# CS61A Week 10 | Now The Mutants Attack (v1.0)

## That Which Look The Same May Not Be The Same (Thy eyes are devil's idle playthings)

Now is a good time to bring up what 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**. We say two things are `eq?` if they point to the same object. For those of you proficient in C, you may think that `x eq? y` if `x` and `y` are both pointers holding the same address values.

Or, in short, `(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 these:

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

Then `(eq? x y)` returns `#f` but `(eq? z x)` returns `#t`. How many lists are created total?

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

**Mutation** refers to changing a data structure. Since our preferred data structure 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))`, with all arguments to `set-car!` being expressions that need to be evaluated.

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 fool 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, I think `set-car!` and `set-cdr!` are pretty useless too; we can just implement them using `set!`. Check out my two proposals for `set-car!` Do they work, or do they work?

   (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 `wei`:

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

   Here's my proposal:

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

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

3. 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?

## Just When You Were Getting Used to Lists...

Finally we are now introducing to you what many of you already know – arrays. You've already seen them countless times in lecture, so I won't go into them in detail. Roughly, an array is a contiguous block of memory - and this is why you can have "instantaneous", random access into the array, instead of having to traverse down the many pointers of a list. Recall the vector operators:

```
(vector [element1] [element2] ...) ;; works just like (list
[element1] ...) (make-vector [num]) ;; creates vector of length num,
all unbound (make-vector [num] [init-val]) ;; creates vector of
length num set to init-val (vector-ref v i) ;; v[i]; gets the ith
element of the vector v (vector-set! v i val) ;; v[i] = val; sets
the ith element of the vector v to val (vector-length v) ;; returns
the length of the vector v
```

Beyond using different operators, there are a few big differences between vectors and lists:

| **Vectors of length $N$** | **Lists of length N** |
|---|---|
| • a contiguous block of memory cells | • many units of two cells linked together by pointers |
| • $O(1)$ for accessing any item in the vector | • $O(N)$ for accessing an item |
| • $O(N)$ for adding an item to the middle of the vector, since you have to move the rest of the vector down | • $O(1)$ for inserting an item anywhere in the list, assuming we have a pointer to the location |
| • $O(N)$ for growing a vector; note that you have to reallocate another, larger block of memory! | • $O(1)$ for growing a list; just add it at the beginning or the end (if you have a pointer to the end) |
| • add 1 to index to get to the next element | • `cdr` down a list |
| • you may have "unbound" elements in the vector; that is, length of vector is not the same as length of your valid data | • length of list is exactly the number of elements you've put into the list |

Note the last bullet. With lists, you allocate a new piece of memory (using `cons`) when you need to add an element, but with vectors, you allocate all the room you need first, even if you don't have enough data to fill it up.

Also, just as you can have deep lists, where elements of a list may be a list as well, you can also have "deep" vectors, often referred to as $n$-dimensional arrays, where $n$ refers to how "deep" the deep vector is. For example, a table would be a 2-dimensional array - a vector of vectors. Note that, unlike in, say, C, each vector in your 2D table does NOT have to have the same size! Instead, you can have variable-length rows inside the same table. In this sense, the vectors of Scheme are more like the arrays of Java than C.

**QUESTIONS:**

1. Write procedure `(sum-of-vector v)` that adds up the numbers inside the vector. Assume all data fields are valid numbers.

2. Write procedure `(insert-at! v i val)`; after a call, vector `v` should have `val` inserted into location `i`. All elements starting from location `i` should be shifted down. The last element of `v` is disgarded.

```
STk> a ==> #(cs61a is cool #[unbound] #[unbound])

STk> (insert-at! 2 'very) ==> okay

STk> a ==> #(cs61a is very cool #[unbound])
```

3. Write procedure `(vector-double! v)`. After a call, vector `v` should be doubled in size, with all the elements in the old vector replicated in the second half. So,

```
STk> a ==> #(1 2 3 4)

STk> (vector-double! a) ==> okay

STk> a ==> #(1 2 3 4 1 2 3 4)
```