Hierarchical Data 7

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

June 30, 2010

1 Review/Clarifications from Last Lecture

1.1 As Printed in Scheme

So you may have seen the dot in a pair in Scheme. I want you to know what that means. Scheme always will print a pair as a pair of parentesis with a period in between the `car` and the `cdr`. However, this would look really ugly in a list, and therefore, whenever you have a dot followed immediately by an open parens, they 'cancel out', and you're left with nothing. Thus, everything that is a proper list will print without periods, which is why you may not have seen them before.

In other words, if we do `(list 1 2 3)`, what we should get under normal rules is `(1 . (2 . (3 . ())))`, as opposed to `(1 2 3)`.

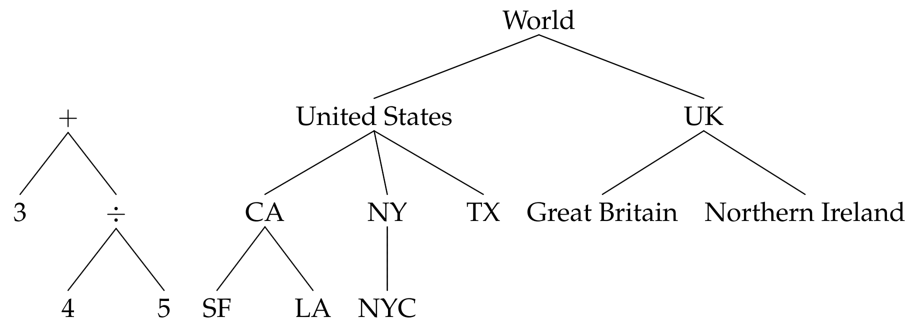
1.2 Lists and Sentences

RETRACTION. My TA's have strongly objected to the characterization that was originally here, 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

2.1 The Big Idea

Trees are a way we have of representing a hierarchy of information. The obvious example are family trees. You have a matriarch and a patriarch followed by all the descendants. Alternately, we may want to organize a series of information geographically. At the very top, we have the world, but below that we have countries, then states, then cities. We can also decompose arithmetic operations into something much the same way.



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.

2.2 Terminology

A NODE is a point in the tree. In these pictures, each node includes a DATUM (the value shown at the node, such as US or 4) 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. We'll look at graphs later in the course when we examine

The CHILDREN of a node are the nodes directly beneath it. For example, the children of the CA node in the picture are the SF node and the LA 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 also a branch node, except in the trivial case of a one-node tree.)

2.3 Variability between Different Trees

Scheme has one built-in way to represent sequences, which is the list, but there isn't a built-in way to represent trees. This isn't a deficiency in Lisp itself, but rather is a commonality that is present among many programming languages. This is because unlike lists, trees have tremendous variability in terms of what they represent. For examples:

- Branch Nodes may or may not have data
- Connections between nodes may or may not have data

- Number of Branches may be limited or not
- Order of Siblings may or may not matter
- Does it make sense to have an empty Tree?

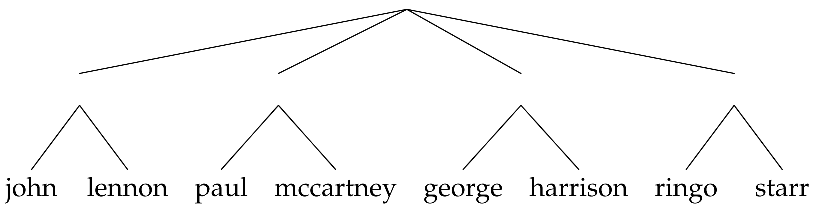
3 Deep Lists

3.1 Overview

In studying sentences and lists, we are looking at a flat sequence of information. Typically, this information tends to come in a homogenous form, in the sense that they are all words, or all numbers, or such. Furthermore, there isn't any real structure to this information. Information doesn't tend to interact or have any relationships with one another, beyond that of ordering.

When we talk about lists, we typically think that our `cdr`s point to more information whereas our `car` points to a single piece of information. However, consider what happens if our `car` can be either a list or a piece of information as well. What does that lead us to, and what is the point of that?

Let's take a look at something like this:



Instead of thinking of this as a Tree, it's easier to think of this as a list of sentences:

```
(define beatles (list (se 'john 'lennon) (se 'paul 'mccartney)
                      (se 'george 'harrison) (se 'ringo 'starr)))
```

3.2 Example: Manipulating a Deep List

What if we wanted to take the first of every word in the list? How do we make that call?

```
(map (lambda (sent) (every first sent) beatles)
```

3.3 Generalization: `deep-map`

Now, say we want to generalize that to be able to do any function to every single thing that isn't a pair in a deep list. How do we deal with that? Essentially, this is going to be a `map` procedure, but this time, you have to take into account that the car may or may not be a list. If it is, you want to `map` yourself over every spine pair. Otherwise, it is an atom, so you should be calling your function on it.

For reference, `map` is provided here:

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

Well, there's two ways you can write this. One way, is to change the (fn (car ls)) call to account for if the car is a list, as follows:

```
(define (deepmap fn deeplist)
  (if (null? deeplist)
      '()
      (cons (if (pair? (car ls))
                (deepmap fn (car ls))
                (fn (car ls)))
            (map fn (cdr deeplist)))))
```

The more elegant way, is to not look in advance at the bottom most level, but to just recurse into it, as follows:

```
(define (deepmap fn deeplist)
  (if (pair? deeplist)
      (map (lambda (dls) (deepmap fn dls)) deeplist)
      (fn deeplist)))
```

This is so powerful, and so short, that it's mindblowing. Try to trace it through some sample inputs to get a better sense of how this is working.

3.4 Characteristics

In a deep list, 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. The number of branches is unlimited, and while this example does not distinguish the order of its children, it could potentially matter.

4 Tree Abstract Data Type

We'll get back to some of these variations later, but first we'll consider a commonly used version of trees, in which every tree has at least one node, every node has a datum, and nodes can have any number of children. Here are the constructor and selectors:

`(make-tree datum children)`

`(datum node)`

`(children node)`

The selector `children` should return a *list of trees*. These children are themselves trees. There is a name for a list of trees: a *forest*. It's very important to remember that Tree and Forest are two different data types! A forest is just a list, although its elements are required to be trees, and so we can manipulate forests using the standard procedures for sequences (cons, car, cdr, etc.). A tree is **not** a sequence, and should be manipulated only with the tree constructor and selectors. A leaf node is one with no children, so its children list is empty:

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

This definition of `leaf?` should work no matter how we represent the ADT, but the simplest implementation of this ADT is as follows:

```
(define make-tree cons)
(define datum car)
```

```
(define children cdr)
```