CS 61A Midterm #2 – March 7, 2006

Name:
Login:
Discussion section number:
TA:

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.

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 other ones you find easier.

Question 1 (8 points): What will Scheme print in response to the following expressions? If an 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 each expression.**

```
(cons (list '(a) '(b)) (append '(c) '(d)))

(car (cons '((portia)) '(black . satin)))

(caadr '( ((a b c) (d e f)) ((g h i) (j k l)) ((m n o) (p q r)) ))

(filter (lambda (x) (if (list? x) (pair? x) (number? x)))
                    '(1 () (2 3) (so) what))
```

Question 2 (4 points): The procedure below takes a Tree (as defined in lecture) as argument. Each `datum` in the tree is a sentence. The procedure returns a tree in which each `datum` is the `first` of the corresponding `datum` of the argument tree.
The procedure has some data abstraction violations. Find and fix them.

```
(define (firsts tree)
 (cons (car (car tree))
       (helper (cdr tree))))
(define (helper x)
  (if (null? x)
      '()
      (cons (firsts (car x))
            (helper (cdr x)))))
```

Question 3 (6 points):

For reference, here are the central procedures of `scheme-1`, with the lines numbered:

```
 1 (define (eval-1 exp)
 2   (cond ((constant? exp) exp)
 3         ((symbol? exp) (eval exp))
 4         ((quote-exp? exp) (cadr exp))
 5         ((if-exp? exp)
 6          (if (eval-1 (cadr exp))
 7              (eval-1 (caddr exp))
 8              (eval-1 (cadddr exp))))
 9         ((lambda-exp? exp) exp)
10         ((pair? exp) (apply-1 (eval-1 (car exp))
11                               (map eval-1 (cdr exp))))
12         (else (error “bad expr: “ exp))))

13 (define (apply-1 proc args)
14   (cond ((procedure? proc)
15          (apply proc args))
16         ((lambda-exp? proc)
17          (eval-1 (substitute (caddr proc)
18                              (cadr proc)
19                              args
20                              ‘())))
21         (else (error “bad proc: “ proc))))
```

For each of the following, say whether it would be done by changing `eval-1` (or its sub-procedures), or by changing `apply-1` (or its sub procedures).

(a) Infix arithmetic: `(2 + 3)` instead of `(+ 2 3)`

_____`eval-1` _____`apply-1`

(b) Evaluate more than one expression in the body of a procedure
e.g., `(lambda (x y) (print x) (print y))`

_____`eval-1` _____`apply-1`

(c) Allow defining global variables

_____`eval-1` _____`apply-1`

Question 4 (5 points):

Some languages, as an alternative to message passing, implement what they consider to be object-oriented programs using *overloaded functions*, which means that different functions can have the same name, if they have different numbers of arguments, or the arguments are of different types. If Scheme worked that way, you'd be able to say, for example,

```
(define (+ (rational x) (rational y))
  (make-rat (+ (* (numer x) (denom y)) (* denom x) (numer y)))
            (* (denom x) (denom y))))

(define (+ (complex x) (complex y))
  (make-complex (+ (real-part x) (real-part y))
                (+ (imag-part x) (imag-part y))))
```

and then when you use the + function name, Scheme would check the type tags of the arguments and pick the right function to match the data.

You're going to implement this idea using data-directed programming, but since the name `define` is already used for Scheme's ordinary functions, we'll use a different notation. To define an overloaded function, you'll say

```
(define-overloaded '+
  '(rational rational)
  (lambda (x y)
    (make-rat (+ (* (numer x) (denom y)) (* (denom x) (numer y)))
              (* (denom x) (denom y)))))
```

and to use an overloaded function you'll say, for example,

```
(overloaded '+ (make-rat 2 3) (make-rat 4 5))
```

So the arguments to `define-overloaded` are the function name, the type signature, and the procedure to call. The arguments to `overloaded` are the function name and the arguments to that function. Assume, in the example above, that we've defined the version of make-rat that attaches a type tag.

Write `define-overloaded` and `overloaded`.

Question 5 (8 points):

We want a procedure `deepen` that takes a deep list (A list of lists of …) as its argument and returns a list that's like the argument except that each atom (non-list) is replaced by a one-element list containing that atom:

```
> (deepen '(a (b c) (d) (e (f g) h)))
((a) ((b) (c)) (d) ((e) ((f) (g)) (h)))
```

(a) Write `deepen` using `car/cdr` recursion.

(b) Write deepen using `map` for the recursion.

Question 6 (8 points):

Write `siblings?`, a predicate that takes three arguments: a tree and two datum values. It returns `#t` if those two data are siblings in the tree (that is, if they have the same parent) and `#f` otherwise. You can assume that all elements only appear once in the tree.