 3 4)

This was a question about understanding the difference between SET! and SET-CAR!. The SET! inside the FOR-EACH changes a local variable of the lambda, but does nothing to the pairs in LS.

Scoring: 1 point, all or nothing.

(b)

```
> (define x '(a b))
> (define y '(c))
> (set-cdr! y x)
> (set-car! (cdr x) '(d))
> y
```

Result: (c a (d))

```
Y ----> XX-------------------+
        | .                  |
        |  .                 |
        V   /                V
        c           X -----> XX--->X/
                             |     :\
                             |     : \
                             V     V  V
                             a     b  X/
                                      |
                                      |
                                      V
                                      d
```

This was a straightforward list mutation. The SET-CDR! changes Y from a one-element list to one that also includes the elements of X, as if we'd said (append! y x). It's important that the pointer from the cdr of Y points to /the pair/ that X points to, neither to the variable X nor to the word A. The SET-CAR! changes the second element of X from

B to (D) -- the list containing D. That list is a pair, which must be
in the diagram; you can't just put "(D)" in the diagram!

Scoring: One point for the printed result, one for the diagram.

2. Concurrency.

```
> (define y 4)
> (parallel-execute
    (lambda () (set! y (* 3 y)))
    (lambda () (set! y (+ y (- y 2)))) )
```

(a) Correct values of Y.

"Correct" with respect to concurrency means "a value that could have resulted from a non-concurrent (sequential) processing of the threads." With two threads, there are only two sequential orderings -- first one first, or second one first. (In general, N threads can be sequentially ordered in N! (N factorial) ways, so that's how many correct answers there are unless multiple orders happen to have the same result.)

If we do the first one first, (set! y (* 3 y)) sets Y to (* 3 4) = 12.
Then (set! y (+ y (- y 2))) sets Y to (+ 12 (- 12 2)) = 22.

If we do the second one first, (set! y (+ y (- y 2))) sets Y to
(+ 4 (- 4 2)) = 6. Then (set! y (* y 3)) sets it to (* 6 3) = 18.

So the two correct answers are 22 and 18.

(b) Additional incorrect values.

There are three of them. The most obvious ones are the cases in which both processes read the original Y=4, both do their calculations, and whichever one stores its result second wins.

If the first thread wins, the answer is (* 3 4) = 12.

If the second thread wins, the answer is (+ 4 (- 4 2)) = 6.

The third case is the one in which thread 1 stores its result in between the two times that thread 2 loads the value of Y. Some people asked about whether to assume left-to-right or right-to-left evaluation, but in fact the order doesn't affect the result for this expression, because (+ 4 (- 12 2)) and (+ 12 (- 4 2)) are both 14.

So the three possible incorrect answers are 12, 6, and 14.

Scoring: 1 point for each value listed in the correct category, minus 1/2 point for each possible value listed in the wrong category (e.g., 12 listed as a correct answer in part (a)), minus 1 point for any value listed other than 22, 18, 12, 6, or 14, minus 1 point for each missing number (i.e., fewer than five values given), truncated downward.

3. Streams

We decided to give everyone the full 3 points for this question, because we had too much trouble reading your minds in arguing about part credit. But there /are/ right and wrong answers.

Wrong answers are the ones that also prove that, say, STREAM-MAP can't work either! For example, "a recursion with no base case" is true of stream-map as applied to infinite streams. "Infinite loop" was one that we argued about; students who said that /probably/ meant something that would be true for any infinite stream procedure, but we weren't sure.

There are two kinds of correct answers. The first kind has to do with Louis's actual code:

> The recursive call to STREAM-SMALLEST is not protected by a DELAY (either explicit or implied by a CONS-STREAM), and therefore runs forever rather than producing a result including a promise.

The key point here is mentioning the need for DELAY.

The other kind of correct answer ignores Louis's code and instead points out that this problem cannot be solved even in principle:

> In order to know the smallest element of a sequence, every element must be examined. This is obvious in the case of a decreasing stream such as the negative integers, but even if the initial elements of the stream suggest an increasing pattern, there's no guarantee that the later elements will follow the same pattern forever.
>
> More generally, MAP and FILTER patterns can be generalized to infinite streams because the result for each element depends only on that element, and can be determined while still delaying the processing of later elements. But an ACCUMULATE pattern (and STREAM-SMALLEST is basically STREAM-ACCUMULATE of MIN) can't work for infinite streams because no partial answer is possible based on just a finite number of elements.

Note, by the way, that it /is/ possible to compute a smallest-so-far /stream/
for a given stream. Its Nth element would be the smallest value among the
first N elements of the input stream.

4. Client/server.

```
(define (receive-message message other-computer)
  (cond ((EQ? MESSAGE 'PING)            ; REPLY TO PING REQUEST
         (SEND-MESSAGE 'PONG OTHER-COMPUTER))
        ((EQ? MESSAGE 'PONG)              ; WE GOT A REPLY
         (display other-computer)
         (display " is online.")
         (newline))
        (else (error "Invalid message received: " name))))
```

The big idea here is that the Ping software has to be able to do two things:
make ping requests to another computer (in which case the other computer will
send it a PONG message), and respond to PING messages from another computer.
So there will be a user-interface-level procedure along the lines of

```
   (define (ping other-computer)
     (send-message 'ping other-computer))
```

This procedure is directly invoked by the user typing a (ping ...) expression
at the Scheme prompt. But receive-message is called by the operating system
in the case of a /network/ message; the local user doesn't directly invoke it.

Scoring: Four parts of understanding the question got one point each:
      * Checking for message names PING and PONG.
      * PONG is the message that displays the "is online" message.
      * SEND-MESSAGE is called to send a message to the other computer.
      * The message that's sent is PONG, not PING.

5. Local state variables.

```
(define previous ; note no parentheses
  (let ((old 'pterodactyl))
    (lambda (msg)
      (let ((result old))
      (set! old msg)
      result))))
```

Of course the crucial point here is that the LAMBDA has to be /inside/ the
body of the (LET ((OLD ...)) ...) in order for it to persist across calls
to PREVIOUS.

Scoring:

5 Correct.

4 Trivial error.

3 Instead of using a temporary variable (RESULT above) to preserve the old
     value, uses DISPLAY or PRINT before the SET! and doesn't return a value.
3 The LET that sets RESULT is outside the lambda, not inside.
3 Returns the wrong value but does update OLD correctly.

2 Has persistent local state OLD, and temporary variable RESULT, but has
     some other logic error.

1 No persistent local state variable -- usually, (DEFINE (PREVIOUS MSG) ...).

0 Even worse.

6. List mutation.

As always in mutation questions, it's crucial not to lose pointers that
you're going to need later. This can sometimes be done just by doing the
mutations in exactly the right order, as in this solution:

```
(define (lists->assoc! keys values)
  (if (null? keys)
      '()
      (begin (lists->assoc! (cdr keys) (cdr values))
             (set-cdr! values (car values))
             (set-car! values (car keys))
             (set-car! keys values))))
```

This is probably the most elegant solution, but it does have the disadvantage
that the recursive call isn't a tail call, so it's not space-efficient.

Another approach is to use a LET to keep around /all/ the useful pointers
while doing the mutations, so it doesn't matter what order they're done in:

```
(define (lists->assoc! keys values)
  (if (null? keys)
      '()
      (let ((thiskey (car keys))
```

```
          (thisvalue (car values))
          (restkeys (cdr keys))
          (restvalues (cdr values)))
      (set-car! keys values)
      (set-car! values thiskey)
      (set-cdr! values thisvalue)
      (lists->assoc! restkeys restvalues))))
```

Here we have more LET variables than we really need; one would have been enough to let us have the recursive call in tail position. But using this
technique ensures that you won't lose a pointer.

Some students thought that saying something like

```
      (let ((kv-pair values))
        ...)
```

would make a /copy/ of (the first pair of the list) VALUES, so that you could then mutate KV-PAIR without losing the information in VALUES. But LET doesn't copy pairs -- if it did, you wouldn't be allowed to use it in this problem! What you have to remember in the LET is (CDR VALUES).

Some people noticed that one of the sample exams in the reader asks for a procedure MAKE-ALIST! that turns a list of alternating keys and values into an association list, and used it this way:

```
(define (lists->assoc! keys values)
  (interleave! keys values)
  (make-alist! keys))
```

That wasn't what we had in mind, but it's correct, given that it uses the mutating INTERLEAVE! and not the copying INTERLEAVE.

Scoring:

7 Perfect.

6 Trivial error (e.g., fails only for empty list).

4-5 Has the idea:

5 Fails for the last key and value in the lists (wrong base case).
5 Changes VALUES to the alist instead of KEYS.
5 No base case at all.
5 Recursive calls on (CDR VALUES) after a (SET-CDR! VALUES ...).
5 Uses the return value, but doesn't return a value!

4 Worse cases of losing pointers than above.
4 Only includes half the keys/values in the result (because only pairs from KEYS appear in the result).
4 Off-by-one error in matching keys with values.

2-3 Has an idea:

3 Creates key/value pairs correctly but fails at making a list of them.
3 Knows about how to mutate, but the logic is way off.

2 Worse than that, but some morsel of making sense.

0-1 Other. No specific examples here, except that creating new pairs (CONS, LIST, or APPEND) always got 0 points.

7. Vectors.

The "set up index variables L and R" suggests using a helper procedure with
those names as parameters.

```
(define (has-sum-pair vec x)
  (define (help L R)
    (cond ((= L R) #f)
        ((> (+ (vector-ref vec L) (vector-ref vec R)) x)
         (help L (- R 1)))
        ((< (+ (vector-ref vec L) (vector-ref vec R)) x)
         (help (+ L 1) R))
        (else #t)))
  (help 0 (- (vector-length vec) 1)))
```

Scoring:

7 Correct.
7 Got increment and decrement backwards because of our error in the exam.

6 Made our correction on the exam, but still got increment and decrement
   backwards.

4-5 Has the idea:

5 R's initial value is (vector-length vec).
5 L's initial value is 1.
5 Bad base case.
5 Correct helper procedure, no actual call in main body.

4 Confuses index (e.g., L) with element (e.g., (vector-ref vec L)).
4 Tries to use SET! to set L and R, but overrides the change in
   a recursive call.

2-3 Has an idea. There were few of these, all unique.

0-1 Other:

0 No recursion.
0 Using CAR or CDR on a vector.

Group question. Environments.

```
(define a 10)
(define b 100)
```

These just add bindings to the global frame G.

```
(define (foo a)
  (let ((b a))
    (lambda (c) (set! b (+ c a))) ))
```

This creates a procedure:

P1: param (a), body (let ...), env G

and makes a binding FOO -> P1 in the global frame G. It doesn't evaluate the let or the inner lambda, so that's all that happens!

((foo 5) 7)

This is a procedure call. The first step is to evaluate all the subexpressions. The value of 7 is 7. So now we have to evaluate the procedure call (foo 5):

We evaluate the subexpressions FOO and 5. The value of FOO is P1; the
value of 5 is 5.

Now we invoke P1 with actual argument 5. The first step is to create a new frame that extends the environment in the right bubble of P1, namely G:

E1: A -> 5, extends G.

With E1 as the current environment, we evaluate the body of P1, which is
the LET expression. A LET is a lambda followed by a procedure call. The lambda creates a procedure:

P2: param (b), body (lambda (c) ...), env E1.

Now we invoke that procedure. Its actual argument expression is A, so
we look that up in the current environment, E1, where we find the value 5. So the actual argument value is 5. (If there hadn't been a binding for A in frame E1, we would have continued to frame G and used
the value 10.) We make a new environment:

E2: B -> 5, extends E1 (because P2's right bubble is E1).

The value of B is 5, not A!!! That's what applicative order means.

With E2 current, we evaluate the body of P2, a lambda expression that creates a procedure:

P3: param (c), body (set! ...), env E2.

So the call (foo 5) returns P3.

Now we call P3 with the argument 7. Calling a procedure creates a new environment:

E3: C -> 7; extends E2 (because P3's right bubble is E2).

The body of P3 is (set! b (+ c a)). The current environment is E3, so we look for bindings of all the variables. We get + = the addition primitive (from G), C = 7 (from E3), and A = 5 (from E1). So the result of the addition (no frame needed for primitives) is 12.

Now we change the value of B to 12. Which B? The current environment is still E3, so we look there for a binding, don't find one, then look in the environment it extends, E2, where we find the binding B=5. That's the one we change, from 5 to 12.

Finally, at the Scheme prompt we get the expression B, which we look up in the /global/ environment. The value we find there is unchanged, so the final result is 100.

Scoring: Most groups got this right. Here are some of the mistakes people made with their scores.

4 points: Correct except arrows pointing backward.

3 points: No mutation done, or B bound to A instead of 5 in E2.

2 points: Minor errors of which frames point to which, missing or extra frames, or extra bindings.

1 point: Final result is 12, or very strange unique diagrams.

Nobody got 0 points.