CS 61A Lecture Notes First Half of Week 3

Topic: Hierarchical data

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

• Trees.

Big idea: representing a hierarchy of information.

Definitions: *node*, *datum*, *root*, *branch*, *leaf*, *parent*, *child.*

The name "tree" comes from the branching structure of the pictures, like real trees in nature except that they're drawn with the root at the top and the leaves at the bottom.

A *node* is a point in the tree. In these pictures, each node includes a *datum* (the value shown at the node, such as `France` or `26`) but also includes the entire structure under that datum and connected to it, so the `France` node includes all the French cities, such as `Paris`. Therefore, **each node is itself a tree**—the terms "tree" and "node" mean the same thing! The reason we have two names for it is that we generally use "tree" when we mean the entire structure that our program is manipulating, and "node" when we mean just one piece of the overall structure. Therefore, another synonym for "node" is "subtree."

The *root node* (or just the *root*) of a tree is the node at the top. Every tree has one root node. (A more general structure in which nodes can be arranged more flexibly is called a *graph*; you'll study graphs in 61B and later courses.)

The *children* of a node are the nodes directly beneath it. For example, the children of the `26` node in the picture are the `15` node and the `33` node. The parent of particular node is the node above it. Note that exactly one node has no parent (namely, the root node).

A *branch* node is a node that has at least one child. A *leaf* node is a node that has no children. (The root node is usually a branch node, except in the trivial case of a one-node tree.)

What are trees good for?

- Hierarchy: world, countries, states, cities.
- Ordering: binary search trees.
- Composition: arithmetic operations at branches, numbers at leaves.

• Below-the-line representation of trees.

Lisp has one built-in way to represent sequences, but there is no official way to represent trees. Why not?

- Branch nodes may or may not have data.
- Binary vs. n-way trees.
- Order of siblings may or may not matter.
- Can tree be empty?

We can think about a tree ADT in terms of a selector and constructors:

```
(make-tree datum children)
(datum node)
(children node)
```

The selector `children` should return a list (sequence) of the children of the node. These children are themselves trees. A leaf node is one with no children:

```
(define (leaf? node)
  (null? (children node)) )
```

This definition of `leaf?` should work no matter how we represent the ADT.

If every node in your tree has a datum, then the straightforward implementation is

```
;;;;;                         Compare file cs61a/lectures/2.2/tree1.scm
(define make-tree cons)
(define datum car)
(define children cdr)
```

On the other hand, it's also common to think of any list structure as a tree in which the leaves are words and the branch nodes don't have data. For example, a list like

```
(a (b c d) (e (f g) h))
```

can be thought of as a tree whose root node has three children: the leaf `a` and two branch nodes. For this sort of tree it's common not to use formal ADT selectors and constructors at all, but rather just to write procedures that handle the car and the cdr as subtrees. To make this concrete, let's look at mapping a function over all the data in a tree.

First we review mapping over a sequence:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES seq)
  (if (null? seq)
      '()
      (cons (SQUARE (car seq))
            (SQUARES (cdr seq)) )))
```

The pattern here is that we apply some operation (`square` in this example) to the data, the elements of the sequence, which are in the `car`s of the pairs, and we recur on the sublists, the `cdr`s.

Now let's look at mapping over the kind of tree that has data at every node:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES tree)
  (make-tree (SQUARE (datum tree))
             (map SQUARES (children tree)) ))
```

Again we apply the operation to every datum, but instead of a simple recursion for the rest of the list, we have to recur for *each child* of the current node. We use `map` (mapping over a sequence) to provide several recursive calls instead of just one.

If the data are only at the leaves, we just treat each pair in the structure as containing two subtrees:

```
;;;;;                         In file cs61a/lectures/2.2/squares.scm
(define (SQUARES tree)
  (cond ((null? tree) '())
        ((atom? tree) (SQUARE tree))
        (else (cons (SQUARES (car tree))
                    (SQUARES (cdr tree)) )) ))
```

The hallmark of tree recursion is to recurse for both the `car` and the `cdr`.

• **Mapping over trees**

One thing we might want to do with a tree is create another tree, with the same shape as the original, but with each datum replaced by some function of the original. This is the tree equivalent of `map` for lists.

```
;;;;;                        In file cs61a/lectures/2.2/tree1.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (map (lambda (t) (treemap fn t))
                  (children tree) )))
```

This is a remarkably simple and elegant procedure, especially considering the versatility of the data structures it can handle (trees of many different sizes and shapes). It's one of the more beautiful things you'll see in the course, so spend some time appreciating it.

Every tree node consists of a datum and some children. In the new tree, the datum corresponding to this node should be the result of applying `fn` to the datum of this node in the original tree. What about the children of the new node? There should be the same number of children as there are in the original node, and each new child should be the result of calling `treemap` on an original child. Since a forest is just a list, we can use `map` (not `treemap`!) to generate the new children.

• Mutual recursion

Pay attention to the strange sort of recursion in this procedure. `Treemap` does not actually call itself! `Treemap` calls `map`, giving it a function that in turn calls `treemap`. The result is that each call to `treemap` may give rise to any number of recursive calls, via `map`: one call for every child of this node.

This pattern (procedure `A` invokes procedure `B`, which invokes procedure `A`) is called *mutual recursion.* We can rewrite `treemap` without using `map`, to make the mutual recursion more visible:

```
;;;;;                        In file cs61a/lectures/2.2/tree11.scm
(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (forest-map fn (children tree))))

(define (forest-map fn forest)
  (if (null? forest)
      '()
      (cons (treemap fn (car forest))
            (forest-map fn (cdr forest)))))
```

`Forest-map` is a helper function that takes a forest, not a tree, as argument. `Treemap` calls `forest-map`, which calls `treemap`.

Mutual recursion is what makes it possible to explore the two-dimensional tree data structure fully. In particular, note that reaching the base case in `forest-map` does not mean that the entire tree has been visited! It means merely that one group of sibling nodes has been visited (a "horizontal" base case), or that a node has no children (a "vertical" base case). The entire tree has been seen when every child of the root node has been completed.

Note that we use `cons`, `car`, and `cdr` when manipulating a forest, but we use `make-tree`, `datum`, and `children` when manipulating a tree. Some students make the mistake of thinking that data abstraction means "always say `datum` instead of `car`"! But that defeats the purpose of using different selectors and constructors for different data types.

• Deep lists

Trees are our first two-dimensional data structure. But there's a sense in which any list that has lists as elements is also two-dimensional, and can be viewed as a kind of tree. We'll use the name *deep lists* for lists that contain lists. For example, the list

```
[[john lennon] [paul mccartney] [george harrison] [ringo starr]]
```

is probably best understood as a sequence of sentences, but instead we can draw a picture of it as a sort of tree:

Don't be confused; this is *not* an example of the Tree abstract data type we've just developed. In this picture, for example, only the "leaf nodes" contain data, namely words. We didn't make this list with `make-tree`, and it wouldn't make sense to examine it with `datum` or `children`.

But we can still use the *ideas* of tree manipulation if we'd like to do something for every word in the list. Compare the following procedure with the first version of `treemap` above:

```
;;;;;                        In file cs61a/lectures/2.2/tree22.scm
(define (deep-map fn lol)
  (if (list? lol)
      (map (lambda (element) (deep-map fn element))
       lol)
      (fn lol)))
```

The formal parameter `lol` stands for "list of lists." This procedure includes the two main tasks of `treemap`: applying the function `fn` to one datum, and using `map` to make a recursive call for each child.

But `treemap` applies to the Tree abstract data type, in which every node has both a datum and children, so `treemap` carries out both tasks for each node. In a deep list, by contrast, the "branch nodes" have children but no datum, whereas the "leaf nodes" have a datum but no children. That's why `deep-map` chooses only one of the two tasks, using `if` to distinguish branches from leaves.

Note: SICP does not define a Tree abstract data type; they use the term "tree" to describe what I'm calling a deep list. So they use the name `tree-map` in Exercise 2.31, page 113, which asks you to write what I've called `deep-map`. (Since I've done it here, you should do the exercise without using `map`.) SICP does define an abstract data type for *binary* trees, in which each node can have a `left-branch` and/or a `right-branch`, rather than having any number of children.

• **Car/cdr recursion**

Consider the deep list `((a b) (c d))`. Ordinarily we would draw its box and pointer diagram with a horizontal spine at the top and the sublists beneath the spine:

But imagine that we grab the first pair of this structure and "shake" it so that the pairs fall down as far as they can. We'd end up with this diagram:

Note that these two diagrams represent the same list! They have the same pairs, with the same links from one pair to another. It's just the position of the pairs on the page that's different. But in this new picture, the structure looks a lot like a binary tree, in which the branch nodes are pairs and the leaf nodes are atoms (non-pairs). The "left branch" of each pair is its `car`, and the "right branch" is its `cdr`. With this metaphor, we can rewrite `deep-map` to look more like a binary tree program:

```
;;;;;                          In file cs61a/lectures/2.2/tree3.scm
(define (deep-map fn xmas)
  (cond ((null? xmas) '())
        ((pair? xmas)
         (cons (deep-map fn (car xmas))
               (deep-map fn (cdr xmas))))
        (else (fn xmas))))
```

(The formal parameter `xmas` reflects the fact that the picture looks kind of like a Christmas tree.)

This procedure strongly violates data abstraction! Ordinarily when dealing with lists, we write programs that treat the car and the cdr differently, reflecting the fact that the car of a pair is a list element, whereas the cdr is a sublist. But here we treat the car and the cdr identically. One advantage of this approach is that it works even for improper lists:

```
> (deep-map square '((3 . 4) (5 6))
((9 . 16) (25 36))
```

• Tree recursion

Compare the car/cdr version of `deep-map` with ordinary `map`:

```
(define (map fn seq)
  (if (null? seq)
      '()
      (cons (fn (car seq))
        (map fn (cdr seq)))))
```

Each non-base-case invocation of `map` gives rise to one recursive call, to handle the cdr of the sequence. The car, an element of the list, is not handled recursively.

By contrast, in `deep-map` there are *two* recursive calls, one for the car and one for the cdr. This is what makes the difference between a sequential, one-dimensional process and the two-dimensional process used for deep lists and for the Tree abstraction.

A procedure in which each invocation makes more than one recursive call is given the name *tree recursion* because of the relationship between this pattern and tree structures. It's tree recursion only if each call (other than a base case) gives rise to two or more recursive calls; it's not good enough to have two recursive calls of which only one is chosen each time, as in the following non-tree-recursive procedure:

```
(define (filter pred seq)
  (cond ((null? seq) '())
    ((pred (car seq)) (cons (car seq) (filter pred (cdr seq))))
    (else (filter pred (cdr seq)))))
```

There are two recursive calls to `filter`, but only one of them is actually carried out each time, so this is a sequential recursion, not a tree recursion.

A program can be tree recursive even if there is no actual tree-like data structure used, as in the Fibonacci number function:

```
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1)) (fib (- n 2)))))
```

This procedure just handles numbers, not trees, but each non-base-case call adds the results of two recursive calls, so it's a tree recursive program.

• Tree traversal

Many problems involve visiting each node of a tree to look for or otherwise process some information there. Maybe we're looking for a particular node, maybe we're adding up all the values at all the nodes, etc. There is one obvious order in which to traverse a sequence (left to right), but many ways in which we can traverse a tree.

In the following examples, we "visit" each node by printing the datum at that node. If you apply these procedures to actual trees, you can see the order in which the nodes are visited.

Depth-first traversal: Look at a given node's children before its siblings.

```
;;;;;                          In file cs61a/lectures/2.2/search.scm
(define (depth-first-search tree)
  (print (datum tree))
  (for-each depth-first-search (children tree)))
```

This is the easiest way, because the program's structure follows the data structure; each child is traversed in its entirety (that is, including grandchildren, etc.) before looking at the next child.

Breadth-first traversal: Look at the siblings before the children.

What we want to do is take horizontal slices of the tree. First we look at the root node, then we look at the children of the root, then the grandchildren, and so on. The program is a little more complicated because the order in which we want to visit nodes isn't the order in which they're connected together.

To solve this, we use an extra data structure, called a *queue*, which is just an ordered list of tasks to be carried out. Each "task" is a node to visit, and a node is a tree, so a list of nodes is just a forest. The iterative helper procedure takes the first task in the queue (the car), visits that node, and adds its children at the end of the queue (using `append`).

```
;;;;;                        In file cs61a/lectures/2.2/search.scm
(define (breadth-first-search tree)
  (bfs-iter (list tree)))

(define (bfs-iter queue)
  (if (null? queue)
      'done
      (let ((task (car queue)))
        (print (datum task))
        (bfs-iter (append (cdr queue) (children task))))))
```

Why would we use this more complicated technique? For example, in some situations the same value might appear as a datum more than once in the tree, and we want to find the *shortest* path from the root node to a node containing that datum. To do that, we have to look at nodes near the root before looking at nodes far away from the root.

Another example is a game-strategy program that *generates* a tree of moves. The root node is the initial board position; each child is the result of a legal move I can make; each child of a child is the result of a legal move for my opponent, and so on. For a complicated game, such as chess, the move tree is much too large to generate in its entirety. So we use a breadth-first technique to generate the move tree up to a certain depth (say, ten moves), then we look for desirable board positions at that depth. (If we used a depth-first program, we'd follow one path all the way to the end of the game before starting to consider a different possible first move.)

For binary trees, within the general category of depth-first traversals, there are three possible variants:

Preorder: Look at a node before its children.

```
;;;;;                        In file cs61a/lectures/2.2/print.scm
(define (pre-order tree)
  (cond ((null? tree) '())
        (else (print (entry tree))
              (pre-order (left-branch tree))
              (pre-order (right-branch tree)) )))
```

Inorder: Look at the left child, then the node, then the right child.

```
;;;;;                        In file cs61a/lectures/2.2/print.scm
(define (in-order tree)
  (cond ((null? tree) '())
        (else (in-order (left-branch tree))
              (print (entry tree))
              (in-order (right-branch tree)) )))
```

Postorder: Look at the children before the node.

```
;;;;;                         In file cs61a/lectures/2.2/print.scm
(define (post-order tree)
  (cond ((null? tree) '())
        (else (post-order (left-branch tree))
              (post-order (right-branch tree))
              (print (entry tree)) )))
```

For a tree of arithmetic operations, preorder traversal looks like Lisp; inorder traversal looks like conventional arithmetic notation; and postorder traversal is the HP calculator "reverse Polish notation."

• Path finding

As an example of a somewhat more complicated tree program, suppose we want to look up a place (e.g., a city) in the world tree, and find the path from the root node to that place:

```
> (find-place 'berkeley world-tree)
(world (united states) california berkeley)
```

If a place isn't found, `find-place` will return the empty list.

To find a place within some tree, first we see if the place is the datum of the root node. If so, the answer is a one-element list containing just the place. Otherwise, we look at each child of the root, and see if we can find the place within that child. If so, the path within the complete tree is the path within the child, but with the root datum added at the front of the path. For example, the path to Berkeley within the USA subtree is

```
((united states) california berkeley)
```

so we put `world` in front of that.

Broadly speaking, this program has the same mutually recursive tree/forest structure as the other examples we've seen, but one important difference is that once we've found the place we're looking for, there's no need to visit other subtrees. Therefore, we don't want to use `map` or anything equivalent to handle the children of a node; we want to check the first child, see if we've found a path, and only if we haven't found it should we go on to the second child (if any). This is the reason for the `let` in `find-forest`.

```
;;;;;                         In file cs61a/lectures/2.2/world.scm
(define (find-place place tree)
  (if (eq? place (datum tree))
      (cons (datum tree) '())
      (let ((try (find-forest place (children tree))))
    (if (not (null? try))
        (cons (datum tree) try)
        '()))))

(define (find-forest place forest)
  (if (null? forest)
      '()
      (let ((try (find-place place (car forest))))
        (if (not (null? try))
            try
            (find-forest place (cdr forest))))))
```

(Note: In 61B we come back to trees in more depth, including the study of *balanced* trees, i.e., using special techniques to make sure a search tree has about as much stuff on the left as on the right.)