

Class 30: Deep Recursion

Held: Tuesday, March 13, 2007

Summary: Today we consider a variant of recursion known as *deep recursion*. In this variant, you often recurse in multiple “directions”, particularly on the `car` of each pair as well as on the `cdr` of each pair.

Related Pages:

- EBoard.
- Reading: Deep Recursion.

Notes:

- Are there any final questions on the first project report?
- I will distribute the second take-home examination on Friday. You need not look at it until you return from break, but I'd like to give those of you who want something academic to do during break an opportunity.

Overview:

- Lists, revisited.
- Trees, introduced.
- Deep recursion, considered.
- Lab.

Review: What is a List?

- As you may have noticed from our work to date, lists are an essential part of Scheme programming.
- However, we have thought about lists in a number of different ways.
- When we see a list, we think of it as a group of data in order.
- When we build or use a list, we think of it in terms of the key operations, `cons`, `car`, `cdr`, `null`, and `null?`.
- We can also think of a list through its recursive definition. A list is either
 - The empty list or
 - Cons of a value and a list.
- Note that our procedural recursion typically uses this recursive data definition.

Complicating Structures: Nested Lists and Trees

- There are many interesting ways to combine values.
- As you've seen, lists can certainly contain other lists. We often refer to such lists as *nested lists*.
- How do we define nested lists? We might limit our definition above slightly

- A nested list is a list in which one of the component values is a list.
- Note that it will be harder to come up with a nice recursive definition, since some component values will be lists, and some will not.
- Note that there are other interesting things we can do with cons cells. In particular, if we don't limit the last value to null, we can define what is traditionally called a *tree*
 - Any simple value (string, symbol, procedure, number, character, or boolean) is a tree.
 - Cons of any two trees is a tree.
- Computer scientists have found a wide variety of applications for trees and tree-like structures.

Complicating Procedures: Deep Recursion

- Note that we may recurse differently over trees and nested lists.
- In particular, if the car of a list or pair is a list (or any pair structure), you might want to recurse over that structure, too.
- And if the car list has its own element list, we'd like to recurse on that list, too.
- Hence, in *deep recursion*, you recurse over both car and cdr of pairs.
- The typical form is

```
(define recursive-proc
  (lambda (tree)
    (if (pair? tree)
        (combine (recursive-proc (car tree))
                 (recursive-proc (cdr tree)))
        (base-case tree))))
```

- Does this interesting theoretical approach have any practical implications?
 - That is, "Are there ways we'd want to use this?"
- There are certainly many cases in which it helps to have a helper of this form.
- Note that when writing list-recursive (and number-recursive) procedures, we often wrote a helper which included something that accumulated results (we typically called that things *so-far*).
 - It is much more difficult to use this kind of helper for deep recursive procedures.
 - We will, nonetheless, still use husk-and-kernel techniques to check preconditions.

Lab

- Do the lab on deep recursion.

Copyright © 2007 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.