COMP 210: FALL 2000
Lecture 31: Memo functions and set!, again

Reminders

1. There will be one more homework–-a smaller two-week assignment.
2. Last exam will be handed out on Friday of the penultimate week of classes , due 1 week later.

Review (of Monday's Lecture)
We looked at how to speed p an expensive computation by remembering the results of previous computations. To do this, we needed to change the value associated with a name. We introduced the Scheme construct set! to accomplish this. Recall that set! takes two arguments, a variable and an expression. It evaluates the expression and causes the resulting value to become the variable's value.

We needed to hide the table of previously computed answers, so we used a cute trick with lambda.

```
(define f
  (lambda (x) (* x x)
```

is entirely equivalent to

```
(define (f x) (* x x))
```

Thus, we hid the table by writing the function f in the form

```
(define f
  (local [(define table empty)]
        (lambda (x) … )))
```

which binds the instance of table created by the local into the anonymous function defined by the lambda–-binds it by rewriting the body of the lambda, as a local does with anything that it defines–-and then saves the result of the local, which is the anonymous function returned by the lambda, as the value of **f**.   Note that define executes the local once–-when the define executes. The define creates an object named **f** and makes the lambda function be its value. Thus, table is defined and created once–-when f is created and filled with the lambda function. The body of the lambda function refers to the rewritten occurrences of table, so the function (now associated with f) refers uniquely and exclusively to this "hidden" instance of table.

Moving Forward

We saw how to change a single value. We used that change to implement the memory in a **memo-function** (a nerdy term for a function that remembers previous results and uses them to shortcut subsequent re-evaluations).

Until now, all the programs that we have written can be transformed with this memo function trick–-*because they always return the same result when given the same inputs!* In the process of developing the memo function, however, we introduced the Scheme construct set!–a feature that allows us to write programs that don't return the same results when given the same set of inputs.

A Brief Polemic

We have avoided set! to this point in the course precisely because it makes reasoning about the behavior of programs much more difficult. Set! lets us write programs (and expressions) whose results depend on things that happened earlier in the computation. (Perhaps, earlier in another computation…) This makes the simple rewriting rules for Scheme that we have used so far in the course somewhat more complex. It doesn't change the way that things work, but it does require that we keep track of much more context–-a subtle and difficult task, at best.

To reason about what a program does, in a world that includes set!, we need to keep track of what happens any time a set! occurs during execution. This is a lot more complex than just copying over the arguments, textually, as we make successive calls. You can write a program that goes deep into some recursion, does a set!, and returns. If that set! changed the value of a variable that is used elsewhere in the computation, you might not recognize it, or, even, be aware of it.

Consider the following simple program:

```
;; mystery: number → number
;; Purpose:  to puzzle 210 students
(define (mystery
        (local [(define memory 0)]
                (lambda (x) (begin (set! memory (add1 memory))
                                          (* memory  x  3/4)))
```

What does this program do? It is hard to derive its operation by calling it with a few trial arguments!

> (mystery 1) → 3/4
>
> (mystery 2) → 3
>
> (mystery 3) → 27/4
>
> (mystery 100) → 300
>
> (mystery 100) → 375 *… and so on, …*

Thus, you should only use set! in carefully chosen and carefully planned ways. The next several lectures will address those issues. Today's lab lecture will address these issues as well. Remember, the **!** is a warning–-to both the program and the reader.

[The discussion of hiding a local variable with a lambda took up so much time, that we stopped at this point.]