# SICP section 2.2.2

August 10, 2007 at 08:34 Tags SICP

Tree-recursion, yay !

### Exercise 2.24

The interpretation is `(1 (2 (3 4)))`. I'll skip the drawings because they're too time consuming for such a simple exercise.

### Exercise 2.25

```
[22]> (setf lst '(1 3 (5 7) 9))
(1 3 (5 7) 9)
[23]> (car (cdr (car (cdr (cdr lst)))))
7

[24]> (setf lst '((7)))
((7))
[25]> (car (car lst))
7

[37]> (setf lst '(1 (2 (3 (4 (5 (6 7)))))))
(1 (2 (3 (4 (5 (6 7))))))
[38]> (cadr (cadr (cadr (cadr (cadr (cadr lst))))))
7
```

The last one is tricky, because `cdr` returns a list, so I use `cadr` (`car` of `cdr`) instead to get inside the lists.

### Exercise 2.26

```
[39]> (setf x '(1 2 3))
(1 2 3)
[40]> (setf y '(4 5 6))
(4 5 6)
[41]> (append x y)
(1 2 3 4 5 6)
[42]> (cons x y)
((1 2 3) 4 5 6)
[43]> (list x y)
((1 2 3) (4 5 6))
```

### Exercise 2.27

Similarly to `count-leaves`, we must differentiate between three cases:

1. nil 2. pair 3. not pair (atom)

The CL function for checking if something is a pair is `consp`[1]:

```
(defun deep-reverse (lst)
  (cond ((null lst) nil)
        ((consp (car lst))
          (append
            (deep-reverse (cdr lst))
            (list (deep-reverse (car lst))))) ; [A]
        (t
          (append
            (deep-reverse (cdr lst))
            (list (car lst))))))
```

Take a look at the line marked with [A]. The call to `list` is very important here, because otherwise `append` just stitches the contents of the list `deep-reverse` returns. `list` makes it append them as a single list, which is what we need.

## Exercise 2.28

```
(defun fringe (lst)
  (cond ((null lst) nil)
        ((not (consp lst)) (list lst))
        (t
          (append
            (fringe (car lst))
            (fringe (cdr lst))))))
```

## Exercise 2.29

**a.**

```
(defun make-mobile (left right)
  (list left right))

(defun left-branch (mobile)
  (first mobile))

(defun right-branch (mobile)
  (second mobile))

(defun make-branch (len structure)
  (list len structure))

(defun branch-len (branch)
  (first branch))

(defun branch-structure (branch)
  (second branch))
```

I'm using CL's convenience accessors for lists. `first` is equivalend to `car`, `second` to `cadr` (the `car` of the `cdr`). CL supports such accessors up to `tenth`, together with the generic accessor `nth`.

**b.**

I'm adding another abstraction – the predicate `structure-is-weight?`.

```
(defun structure-is-weight? (structure)
  (atom structure))

(defun weight-of-branch (branch)
  (let ((struct (branch-structure branch)))
    (if (structure-is-weight? struct)
      struct
      (weight-of-mobile struct))))

(defun weight-of-mobile (mobile)
  (+  (weight-of-branch (left-branch mobile))
      (weight-of-branch (right-branch mobile))))
```

Note how `weight-of-mobile` and `weight-of-branch` are defined. These are mutually recursive functions. Defining them this way makes for a very natural code, IMO.

**c.**

I'm going to use the same technique (mutually recursive functions) to figure out whether a given mobile is balanced:

```
(defun torque-of-branch (branch)
  (*  (branch-len branch)
      (weight-of-branch branch)))

(defun branch-balanced? (branch)
  "A branch is balanced either when it has a structure
  that's a simple weight, or when the structure is
  a balanced mobile"
  (let ((struct (branch-structure branch)))
    (or
      (structure-is-weight? struct)
      (mobile-balanced? struct))))

(defun mobile-balanced? (mobile)
  (let ((lb (left-branch mobile))
        (rb (right-branch mobile)))
    (and
      (=
        (torque-of-branch lb)
        (torque-of-branch rb))
      (branch-balanced? lb)
      (branch-balanced? rb))))
```

Note the *documentation string* added to `branch-balanced?`. It is a standartized feature of

Common Lisp, allowing me to find out what a function does from the REPL:

```
[5]> (documentation  'branch-balanced? 'function)
"A branch is balanced either when it has a structure
  that's a simple weight, or when the structure is
  a balanced mobile"
[6]>
```

**d.**

I only need to change the accessors `left-branch`, `right-branch`, `branch-len`, `branch-structure` and the predicate `structure-is-weight?`. The rest of the code builds on top of the abstraction created by these functions and doesn't need to be changed.

## Exercise 2.30

```
(defun square-tree-direct (tree)
  (cond ((null tree) nil)
        ((not (consp tree)) (square tree))
        (t
          (cons (square-tree-direct (car tree))
                (square-tree-direct (cdr tree))))))

(defun square-tree-map (tree)
  (mapcar
    (lambda (subtree)
      (if (consp subtree)
        (square-tree-map subtree)
        (square subtree)))
    tree))
```

## Exercise 2.31

```
(defun tree-map (func tree)
  (mapcar
    (lambda (subtree)
      (if (consp subtree)
        (tree-map func subtree)
        (funcall func subtree)))
    tree))
```

## Exercise 2.32

In set theory, the set of all subsets of some set S is called the *powerset* of S. So I'm naming the function accordingly. This is a piece of completely futile mathematical pedantry :-)

```
(defun powerset (s)
  (if (null s)
    (list nil)
    (let ((rest (powerset (cdr s))))
      (append
        rest
        (mapcar (lambda (r)
                  (cons (car s) r))
                rest)))))
```

To understand why this works, let's do the most natural thing and "walk over" the code mentally. So, the powerset of S is `nil` if S itself is `nil`. This is trivial. But what happens when S is a list ?

Well, it is somewhat similar to change counting from the previous sections. We "pick up" the first element of the list and concatenate two sets:

1. The powerset of all the elements without the first. 2. The powerset of all the elements without the first, with the first element prepended to each subset.

Consider the set `(1)`. Its powerset is `(nil (1))`. Now, consider the larger set `(2 1)`. According to the procedure above, its powerset is the concatenation of:

1. The powerset of all the elements without 2, that is `(1)`. Which we already saw is `(nil (1))`. 2. The element 2 prepended to each subset of the powerset of `(1)`, that is, 2 prepended to `nil` and 2 prepended to `(1)`, in total: `((2) (2 1))`

Concatenating the above two gives: `(nil (1) (2) (1 2))`, which is indeed the powerset of `(2 1)`. Using the same reasoning, we can compute the powerset of `(3 2 1)`. I hope it is clear by now why the procedure is correct.

[1] The convention of CL is to use the ending `p` for predicates, although it's not too consistent – consider the predicates `null` and `atom`, for example. Personally I prefer Scheme's idiom of ending a predicate with a question sign.

For comments, please send me an email.