COMP 210, FALL 2000
Lecture 32: More fun with set! and set-structure!

Reminders

1. Last homework is due Friday, in class.
2. Exam will be available Friday, due one week later. It will be a closed-book, closed-notes, three-hour exam.

Review

We went over **set!** again, and revisited the example from lecture 30 (and, apparently, from lab) that confused people. It involved hiding a variable with a local that returns a lambda expression. We talked about some of the ways that set! makes analysis and understanding of programs harder.

More set!

Okay. We know how to change a single value. Let's work with that idea for a little bit today, in preparation for introducing data structures that need set! and **begin**. Recall that begin allows us to group together a set of expressions. It evaluates each of the expressions in normal fashion, and the value of the entire begin expression is simply the value of its final expression.

The begin construct is only useful if we have expressions whose effects deal with something other than returning a value. (Otherwise, there is no reason to put something in any slot of the begin except the last slot!) Thus, begin is useful with set! precisely because set! has an outside effect–-it changes the value of some variable. Begin is also useful with I/O operations, such as **printf** (which you saw in lab) and **write** and **newline** (which you didn't see in lab)

Consider the following trivial example:

```
(define n 5)
(begin
  (set!  n  (add1  n))
   n)
```

If we evaluate this, the define expression creates an object named **n** and gives it the value five. Evaluating the begin expression first evaluates the set!, which sets **n**'s value to 6, and then evaluates **n**, which returns 6.

What happens with this piece of code:

```
(define x 3)
(define y 4)
(begin
  (set!  x  y)
  (set!  y  x))
```

The defines create objects x & y that have the values 3 & 4, respectively. The begin expression looks like it should swap those values. However, that's not what happens.

<hand evaluate the expression>

Contrary to all the intuitions that we've built up over the past twelve weeks, this does not interchange the values of x & y. The first set! changes x's value by overwriting it with y's value. The second set! takes the value of x (which is now identical to the value of y) and uses it to overwrite the value of y. Thus, after executing the begin, x has the value of 4, as does (surprise) y. The net effect is the same as if we had just written two defines that set them to 4, or if we had never executed the second set!

Can we write a program **swap: number number → void** that takes two numbers and swaps their values? This should take less than two minutes.

```
;; swap: number number → void
;; Purpose: interchange the values of the argument numbers
(define (swap  x  y)
  (local  [(define temp  x)]
        (begin
          (set!  x  y)
          (set!  y  temp))))
```

This program uses a variable temp to preserve the value of x while it overwrites x with y's value. Then, it takes the preserved value (x's old value) and assigns it to y. Because the expressions inside the begin execute in sequential order rather than concurrently (at the same time, or in parallel), we need an extra place to hide one of the values. (Can you think of a way to do this without a temporary value?) [*Hint*: look up the logical function exclusive-or in a textbook for computer organization or computer architecture.]

But does this work? No, it does not work. If we execute it using DrScheme, we get the following behavior:

```
(define a  5)
(define b  6)
(swap  a  b)
a
> 5
b
> 6
```

Why? Remember our rewriting rules. When we rewrite (swap a b), what happens? The rewriting engine replaces any occurrences of a and b in the body of swap with their respective values. Does this mean that the set! expressions inside the body actually change the values of the constants?

What happens if we use the value of x again inside the value of swap?

```
;; swap: number number → void
;; Purpose: interchange the values of the argument numbers
(define (swap  x  y)
  (local  [(define temp  x)]
        (begin
          (set!  x  y)
          (set!  y  temp)
          x)))
```

If we use DrScheme to evaluate this version of swap, we get the following results.

```
(define a  5)
(define b  6)
(swap  a  b)
>  6
a
> 5
b
> 6
```

Swap returns 6–-the value of x, which is bound to a–-but outside swap the value of a remains unchanged. How can this happen? It's almost as if we changed the values associated with the constants. However, if we ask DrScheme to evaluate (write 6) inside the body of swap, DrScheme gets that correct. Similarly, if we ask DrScheme to evaluate

the expression containing just the number 6–-outside the body of swap–-it gets that correct. What happened?

This simple little program exposes one way in which the rewrite model, as explained this far, does not quite match the implementation in DrScheme. In fact, DrScheme implements it correctly. The rewrite model, as explained so far this semester, is a simplified version of reality. [Until we had set!, we could not tell the difference!] What really happens when it evaluates (swap a b) is that DrScheme creates new objects for the parameters x & y inside swap, and copies the value from a into x and the value from b into y. This completely explains the behavior that we have observed. The effects that we have seen from set! are both correct and appropriate–-maybe even *intuitive*–-once you know what really happens.

Again, set! changed the nature of computation. We need a more sophisticated model of the rewriting process to account for this brave new world.

An Example for set-structure!

Consider implementing an online address book. It needs at least two features–-you must be able to insert new addresses and you need to be able to look up a name and get back the phone number. We can represent addresses with a simple structure that has two fields.

```
;; An entry is a structure
;;   (make-entry Na Nu)
;; where Na is a symbol and Nu is a number
(define-struct entry (name number))


;; address-book : list of entry
;; keep track of the current address book entries
(define address-book empty)
```

Now we need two functions

```
;; lookup-number : symbol address-book → (number or false)
;; Purpose: returns the phone number associated with the symbol,
;;                    or false if the symbol is not found
(define (lookup-number name) ...)


;; add-to-address-book : symbol number → true
;; Purpose: adds the given name & number to the address book
(define (add-to-address-book name phone) ...)
```

Can we write down the test data for these programs? It is somehow more complex than the cases that we have seen in the past. If we try

(lookup-number ‘Todd)

the answer depends on what has happened since we last clicked the execute button in DrScheme. If we have already executed the expression

(add-to-address-book ‘Todd 7135551212)

then the call to lookup-number should return 7135551212. If we have never added ‘Todd to the address book, then it should return false.

To write down something that has definite results, we need a sequence of calls. For example, we can state that the sequence

(add-to-address-book ‘Keith 7133486013)

(lookup-number ‘Keith)

should always have the same results. The call to add-to-address-book returns true, and the call to lookup-number returns 7133486013. For programs that have persistent internal state, we need to write more complicated test data that ensures some knowledge of what is preserved in that internal state and then uses that knowledge for testing.

Both of these programs are pretty straight forward:

```
;; lookup-number : symbol → (number or false)
;; Purpose: returns the phone number associated with the symbol,
;;                  or false if the symbol is not found
(define (lookup-number name)
   (local [(define matches
                (filter (lambda (an-entry)
                          (symbol=? name (entry-name an-entry)))
                        address-book))]
          (cond
                [(empty? matches) false]
                [else (entry-number (first matches))])))
```

```
;; add-to-address-book : symbol number → true
;; Purpose: adds the given name & number to the address book
 (define (add-to-address-book name num)
    (begin
        (set! address-book
              (cons (make-entry name num) address-book))
         true))
```

Now, this still is COMP 210. Whenever we write a program that changes the value of a variable using set! (or set-structure!), we must document what those changes will be. Thus, we add a comment to the program, below the purpose, that describes any effects that the program has on variables that are defined outside of it. For example

```
;; add-to-address-book : symbol number → true
;; Purpose: adds the given name & number to the address book
;; Effect:    changes the value for address book to include an entry for name
 (define (add-to-address-book name num)
    (begin
        (set! address-book
              (cons (make-entry name num) address-book))
         true))
```

In this case, the effect comment is almost redundant, given the purpose. What would you write for the effect in the function mystery that we wrote earlier? The effect can be subtle and not obviously related to the function's purpose.

What happens when someone moves? How do we update the address book? We need a function **update** that takes a name and a number and changes the phone number for that name. How could you write this?

The classic approach, from the days when we thought primarily about structural recursion, would be to rebuild the phone book around a new entry for the person who moved. This would require searching through the phone book for the entry corresponding to name and rebuilding the list on the way back out of the recursion.

```
;; update-address:  symbol  number → void
;; Purpose: given a name and number, updates the phone number
;;              for that name
(define (update-address name num)
        (local [(define updated-book
                  (map (lambda (entry)
                         (cond
                           [(symbol=? (entry-name entry) name)
                             (make-entry name num)]
                           [else entry]))
                       address-book))]
          (set! address-book updated-book)
        ))
```

There is, however, reason that you might not want to do it that way. ***Efficiency.*** If your address book approaches the size of the White Pages ™ for Houston, you might want to avoid building and rebuilding it every time some customer moves. To let you build this more efficient version, Scheme includes a couple of functions for modifying the elements of a structure.

When we defined **entry**, the function **define-struct** also created

set-entry-name! : entry symbol → void

set-entry-number!: entry number → void

These both behave like set!, except that they apply to elements of an object that is, itself, an entry. Thus, the sequence

```
(define e1 (make-entry ‘Keith  7135276013))
e1
> (make-entry ‘Keith  7135276013)
(set-entry-number! e1  7133486013)
e1
>(make-entry `Keith 7133486013)
```

Using set-structure!, how can we write update-address?

```
;; update-address: symbol number → void
;; Purpose: given a name and number, updates the phone number
;;              for that name
;; Effect: changes the phone number stored with the given name
;;          in address book
(define (update-address-book! name new-num)
  (local [(define (helper! a-book)
            (cond [(empty? helper) void]
                  [else
                   (cond [(symbol=? name
                                    (entry-name (first a-book)))
                          (set-entry-phone! (first a-book) new-num)]
                         [else (helper! (rest a-book))])]))]
    (helper! address-book)))
```

Notice that this version of update-address does not use set! at all. It does not need to change the global variable address-book, because it changed one of the entries inside address-book directly. (The earlier version build a whole new address book to incorporate the change.)