10/10 Good! (but see 7.6.)

ps7.scm


```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS7
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; $Date: 2007-04-04 12:44:58 -0400 (Wed, 04 Apr 2007) $ $Rev: 2316 $
;;;
;;; 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)
  'assertion-ok)

;;;
;;; Problem 1
;;;

;; Take a list of possible last names for Moore's daughter; this lets
;; us either impose the constraint that Mary Ann's last name is Moore
;; or not.
(define (yacht moore-daughter)
  (let ((downing-daughter (amb 'mary-ann 'gabrielle 'lorna 'rosalind
                               'melissa))
        (hall-daughter (amb 'mary-ann 'gabrielle 'lorna 'rosalind
                            'melissa))
        (hood-daughter 'melissa)
        (parker-daughter (amb 'mary-ann 'gabrielle 'lorna 'rosalind
                              'melissa))
        (moore-yacht 'lorna)
        (downing-yacht 'melissa)
        (hall-yacht 'rosalind)
        (hood-yacht 'gabrielle)
        (parker-yacht (amb 'mary-ann 'gabrielle 'lorna 'rosalind
                           'melissa)))
    (require (distinct? (list moore-daughter downing-daughter
                              hall-daughter hood-daughter
                              parker-daughter)))
    (require (distinct? (list moore-yacht downing-yacht
                              hall-yacht hood-yacht
                              parker-yacht)))

    (for-each (lambda (x) (require (distinct? x)))
              (list (list moore-daughter moore-yacht)
                    (list downing-daughter downing-yacht)
                    (list hall-daughter hall-yacht)
                    (list hood-daughter hood-yacht)
                    (list parker-daughter parker-yacht)))

    (for-each (lambda (x) (if (eq? (first x) 'gabrielle)
                              (require (eq? (second x) parker-daughter))))
              (list (list moore-daughter moore-yacht)
                    (list downing-daughter downing-yacht)
                    (list hall-daughter hall-yacht)
                    (list hood-daughter hood-yacht)
                    (list parker-daughter parker-yacht)))

    `((moore-daughter ,moore-daughter)
      (downing-daughter ,downing-daughter)
      (hall-daughter ,hall-daughter)
      (hood-daughter ,hood-daughter)
      (parker-daughter ,parker-daughter)
      (moore-yacht ,moore-yacht)
      (downing-yacht ,downing-yacht)
      (hall-yacht ,hall-yacht)
      (hood-yacht ,hood-yacht)
      (parker-yacht ,parker-yacht))))

;;; Lorna's father is Colonel Downing:

#|

(with-depth-first-schedule (lambda () (yacht 'mary-ann)))

;Value 1: ((moore-daughter mary-ann) (downing-daughter lorna) (hall-daughter gabrielle
) (hood-daughter melissa) (parker-daughter rosalind) (moore-yacht lorna) (downing-yach
t melissa) (hall-yacht rosalind) (hood-yacht gabrielle) (parker-yacht mary-ann))

|#

;;; Without assuming Mary Ann's last name is Moore, there are two
;;; possible solutions:

#|
(init-amb)
;Value: done

(yacht (amb 'mary-ann 'gabrielle 'lorna 'rosalind
            'melissa))
;Value 2: ((moore-daughter mary-ann) (downing-daughter lorna) (hall-daughter gabrielle
) (hood-daughter melissa) (parker-daughter rosalind) (moore-yacht lorna) (downing-yach
t melissa) (hall-yacht rosalind) (hood-yacht gabrielle) (parker-yacht mary-ann))

(amb)
;Value 3: ((moore-daughter gabrielle) (downing-daughter rosalind) (hall-daughter mary-
ann) (hood-daughter melissa) (parker-daughter lorna) (moore-yacht lorna) (downing-yach
t melissa) (hall-yacht rosalind) (hood-yacht gabrielle) (parker-yacht mary-ann))

(amb)
;Value: #f
|#
```

```
;;;
;;; Problem 2
;;;

;; Just the place for a Snark! I have said it thrice:
;; What I tell you three times is true.

(define (snark-hunt tree)
  (call-with-current-continuation
   (lambda (exit)
     (let walk ((tree tree))
       (if (not (pair? tree))
           (if (eq? tree 'snark) (exit #t))
           (for-each walk tree)))
     #f)))

;; Test cases
(assert-equal (snark-hunt '(bellman baker boots)) #f)
(assert-equal (snark-hunt '(bellman baker boots snark)) #t)
(assert-equal (snark-hunt '(((bellman baker boots)
                             barrister (billiard-marker banker))
                            bonnet-maker (((boojum)
                                           broker) (beaver butcher)))) #f)
(assert-equal (snark-hunt '(((bellman baker boots)
                             barrister (billiard-marker banker))
                            bonnet-maker (((snark)
                                           broker) (beaver butcher)))) #t)
(assert-equal
 (snark-hunt '(((a b c) d (e f)) g (((snark . "oops") h) (i . j))))
 #t)

;;; We'll verify that the exit is in fact immediate by having a
;;; variable that's set to #f at the beginning and only changed to #t
;;; when a snark is found. In order to ensure that the exit doesn't
;;; return through multiple return levels, we'll assert at the end of
;;; each call that the value of this variable is still #f; this will
;;; produce an error if the return path is not direct.

(define (snark-hunt/instrumented tree)
  (let ((snark-found? #f))
    (call-with-current-continuation
     (lambda (exit)
       (let walk ((tree tree))
         (if (not (pair? tree))
             (if (eq? tree 'snark)
                 (begin
                   (set! snark-found? #t)
                   (exit #t)))
             (for-each walk tree))
         (assert-equal snark-found? #f))
       #f))))


(assert-equal (snark-hunt/instrumented '(bellman baker boots)) #f)
(assert-equal (snark-hunt/instrumented '(bellman baker boots
                                                 snark)) #t)
(assert-equal (snark-hunt/instrumented
               '(((bellman baker boots)
                  barrister (billiard-marker banker))
                 bonnet-maker (((boojum)
                                broker) (beaver butcher)))) #f)
(assert-equal (snark-hunt/instrumented
               '(((bellman baker boots)
                  barrister (billiard-marker banker))
                 bonnet-maker (((snark)
                                broker) (beaver butcher)))) #t)
(assert-equal
 (snark-hunt/instrumented
  '(((a b c) d (e f)) g (((snark . "oops") h) (i . j))))
 #t)
```

```
;;;
;;; Problem 3
;;;
;;; The depth-first search does less work than the breadth-first
;;; search because it explores the tree depth-first and can stop when
;;; it finds an answer. For any particular i and j, it will test all
;;; values of k before proceeding, which is faster than the
;;; breadth-first search, which generates a number of other possible
;;; combinations of i and j in the meantime.
;;;
;;; The depth-first search will not terminate on an infinite set. It
;;; will keep trying new values for k -- of which there are an
;;; infinite number, of course --- and will never try incrementing i
;;; or j.
```

```
;;;
;;; Problem 4
;;;
;;; a. append is a O(n) operation in both space and time: it requires
;;; cdring down the entirity of the first input list, consing elements
;;; onto each other. This could mean that each call to amb requires
;;; time and space proportional to the length of the schedule.
;;;
;;; b. append! is more efficient in space since it can be implemented
;;; by a set-cdr! of the last pair of one list to the other list. No
;;; conses are required. However, it still requires cdring down one of
;;; the lists to find the last pair. The tconc queue avoids this
;;; problem, and can perform append! operations in constant space and
;;; time.
;;;
;;; c.

;; Internal representation
(define-record-type :queue
  (queue head tail)
  queue?
  (head queue-head set-queue-head!)
  (tail queue-tail set-queue-tail!))

;; Public interface
(define (make-queue)
  (queue '() '()))

(define (append-to-queue! q l)
  (if (not (null? l))
      (begin
        (append! (queue-tail q) l)
        (set-queue-tail! q (last-pair l))
        (if (null? (queue-head q))
            (set-queue-head! q l)))))

(define (add-to-front-of-queue! q l)
  (if (not (null? l))
      (begin
        (append! l (queue-head q))
        (set-queue-head! q l))))

(define (pop-from-queue! q)
  (if (null? (queue-head q))
      #f
      (let* ((old-head (queue-head q))
             (data (car old-head))
             (new-head (cdr old-head)))
        (set-queue-head! q new-head)
        (if (null? new-head)
            (set-queue-tail! q '()))
        data)))

(define (queue-empty? q)
  (null? (queue-head q)))

(define (queue-data q)
  (queue-head q))

;; Test cases
(let ((q (make-queue)))
  (append-to-queue! q '(1 2 3 4 5))
  (assert-equal (queue-data q) '(1 2 3 4 5))
  (append-to-queue! q '(6 7))
```

```
  (assert-equal (pop-from-queue! q) 1)
  (assert-equal (queue-data q) '(2 3 4 5 6 7))
  (assert-equal (pop-from-queue! q) 2)
  (assert-equal (queue-data q) '(3 4 5 6 7))
  (add-to-front-of-queue! q '(8 9))
  (assert-equal (queue-data q) '(8 9 3 4 5 6 7))
  (assert-equal (pop-from-queue! q) 8)
  (assert-equal (queue-data q) '(9 3 4 5 6 7)))

;;;
;;; New search schedule implementation
;;;

;;; Representation of the search schedule

(define *search-schedule*)

(define (empty-search-schedule) (make-queue))

(define (yield)
  (if (not (queue-empty? *search-schedule*))
      ((pop-from-queue! *search-schedule*))
      (*top-level* #f)))

(define (force-next thunk)
  (add-to-front-of-queue! *search-schedule* (list thunk)))

;;; Alternative search strategies

(define (add-to-depth-first-search-schedule alternatives)
        (add-to-front-of-queue! *search-schedule* alternatives))

(define (add-to-breadth-first-search-schedule alternatives)
        (append-to-queue! *search-schedule* alternatives))


;;; Simple tests

#|
(with-depth-first-schedule elementary-backtrack-test)
(1)
(1 a)
(1 a #t)
(1 a #f)
(1 b)
(1 b #t)
(1 b #f)
(2)
(2 a)
(2 a #t)
(2 a #f)
(2 b)
(2 b #t)
(2 b #f)
(3)
(3 a)
(3 a #t)
(3 a #f)
(3 b)
(3 b #t)
(3 b #f)
;Value: #f

(with-breadth-first-schedule elementary-backtrack-test)
(1)
(2)
(3)
(1 a)
(1 b)
(2 a)
(2 b)
(3 a)
(3 b)
(1 a #t)
(1 a #f)
(1 b #t)
(1 b #f)
(2 a #t)
(2 a #f)
(2 b #t)
(2 b #f)
(3 a #t)
(3 a #f)
(3 b #t)
(3 b #f)
;Value: #f
|#
```

```
;;;
;;; Problem 5
;;;
;;; a. I disagree that random ordering of amb arguments is useful. In
;;; general, unless some branch of the tree has infinite depth,
;;; randomized amb is semantically identical to regular amb in that
;;; they both generate the same set of possibilities (albeit perhaps
;;; in a different order). One can easily construct a case in which
;;; randomized amb terminates under depth-first search but
;;; left-to-right amb doesn't, e.g.:

(define (death-amb)
  (case (amb 1 2 3)
    ((1 3) (death-amb))
    ((2) 42)))

;;; death-amb doesn't terminate under depth-first left-to-right or
;;; right-to-left search, but terminates with high probability in
;;; randomized search. But this isn't particularly useful, since
;;; it will also terminate under breadth-first search. Moreover, only
;;; a tree with infinite branching factor will fail to terminate under
;;; breadth-first search, and such a thing is impossible to construct
;;; with our amb form, which can (of course) only take a finite number
;;; of arguments.
;;;
;;; Given that the same set of possibilities are generated, the only
;;; difference is the ordering. We might want to use randomized amb to
;;; generate elements from a sample space, or in some randomized
;;; algorithm. But randomized amb is not really the correct solution
;;; here either. For any non-trivial use of randomization, we need
;;; more control over the distribution; randomized amb only gives us a
;;; uniform random choice of the arguments. Many functions, like
;;; an-integer-from, use a recursive construction; since the
;;; randomized amb will choose randomly at each step, it gives 1 with
;;; probability 1/2, 2 with probability 1/4, and so on, which is
;;; almost certainly not what we want.
;;;
;;; On the other hand, randomization might be useful to catch
;;; programmers who make assumptions about the order of amb
;;; evaluation.

;;; b.

(define (identity x) x)

(define (list-remove l ind)
  (if (= ind 0)
      (cdr l)
      (cons (car l) (list-remove (cdr l) (- ind 1)))))

(define (shuffle x)
  (if (null? x) '()
      (let ((ind (random (length x))))
        (cons (list-ref x ind)
              (shuffle (list-remove x ind))))))

(define (with-left-to-right-alternation thunk)
  (lambda ()
    (fluid-let ((transform-alternatives identity))
      (thunk))))

(define (with-right-to-left-alternation thunk)
  (lambda ()
    (fluid-let ((transform-alternatives reverse))
      (thunk))))

(define (with-random-alternation thunk)
  (lambda ()
    (fluid-let ((transform-alternatives shuffle))
      (thunk))))

(define (add-to-depth-first-search-schedule alternatives)
        (add-to-front-of-queue! *search-schedule*
                                (transform-alternatives alternatives)))

(define (add-to-breadth-first-search-schedule alternatives)
        (append-to-queue! *search-schedule*
                          (transform-alternatives alternatives)))

(define (transform-alternatives alternatives)
  (identity alternatives))              ; default left to right


;; Test cases

#|
(with-depth-first-schedule
 (with-left-to-right-alternation elementary-backtrack-test))
(1)
(1 a)
(1 a #t)
(1 a #f)
(1 b)
(1 b #t)
(1 b #f)
(2)
(2 a)
(2 a #t)
(2 a #f)
(2 b)
(2 b #t)
(2 b #f)
(3)
(3 a)
(3 a #t)
(3 a #f)
(3 b)
(3 b #t)
(3 b #f)
;Value: #f

(with-depth-first-schedule
 (with-right-to-left-alternation elementary-backtrack-test))
(3)
(3 b)
(3 b #f)
(3 b #t)
(3 a)
(3 a #f)
(3 a #t)
(2)
(2 b)
(2 b #f)
(2 b #t)
(2 a)
(2 a #f)
(2 a #t)
(1)
(1 b)
```

```
(1 b #f)
(1 b #t)
(1 a)
(1 a #f)
(1 a #t)
;Value: #f

(with-depth-first-schedule
 (with-random-alternation elementary-backtrack-test))
(2)
(2 a)
(2 a #t)
(2 a #f)
(2 b)
(2 b #t)
(2 b #f)
(1)
(1 b)
(1 b #t)
(1 b #f)
(1 a)
(1 a #f)
(1 a #t)
(3)
(3 a)
(3 a #f)
(3 a #t)
(3 b)
(3 b #t)
(3 b #f)
;Value: #f
|#
```

```
;;;
;;; Problem 6
;;;
;;; This trace is precisely the same as that of the breadth-first
;;; elementary backtrack test, except that after proceeding to the
;;; next level of the tree, the previous variable retains its final
;;; value (e.g. x = 3). This occurs because set! is used; the
;;; assignment does not get undone on backtracking.
;;;
```

ACTUALLY, THE CORE ISSUE IS

DATAFLOW -v- CONTROL FLOW

amb-set! HAS THE SAME BEHAVIOR