CS 61A Final Exam — December 17, 2009

Your name _____

login: cs61a–_____

This exam is worth 70 points, or about 23% of your total course grade. The exam contains 12 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	/4
2	/4
3–4	/10
5	/8
6	/4
7	/6
8	/6
9	/4
10	/8
11	/8
12	/8
total	/70

Question 1 (4 points):

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

```
> (define x '(1))
> (define y '(2))
> (define z '(3))
> (set! y z)
> (set! x y)
> (set! z '(4))
> x
```

_____

```
> (define mylist (list 2 4 6 8))
> (define (magic! ls)
      (set-car! (cdr ls) 100)
      (set! ls (cons 1 ls))
      (set-car! (cdr ls) 4))
> (magic! mylist)
okay
> mylist
```

_____

Your name _____ login cs61a–_____

Question 2 (4 points):

Consider the following sequence of expressions:

```
> (define y 6)
> (define z 3)
> (define (foo x)
    (set! y (+ x y z))
    y)
> (define (bar z)
    (foo 10))
> (bar (foo 7))
```

What is the result of the last expression...

...using **lexical scope**? _____ ...using **dynamic scope**? _____

Now, draw the environment that would be created using **dynamic scope**:

Question 3 (6 points):

Answer each of the following in order-of-growth notation.

(a) Suppose you have n different keys and one locked box to open. You know that one and only one key opens the box, but forgot which one. In the worst case, how many tries will it take to open the box?

(b) Now suppose there is a second box inside the first. Another of your n keys opens this box. In the worst case, how many tries will it take to open *both* boxes?

(c) To open a safety deposit box, you need to insert two keys and turn them simultaneously; if either key is incorrect, the box will not open. In the worst case, how many tries will it take to open a safety deposit box with n possible keys?

Question 4 (4 points):

Mark whether the following statements are true or false:

Applicative order guarantees that arguments to a procedure are evaluated before the procedure body.

True _____ False _____

Normal order guarantees that arguments to a procedure are evaluated from left to right.

True _____ False _____

Your name _____ login cs61a–_____

Question 5 (8 points):

A list is a _____whose _____is _____, or the empty list.

For each of the following, circle the **one best** answer.

A list is more efficient than a vector when...

a) Removing items from the front of the sequence.

b) Replacing values at the front of the sequence.

c) Printing out every element.

d) Never; vectors are always more efficient.

A client-server connection's three-way handshake ensures...

a) The network is reliable.

b) Both sides can send and receive messages.

c) The connection will be terminated when the client quits.

d) None of the above.

The argument to a continuation is...

a) The previous expression evaluated by the interpreter.

b) The expression to evaluate next.

c) The value computed by a procedure call.

d) An environment.

Question 6 (4 points):

Circle all blocks of code (if any) that guarantee that the results of their parallel execution are **correct** and that no deadlock will occur. Assume `x`, `y`, and `z` have previously been defined with integer values.

```
> (define s (make-serializer))
> ((s (lambda ()
        (parallel-execute
         (lambda () (set! z (- x y)))
         (lambda () (set! y (+ y x)))
         (lambda () (set! x 10)) ))))
```

```
> (define s (make-serializer))
> (parallel-execute
   (lambda () (set! z (- x y)))
   (s (lambda () (set! y (+ y x))))
   (s (lambda () (set! x 10))) )
```

```
> (define ser-x (make-serializer))
> (define ser-y (make-serializer))
> (parallel-execute
   (ser-x (ser-y (lambda () (set! z (- x y)))))
   (ser-y (ser-x (lambda () (set! y (+ y x)))))
   (ser-x (lambda () (set! x 10))) )
```

```
> (define s (make-serializer))
> (parallel-execute
   (s (lambda () (set! z (- x y))))
   (s (lambda () (set! y (+ y x))))
   (s (lambda () (set! x 10))) )
```

Your name _____ login cs61a–_____

Question 7 (6 points):

Given a list of search keywords, we want to find out which documents in our input stream contain each word, using MapReduce. The input key-value pairs have a document **name** as the key, and a **list of words** from one line of that document as the value. (Note: The same document may have many lines.)

Write a procedure `search` that, given a list of keywords and the input stream, returns a stream sorted by search word, in which we can find, for any keyword, the names of the documents containing that word. (Don't worry about cases in which the same keyword appears more than once in a file.)

Reminder: A call to `mapreduce` looks like this:

```
(mapreduce mapper reducer base input-stream)
```

```
(define (search keywords input-stream)
```

Question 8 (6 points):

In this question, we concern ourselves with a robot that takes steps along a 1-dimensional path. The robot's position on the path is represented by a single integer. We will use **functional** programming (that is, **no mutation**) to model the robot.

The robot understands three possible commands: `forward`, which moves the robot one step in the positive direction, `back`, which moves it one step in the negative direction, and `home`, which moves the robot back to position 0.

(a) You have decided that you want to use **data-directed programming** to control the robot by putting the three commands in a table. Write procedures for the `forward`, `back`, and `home` commands, and show how they should be inserted in the table. Each procedure should take the robot's old position as its argument, and return the robot's new position.

Question 8 continues on the next page.

Your name _____ login cs61a–_____

Question 8 continued

(b) Now, implement a function `final-position` that takes an initial position and a list of command names, simulates the application of the commands **in order,** and returns the final position. Your code should **not** use a `cond` or `if` to test the command names!

```
(define (final-position init cmds)
```

Question 9 (4 points):

Show the first eight elements of the stream defined as follows:

```
(define my-stream
  (cons-stream 5 (interleave my-stream
                             (stream-map (lambda (x) (* x 2))
                                         my-stream))))
```

_____ _____ _____ _____ _____ _____ _____ _____

Question 10 (8 points):

(a) Write logic (query) language rules for `prefix`, a relation between two lists that is satisfied **if and only if** the elements of the first list are the **first** elements of the second list, in order. For example, each of these examples matches the relation:

```
(prefix (being for the) (being for the benefit of mister kite))
(prefix (for no one) (for no one))
(prefix () (got to get you into my life))
```

But these do not satisfy the relation:

```
(prefix (want i to) (i want to hold your hand))
(prefix (to hold your) (i want to hold your hand))
(prefix (i want to tell you) (i want to))
```

Do not use `lisp-value`!

This question continues on the next page.

Your name _____ login cs61a–_____

Question 10 continued:

(b) Now write rules for `sublist`, a relation between two lists that is satisfied if and only if the first is a consecutive sublist of the second. For example, each of these examples matches the relation:

```
(sublist (give) (never gonna give you up))
(sublist (you up) (never gonna give you up))
(sublist (never gonna give) (never gonna give you up))
(sublist () (never gonna give you up))
```

And these do not:

```
(sublist (never give up) (never gonna give you up))
(sublist (let you down) (never gonna give you up))
```

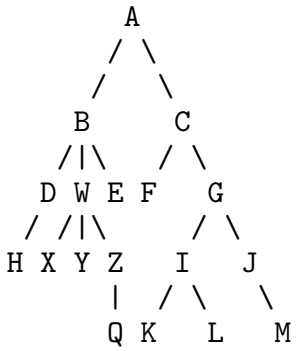
Do not use `lisp-value`!

Hint: You will want to use the `prefix` relation from part (a) to help you.

Question 11 (8 points):

Write `level`, a function that takes two arguments: a **nonnegative integer** and a **Tree** (`datum`, `children` ADT). It returns a **forest** containing all of the children of children of children... to the specified number of levels. (If the integer is 0, the Tree itself is the only member of this level. Level 1 is the tree's children; level 2 is its grandchildren.)

For example, if `my-tree` is the Tree



then `(level 3 my-tree)` will return the trees whose data are `H`, `X`, `Y`, `Z`, `I`, and `J`. Note that not all paths through the tree reach down to level 3; for example, `E` has no children.

```
(define (level depth Tree)
```

Your name _____ login cs61a–_____

Question 12 (8 points):

Cy D. Fect is fed up with programming using recursion and begs you to add the `while` special form to the metacircular evaluator. You know recursion is just as powerful as iteration (if not more so) but just to please him, you've decided to do it.

The `while` special form takes at least two arguments; the first is a test expression and the rest are actions. It will evaluate all of the actions as long as the value of the test expression is true. For example:

```
> (define x 5)
> (while (> x 0)
         (display x)
         (newline)
         (set! x (- x 1)))
5
4
3
2
1
okay
```

Add the `while` special form to the metacircular evaluator. If the test expression initially evaluates to false, the actions should not be run at all. The return value is unimportant.

Some possibly relevant metacircular evaluator procedures are listed on the next page. **On this page, write the names of any procedures that you modify on the next page.** New procedures go on this page.

```
(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 (eval-if exp env)
  (if (true? (mc-eval (if-predicate exp) env))
      (mc-eval (if-consequent exp) env)
      (mc-eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (mc-eval (first-exp exps) env))
        (else (mc-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

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