CS 61A Final Exam — May 15, 2009

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.

_____

1–2	/9
3–4	/6
5	/4
6	/4
7	/7
8	/6
9	/6
10–11	/4
12	/6
13	/6
14	/6
15	/6
total	/70

Question 1 (4 points):

```
(define (if-false x y)
  (if x 'true y))
```

For each of the following expressions, state the result in applicative order and in normal order. If the expression results in an error, just say ERROR; you don't have to give the exact message.

```
(if-false (/ 33 0) (/ 33 1))
```

Applicative: _____ Normal: _____

```
(if-false (/ 33 1) (/ 33 0))
```

Applicative: _____ Normal: _____

Question 2 (5 points):

Define a function `maxnum` that takes a list as argument. The elements of the list might or might not include numbers, along with other values. If any elements are numbers, return the largest number. If not, return `#f`. If an element is a list, don't look inside it; consider only top-level elements.

```
> (maxnum '(20 minutes and 45 seconds after 10 (2009 5 15)))
45
```

Use higher-order functions, not recursion!

Note: You get 4 points if your function works only for positive numbers.

Your name _____ login cs61a–_____

Question 3 (4 points):

Here are two procedures that use two different algorithms to solve the same problem: Given two lists of length n as arguments, return true if they have any elements in common, false if not. For each procedure, indicate the order of growth in time, and whether the procedure generates an iterative or a recursive process.

```
(define (common? ls1 ls2)
  (cond ((null? ls1) #f)
        ((null? ls2) #f)
        ((equal? (car ls1) (car ls2)) #t)
        (else (or (common? ls1 (cdr ls2))
                  (common? (cdr ls1) ls2)))))
```

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

____Iterative ____Recursive

```
(define (common? ls1 ls2)
  (cond ((null? ls1) #f)
        ((member (car ls1) ls2) #t)
        (else (common? (cdr ls1) ls2))))
```

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

____Iterative ____Recursive

Question 4 (2 points):

```
(define my-stream
  (cons-stream 3 (cons-stream 4 (add-streams my-stream
                                             (stream-cdr my-stream)))))

> (show-stream my-stream 5)
```

_____ _____ _____ _____ _____

Question 5 (4 points):

Write a procedure `sumpath` that takes a Tree (with `datum` and `children`) of numbers as its argument, and returns a tree of the same shape in which each datum is the sum of all the data between the original datum and the root. For example, in the case below the value 5 in the original tree becomes 8 $(1 + 2 + 5)$ in the returned tree:

Your name _____ login cs61a–_____

Question 6 (4 points):

(a) Fill in the blanks in the interactions below:

```
(define (swap ls1 ls2)
  (let ((temp ls1))
    (set! ls1 ls2)
    (set! ls2 temp) ) )

STk> (define foo (list 1 2 3))
foo
STk> (define bar (list 'a 'b 'c))
bar
STk> (swap foo bar)
okay
STk> foo
```

_____

```
STk> bar
```

_____

(b) Fill in the blank below:

```
STk> (define x 1)
STk> (define y x)
STk> (set! x 2)
STk> y
```

_____

Question 7 (7 points):

We want to use OOP language (with `define-class`) to simulate a "smartphone" that combines a cell phone with a phonebook, calendar, etc. Assume that you are given a `cellphone` class, representing a bare-bones cell phone, that accepts a `dial` message with a phone number as argument.

```
> (define dumb-phone (instantiate cellphone))
> (ask dumb-phone 'dial '555-2368)
; ... phone dials
```

(a) Define a `contact` class that is instantiated with a person's name, address, and phone number, and accepts messages `name`, `address`, and `phone` that return the corresponding values:

```
> (define bh (instantiate contact 'Brian '(781 Soda) '642-8311))
> (ask bh 'phone)
642-8311
```

This question continues on the next page.

Your name _____ login cs61a–_____

Question 7 continued:

(b) Define a `smartphone` class that takes no instantiation arguments. The message `add-contact` takes three arguments for name, address, and phone number; it creates a `contact` object and adds it to its list of contacts. The message `call` takes a name as argument, finds the name in the list of contacts, and dials the corresponding phone number.

```
> (define my-phone (instantiate smartphone))
> (ask my-phone 'add-contact 'Dan '(777 Soda) '642-9595)
> (ask my-phone 'add-contact 'Brian '(781 Soda) '642-8311)
> (ask my-phone 'add-contact 'Mike '(779 Soda) '642-7017)
> (ask my-phone 'call 'Brian)
; ... phone dials 642-8311 (using dial method) ...
> (ask my-phone 'call 'Alonzo)
"Name not found"
```

Question 8 (6 points):

What will the Scheme interpreter print in response to each of the following expressions? Also, **draw a "box and pointer" diagram 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 ((x (list 'a 'b 'c)))
 (set-cdr! (cdr x) 3)
 x)
```

```
(let ((x (list 'a 'b 'c)))
 (set-car! x (caddr x))
 (set-car! (cddr x) (car x))
 x)
```

```
(let ((x (list 'a 'b 'c)))
 (set! (car x) (caddr x))
 x)
```

Your name _____ login cs61a–_____

Question 9 (6 points):

```
(define x 100)
(define y 10)
(define s (make-serializer))
(define t (make-serializer))
(parallel-execute (s (lambda () (set! x (+ x y))))
                  (t (lambda () (set! y (* x y)))))
(list x y)
```

What are the possible correct answers for `(list x y)`?

What are the additional possible incorrect answers, if any?

Is deadlock possible? ____Yes ____No

```
(define x 100)
(define y 10)
(define s (make-serializer))
(define t (make-serializer))
(parallel-execute (s (t (lambda () (set! x (+ x y)))))
                  (s (t (lambda () (set! x (* x y)))))) ; x this time!
(list x y)
```

What are the possible correct answers for `(list x y)`?

What are the additional possible incorrect answers, if any?

Is deadlock possible? ____Yes ____No

Question 10 (2 points):

For each of the following interactions, indicate whether it will execute faster in the analyzing evalutor than in the original metacircular evalutor, or the same speed. Circle Faster or Same for each.

```
(define (mystery n)
  (if (< n 1)
      1
      (* n (mystery (/ n 2)))))

> (mystery 100)
```

Analyzing will be: Faster Same

```
(define (f x)
  (* (+ x 30) 7))

(define (g x)
  (* x x))

(define (h x)
  (/ (+ (* x 2) 10) 2))

(define (fgh n)
  (f (g (h n))))

> (fgh 100)
```

Analyzing will be: Faster Same

Question 11 (2 points):

The following interaction is run in the *lazy* evaluator:

```
(define (darrentron a b c)
    (if a (* a b) c))

> (darrentron (* 1 2) (* 3 4) (* 1 2))
```

How many times is * called if promises are memoized? _____

How many times is * called if promises are *not* memoized? _____

Your name _____ login cs61a–_____

Question 12 (6 points):

This question concerns the *nondeterministic* (`amb`) evaluator.

(a) Write a procedure `first-n` that takes a list `lst` and a nonnegative integer `num` as argument, returning the first `num` elements of `lst`, or failing if there aren't enough elements.

(b) Using your answer to part (a), write a procedure `sublist-of-length` that takes a list and a nonnegative integer, and returns *any* sublist of `num` consecutive elements (not necessarily the first `num` elements). Each use of `try-again` should give another sublist, until there are no more of the required length.

Question 13 (6 points):

Justin wants to improve his score in Rock Band, so he wants to find the song with the lowest average number of words per phrase. The file `rock-band-songs` has key-value pairs with a song title as the key and a phrase from the song as the value, like this:

```
(limelight . (living on a lighted stage))
(limelight . (approaches the unreal))
...
```

(a) Write a `mapreduce` call that will turn that file into a stream of key-value pairs, one per song, in which the value for each song is a list of two numbers: the total number of words in all the phrases of the song, and the number of phrases in the song. Call the returned stream `countstream`.

This question continues on the next page.

Your name _____ login cs61a–_____

Question 13 continued.

(b) Now write a `mapreduce` call that takes the stream from part (a) and returns a stream with song titles and average number of words per phrase for that song, sorted by the average, so that `stream-car` of the result will represent the song with the smallest average phrase length. Hint: The result from `mapreduce` is a stream that is sorted in *key* order; the smallest key comes first.

Question 14 (6 points):

Write a set of rules for the query evaluator to implement the `subst` relation, with components `old`, `new`, `input`, and `output`. The first two are atoms; the last two can be atoms or (possibly deep) lists. The `output` should be the same as `input` except that every appearance of `old`, at any depth, is replaced by `new`:

```
;; query input:
(subst romeo fred (romeo (oh romeo) why art thou romeo) ?x)

;; query result:
(subst romeo fred (romeo (oh romeo) why art thou romeo)
                  (fred (oh fred) why art thou fred))

;; query input:
(subst romeo fred romeo ?x)

;; query result:
(subst romeo fred romeo fred)
```

Do not use `lisp-value`!

Your name _____ login cs61a–_____

Question 15 (6 points):

We want to add a new special form called `let!` to the metacircular evaluator. It takes a list of bindings, like `let`, but has no body. It doesn't make a new environment; instead, it changes the current environment. If there is already a binding for a variable named in the `let!` expression, then its value is changed, as in `set!`. If there isn't a binding, a new one is made in the current frame, as in `define`.

```
> (define count 1)
> (define (foo)
    (let! ((oldcount count) (count (+ count 1))))
    (* oldcount count))
> (foo)
2
> (foo)
6
> count
3
> oldcount
ERROR -- unbound variable: oldcount
```

The last expression is an error because `oldcount` was defined in `foo`'s local environment, not in the global 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 the back 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 (definition? exp)
  (tagged-list? exp 'define))

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

Your name _____ login cs61a–_____

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

(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))))

(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```