

# 6.001 Tutorial 4 – Solutions

David Ziegler

October 4, 2004

## 1 Quiz Stuff

- Quiz, Thursday, 7:30 – 9:30 pm
- Room 32-123 (where we have lecture)
- One page of notes
- Review session tonight and tomorrow (LAs), 7 pm, 34-302
- Office hours tomorrow (me), 3 – 4 pm, here

```
;; This procedure composes two
;; functions f and g, each of one
;; argument, and returns a procedure
;; of one arg that does (f (g x))
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

## 2 Project Comments

- Please, please, please indent your code.
- Please, please, please document your code.
- Please, please, please test your code and include the results.

```
;; This procedure applies f to each
;; element of the list, and returns a
;; new list made from those values
(define (map f lst)
  (if (null? lst)
      nil
      (cons (f (car lst))
            (map f (cdr lst)))))
```

## 3 Higher-Order Procedures

Higher-order procedures are procedures that either accept procedures as arguments or return procedures their value.

```
;; This procedure carries a function,
;; e.g. it takes a function of two
;; inputs and returns a function of
;; one input, that returns a function
;; of one input, that does the same
;; thing
;; ex: (+ 1 2) ==> 3
;; (((curry +) 1) 2) ==> 3
(define (curry f)
  (lambda (x)
    (lambda (y)
      (f x y))))
```

```
;; This procedure returns a new list
;; containing the elements in the
;; original for which pred is true
(define (filter pred lst)
  (cond ((null? lst) nil)
        ((pred (car lst)) (cons (car lst)
                                 (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))
```

```
;; This procedure combines all the
;; elements of lst using the binary
;; operation op, terminating with init
(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (fold-right op init (cdr lst)))))
```

## 4 Queues

A queue is a data structure that stores elements in order. Elements are enqueued onto the tail of the queue. Elements are dequeued from the head of the queue. Thus, the first element enqueued is also the first element dequeued (FIFO, first-in-first-out). The head operation is used to get the element at the head of the queue.

```
(head (enqueue 5 (empty-queue)))  
;Value: 5
```

```
(define q (enqueue 4 (enqueue 5 (enqueue 6 (empty-queue)))))
```

```
(head q)  
;Value: 6
```

```
(head (dequeue q))  
;Value: 5
```

Decide on an implementation for queues, then draw a box-and-pointer representation of the value of `q` as defined above.

```
;; This procedure returns an empty queue  
(define (empty-queue)  
  nil)
```

Time =  $\Theta(1)$ , space =  $\Theta(1)$ ,  $n =$

```
;; This procedure returns a new queue with the element added to the tail  
(define (enqueue x q)  
  (fold-right cons  
              (list x)  
              q))
```

Can you do this with `fold-right`?

Time =  $\Theta(n)$ , space =  $\Theta(n)$ ,  $n =$  length of `q`.

```
;; This procedure returns a new queue with the head removed  
(define (dequeue q)  
  (cdr q))
```

Time =  $\Theta(1)$ , space =  $\Theta(1)$ ,  $n =$

```
;; This procedure returns the value of the head of the queue  
(define (head q)  
  (car q))
```

Time =  $\Theta(1)$ , space =  $\Theta(1)$ ,  $n =$

## 5 Practice With HOPs

Suppose `lst` is bound to the list `(1 2 3 4 5 6 7)`. Using `map`, `filter`, and/or `fold-right`, write an expression involving `lst` that returns:

- `(1 4 9 16 25 36 49)`

```
(map square lst)
```

- `(1 3 5 7)`

```
(filter odd? lst)
```

- `((1 1) (2 2) (3 3) (4 4) (5 5) (6 6) (7 7))`

```
(map (lambda (x) (list x x)) lst)
```

- `((2) ((4) ((6) #f)))`

```
(fold-right list nil lst)
```

- The maximum element of `lst`: `7`

```
(fold-right max (car lst) (cdr lst))
```

- The last pair of `lst`: `(7)`

*Impossible!* `map`, `filter`, and `fold-right` only give you access to the *members* of the list, not the *backbone* – the cons cells which make up the list.