6.001 Tutorial 6 Notes

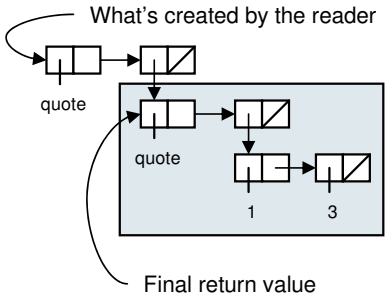
Gerald Dalley
14–15 Mar 2005

Announcements

- Feedback on feedback by next week.
- Spring break next week. Let me know if you'd like to meet with me for office hours.
- Project 3 due Friday, 1 April.
- Last week's answers: I changed my mind and made the ADTs use fully-tagged data structures with type checking.

Review

- **Quotation:** The `quote` special form just returns exactly the argument it was given – whatever the reader built. The sugared form of quote is an apostrophe, *e.g.* `'1` ⇔ `(quote 1)` ⇔ `1`.
  When evaluating: `''(1 3)`



- **Symbols:** True or false:
  - Symbols are strings? F, they are *interned* strings.
  - Symbols are pointers? F
  - All symbols with the same name are `eq?`? T
  - `quote` always creates a symbol? F, `'(1 2 3)` generates no symbols.
  - Comparing symbols is as slow as comparing strings F, symbols → constant time
  - To what does `(eq? 'a 'a)` evaluate? `#t`
- **Equality:**
  - Which is faster: `eq?` or `equal?` `eq? is faster`
  - Which is faster: `=` or `eq?`? `eq? is faster for large numbers`
  - What data types should typically be used with `=`? only numbers
  - What data types should typically be used with `eq?`? booleans and symbols
  - What data types should typically be used with `equal?`? just about anything

Pair/List Mutation

Up to this point, everthing we have done in Scheme has been *functional programming* – each procedure we write is a *function*; that is, a procedure that always returns the same value(s) for any set of inputs.

Mutation changes all that, specifically by giving us functions that can change things. Right now we have `set-car!` and `set-cdr!`. Tuesday we'll see `set!` as well. Soon we'll develop the environment model to better describe how mutation works.

`set-car!` and `set-cdr!`

- `(set-car! pair object)`:
  Stores `object` in the `car` field of `pair`. The value returned by `set-car!` is unspecified.
- `(set-cdr! pair object)`:
  Stores `object` in the `cdr` field of `pair`. The value returned by `set-cdr!` is unspecified.

Assume that each of the following statements are executed in order. Indicate the value of `lst`.

```
(define lst (list 1 2 3))
(set-car! lst 5)          (5 2 3)
(set-car! lst '(7 8))     ((7 8) 2 3)
(set-cdr! lst '(9 10))    ((7 8) 9 10)
(set-cdr! (car lst) 11)
((7 . 11) 9 10) in our implementation
```

Special note: technically speaking, you are not allowed to mutate constants, and technically things created by `quote` are constants. Both 6.001 Scheme and Dr. Scheme let you get away with it. Otherwise the last example would have generated an error.

Let's reimplement the built-in procedure `append!`.

```
;; Does destructive append -- changes the cdr
;; of the last pair of lst1 to point to lst2
(define (append! lst1 lst2)
  (cond ((null? lst1)
         (error "Cannot append! to nil"))
        ((null? (cdr lst1))
         (set-cdr! lst1 lst2))
        (else (append! (cdr lst1) lst2))))

; Simple test (uses assert-equal from
; tutorial05)
(define a (list 1 2 3))
(define b (list 4 5 6))
(append! a b)
(assert-equal a '(1 2 3 4 5 6))
```

alists

Recall that association lists, or alists, are a common data structure in Scheme. Each element of the list is a key-value pair.

`(assoc object alist)` is a built-in procedure that searches through `alist` looking for the given key `object`. If `object` is found, the key-value pair is returned. If not, `#f` (**not** the empty list) is returned. Indicate the results of each computation, assuming that they are evaluated in order.

```
(define e '((a 1) (b 2) (c 3)))
(assoc 'd e)   #f
(assoc 'a e)   (a 1)
(assoc 'c (cdr e))   (c 3)
(assoc 'a (car e))   error: (a 1) is not an alist
```

Other Data Types

Last week in lecture, we saw hash tables and vectors for the first time.

Vectors are like lists, except vectors:

- are fixed-length
- `vector-ref` is constant-time
- Uses `vector-set!` instead of `set-car!` and `set-cdr!` for mutation.

Hash tables are a type of optimized associative data structure. A hash function maps any (valid) key to a small integer (the size of the hash table). A mechanism is required for dealing with different keys with the same hash code. A simple solution is to store an alist of key-value pairs in each hash table entry.

Please refer to the lecture notes, recitation notes, *etc.* for more information.

`http://people.csail.mit.edu/people/bkph/courses/6001/2005-03-11.scm`

Class Database

In grading project 1, I annotated the submissions with special comments that indicated how they were to be assessed. I then wrote a set of scripts to parse those comments and determine the final grade for each person's project. Today we'll look at how to build some abstractions to allow us to merge scores accumulated during the semester and maintain a database of the current grades for each student. We'll assume that the parsing script generates code of the following form, which we will execute to obtain the current grades for everyone.

```
(define mit6001 (make-class "6.001"))
(add-student! mit6001 (make-student
  'hacker "Alyssa P." "Hacker"
  "hacker@mit.edu"))
(add-student! mit6001 (make-student
  'bitdiddle "Ben" "Bitdiddle"
  "bitdiddle@mit.edu"))
(add-student! mit6001 (make-student
  'plob "Ebenezer" "Plob" "mrplob@mit.edu"))

(add-points! mit6001 'hacker    93. 100)
(add-points! mit6001 'bitdiddle 82. 100)
(add-points! mit6001 'plob      76. 100)
(add-points! mit6001 'hacker    50. 50)
(add-points! mit6001 'bitdiddle 45. 50)
(add-points! mit6001 'plob      30. 50)

(table/for-each
 (get-grades mit6001)
 (lambda (uname grade)
   (display ";") (display uname) (display " ")
   (display grade) (newline)))
;hacker 0.9533333333333334
;bitdiddle 0.8466666666666667
;plob 0.7066666666666667
```

To run this code, we will create three ADTs: `table`, `student`, and `class`.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Table abstraction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define *table-tag* 'table)
;; Returns a newly allocated empty association table.
(define (make-table)
  (cons *table-tag* nil))
;; Returns #t if tbl is a table, otherwise returns #f.
(define (table? tbl)
  (and (pair? tbl)
       (eq? (car tbl) *table-tag*)))
;; Indicates whether there is an association of key to
;; some value in the table.
(define (table/has-key? tbl key)
  (assert-t (table? tbl))
  (not (eq? #f (assoc key (cdr tbl)))))
;; Returns the value associated with key in tbl. If there
;; is no association for key, #f is returned.  table/has-key?
;; can be used to differentiate between a value being #f and
;; having no association.
(define (table/get tbl key)
  (assert-t (table? tbl))
  (if (table/has-key? tbl key)
      (cdr (assoc key (cdr tbl)))
      #f))
;; Associates val with key in table and returns an
;; unspecified result.
(define (table/put! tbl key val)
  (assert-t (table? tbl))
  (if (table/has-key? tbl key)
      (set-cdr! (assoc key (cdr tbl)) val)
      (set-cdr! tbl (cons (cons key val)
                          (cdr tbl)))))
;; proc must be a procedure of two arguments. Invokes
;; proc once for each association in hash-table, passing
;; the association's key and value as arguments, in that
;; order. Returns an unspecified result. Proc must not
;; modify tbl.
(define (table/for-each tbl proc)
  (assert-t (table? tbl))
  (for-each (lambda (entry) (proc (car entry) (cdr entry)))
            (cdr tbl)))
;; Returns a newly allocated list of the keys in table.
;; The ordering of the keys is unspecified.
(define (table/key-list tbl)
  (assert-t (table? tbl))
  (fold-right cons nil
              (map car (cdr tbl))))
;; Returns a newly allocated list of the values in tbl.
;; Each element of the list corresponds to one of the
;; associations in tbl.
(define (table/value-list tbl)
  (assert-t (table? tbl))
  (fold-right cons nil
              (map cdr (cdr tbl))))
;; If-found must be a procedure of one argument, and
;; if-not-found must be a procedure of no arguments. If
;; tbl contains an association for key, if-found is
;; invoked on the value of the association. Otherwise,
;; if-not-found is invoked with no arguments. In either
;; case, the result of the invoked procedure is
;; returned as the result of table/lookup.
(define (table/lookup tbl key if-found if-not-found)
  (assert-t (table? tbl))
```

```
  (if (table/has-key? tbl key)
      (if-found (table/get tbl key))
      (if-not-found)))

(define tbl (make-table))
(assert-f (table/has-key? tbl 'a))
(assert-f (table/get tbl 'a))
(table/put! tbl 'a 1)
(assert-t (table/has-key? tbl 'a))
(assert-equal (table/get tbl 'a) 1)
(table/put! tbl 'b 2)
(assert-equal (table/get tbl 'a) 1)
(assert-equal (table/get tbl 'b) 2)
(table/put! tbl 'a 3)
(assert-equal (table/get tbl 'a) 3)
(assert-equal (table/get tbl 'b) 2)
(table/put! tbl 'b 4)
(assert-equal (table/get tbl 'a) 3)
(assert-equal (table/get tbl 'b) 4)
(assert-t (set-equal? (list-to-set (table/key-list tbl)) (list-to-set '(a b))))
(assert-t (set-equal? (list-to-set (table/value-list tbl)) (list-to-set '(3 4))))
(assert-equal (table/lookup tbl 'a square (lambda () 0)) 9)
(assert-equal (table/lookup tbl 'c square (lambda () 0)) 0)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Student abstraction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *student-tag* 'student)

;; Creates a new student record.  Students have the following
;; attributes:
;;    uname           : symbol
;;    given-name      : string
;;    family-name     : string
;;    email           : string
;;    points-earned   : number
;;    points-possible : number
;; All of these attributes should be stored in a table ADT.
(define (make-student uname given-name family-name email)
  (let ((student (cons *student-tag* (make-table))))
    (table/put! (cdr student) 'uname uname)
    (table/put! (cdr student) 'given-name given-name)
    (table/put! (cdr student) 'family-name family-name)
    (table/put! (cdr student) 'email email)
    (table/put! (cdr student) 'points-earned 0)
    (table/put! (cdr student) 'points-possible 0)
    student))

;; Determines whether obj is a student record.
(define (student? obj)
  (and (pair? obj)
       (eq? *student-tag* (first obj))
       (table? (cdr obj))))

;; Returns the username symbol of a student record
(define (student-uname student)
  (assert-t (student? student))
  (table/get (cdr student) 'uname))

;; Returns the given name(s) for the student
(define (student-given-name student)
  (assert-t (student? student))
  (table/get (cdr student) 'given-name))

;; Returns the family name for the student
(define (student-family-name student)
  (assert-t (student? student))
  (table/get (cdr student) 'family-name))

;; Returns the email address for the student
(define (student-email student)
  (assert-t (student? student))
  (table/get (cdr student) 'email))

;; Returns the total number of points earned so
;; far by this student
(define (student-points-earned student)
  (assert-t (student? student))
  (table/get (cdr student) 'points-earned))

;; Returns the total number of possible points
;; the student could have earned so far.
(define (student-possible-points student)
  (assert-t (student? student))
  (table/get (cdr student) 'points-possible))

;; Calculates the current grade of the student
```

```
;; as a number between 0 and 1.
(define (student-grade student)
  (assert-t (student? student))
  (/ (student-points-earned student)
     (student-possible-points student)))

;; Augments the current score of the student by
;; incrementing the points earned and total number
;; of possible points.
(define (student-add-points! student earned possible)
  (assert-t (student? student))
  ; Update points earned
  (table/put! (cdr student)
              'points-earned
              (+ earned
                 (table/get (cdr student) 'points-earned)))
  ; Update total points possible
  (table/put! (cdr student)
              'points-possible
              (+ possible
                 (table/get (cdr student) 'points-possible))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Class abstraction
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define *class-tag* 'class)

;; Creates a new class database.  A class has a name (string)
;; and a table of students.
(define (make-class name)
  (assert-t (string? name))
  (list *class-tag* name (make-table)))

;; Determines whether the passed-in object is a class database.
(define (class? obj)
  (and (list? obj) ; Could just use pair? if we wanted
       (eq? (first obj) *class-tag*) ; Important check
       (string? (second obj)) ; Pedantic check
       (table? (third obj)))) ; Pedantic check

;; Retrieves the name of the class
(define (class-name class)
  (assert-t (class? class))
  (second class))

;; Adds a new student record to the class
(define (add-student! class student)
  (assert-t (student? student))
  (assert-t (class? class))
  (table/put! (third class) (student-uname student) student))

;; Obtains a student record given the username (a symbol)
(define (get-student class uname)
  (assert-t (class? class))
  (assert-t (symbol? uname))
  (table/get (third class) uname))

;; Updates a student's score
(define (add-points! class uname earned possible)
  (assert-t (class? class))
  (assert-t (symbol? uname))
  (table/lookup
   (third class)
   uname
   (lambda (s) (student-add-points! s earned possible))
   (lambda () (error "student " uname " not found!"))))

;; Applies proc to each student in the class.  Proc is a
;; procedure that takes a student record as its only input.
(define (class/for-each class proc)
  (assert-t (class? class))
  (table/for-each (third class) proc))

;; Creates a new table indexed by student usernames containing
;; their current grades (as a number between 0 and 1).
(define (get-grades class)
  (assert-t (class? class))
  (let ((unames (table/key-list (third class)))
        (grades-table (make-table)))
    (for-each
     (lambda (uname)
       (table/put! grades-table
                   uname
                   (student-grade (get-student class uname))))
     unames)
```

```
    grades-table))

(define (print-grades class)
  (class/for-each
   class
   (lambda (uname s) (display uname) (display " ")
     (display (student-grade s)) (newline))))
```