



## SICP section 2.3.3

📅 September 11, 2007 at 05:04    **Tags** [SICP](#)

The solutions for this section are coded in Common Lisp.

### Exercise 2.59

The union of two sets is the combination of elements from both sets, without duplicates. For sets `set1` and `set2`, we will walk over the elements of `set2`, and keep only those that aren't already members of `set1`. The remaining elements will be appended to `set1`.

```
(defun union-set (set1 set2)
  (append
    set1
    (remove-if
      (lambda (x)
        (element-of-set? x set1))
      set2)))
```

### Exercise 2.60

Sets with duplicates are traditionally called *multisets*. `element-of-set?` is the same function. I'll implement it using CL's built in function `member`:

```
(defun element-of-multiset? (x set)
  (member x set :test #'equal))
```

Intersection is also similar:

```
(defun intersection-multiset (set1 set2)
  (cond ((or (null set1) (null set2)) '())
        ((element-of-multiset? (car set1) set2)
         (cons (car set1)
               (intersection-multiset (cdr set1) set2)))
        (t (intersection-multiset (cdr set1) set2))))
```

However, `adjoin-set` and `union-set` are much simpler. Since we no longer care about duplicates, we don't have to check that the elements we're adding to a set don't already exist in it:

```
(defun adjoin-multiset (x set)
  (cons x set))

(defun union-multiset (set1 set2)
  (append set1 set2))
```

Efficiency: `element-of-multiset?` and `intersection-multiset` will perform roughly the same as their regular set counterparts<sup>1</sup>. `adjoin-multiset` is  $O(1)$  (as opposed to  $O(1)$  for unordered-list sets) and `union-multiset` is  $O(n)$  (as opposed to  $O(n^2)$ , assuming that both sets have roughly  $n$  elements).

This representation makes sense when most of the operations we do on sets are unions and adjoins, of course.

### Exercise 2.61

```
(defun adjoin-set (x set)
  (cond ((null set) (cons x '()))
        ((< x (car set)) (cons x set))
        ((= x (car set)) set)
        (t (cons
             (car set)
             (adjoin-set x (cdr set))))))
```

### Exercise 2.62

```
(defun union-set (set1 set2)
  (let ((x1 (car set1)) (x2 (car set2)))
    (cond ((null x1) set2)
          ((null x2) set1)
          ((= x1 x2)
           (cons x1 (union-set (cdr set1) (cdr set2))))
          (< x1 x2)
           (cons x1 (union-set (cdr set1) set2)))
          (t
           (cons x2 (union-set set1 (cdr set2)))))))
```

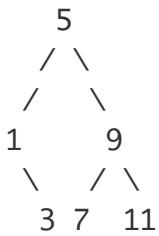
### Exercise 2.63

- Yes, both procedures produce the same result for all trees—the in-order traversal.
- There's no difference in the order of growth, both require  $O(n)$  steps.

### Exercise 2.64

- `partial-tree` divides the first  $n$  elements from `elts` into two groups—left and right, such that the sizes of the groups are  $n/2$  (the right group gets the extra element in case of an odd  $n$ ). It recursively creates a right subtree and a left subtree from these two groups and then `cons-es` them into the current node, producing a larger tree. It's a really beautiful example of recursive code, I must add.

The tree produced by it for the list (1 3 5 7 9 11) is (this should be printed out in fixed font):



b. The order of growth is  $O(n)$ , as eventually `partial-tree` traverses each element of the list once.

### Exercise 2.65

Since we now know how to:

- Convert from an ordered binary tree to an ordered list<sup>2</sup> in  $O(n)$
- Compute intersection and union of two ordered lists in  $O(n)$
- Convert back from an ordered list to a balanced binary tree in  $O(n)$

We can just combine the operations and end up with a  $O(n)$  procedure.

If we keep the name `union-set` for union computation on ordered lists, here is the union for binary trees:

```
(defun union-set-bintree (set1 set2)
  (let* ((lset1 (tree->list-1 set1))
         (lset2 (tree->list-1 set2))
         (lunion (union (union-set lset1 lset2)))
         (union (list->tree lunion)))
    union))
```

The intersection is very similar<sup>3</sup>:

```
(defun intersection-set-bintree (set1 set2)
  (let* ((lset1 (tree->list-1 set1))
         (lset2 (tree->list-1 set2))
         (lintersect (intersection-set lset1 lset2))
         (intersect (list->tree lintersect)))
    intersect))
```

### Exercise 2.66

The procedure is a simple conversion of the `element-of-set?` procedure:

```
(defun lookup (given-key set)
  (if (null set)
      nil
      (let* ((cur-entry (entry set))
             (cur-key (key cur-entry)))
        (cond ((= cur-key given-key) cur-entry)
              ((< given-key cur-key)
               (lookup
                given-key
                (left-branch set)))
              ((> given-key cur-key)
               (lookup
                given-key
                (right-branch set)))))))
```

Note that I've defined the following simple abstraction for storing key-value pairs in the set instead of simple values:

```
(defun make-record (key data)
  (list key data))

(defun key (record)
  (car record))

(defun data (record)
  (cadr record))
```

<sup>1</sup> They will be a little slower on average, since multisets are longer than sets (because of all the duplicates). This depends on the actual data a lot.

<sup>2</sup> Since `tree->list-1` (and `tree->list-2`) traverse the tree in-order, they will print out an ordered list for an ordered binary tree.

<sup>3</sup> Naturally, in real code we'd find away to reuse the common code, perhaps abstracting the relations between the same operators on different representations in a dispatch table.

---

For comments, please send me [✉](#) an email.