MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945 Spring 2013
Problem Set 8

Issued: Wed. 3 Apr. 2013   Due: Wed. 17 Apr. 2013

Note: This problem set is not due until 17 April. You now have a term
      project to work on, so we will lighten up on the problem sets.

Readings:

Online MIT/GNU Scheme Documentation,

```
   Section 10.6: Streams         - cons-stream, etc.
   Section  2.3: Dynamic Binding - fluid-let
   Section 12.4: Continuations   - call-with-current-continuation &
                                              within-continuation
```

Here is a nice paper about continuations and threads:
    http://repository.readscheme.org/ftp/papers/sw2003/Threads.pdf

In fact, there is an entire bibliography of stuff about this on:
    http://library.readscheme.org/page6.html

The MIT/GNU Scheme reference manual is here:
    http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/

Code: load.scm, conspire.scm, try-two-ways.scm (attached)
       same-fringe.scm (not attached, since it is all shown here in line)

On The Fringes of Fun with Control Structures

Let’s look at a variety of ways to solve a famous problem, the classic "same fringe" problem.

The "fringe" of a tree is defined to be the ordered list of terminal leaves of the tree encountered when the tree is traversed in some standard order, say depth-first left-to-right. We can easily compute the fringe of a tree represented as a list structure. In the programs that follow we add an explicit test to exclude the empty list from the answer:

```
(define (fringe subtree)
  (cond ((pair? subtree)
         (append (fringe (car subtree))
                 (fringe (cdr subtree))))
        ((null? subtree) '())
        (else (list subtree))))
```

Where append is usually defined as:

```
(define (append l1 l2)
  (if (pair? l1)
      (cons (car l1) (append (cdr l1) l2))
      l2))
```

```
So the fringe of a typical tree is:

#|
     (fringe '((a b) c ((d)) e (f ((g h)))))
     ;Value: (a b c d e f g h)

     (fringe '(a b c ((d) () e) (f (g (h)))))
     ;Value: (a b c d e f g h)
|#

That was a horribly inefficient computation, because append keeps
copying parts of the fringe over and over.

-------------
Problem 8.1:

What is the worst-case algorithmic complexity of this procedure in
both time and space?  This is a bit tricky because your answer depends
on how you decide to measure the input argument.  Is it the length of
the fringe?  The number of nodes?  Is the depth of the tree relevant?
So make sure you explain your answer clearly.

For example, O(N), O(N lg N), O(N^2), etc. might all be correct
depending on how you define N.  Please be specific.
-------------


Here is a nicer procedure that computes the fringe, without any
nasty re-copying.

     (define (fringe subtree)
       (define (walk subtree ans)
         (cond ((pair? subtree)
                (walk (car subtree)
                      (walk (cdr subtree) ans)))
               ((null? subtree) ans)
               (else (cons subtree ans))))
       (walk subtree '()))

So the "same fringe" problem appears really simple:

     (define (same-fringe? tree1 tree2)
       (equal? (fringe tree1) (fringe tree2)))

Indeed, this works:

     #|
     (same-fringe? '((a b) c ((d)) e (f ((g h))))
                   '(a b c ((d) () e) (f (g (h)))))
     ;Value: #t

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

Unfortunately, this requires computing the entire fringe of each tree
before comparing the fringes.  Suppose that the trees were very big,
but that they were likely to differ early in the fringe.  This would
be a terrible strategy.  We would rather have a way of generating the
next element of the fringe of each tree only as needed to compare them.
```

One way to do this is with "lazy evaluation". This method requires
examining only as much of the input trees as is necessary to decide
when two fringes are not the same:

```
     (define (lazy-fringe subtree)
       (cond ((pair? subtree)
              (stream-append-deferred (lazy-fringe (car subtree))
                 (lambda () (lazy-fringe (cdr subtree)))))
             ((null? subtree) the-empty-stream)
             (else (stream subtree))))

     (define (lazy-same-fringe? tree1 tree2)
       (let lp ((f1 (lazy-fringe tree1))
                (f2 (lazy-fringe tree2)))
         (cond ((and (stream-null? f1) (stream-null? f2)) #t)
               ((or  (stream-null? f1) (stream-null? f2)) #f)
               ((eq? (stream-car   f1) (stream-car   f2))
                (lp  (stream-cdr   f1) (stream-cdr   f2)))
               (else #f))))

     (define (stream-append-deferred stream1 stream2-thunk)
       (if (stream-pair? stream1)
           (cons-stream (stream-car stream1)
                        (stream-append-deferred (stream-cdr stream1)
                                                stream2-thunk))
           (stream2-thunk)))

     (define the-empty-stream (stream))

     #|
     (lazy-same-fringe?
      '((a b) c ((d)) e (f ((g h))))
      '(a b c ((d) () e) (f (g (h)))))
     ;Value: #t

     (lazy-same-fringe?
      '((a b) c ((d)) e (f ((g h))))
      '(a b c ((d) () e) (g (f (h)))))
     ;Value: #f
     |#
```

Problem 8.2:

This implementation of fringe has the same problem of copying the
stream that our original fringe program had copying the list.

A. What would have gone wrong had we not thunkified the second argument
   to be appended and instead just used the stream-append procedure:

```
     (define (stream-append stream1 stream2)
       (if (stream-pair? stream1)
           (cons-stream (stream-car stream1)
                        (stream-append (stream-cdr stream1) stream2))
           stream2))
```

B. Redefine lazy-fringe to be a stream-based program that eliminates this
   re-copying while avoiding unnecessary subtree fringe generation, using
   a technique similar to the "nicer" (walk-based) fringe above.
   [Hint: Consider the cons-stream special form with deferred cdr walk.]

An alternative incremental idea is to make coroutines that generate
the fringes, using an explicit continuation argument and local state.

Notice that in the following code we invent a special object *done*.
Because it is a newly consed list, this object is eq? only to itself,
so it cannot be confused with any other object. This is a very common
device for making unique objects.

```
     (define *done* (list '*done*))

     (define (coroutine-fringe-generator tree)
       (define (resume-thunk)
         (walk tree (lambda () *done*)))
       (define (walk subtree continue)
         (cond ((null? subtree)
                (continue))
               ((pair? subtree)
                (walk (car subtree)
                      (lambda ()
                        (walk (cdr subtree)
                              continue))))
               (else
                (set! resume-thunk continue)
                subtree)))
       (lambda () (resume-thunk)))

     (define (coroutine-same-fringe? tree1 tree2)
       (let ((f1 (coroutine-fringe-generator tree1))
             (f2 (coroutine-fringe-generator tree2)))
         (let lp ((x1 (f1)) (x2 (f2)))
           (cond ((and (eq? x1 *done*) (eq? x2 *done*)) #t)
                 ((or  (eq? x1 *done*) (eq? x2 *done*)) #f)
                 ((eq? x1 x2) (lp (f1) (f2)))
                 (else #f)))))
```

Also notice the peculiar SET! assignment in this code. This makes it
possible for the procedures f1 and f2 (two distinct results of calling
the fringe generator) to maintain independent resume continuations
each time they are re-invoked to proceed generating their fringes.
This assignment is what gives each new fringe generator its own
dynamic local state.

```
     #|
     (coroutine-same-fringe?
      '((a b) c ((d)) e (f ((g h))))
      '(a b c ((d) () e) (f (g (h)))))
     ;Value: #t

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

Problem 8.3:

Why is it necessary to use the expression "(lambda () (resume-thunk))"
rather than just "resume-thunk" as the returned value of the fringe
generator? Aren't they the same, by the eta rule of lambda calculus?

We can abstract this control structure, using continuations. Now things get very complicated. Here, a procedure that is to be used as a coroutine takes an argument: return. Its value is a thunk that can be called to start the coroutine computing.

When the execution of the coroutine thunk calls the return procedure that was passed to its creator, it saves its state as a new thunk that invokes the continuation of the return. It then invokes a procedure with a value that the caller of the thunk will see as the value of the thunk.

```
     (define (make-coroutine his-job) ;; his-job ::= (.\ (return) ...)
       (let ((resume-thunk) (k_yield))

         (define (my-job value)
           (call-with-current-continuation
            (lambda (k_his-job)
              (set! resume-thunk (lambda () (k_his-job unspecific)))
              (k_yield value))))

         (define (his-job-coroutine-thunk)
           (call-with-current-continuation
            (lambda (k_my-job)
              (set! k_yield k_my-job)
              (resume-thunk))))

         (set! resume-thunk (his-job my-job))

         his-job-coroutine-thunk))
```

With this abstraction, we can make a fringe generator producer and fringe comparator consumer rather elegantly:

```
     (define *done* (list '*done*))

     (define (acs-coroutine-same-fringe? tree1 tree2)
       (let ((f1 (make-coroutine (acs-coroutine-fringe-generator tree1)))
             (f2 (make-coroutine (acs-coroutine-fringe-generator tree2))))
         (let lp ((x1 (f1)) (x2 (f2)))
           (cond ((and (eq? x1 *done*) (eq? x2 *done*)) #t)
                 ((or  (eq? x1 *done*) (eq? x2 *done*)) #f)
                 ((eq? x1 x2) (lp (f1) (f2)))
                 (else #f)))))

     (define (acs-coroutine-fringe-generator tree)
       (lambda (return)

         (define (lp tree)
           (cond ((pair? tree)
                  (lp (car tree))
                  (lp (cdr tree)))
                 ((null? tree) unspecific)
                 (else
                  (return tree))))

         (define (initial-generation-coroutine-thunk)
           (lp tree)
           (return *done*))

         initial-generation-coroutine-thunk))
```

```
          #|
          (acs-coroutine-same-fringe?
           '((a b) c ((d)) e (f ((g h))))
           '(a b c ((d) () e) (f (g (h)))))
          ;Value: #t

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

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

Problem 8.4:

Suppose I accidentally left out the return when done, so that the last line of the fringe generator's initial generation coroutine thunk were just *done* rather than "(return *done*)". What behavior would I get? Why? (I actually made this mistake. It took me about a 1/2 hour to figure out what went wrong! --GJS)

Communication among Threads

Now that we are all warmed up about continuations, you are ready to look at the time-sharing thread code in "conspire.scm", and the parallel execution code in "try-two-ways.scm". The time-sharing monitor can easily implement coroutines. You have an example with an explicit thread-yield in the first simple example in "conspire.scm". The return procedure above can be thought of as a thread yield. However, the coroutines in the time-shared environment do not easily communicate except through shared variables.

Time-sharing systems, such as GNU/Linux, provide explicit mechanisms, such as pipes, to make it easy for processes to communicate. A pipe is basically a FIFO communication channel which provides a reader and a writer. The writer puts things into the pipe and the reader takes them out. If we had pipes in conspire we could write the same-fringe? program as follows:

```
(define *done* (list '*done*))

(define (piped-same-fringe? tree1 tree2)
  (let ((p1 (make-pipe)) (p2 (make-pipe)))
    (let ((thread1
           (conspire:make-thread
            conspire:runnable
            (lambda ()
              (piped-fringe-generator tree1 (pipe-writer p1)))))
          (thread2
           (conspire:make-thread
            conspire:runnable
            (lambda ()
              (piped-fringe-generator tree2 (pipe-writer p2)))))
          (f1 (pipe-reader p1))
          (f2 (pipe-reader p2)))
      (let lp ((x1 (f1)) (x2 (f2)))
        (cond ((and (eq? x1 *done*) (eq? x2 *done*)) #t)
              ((or  (eq? x1 *done*) (eq? x2 *done*)) #f)
              ((eq? x1 x2) (lp (f1) (f2)))
              (else #f))))))

(define (piped-fringe-generator tree return)
  (define (lp tree)
    (cond ((pair? tree)
           (lp (car tree))
           (lp (cdr tree)))
          ((null? tree) unspecific)
          (else
           (return tree))))
  (lp tree)
  (return *done*))
```

Problem 8.5:

Implement the pipe mechanism implied by the program above. It should work under the conspire time-sharing monitor. Remember, if the pipe is empty a reader must wait until something is available to be read. Also, since this is supposed to work under preemptive time sharing, the pipe must be correctly interlocked.

With appropriate abstraction we can make the program look almost
exactly the same as the coroutine version:

```
(define *done* (list '*done*))

(define (tf-piped-same-fringe? tree1 tree2)
  (let ((f1 (make-threaded-filter (tf-piped-fringe-generator tree1)))
        (f2 (make-threaded-filter (tf-piped-fringe-generator tree2))))
    (let lp ((x1 (f1)) (x2 (f2)))
      (cond ((and (eq? x1 *done*) (eq? x2 *done*)) #t)
            ((or  (eq? x1 *done*) (eq? x2 *done*)) #f)
            ((eq? x1 x2) (lp (f1) (f2)))
            (else #f)))))

(define (tf-piped-fringe-generator tree)
  (lambda (return)
    (define (lp tree)
      (cond ((pair? tree)
             (lp (car tree))
             (lp (cdr tree)))
            ((null? tree) unspecific)
            (else
             (return tree))))
    (lp tree)
    (return *done*)))

#|
(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)))))
   ))
;Value: #t

(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)))))
   ))
;Value: #f

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

Problem 8.6:

Write make-threaded-filter to implement this interface. Demonstrate
your program.

```
;;;; File:  load.scm

;; On The Fringes of Fun with Control Structures
(load "same-fringe")

;; Communication among Threads
(load "conspire")
(load "try-two-ways")

':have-fun!
```

```
;;;; File:  conspire.scm

;;;;  CONSPIRE: Time Sharing in Scheme
;;;     "Processes scheming together
;;;       constitute a conspiracy"

;;; The essence of this system is that the state of a
;;; thread is specified by its continuation.  To switch
;;; threads we need to make a continuation, store it
;;; for the scheduler, and then retrieve a thread from
;;; the scheduler and start it running.  The thread has
;;; an identity, even though its continuation changes
;;; from time to time.
;;; A running thread can block itself until some
;;; predicate thunk becomes true by calling
;;; conspire:switch-threads with the predicate.

(define (conspire:switch-threads runnable?)
  (conspire:save-current-thread runnable?
        conspire:start-next-thread))

(define (conspire:save-current-thread runnable? after-save)
  (call-with-current-continuation
   (lambda (current-continuation)
     (without-interrupts
      (lambda ()
        (conspire:set-continuation! *running-thread* current-continuation)
        (conspire:add-to-schedule! runnable? *running-thread*)))
     (after-save))))

(define (conspire:start-next-thread)
  ((conspire:continuation
    (without-interrupts
     (lambda ()
       (set! *running-thread*
             (conspire:get-runnable-thread-from-schedule!))
       *running-thread*)))
   unspecific))

;;; A thread can explicitly yield control, remaining
;;; runnable.

(define (conspire:thread-yield)
  (conspire:switch-threads conspire:runnable))

(define conspire:runnable (lambda () #t))

;;; A thread can kill itself by starting some other thread
;;; without saving itself for rescheduling.

(define (conspire:kill-current-thread)
  (conspire:start-next-thread))

(define (conspire:kill-other-threads threads)
  (without-interrupts
   (lambda ()
     (for-each conspire:delete-from-schedule! threads))))
```

```
;;; A thread can make another thread and continue running.
;;; The thunk specified is the work order for the new thread.
;;; When the thunk returns the thread kills itself.

(define (conspire:make-thread runnable? thunk)
  (call-with-current-continuation
   (lambda (current-continuation)
     (within-continuation *root-continuation*
       (lambda ()
         (call-with-current-continuation
          (lambda (new-continuation)
            (current-continuation
             (without-interrupts
              (lambda ()
                (let ((new-thread
                       (conspire:make-new-thread
                        new-continuation)))
                  (conspire:add-to-schedule!
                     runnable? new-thread)
                  new-thread))))))
         (thunk)
         (conspire:kill-current-thread))))))


;;; A simple scheduler is just round-robin.

(define (conspire:add-to-schedule! runnable? thread)
  (queue:add-to-end! *thread-queue*
                     (cons runnable? thread)))

(define (conspire:get-runnable-thread-from-schedule!)
  (if (not (queue:empty? *thread-queue*))
      (let lp ((first (queue:get-first *thread-queue*)))
        (if ((car first))               ; runnable?
            (cdr first)
            (begin
              (queue:add-to-end! *thread-queue* first)
              (lp (queue:get-first *thread-queue*)))))
      (error "No current thread")))

(define (conspire:delete-from-schedule! thread)
  (let ((entry
         (find-matching-item
             (queue:front-ptr *thread-queue*)
           (lambda (entry)
             (eq? (cdr entry) thread)))))
    (if entry
        (queue:delete-from-queue! *thread-queue*
                                  entry))))
```

```
;;; We use the queue design similar to SICP Section 3.3.2

(define-record-type queue
    (queue:make-record front-ptr rear-ptr)
    queue?
  (front-ptr queue:front-ptr queue:set-front-ptr!)
  (rear-ptr  queue:rear-ptr  queue:set-rear-ptr!))

(define (queue:make)
  (queue:make-record '() '()))

(define (queue:empty? queue)
  (null? (queue:front-ptr queue)))

(define (queue:get-first queue)
  (if (null? (queue:front-ptr queue))
      (error "get-first called with an empty queue" queue)
      (let ((first (car (queue:front-ptr queue)))
            (rest (cdr (queue:front-ptr queue))))
        (queue:set-front-ptr! queue rest)
        (if (null? rest)
            (queue:set-rear-ptr! queue '()))
        first)))

(define (queue:add-to-end! queue item)
  (let ((new-pair (cons item '())))
    (cond ((null? (queue:front-ptr queue))
           (queue:set-front-ptr! queue new-pair)
           (queue:set-rear-ptr! queue new-pair))
          (else
           (set-cdr! (queue:rear-ptr queue) new-pair)
           (queue:set-rear-ptr! queue new-pair))))
  'done)

(define (queue:delete-from-queue! queue item)
  (queue:set-front-ptr! queue
                        (delq item
                              (queue:front-ptr queue)))
  (if (pair? (queue:front-ptr queue))
      (queue:set-rear-ptr! queue
                           (last-pair (queue:front-ptr queue)))
      (queue:set-rear-ptr! queue '()))
  'done)
```

```
(define-record-type conspire:thread
    (conspire:make-new-thread continuation)
    conspire:thread?
  (continuation conspire:continuation
                conspire:set-continuation!))


;;; Startup: have to make queue and first process

(define (with-conspiracy thunk)
  (fluid-let ((*running-thread*
               (conspire:make-new-thread unspecific))
              (*thread-queue* (queue:make))
              (*root-continuation*))
    (call-with-current-continuation
     (lambda (k)
       (set! *root-continuation* k)
       (thunk)))))

(define *running-thread*)

(define *thread-queue*)

(define *root-continuation*)
```

```
#|
;;; An elementary example:

(define (loop n)
  (let lp ((i 0))
    (if (< global-counter 1)
        'done
        (begin (set! global-counter (- global-counter 1))
               (if (= i n)
                   (begin (write-line `(,n ,global-counter))
                          (conspire:thread-yield)
                          (lp 0))
                   (lp (+ i 1)))))))

(define global-counter)

(with-conspiracy
    (lambda ()
      (set! global-counter 200)
      (conspire:make-thread conspire:runnable (lambda () (loop 31)))
      (conspire:make-thread conspire:runnable (lambda () (loop 37)))
      (repl/start (push-repl (nearest-repl/environment))
                  "; Entering conspiracy")))

(pp *thread-queue*)
#[queue 4]
(front-ptr
 ((#[compound-procedure 6 conspire:runnable] . #[conspire:thread 7])
  (#[compound-procedure 6 conspire:runnable] . #[conspire:thread 5])))
(rear-ptr
 ((#[compound-procedure 6 conspire:runnable] . #[conspire:thread 5])))

(conspire:thread-yield)
(31 168)
(37 130)
;Unspecified return value

;;; Got back to repl.

(conspire:thread-yield)
(31 98)
(37 60)
;Unspecified return value

(conspire:thread-yield)
(31 28)
;Unspecified return value

(conspire:thread-yield)
;Unspecified return value

(pp *thread-queue*)
#[queue 4]
(front-ptr ())
(rear-ptr ())

(abort->previous)                       ; Get out of repl.
|#
```

```
;;; Preemptive scheduling.

(define conspire:quantum 10)

(define conspire:running? #f)

;;; This is an MIT Scheme specific detail.  register-timer-event is
;;; the MIT Scheme mechanism for delivering a timer interrupt -- when
;;; the time specified by its first argument expires, it invokes the
;;; second argument.

(define (start-time-sharing)
  (let lp ()
    (if *debugging-time-sharing* (display "."))
    (if conspire:running?
        (begin
          (register-timer-event conspire:quantum
                                lp)
          (conspire:thread-yield))))
  'done)

(define *debugging-time-sharing* #f)


(define (with-time-sharing-conspiracy thunk)
  (fluid-let ((conspire:running? #t))
    (with-conspiracy
        (lambda ()
          (start-time-sharing)
          (thunk)))))

(define (conspire:null-job)
  (conspire:thread-yield)
  (if (queue:empty? *thread-queue*)
      'done
      (conspire:null-job)))
```

```
#|
;;; Our elementary example, again

(define (loop n)
  (let lp ((i 0))
    (if (< global-counter 1)
        'done
        (begin (set! global-counter (- global-counter 1))
               (if (= i n)
                   (begin (write-line `(,n ,global-counter))
                          (lp 0))
                   (lp (+ i 1)))))))


(define global-counter)

(with-time-sharing-conspiracy
    (lambda ()
      (set! global-counter 100000)
      (conspire:make-thread conspire:runnable (lambda () (loop 5555)))
      (conspire:make-thread conspire:runnable (lambda () (loop 4444)))
      (conspire:null-job)))

(5555 94444)
(5555 88888)
(5555 83332)
(5555 77776)
(4444 71412)
(4444 66967)
(4444 62522)
(4444 58077)
(4444 53632)
(4444 49187)
(4444 44742)
(5555 39853)
(5555 34297)
(5555 28741)
(5555 23185)
(5555 17629)
(4444 9782)
(4444 5337)
(4444 892)
;Value: done
|#
```

```
;;; Interlocks

(define-record-type conspire:lock
    (conspire:make-lock-cell state)
    conspire:lock?
  (state conspire:lock-state conspire:set-lock-state!))

(define (conspire:make-lock)
  (conspire:make-lock-cell #f))

(define (test-and-set-lock?! cell)
  (if (not (conspire:lock? cell))
      (error "Bad lock"))
  (without-interrupts
   (lambda ()
     (if (eq? (conspire:lock-state cell) #f)
         (begin (conspire:set-lock-state! cell #t)
                #t)
         #f))))

(define (conspire:unlock cell)
  (conspire:set-lock-state! cell #f))

(define (conspire:acquire-lock lock)
  (if (test-and-set-lock?! lock)
      'OK
      (conspire:switch-threads
       (lambda () (test-and-set-lock?! lock)))))
```

```
#|
;;; Our elementary example again:

(define global-counter-lock (conspire:make-lock))

(define (loop n)
  (let lp ((i 0))
    (let delaylp ((k 100))
      (if (> k 0)
          (delaylp (- k 1))))
     (conspire:acquire-lock global-counter-lock)
     (if (< global-counter 1)
         (begin
           (conspire:unlock global-counter-lock)
           'done)
         (begin (set! global-counter (- global-counter 1))
                (if (= i n)
                    (begin (write-line `(,n ,global-counter))
                           (conspire:unlock global-counter-lock)
                           (lp 0))
                    (begin
                      (conspire:unlock global-counter-lock)
                      (lp (+ i 1)))))))
  (write-line `(,n terminating)))

(define global-counter)

(set! conspire:quantum 5)

(with-time-sharing-conspiracy
    (lambda ()
      (set! global-counter 100000)
      (conspire:make-thread conspire:runnable (lambda () (loop 999)))
      (conspire:make-thread conspire:runnable (lambda () (loop 1000)))
      (conspire:null-job)))
|#
```

```
;;;; File:  try-two-ways.scm

(define (try-two-ways thunk1 thunk2)
  (let ((value) (done? #f))
    (let ((thread1
           (conspire:make-thread
            conspire:runnable
            (lambda ()
              (set! value (thunk1))
              (set! done? #t))))
          (thread2
           (conspire:make-thread
            conspire:runnable
            (lambda ()
              (set! value (thunk2))
              (set! done? #t)))))

      (conspire:switch-threads
       (lambda () done?))

      (conspire:kill-other-threads
       (list thread1 thread2))

      value)))
```

```
(define (test n1 n2)
  (with-conspiracy
      (lambda ()
        (try-two-ways
         (lambda ()
           (let lp ((n n1))
             (if (= n 0)
                 'a-done
                 (begin
                   (if (= (remainder n 100000) 0)
                       (begin (display 'a)
                              (conspire:thread-yield)))
                   (lp (- n 1))))))
         (lambda ()
           (let lp ((n n2))
             (if (= n 0)
                 'b-done
                 (begin
                   (if (= (remainder n 100000) 0)
                       (begin (display 'b)
                              (conspire:thread-yield)))
                   (lp (- n 1)))))))))))

#|
(test 1000000 1200000)
abababababababababab
;Value: a-done

(test 1200000 1000000)
bababababababababaa
;Value: b-done
|#
```

```
(define (test1 n1 n2)
  (with-time-sharing-conspiracy
      (lambda ()
        (try-two-ways
         (lambda ()
           (let lp ((n n1))
             (if (= n 0)
                 'a-done
                 (begin
                   (if (= (remainder n 100000) 0)
                       (display 'a))
                   (lp (- n 1))))))
         (lambda ()
           (let lp ((n n2))
             (if (= n 0)
                 'b-done
                 (begin
                   (if (= (remainder n 100000) 0)
                       (display 'b))
                   (lp (- n 1))))))))))

#|
(test1 1000000 1200000)
baabbaabbaabbaabbaabb
;Value: a-done

(test1 1200000 1000000)
babaabbaabbaabbaabbaa
;Value: b-done
|#
```

```
;;; Interesting example

;;; Suppose we want to search a list, that
;;; may be infinite (circular).  We could
;;; use the fast algorithm, but sometimes
;;; go into an infinite loop, or we could
;;; use the slow algorithm that marks the
;;; list (with a hash table) but always
;;; works.  If the statistics are right,
;;; a better strategy is to time-share the
;;; two methods and take the one which
;;; finishes first:

(define (safe-mem? item lst)
  (let ((table (make-eq-hash-table)))
    (let lp ((lst lst))
      (if (pair? lst)
          (if (hash-table/get table lst #f)
              #f                         ;circular
              (if (eq? item (car lst))
                  #t
                  (begin
                    (hash-table/put! table lst #t)
                    (lp (cdr lst)))))
          #f))))

(define (unsafe-mem? item lst)
  (let lp ((lst lst))
    (if (pair? lst)
        (if (eq? item (car lst))
            #t
            (lp (cdr lst)))
        #f)))

#|
(define foo (list 'a 'b 'c 'd))
;Value: foo

(begin (set-cdr! (last-pair foo) foo) 'foo)
;Value: foo

(unsafe-mem? 'b foo)
;Value: #t

(unsafe-mem? 'e foo)
;Quit!

(safe-mem? 'b foo)
;Value: #t

(safe-mem? 'e foo)
;Value: #f
|#
```

```
(define (mem? item lst)
  (with-time-sharing-conspiracy
      (lambda ()
        (try-two-ways
         (lambda ()
           (unsafe-mem? item lst))
         (lambda ()
           (safe-mem? item lst))))))

#|
(mem? 'b foo)
;Value: #t

(mem? 'e foo)
;Value: #f
|#
```