10/10 WELL DONE!!

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS6
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; $Date: 2007-03-21 12:17:47 -0400 (Wed, 21 Mar 2007) $ $Rev: 2305 $
;;;
;;; Collaborators: '(iyzhang amdragon)
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(load "load.scm")

;;; Some helper functions

(define (assert x . msg)
  (if (not x) (apply error msg)))

(define (assert-equal a b)
  (assert (equal? a b)
          "test failed"
          "Expected: "
          b
          " got: "
          a)
  #t)

;;;
;;; Problem 1
;;;
;;; The rats manage to find the food at D and E because they are
;;; reachable via the path
;;;   start -> b -> a -> e -> c -> d
;;;
;;; The depth-first rat initially finds the food at d after traversing
;;; a path of length 5 (start -> b -> a -> e -> c -> d), because this
;;; is the first path that the depth-first search finds. Later in the
;;; search, however, it finds the shorter path start -> b -> d.
;;;
```

3/3 ✓ ✓

4/4

WELL DONE

```
;;;
;;; Problem 2
;;;
;;; We define a new graph representation that supports lazy
;;; incremental construction of the graph.
;;;
;;; This provides nearly the same interface as the old graph API,
;;; except that now nodes are represented by some sort of state
;;; (instead of a name), and a function to generate a list of the next
;;; states from the current state. make-node takes the function and
;;; initial state as its arguments. node->out-edges lazily generates
;;; up the next set of nodes and edges to them; node->in-edges no
;;; longer works. make-graph is now supplied with a list of initial
;;; nodes rather than the full list of nodes in the system.
;;;
;;; Since the lazily constructed graph could be any arbitrary graph,
;;; not just a tree, we need a way of determining whether a newly
;;; constructed node is the same as some other node. We do this by
;;; maintaining a hash table as part of the graph that maps states
;;; that are equal? to nodes (which will be eq?). This means that the
;;; graph isn't restricted to be a tree, but does require keeping some
;;; state around.
;;;

(define (make-node fn state)
  (list 'node fn state #f))

(define (node? x)
  (and (pair? x) (eq? (car x) 'node)))

(define (node->fn x) (second x))
(define (set-node-fn! x fn) (set-car! (cdr x) fn))

(define (node->name x) (third x))
(define (set-node-name! x name) (set-car! (cddr x) name))

(define (node->graph x) (fourth x))
(define (set-node-graph! x graph) (set-car! (cdddr x) graph))

(define (node->out-edges node)
  (map (lambda (target)
         (%make-edge node target))
       (let ((next-states ((node->fn node) (node->name node))))
         (map (lambda (state)
                (or (hash-table/get (graph->hash (node->graph node))
                                    state #f)
                    (let ((new-node (make-node (node->fn node) state)))
                      (set-node-graph! new-node (node->graph node))
                      (hash-table/put! (graph->hash (node->graph node))
                                       state
                                       new-node)
                      new-node)))
              next-states))))

(define-record-type <graph>
    (%make-graph nodes)
    graph?
  (nodes graph->nodes)
  (hash graph->hash set-graph-hash!))

(define (make-graph . nodes)
  (let ((g (%make-graph nodes)))
    (set-graph-hash! g (make-equal-hash-table))
    (for-each (lambda (node)
```

Good

```
            (if (node? node)
                (set-node-graph! node g)
                (error "bad node -- make-graph" node)))
          nodes)
    g))


;; Do a depth-first search of a lazily constructed graph, and print
;; any accepting states found.
(define (dfs-print-accepting-nodes fn init-state accept?)
  (let* ((start-node (make-node fn init-state))
         (graph (make-graph start-node)))
    (graph:search start-node
                  (lambda (x) (accept? (node->name x)))
                  (lambda (found proceed)
                    (write-line
                     `(start -->
                             ,(node->name found)))
                    (proceed))
                  depth-first-add-edges!
                  (lambda (x) (lambda (y) #f)))))

;; Boring test: build a graph of the numbers 1 through max and search
;; it for some n
(define (search-for-number n max)
  (dfs-print-accepting-nodes
   (lambda (state) (if (< state max)
                       (list (+ 1 state))
                       '()))
   1
   (lambda (x) (eqv? x n))))

#|
(search-for-number 10 100)
(start --> 10)
;Value: #f
|#


;; Search for Ramanujan numbers
;; First two are
;;  1729 = 12^3 + 1^3 = 10^3 + 9^3
;;  4104 = 16^3 + 2^3 = 15^3 + 9^3

;; State is represented as (a b c d), where we're searching for
;; a^3 + b^3 = c^3 + d^3; we require that b < a, and d < c < a.
(define (find-ramanujan-number max)
  (bfs-print-accepting-nodes
   (lambda (state)
     (let ((a (first state))
           (b (second state))
           (c (third state))
           (d (fourth state)))
       (append
        (if (< a max) (list (list (+ a 1) b c d)) '())
        (if (< b (- a 1)) (list (list a (+ b 1) c d)) '())
        (if (< c (- a 1)) (list (list a b (+ c 1) d)) '())
        (if (< d (- c 1)) (list (list a b c (+ d 1))) '()))))
   '(3 1 2 1)
   (lambda (state)
     (let ((a (first state))
           (b (second state))
           (c (third state))
           (d (fourth state)))
       (eqv? (+ (expt a 3) (expt b 3))
             (+ (expt c 3) (expt d 3)))))))

#|
(find-ramanujan-number 17)
(start --> (16 2 15 9))
(start --> (12 1 10 9))
|#
```

```
;;;
;;; Problem 3
;;;
;;; To implement some kind of prioritized search (e.g. best-first),
;;; the most elegant option would be to replace the queue of edges to
;;; search with a priority queue, where the item with highest priority
;;; is retrieved first. Then we'd create a function
;;; best-first-add-edges!, along the lines of depth-first-add-edges!
;;; that evaluates the edges and adds them to the priority queue.
;;;
;;; A simple (but inefficient) way to implement such a priority queue
;;; would be to represent it as a list kept in sorted order; the
;;; function for adding new edges to the list would have to cdr down
;;; the list, inserting the new edge at the appropriate spot to keep
;;; it in sorted order. A more efficient implementation would use
;;; something like a binary heap or a Fibonacci heap. Note that doing
;;; so would require a small modification to the search code, calling
;;; the appropriate function for the new data structure instead of
;;; queue:first.
;;;
```

3/3 WELL ARGUED

WOO HOO!

I LOVE FIBONACCI HEAPS! ☺

-Z