CS61A – Homework 3.1 | **Kurt Meinz**
University of California, Berkeley | Summer 2005

Topic: Hierarchical data

Lectures: Monday July 7, Tuesday July 8

Reading: Abelson & Sussman, Section 2.2.2–2.2.3, 2.3.1, 2.3.3

In this assignment you'll gain experience working with structures that have variable depth such as lists of lists and the Tree and Mobile abstract data types. However, the book does not have a tree ADT. In fact, when the book refers to "trees"—as in `scale-tree` on Page 112—it's really talking about deep lists.

Our tree ADT—which consists of the functions `make-tree`, `children` and `datum`—is itself usually implemented using lists:

```
(define make-tree cons)
(define datum car)
(define children cdr)
```

But remember, the underlying representation of any ADT is irrelevant! We can define `make-tree` and friends in a thousand different ways. As you do this homework, fight the desire to think of a Mobile or a Tree in terms of their underlying representations as lists.

This assignment is due at **8 PM on Sunday, July 10**. Put your answers into a file called `hw3-1.scm` and turn it in online with `submit hw3-1` as usual.

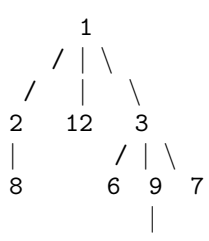
Question 1. Write a function `deep-map` that takes a unary function and a (possibly) nested list. It should apply the function to each atomic element of the list and return a new list with the same nested structure:

```
STk> (deep-map not '(#f ((#f) (#t))))
(#t ((#t) (#f)))
STk> (deep-map (lambda (a) 'foo) '())
()
STk> (deep-map (lambda (x) 'foo) '(((((3) 4) (5)) 6)))
(((((foo) foo) (foo)) foo))
STk> (deep-map square '(1 2 (3) 4))
(1 4 (9) 16)
STk> (deep-map list '(1 2 (3) 4))
((1) (2) ((3)) (4))
```

The hard work continues on the next page.

Question 2. In this question we'll make use of the Tree ADT presented in lecture. A Tree can have any number of children. The constructor is `make-tree` and takes two arguments, the second of which is a *list of Trees* which are the children. The selectors are `datum` and `children`. The following code builds up the tree at right (from the bottom up):

```
(define eight (make-tree 8 '()))
(define twelve (make-tree 12 '()))
(define ten (make-tree 10 '()))
(define six (make-tree 6 '()))
(define seven (make-tree 7 '()))
(define two (make-tree 2 (list eight)))
(define nine (make-tree 9 (list ten)))
(define three (make-tree 3 (list six nine seven)))
(define one (make-tree 1 (list two twelve three)))
(define thirteen (make-tree 13 '()))
(define fourteen (make-tree 14 '()))
(define fifteen (make-tree 15 (list thirteen fourteen)))
(define zero (make-tree 0 (list one fifteen)))
```



This is a large Tree specifically so that you can play with it. In testing your code, you may want to work with one of the subtrees, such as `three` or `nine`.

Write a function `fringe` that takes a Tree and returns a list of the datums of the leaf nodes, in any order:

```
STk> (fringe zero)
(8 12 6 10 7 13 14)
STk> (fringe three)
(6 10 7)
STk> (fringe six)
(6)
```

Question 3. This question explores a Mobile ADT. A Mobile is like a tree with only two branches at every node: a right branch and a left branch. From each branch hangs either a weight, which is just a number, or another Mobile. Here is a constructor:

```
(define (make-mobile left-branch right-branch)
    (list 'mobile left-branch right-branch))
```

A branch also consists of two parts: a length and a structure. The length of a branch is numeric; the structure at the end, however, can be *either* another Mobile or a weight (a number).

```
(define (make-branch branch-length branch-structure)
    (list 'branch branch-length branch-structure))
```

The following code builds up the Mobile at right (from the bottom up):

```
STk> (define mobile-1 (make-mobile (make-branch 4 5)
                                   (make-branch 2 10)))
STk> (define mobile-2 (make-mobile (make-branch 3 10)
                                   (make-branch 2 mobile-1)))
STk> (define mobile-3 (make-mobile (make-branch 8 mobile-2)
                                   (make-branch 5 10)))
```

The learning continues on the next page.

A. There are four selectors that need to be written. Two are for Mobiles: `right-branch` and `left-branch`, and two are for branches: `branch-structure` and `branch-length`. We'll build a simple error check into the selectors to ensure they're applied to the right type:

```
(define (left-branch mobile)
   (if (and (list? mobile) (equal? (car mobile) 'mobile))
       (cadr mobile)
       (error "Not a mobile -- LEFT-BRANCH: " mobile)))
```

Write the three remaining selectors analogously. Try them out on `mobile-3` above:

```
STk> (branch-structure (right-branch mobile-3))
10
STk> (branch-length (left-branch (branch-structure (left-branch mobile-3))))
3
STk> (branch-structure
        (left-branch
           (branch-structure
              (right-branch (branch-structure (left-branch mobile-3))))))
5
```

B. Write a function `total-weight` which returns the weight of a Mobile. Assume branches are weightless; hence, only the weights increase the total weight of a Mobile:

```
STk> (total-weight mobile-1)
15
STk> (total-weight mobile-2)
25
STk> (total-weight mobile-3)
35
```

Students tend to solve this problem by performing an unnecessarily exhaustive case analysis: is the left branch a weight? is the right branch a weight? are both of them weights? This approach indicates that you don't trust the recursion. You only need *one* base case and one recursive case! Ask yourself, what are the "leaves" of a Mobile?

C. A mobile is said to be *balanced* if the torque applied by its top-left branch is equal to that applied by its top-right branch, and if all the other mobiles hanging beneath it are themselves balanced. The "torque applied by a branch" means the product of the `branch-length` and the `total-weight` of the `branch-structure`. For example, the torque applied by the top-right branch of the mobile (whose length is 5 and whose structure is the weight 10) is 50. Write a `balanced?` predicate that takes a Mobile and returns a true value if it is balanced, #f otherwise:

```
STk> (balanced? mobile-1)
#t
STk> (balanced? mobile-2)
#t
STk> (balanced? mobile-3)
#f
```

Aim for the simplest possible base case. You may assume that a weight by itself is *always* balanced.

Question 4. We can represent a set as a list of distinct, unordered elements. We'd like to find the subsets of such a set. The subsets of a set S are all the sets that can be formed by selecting any number of the elements of S. For example:

```
STk> (subsets '(1 2 3))
(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

Notice that the empty set is a subset of *every* set, and every set is a subset of itself. Complete the following definition of `subsets`.

```
(define (subsets s)
   (if (null? s)
       (list '())
       (let ((rest (subsets (cdr s))))
         (append rest (map ?? rest)))))
```

Trust the recursion!