COMP 200: Elements of Computer Science
Fall 2004
Lecture 13: September 24, 2004

More Abstractions: Trees & Graphs

On the Board

Homework 3: Due Monday
First Test: Due Wednesday at 5PM, DH 2065

Abstraction

Last class, we talked about the general issue of abstraction and about two specific data structures: a stack and a queue. Because stacks and queues inherently incorporate the notion of state — the outcome of an operation on a stack or queue depends on the history of prior actions — we had to use an assignment (written ← in our pidgin-Algol or written ***set!*** in Scheme) in our implementation of the stack primitives ***push*** and ***pop***.

In a similar way, arrays and vectors (recall *Keys* in our bubblesort algorithm from lecture 11) are inherently stateful. We could design a funky constructor that built a new array from an old array and a single new element or a single new row, but the result would be unnatural, inelegant, and (for those reasons) difficult to use. Scheme's support for arrays is clumsy — both notation and operation — so we won't use Scheme when we talk about arrays. However, we can build interesting data structures from the combination of language features that led to lists — the combination of ***define-struct*** and recursion.

Trees

As an example, consider the problem of representing your ancestry.

> ***Draw an example on the board, including at least four generations of family. Explain cousin relationships (1st cousin, 2nd cousin, once-removed, twice-removed, etc.)***

We call this a family tree. Why is it a tree?

> This tree encodes a single relationship — MyParents. The tree has roots — the children — and leaves — the great grandparents. (Of course, we can move the leaves back much farther in time with research, but one can assume that we have a finite supply of humans.)

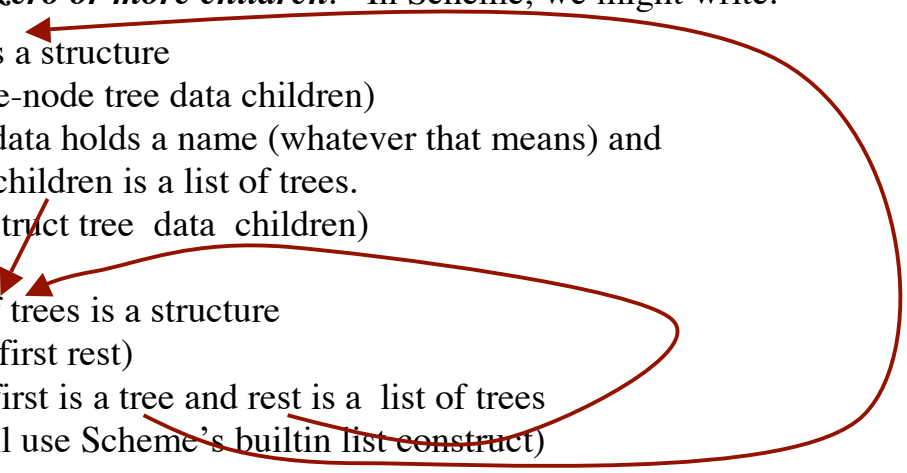
Looking at more members of the family, we can see that human families form multiple overlapping trees — and that we understand the complexity inherent in such structures. (What is a cousin? A second cousin?)

We can also derive trees that encode the relationship MyChildren. The two trees are related and the overall family trees are interwoven in a complex way. Still, we can see the relationship between them and navigate them.

As an abstract data structure, a tree consists of a set of ***nodes*****,** where each node can have ***zero or more children***. In Scheme, we might write:

```
; a tree is a structure
;    (make-node tree data children)
; where data holds a name (whatever that means) and
;           children is a list of trees.
(define-struct tree  data  children)

; a list of trees is a structure
;   (cons first rest)
; where first is a tree and rest is a  list of trees
; (we will use Scheme's builtin list construct)
```



If we want to write programs with these trees, we need a template for the tree structure and a template for list-of-trees. The templates are interwoven along the lines of the arrows.

Draw the templates

Programs written along these lines take on the complexity of the templates. If you apply the methodology and work through examples, it really is as easy as working with the lists that we have seen.

Binary Trees

To simplify the picture, computer scientists often work with a restricted class of trees, called ***binary trees***. A binary tree contains nodes that have at most two children (historically designated as ***left*** and ***right*** children). Limiting the definition to a small number of children avoids the need for a ***list*** in the definition which, in turn, simplifies the template and all the programs.

```
; a binary tree node (btn) is either
;    empty
;  or a structure
;    (make-node  data left right)
;  where data is a name (or whatever) and left and right are
;             binary-tree nodes
(define-struct  node  (data left right))
```

This definition leads to a much simpler template than the arbitrary trees we saw earlier (for ancestry).

```
; template for btn
(define (BTNProg  fee)
   (cond
     ((empty? fee)  … )
     (else
                … (node-data btn) …
                … (BTNProg (node-left btn)) …
                … (BTNProg (node-right btn)) … )
    ))
```

If ***data*** holds a number, we can easily write a program that sums all the numbers in a tree. — work it out at the board.

The book goes into tree sort as a natural use for trees.