61A Fall 2009 Midterm 2 solutions

1. What will Scheme print?

```
> (let ((the-professor bh))
    (ask the-professor 'greet))

(HI MY NAME IS BRIAN)
```

The main point of this problem was to make sure you understand that more than one variable name can refer to the same object, but in fact the most common error was to leave out the parentheses.

```
> (greet bh)

ERROR
```

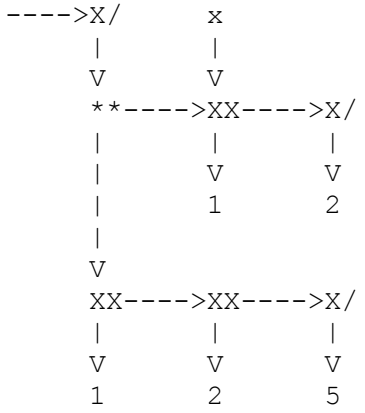
Most people got this: a method defined inside an object class isn't a global variable (or a variable of any kind, in fact), so there is no procedure named GREET.

Scoring: One point each.

2. Box and pointer diagram

```
> (define x '(1 2))
> (list [cons (append x '(5)) x])
(((1 2 5) 1 2))
```

It's important not to make assumptions about which parenthesis matches which in this problem! I've replaced one matching pair of parentheses with [square brackets] above to make it easier to see that the call to LIST has only one argument (the entire expression in square brackets), so it returns a list of one element. That element is itself a list, which is why there are two pairs of parentheses around the entire result: one for the inner list and one for the single-element list made by LIST. Here's the diagram:



The pair shown as ** above is the one made by the call to CONS. (CONS always makes exactly one pair.) Its CAR, the result of the APPEND call, is the bottom row of three pairs. Its CDR is the original list X. (The diagram above shows which pair is the value of the variable X, but you weren't required to do that.)

The most common error was to leave out the topmost pair, the one made by the LIST call.

Scoring: One point for the printed result, one for the diagram.

3. EVAL and APPLY in the calculator program

As I explained in lecture, the importance of the calculator example lies in the fact that interpreters for real programming languages, including Scheme, revolve around procedures EVAL and APPLY that work much like (but not exactly like, as we'll see in chapter 4) the ones here.

CALC-EVAL takes one argument, an /expression/, and returns the /value/ of that expression. So its domain is expressions. A typical call would be

```
(calc-eval '(+ (* 5 4) (- 9 3)))
```

which (to save you time in writing your answer) we abbreviated as

```
E (+ (* 5 4) (- 9 3))
```

CALC-APPLY takes two arguments: an arithmetic operation symbol (+, -, *, /) and a list of actual argument /values/. EVAL works in the world of expressions -- things typed in by users -- but APPLY works in the world of values, after all the notational issues have been handled by EVAL. Thus, the call to CALC-APPLY that finally determines the value of the expression above will be

```
(calc-apply '+ '(20 6))
```

which you would have abbreviated as

```
A + (20 6)
```

But that call to CALC-APPLY can't happen until we have argument /values/ to give it, which means that all of the evaluation of argument subexpressions happens before it.

This is another way of saying that the calculator, like Scheme, uses

applicative order evaluation: first evaluate the argument expressions, then
call the procedure with the argument /values/.

So, here's the solution:

```
E  (+ (* 5 4) (- 9 3))

E  (* 5 4)       Evaluate first argument.
E  5             Numbers are expressions too!
E  4
A  *  (5 4)      Now we can multiply the values.

E  (- 9 3)
E  9
E  3
A  -  (9 3)

A  +  (20 6)
```

It's okay if you evaluated the arguments right to left, but it's not okay if
you called CALC-APPLY before evaluating all the relevant argument
expressions.

Note that even trivial argument expressions (numbers) must be evaluated; the
program doesn't know they're trivial until CALC-EVAL checks for that case!
On the other hand, since procedures are not first class in the calculator
program, the operator symbols (+, etc.) are /not/ evaluated. This is the
most important way in which the calculator works differently from a Scheme
interpreter. (Of course there are many missing features, like variables
and lambda, but this is the only thing that's different rather than just
missing.)

Although the argument expressions are evaluated one at a time (unless maybe
you have a parallel-processing computer!), the overall expression is not
re-evaluated after each argument is evaluated:

```
     E  (* 20 (- 9 3))WRONG!

     E  (* 20 6)      WRONG!
```

The only expressions that are evaluated are ones actually typed by the user,
including subexpressions, but not including value-substituted expressions.
People who did that may have been misled by using the replacement modeler in
CS 3; the modified expressions produced by the modeler /give the same answer/

as the original expression you give it, but the modeler is not modeling the
actual sequence of events in a Scheme interpreter. It's just suggesting how
you, a human being, might understand the value of a complicated
expression.

Another common mistake was to show something like argument expressions rather
than argument values in the calls to CALC-APPLY:

```
A  +  (map calc-eval '((* 5 4) (* 2 3)))
```

or

```
A  +  ((E (* 5 4)) (E (* 2 3)))
```

We considered this to be understanding that the EVAL happens before the
APPLY, but not entirely getting the fact that the arguments to APPLY are
values, not expressions, or that the APPLY really happens after the EVAL.

Scoring: One point for each of the following:

- Call CALC-APPLY with list of values, not list of expressions.
- Call to CALC-APPLY comes /after/ all arguments are evaluated.
- CALC-EVAL is called for numbers.
- CALC-APPLY takes two arguments, the second a list of values.
- CALC-EVAL is /not/ called for operator symbols.

4. Data directed programming

People who had trouble with this problem mostly didn't recognize it as an
example of data-directed programming, just like the homework question about
the different divisions of Insatiable Enterprises.

(a) Generic GET-CAPTAIN.

```
(define (get-captain team)
  (operate 'get-captain team))
```

or

```
(define (get-captain team)
  (apply-generic 'get-captain team))
```

The first of these uses the procedure name OPERATE that I used in
lecture;
the second uses the name APPLY-GENERIC from the textbook. These are the
best solutions because they reuse a general mechanism already provided.
But
if you didn't remember or trust those, we accepted rewriting it:

```
(define (get-captain team)
  (let ((method (get 'get-captain (type-tag team))))
    (if method
      (method (contents team))
      (error "No method for team" (type-tag team)))))
```

Or, since we didn't require error checking,

```
(define (get-captain team)
  ((get 'get-captain (type-tag team)) (contents team)))
```

One fairly common error was to use APPLY, e.g.:

```
      (apply method (contents team))
```

This is wrong because APPLY requires a /list of/ argument values, not just
an argument value. (It can be a list of length 1, of course.)

Scoring: 3 points for this part. All three points for using OPERATE or APPLY-GENERIC. Otherwise, one point for each of the following:

* Correct call to GET (ignoring order of arguments).
* Taking CONTENTS of the team.
* Calling the method correctly.

We only took off one point for interpreting the question as calling for a /list of type-tagged players/ rather than what we wanted, which was a /type-tagged list of players/. That is, there should only be one type tag,
for the entire team, not one for each player.

(b) Methods for each team.

```
(define (a-captain players)
  (cdar (filter (lambda (player) (equal? (car player) 'captain))
          players)))

(define (s-captain players)
  (caar (filter (lambda (player) (list? (car player)))
          players)))

(define (j-captain player) player)
```

Here's a case in which combined names like CDAR and CAAR are helpful.
In the case of A-CAPTAIN, the call to FILTER will return something like

```
      ((captain ahmed owainati))
```

because it returns a list containing all of the elements that satisfy the predicate, even if there is only one such element. So CAR of that result

is the list (captain ahmed owainati) and CDR of /that/ is (ahmed owainati)
which is what we want.

(By the way, even though the team is called the Ahminators, you're not supposed to assume that Ahmed is its captain! You're supposed to use the player list you're given and find the captain according to the rules given
in the problem.)

We accepted (filter (lambda (player) (= (length player) 3)) players) also.

In the case of the Stewpifiers, FILTER returns a list whose only element is
the list-in-a-list with the captain's name:

```
      (((stewart liu)))
```

CAR of that is ((...)) and CAR again gives us the result we need.

For the Jerricalities, the "player list" is already exactly what we need to return, so the function is trivial. We also accepted

```
(define j-captain identity)
```

for this case.

Scoring: This part had a total of 6 points, assigned as follows:

A-captain: 2 points if correct, 1 for "has the idea" (e.g., MAP instead of FILTER).

S-captain: 2 points, as above.

J-captain: 1 point.

"General understanding": 1 point. Mostly this meant having the domains and ranges right, e.g., returning a list rather than a procedure that returns
a list.

(c) Install methods in table.

```
(put 'get-captain 'ahminators a-captain)
(put 'get-captain 'stewpifiers s-captain)
(put 'get-captain 'jerricalities j-captain)
```

In some cases, the solution given to part (b) used GET-CAPTAIN from part (a),
and the procedures we wanted in part (b) were given as lambda expressions in

the calls to PUT here. We accepted this, awarding both (b) points and (c)
points based on the solutions to (c).

Scoring: This part had a total of 2 points if correct, 1 point if PUT is used with some approximation to these arguments but inconsistent with earlier
answers, e.g.,

* Argument order for PUT inconsistent with GET in part (a).
* Type signature (list of types) here but just type in (a).
* Function name (e.g., A-CAPTAIN) used instead of type tag.

But we accepted other variants on the type tags, e.g., A-TEAM or AHMED for AHMINATORS.

5. OOP

(a) In this part we wanted only a trivial BOOK class:

```
(define-class (book title author page-count))
```

Scoring: 2 if correct. 1 if undesired extras added, such as a BOOKSHELF instance variable. (But redundant methods to return the instantiation variables were allowed.) 0 if the solution isn't a proper class definition
at all, such as

```
(define-class (book title author page-count)       ; WRONG!
  (list title author page-count))
```

(b) Fill in the blanks.

SICP is an /instance/ of the BOOK class, because it's a particular book, not, e.g., a kind of book.

"Bookshelf" is /X (nothing)/ of the BOOK class. Books and bookshelves are
two different kinds of thing.

"Paperback" is a /child/ of the BOOK class, because it's a /kind/ of book.

People made ingenious arguments for alternative answers, such as taking SICP as meaning the /title/ of the book (thus a variable) rather than the book itself. But we didn't find any such arguments really convincing; our
answers are clearly the best answers, showing the clearest understanding of how objects and classes would be used to model this book simulation.

So instead of arguing with us, try to learn to understand why we chose the
answers we did!

Scoring: 1 point each.

(c) Bookmark class and improved book class.

```
(define-class (bookmark page-number color))

(define-class (book title author page-count)
  (instance-vars (bookmark-list '()))
  (method (add-bookmark pagenum color)
    (set! bookmark-list (cons (instantiate bookmark pagenum color)
                        bookmark-list)))
  (method (find-pages color)
    (map (lambda (bookmark) (ask bookmark 'page-number))
       (filter (lambda (bookmark) (equal? color (ask bookmark 'color)))
            bookmark-list))))
```

Scoring:

One point for bookmark class.

In the book class:
      1 point for correct INSTANCE-VARS.
      2 points for ADD-BOOKMARK:
          2 if correct.
          1 for "has the idea": error in INSTANTIATE, no set!, or
            error in CONS.
          0 if no instantiate at all.
      2 points for FIND-PAGES:
          1 point for filtering objects by color, plus
          1 point for extracting page numbers from them.

One common error was leaving empty lists in the list of objects (usually by calling MAP instead of FILTER, but sometimes from incorrect helper procedures).

No points for not using objects at all!

6. Tree recursion

```
(define (tree-filter tree pred)
  (if (pred (datum tree))
      (make-tree (datum tree)
            (filter (lambda (x) x)
                  (map (lambda (child) (tree-filter child pred))
                       (children tree))))
      #f))
```

The first point to note here is that PRED is called with (DATUM TREE) as its
argument, not TREE. That should have been clear from the example, in which
EVEN? is used as the predicate; the domain of EVEN? is numbers, not Trees.

If PRED returns false, then TREE-FILTER should return false (that is, #F).
Many Tree programs don't need an explicit base case, because the MAP call
just does the right thing for leaf nodes, /and there is always a root node
in the result/. But that's not true in this problem; sometimes we don't
return a Tree at all. So we have to check for that situation.

If PRED returns true (anything but #F), then we /do/ want to return a Tree,
which will have at least the root node intact. So we call MAKE-TREE, and
its first argument is the original datum -- in this problem, the data in the
nodes are never altered, only removed altogether.

The second argument to MAKE-TREE is more complicated. Of course we want to
call TREE-FILTER for all the children of this node, and that's what the MAP
call above does. But the value returned by MAP may have some #F elements.
For example, the tree in the example has three children, with root data
2, 6, and 7. The third of those fails the EVEN? test, so MAP returns a list
that looks like this:

      (<tree-with-2> <tree-with-6> #F)

We have to eliminate those #F values, thus the call to FILTER. The first
argument to FILTER is the rather unusual "predicate" (LAMBDA (X) X) because
this identity function will return a true value for every argument except #F.
This is really convenient, and it's the reason we specified a return value of
#F if the root datum fails the predicate test.

Of course the calls to MAP and FILTER could be replaced with helper functions
that take a forest as argument.

Several people gave us the solution to a sample problem from an earlier
midterm, called DATUM-FILTER, that returns not a Tree but a flat list of
data -- for our example, something like (0 2 4 6). It's really not a good
idea to copy mindlessly from a different problem! Always think about the
domain and range of the procedure you are being asked to write.

Some people tried to solve the filtering problem by returning an empty list,
instead of #F, for an omitted child, and then applying APPEND to the returned
values to remove the empty lists. Although that does indeed remove the empty
lists, it also flattens out the non-empty lists (Trees). For example:

```
STk> (apply append '((a b) () (c d) () (e f)))
(a b c d e f)
```

(This is a simplified example; (A B) isn't a Tree, but you can still see the
flattening effect of APPEND here.)

Some people, instead of trusting MAP to handle leaf nodes correctly, made a
special test for (NULL? (CHILDREN TREE)) and then returned (DATUM TREE)
instead of the correct TREE. The range of TREE-FILTER is Trees!

Some people solved the wrong problem, thinking that the argument should be
a binary tree, or a deep list.

Scoring: We started with this scale:

8 Correct.

7 Returns () when root datum fails the PRED test.
7 APPENDs the children.

6 (PRED TREE) instead of (PRED (DATUM TREE)).
6 Return datum instead of Tree for leaf nodes.

5 No filtering of failed children.

3 Flat list of data (DATUM-FILTER).

2 Correct solution for binary tree or deep list.

(Other scores possible for uncommon errors.)

Then we subtracted one point for data abstraction violations (e.g., CAR of
a Tree or DATUM of a forest).