CS61A Notes 02b – Fake Plastic Trees

Box and Pointer Diagrams

QUESTIONS: Evaluate the following, and draw a box-and-pointer diagram for each. (Hint: It may be easier to draw the box-and-pointer diagram first.)

1. `(cons (cons 1 2) (cons 3 4))`

2. `(cons '((1 a) (2 o)) '(3 g))`

3. `(list '((1 a) (2 o)) '(3 g))`

4. `(append '((1 a) (2 o)) '(3 g))`

5. `(cdr (car (cdr '(((1) 3) (4 (5 6))) )))`

6. `(map (lambda (fn) (cons fn (fn 6))) (list square 1+ even?))`

(Slightly) Harder Lists

1. Define a procedure `(depth ls)` that calculates the maximum depth of sublists 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 `(remove item ls)` that takes in a list and returns a new list with `item` removed from `ls`.

3. Define a procedure `(unique-elements ls)` that takes in a list and returns a new list without duplicates. You've already done this with `remove-dups`, and it used to do this:

   `(remove-dups '(3 5 6 3 3 5 9 8)) ==> (6 3 5 9 8)`
   where the *last* occurrence of an element is kept. We'd like to keep the *first* occurrences:
   `(unique-elements '(3 5 6 3 3 5 9 8)) => (3 5 6 9 8)`
   Try doing it without using `member?`. You might want to use `remove` above.

4. Define a procedure `(count-of item ls)` that returns how many times a given item occurs in a given list; it 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`

5. Define a procedure `(count-unique ls)` which, given a list of elements, returns a list of pairs whose `car` is an element and whose `cdr` is its number of occurrences in the list. For example,

```
(count-unique '(a b b b c d d a e e f a a))
=> ((a . 4) (b . 3) (c . 1) (d . 2) (e . 2) (f . 1))
```

You might want to use `unique-elements` and `count-of` defined above.

6. Define a procedure `(interleave ls1 ls2)` that takes in two lists and returns one list with elements from both lists interleaved. So,

```
(interleave '(a b c d) (1 2 3 4 5 6 7)) => (a 1 b 2 c 3 d 4 5 6 7)
```

7. 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)
```

Fake Plastic Trees

A tree is, abstractly, an acyclic, connected set of nodes (of course, that's not a very friendly definition). Usually, it is a node that contains two kinds of things – data and children. Data is whatever information may be associated with a tree, and children is a set of subtrees with a node as the parent. Concretely, it is often just a list of lists of lists of lists in Scheme, but it's best NOT to think of trees as lists at all. Trees are trees, lists are lists. They are completely different things, and if you, say, call `(car tree)` or something like that, that violates the 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 have 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 grouped into smaller-than and larger-than). Its operators are:

```
;; takes in a datum and a LIST of trees that will be the children of this
;; tree, and returns a tree.
(define (make-tree label 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 BELIEVE 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. Write `(listify-tree tree)` that turns the tree into a list in any order. (This one you can't use map even if you tried... Muwahahaha.)

Binary Search Trees

A binary search tree is a special kind of tree with an interesting restriction – each node only has two children (called the "left subtree" and the "right subtree", and every node in the left subtree has datum smaller than the node of the root, and every node in the right subtree has datum larger than the node of the root. Here are some operators:

```
;; takes a datum, a left subtree and a right subtree and make a bst
(define (make-tree datum left-branch right-branch) ...)

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

;; returns the left-subtree of this bst
(define (left-subtree bst) ...)

;; returns the right-subtree of this bst
(define (right-subtree bst) ...)
```

So then, let's get to it!

QUESTIONS

1. Jimmy the Smartass was told to write `(valid-bst? bst)` that checks whether a tree satisfies the binary-search-tree property – elements in left subtree are smaller than datum, and elements in right subtree are larger than datum. He came up with this:

```
(define (valid-bst? bst)
    (cond ((null? bst) #t)
          (else (and (or (null? (left-branch bst))
                         (and (< (datum (left-branch bst)) (datum bst))
                              (valid-bst? (left-branch bst))))
```

```
(or (null? (right-branch bst))
    (and (> (datum (right-branch bst)) (datumbst))
            (valid-bst? (right-branch bst))))))))
```

Why will Jimmy never succeed in life? Give an example that would fool his pitiful procedure.

2. Write `(sum-of bst)` that takes in a binary search tree, and returns the sum of all the data in the tree.

3. Write `(max-of bst)` that takes in a binary search tree, and returns the maximum datum in the tree. The tree has at least one element.

4. Write `(listify bst)` that converts elements of the given `bst` into a list. The list should be in NON-DECREASING ORDER!

5. Write `(remove-leaves bst)` that takes in a `bst` and returns the `bst` with all the leaves removed.

6. Write `(height-of tree)` that takes in a tree and returns the height – the length of the longest path from the root to a leaf.

Deep Lists

"Deep lists" are lists that contain sublists. You've already been working with them in the lab with `deep-reverse`, and in homeworks with `substitute2`. You'll find, however, that sometimes, they'll have recursive properties rather like those of general trees. Here's an example.

QUESTION

Consider the following Scheme representation for a hierarchical file system. A "file-entry" can either be a file or a directory, and it is represented by a list. The file entry for a file is a list whose first element is the word `FILE`, and the second element is the name of the file. The file entry for a directory is a list whose first two elements are the word `DIRECTORY` and the name of the directory, and whose remaining elements are file entries for the files within the directory (which may be directories themselves. For example,

```
(DIRECTORY proj2
    (DIRECTORY test)
    (FILE proj2.scm)
```

```
(DIRECTORY cheat
    (DIRECTORY my-friends-proj2
        (FILE proj2-2.scm)
        (FILE readme)
        (FILE transcript))
        (FILE proj2-copy.scm)))
```

Write a procedure `(file-list file-entry)` that, given a file entry for a directory, returns a list of names of the non-directory files anywhere in the corresponding directory tree. For example, given the file entry above, `file-list` should return the list

```
(proj2.scm proj2-2.scm readme transcript proj2-copy.scm),
```

not necessarily in that order.