CS 61A Final Exam — December 18, 2008

Your name _____

login: cs61a–_____

This exam is worth 70 points, or about 23% of your total course grade. The exam contains 15 questions.

This booklet contains 14 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.

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.

***** IMPORTANT *****

Check here if you are one of the people with whom we arranged to replace a missed/missing exam with other exam scores: ____

***** IMPORTANT *****

If you have made grading complaints **that have not yet been resolved**, put the assignment name(s) here:

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.

_____

1–3	/10
4–5	/11
6–7	/6
8–9	/7
10	/4
11	/3
12	/5
13	/8
14	/8
15	/8
total	/70

Question 1 (4 points):

What will the Scheme interpreter print in response to each of the following expressions? If any expression results in an error, just write "ERROR"; you don't have to give the precise message.

```
(map (lambda (x) (x 3))
     '(number? pair? even?))
```

```
(map (lambda (x) (x 3))
     (list number? pair? even?))
```

Question 2 (4 points):

Circle TRUE or FALSE for each of the following:

TRUE FALSE The `first` procedure that we provided for you can be defined by `(define first car)`

TRUE FALSE If `(pair? x)` returns `#f`, then `(list? x)` will always return `#f` also.

Question 3 (2 points):

What is the smallest number of pairs that can be included in a structure `x` so that the expression

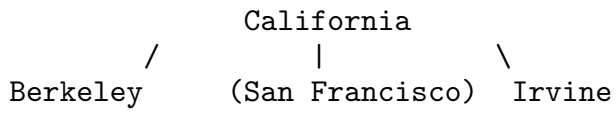
```
(set-cdr! (cddar x) 'cs61a)
```

returns a value rather than causing an error message?

Your name _____ login cs61a–_____

Question 4 (5 points):

If the value of variable `mytree` is the Tree (with datum and children) shown here:



and each datum is a word or sentence, write an expression using `mytree`, and not using any quoted values, that returns the word `San` as its value. **Respect all relevant data abstractions!** Assume that the children appear in the list in the order shown above.

Question 5 (6 points):

In the OOP language (using `define-class`), define a `Tree` class with `datum` and `children` as instantiation variables, and with a method `treemap!` that takes a one-argument function as its argument, and *mutates the tree* by applying the function to every value in the tree.

Question 6 (3 points):

Write a function `pairup` that takes two arguments: a two-argument function `fn` and a nonempty list. It returns a list of length one less than the length of the argument list, in which each element is the value returned by `fn` called with two consecutive elements of the data list, like this:

```
> (pairup + '(5 7 1 22 6))
(12 8 23 28)
```

Question 7 (3 points):

Suppose Scheme didn't provide pairs as a primitive data type, but did provide vectors. We want to implement pairs as an abstract data type, using the following constructor:

```
(define (cons a b)
  (vector 'pair a b))
```

Define the corresponding selectors and mutators `car`, `cdr`, `pair?`, `set-car!`, and `set-cdr!`.

Your name _____ login cs61a–_____

Question 8 (4 points):

Here's the situation: You're buying an airplane ticket online and you've just clicked on "Show seat map." They put a map of the airplane on your screen, with each seat marked as available or taken. You click on a seat to reserve it.

Technique 1: They acquire a mutex as soon as you click "show seat map" and release it after you click on a seat and they've assigned it to you.

Technique 2: When you click on a seat, they acquire a mutex, see if the seat is still available, assign it to you if available, and then release the mutex.

In one sentence, what undesirable result could happen if they use technique 1?

In one sentence, what undesirable result could happen if they use technique 2?

Question 9 (3 points):

Write a definition of the stream s1, whose first 15 elements are given here:

```
1, 2, 3,   2, 3, 4,   3, 4, 5,   4, 5, 6,   5, 6, 7, ...
```

(Notice we've used spacing to show you the pattern.)

Question 10 (4 points):

The TAs, trying to keep their jobs in the face of heavy budget cuts, are busily writing discussion notes to seem useful. They want to use `mapreduce` to extract key elements of Chung Wu's existing notes. The file `/chung-notes` is keyed on topic (`ordersofgrowth`, `higherorder`, `logic`, etc.) with a line from the notes as the value.

They want to supplement the existing notes wherever they are weakest. With this in mind, write a `mapreduce` call to get the number of discussion questions written for each topic. You can assume that all discussion questions, and only such questions, start with a line whose first word is a number and a period, e.g., `23.`; we've provided the predicate `exercise?` that checks a line for this:

```
(define (exercise? line)
  (and (not (empty? line))
       (equal? (last (first line)) ".")
       (number? (butlast (first line)))))
```

Your name _____ login cs61a–_____

Question 11 (3 points):

Write a procedure `an-element-satisfying` to be run using the nondeterministic evaluator. It takes a predicate function and a list as arguments, and returns an element from the list for which the predicate returns a true value. If a later `amb` expression fails, then `an-element-satisfying` will return a different element for which the predicate is true. When there are no more such elements, it will fail. (This is the nondeterministic equivalent to the `filter?` function. But don't use `filter?` in your solution!) For example:

```
ambeval> (an-element-satisfying even? '(5 7 28 4 19 23))
28

ambeval> try-again
4

ambeval> try-again
There are no more values of (an-element-satisfying even? '(5 7 28 4 19 23))
```

Question 12 (5 points):

Write a query system rule or rules for `doubled-element`, a relation among an element and two lists, which holds if the second list is like the first except that every occurrence of the given element is doubled. For example:

```
query> (doubled-element la (ob la di ob la dah) ?x)
(doubled-element la (ob la di ob la dah) (ob la la di ob la la dah))
```

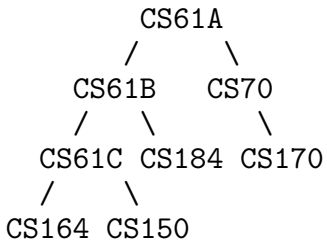
Do not use `lisp-value`!

Your name _____ login cs61a–_____

Question 13 (8 points):

At Depth University, a student must complete at least one advanced class to graduate. However, every advanced class has a single prerequisite, which may itself have a prerequisite, and so on. Write a procedure `fast-grad` that, given a prerequisite Tree (with constructor `make-tree` and selectors `datum` and `children`) returns the *shortest possible* list of courses needed to graduate. If there is a tie, `fast-grad` may return any of the shortest lists. You may assume that all leaf nodes are advanced classes, and vice versa.

For example, `fast-grad` called on the following tree can return `(CS61A CS70 CS170)` or `(CS61A CS61B CS184)`; either one is correct.



Question 14 (8 points):

Alyssa asks Ben to write a procedure `oddeven!` that takes a list of integers as its argument, and returns the same list, but rearranged so that all the odd numbers come first, followed by all the even numbers:

```
> (oddeven! (list 8 4 15 3 6 11 9 20))
(15 3 11 9 8 4 6 20)
```

Note that the order of the odd numbers is unchanged, and the order of the even numbers is unchanged. Ben is supposed to write this **by mutation, not allocating any new pairs.**

After thinking about this for a while, Ben says, "This will be much easier if I can have sentinel nodes for the odd sublist and for the even sublist. May I allocate just two pairs for that purpose?" Alyssa agrees, and Ben immediately writes this:

```
(define (oddeven! nums)
  (let ((odds (list 'odd))
        (evens (list 'even)))
    (oe-help odds odds evens evens nums)))

(define (oe-help odds oddlast evens evenlast nums)
```

... but then he's called away urgently, so he asks you to finish the definition of `oe-help`. Ben's plan is that `odds` will be a list starting with the sentinel node and then containing all the odd numbers examined so far; `oddlast` will be the last pair of that odd-number list. Similarly, `evens` and `evenlast` will be the first and last pair of the even numbers so far, with sentinel node.

Ben shows you this picture of the arguments to `oe-help` after the first three elements of the list in the example above have been examined. The dotted lines indicate pointers that, although unchanged so far, are irrelevant to the solution because they *will* be changed later.

Finish Ben's helper procedure. **Do not allocate any new pairs.**

Write your answer on the next page.

Your name _____ login cs61a–_____

Question 14 answer goes here:

Question 15 (8 points):

Sometimes, when debugging a procedure with local state, you wish you could get access to local state variables from outside. You're going to add a special form called `eval-in-env` that takes two arguments: an expression, and a procedure whose defining environment will be used to evaluate the expression. For example, suppose we've said

```
(define count
  (let ((counter 0))
    (lambda ()
      (set! counter (+ counter 1))
      counter)))
```

After using the counter a few times, we could say

```
(eval-in-env (set! counter 0) count)
```

to reset the internal `counter` variable.

The relevant metacircular evaluator procedures are listed on the remaining pages of the exam. **On this page, write the names of all the procedures that you modify elsewhere.** New procedures can go here or on page 14.

Your name _____ login cs61a–_____

```
(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (mc-eval (cond->if exp) env))
        ((application? exp)
         (mc-apply (mc-eval (operator exp) env)
                   (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))

(define (mc-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
           (procedure-body procedure)
           (extend-environment
             (procedure-parameters procedure)
             arguments
             (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))



(define (definition? exp)
  (tagged-list? exp 'define))

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (mc-eval (definition-value exp) env)
                    env)
  'ok)
```

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))

(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))


(define (make-frame variables values)
  (cons variables values))

(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```