

Principles of Programming Languages Handout

This is based on an anecdote I heard from a renowned computer scientist in a pub. He said he looked at the Lisp code of `map` at some time in the 1960s, saw a bunch of bizarre variable names in it, and has been horrified by dynamic binding ever since.

Recall that we have seen the procedure `map` in Scheme, and that Emacs Lisp is a lot like Scheme, but also very different in that it uses dynamic binding. Let us write `map` in Emacs Lisp:

```
(defun map (p L)
  (if (null L)
      (list)
      (cons
       (funcall p (car L))
       (map p (cdr L))))))
```

So far so good. We can use it as follows:

```
(let ((x 0))
  (map
   (lambda (n) x)
   (list 1 2 3)))
```

This evaluates to `(0 0 0)`, as one would expect. Now rename `x` to `L`:

```
(let ((L 0))
  (map
   (lambda (n) L)
   (list 1 2 3)))
```

This evaluates to `((1 2 3) (2 3) (3))`. We get a funny answer because of dynamic binding.

If we are sufficiently desperate to write `map` in the presence of dynamic binding, we could try the following. Who in his right mind would call variables `&^$^@**&&` or `$$$$^&^&^`? We just have to hope that the *user* of the procedure `map` does not come up with them.

```
(defun map (&^$^@**&& $$$$$^&^&^)
  (if (null $$$$$^&^&^)
      (list)
      (cons
       (funcall &^$^@**&& (car $$$$$^&^&^))
       (map &^$^@**&& (cdr $$$$$^&^&^)))))
```

Moral Static binding is necessary to keep abstraction boundaries; dynamic binding does not do this.