

CS61A Notes – Week 9: Mutation

Revenge of the Box-and-pointers

I've been waiting for several weeks to bring up something you've noticed all along – there are different degrees of “sameness” in Scheme. Or, more specifically, things can be `equal?`, and things can be `eq?`. Now, you're finally old enough to know the truth.

`equal?` is used to compare **values**. We say two things are `equal?` if they evaluate to the same thing. For example, `(equal? '(2 3 4) '(2 3 4))` returns `#t`, since both are lists containing three elements: 2, 3, and 4. This is the comparison method that you're all familiar with.

`eq?`, however, is used to compare **objects** (not in the OOP sense of the word). We say two things are `eq?` if they point to the same object. For those of you proficient in C, you may think that `(eq? x y)` if `x` and `y` are both pointers holding the same address values. In other words, `(eq? '(2 3 4) '(2 3 4))` returns `#f`, because, though the two lists hold the same values, **they are not the same list!**

Consider this:

```
> (define x (list 1 2 3))
> (define y (list 1 2 3))
> (define z x)
```

1. What does `(eq? x y)` return?
2. What does `(eq? z x)` return?
3. How many lists are created in total?

Because you guys don't do the lab...

Take a look at this procedure from *SICP*, exercise 3.14:

```
(define (mystery x)
  (define (loop x y)
    (if (null? x)
        y
        (let ((temp (cdr x)))
            (set-cdr! x y)
            (loop temp x))))
  (loop x '()))

(define v (list 'a 'b 'c 'd))
(define w (mystery v))
```

To help clarify, `loop` uses the “temporary” variable `temp` to hold the old value of the `cdr` of `x`, since the `set-cdr!` on the next line destroys the `cdr`.

QUESTION

Draw box-and-pointer diagrams that show the structures v and w after evaluating those expressions. What does Scheme print for the values of v and w ?

Teenage Mutant Ninja... erm, Schemurtle (you try to do better)

Mutation refers to changing a data structure. Since our preferred data structures are pairs, naturally, then, to perform mutation on pairs, we have `set-car!` and `set-cdr!`. Note that `set-car!` and `set-cdr!` are NOT special forms! That's why you can execute things like `(set-car! (cdr lst) (+ 2 5))`.

To write procedures that deal with lists by mutation (rather than by constructing entirely new lists like we've done so far), here's a possible approach: first, try to do the problem without using mutation, as you normally would. Then, whenever you see `cons` used in your procedure, think about how you can modify the procedure to use `set-car!` or `set-cdr!` instead.

Do not confuse `set-car!` and `set-cdr!` with `set!`. `set!` is used to change the value of a *variable*, or, what some symbol in the environment points to. `set-car!` and `set-cdr!` are used to change the value *inside a cons pair*, and thus to change elements and structure of lists, deep-lists, trees, etc. They are *not the same!*

Also, in working with lists, you'll often find that you use `set-car!` to change elements of the list, and `set-cdr!` to alter the structure of the list. This shouldn't be a surprise – recall that in a list, the elements are the `car` of each pair, and the subsequent sublists are the `cdr`. But don't be fooled into thinking `set-car!` is always for element changes and `set-cdr!` is always for structural changes; in a richer data structure, either can be used for anything.

QUESTIONS

1. Personally – and don't let this leave the room – I think `set-car!` and `set-cdr!` are useless; we can just implement them using `set!`. Check out my two proposals for `set-car!`. Do they work, or do they work? Prove me wrong:

a.

```
(define (set-car! thing val)
  (set! (car thing) val))
```

b.

```
(define (set-car! thing val)
  (let ((thing-car (car thing)))
    (set! thing-car val)))
```

2. I'd like to write a procedure that, given a deep list, destructively changes all the atoms into the symbol colleen:

```
> (define ls `(1 2 (3 (4) 5)))
> (glorify! ls) => return value unimportant
> ls => (colleen colleen (colleen (colleen) colleen))
```

Here's my proposal:

```
(define (glorify! L)
  (cond ((atom? L)
        (set! L `colleen))
        (else (glorify! (car L))
              (glorify! (cdr L)))))
```

Does this work? Why not? Write a version that works.

3. We'd like to rid ourselves of odd numbers in our list:

```
(define my-1st `(1 2 3 4 5))
```

a. Implement (no-odd! ls) that takes in a list of numbers and returns the list without the odds, using mutation: (no-odd! my-1st) => `(2 4)

b. Implement (no-odd! ls) again. This time, it still takes in a list of numbers, but can return anything. But after the call, the original list should be mutated so that it contains no odd numbers.

Or,

```
(no-odd! my-1st) => return value unimportant
my-1st => `(2 4)
```

Hint: Try to consider if this is possible before you start!

4. Implement our old friend's ruder cousin, `(reverse! ls)`. It reverses a list using mutation. (This is a standard programming job interview question.)

5. Implement `(interleave! ls1 ls2)` that takes in two lists and interleaves them without allocating new cons pairs.

EXTRA PRACTICE

6. It would also be nice to have a procedure which, given a list and an item, inserts that item at the end of the list by making only one new cons cell. The return value is unimportant, as long as the element is inserted. In other words,

```
> (define ls `(1 2 3 4))
> (insert! ls 5) => return value unimportant
> ls => (1 2 3 4 5)
```

Does the following procedure work? If not, can you write one that does?

```
(define (insert! L val)
  (if (null? L)
      (set! L (list val))
      (insert! (cdr L) val)))
```

7. Write a procedure, `remove-first!` which, given a list, removes the first element of the list destructively. You may assume that the list contains at least two elements. So,

```
> (define ls `(1 2 3 4))
> (remove-first! ls) => return value unimportant
> ls => (2 3 4)
```

And what if there's only one element?

8. Implement `(deep-map! proc deep-ls)` that maps a procedure over every element of a deep list, without allocating any new `cons` pairs. So,

```
(deep-map! square `(1 2 (3 (4 5) (6 (7 8)) 9))) =>
  `(1 4 (9 (16 25) (36 (49 64)) 81))
```