CS 61A Midterm #2 — October 15, 2008

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 six 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 eight 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 especially 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.

____

Question	Score
0	/1
1	/4
2	/6
3	/6
4	/7
5	/8
6	/8
total	/40

Question 1 (4 points):

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

```
> (cons (list 'a '(b)) (cons (cons 'c 'd) (cons (list 'e) (list 'f))))
```

____

Draw your box and pointer diagram here:

Your name ____ login cs61a–_______

Question 2 (6 points): We want to define an automobile class. Here's an example of how it works:

```
> (define civic (instantiate automobile 33)) ; Car gets 33 miles per gallon.
> (ask civic 'add-gas 9) ; add 9 gallons.  Assume unlimited tank capacity.
done
> (ask civic 'go 99) ; Drive 99 miles.  Uses 99/33=3 gallons.
done
> (ask civic 'odometer) ; Odometer starts at 0, records total miles travelled.
99
> (ask civic 'gas) ; How much gas left?
6
> (ask civic 'go 400) ; Too far to travel with 6 gallons in tank.
(you do not have enough gas to get there!)
```

These are the messages to which your objects must respond: `add-gas`, `go`, `odometer`, and `gas`.

Complete this class definition:

```
(define-class (automobile mpg)
```

Question 3 (6 points):

Check all (and only) the calls to `eval-1` and `apply-1`, with their arguments, that occur when Scheme-1 evaluates the following expression:

```
((lambda (x) (if (> x 0) 1 -1)) 2)
```

Some of these calls might happen more than once.

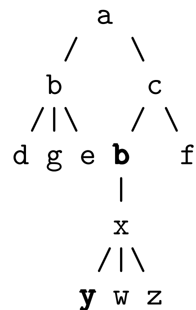
(Note: In the calls to `apply-1`, we are representing procedures that are used as arguments by the expression that was evaluated to get the procedure, since procedures themselves have no real printable representation.)

```
________  (eval-1 '((lambda (x) (if (> x 0) 1 -1)) 2))
________  (eval-1 '(lambda (x) (if (> x 0) 1 -1)))
________  (eval-1 'lambda)
________  (eval-1 '(x))
________  (eval-1 'x)
________  (eval-1 2)
________  (eval-1 '(if (> x 0) 1 -1))
________  (eval-1 '(if (> 2 0) 1 -1))
________  (eval-1 '(> x 0))
________  (eval-1 '(> 2 0))
________  (eval-1 '>)
________  (eval-1 0)
________  (eval-1 1)
________  (eval-1 -1)
________  (apply-1 lambda '((x) (if (> x 0) 1 -1)))
________  (apply-1 (lambda (x) (if (> x 0) 1 -1)) 2)
________  (apply-1 (lambda (x) (if (> x 0) 1 -1)) '(2))
________  (apply-1 if '((> 2 0) 1 -1))
________  (apply-1 if '(#t 1 -1))
________  (apply-1 > '(x 0))
________  (apply-1 > '(2 0))
________  (apply-1 > 2 0)
```

Your name ____ login cs61a–_______

Question 4 (7 points):

Write `ancestor?`, a procedure that takes three arguments: two values and a Tree. It should return `#t` if the second value occurs as a datum anywhere within a subtree of the Tree, at any depth, whose root datum is the first value. For example, in the Tree



the value `b` is an ancestor of `y`, even though it's not the leftmost or topmost `b` that has a `y` under it. But `d` is not an ancestor of `y`.

You may use, without writing it, the function `tree-member?`, which takes a datum and a Tree as its arguments, returning `#t` if the datum appears anywhere in the Tree, or `#f` if not.

Question 5 (8 points):

We want to use data-directed programming to implement abstract data types. As it is, for every new data type we have to write a constructor and as many selectors as the type has fields, like this:

```
(define (make-family oldest middle youngest)
  (list oldest middle youngest))
(define oldest car)
(define middle cadr)
(define youngest caddr)
```

We're going to automate this process, so ADTs work this way:

```
> (make-adt 'family '(oldest middle youngest))          ; create the ADT
okay

> (define f1 (make-oneof 'family '(yakko wakko dot)))  ; make an instance
f1

> (f1 'middle)                                          ; use name as selector
wakko
```

The implementation must use data-directed programming using the `get/put` **table just as in the text and lectures.** That is, after the call to `make-adt` above, there will be table entries more or less equivalent to what we'd have gotten if we'd said

```
(put 'family 'oldest car)
(put 'family 'middle cadr)
(put 'family 'youngest caddr)
```

The procedure representing a particular `family` instance, such as `f1`, will look up its argument in the table to know what function to apply to the saved data for that instance.

Hint: You might want to use `list-ref`, although that's not the only good solution.

This question continues on the next page.

Your name ____ login cs61a–_______

Question 5 continued:

(a) Write `make-adt`. It takes an ADT name and a list of field names, and puts appropriate entries in the table. (Don't assume there are three field names! That was just an example.)

(b) Write `make-oneof`. It takes an ADT name and a list of values as arguments, and returns a specific instance of the ADT, as described earlier.

Question 6 (8 points):

Write a procedure `semirev` that's like `deep-reverse` except that it only reverses every other level of the deep list. That is, it reverses the elements of the list; it does *not* reverse the elements of the elements; it *does* reverse the elements of the elements of the elements; and so on. Remember that the list may not be of uniform depth, e.g., it might be `(a (b c))`.

```
STk> (semirev '( ( ((a b c) (d e f)) ((g h i) (j k l)) )
                 ( ((m n o) (p q r)) ((s t u) (v w x)) ) ))
((((p q r) (m n o)) ((v w x) (s t u)))
 (((d e f) (a b c)) ((j k l) (g h i))))
```

In the example above, the `a-l` sublist comes after the `m-x` sublist in the result, so the elements of the overall list are reversed. But the `a-f` sublist comes *before* the `g-l` sublist, so the elements of the elements aren't reversed. The `a-c` sublist comes after the `d-f` sublist, so the third-level elements are reversed; but `a` comes before `c`, so the fourth-level elements aren't.

You may assume that you have the `reverse` function for regular lists without writing it.

Hint: My solution uses a helper procedure, and also uses `map`.