6.001 Tutorial 7 Notes

Gerald Dalley
28–29 Mar 2005

1 Announcements and Recursive Feedback

- Problem set due Tuesday.
- Project 3 due Friday.
  - If you like using DrScheme, talk to me. There are some gotchas.
- If you'd like to bring food or beverages to share, I'm sure others will appreciate it ☺.
- No short break in the middle. Sorry.
- Most of the time, we will not review of previous week's material–it takes too long (remember tutorial 6).
- Tell me if you'd like to attend more than one tutorial (so I can organize it and avoid having 10 people show up at the same time).
- I've updated my website to include many of the suggestions made (`people.csail.mit.edu/~dalleyg/6.001/`).
- I'm still experimenting with incorporating some of the suggestions about enhancing the explanations, *etc.*
- Some data structures we will *not* discuss in detail today, but are nonetheless important.
  - Hash tables (lecture 11)
  - Stacks (lecture 12)
  - Queues (lecture 12)
  - Trees (lecture 13)

2 Data Structures

Over the last few weeks, we've started seeing many data structures that can be used for storing one-dimensional data (refer to the lecture notes for details on how to use the structures). It's important to keep track of what the differences are in terms of how we look up elements of each structure. Fill out Table 1, assuming the length of the list is N.

3 New Procedure: `set!`

Last time, we talked about `set-car!` and `set-cdr!` and how they introduced mutation. Today we'll talk about `set!`:

`(set! name exp)` `set!` evaluates the `exp`, and *replaces* the binding of `name` with the new value.

`(set-car! p exp)` `set-car!` evaluates `p` to get a pair, and the `exp`, and replaces the car of `p` with the value of `exp`.

`(set-cdr! p exp)` `set-cdr!` evaluates `p` to get a pair, and the `exp`, and replaces the cdr of `p` with the value of `exp`.

Note: when we implemented `append!` last week, we chose not to deal with `(append! '() some-list)`, but in lecture we saw an implementation that just returns the second argument if the first is null. For many mutator calls, the only safe way to make the call is to do something like:

```
(set! lst (append! lst new-stuff))
```

3.1 Examples

For the following examples, use `set!`.

```
;; This function takes one argument
;; and returns the value of the argument
;; the LAST time it was called:
;; (buffer 1) => #f
;; (buffer 'foo) => 1
;; (buffer '(1 2 3)) => foo
(define buffer
  (let ((last #f))
    (lambda (x)
      (let ((old last))
        (set! last x)
        old))))
```

```
;; This function takes one argument
;; and returns #t if it has ever seen
;; that argument before
(define seen?
  (let ((seen nil))
    (lambda (x)
```

Data Structure	Lookup Times			Lookup by	Restrictions
	Best-case	Worst-case	Typical		
list	1	N	$N/2$	integer (using `list-ref`)	none
alist	1	N	$N/2$	anything with equivalence	none
vector	1	1	1	integer	vectors are fixed-length (expensive to resize)
hash table	1	N	1	anything hashable	Needs a "good" hash function and a big enough vector to get typical behavior
queue	1	1	1	first in, first out (FIFO)	none
stack	1	1	1	last in, last out (LIFO)	none
trees (values in leaves)	1	N	$\log N$	anything comparable	none
trees (values everywhere)	1	N	$\log N$	anything comparable	none

Table 1: Data structures table (see Section 2). A few items have been filled out for you.

```
    (if (member x seen)
        #t
        (begin
          (set! seen (cons x seen))
          #f)))))
```

3.2 Memoization

A useful optimization that some languages perform is *memoization*. The idea of memoization is to figure out when a function returns a constant answer for a given set of arguments. If it does, whenever we call a function, we remember the arguments and the result. Later, we can check if we've seen these arguments before and return the result without doing the computation again. For expensive functions (lots of work), this can save a lot of time. The following procedure takes a single argument function and returns a memoized version.

```
;; Creates a new procedure that memoizes
;; calls to proc.
;; TYPE: (A->B) -> (A->B)
(define (memoize proc)
  (let ((vals '()))
    (lambda (arg)
      (let ((previous (assoc arg vals)))
        (if previous
            (cadr previous)
            (let ((temp (proc arg)))
              (set! vals
                (cons (list arg temp)
                      vals))
              temp))))))
```

A little hint – remember that `assoc` is a procedure that operates on association lists. It compares the `car` of each element of the list to the first argument and returns that element if it is. It has the following behavior:

```
(assoc 5 '((5 25) (6 36))) ; Value: (5 25)

(assoc 3 '((5 25) (6 36))) ; Value: #f

(assoc 3 '()) ; Value: #f
```

Now imagine we memoize a simple procedure:

```
(define fast-square
  (memoize
    (lambda (x)
      (display ";squaring ") (display x)
      (newline)
      (* x x))))

(fast-square 5)
(fast-square 5)
(fast-square 10)
(fast-square 5)
```

What gets printed if this is evaluated in 6.001 Scheme?

```
;squaring 5
;Value: 25
;Value: 25
;squaring 10
;Value: 100
;Value: 25
```

4 Project 3 Clarifications

- **Read:** Please read all of the comments I gave back from project 2. Many of the reasons points were taken off are easily fixable.
- **DrScheme:** As mentioned before, if you're using DrScheme for project 3, let me know so I can tell you how to make your code portable.
- **Exercise 5:** In exercise 5, we're asked to create a procedure named `make-web-index`. The difficulty here is that `make-web-index`'s job is not to make a web index. Its job is to provide an "access method" to the web that *uses* an index. This can be a bit confusing when we reach exercise 9 and we need to actually build indices, not access methods. I strongly recommend creating a procedure called something like `build-index` that just builds the index. Then have `make-web-index` use `build-index` to create the access method.
- **Exercise 8 (#1):** The old version of `generate.scm` has numbers as well as symbols as document contents. The problem here is that it's hinted that you use `symbol<?` for a comparison operator. Unfortunately, this operator only handles symbols, not symbols and numbers. A new version of `generate.scm` is being posted that does not include any numbers. Make sure you use it.
- **Exercise 8 (#2):** You'll likely need to extend the index abstraction in order to avoid getting points knocked off for abstraction violations when implementing `optimize-index`. Note that `Optimized-Index` and `Index` are two distinct ADTs and `Optimized-Index` has no more right to know about the internal structure of `Index` than any other piece of code that's not part of `Index`'s abstraction.
- **Exercise 9:** You're likely to find some very strange timing results in exercise 9 if you don't follow the suggestions here. I recommend reimplementing the `assv` procedure (it's very similar to `assoc`—read the Scheme documentation). The version of `assv` built into 6.001 Scheme and DrScheme is compiled and thus incredibly fast for small lists like the ones we're using even though it has a bad order-of-growth. Since we haven't shown you how to make your own optimized compiled procedures, the comparision between your binary search and the built-in `assv` is extremely unfair.
- **Documentation:** Please label each computer exercise with the text "Computer Exercise N" (with N being replaced with the appropriate exercise number). This makes your grader happy.
- **Load:** It gets a bit ugly to include all of the code from `generate.scm` and `search.scm` directly in your solution code. Assuming you make no modifications to either of these two files (you shouldn't anyway), you may use the `load` procedure to evaluate them. Here's an example of how to do this:

```
;;;;;
;;;;; Ben Bitdiddle
;;;;; 6.001 Project 3, Spring 2005
;;;;; TA: Gerald Dalley
;;;;;

; Load supplied project code
;(load "drscheme.scm")
(load "search.scm")
(load "generate.scm")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Extensions to the index ADT
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

...

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Computer Exercise 1
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

...
```