CS61A Week 6 | More Lists and Trees (v1.0)

More Lists

QUESTIONS:

1. Define a procedure `(depth ls)` that calculates how maximum levels of sublists there are in ls. For example,

```
(depth '(1 2 3 4)) ==> 1

(depth '(1 2 (3 4) 5)) ==> 2

(depth '(1 2 (3 4 5 (6 7) 8) 9 (10 11) 12)) ==> 3
```

   Remember that there's a procedure called `max` that takes in two numbers and returns the greater of the two.

2. Define a procedure `(count-of item ls)` that returns how many times a given item occurs in a given list. The given item could also be in a sublist. So,
   `(count-of 'a '(a b c a a (b d a c (a e) a) b (a))) ==> 7`

3. Write a procedure `(apply-procs procs args)` that takes in a list of single-argument procedures and a list of arguments. It then applies each procedure in procs to each element in args in order. It returns a list of results. For example,
   `(apply-procs (list square double +1) '(1 2 3 4)) ==> (3 9 19 33)`

4. Recall that there are two ways of defining procedures: the "real" way, and the sugar-coated way. Write a procedure `(unsugar def)` that takes in a procedure definition in sugar-coated syntax, and returns the same definition without using the syntactic sugar. For example,
   `(unsugar '(define (square x) (* x x))) ==> (define square (lambda (x) (* x x)))`

5. Recall that a `let` expression is actually just a `lambda` expression. Write a procedure `(let->lambda exp)` that takes in a `let` expression and returns the corresponding `lambda` expression. For example,

```
(let->lambda '(let ((x 3) (y 10)) (+ x y)))
    ==> ((lambda (x y) (+ x y)) 3 10)
```

6. *(Extra for Experts)* Implement `(median ls)` that returns the median of a list of numbers. You *cannot* use recursion, and you *cannot* define additional procedures. The only higher-order procedure you can use is `filter`. Assume nonempty lists.

Of Trees And Forests

A tree is, abstractly, an acyclic, connected set of nodes (of course, that's not a very friendly definition). Usually, we talk about a tree node as something that contains two kinds of things - data and children. **Data** is whatever information may be associated with a tree node, and **children** is a set of subtrees with this node as the parent. Concretely, it's often just a list of lists of lists of lists in Scheme, but *you should NOT to think of trees as lists at all.* Trees are trees, lists are lists. They are completely different things, and if you, say, do `(car tree)` or something like that, we'll yell at you for violating data abstraction. `car`, `cdr`, `list` and `append` are for lists, not trees! And don't bother with box-and-pointer diagrams - they get way too complicated for trees. Just let the data abstraction hide the details from you, and trust that the procedures like `make-tree` work as intended.

Of course, that means we need our own procedures for working with trees analogous to `car`, `cdr`, etc. Different representations of trees use different procedures. You're already seen the ones for a general tree, which is one that can have any number of children (not just two) in any order (not ordered by smaller-than and larger-than). Its operators are things like:

```
;; takes in a datum and a LIST of trees that will be the children
of this ;; tree, and returns a tree (define (make-tree datum
children) ...)

;; returns the datum at this node (define (datum tree) ...)

;; returns a LIST of trees that are the children of this tree. ;;
NOTE: we call a list of trees a FOREST (define (children tree)
...)
```

With general trees, you'll often be working with mutual recursion. This is a common structure:

```
(define (foo-tree tree)
     ...
     (foo-forest (children tree)))

(define (foo-forest forest)
     ...
     (foo-tree (car forest))
     ...
     (foo-forest (cdr forest)))
```

Note that `foo-tree` calls `foo-forest`, and `foo-forest` calls `foo-tree`! Mutual recursion is absolutely mind-boggling if you think about it too hard. The key thing to do here is - of course - **TRUST THE RECURSION!** If when you're writing `foo-tree`, you *belive* that `foo-forest` is already written, then `foo-tree` should be easy to write. Same thing applies the other way around.

QUESTIONS:

1. Write `(square-tree tree)`, which returns the same tree structure, but with every element squared. Don't use `map`!

2. Write `(max-of-tree tree)` that does the obvious thing. The tree has at least one element.

3. A maximum heap is a tree whose children's data are all less-than-or-equal-to the root's datum. And its children are all maximum heaps as well. Write `(valid-max-heap? tree)` that checks if this is true for a given tree.