CS 61A Final Exam — May 16, 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 18 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.

____

Question	Score
1	/4
2–3	/7
4–5	/7
6	/7
7–8	/7
9	/3
10	/3
11–12	/8
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? Also, **<u>draw a "box and pointer" diagram</u> for the result of each printed expression.** If any expression results in an error, just write "ERROR"; you don't have to give the precise message. Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((foo (list 'a 'b 'c))
      (bar (list 'x 'y 'z)))
  (set-car! (cdr foo) (cadr bar))
  (set-car! (cddr foo) (cdr bar))
  foo)
```

```
(let ((foo (list 'a 'b 'c))
      (bar (list 'x 'y 'z)))
  (set-cdr! (cddr bar) (cdr foo))
  (set-cdr! (cdr bar) (car foo))
  bar)
```

Your name ____ login cs61a–_______

Question 2 (3 points):

```
(define (foo x y) (+ x (* y x)))

(foo (* 2 2) (+ 8 4))
```

How many times are + and * called from the invocation of `foo` above...

in applicative order? + invocations _______ * invocations _______

in normal order without memoization? + invocations _______ * invocations _______

in normal order with memoization? + invocations _______ * invocations _______

Question 3 (4 points):

Mark whether each of the following generate an iterative or recursive process by circling the appropriate word, and fill in the order of growth:

```
(define (foo x)                  ; Assume factorial is Θ(n)
  (define (helper n)
    (if (= n x)
        0
        (* (factorial n) (helper (+ n 1)))))
  (helper 0))

;   ITERATIVE  /  RECURSIVE        Θ(_______)
```

```
(define (mystery sent)
  (define (help sent result)
    (if (empty? sent)
        result
        (if (member? (first sent) (help (bf sent) result))
          (help (bf sent) (se (first sent) result))
          (help (bl sent) (se result (last sent))))))
  (help sent '()))

;   ITERATIVE  /  RECURSIVE        Θ(_______)
```

Question 4 (3 points):

```
(define (transform1 f)
   (lambda (x) (word (f x) (f x))))

(define (transform2 f)
   (lambda (x) (f (word x x))))

(define (transform3 f)
   (lambda (x) (f (f x))))

(define f1 (transform1 count))
(define f2 (transform2 count))
(define f3 (transform3 count))
```

Fill in the blanks with what Scheme returns. (Remember that numbers are words too.)

```
(f1  'hello)  =>  ______________

(f2  'bye)    =>  ______________

(f3  'abcdefghijklmnopqrstuvwxyz)    =>  ______________
```

Question 5 (4 points):

```
(define (startpoint segment) (car segment))
(define (endpoint segment) (cdr segment))
(define (x-coord point) (car point))
(define (y-coord point) (cdr point))
(define (datum tree) (car tree))
(define (children tree) (cdr tree))

(define (slope segment)
  (/ (- (cddr segment) (cdar segment))
     (- (cadr segment) (caar segment))))
```

Rewrite `slope` to fix the data abstraction violations.

Your name ____ login cs61a–_______

Question 6 (7 points):

We are going to add buses to the adventure game. A bus can carry people from one place to another. Here's an example (assuming nasty and hacker are at telegraph):

```
> (define fifty-one (instantiate bus telegraph-ave))
> (ask fifty-one 'load nasty)
> (ask fifty-one 'load hacker)
> (ask fifty-one 'go 'south)
> (ask fifty-one 'unload nasty)
> (whereis nasty)
noahs
```

When the bus moves somewhere, everybody on it should move as well. Implement the bus class. The relevant code from the adventure game is provided on the following pages. HINT: Each of these methods should require very little work from you!

```
(define-class (bus place)
  (initialize


  (instance-vars


  (class-vars


  (method (load person)




  (method (unload person)





  (method (go direction)





      ))
```

```
(define-class (thing name)
  (parent (basic-object))
  (instance-vars (possessor 'no-one))
  (method (thing?) #t)
  (method (type) 'thing)
  (method (change-possessor new-possessor)
    (set! possessor new-possessor))
  (method (may-take? who)
    (cond ((eq? possessor 'no-one) self)
          ((> (ask who 'strength) (ask possessor 'strength))
           self)
          (else #f)))
  )


(define-class (place name)
  (parent (basic-object))
  (instance-vars
   (directions-and-neighbors '())
   (things '())
   (people '())
   (entry-procs '())
   (exit-procs '()))
  (method (place?) #t)
  (method (type) 'place)
  (method (neighbors) (map cdr directions-and-neighbors))
  (method (exits) (map car directions-and-neighbors))
  (method (look-in direction)
    (let ((pair (assoc direction directions-and-neighbors)))
      (if (not pair)
          '()                     ;; nothing in that direction
          (cdr pair))))           ;; return the place object
  (method (appear new-thing)
    (if (memq new-thing things)
        (error "Thing already in this place" (list name new-thing)))
    (set! things (cons new-thing things))
    'appeared)
  (method (enter new-person)
    (if (memq new-person people)
        (error "Person already in this place" (list name new-person)))
    (set! people (cons new-person people))
    (for-each (lambda (proc) (proc)) entry-procs)
    (for-each (lambda (pers) (ask pers 'notice new-person))
              (delete new-person people))
    'appeared)
```

```
  (method (gone thing)
    (if (not (memq thing things))
        (error "Disappearing thing not here" (list name thing)))
    (set! things (delete thing things))
    'disappeared)
  (method (exit person)
    (for-each (lambda (proc) (proc)) exit-procs)
    (if (not (memq person people))
        (error "Disappearing person not here" (list name person)))
    (set! people (delete person people))
    'disappeared)
  ; ... ... ... ... ... (some methods omitted)
 )

(define-class (person name place)
  (parent (basic-object))
  (instance-vars
   (possessions '())
   (saying "")
   (money 100))
  (initialize
   (ask self 'put 'strength 100)
   (ask place 'enter self))
  (method (person?) #t)
  (method (type) 'person)

  (method (lose thing)
    (set! possessions (delete thing possessions))
    (ask thing 'change-possessor 'no-one)
    'lost)

  (method (take-all)
    (for-each (lambda (thing) (ask self 'take thing))
              (filter (lambda (thing)
                        (eq? (ask thing 'possessor) 'no-one))
                      (ask place 'things) ) ))

  (method (go-directly-to new-place)
    (announce-move name place new-place)
    (for-each (lambda (p)
                (ask place 'gone p)
                (ask new-place 'appear p))
              possessions)
    (ask place 'exit self)
    (set! place new-place)
    (ask new-place 'enter self))
```

```
(method (take thing)
    (cond ((not (thing? thing)) (error "Not a thing" thing))
          ((not (memq thing (ask place 'things)))
           (error "Thing taken not at this place"
                  (list (ask place 'name) thing)))
          ((memq thing possessions) (error "You already have it!"))
              (else
               (let ((reply (ask thing 'may-take self)))
                 (if reply
                     (begin
                      (announce-take name reply)
                      ;; add it to my possessions
                      (set! possessions (cons reply possessions))
                      ;; go through all the people at the place
                      ;; if they have the object we are taking
                      ;; make them lose it and have a fit
                      (for-each
                       (lambda (pers)
                         (if (and (not (eq? pers self)) ; ignore myself
                                  (memq reply (ask pers 'possessions)))
                             (begin
                              (ask pers 'lose reply)
                              (have-fit pers))))
                       (ask place 'people)))

                     ;; actually change the possessor
                     (ask reply 'change-possessor self)
                     'taken)
                 (announce-too-weak name thing) ))))

  (method (exits) (ask place 'exits))
  (method (notice person) (ask self 'talk))
  (method (go direction)
    (let ((new-place (ask place 'look-in direction)))
      (cond ((null? new-place)
             (error "Can't go" direction))
            ((not (ask new-place 'may-enter? self))
             (error "can't enter locked place" (ask new-place 'name)))
            (else
             (ask place 'exit self)
             (announce-move name place new-place)
             (for-each
              (lambda (p)
                (ask place 'gone p)
                (ask new-place 'appear p))
              possessions)
             (set! place new-place)
             (ask new-place 'enter self)))))

  ; ... ... ... ... ... (some methods omitted)
)
```

Your name ____ login cs61a–________

Question 7 (3 points):

```
(define wd 'foo)
(define s (make-serializer))
(define (f) (set! wd (word wd 'd)))
(define (g) (set! wd (word wd wd)))
```

(a) List all the possible values of `wd` after `(parallel-execute (s f) (s g))`.

(b) List all the possible values of `wd` (starting from its initial value, without part (a)'s effect) after `(parallel-execute f g)`.

Question 8 (4 points):

Consider the following stream procedure:

```
(define (process-stream S)
  (cons-stream (stream-car S)
               (stream-filter (lambda (x) (not (equal? x (stream-car S))))
                              (process-stream (stream-cdr S)))))
```

(a) In one sentence, what function does `process-stream` compute?

(b) We also have these streams:

```
(define ones (cons-stream 1 ones))
(define twos (cons-stream 2 twos))
(define integers (cons-stream 1 (stream-map + ones integers)))
```

Remember that `(ss stream 10)` displays the first 10 elements of a stream. What does scheme print for each of the following?

```
> (ss (process-stream (interleave ones integers)) 10)
```

```
> (ss (process-stream (interleave ones twos)) 10)
```

Question 9 (3 points):

```
(define mystery                              ; procedure A
  (let ((y 2))
    (lambda (z)
      (let ((x 4))
        (lambda ()
          (set! x (+ z y))
          (set! y (+ x z))
          y)))))
(define foo (mystery 3))
```

```
(define mystery                              ; procedure B
  (lambda (z)
    (let ((x 4)
          (y 2))
      (lambda ()
          (set! x (+ z y))
          (set! y (+ x z))
          y))))
(define foo (mystery 3))
```

```
(define mystery                              ; procedure C
  (lambda (z)
    (lambda ()
      (let ((x 4)
            (y 2))
        (set! x (+ z y))
        (set! y (+ x z))
        y))))
(define foo (mystery 3))
```

On the next page are four environment diagrams. Label three of them A, B, and C, corresponding to the procedures above (write the letter, large, in the global environment box); leave the fourth diagram unlabelled.

Question continued on next page.

Question 9 continued:

Question 10 (3 points):

For each of the following code sequences, will the analyzing evaluator be faster than the ordinary metacircular evaluator? Assume the evaluator is restarted for each sequence.

(a)

```
(define (member thing ls)
  (cond
   ((null? ls) #f)
   ((equal? thing (car ls)) ls)
   (else (member thing (cdr ls)))))

(member 2000 (enumerate-interval 0 1990))
```

Is (a) faster in the analyzing evaluator? _______Yes. _______No.

(b)

```
(define (member thing ls)
  (cond
   ((null? ls) #f)
   ((equal? thing (car ls)) ls)
   (else (member thing (cdr ls)))))

(member 0 (enumerate-interval 0 1990))
```

Is (b) faster in the analyzing evaluator? _______Yes. _______No.

(c)

```
(define (cube x)
  (* x x x))

(cube (+ 2 3))
```

Is (c) faster in the analyzing evaluator? _______Yes. _______No.

Your name ____ login cs61a–_______

Question 11 (4 points):

Write a `mapreduce` program to find how often each letter occurs in a document. (You should use the procedure `strip-punctuation`, which takes a word and removes all punctuation from it. You don't have to write that procedure.)

Question 12 (4 points):

The following is run in the nondeterministic evaluator. Fill in the blanks.

```
(define z 7)

(define (indecisive x y)
  (if (amb x (not x))
      (begin
        (set! z 2)
        (cons z (amb y (+ y 3))))
      (cons z (+ y 1))))

(indecisive #t 30)
```

____

`try-again`

____

`try-again`

____

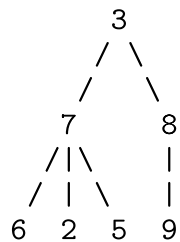
`try-again`

____

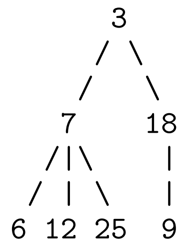
Question 13 (8 points):

This question is about the Tree abstract data type, with selectors `datum` and `children`.

Write a procedure `sib-map` that works like `tree-map`, taking a function f and a Tree as arguments and returning a new Tree. The difference is that f is a function of two arguments: the datum and the *sibling number*, which is 0 for the leftmost child of a node, 1 for the next child, and so on. If `my-tree` is the following tree:



`(sib-map (lambda (elt posn) (+ (elt (* 10 posn)))) my-tree)` would give



Your name ____ login cs61a–_______

Question 14 (8 points):

Write a query system rule or rules for `next-to`, a relation among two names and a list, which can determine whether two names are next to each other in the list. For example:

```
(Jerry is next to Evan in (Albert Brian Evan Jerry Ramesh Yaping))
```

matches the rule, and the query

```
(?who is next to Evan in (Albert Brian Evan Jerry Ramesh Yaping))
```

yields

```
(Jerry is next to Evan in (Albert Brian Evan Jerry Ramesh Yaping))
(Brian is next to Evan in (Albert Brian Evan Jerry Ramesh Yaping))
```

Do not use `lisp-value`!

Question 15 (8 points):

We want to add *optional* dynamic scope to the metacircular evaluator. Add a new special form `(lambda-dynamic (arg ...)  ...)`, which creates a "dynamic procedure." When a dynamic procedure is invoked, the frame for that procedure should extend the current environment instead of the procedure's creation environment.

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 18.

```
(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-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (mc-eval (assignment-value exp) env)
                       env)
  'ok)

(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 (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))))
```