CS 61A Midterm #3 — November 18, 2009

Your name _____

login: cs61a–_____

Discussion section number _____

TA's name _____

This exam is worth 40 points, or about 13% of your total course grade. It includes two parts: The individual exam (this part) is worth 35 points, and the group exam is worth 5 points. The individual exam contains seven substantive questions, plus the following:

Question 0 (1 point): Fill out this front page correctly and put your name and login correctly at the top of each of the following pages.

This booklet contains seven numbered pages including the cover page. Put all answers on these pages, please; don't hand in stray pieces of paper. This is an open book exam.

When writing procedures, don't put in error checks. Assume that you will be given arguments of the correct type.

Our expectation is that many of you will not complete one or two of these questions. If you find one question especially difficult, leave it for later; start with the ones you find easier.

If you want to use procedures defined in the book or reader as part of your solution to a programming problem, you must cite the page number on which it is defined so we know what you think it does.

READ AND SIGN THIS:

I certify that my answers to this exam are all my own work, and that I have not discussed the exam questions or answers with anyone prior to taking this exam.

If I am taking this exam early, I certify that I shall not discuss the exam questions or answers with anyone until after the scheduled exam time.

_____

0	/1
1-3	/11
4	/4
5	/5
6	/7
7	/7
total	/35

Question 1 (3 points):

(a) What will Scheme print in response to the following expression? If the expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message.

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

(b) What will Scheme print in response to the last of the following expressions? If the expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message. **Also, draw a box and pointer diagram for the value produced by the last expression,** and show which pairs `x` and `y` refer to.

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

Your name _____ login cs61a–_____

Question 2 (5 points):

After executing the following code:

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

(a) List all **correct** values of `y`:

(b) List all **additional** possible values of `y`:

Question 3 (3 points):

Louis Reasoner wants to write a procedure `stream-smallest` that takes an infinite stream and returns the smallest element in it. "It's easy," he says, "It's just like `min` or `smallest` for lists!"

```
(define (stream-smallest strm)
  (let ((small (stream-smallest (stream-cdr strm))))
    (if (< (stream-car strm) small)
        (stream-car strm)
        small)))
```

This doesn't work. In no more than two sentences, explain to him (and us) why not. ("The stream is infinite" is not a good enough answer!)

Question 4 (4 points):

`Ping` is a utility program used to find out if another computer is on the network; the basic idea is that one computer sends the other a PING message and the other sends the first one back a PONG message. If the first computer receives the PONG, it knows the second is running and prints out a message saying so. Here's a simplified example:

```
% ping star.cs.berkeley.edu
star.cs.berkeley.edu is online.
```

The following procedure is meant to be used as a callback for any network activity. Fill in the blanks to make it handle PING and PONG messages correctly.

```
(define (receive-message message other-computer)

  (cond (_____________________________________

         _____________________________________)

        (_____________________________________
          (display other-computer)
          (display " is online.")
          (newline))
        (else
          (error "Invalid message received: " name))))


; you may assume this procedure is already defined
(define (send-message message other-computer) ...)
```

Your name _____ login cs61a–_____

Question 5 (5 points):

Write a procedure `previous`, with one argument, that behaves as follows: The first time it is called, it returns the name of your favorite prehistoric creature, such as `pterodactyl`. Thereafter, it returns the argument of the previous call.

For example:

```
> (previous 'hello)
pterodactyl
> (previous 42)
hello
> (previous #t)
42
```

Your implementation may **not** use `define-class`!

Question 6 (7 points):

Implement a procedure `lists->assoc!` that takes a list of keys and a list of values as its arguments and, using mutation (`set-car!` and `set-cdr!`), creates a single association list.

```
> (define L1 (list 1 2 3))
L1
> (define L2 (list 'a 'b 'c))
L2
> (lists->assoc! L1 L2)
okay              ;; return value is ignored
> L1
((1 . a) (2 . b) (3 . c))
```

The pairs of the first argument should become the spine pairs of the result; the second argument need not be preserved. Your implementation may **not** create any new pairs! You may assume the two arguments are always lists of equal length.

```
(define (lists->assoc! keys values)
```

Your name _____ login cs61a–_____

Question 7 (7 points):

Given a *sorted* vector and a number `x`, we want to find out if there are two numbers in the vector that add up to `x`. The following algorithm solves the problem:

Set up index variables `L` and `R`, starting at the first and last elements respectively.
Add the two elements selected by indices `L` and `R`.
If their sum is larger than `x`, increment `L` by one and try again.
If their sum is smaller than `x`, decrement `R` by one and try again.
If their sum is `x`, return true.
If the two indices are equal to each other, return false.

Implement this algorithm. Do not use `list->vector` or `vector->list`.

```
(has-sum-pair '#(1 3 7 11 13) 18)

#(1 3 7 11 13)
  ^         ^ = 14 ;; lower than 18, so move first index up
#(1 3 7 11 13)
    ^       ^ = 16 ;; lower than 18, so move first index up
#(1 3 7 11 13)
      ^     ^ = 20 ;; higher than 20, so move last index down
#(1 3 7 11 13)
      ^  ^    = 18 ;; done!


(define (has-sum-pair vec x)
```