CS 61A Midterm #2 — February 28, 1994

Your name _____

login cs61a–_____

Discussion section number _____

TA's name _____

This exam is worth 20 points, or about 11.5% of your total course grade. The exam contains four 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 five 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.

CS 3 alumni please note: Don't use the CS-3-only higher order functions (every, keep, accumulate) in these problems!!

0	/1
1	/4
2	/5
3	/5
4	/5
total	/20

Question 1 (4 points):

What will Scheme print in response to the following expressions? Also, draw a "box and pointer" diagram for the result of each expression:

```
(list '(2 3) '(4 5))
```

```
(cons (list 2 3) 4)
```

```
(cddadr '((a b c d e) (f g h i j) (l m n o p) (q r s t u)))
```

```
(cons (cdr '(a)) (cdr '(b)))
```

Your name _____ login cs61a–_____

Question 2 (5 points):

(a) Using the binary tree abstract data type as defined on page 115 of the text (with selectors `entry`, `left-branch`, and `right-branch` and constructor `make-tree`), write the predicate `all-smaller?` that takes two arguments, a binary tree of numbers and a single number, and returns `#t` if every number in the tree is smaller than the second argument. Examples:

```
> (define my-tree (make-tree 8 (make-tree 5 '() '())
                               (make-tree 12 '() '())))
> (all-smaller? my-tree 15)
#T
> (all-smaller? my-tree 10)
#F
```

(This question continues on the next page.)

Question 2 continued:

(b) Using `all-smaller?` and, if you wish, a similar `all-larger?` (which you don't have to write), write a predicate `bst?` that takes a binary tree of numbers as its argument, returning `#t` if and only if the tree is a binary *search* tree. (That is, your procedure should return true only if, at every node, all of the numbers in that node's left branch are smaller than the entry at the node, and all of the numbers in the node's right branch are larger than the entry.)

Your name _____ login cs61a–_____

Question 3 (5 points):

We are creating a database of the greatest songs in the world. The first step is to define an abstract data type for a song:

```
(define title car)
(define artist cadr)
(define make-song list)
```

Now we set up a global variable `great-songs` whose value is a list of songs:

```
(define great-songs (list (make-song '(she loves you) '(the beatles))
                          (make-song '(waterloo sunset) '(the kinks))
                          (make-song '(pictures of lily) '(the who))
                          (make-song '(davy the fat boy) '(randy newman))
                          (make-song '(expecting to fly)
                                     '(buffalo springfield))
                          (make-song '(tell her no) '(the zombies))))
```

Your job is to write a procedure `who-sang` that takes a song title as its argument and returns the corresponding artist, or `#f` if the song isn't one of the greatest in the world:

```
> (who-sang '(waterloo sunset))
(THE KINKS)

> (who-sang '(stairway to heaven))
#F
```

Respect the data abstraction.

Question 4 (5 points):

We want to combine the techniques of data-directed programming and message-passing as follows: Instead of using a symbol like `complex` as a manifest type tag, we'll use a list of messages and their associated methods, as in the following example.

```
> (define complex-methods (list (cons 'add +complex)
                                (cons 'sub -complex)
                                (cons 'mul *complex)
                                (cons 'div /complex)))
> (define (make-complex z)
    (attach-type complex-methods z))
```

Your job is to rewrite `operate-2` (from page 144) to work with this new system instead of using a table of operators and types. If any other procedures must be changed, change them too. **You may leave out the error checks.** For your convenience, here is the book's `operate-2`, without its error checks:

```
(define (operate-2 op arg1 arg2)
  (let ((t1 (type arg1)))
    (let ((proc (get t1 op)))
      (proc (contents arg1) (contents arg2)))))
```