CS61A   Solutions to mt2, Spring 2010

1. What will Scheme print?

```
> (car tree)
DAV
```

While yes, the datum of a tree is really its "car", it violates the Tree data abstraction, and as such is a DAV.

```
> (datum tree)
A
```

The variable tree is bound to a Tree that starts with the node whose datum is A. Most people got this one.

```
> (children (datum tree))
ERROR
```

As mentioned before, the datum of tree is A. Note that this is A the symbol, NOT another Tree node, and as such you cannot take the children of it! This would produce an error because it tries to call "children" on a letter. (It's also a DAV, but we mentioned that if there's an error you should say it returns an error.)

```
> (eval-1 '(sentence a b))
ERROR
```

The first thing to note is that this was typed into STk, not Scheme-1. As a result, the procedure and its argument are evaluated by STk before being applied. What does EVAL-1 evaluate to? The proc edure EVAL-1. What does '(sentence a b) evaluate to? The list of the words "sentence", "a", and "b".

Next, EVAL-1 will treat this list as a pair, and will attempt to apply (eval-1 sentence) on (map eval-1 (a b)). EVAL-1 of sentence returns the STk primitive procedure bound to sentence, but what does mapping EVAL-1 onto "a" and "b" do? Well, they are symbols, so EVAL-1 will treat them as such and look for an _STk_ binding for both. Of course, a and b are unbound, and so the expression will return unbound variable error.

```
> (cadr (children tree))
E   , drawn in a box (because it's a tree)
 \
  F
```

Recall that the children of a Tree is a list of _Trees_, not their datums! That is, the children of tree is this list, whose first element is the Tree starting at the node whose datum is B, and whose last element is the Tree starting at the node whose datum is E:

```
        B   E
    (  / \   \  )
      C   D   F
```

```
         E
    The cadr of this list returns the second tree,
the tree  \   .

           F
```

```
> (make-tree 'a '())
```

a, drawn in a box (because it's a tree)

It should have been immediately obvious that the return value has to be a Tree; after all, the range of the make-tree function is a Tree! MAKE-TREE is a constructor that takes two arguments, the datum and the children. So all this says is return a new Tree whose datum is the letter a and whose children is an empty list (i.e. no children).

Furthermore, a Tree drawn with the empty list as a child is not the same as a Tree with no children. We only gave credit if you demonstrated that you knew the difference.

```
> (make-tree (datum tree) (children tree))
    A     , drawn in a box (becuase it's a tree)
   / \
  B   E
 / \   \
C   D   F
```

Again, recall that MAKE-TREE is a constructor that takes two arguments, the datum and the children of the new Tree. The children of a Tree is always a list of Trees. So what this is saying is "create a new Tree whose datum is tree's datum, and whose children is tree's children". As we saw in the (cadr (children tree)), each element of the children of tree is a whole Tree. So really what we're creating here has the _exact same structure_ as the original tree!

```
> (eval-1 '(sentence 'a 'b))
```

```
(a b)
```

Finally, Scheme-1 doesn't error! As before, STk will first evaluate both: EVAL-1 is a variable bound to the procedure EVAL-1, but here '(sentence 'a 'b) becomes (sentence 'a 'b). Then, when it's time for EVAL-1 to recursively evaluate the arguments to the sentence procedure, EVAL-1 will note them as quoted expressions and return just the a and b by themselves (instead of trying to use Scheme's eval on them, as in the previous EVAL-1 question). This will allow SENTENCE to work properly on the words "a" and "b".

Scoring: One point each, all or nothing.

2. Tree recursion

Write a function DEEP-TREE-REVERSE which deep reverses a Tree, i.e. it reverses the order of the children of each node. Use MAKE-TREE, DATUM, and CHILDREN for the Tree data structure. You can use the REVERSE procedure which reverses a list.

Usually, if we give you some helper, the easiest solution includes the use of that helper. Here, the insight is supposed to be that since the children of a tree is a list, and we ask you to "reverse the order of the children of each node", you should reverse the children of the tree before you MAP DEEP-TREE-REVERSE over them:

```
    (define (deep-tree-reverse tree)
      (make-tree (datum tree)
                 (map deep-tree-reverse (reverse (children tree)))))
```

The lack of base case here is on purpose; there's really no such thing as the "null Tree", since if the tree didn't exist, map would not call DEEP-TREE-REVERSE on it.

By far the most common mistake was to check if the (children tree) was NULL?, but then return (datum tree) instead of the tree itself. (Remember that the children of a Tree is always a list of /Trees/!)

Scoring:
5 Perfect
4 Trivial mistakes
3 Returning (datum tree) instead of tree for the base case
  "Re-reversing" at each step; that is, reversing within some DEEP-FOREST-REVERSE procedure, so that every recursive call causes another reverse
  Incorrectly constructing the reversed list of children (usually as a result of using LIST or APPEND incorrectly, instead of CONS)
1-2 "An idea"

3. Deep lists

Let L be a deep proper list *built only of pairs and empty lists*, e.g. (), (()), (()()), ((())). Write a fucntion DEEP-NULL-COUNT which takes a list and counts the number of empty lists in the list.

Many people forgot that the CDR of the last pair of each list contains another empty list! There are two ideas for solutions, the first involving CAR/CDR recursion:

```
    (define (deep-null-count L)
      (if (null? L)
          1
          (+ (deep-null-count (car L)) (deep-null-count (cdr L)))))
```

This works because if L isn't NULL?, we know it's a pair (since we said that so in the question), and can ask both the CAR and the CDR its DEEP-NULL-COUNT, and then add those values together. The second solution looked like:

```
    (define (deep-null-count L)
      (accumulate + 1 (map deep-null-count L)))
```

This works because if L is null, the MAP call does nothing, and so ACCUMULATE simply returns 1. Then all the previous ACCUMULATE calls will add those 1's together.

Scoring:
5 Perfect
4 Trivial mistakes
3 Number in the base case was off, or if the procedure worked for every list except
  when the input was '()
2 "An idea"; either a forgotten base case, or forgetting to accumulate the result
  of mapping
0 Not traversing the entire list, or didn't work for deep lists

4. Tree recursion

What will MYSTERY applied to the following Tree (which uses the usual Tree data
abstraction using the DATUM and CHILDREN selectors) give? If the result is a Tree, you
can just draw the tree.

```
          Jerry
         /  |  \
        /   |   \
   Alice   Bob  Charles
   /   \
Ryan  Rudolf
```

```
(define (mystery tree)
  (make-tree (foo tree)
             (map mystery (children tree))))

(define (foo tree)
  (+ 1 (bar (children tree))))

(define (bar forest)
  (accumulate + 0 (map foo forest)))
```

First, the solution, a tree:

```
        6
       /|\
      3 1 1
     / \
    1   1
```

The first insight to make is that MYSTERY will be called on /every node/ in the tree. This is because a call to MYSTERY also calls MAP with MYSTERY on every child of the tree. The MYSTERY call returns the result of MAKE-TREE (the range of which is a Tree), so the resulting output is a Tree, and furthermore, because we know there is a call to MAKE-TREE for each node in the Tree, the resulting output is a Tree of the same size as the input Tree!

Next, we should take a look at FOO, because the result of that is the new datum of MYSTERY's return value. What does FOO appear to do? It should add 1 to the result of calling BAR on the children of FOO's argument, a Tree. What does BAR do then? It accumulates over the result of calling MAP with FOO as it's function on all the Trees in the forest.

This mutual recursion only ends once there are no children in the argument to FOO, tree. When there are no children, FOO returns 1. This means that the result of calling FOO on a leaf node is 1. Taking one step back, calling BAR on a list of leaves simply returns the total number of nodes in the forest. We can then surmise that FOO simply returns the number of Trees within its argument tree. (Starting from the base case, if FOO is given a leaf, FOO returns 1. If FOO is given a Tree that has only a single child, FOO returns 2, and so on...)

So recall that MYSTERY creates exactly one call to FOO, and then calls MYSTERY on the children of the tree. So there will be one FOO call for each node in the original tree, so each node in the original tree maps to a number that expresses the number of nodes in that tree.

Scoring:
4 Perfect
3 If tree is represented as list
  Wrong root, or if leaves have extra 0 as children

2 Only two levels in output tree
  Correct structure, but fundamental misunderstanding in output (i.e. all leaves 0, numbers larger on bottom, etc.)
1 Returning 6
  Right tree structure
  Two levels deep with errors in numbers
0 Error or DAV
  Unrelated lists

Wrong structure in tree

5. Recursion

Let a set of numbers be represented as an ordered list, e.g. (4 5 10), where each element appears only once in the list. Write a function (SET-DIFFERENCE LEFT RIGHT) which returns the set of all elements in LEFT which are not in RIGHT. Here is an example:

```
> (set-difference '(4 5 10 16) '(1 2 5 10))
(4 16)
```

Write SET-DIFFERENCE as efficiently as possible using hte orderedness of the list, i.e. do *not* use MEMBER or any helper functions.

As noted, we wanted you to use the fact that both lists are ordered to make intelligent decisions about when to include a number in the final set, and when to exclude a number. Let's start with the easiest case; if the CAR of LEFT is equal to CAR of RIGHT, then by definition of SET-DIFFERENCE, we do not want to include the number. This implies that, for this case, return (set-difference (cdr left) (cdr right)).

The next case to think about is if the CAR of LEFT is greater than the CAR of RIGHT. Well, although these two numbers are not equal to each other, there's nothing to say that some of the next numbers in RIGHT are not equal to the CAR of LEFT, since they are sorted. For example:

```
    LEFT: (4 6 7)   RIGHT: (2 3 4 9)
```

Although CAR of LEFT is not equal to CAR of RIGHT, we still need to check down the CDR of

RIGHT before we can make any decisions. This implies that, for this case, return
(set-difference left (cdr right)).

Finally, the last case is if the CAR of LEFT is less than the CAR of RIGHT. Now we know for
sure that the RIGHT cannot contain the CAR of LEFT, because all the numbers in RIGHT are
larger than the CAR of LEFT:

```
    LEFT: (4 6 7)   RIGHT: (6 8 9)
```

So we include the CAR of LEFT in our final answer,
(cons (car left) (set-difference (cdr left) right))

Combining it with the null base cases, putting it all together gives us:

```
    (define (set-difference left right)
      (cond ((null? left) '())
            ((null? right) left)
            ((= (car left) (car right))
             (set-difference (cdr left) (cdr right)))
            ((> (car left) (car right))
             (set-difference left (cdr right)))
            ((< (car left) (car right))
             (cons (car left) (set-difference (cdr left) right)))))
```

Scoring:
7 Perfect
6 Missed either base case (or both)
4 Did not cons the (car left) when (car left) less than (car right)
3 A correct set-union solution
2 "An idea"
0 Nothing, used helpers

6. OOP

We would like to build a system for registering participants in a course, much like you did at the beginning of the semester.

We will use our OO syntax. First we define a class PARTICIPANT. A login is of the form "cs61a-xx", so a PARTICIPANT object's COURSE can be figured out from the LOGIN.

```
(define-class (participant name login)
  (method (course) (bl (bl (bl login)))))
```

(a) Not all partcipants are the same! Define a class STUDENT and a class TA, both of which are PARTICIPANTS. Each TA has a list of STUDENTs, and we will call this list SECTION (which is initially empty). You should also provide an ADD-STUDENT method which takes one argument STUD of type STUDENT and adds the student to a TA's SECTION.

```
    (define-class (student name login)
      (parent (participant name login)))

    (define-class (ta name login)
      (parent (participant name login))
      (instance-vars (section '()))
      (method (add-student stud)
        (set! section (cons stud section))))
```

Most people got this question right. One common mistake was forgetting that we wanted the STUD itself, not its NAME, added to SECTION. Another was forgetting to pass the PARENT the instantiation variables given to the STUDENT/TA.

Scoring: 4pts total, 1pt each:
Instantiation vars
Correct PARENT

Instance var for TA
Correct ADD-STUDENT method

(b) Write a sequence of Scheme expressions: 1. to create a STUDENT object for yourself;
2. to create a TA object for your TA; 3. to add your STUDENT object to your TA's SECTION

```
    > (define you (instantiate student 'yourname 'cs61a-aa))
    > (define us (instantiate ta 'ourname 'cs61a-ta))
    > (ask us 'add-student you)
```

Remember again that the ADD-STUDENT method takes in a STUDENT /object/, not a STUDENT's name! Otherwise this part should have been fairly straightforward.

Scoring: 3pts total, 1pt each line

(c) Write a procedure MAKE-ROSTER that, given a TA, will return a list of the NAMEs of the students in his or her SECTION.

```
    (define (make-roster ta-obj)
      (map (lambda (stud) (ask stud 'name)) (ask ta-obj 'section)))
```

In this question we did want the NAMEs of the STUDENTs, not the objects themselves, so for each STUDENT in a TA's SECTION, you simply ASK the STUDENT for its NAME.

Scoring: 3pts total, 1pt each:
(ask stud 'name)
(ask ta-obj 'section)
Getting the MAP correct