```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; 6.891 PS8
;;;
;;; Dan R. K. Ports <drkp@mit.edu>
;;;
;;; $Date: 2007-04-18 12:44:36 -0400 (Wed, 18 Apr 2007) $ $Rev: 2324 $
;;;
;;; 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
;;;
;;; The worst-case runtime of this procedure has order of the sum of
;;; the depths of the leaf nodes in the tree. Since append takes time
;;; proportional to the number of elements in its first argument, the
;;; runtime of this procedure is proportional to the sum over all
;;; nodes of the length of the fringe rooted at that node. Since each
;;; leaf node appears in the fringe of each subtree containing it, and
;;; one such subtree exists for each of its ancestors, the total
;;; runtime is equivalent to the sum of the depths of the leaf nodes
;;; (modulo constant factors, of course).
;;;
;;; Given that the number of leaves is the fringe and their depths are
;;; bounded by the height of the tree, this runtime is
;;; O(size of fringe * height of tree).
;;;
;;; The time complexity is O(size of fringe + height of tree). The only
;;; data kept in the heap is the portion of the fringe that has been
;;; generated. However, we need also consider the stack; the recursion
;;; depth can be the height of the tree.

;;;
;;; Problem 2
;;;
;;; a) Using stream-append without thunkification of the second
;;; argument works correctly in the sense that it returns the same
;;; result, but would not be lazy in any useful sense:
;;; (lazy-fringe (cdr subtree)) is immediately evaluated at the time
;;; of the stream-append, so the entire fringe is generated at once.
;;;
;;; b) We'll start with the walk-based definition of fringe. To make
;;; the output a stream, we'll need to replace the cons in the else
;;; clause with cons-stream. But this encounters the same problem
;;; discussed in part (a). To avoid this, we need to delay evaluation
;;; of the cdr walk call, which we do with delay/force (though
;;; wrapping a lambda around it would work too).

(define (lazy-fringe subtree)
  (define (walk subtree ans)
    (cond ((pair? subtree)
           (walk (car subtree)
                 (delay (walk (cdr subtree) ans))))
          ((null? subtree)
           (force ans))
          (else
           (cons-stream subtree (force ans)))))
  (walk subtree (delay '())))

(assert-equal
 (stream->list (lazy-fringe '((a b) c ((d)) e (f ((g h))))))
 '(a b c d e f g h))
(assert-equal
 (stream->list (lazy-fringe '(a b c ((d) () e) (f (g (h))))))
 '(a b c d e f g h))
(assert-equal
 (stream->list (lazy-fringe '(a b c ((d) () e) (g (f (h))))))
 '(a b c d e g f h))
```

```
;;;
;;; Problem 3
;;;
;;; This isn't lambda calculus: Scheme uses applicative-order
;;; evaluation and provides mutation, so the eta rule doesn't apply.
;;;
;;; Wrapping a lambda around resume-thunk delays the lookup of the
;;; name resume-thunk. Since this is mutated (by the set! line), we
;;; need to re-perform this name lookup each time the thunk is
;;; called. Otherwise, repeatedly evaluating it would always give the
;;; same result.
;;;
```

```
;;;
;;; Problem 4
;;;
;;; If we instrument acs-coroutine-same-fringe? to print the values of
;;; x1 and x2 on each iteration of the loop, we see:

#|
(acs-coroutine-same-fringe?
 '((a b) c ((d)) e (f ((g h))))
 '(a b c ((d) () e) (f (g (h)))))
(a a)
(b b)
(c c)
(d d)
(e e)
(f f)
(g g)
(h h)
((*done*) a)
;Value: #f
|#

;;; The last execution of f2 returns the first value. This occurs
;;; because, if *done* is returned directly rather than calling the
;;; return continuation, it returns from the original continuation of
;;; the coroutine. That is, we return to the state as it existed at
;;; the first call of the loop. Then the next call to the other
;;; coroutine will execute with its initial state, causing it to
;;; return the same value it did on its first call.
```

```
;;;
;;; Problem 5
;;;

;; OK, let's give this macrology thing a try...

;; Run a thunk while holding a lock (acquire it before and release it
;; after.)
(define-syntax with-lock
  (sc-macro-transformer
   (lambda (form uenv)
     `(with-lock-thunkified
       ,(close-syntax (second form) uenv)
       (lambda ()
         ,(close-syntax (third form) uenv))))))

(define (with-lock-thunkified lock thunk)
  (conspire:acquire-lock lock)
  (let ((result (thunk)))
    (conspire:unlock lock)
    result))

(define-record-type pipe
  (%pipe reader-proc writer-proc)
  pipe?
  (reader-proc pipe-reader set-pipe-reader!)
  (writer-proc pipe-writer set-pipe-writer!))

(define (make-pipe)
  (let ((queue (queue:make))
        (lock (conspire:make-lock)))
    (letrec ((reader-proc
              (lambda ()
                (if (queue:empty? queue)
                    (begin
                      (conspire:thread-yield)
                      (reader-proc))
                    (with-lock lock
                               (queue:get-first queue)))))
             (writer-proc
              (lambda (item)
                (with-lock lock
                           (queue:add-to-end! queue item)))))
      (%pipe reader-proc writer-proc))))

;; Test fringe

(assert-equal
 (with-time-sharing-conspiracy
  (lambda ()
    (piped-same-fringe?
     '((a b) c ((d)) e (f ((g h))))
     '(a b c ((d) () e) (f (g (h)))))
   ))
 #t)

(assert-equal
 (with-time-sharing-conspiracy
  (lambda ()
    (piped-same-fringe?
     '((a b) c ((d)) e (f ((g h))))
     '(a b c ((d) () e) (g (f (h)))))
   ))
 #f)

(assert-equal
 (with-time-sharing-conspiracy
  (lambda ()
    (piped-same-fringe?
     '((a b) c ((d)) e (f ((g h))))
     '(a b c ((d) () e) (g (f))))
   ))
 #f)
```

```
;;;
;;; Problem 6
;;;

(define (make-threaded-filter proc)
  (let ((pipe (make-pipe)))
    (conspire:make-thread conspire:runnable
                          (lambda ()
                            (proc (pipe-writer pipe))))
    (pipe-reader pipe)))

;;  Test cases

(assert-equal
 (with-time-sharing-conspiracy
  (lambda ()
    (tf-piped-same-fringe?
     '((a b) c ((d)) e (f ((g h))))
     '(a b c ((d) () e) (f (g (h)))))
   ))
 #t)

(assert-equal
 (with-time-sharing-conspiracy
  (lambda ()
    (tf-piped-same-fringe?
     '((a b) c ((d)) e (f ((g h))))
     '(a b c ((d) () e) (g (f (h)))))
    ))
 #f)

(assert-equal
 (with-time-sharing-conspiracy
  (lambda ()
    (tf-piped-same-fringe?
     '((a b) c ((d)) e (f ((g h))))
     '(a b c ((d) () e) (g (f ))))
    ))
 #f)
```