HIERARCHICAL DATA 8

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

July 1, 2010

1 Sentences vs. Lists

My TA's have strongly objected to the characterization I gave yesterday and so here's the final word on that. Lists are NOT sentences, and sentences are NOT lists. We should think about sentences as something completely different from lists. Specifically, we should think about sentences as being an ADT (abstract data type) that is implemented using lists, but doesn't have to be. We should always respect the abstraction barrier between sentences and lists, by using the sentence constructs (`sentence`, `every`, `keep`, `first`, `butfirst`, etc) only on sentences, and using the list constructs (`list`, `car`, `cdr`, `cons`, `append`, etc) only on lists.

2 Trees from Last Lecture

2.1 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.

2.1.1 Exercise: Generalize!

Here's two examples. Generalize this into a pattern!

```
(define (square-tree tree)
  (make-tree (square (datum tree))
             (map (lambda (tr) (square-tree tr))
                  (children tree) )))
```

```
(define (cube-tree tree)
  (make-tree (cube (datum tree))
             (map (lambda (tr) (cube-tree tr))
                  (children tree) )))

(define (treemap fn tree)
  (make-tree (fn (datum tree))
             (map (lambda (tr) (treemap fn tr))
                  (children tree))))
```

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.

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).

3 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.

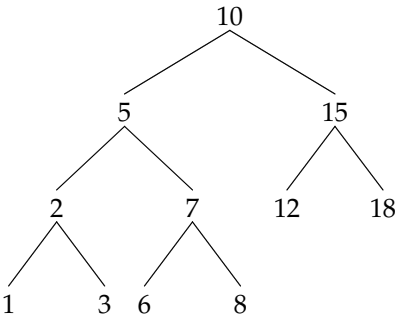
Note that we use `cons`, `car`, and `cdr` when manipulating a forest, but we use `make-tree`, `datum`, and `children` when manipulating a tree. Some students make the mistake of thinking that data abstraction means "always say datum instead of car"! But that defeats the purpose of using different selectors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

4 Binary Trees

Binary trees are particularly powerful ways of organizing information. You will study them extensively in CS61B, but for now, we want to introduce the data abstraction:

```
(make-binary-tree DATUM LEFT-BRANCH RIGHT-BRANCH)
(datum BINARY-TREE)
(left-branch BINARY-TREE)
(right-branch BINARY-TREE)
```

One particular special case of binary trees are binary search trees, which allow for rapid data lookup. They do so by forcing a property on every node. This property is that everything in the tree rooted at the right branch must be greater than the datum and everything in the tree rooted at the left branch must be less than the datum. To see that in action:



We'll deal only with a lookup operation and an insertion operation. Both these operations are detailed in the book, so I'll omit time in lecture going over them in excessive detail, but they're reproduced here.

```
(define (lookup number bst)
  (if (null? bst)
      #f ;Can't be in the tree!
      (cond ((= (datum bst) number) #t) ;found it!
            ((> (datum bst) number) (lookup number (left-branch bst)))
            ((< (datum bst) number) (lookup number (right-branch bst)) ))))

 (define (insert number bst)
    (if (null? bst)
        (make-binary-tree number '() '())
        (cond ((= (datum bst) number) bst) ;number already in tree!
              ((> (datum bst) number)
               (make-binary-tree (datum bst)
                                 (insert number (left-branch bst))
                                 (right-branch bst)))
              ((< (datum bst) number)
               (make-binary-tree (datum bst)
                                 (left-branch bst)
                                 (insert number (right-branch bst)))))))
```

5 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 look at a given node's children before its siblings.

5.1 Depth-First Traversal

```
;;;;; 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.

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."

5.2 Breadth-First Traversal

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.)

5.3 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))
      (list (datum tree)) ;We've found it! Return a list.
      (let ((attempt (find-forest place (children tree))))
           (if (not (null? attempt))
               (cons (datum tree) attempt) '()))))
```

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