We really do not need `abs-all`, however, since the corresponding direct application of `map` is just as short and perhaps clearer.

```
(map abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)
```

Of course, we can use `lambda` to create the procedure argument to `map`, e.g., to square the elements of a list of numbers.

```
(map (lambda (x) (* x x))
     '(1 -3 -5 7)) ⇒ (1 9 25 49)
```

We can map a multiple-argument procedure over multiple lists, as in the following example.

```
(map cons '(a b c) '(1 2 3)) ⇒ ((a . 1) (b . 2) (c . 3))
```

The lists must be of the same length, and the procedure must accept as many arguments as there are lists. Each element of the output list is the result of applying the procedure to corresponding members of the input list.

Looking at the first definition of `abs-all` above, you should be able to derive, before studying it, the following definition of `map1`, a restricted version of `map` that maps a one-argument procedure over a single list.

```
(define map1
  (lambda (p ls)
    (if (null? ls)
        '()
        (cons (p (car ls))
              (map1 p (cdr ls))))))

(map1 abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)
```

All we have done is to replace the call to `abs` in `abs-all` with a call to the new parameter `p`. A definition of the more general `map` is given in Section 5.4.

Exercise 2.8.1. Describe what would happen if you switched the order of the arguments to `cons` in the definition of `tree-copy`.

Exercise 2.8.2. Consult Section 6.3 for the description of `append` and define a two-argument version of it. What would happen if you switched the order of the arguments in the call to `append` within your definition of `append`?

Exercise 2.8.3. Define the procedure `make-list`, which takes a nonnegative integer n and an object and returns a new list, n long, each element of which is the object.

```
(make-list 7 '()) ⇒ (() () () () () () ())
```


[*Hint*: The base test should be `(=` n `0)`, and the recursion step should involve `(-` n `1)`. Whereas `()` is the natural base case for recursion on lists, 0 is the natural base case for recursion on nonnegative integers. Similarly, subtracting 1 is the natural way to bring a nonnegative integer closer to 0.]

Exercise 2.8.4. The procedures `list-ref` and `list-tail` return the *nth* element and *nth* tail of a list *ls*.

```
(list-ref '(1 2 3 4) 0) ⇒ 1
(list-tail '(1 2 3 4) 0) ⇒ (1 2 3 4)
(list-ref '(a short (nested) list) 2) ⇒ (nested)
(list-tail '(a short (nested) list) 2) ⇒ ((nested) list)
```

Define both procedures.

Exercise 2.8.5. Exercise 2.7.2 had you use `length` in the definition of `shorter`, which returns the shorter of its two list arguments, or the first if the two have the same length. Write `shorter` without using `length`. [*Hint*: Define a recursive helper, `shorter?`, and use it in place of the length comparison.]

Exercise 2.8.6. All of the recursive procedures shown so far have been directly recursive. That is, each procedure directly applies itself to a new argument. It is also possible to write two procedures that use each other, resulting in indirect recursion. Define the procedures `odd?` and `even?`, each in terms of the other. [*Hint*: What should each return when its argument is 0?]

```
(even? 17) ⇒ #f
(odd? 17) ⇒ #t
```

Exercise 2.8.7. Use `map` to define a procedure, `transpose`, that takes a list of pairs and returns a pair of lists as follows.

```
(transpose '((a . 1) (b . 2) (c . 3))) ⇒ ((a b c) 1 2 3)
```

[*Hint*: `((a b c) 1 2 3)` is the same as `((a b c) . (1 2 3))`.]

2.9. Assignment

Although many programs can be written without them, assignments to top-level variables or `let`-bound and `lambda`-bound variables are sometimes useful. Assignments do not create new bindings, as with `let` or `lambda`, but rather change the values of existing bindings. Assignments are performed with `set!`.