CS 61A Midterm #2 — October 17, 2001

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 5 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 seven 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.

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.

____

0	/1
1	/8
2	/8
3	/8
4	/7
5	/8
total	/40

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, <u>draw a box and pointer diagram</u> for the value produced by each expression.**

```
(cons (list 1 2) 4)
```

```
(cons (caddr '(a b c d e)) (list 3 4))
```

```
(append (cons 3 '()) (list 4 5))
```

```
(list (cons 1 2) (list 3 4))
```

Your name ______________________________ login cs61a-______

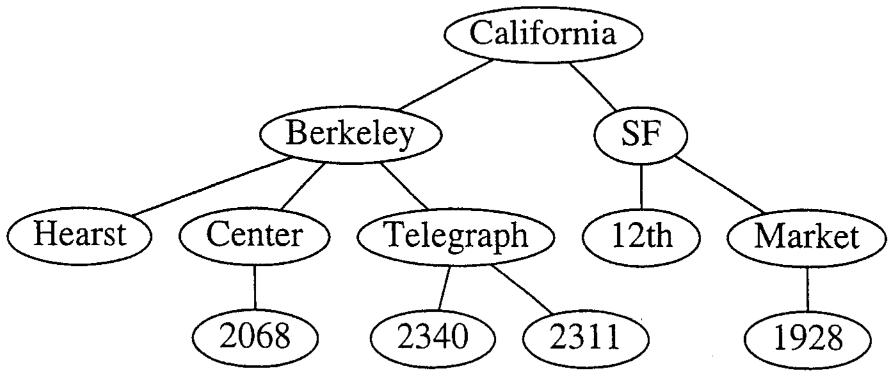
Question 2 (8 points):

This question relates to trees structured like the world tree discussed in lecture, defined using the constructor `make-tree` and selectors `datum` and `children`.

The `address-in-tree?` procedure takes as arguments a tree and an *address*. An address is a list of the contents of nodes. The procedure should return true if the datum at the root of the tree is the same as the first element of the address, and there exists a child whose datum is the second element of the address, and so on. If the address is not in the tree, your procedure should return `#f`.

For example, given the tree `ca-tree` shown below, with words as the data:

```
> (address-in-tree? ca-tree '(california berkeley))
#t
> (address-in-tree? ca-tree '(california sf market 1928))
#t
> (address-in-tree? ca-tree '(california berkeley shattuck))
#f
```



Write `address-in-tree?`, **using mutual recursion with a helper procedure named** `address-in-forest?` **and *not* using higher-order procedures.**

Question 3 (8 points):

Given a database of prices for various things, we'd like to be able to take a shopping list (3 apples, 2 quarts of milk, etc.) and figure out the total price. We'll maintain this database using the get/put table.

For most things, there is a single price regardless of how many you buy. For example, if an apple costs 50 cents, we'll say

```
(put 'price 'apple 50)
```

But some things cost less in quantity. For example, suppose that cans of soda cost 65 cents each, but if you get six-packs, each can costs only 60 cents. To represent that, instead of putting a number in the table, we'll put in a function that takes the quantity purchased as its argument and returns the total price:

```
(put 'price 'soda (lambda (quantity)
                    (+ (* (quotient quantity 6) 360)    ; 6 cans * 60 cents
                       (* (remainder quantity 6) 65))))
```

We'll represent an *item* such as "5 apples" using tagged data: (attach-tag 'apple 5). **Respect the data abstraction.** A *shopping list* is a list of items.

Write the procedure price that takes a shopping list as its argument and returns the total price for all items:

```
> (price (list (attach-tag 'apple 5) (attach-tag 'soda 8)))
740        ; 5*50=250 for apples, (6*60)+(2*65)=360+130=490 for soda
```

Your name ________________________________ login cs61a-________

Question 4 (7 points):

Ben Bitdiddle wants to combine data-directed programming with message-passing. He likes the message-passing notation, in which a datum is represented using a procedure that accepts messages, but he wants to be able to use put to add new messages to a type. For example, he wants to say

```
(put 'square 'perimeter (lambda (s) (* 4 s)))
(put 'square 'area (lambda (s) (* s s)))

(define s3 (make-shape 'square 3))
```

and then have s3 behave just like the message-passing square as discussed in lecture:

```
> (s3 'area)
9
> (s3 'perimeter)
12
```

The central new procedure here is make-shape, a procedure that takes a type name and a linear size as its arguments and returns a datum of that shape and size. (In the example, make-shape was called to create a square of size 3.) The procedure representing a datum (such as s3 in the example) should look in the table to find the method stored using put for whatever message is given to it.

Write make-shape.

Question 5 (8 points):

This question concerns object-oriented programming.

You have been hired by the registrar to handle course enrollment.

(a) Create a `course` class with instantiation variables `department`, `number`, and `size` (how many students can enroll).

A `course` object should accept the message `enroll` with the name of a student as its argument. If there is room in the course, that student should be enrolled. The `enroll` method will return `#t` if the student was enrolled, or `#f` if there was no room. (To simplify the problem, we won't implement waiting lists.)

The message `students`, with no arguments, should return a list of the students enrolled in the course. The message `space-left`, with no arguments, should return the number of spaces remaining for more students.

```
> (define cs61a (instantiate course 'cs '61a 2))
> (ask cs61a 'enroll 'Ryan)
#t
> (ask cs61a 'space-left)
1
> (ask cs61a 'enroll 'Yaping)
#t
> (ask cs61a 'enroll 'Chris)
#f
> (ask cs61a 'students)
(Yaping Ryan)
```

(This question continues on the next page.)

(Question 5 continued.)

(b) Create a catalog class with one instantiation variable, courses, which will be a list of course objects. Give it a method available that takes a department as argument and returns a list of the numbers of courses in that department with room for more students:

```
> (define cs61b (instantiate course 'cs '61b 10))
> (define cs61c (instantiate course 'cs '61c 5))
> (define ps2 (instantiate course 'ps 2 100))
> (define ucb-catalog (instantiate catalog (list cs61a cs61b cs61c ps2)))
> (ask ucb-catalog 'available 'cs)
(61b 61c)
```

In this example, cs61a is not available because it's full, and ps2 is not listed because it's in the wrong department.