CS 61A Lecture Notes First Half of Week 5

Topic: Mutable data, queues, tables, vectors

Reading: Abelson & Sussman, Section 3.3.1–3

Play the animal game:

```
> (load "lectures/3.3/animal.scm")
#f
> (animal-game)
Does it have wings? no
Is it a rabbit? no

I give up, what is it? gorilla

Please tell me a question whose answer is YES for a gorilla
and NO for a rabbit.
Enclose the question in quotation marks.
"Does it have long arms?"
"Thanks.  Now I know better."
> (animal-game)
Does it have wings? no
Does it have long arms? no
Is it a rabbit? yes
"I win!"
```

The crucial point about this program is that its behavior changes each time it learns about a new animal. Such *learning* programs have to modify a data base as they run. We represent the animal game data base as a tree; we want to be able to splice a new branch into the tree (replacing what used to be a leaf node).

Changing what's in a data structure is called *mutation*. Scheme provides primitives `set-car!` and `set-cdr!` for this purpose.

They aren't special forms! The pair that's being mutated must be located by computing some expression. For example, to modify the second element of a list:

```
(set-car! (cdr lst) 'new-value)
```

They're different from `set!`, which changes the binding of a variable. We use them for different purposes, and the syntax is different. Still, they are connected in two ways: (1) Both make your program non-functional, by making a permanent change that can affect later procedure calls. (2) Each can be implemented in terms of the other; the book shows how to use local state variables to simulate mutable pairs, and later we'll see how the Scheme interpreter uses mutable pairs to implement environments, including the use of `set!` to change variable values.

The only purpose of mutation is efficiency. In principle we could write the animal game functionally by recopying the entire data base tree each time, and using the new one as an argument to the next round of the game. But the saving can be quite substantial.

Identity. Once we have mutation we need a subtler view of the idea of equality. Up to now, we could just say that two things are equal if they look the same. Now we need *two* kinds of equality, that old kind plus a new one: Two things are *identical* if they are the very same thing, so that mutating one also changes the other. Example:

```
> (define a (list 'x 'y 'z))
> (define b (list 'x 'y 'z))
```

```
> (define c a)
> (equal? b a)
#T
> (eq? b a)
#F
> (equal? c a)
#T
> (eq? c a)
#T
```

The two lists `a` and `b` are equal, because they print the same, but they're not identical. The lists `a` and `c` are identical; mutating one will change the other:

```
> (set-car! (cdr a) 'foo)
> a
(X FOO Z)
> b
(X Y Z)
> c
(X FOO Z)
```

If we use mutation we have to know what shares storage with what. For example, `(cdr a)` shares storage with `a`. `(Append a b)` shares storage with `b` but not with `a`. (Why not? Read the `append` procedure.)

The Scheme standard says you're not allowed to mutate quoted constants. That's why I said `(list 'x 'y 'z)` above and not `'(x y z)`. The text sometimes cheats about this. The reason is that Scheme implementations are allowed to share storage when the same quoted constant is used twice in your program.

Here's the animal game:

```
;;;;;                          In file cs61a/lectures/3.3/animal.scm
(define (animal node)
  (define (type l) (car l))
  (define (question l) (cadr l))
  (define (yespart l) (caddr l))
  (define (nopart l) (cadddr l))
  (define (answer l) (cadr l))
  (define (leaf? l) (eq? (type l) 'leaf))
  (define (branch? l) (eq? (type l) 'branch))
  (define (set-yes! node x)
    (set-car! (cddr node) x))
  (define (set-no! node x)
    (set-car! (cdddr node) x))

  (define (yorn)
    (let ((yn (read)))
      (cond ((eq? yn 'yes) #t)
            ((eq? yn 'no) #f)
            (else (display "Please type YES or NO")
                  (yorn)))))
```

```
  (display (question node))
  (display " ")
  (let ((yn (yorn)) (correct #f) (newquest #f))
    (let ((next (if yn (yespart node) (nopart node))))
      (cond ((branch? next) (animal next))
            (else (display "Is it a ")
                  (display (answer next))
                  (display "? ")
                  (cond ((yorn) "I win!")
                        (else (newline)
                              (display "I give up, what is it? ")
                              (set! correct (read))
                              (newline)
                               (display "Please tell me a question whose answer ")
                              (display "is YES for a ")
                              (display correct)
                              (newline)
                              (display "and NO for a ")
                              (display (answer next))
                              (display ".")
                              (newline)
                              (display "Enclose the question in quotation marks.")
                              (newline)
                              (set! newquest (read))
                              (if yn
                                  (set-yes! node (make-branch newquest
                                                              (make-leaf correct)
                                                              next))
                                  (set-no! node (make-branch newquest
                                                             (make-leaf correct)
                                                             next)))
                              "Thanks.  Now I know better."))))))

(define (make-branch q y n)
  (list 'branch q y n))

(define (make-leaf a)
  (list 'leaf a))

(define animal-list
  (make-branch "Does it have wings?"
               (make-leaf 'parrot)
               (make-leaf 'rabbit)))


(define (animal-game) (animal animal-list))
```

Things to note: Even though the main structure of the program is sequential and BASIC-like, we haven't abandoned data abstraction. We have constructors, selectors, and *mutators*—a new idea—for the nodes of the game tree.

• Tables. We're now ready to understand how to implement the `put` and `get` procedures that A&S used at the end of chapter 2. A table is a list of key-value pairs, with an extra element at the front just so that adding the first entry to the table will be no diffferent from adding later entries. (That is, even in an "empty" table we have a pair to `set-cdr!`)

```
;;;;;                          In file cs61a/lectures/3.3/table.scm
(define (get key)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        #f
        (cdr record))))

(define (put key value)
  (let ((record (assoc key (cdr the-table))))
    (if (not record)
        (set-cdr! the-table
                  (cons (cons key value)
                        (cdr the-table)))
        (set-cdr! record value)))
  'ok)

(define the-table (list '*table*))
```

`Assoc` is in the book:

```
(define (assoc key records)
  (cond ((null? records) #f)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records))) ))
```

In chapter 2, A&S provided a single, global table, but we can generalize this in the usual way by taking an extra argument for which table to use. That's how `lookup` and `insert!` work.

One little detail that always confuses people is why, in creating two-dimensional tables, we don't need a `*table*` header on each of the subtables. The point is that `lookup` and `insert!` don't pay any attention to the `car` of that header pair; all they need is to represent a table by *some* pair whose cdr points to the actual list of key-value pairs. In a subtable, the key-value pair from the top-level table plays that role. That is, the entire subtable is a value of some key-value pair in the main table. What it means to be "the value of a key-value pair" is to be the `cdr` of that pair. So we can think of that pair as the header pair for the subtable.

• Memoization. Exercise 3.27 is a pain in the neck because it asks for a very complicated environment diagram, but it presents an extremely important idea. If we take the simple Fibonacci number program:

```
;;;;;                          In file cs61a/lectures/3.3/fib.scm
(define (fib n)
  (if (< n 2)
      1
      (+ (fib (- n 1))
         (fib (- n 2)) )))
```

we recall that it takes $O(2^n)$ time because it ends up doing a lot of subproblems redundantly. For example, if we ask for `(fib 5)` we end up computing `(fib 3)` twice. We can fix this by *remembering* the values that we've already computed. The book's version does it by entering those values into a local table. It may be simpler to understand this version, using the global get/put:

```
;;;;;                          In file cs61a/lectures/3.3/fib.scm
(define (fast-fib n)
  (if (< n 2)
      n                                ; base case unchanged
      (let ((old (get 'fib n)))
        (if (number? old)              ; do we already know the answer?
            old
            (begin                     ; if not, compute and learn it
             (put 'fib n (+ (fast-fib (- n 1))
                            (fast-fib (- n 2))))
             (get 'fib n))))))
```

Is this functional programming? That's a more subtle question than it seems. Calling `memo-fib` makes a permanent change in the environment, so that a second call to `memo-fib` with the same argument will carry out a very different (and much faster) process. But the new process will get the same answer! If we look inside the box, `memo-fib` works non-functionally. But if we look only at its input-output behavior, `memo-fib` *is* a function because it always gives the same answer when called with the same argument.

What if we tried to memoize `random`? It would be a disaster; instead of getting a random number each time, we'd get the same number repeatedly! Memoization only makes sense if the underlying function really *is* functional.

This idea of using a non-functional implementation for something that has functional behavior will be very useful later in the course when we look at streams.

• **Vectors** So far we have seen one primitive data aggregation mechanism: the pair. We use linked pairs to represent *sequences* (an abstract type) in the form of *lists*.

The list suffers from one important weakness: Finding the nth element of a list takes time $O(n)$ because you have to call `cdr` $n-1$ times. Scheme, like most programming languages, also provides a primitive aggregation mechanism without this weakness. In Scheme it's called a *vector*; in many other languages it's called an *array,* but it's the same idea. Finding the nth element of a vector takes $O(1)$ time.

• **Vector primitives**

Some of the procedures for vectors are exact analogs to procedures for lists:

```
(vector a b c d ...)          (list a b c d ...)
(vector-ref vec n)            (list-ref lst n)
(vector-length vec)           (length lst)
```

Most notably, the *selector* for vectors, `vector-ref`, is just like the selector for lists (except that it's faster).

What about constructors? There's a `vector` procedure, just like the `list` procedure, that's good for situations in which you know exactly how many elements the sequence will have, and all of the element values, all at once. But there are no vector analogs to the list constructors `cons` and `append`, which are useful for *extending* lists. In particular, `cons` is the workhorse of recursive list processing procedures; we'll see that vector processing is done quite differently.

The weakness of vectors is that they can't be extended. You have to know the length of the vector when you create it. So instead of `cons` and `append` we have

```
(make-vector len)
```

which creates a vector of length `len`, in which the element values are unspecified. (You then use mutation, discussed below, to fill in the desired values.) Alternatively, if you want to create a vector in which every element has the same initial value, you can say

```
(make-vector len value)
```

Because vectors are created all at once, rather than one element at a time, *mutation* is crucial to any useful vector program. The primitive mutator for vectors is

```
(vector-set! vec n value)
```

This procedure is comparable to `set-car!` and `set-cdr!` for pairs. (It's interesting to note that Scheme doesn't provide a mutator for the nth element of a list; this is because most list processing is done using functional programming style, and pair mutation is mainly for special cases such as tables.)

The printed format of a vector is

```
#(a b c d)
```

You can quote this to include a constant vector in a program. (Note: In STk, vectors are self-evaluating, so you can omit the quotation mark, but this is a nonstandard extension to Scheme.)

Scheme also provides functions `list->vector` and `vector->list` that let you convert between the two sequence implementations.

• **Vector programming style**

Let's write a mapping function for vectors; it will take a function and a vector as arguments, and return a vector.

For reference, here's the `map` function for lists:

```
(define (map fn lst)
  (if (null? lst)
      '()
      (cons (fn (car lst))
        (map fn (cdr lst)))))
```

To do the same task for vectors, we must first create a new vector of the same length as the argument vector, then fill in the values using mutation:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (vector-map fn vec)
  (define (loop newvec n)
    (if (< n 0)
    newvec
    (begin (vector-set! newvec n (fn (vector-ref vec n)))
           (loop newvec (- n 1)))))
  (loop (make-vector (vector-length vec)) (- (vector-length vec) 1)))
```

This is a lot more complicated! It requires a helper procedure, and an extra *index variable*, `n`, to keep track of the element number within the vector. By contrast, the list version of `map` never actually knows how long its argument list is.

• **Strengths and weaknesses**

Of course, if we wanted, we could write our own equivalent to `cons` for vectors:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (vector-cons value vec)
  (define (loop newvec n)
    (if (= n 0)
    (begin (vector-set! newvec n value)
           newvec)
    (begin (vector-set! newvec n (vector-ref vec (- n 1)))
           (loop newvec (- n 1)))))
  (loop (make-vector (+ (vector-length vec) 1)) (vector-length vec)))
```

If we wrote similar procedures `vector-car` and `vector-cdr`, we could then write `vector-map` in a style exactly like `map`. But this would be a bad idea, because our `vector-cons` requires $O(n)$ time to copy the elements from the old vector to the new one.

operation	**lists**	**vectors**
nth element	`list-ref`, $O(n)$	`vector-ref`, $O(1)$
add new element	`cons`, $O(1)$	`vector-cons`, $O(n)$

This is why there isn't one best way to represent sequences. Lists are faster (and allow for cleaner code) at adding elements, but vectors are faster at selecting arbitrary elements.

(Note, though, that if you want to select *all* the elements of a sequence, one after another, then lists are just as fast as arrays. It's only when you want to jump around within the sequence that arrays are faster.)

• **Example: Shuffling**

Suppose we want to shuffle a deck of cards — we want to reorder the cards randomly. We'll look at three solutions to this problem.

First, here's a solution using functional programming with lists. Because we aren't allowing mutation of pairs, this version does a lot of recopying:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (shuffle1 lst)
  (define (loop in out n)
    (if (= n 0)
    (cons (car in) (shuffle1 (append (cdr in) out)))
    (loop (cdr in) (cons (car in) out) (- n 1))))
  (if (null? lst)
      '()
      (loop lst '() (random (length lst)))))
```

This is a case in which functional programming has few virtues. The code is hard to read, and it takes $O(n^2)$ time to shuffle a list of length n. (There are n recursive calls to `shuffle1`, each of which calls the $O(n)$ primitives `append` and `length` as well as $O(n)$ calls to the helper function `loop`.)

We can improve things using list mutation. Any list-based solution will still be $O(n^2)$, because it takes $O(n)$ time to find one element at a randomly chosen position, and we have to do that n times. But we can improve the constant factor by avoiding the copying of pairs that `append` does in the first version:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (shuffle2! lst)
  (if (null? lst)
      '()
      (let ((index (random (length lst))))
    (let ((pair ((repeated cdr index) lst))
          (temp (car lst)))
      (set-car! lst (car pair))
      (set-car! pair temp)
      (shuffle2! (cdr lst))
      lst))))
```

(Note: This could be improved still further by calling `length` only once, and using a helper procedure to subtract one from the length in each recursive call. But that would make the code more complicated, so I'm not bothering. You can take it as an exercise if you're interested.)

Vectors allow a more dramatic speedup, because finding each element takes $O(1)$ instead of $O(n)$:

```
;;;;;                          In file cs61a/lectures/vector.scm
(define (shuffle3! vec)
  (define (loop n)
    (if (= n 0)
    vec
    (let ((index (random n))
          (temp (vector-ref vec (- n 1))))
      (vector-set! vec (- n 1) (vector-ref vec index))
      (vector-set! vec index temp)
      (loop (- n 1)))))
  (loop (vector-length vec)))
```

The total time for this version is $O(n)$, because it makes n recursive calls, each of which takes constant time.

- **How it works**

One handwavy paragraph on why vectors have the performance they do:

A pair is two pointers attached to each other in a single block of memory. A vector is similar, but it's a block of n pointers for an arbitrary (but fixed) number n. Since a vector is one contiguous block of memory, if you know the address of the beginning of the block, you can just add k to find the address of the kth element. The downside is that in order to get all the elements in a single block of memory, you have to allocate the block all at once.

If you don't understand that, don't worry about it until 61B.