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*, *root*, *branch*, *leaf*.

A node is a particular point in the tree, but it's also a subtree, just as a pair *is* a list at the same time that it's a pair.

What are trees good for?

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

Many problems involve tree *search*: visiting each node of a tree to look for 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 search a sequence (left to right), but many ways in which we can search a tree.

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

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

Within the DFS category there are more kinds of orderings:

Preorder: Look at a node before its children.

Postorder: Look at the children before the node.

Inorder (binary trees only): Look at the left child, then the node, then the right child.

For a tree of arithmetic operations, preorder is Lisp, inorder is conventional arithmetic notation, postorder is HP calculator.

(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.)

• 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 recur for both the `car` and the `cdr`.