Fundamentals of Computer Science I (CS151.02 2007S)

# Deep Recursion

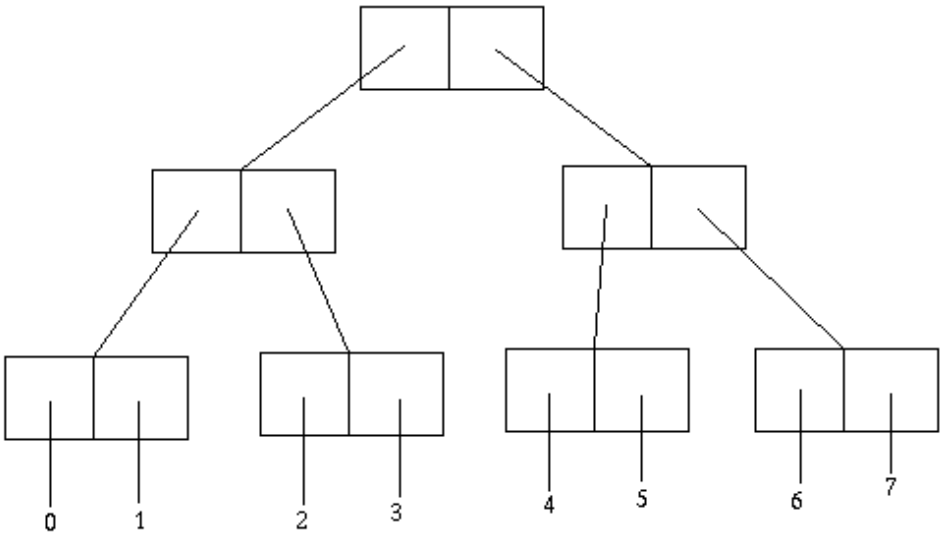
**Summary:** In our exploration of pair structures, we noted that you can build structures using `cons` that are not always lists. We typically call these structures *trees*. The technique for recursing over such structures is often called *deep recursion*.

**Contents:**


- Number Trees, Revisited
- Patterns of Tree Recursion
- Trees vs. Nested Lists


## Number Trees, Revisited

Recall the following structure from the reading on pairs and pair structures.




We typically call such a structure a *tree*. In this case, because all of the values in the tree are numbers, we call it a *number tree* or a *tree of numbers*

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

- the *size* of a tree, which indicates the number of values stored in the tree (sometimes the size also includes the number of pairs);
- the *depth* of a tree, which indicates the length of the longest path from the top of the tree to the furthest value; and
- the *base type* of the tree, such as "number" in number trees, 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 tree built with a pair, we usually refer to the parts below it as the *left subtree* (the car) and the *right subtree* (the cdr).

# 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 other)
      (if (null? lst)
          (base-case other)
          (combine other
                   (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 when for the cons case. Since the cdr of a list is itself a list, in makes sense to recurse on the cdr.

A tree, in comparison, is either a value or a pair of trees. Our base case is when we do not have a pair. Since both parts of the pair are trees, we should therefore recurse on both halves, rather than just the cdr. Hence, we define the common pattern for recursion over trees as follows.

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

For example, if we know that the tree contains only numbers, we can build `sum-of-number` tree by using `+` to combine the recursive calls and simply return `val` for the base case.

```
(define sum-of-number-tree
  (lambda (val)
    (if (pair? val)
        (+ (sum-of-number-tree (car val))
           (sum-of-number-tree (cdr val)))
        val)))
```

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

```
;;; Procedure:
;;;   depth
;;; Parameters:
;;;   tree, a tree
;;; Purpose:
;;;   Computes the length of the longest path from the start of the tree
;;;   to the furthest value.
;;; Produces:
;;;   dep, a nonnegative integer
;;; Preconditions:
;;;   None.
;;; Postconditions:
;;;   If tree contains no pairs, then dep is 0.
;;;   Otherwise, dep is one more than the depth of the deepest subtree.
(define depth
  (lambda (tree)
    (if (pair? tree)
        (+ 1 (max (depth (car tree))
                  (depth (cdr tree))))
        0)))
```

For example,

```
> (depth (cons 1 2))
1
> (depth (cons (cons 1 2) (cons 3 4)))
2
> (depth (cons (cons 1 (cons 2 3))
               (cons 4 5)))
3
```

We can use the pattern to find the number of values in a tree. In this case, if we have a non-tree, there's only one value. Otherwise, we need to combine the number of values in the left and right subtrees.

```
;;; Procedure:
;;;   number-of-values
;;; Parameters:
;;;   tree, a tree
;;; Purpose:
;;;   Count the number of values in tree.
;;; Produces:
;;;   count, a non-negative integer
;;; Preconditions:
;;;   (none)
;;; Postconditions:
;;;   count is the number of values in tree.  (pairs are not considered
;;;   values but null is considered a value).
(define number-of-values
  (lambda (tree)
    (if (pair? tree)
        (+ (number-of-values (car tree))
           (number-of-values (cdr tree)))
        1)))
```

For example,

```
> (number-of-values (cons 1 2))
2
> (number-of-values (cons 1 (cons 2 3)))
3
> (number-of-values (cons (cons 1 2) (cons 3 4)))
4
```

We can even use this pattern to “flatten” the tree into a list with the same values, in the order they appear from left-to-right. In this case, we turn non-paired values into lists, and append the results of flattening subtrees.

```
;;; Procedure:
;;;   flatten
;;; Parameters:
;;;   tree, a tree
;;; Purpose:
;;;   Convert a tree structure into a similar list structure.
;;; Produces:
;;;   lst, a list
;;; Preconditions:
;;;   none
;;; Postconditions:
;;;   lst is a simple list (that is, it contains no sublists).
;;;   Each value that appears in lst appears in tree.
;;;   Each value that appears in tree appears in lst.
;;;   If a precedes b in tree, then a precedes b in lst.
(define flatten
  (lambda (tree)
    (if (pair? tree)
        (append (flatten (car tree))
                (flatten (cdr tree)))
        (list tree))))
```

For example,

```
> (flatten (cons 1 2))
(1 2)
> (flatten (cons (cons 1 2) (cons 3 4)))
(1 2 3 4)
> (flatten (cons (cons 1 (cons 2 (cons 3 4)))
                 (cons 5 (cons 6 7))))
(1 2 3 4 5 6 7)
```

# Trees vs. Nested Lists

As you may recall, a simple list (such as a list of numbers) is simply a tree in which all the left subtrees (the car elements) are values and the final right subtree is null.

When we nest lists (that is, make lists elements of lists), we build structures that are very much like trees, except that we maintain the limitation that the rightmost subtree at every stage must be null.

For example, consider the list `(1 (2 3) (4 (5) (6 7)) (8 9))`. It has four elements (the `1`, the `(2 3)`, the `(4 (5) (6 7))` and the `(8 9)`), all but the first of which are themselves lists. We might consider it a number tree, except for the problem that the non-pair values include not just numbers, but also a variety of nulls: at the end of the top-level list, at the end of the list `(2 3)`, at the end of the next list and each of its sublists, and so on and so forth. Hence, if we try to apply the `sum-of-number-tree` procedure, it will try to add these `null` values and therefore stop with an error. We get similar problems when we try to use procedures like `flatten` and `number-of-values`.

```
> (define example '(1 (2 3) (4 (5) (6 7)) (8 9)))
> example
(1 (2 3) (4 (5) (6 7)) (8 9))
> (flatten example)
(1 2 3 () 4 5 () 6 7 () () 8 9 () ())
> (number-of-values example)
15
```

What do we do? We use a similar technique for nested list structures that we use for trees, except that we incorporate the chance that a value is null. For the case of `sum`, when we hit null, we return 0.

```
(define sum-of-nested-number-lists
  (lambda (lst)
    (cond
      ((null? lst) 0)
      ((pair? lst)
       (+ (sum-of-nested-number-lists (car lst))
          (sum-of-nested-number-lists (cdr lst))))
      (else lst))))
```

Here are some examples that contrast the behavior of `sum-of-nested-number-lists` with `sum-of-number-tree` and the old `sum` procedure we wrote when first exploring lists.

```
> (sum-of-nested-number-lists 1)
1
> (sum-of-nested-number-lists (list 1 2 3))
6
> (sum-of-nested-number-lists (list (list 1 2 3)))
6
> (sum 1)
car: expects argument of type <pair>; given 1
> (sum (list 1 2 3))
6
> (sum (list (list 1 2 3)))
+: expects type <number> as 1st argument, given: (1 2 3); other arguments were: 0
> (list 1 (list 2 3) (list 4 (list 5) (list 6 7)))
(1 (2 3) (4 (5) (6 7)))
> (sum-of-nested-number-lists (list 1 (list 2 3) (list 4 (list 5) (list 6 7))))
28
> (sum-of-number-tree (list 1 (list 2 3) (list 4 (list 5) (list 6 7))))
+: expects type <number> as 2nd argument, given: (); other arguments were: 3
```

Using this particular procedure as an example, we can also suggest a typical form for procedures that recurse over nested lists.

```
(define recursive-proc
  (lambda (val)
    (cond
      ((null? val) null-case)
      ((pair? val)
       (combine (recursive-proc (car val))
                (recursive-proc (cdr val))))
      (else (base-case val)))))
```

We might use this form to tally the number of times a symbol appears in a nested list structure (this is much like our count of the number of values in a tree, except we also check the type of the values we find). In this case, we return 0 for the null case and 1 or 0 for the base case (depending on whether val is a symbol or not).

```
;;; Procedure:
;;;   tally-symbols
;;; Parameters:
;;;   lst, a nested list structure
;;; Purpose:
;;;   Count the number of symbols that appear in the structure.
;;; Produces:
;;;   symbol-tally, a nonnegative integer
;;; Preconditions:
;;;   (none)
;;; Postconditions:
;;;   symbol-tally is the number of symbols at all levels of lst.
(define tally-symbols
  (lambda (val)
    (cond
      ((null? val) 0)
      ((pair? val)
       (+ (tally-symbols (car val))
          (tally-symbols (cdr val))))
      ((symbol? val) 1)
      (else 0))))
```

We can also tally the number of times a particular symbol appears by adding another parameter (the symbol) and changing the extra test.

```
;;; Procedure:
;;;   tally-symbol
;;; Parameters:
;;;   sym, a symbols
;;;   lst, a nested list structure
;;; Purpose:
;;;   Count the number of times the given symbol appears in the structure.
;;; Produces:
;;;   symbol-tally, a nonnegative integer
;;; Preconditions:
;;;   (none)
;;; Postconditions:
;;;   symbol-tally is the number of occurences of sym at all levels of lst.
(define tally-symbol
  (lambda (sym val)
    (cond
      ((null? val) 0)
```

```
      ((pair? val)
       (+ (tally-symbol sym (car val))
          (tally-symbol sym (cdr val))))
      ((eq? sym val) 1)
      (else 0))))
```

And, as these examples suggest, we can write a better version of `flatten` that works for both trees and nested lists.

```
(define flatten
  (lambda (tree)
    (cond
      ((null? tree) null)
      ((pair? tree)
       (append (flatten (car tree))
               (flatten (cdr tree))))
      (else (list tree)))))
```

As the following examples suggest `flatten` works for trees, nested lists, and even simple values.

```
> (flatten (cons (cons 1 (cons 2 (cons 3 4)))
                 (cons 5 (cons 6 7))))
> (define example '(1 (2 3) (4 (5) (6 7)) (8 9)))
> example
(1 (2 3) (4 (5) (6 7)) (8 9))
> (flatten example)
(1 2 3 4 5 6 7 8 9)
> (flatten 'a)
(a)
```

---

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.