CS 61A Midterm #1 — February 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. The exam contains 7 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 6 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 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
1–2	/8
3–4	/8
5	/5
6	/8
7	/10
total	/40

Question 1 (4 points):

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just write "procedure"; you don't have to show the form in which Scheme prints procedures.

```
(every (lambda (x) (se x x))
       (map (lambda (x) (* x x)) '(1 2 3)))
```
_____

```
(keep (lambda (x) x) (keep even? '(1 2 3 4 5 6 7)))
```
_____

```
(let ((first last) (last first))
  (last (first '(for no one))))
```
_____

```
(cadadr '((a b c d e) (f g h i j) (k l m n o)))
```
_____

Question 2 (4 points):

What will Scheme print in response to the following expressions? If an 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 each expression.**

```
(append (cons '(1) '(2)) (list '(1) '(2)))
```
_____

```
(list (cons 2 3))
```
_____

Your name _____ login cs61a–_____

Question 3 (4 points):

```
(define (square x)
        (* x x))

(define (foo x y)
        (+ (square x) y))

(foo (square 1) (square 2))
```

How many times is `square` called in:

Normal order _____ Applicative order _____

Question 4 (4 points):

What is the order of growth in time of the following procedure, in terms of its argument value n? Also, does it generate an iterative process or a recursive process?

```
(define (mystery n)
  (cond ((< n 1) 0)
        ((even? n) (+ 1 (mystery (- n 1))))
        (else (+ 2 (mystery (- n 2))))))
```

_____$\Theta(1)$ _____$\Theta(n)$ _____$\Theta(n^2)$ _____$\Theta(2^n)$

_____Iterative _____Recursive

Question 5 (5 points):

We are creating a database of the greatest songs in the world. The first step is to define an abstract data type for a song:

```
(define title car)            ; Each TITLE and ARTIST
(define artist cadr)          ; is a sentence, so a song
(define make-song list)       ; is a list of sentences.
```

Now we set up a global variable `great-songs` whose value is a list of songs:

```
(define great-songs (list (make-song '(she loves you) '(the beatles))
                          (make-song '(waterloo sunset) '(the kinks))
                          (make-song '(pictures of lily) '(the who))
                          (make-song '(davy the fat boy) '(randy newman))
                          (make-song '(expecting to fly)
                                     '(buffalo springfield))
                          (make-song '(tell her no) '(the zombies))))
```

We want a procedure `who-sang` that takes a song title as its argument and returns the corresponding artist, or `#f` if the song isn't one of the greatest in the world:

```
> (who-sang '(waterloo sunset))
(THE KINKS)

> (who-sang '(stairway to heaven))
#F
```

Below is an implementation of `who-sang`. It works correctly, but it doesn't respect data abstraction. Fix it to use the appropriate abstract data types:

```
(define (who-sang song)

  (define (helper song songs)

    (cond ((null? songs) #f)

          ((equal? song (caar songs)) (cadar songs))

          (else (helper song (cdr songs)))))

  (helper song great-songs))
```

Your name _____ login cs61a–_____

Question 6 (8 points):

Note: Use recursion, not higher-order functions, to solve this problem.

Write a procedure `remove-n` that takes a sentence, a target word, and a number `n` as arguments and returns a sentence with the fragment of length `n` starting at the target word removed:

```
(remove-n '(i hate to miss discussions for cs61a) 'to 4)
> (i hate cs61a)
```

You may assume that the target word appears at most once in the sentence. (If it doesn't appear, return the original sentence.) You may also assume that if the target word is found, then there are enough words in the sentence, starting with the target word, to satisfy the count `n`.

It's okay to write helper functions if you want.

Question 7 (10 points):

For this question (both parts), use only higher order procedures, not recursion, even in helper procedures!

(a) Write a procedure `cross` that takes a word as argument, and returns a sentence of all possible two-letter words made from the letters of the given word. For example:

```
> (cross 'abc)
(aa ab ac ba bb bc ca cb cc)
```

(b) Now write a procedure `make-crosser` that takes a *two-argument function* `fn` as its argument, and returns a function of one word `wd` that returns a sentence of all possible results of combining two letters of `wd` using function `fn`. For example, `(make-crosser word)` should return the function `cross` from part (a).

Another example:

```
> ((make-crosser *) 2345)
(4 6 8 10 6 9 12 15 8 12 16 20 10 15 20 25)
```