```
; copy-list
(lrec #'(lambda (x f) (cons x (funcall f))))

; remove-duplicates
(lrec #'(lambda (x f) (adjoin x (funcall f))))

; find-if, for some function fn
(lrec #'(lambda (x f) (if (fn x) x (funcall f))))

; some, for some function fn
(lrec #'(lambda (x f) (or (fn x) (funcall f))))
```

Figure 5.6: Functions expressed with `lrec`.

case must not be a value but a function, which we can call (if we want) in order to get a value.

Figure 5.6 shows some existing Common Lisp functions defined with `lrec`.[4] Calling `lrec` will not always yield the most efficient implementation of a given function. Indeed, `lrec` and the other recurser generators to be defined in this chapter tend to lead one away from tail-recursive solutions. For this reason they are best suited for use in initial versions of a program, or in parts where speed is not critical.

5.6 Recursion on Subtrees

There is another recursive pattern commonly found in Lisp programs: recursion on subtrees. This pattern is seen in cases where you begin with a possibly nested list, and want to recurse down both its car and its cdr.

The Lisp list is a versatile structure. Lists can represent, among other things, sequences, sets, mappings, arrays, and trees. There are several different ways to interpret a list as a tree. The most common is to regard the list as a binary tree whose left branch is the car and whose right branch is the cdr. (In fact, this is usually the internal representation of lists.) Figure 5.7 shows three examples of lists and the trees they represent. Each internal node in such a tree corresponds to a dot in the dotted-pair representation of the list, so the tree structure may be

[4]In some implementations, you may have to set `*print-circle*` to `t` before these functions can be displayed.

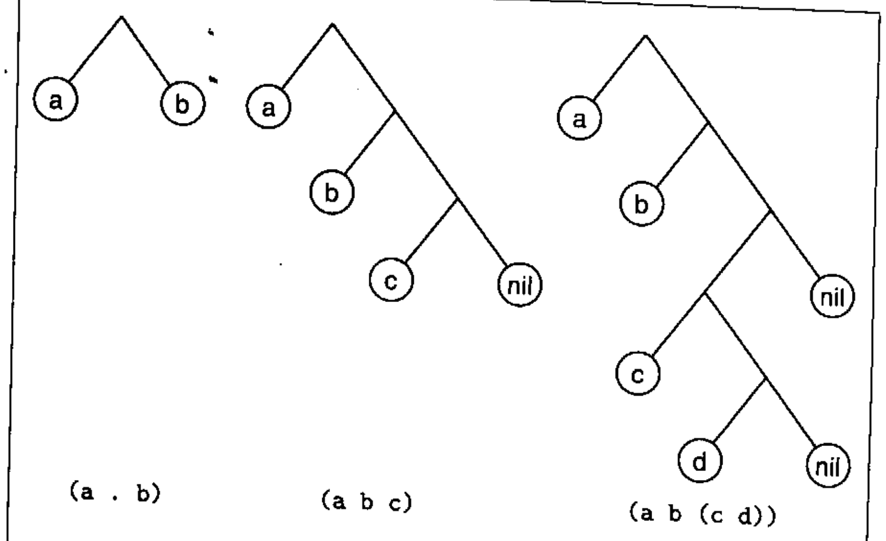

Figure 5.7: Lists as trees.

easier to interpret if the lists are considered in that form:

```
(a b c)      = (a . (b . (c . nil)))
(a b (c d))  = (a . (b . ((c . (d . nil)) . nil)))
```

Any list can be interpreted as a binary tree. Hence the distinction between pairs of Common Lisp functions like `copy-list` and `copy-tree`. The former copies a list as a sequence—if the list contains sublists, the sublists, being mere elements in the sequence, are not copied:

```
> (setq x      '(a b)
        listx (list x 1))
((A B) 1)
> (eq x (car (copy-list listx)))
T
```

In contrast, `copy-tree` copies a list as a tree—sublists are subtrees, and so must also be copied:

```
> (eq x (car (copy-tree listx)))
NIL
```