Chapter 30

Mutation

For this chapter, switch languages in DrRacket to "Advanced Student Language".

30.1 Remembering changes

Suppose you wanted to keep track of a grocery shopping list. You could easily define a variable to hold such a list:

```
> (define groceries (list "milk" "eggs" "chocolate" "cheese"))
```

And you know how to write functions that search the list, count elements in the list, do something to every element in the list . . . but how do people *really* use shopping lists? At least in my house, there's a list stuck to the refrigerator. Every time I run out of an ingredient, or decide I need a particular ingredient for tomorrow's dinner, I add it to the list. The next time I go to the grocery store, I take the list with me, crossing things off the list as I put them in the cart. In other words, I'm *changing* the list all the time.

How would we do this in Racket? Well, we know how to add something to a list, sort of:

```
> (cons "broccoli" groceries)
(list "broccoli" "milk" "eggs" "chocolate" "cheese")
> groceries
(list "milk" "eggs" "chocolate" "cheese")
```

Notice that `cons` creates a new list one longer than the old one, but it doesn't change the old list.

For purposes of maintaining a shopping list, we'd really like a function that behaves like this:

```
> (add-grocery "broccoli")
> groceries
(list "broccoli" "milk" "eggs" "chocolate" "cheese")
> (add-grocery "cereal")
> groceries
(list "cereal" "broccoli" "milk" "eggs" "chocolate" "cheese")
```

In other words, we can find what's on the grocery list at any time by typing `groceries`, and we can *add* things to the list at any time by typing `(add-grocery` *`ingredient`*`)`, which *changes the value* of the variable `groceries`.

So far, we haven't seen any way to do that in Racket. Here's how.

30.2 Mutating variable values

Syntax Rule 11 `(set! variable expression)`

is an expression with no value. It evaluates expression*, and then* changes *the already-defined variable* variable *to have that value. If* variable *isn't already defined, it's illegal.*

SIDEBAR:

The exclamation point is part of the function name; Racketeers usually pronounce it "bang", as in "I used set-bang to change the grocery list." Recall the convention that functions that return a boolean usually have names ending in a question mark ("?"). There's a similar convention that functions that *modify* one of their arguments have names that end in an exclamation point.

With this in mind, we can do things like

```
> (define groceries (list "milk" "eggs" "chocolate" "cheese"))
> groceries
(list "milk" "eggs" "chocolate" "cheese")
>  (set!  groceries (list "broccoli" "milk" "eggs" "chocolate" "cheese"))
> groceries
(list "broccoli" "milk" "eggs" "chocolate" "cheese")
```

Of course, re-typing the whole list just to add `"broccoli"` to the front is a pain. But here's the magic: `set!` evaluates the expression *before* changing the variable, and there's no rule against the expression containing the same variable. So realistically, we would probably write something like

```
> (define groceries (list "milk" "eggs" "chocolate" "cheese"))
> groceries
(list "milk" "eggs" "chocolate" "cheese")
> (set!  groceries  (cons "broccoli" groceries))
> groceries
(list "broccoli" "milk" "eggs" "chocolate" "cheese")
```

Even more realistically, we could write the `add-grocery` function as follows:

Worked Exercise 30.2.1 ***Develop a function*** *`add-grocery` that takes in a string and adds that string to the list `groceries`.*

Solution: The assignment doesn't say what the function should *return*, because we're really interested in it more for its side effects than for its return value. There are two reasonable choices: we could have it return nothing, or we could have it return the new list of groceries. Let's first try returning nothing.

```
  ; add-grocery :  string -> nothing, but modifies groceries
```

How do we write test cases for such a function? Since it doesn't return anything, we can't use `check-expect` on the result of `add-grocery`. On the other hand, we need to be sure `add-grocery` is called *before* we look at `groceries`. This sounds like a job for `begin`:

```
(define groceries empty)
(check-expect (begin (add-grocery "cheese")
                     groceries)
              (list "cheese"))
(check-expect (begin (add-grocery "chocolate")
                     groceries)
              (list "chocolate" "cheese"))
(check-expect (begin (add-grocery "eggs")
                     groceries)
              (list "eggs" "chocolate" "cheese"))
```

The definition is easy, now that we know about `set!`:

```
(define (add-grocery item)
  (set!  groceries (cons item groceries)))
```

█

Worked Exercise 30.2.2 ***Modify*** *the definition of* `add-grocery` *so it returns the new grocery list (as well as modifying* `groceries`*).*

Solution: The contract changes to

```
; add-grocery :  string -> list of strings, and modifies groceries
```

The test cases change too, since now `add-grocery` is supposed to return something specific.

```
(define groceries empty)
(check-expect (add-grocery "cheese")
              (list "cheese"))
(check-expect groceries
              (list "cheese"))
(check-expect (add-grocery "chocolate")
              (list "chocolate" "cheese"))
(check-expect (add-grocery "eggs")
              (list "eggs" "chocolate" "cheese"))
(check-expect groceries
              (list "eggs" "chocolate" "cheese"))
```

The definition changes slightly:

```
(define (add-grocery item)
  (begin (set!  groceries (cons item groceries))
         groceries))
```

█

Exercise 30.2.3 ***Define*** *the variable* `age` *to be your current age.* ***Develop*** *a function* `birthday` *that takes no arguments, increases* `age` *by 1, and returns your new age.*

Exercise 30.2.4 ***Develop*** *a function* `buy-first` *that takes no arguments, returns nothing, and removes the first element of* `groceries` *(as though you had bought it and crossed it off the list). If* `groceries` *is empty, it should throw an appropriate error message.*

Exercise 30.2.5 ***Develop*** *a function `lookup-grocery` that takes in a string and tells whether that string appears in `groceries`.*

Write a bunch of test cases involving `add-grocery`, `buy-first`, and `lookup-grocery` in various sequences. (For example, if you add a particular item, then look it up, you should get `true`; if you then buy the first item and look it up again, you should get `false`.)

Hint: The function definition doesn't require `set!`.

Exercise 30.2.6 ***Develop*** *a function `buy` that takes in a string, returns nothing, and removes the specified string from `groceries`. If `groceries` is empty, or doesn't contain that string, it should throw an appropriate error message. If `groceries` contains more than one copy of the string, it should remove all of them.*

Write a bunch of test cases involving various combinations of `add-grocery`, `buy-first`, `buy`, and `lookup-grocery`.

Hint: The easiest way to write this function is to use non-mutating list functions to see whether the string appears in `groceries` and build the correct new grocery list, and then use `set!` to change the variable's value.

The functions `add-grocery`, `buy-first`, `lookup-grocery`, and `buy` illustrate a common situation in which we need mutation: when you want to provide *several functions* that share information (as well as needing to remember things from one call to the next).

Exercise 30.2.7 ***Develop a function*** *`next` that takes no arguments and returns how many times it has been called. For example,*

```
> (next)
1
> (next)
2
> (next)
3
```

Exercise 30.2.8 ***Develop a function*** *`reset` that resets the counter on `next` (see exercise 30.2.7) back to zero, and returns nothing. For example,*

```
> (next)
1
> (next)
2
> (next)
3
> (reset)
> (next)
1
```

Exercise 30.2.9 ***Develop a function*** *`next-color` that takes no arguments; each time you call it, it returns the next element in the list `(list "red" "orange" "yellow" "green" "blue" "violet")`. If you call it more than six times, it returns `false`. For example,*

```
> (next-color)
"red"
> (next-color)
"orange"
...
> (next-color)
"violet"
> (next-color)
false
```

30.3 Memoization

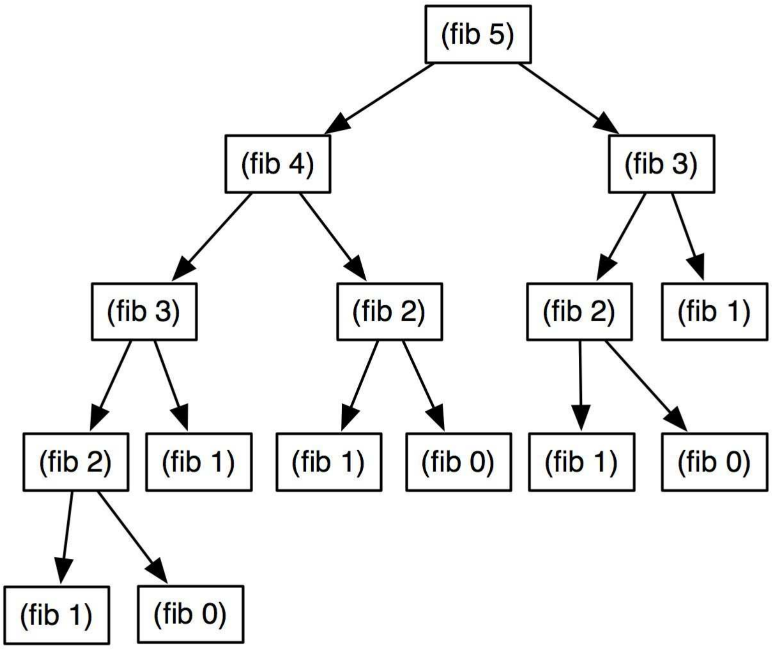
Recall Exercise 25.4.15, finding the longest sequence of characters that appear (in the same order) in two strings. In that exercise, you found a solution that works, but was probably fairly slow and inefficient. If you use the Stepper to watch what's going on in the execution of your function, you'll probably find that it's solving the exact same problem many times.

For another example of this, consider the Fibonacci function of Exercise 24.1.9. The "obvious" recursive definition works, but once n gets larger than about 20, it's surprisingly slow and inefficient. To illustrate this, let's change the test cases to use the `time` function, which displays how long it took to do something before returning the answer:

```
; fibonacci :  natural -> natural
(check-expect (time (fibonacci 0)) 1)
(check-expect (time (fibonacci 1)) 1)
(check-expect (time (fibonacci 2)) 2)
(check-expect (time (fibonacci 3)) 3)
(check-expect (time (fibonacci 4)) 5)
(check-expect (time (fibonacci 5)) 8)
(check-expect (time (fibonacci 6)) 13)
(check-expect (time (fibonacci 10)) 89)
(check-expect (time (fibonacci 15)) 987)
(check-expect (time (fibonacci 20)) 10946)
(check-expect (time (fibonacci 25)) 121393)
```

Exercise 30.3.1 ***Tabulate*** *how long the function takes on the above arguments.* ***Predict*** *approximately how long it will take on 30.* ***Try it.***

To see what's going wrong, let's use the Stepper on a simple example like `(fibonacci 5)`.



Notice that `(fib 3)` is called twice, `(fib 2)` three times, and `(fib 1)` five times. We're asking and answering the *exact same question* over and over.

We've seen something like this before: in Section 27.1 we had a function that called itself on the exact same question twice, and we made it much more efficient by using a local variable. Unfortunately, the `fibonacci` function doesn't call itself on the same question twice as "siblings" in the call tree, but rather as cousins, aunts, and more distant places in the call tree. So it's not clear how a local variable defined inside the function body could avoid this duplication. Instead, we'll need a more "global" solution.

Worked Exercise 30.3.2 *Modify the definition of* `fibonacci` *so it doesn't ask the same question over and over, and runs much faster.*

Solution: The contract and test cases are exactly the same as before; we're just trying to make things faster.

The idea of *memoization* is that once you answer a particular question, you *write down the answer* (a "memo to yourself") so the next time you are asked the same question, you can just return the same answer again rather than re-calculating it. Of course, this means every time you are asked a question, you first need to check whether you've been asked that question before. This calls for a mutable data structure to remember what we've been asked, and what the answer was last time.

One way to do this is to build up, in a global variable, a list of structures that each contain the question being asked and its answer.

```
; A pair has two natural numbers:  n and answer
(define-struct pair [n answer])

; *fibonacci-answers* is a list of pairs.
(define *fibonacci-answers* empty)
```

Every time fibonacci is called, it will start by looking in this global variable to see whether it already knows the answer; if so, it'll return it immediately. If not, it'll compute the answer as above, but before returning the answer, it will *record* the answer in the global variable.

```
(define (fibonacci n)
  (local [(define old-answer (lookup n *fibonacci-answers*))]
     (cond [(number? old-answer) old-answer]
           [(<= n 1) (record-and-return n 1)]
           [else (record-and-return n (+ (fibonacci (- n 1))
                                          (fibonacci (- n 2))))])))
```

This assumes we have two helper functions: a lookup function that looks in the table for a question and returns the answer, if known, or false if not; and a record-and-return function that takes in a question and its computed answer, adds that information to the table, and returns the answer. These are both easy:

```
; lookup : nat-num(n) list-of-pairs -> nat-num or false
(check-expect (lookup 4 empty) false)
(check-expect (lookup 4 (list (make-pair 4 12))) 12)
(check-expect (lookup 4 (list (make-pair 3 12))) false)
(check-expect
  (lookup 4 (list (make-pair 5 3) (make-pair 4 12) (make-pair 3 2)))
  12)
(define (lookup n pairs)
  (cond [(empty? pairs) false]
        [(= n (pair-n (first pairs)))
         (pair-answer (first pairs))]
        [else (lookup n (rest pairs))]))


; record-and-return : nat-num(n) nat-num(answer) -> nothing, but
;   modifies *fibonacci-answers*
(check-expect
  (begin (set! *fibonacci-answers*
          (list (make-pair 5 3) (make-pair 4 12) (make-pair 3 2)))
         (record-and-return 6 213))
  213)
(check-expect
  (begin (set! *fibonacci-answers*
          (list (make-pair 5 3) (make-pair 4 12) (make-pair 3 2)))
         (record-and-return 6 213)
         *fibonacci-answers*)
  (list (make-pair 6 213) (make-pair 5 3)
        (make-pair 4 12) (make-pair 3 2)))
(define (record-and-return n answer)
  (begin (set! *fibonacci-answers*
               (cons (make-pair n answer) *fibonacci-answers*))
         answer))
```

The resulting `fibonacci` function passes all the same tests as before, but much faster: on my old computer,

n	CPU time, simple version	CPU time, memoized version
10	4 ms	2 ms
15	40 ms	2 ms
20	366 ms	3 ms
25	4051 ms	4 ms

Exercise 30.3.3 *Let the mathematical function* fib3 *be as follows:*

- $fib3(0) = fib3(1) = fib3(2) = 1$
- $fib3(n) = fib3(n-1) + fib3(n-2) + fib3(n-3)$ *for* $n \geq 3$

Define a `fib3` *function in Racket, the obvious way without memoization.*
Define a `fib3-memo` *function in Racket, computing the same thing using memoization.*
Test that both functions produce the same answers on a variety of inputs.
*Compare (*e.g. *using the* `time` *function) the time it takes to compute each of the two. Try to* predict *how long it will take on a new input, and compare your prediction with the reality.*

Exercise 30.3.4 *Write a memoized version of the* `lcsubsequence` *function from Exercise 25.4.15. Run some timing tests: is it significantly faster?*

Hint: Since `lcsubsequence` takes *two* parameters rather than one, you'll need a table of structures that each have room for *both* parameters, as well as an answer.

Memoization and a closely-related technique called *dynamic programming* can sometimes turn an unacceptably-slow program into quite an efficient one; they're important techniques for professional programmers to know.

30.4 Static and dynamic scope

Recall the `add-grocery` function of Section 30.2. Suppose we called this inside a `local` definition of `groceries`:

```
(define groceries (list "milk" "eggs"))
(define (add-grocery item)
  (begin (set!  groceries (cons item groceries))
         groceries))
...
(local [(define groceries (list "tuna"))]
      (add-grocery "chocolate"))
```

Here's a puzzle: without actually typing in the code and running it, try to predict what this expression returns.

The problem is that we have *two different variables* named `groceries`: the one defined at the top level, and the one defined in the `local`. This is perfectly legal: the one in the `local` temporarily hides the outer one, as we've seen before. But which one does `add-grocery` look at and change? Is it the variable that was in effect when `add-grocery` was *defined* (returning `(list "chocolate" "milk" "eggs")`), or the variable that's in effect when `add-grocery` is *called* (returning `(list "chocolate" "tuna")`)?

Different programming languages have made this subtle decision differently. The former (use the variable in effect at the time the function was *defined*) is called *static scope*, because it depends on where the function is defined in the program source code, which doesn't change while the program is running. The latter (use the variable in effect at the time the function was *used*) is called *dynamic scope*, because it depends on which function calls which as the program is running.

Racket uses static scope, as do most programming languages. But not all: some versions of the Lisp language, which is closely related to Racket, use dynamic scope. For purposes of this book, you only need to worry about static scope. **Functions use the variables in effect at the time they were defined.**

30.5 Encapsulating state

We've seen a number of functions that "remember" something from one call to the next by changing the value of a global variable. This is somewhat inelegant, and raises possible security issues: if some user mistakenly changed the value of that variable him/her-self, our function would no longer work correctly. It would be cleaner if the function had its own private variable that nobody else could see.

Fortunately, `local` is very good at creating private variables that can only be seen in one small part of the program.

Exercise 30.5.1 ***Rewrite*** *the* `next` *function of exercise 30.2.7 to not use a global variable.*

Hint: See section 28.7 for ideas.

Exercise 30.5.2 ***Rewrite*** *the* `next-color` *function of exercise 30.2.9 to not use a global variable.*

Exercise 30.5.3 ***Rewrite*** *the* `fibonacci` *function from exercise 30.3.2 to avoid using a global variable (or struct, for that matter).*

Hint: Remember that the `record-and-return` function refers to the table variable, so it needs to be defined inside the scope of that variable.

Exercise 30.5.4 ***Rewrite*** *the* `fib3` *function from exercise 30.3.3 to avoid using a global variable (or struct, for that matter).*

Exercise 30.5.5 ***Develop a function*** `make-lister` *that takes in a list, and returns a function like* `next-color`*: it takes no arguments, but each time you call it, it returns the next element in the list that was given to* `make-lister`*. If you run out of list elements, it returns* `false`*. For example,*

```
> (define next-bird (make-lister (list "robin" "bluejay" "crow")))
> (define next-day
  (make-lister (list "sun" "mon" "tues" "wed" "thurs" "fri" "sat")))
> (next-bird)
"robin"
> (next-day)
"sun"
> (next-day)
"mon"
> (next-bird)
"bluejay"
> (next-day)
"tues"
```

Note: Note that if `make-lister` is called several times, each of the resulting functions *must* have its own "memory"; you *can't* use a global variable, but *must* use the technique of section 28.7.

Exercise 30.5.6 ***Rewrite*** *the* `lcsubsequence` *function of exercise 30.3.4 to avoid using a global variable (or struct, for that matter).*

Note: This one *must* be done with its own private table, not a global variable, because the right answers to sub-problems would be different if somebody called the function again on different strings.

Exercise 30.5.7 ***Develop a function*** `make-cyclic-lister` *that takes in a list, and returns a function that lists its elements, but once it gets to the end, it starts over from the beginning rather than returning* `false`*.*

SIDEBAR:

The Java and C++ languages don't allow you to define a variable locally and then create a function that modifies it, but they give you another way to encapsulate state: a *class* with *private instance variables* and *public methods* that modify those instance variables.

Exercise 30.5.8 ***Rewrite*** *the definitions of* `next` *and* `reset` *so they don't use a global variable (but they still talk to one another).*

Hint: You've already rewritten `next` by itself, in exercise 30.5.1, but now there's a difficulty: you need to define *two* functions inside the scope of the `local` and give them both top-level names. One way to do this is to have the body of the `local` return a list of two functions, store this in a top-level variable, then define `next` as one element of this list and `reset` as the other element.

The above solution has a couple of problems. First, it seems inelegant that we need to define a top-level variable in order to hold the list of two functions just so we can give them names. (One way around this is a Racket form named `define-values`, but that's not available in the Student languages.) Second, it's a fair amount of work just to build a resettable counter, and we'll have to do it all over again with different names if we want to build another.

Inspired by exercise 30.5.5, we might define a `make-counter` function that returns a two-element list of functions, the "next" and "reset" functions for this particular counter. That would solve the second problem, but not the first.

Another approach is to have `make-counter` return a function that serves both purposes. It takes in a string: if the string is `"next"`, it acts like `next`, and if it's `"reset"`, the function acts like `reset`. (If the string is anything else, it should produce an error message. And of course, if you'd prefer to use symbols rather than strings, that's fine, and slightly more efficient.) For example,

```
> (define count-a (make-counter))
> (define count-b (make-counter))
> (count-a "next")
1
> (count-a "next")
2
> (count-b "next")
1
> (count-a "next")
3
> (count-b "next")
2
> (count-a "reset")
> (count-a "next")
1
> (count-b "next")
3
```

SIDEBAR:

Languages such as Java and C++ have this technique built in; they call it "method dispatch". Now that you've seen how you could have done it by yourself, you may better appreciate having it built into the language.

Exercise 30.5.9 ***Develop*** *this* `make-counter` *function.*

Exercise 30.5.10 ***Develop*** *a function* `make-door` *that constructs a "door" object that represents whether a door is open or closed. A "door" object is a one-argument function:*

- *if the argument is "get-state", it returns the current state of the door, either "open" or "closed"*
- *if the argument is "open", it makes the state open, and returns nothing*
- *if the argument is "closed", it makes the state closed, and returns nothing*
- *if the argument is "toggle", it makes the state the opposite of what it was, and returns nothing.*

30.6 Mutating structures

When we learned about `define-struct` in chapter 21, we learned that certain functions "come for free": one constructor, one discriminator, and a getter for each field. In fact, another group of functions also "come for free": a *setter* for each field.

For example, you already know about `make-posn`, `posn?`, `posn-x`, and `posn-y`. There are also two other functions

```
; set-posn-x!  :  posn number -> nothing
; modifies the x coordinate of an existing posn
; set-posn-y!  :  posn number -> nothing
; modifies the y coordinate of an existing posn
```

Similarly, if you type

```
; An employee has a string (name) and two numbers (id and salary).
(define-struct employee [name id salary])
```

you get not only a constructor, a discriminator, and three getters, but also three setters:

```
; set-employee-name!  :  employee string -> nothing
; set-employee-id!  :  employee number -> nothing
; set-employee-salary!  :  employee number -> nothing
```

This has subtly different effects from `set!`. Consider

```
> (define joe (make-employee "joe" 386 80000))
> (define schmoe joe) ; two names for the same employee
> (define emps (list joe (make-employee "alice" 279 72000)))
> (set-employee-salary!  joe 85000)
> (employee-salary joe) ; 85000
> (employee-salary schmoe) ; 85000
> (employee-salary (first emps)) ; 85000
```

By contrast, if we changed joe's salary with `set!`, the results would be different:

```
> (define joe (make-employee "joe" 386 80000))
> (define schmoe joe) ; two names for the same employee
> (define emps (list joe (make-employee "alice" 279 72000)))
> (set!  joe (make-employee "joe" 386 85000))
> (employee-salary joe) ; 85000
> (employee-salary schmoe) ;  80000
> (employee-salary (first emps)) ;  80000
```

When we define `schmoe` to be `joe`, Racket now thinks of both variable names as referring to the same location in the computer's memory, so any change to the contents of that memory (as with `set-employee-salary!`) will be reflected in both. But `(set! joe ...)` tells Racket that the variable name `joe` should now refer to a different object in a different place in memory; `schmoe` still refers to the same thing it did before, and doesn't show a change.

Similarly, when we define `emps` to be a list containing `joe`, the list structure refers to the same location in memory that `joe` currently refers to. If `joe` is redefined to refer to something else, that doesn't change what the list refers to.

The phenomenon of two names referring to the same object in the computer's memory (rather than two objects with the same value) is called *aliasing*. We haven't had to worry about it until this chapter, because without setters and `set!`, there's no detectable difference between "two names for the same object" and "two objects with the same value". But professional programmers (in almost *any* language) have to worry about the difference.

Neither of these behaviors is inherently better than the other: sometimes you want one behavior and sometimes the other. The point is that before you write code that modifies things, you need to decide which of these behaviors you want.

Another interesting difference is that the first argument of `set!` *must* be a variable name:

```
(set! (first emps) ...)
```

wouldn't make sense. However,

```
(set-employee-salary! (first emps) ...)
```

makes perfectly good sense, and can be quite useful.

Worked Exercise 30.6.1 ***Develop a function*** *`give-raise!` that takes in an `employee` struct and a number (*e.g. *0.10 for a* 10% *raise), and* modifies *the employee to earn that much more than before.*

Solution:

```
; give-raise!  :  employee number -> nothing, but modifies the employee
(define joe (make-employee "joe" 386 80000))
(define schmoe joe) ; two names for the same employee
(define emps (list joe (make-employee "alice" 279 72000)))
(give-raise!  joe 0.10)
(check-expect (employee-salary joe) 88000)
(check-expect (employee-salary schmoe) 88000)
(check-expect (employee-salary (first emps)) 88000)
(give-raise!  (second emps) 0.15)
(check-expect emps
              (list (make-employee "joe" 386 88000)
                    (make-employee "alice" 279 82800)))

(define (give-raise!  emp percentage)
    ; emp                     an employee
    ; percentage              a number
    ; (employee-name emp)     a string
    ; (employee-id emp)       a number
    ; (employee-salary emp) a number
    (set-employee-salary!  emp
                           (* (+ 1 percentage) (employee-salary emp))))
```

■

Exercise 30.6.2 *Develop a function* `change-name-to-match!` *that takes in two* `person` *structures and* modifies *the first one to have the same last name as the second. Any other variables or lists that already referred to the first person should now show the changed name.*

Exercise 30.6.3 *Develop a function* `flip-posn!` *that takes in a* `posn` *and* modifies *it by reversing its x and y coordinates.*

Hint: You may need a `local` for this.

Exercise 30.6.4 *Develop a function* `give-raises!` *that takes a list of* `employees` *and a number, and gives them all that percentage raise.*

Develop a function `give-raises-up-to-100K!` *that takes a list of* `employees` *and a number, and gives that percentage raise to everybody who earns at most $100,000.*

Hint: It makes sense to do this with `map`, but `map` always returns a list of the same length as what it was given, even if the function it's calling doesn't return anything. So you may well get back a list of `(void)`'s: `void` is a built-in function that takes no arguments, does nothing with them, and returns nothing.

If this bothers you, you could rewrite `give-raise!` so it returns the modified employee; then `give-raises!` will naturally return a list of all the modified employees. What will `give-raises-up-to-100K!` return? What do you *want* it to return?

Exercise 30.6.5 *Develop a function* `ask-and-give-raises` *that takes in a list of* `employees`*. For each one in turn, it prints the person's name and current salary, asks (*via *console I/O) how much percentage raise to give, and does so. It should return the list of all the employees with their new salaries.*

30.7 Review of important words and concepts

The `set!` form changes the value of a variable.

Some functions are terribly inefficient because they call themselves recursively more than once. In particular, functions that, in the course of a call tree, ask the exact same question several times can often be improved through a technique called *memoization*: maintain a table of known function answers, and start each function call by checking whether you've already computed the answer.

When a function needs to remember things in a variable (*e.g.* for memoization), it's often safer to define the variable locally, so that only that function can see them. We *encapsulate* the state of the program in this hidden variable. In some cases (*e.g.* `make-lister` and `lcsubsequence`) it's not just a matter of safety. In addition, when several different functions need to share the same state variable, sometimes the best solution is to encapsulate the information into a single function that performs several different tasks.

The `set!` form modifies a variable definition, but sometimes it's more useful to modify *part* of a data structure, leaving it in place so that all existing references to it show the change. This can be done using *setters*, functions that "come for free" with `define-struct`.

30.8 Reference: Built-in functions for mutation and assignment

This chapter introduced one new function (technically a special form): `set!` . It also introduced the family of *setter* functions: when you do a `define-struct`, you get not only a constructor, a discriminator, and a getter for each field, but also a setter for each field. These have names like `set-posn-x!`, `set-employee-salary!`, *etc.*