CS 61A   Spring 2008   Midterm 2 solutions

1. What will Scheme print?

```
> (map list '(1 2 3))

((1) (2) (3))

---->XX------>XX------>X/
     |        |        |
     |        |        |
     V        V        V
     X/       X/       X/
     |        |        |
     V        V        V
     1        2        3
```

MAP calls LIST three times, once for each number in the argument list: (LIST 1), (LIST 2), and (LIST 3). The values returned by these LIST calls make up the elements of the returned list.

```
> (let ((x '(1 2))
        (y '(8 9)))
    (cons x (append y x)))

((1 2) 8 9 1 2)

    +-------------+
    |             |
--->**            |    x
     |            |    |
     V            |    V
     XX------>XX--+--->XX------>X/
     |        |        |        |
     V        V        V        V
     8        9        1        2
     ^        ^
     |        |
 y-->XX------>X/
```

This picture shows why start arrows are important! The first pair of the result is the one shown as ** (instead of XX like the other pairs). The CAR of that pair is the value of X, which is the list (1 2) formed by the two rightmost pairs of the diagram. The ** pair is the one pair formed by the call to CONS.

The call to APPEND produces the four-element list whose spine is the row of four pairs below the ** pair. The first two pairs are made by APPEND; the last two pairs are the ones shared with the value of X.

(Note that the two pairs forming the value of Y are /not/ part of the returned value, so it's okay if you left them out of your diagram.)

```
> (cons (cons 1 2) (append '(18 3) '()))

((1 . 2) 18 3)

---->XX------>XX------>X/
     |        |        |
     |        V        V
     |        18       3
     V
     XX--->2
     |
     V
     1
```

The main point here was to see if you could draw a correct box and pointer diagram for a pair that isn't a list, the one that prints as (1 . 2). (By the way, is this result a list? Should LIST? of it return true? Yes, because the definition of a list only mentions the CDRs of the spine pairs: a list is either the empty list or /a pair whose CDR is a list/. That's true of this structure, even though the /CAR/ of one of the pairs isn't a list.

Some people thought that (1 . 2) was a 3-element list with 1, a period, and 2. The dot in "dotted pair" notation is part of the notation, just as the parentheses are part of the notation, not part of the actual data.

Scoring: One point per printed form, one point per diagram. At most one point lost for quotation marks anywhere in the answer.

2. OOP nomenclature

SICP	instance	book
novel	child class	book
title	instantiation variable	book
ASUC Bookstore	instance	bookstore
inventory	instance variable*	bookstore

SICP is a /particular/ book, so it's an instance of the BOOK class.

A novel is a /kind of/ book, so it's a child class. (It might, for example, have an instance variable GENRE, with possible values SF, MYSTERY, ROMANCE, etc., which other kinds of books might not have.)

A title is a property of a particular book, so it might be an instance variable or an instantiation variable. But we thought that it would almost certainly be an instantiation variable, because the title is specified when the book comes into existence, rather than being computed later as a result of messages sent to the book.

The ASUC Bookstore is a particular bookstore, hence an instance.

(*) The inventory of books is a property of a particular bookstore -- Cody's carries different books in stock from the ASUC Bookstore. We think it's more
likely to be an instance variable than an instantiation variable, on the theory that you don't create a bookstore by making a huge list of books; you
start with empty shelves and then you buy books from publishers to fill them.
But we accepted "instantiation variable" also for this one.

Scoring: -1 point per error, floor of 0.

3. Data abstraction.

There were ten /possible/ abstraction violations in the program:

```
(define (pairtree-map car-fn cdr-fn tree)

  (let ((this (car tree)))
               ---
    (make-tree (make-tree (car-fn (datum this)) (cdr-fn (cdr this)))
     ---------  ---------          -----                 ---
            (pair-forest-map car-fn cdr-fn (children tree)))))
                                            --------

(define (pair-forest-map car-fn cdr-fn forest)

  (if (null? forest)
       -----
      '()

      (make-tree (pairtree-map car-fn cdr-fn (datum forest))
       ---------                              -----
            (pair-forest-map car-fn cdr-fn (cdr forest)))))
                                            ---
```

Some of these were correct; some needed to be changed:

```
(define (pairtree-map car-fn cdr-fn tree)
  (let ((this (DATUM tree)))
    (MAKE-TREE (CONS (car-fn (CAR this)) (cdr-fn (CDR this)))
            (pair-forest-map car-fn cdr-fn (CHILDREN tree)))))
```

```
(define (pair-forest-map car-fn cdr-fn forest)
  (if (NULL? forest)
      '()
      (CONS (pairtree-map car-fn cdr-fn (CAR forest))
         (pair-forest-map car-fn cdr-fn (CDR forest)))))
```

There are three data types in this problem: Trees, Forests, and Pairs (one pair in each datum). The variable TREE is a Tree; FOREST is a Forest; and THIS, the datum of a tree node, is a Pair. So whenever you see something of the form (_______ TREE), what goes in the blank has to be a selector for
Trees -- either DATUM or CHILDREN. Similarly, (______ FOREST) or (_____ THIS)
require pair selectors (because a Forest is a list, and lists use pair selectors and constructors) -- either CAR or CDR.

Scoring: 1/2 point off per error (either failing to change something that should be changed, or changing something that shouldn't). This includes errors in places other than the ten we identified as possibilities, i.e., changing the name of CAR-FN to DATUM-FN counts as an error. The total is rounded down, but with a floor of zero.

4. Scheme-1.

(a) Substituting the actual arguments 5 and 7 for the parameters X and Y in the body of the outer LAMBDA expression gives

```
(lambda (z) (z 5 7))
```

Scheme-1 then evaluates /that/ expression, and its value is a procedure. But, unlike STk, Scheme-1 represents a procedure using the lambda expression
itself, so the value returned is

```
(lambda (z) (z 5 7))
```

If you said something like #[CLOSURE ...], then at least you understand that
the returned value is a procedure, but the point of this question is to know
how Scheme-1, specifically, works, so giving us the STk return value doesn't indicate that knowledge.

Scoring: 2 points for this one.

(b) How many calls to EVAL-1 with each kind of expression?

The first call to EVAL-1 is with the entire expression we typed in:

```
((lambda (x y) (lambda (z) (z x y))) 5 7)
```

This is an application of a non-primitive procedure, namely the procedure returned by the outer lambda expression. The first step in evaluating a procedure call is to evaluate all the subexpressions! That's three calls to EVAL-1, which we'll list from right to left because it's easier that way
in this case:

7 is a number.

5 is a number.

(lambda (x y) (lambda (z) (z x y))) is a special form.

Then we actually do the invocation, by substituting the actual argument values for the formal parameters in the body, and evaluating the result, which, as explained earlier, is the expression

(lambda (z) (z 5 7)) which is a special form.

That's all; we aren't /invoking/ the procedure created by the inner lambda expression.

So, adding these up, we get

2 numbers
2 special forms
0 applications of primitive procedures
1 application of a non-primitive procedure.

Some students misunderstood the problem as asking about the different expressions evaluated /by STk/ during the invocation of EVAL-1, i.e., the number of numbers, etc., within the body of EVAL-1 itself. But the problem statement was unambiguous; it doesn't allow for that interpretation.

Scoring: 1/2 point each, rounded down.

5. Mobiles.

If you understand the abstract data types in this problem, the actual code is easy; students who had trouble generally mixed up mobiles with branches.

The question reminded you what those types are:

* a Mobile has two Branches.

* a Branch consists of a length and a structure, which is either a number (the weight) or another Mobile.

Since you are asked to write a predicate, which shouldn't have to build any
new mobiles, you can concentrate on the /selectors/ for these types:

(left-branch mobile) returns a Branch.
(right-branch mobile) returns a Branch.

(branch-length branch) returns a number.
(branch-structure branch) returns either a number or a Mobile.

(The problem allows you to abbreviate these names, but in these solutions
I've used the full names because it's easier to read this way.)

The base cases of the recursion -- the leaf nodes -- are structures that
are just numbers.

Two structures are the same if:
  - they are equal numbers;
  - they are mobiles, and their left branches are the same, and their
    right branches are the same;
  - they are mobiles, and the left branch of one is the same as the right
    branch of the other, and vice versa.

It's the third possibility that makes "sameness" different from strict
equality, which we could have compared using EQUAL?.

Two branches are the same if their lengths are equal and their structures
are the same. (Not "their lengths are the same"; the lengths are just
numbers, and have to be plain old numerically equal.)

Watch how the Scheme code exactly follows these definitions:

```
(define (same-structure? struct1 struct2)
  (or (and (number? struct1) (number? struct2) (= struct1 struct2))
      (and (not (number? struct1))
	   (not (number? struct2))
	   (same-branch? (left-branch struct1) (left-branch struct2))
	   (same-branch? (right-branch struct1) (right-branch struct2)))
      (and (not (number? struct1))
	   (not (number? struct2))
	   (same-branch? (left-branch struct1) (right-branch struct2))
	   (same-branch? (right-branch struct1) (left-branch struct2)))))

(define (same-branch? branch1 branch2)
  (and (= (branch-length branch1) (branch-length branch2))
       (same-structure? (branch-structure branch1)
			(branch-structure branch2))))
```

The reason there's all that type-checking in SAME-STRUCTURE? but not in
SAME-BRANCH? is that a "structure" can really be either of two different
data types, whereas there's only one kind of branch. If we're comparing
a weight (a number) in one mobile against a sub-mobile of another mobile,

they clearly shouldn't be considered equal, but the program shouldn't
crash either, which is what would happen if we apply = to a mobile, or
LEFT-BRANCH to a number.

The style of ANDs inside ORs may seem strange to you, in which case it can
also be written with IF or COND:

```
(define (same-structure? struct1 struct2)
  (cond ((and (number? struct1) (number? struct2))
	 (= struct1 struct2))
	((and (not (number? struct1)) (not (number? struct2)))
	 (cond ((same-branch? (left-branch struct1) (left-branch struct2))
		(same-branch? (right-branch struct1) (right-branch struct2)))
	       ((same-branch? (left-branch struct1) (right-branch struct2))
		(same-branch? (right-branch struct1) (left-branch struct2)))
	       (else #f)))
	(else #f)))	; one number and one mobile
```

... but I find the first version easier to read.

The most common way to go wrong in problems about mobiles is to call the
same
procedure sometimes with a mobile as argument, other times with a branch
as
argument. This can happen because of a specific confusion about mobiles,
thinking "a mobile has two branches, each of which is a mobile" -- almost
true, but leaving out the fact that the /length/ of each branch is important
in determining whether or not the mobile balances. It can also happen
because
of a general failure to believe that a function's domain is important.

You may be thinking, "why is it wrong to call a procedure sometimes with
a
mobile as argument, and sometimes with a branch as argument; but it's okay
to
call a procedure sometimes with a mobile as argument, and sometimes with
a
/number/ as argument?" The reason is that both mobiles and numbers can
play
the same role in this system of types, namely, both can be the /structure/
that hangs off the end of a branch. Not any number will do; it has to be
a
/weight/, not a length -- to say something like

```
	(same-structure? (branch-structure branch1) (branch-length branch1))
```

would be really confused.

Many students tried to combine SAME-STRUCTURE? and SAME-BRANCH? into a
single procedure. This is theoretically possible (just substitute the
body of SAME-BRANCH? in place of each invocation of SAME-BRANCH? inside
SAME-STRUCURE?), but the result is long and ugly, and it's very easy to

get wrong. One common mistake was to check if the lengths match (maybe rotated), and check if the structures match (ditto), but allow the lengths to be rotated while the structures aren't or vice versa. For example, a (balanced) mobile whose left branch is a 3-gram weight at length 4 cm, and whose right branch is a 2-gram weight at length 6 cm was incorrectly said to match an unbalanced mobile whose left branch is a 2-gram weight at length 4 cm, and whose right branch is a 3-gram weight at length 6 cm.

Another mistake that some people made by tring to cram everything into one procedure was to check that the left branch of struct1 matches either branch of struct2, and to check that the right branch of struct1 matches either branch of struct2. This incorrectly returns #T if struct1 is symmetric (both branches the same), but struct2 isn't.

An even worse confusion was to treat the mobile as if it were a non-binary tree, with a forest full of children. Invocations of CAR and CDR were a likely indicator of this sort of problem; invocations of DATUM and CHILDREN definitely showed this confusion.

Scoring:

8: perfect.
6: Doesn't check for length and structure simultaneously, doesn't check if
   one is mobile and one is not.
5: Missing weight base case.
4: Mix up mobiles and weights.
2: Just total weight compare; uses list procedures.

6. List processing.

Each element of a three-branching list must be either a three-branching list
/or a non-list/. Therefore, it's easiest to write this procedure if we extend its domain to include non-lists:

```
(define (three-branching? lst)
  (if (not (list? lst))
      #t
      (and (= (length lst) 3)
         (three-branching? (car lst))
         (three-branching? (cadr lst))
         (three-branching? (caddr lst)))))
```

You may think it inelegant to use three separate expressions to test the three elements of LST individually, and, indeed, if the problem had been to write THREE-HUNDRED-BRANCHING? we wouldn't have wanted to do it that way.

Those three tests can be replaced by a use of higher order functions:

```
      (and (= (length lst) 3)
	   (= (length (filter three-branching? lst) 3))
```

(As always, when you are asking a yes-or-no question about a list it's much cleaner to use FILTER than to use MAP and then have a second pass going through the resulting list of #T and #F values.)

Alternatively, you can use a recursive helper:

```
      (and (= (length lst) 3)
	   (all-three-branching? lst))

(define (all-three-branching? lst)
  (cond ((null? lst) #t)
	((three-branching? (car lst))
	 (all-three-branching? (cdr lst)))
	(else #f)))
```

But you can't use THREE-BRANCHING? itself as the helper:

```
	... (three-branching? (cdr lst)) ...	;; wrong!
```

If LST is three-branching, then (CDR LST) has two elements, and therefore /isn't/ three-branching, even if its elements are!

If you didn't think to extend the domain as described above, then the testing of the three elements is a little more complicated:

```
      (and (= (length lst) 3)
	   (or (not (list? (car lst)))
	       (three-branching? (car lst)))
	   (or (not (list? (cadr lst)))
	       (three-branching? (cadr lst)))
	   (or (not (list? (caddr lst)))
	       (three-branching? (caddr lst))))
```

Of course the OR could be put in a helper procedure, to shorten this code.

Scoring:

8: Perfect.
6: Miss atomic case.
4: MAP solution, forgets to process #T and #Fs.
2: Assume 2-level list.

If you don't like your grade, first check these solutions. If we graded your paper according to the standards shown here, and you think the standards were wrong, too bad -- we're not going to grade

you differently from everyone else. If you think your paper was
not graded according to these standards, bring it to Brian or your TA.
We will regrade the entire exam carefully; we may find errors that
we missed the first time around. :-)

If you believe our solutions are incorrect, we'll be happy to discuss
it, but you're probably wrong!