Lectures 18 & 19: Mutable data, tables, vectors

Summer 2006

July 26 & 27, 2006

Instructor: Ana Ramírez Chang

Administrative stuff

- Homework 5b out today
- Project 3 part I due tomorrow night
- Midterm 2
  - Friday
  - 4 – 6pm
  - 10 Evans hall
  - You can bring 2 sheets of paper
    - Possibly the one you made for midterm 1, plus another one.
  - Material covered listed on the web site

Animal game

```
STk> (load "lectures/3.3/animal.scm")
STk> (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."
STk> (animal-game)
Does it have wings? no
Does it have long arms? yes
Is it a gorilla? yes
"I win!"
```

Animal game

- Crucial point about game
  - Behavior changes each time it learns about a new animal.
  - This type of *learning* program has to modify a data base as it runs.
- Database represented as a tree
  - 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*

set-car! & set-cdr!

- Used to change what's in a data structure.
- They aren't special forms
  - Pair that's being mutated must be located by computing some expressions
  - Example: To modify the second element of a list
    - (set-car! (cdr lst) 'new-value)

set! & (set-car!, set-cdr!)

- Similar
  - Both make your program non-functional, by making a permanent change that can affect later procedure calls.
  - Each can be implemented in terms of the other
- Differences
  - set! changes the binding of a variable
  - set-car!, set-cdr! change a value in a pair

Mutation

- The only purpose of mutation is efficiency
- In principal, the animal game can be implemented by recopying the entire data base tree each time, using the new one as an argument to the next round of the game. But the savings can be quite substantial.

Identity

- Once we have mutation we need a subtler view of equality.
- We need two kinds of equality
  - the kind we've used up until now – two things are equal if they look the same
  - a new kind – two things are *identical* if they are the very same thing, so that mutating one also changes the other.

Example:

```
STk> (define a (list 'x 'y 'z))
STk> (define b (list 'x 'y 'z))
STk> (define c a)
STk> (equal? b a)
#T
STk> (eq? b a)
#F
STk> (equal? c a)
#T
STk> (eq? c a)
#T
```

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

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

Using mutation

- If we use mutation we have to know what share storage with what.
- For example, `(cdr a)` shares storage with `a`.
- `(append a b)` shares storage with `b` but not with a. Why not?
- The Scheme standard says you're not allowed to mutate quoted constants. That's why the slides say `(list 'x 'y 'z)` on the previous slide and not `'(x y z)`. The text sometimes cheats about this.

```
(define (animal node)
  (define (type node) (car node))
  (define (question node) (cadr node))
  (define (yespart node) (caddr node))
  (define (nopart node) (cadddr node))
  (define (answer node) (cadr node))
  (define (leaf? node) (eq? (type node) 'leaf))
  (define (branch? node) (eq? (type node) '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)))))

  (define (correctflag yorn)
    (cond (correct "I win!")
          (else (newline)
                (display "I give up, what is it? ")
                (let ((correct (read)))
                  (newline)
                  (display "Please tell me a question whose ")
                  (display "answer is YES for a ")
                  (display correct)
                  (newline)
                  (display "and NO for a ")
                  (display (answer next))
                  (display ".")
                  (newline)
                  (display "Enclose the question in ")
                  (display "quotation marks.")
                  (newline)
                  (let ((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 (animal node) continued
  (display (question node))
  (display " ")
  (let ((yn (yorn)))
    (let ((next (if yn (yespart node) (nopart node))))
      (cond ((branch? next) (animal next))
            (else (display "Is it a ")
                  (display (answer next))
                  (display "? ")
                  (let ((correctflag (yorn)))
                    (cond (correct "I win!")
                          (else (newline)
                                (display "I give up, what is it? ")
                                (let ((correct (read)))
                                  (newline)
                                  (display "Please tell me a question whose ")
                                  (display "answer is YES for a ")
                                  (display correct)
                                  (newline)
                                  (display "and NO for a ")
                                  (display (answer next))
                                  (display ".")
                                  (newline)
                                  (display "Enclose the question in ")
                                  (display "quotation marks.")
                                  (newline)
                                  (let ((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))
```

Tables

- 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 different from adding later entries.
- (That is, even in an "empty" table we have a pair to set-cdr!)

Tables

```
(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*))

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

Memoization

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
       (fib (- n 2)) )))
(define (fast-fib n)
  (if (< n 2)
      n
      (let ((old (get 'fib n)))
        (if (number? old)
            old
            (begin
             (put 'fib n (+ (fast-fib (- n 1))
                            (fast-fib (- n 2))))
             (get 'fib n))))))
```

- `fib` takes $\Theta(2^n)$ because it ends up doing a lot of sub problems redundantly. For example, if we ask for `(fib 5)` we end up computing `(fib 3)` twice.
- We can fix this problem by *remembering* the values that we've already computed.

Memoization

- You can only use memoization to speed up functions that are *function* or always return the same value for a given input value.
- For example, you cannot use memoization with `random`.

Vectors

- Weakness of lists:
  - Finding the n^{th} element takes $\theta(n)$ because you have to call `cdr` $n-1$ times.
- Vectors
  - Finding the n^{th} element takes $\theta(1)$.
  - In other languages, sometimes called an array
- Vector primitives and analogous list primitives

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

- There are no vector analogs to the list constructors `cons` and `append` which are useful for extending lists.
- Weakness of vectors: they cannot be extended, you have to know the length when you create it.
  - `(make-vector len)` instead of `cons` and `append`

Vectors

- *Mutation* is crucial since vectors are created all at once
  - `(vector-set! vec n value)`
  - Analogus to `set-car!` and `set-cdr!`
- Printed format of a vector is
  - `#(a b c d)`
- Scheme also provides
  - `list-vector` and `vector->list`

Vectors

- Write functions for vectors in scheme interpreter.
  - All code is in cs61a/lectures/vector.scm
  - Vector-map
  - Vector-cons

Vectors vs. Lists

operation	**lists**	**vectors**
n th element	`list-ref`, $\theta(n)$	`vector-ref`, $\theta(1)$
add new element	`cons`, $\theta(1)$	`vector-cons`, $\theta(n)$

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

Vector example

- Look at shuffle example in scheme interpreter.
  - all the code is in: cs61a/lectures/vector.scm
  - shuffle1
  - shuffle2
  - shuffle3