Binary Trees

Warning! This reading has been significantly rewritten for Fall 2015. Some errors may have crept in.

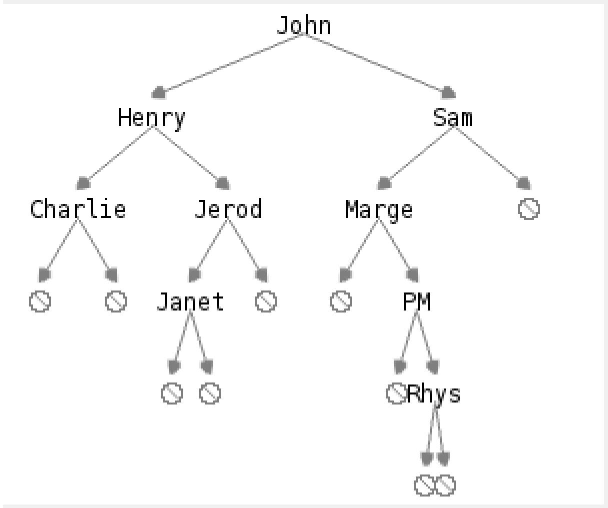
Summary: We consider trees, a way of organizing information hierarchically. Like lists, trees have a straightforward (but recursive) definition and a number of applications. And, as in the case of lists, you may find it useful to think about how trees are implemented.

In this reading, we explore a variety of issues of trees, including some terminology for talking and writing about trees, some formal definitions of trees, and some techniques for recursion over trees (using what is often called deep recursion).

Tree Basics

A list is a collection of values in which each value has at most one successor. (Every value but the last value has one successor; the last value has no successor.) While lists are useful for a wide variety of situations, there are situations in which they do not suffice. One is the problem of modeling a hierarcy. For example, the president of a company may have two vice presidents who report to him, her, or zir; each vice president may have assistant vice presidents who report to them; and so on and so forth.

We use the term tree for structures in which each value may have multiple successors (or underlyings, as in the example above). When each value may have at most two successors, we call them binary trees. In this reading, we will focus primarily on binary trees. Here's a sample binary tree written in the way that many computer scientists make them.



There are a number of important characteristics we associate with binary trees, including

- the root of a tree, which refers to the top of the tree;
- the size of a tree, the number of values stored in the tree;
- the depth of a tree, the length of the longest path from the top of the tree to the furthest value; and
- the base type of the tree, which indicates what kind of values the tree is built from. We call trees that have no single base type heterogeneous trees.

At any point in the binary tree built we usually refer to the two trees below it as the left subtree and the right subtree. We refer to the equivalent of a pair as a node. When both of the subtrees of a node are empty, we call that node a leaf.

In the sample tree, the root value is "John", its left subtree is rooted with "Henry" and its right subtree is rooted with "Sam". The size of the tree is nine because there are nine values in the tree. The depth of the tree is five for the John -> Sam -> Marge -> PM -> Rhys path. The leaves are Charlie, Janet, and Rhys.

Formalizing Trees

We have an informal definition of trees, given by the figure above. Can we define them formally? Yes. Just as we can define lists recursively, so can we define trees recursively. As you may recall, the definition of a list goes something like the following:

- The empty list (null) is a list.
- If `lst` is a list and `val` is any Scheme value, then `(cons val lst)` is a list.
- Nothing else is a list.

Why do we call this a “recursive definition”? Because the definition of a list is in terms of lists (at least in the middle item). We can write a similarly recursive definition for trees. We'll write one for binary trees, in which each value has two successors, rather than the one successor in lists.

- The special value `empty` is a binary tree.
- If `t1` and `t2` are trees, and `val` is a value, then `(node val t1 t2)` is a binary tree.
- Nothing else is a binary tree.

If all of the values have the same type, then we can refer to the tree in terms of the type name. For example, if all of the values are numbers, we would call it a “number tree”.

Binary Tree Operations

The values at the bottom of the tree we refer to as `empty`, which plays the same role that `null` plays in lists. We can check whether a value is that value with `empty?`.

As the description above suggests, we can build the tree with `node`, which plays the same role in binary trees that `cons` plays in lists. We can check if a value is a node with `node?`.

In lists, we get the value with `car` and the rest of the list with `cdr`. In binary search trees, we will use the more sensibly named `contents`, `left`, and `right`.

Patterns of Tree Recursion

As you should recall from our initial explorations of recursion, there is a traditional pattern for recursion over lists:

```
(define recursive-proc
  (lambda (lst)
      (if (null? lst)
          (base-case)
          (combine (car lst)
                   (recursive-proc (cdr lst) other)))))
```

We chose this pattern because of the common definition of a list. Because a list is either null or the cons of a value and a list we have two cases: one for when the list is null and one for the cons case. Since the cdr of a list is itself a list, in makes sense to recurse on the cdr.

A binary tree, in comparison, is either the empty tree or a node. If it's a node, it contains a value and two successors. Hence, we will need to recurse on both halves, as well as look at the value in the node.

```
(define tree-proc
  (lambda (tree)
      (if (empty? tree)
          (base-case other)
          (combine (contents tree)
                   (tree-proc (left tree))
                   (tree-proc (right tree))))))
```

We can use this pattern to count the number of values in a tree.

```
(define tree-size
  (lambda (tree)
    (if (empty? tree)
        0
        (+ 1
           (tree-size (left tree))
           (tree-size (right tree))))))
```

We can also use this pattern to find the depth of a tree. In this case, the depth of the empty the tree is 0, and the depth of any other tree is one higher than the depth of its largest subtree.

```
(define tree-depth
  (lambda (tree)
    (if (empty? tree)
        0
        (+ 1 (max (tree-depth (left tree))
                  (tree-depth (right tree)))))))
```

We can use a variant of this pattern to search the tree for a value.

```
(define tree-contains?
  (lambda (tree val)
    (cond
      [(empty? tree)
       #f]
      [(equal? (contents tree) val)
       #t]
      [(tree-contains? (left tree) val)
       #t]
      [(tree-contains? (right tree) val)
       #t]
      [else
       #f])))
```

Of course, if we find ourselves writing that many explicit `#t` and `#f`, we should probably rewrite it using the Boolean operations.

```
(define tree-contains?
  (lambda (tree val)
    (and (not (empty? tree))
         (or (equal? (contents tree) val)
             (tree-contains? (left tree) val)
             (tree-contains? (right tree) val)))))
```

You will note that we always use direct recursion, rather than helper recursion. That's because it's very difficult to express recursion on both subtrees using helper recursion.

Implementing Binary Trees

So, how do we implement these binary trees? One mechanism is to use pairs in a clever way. Since we have to store three values, rather than two, we can use two pairs.

```
(define node
  (lambda (val l r)
    (cons val (cons l r))))

(define contents car)

(define left cadr)

(define right cddr)
```

However, we may find it easier to implement binary trees using vectors. Why vectors? One reason is that they are a bit easier to read. We can also insert a special symbol (e.g., `'node`) to indicate that a vector is supposed to represent a node. With pairs, we will have trouble distinguishing lists from trees from other pair structures.

```
(define node
  (lambda (val l r)
    (vector 'node val l r)))

(define contents (r-s vector-ref 1))

(define left (r-s vector-ref 2))

(define right (r-s vector-ref 3))

(define node?
  (lambda (val)
    (and (vector? val)
         (= (vector-length val) 4)
         (equal? (vector-ref val 0) 'node))))
```

Self Checks

Check 1: Tree Recursion Patterns

Recall the pattern for tree recursion.

a. Identify the base case and combination of both `tree-size` and `tree-depth`.

b. Relate the two pieces for each function. How is the base case value related to (or working with) the combination step for each function?

c. Contrast the values of each of these pieces between the two functions. How and why do the base cases differ? How and why do the combinations differ?