MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.945/6.S081 Spring 2014
Problem Set 8

Issued: Wed. 9 Apr. 2014                Due: Wed. 16 Apr. 2014

This is the last problem set for 6.S081.  There is one more for 6.945.

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)

## On The Fringes of Fun with Control Structures

To warm up with continuations, let’s review 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?
-------------

Remember, the goal is to enumerate the fringes of the two trees so that the corresponding pairs of elements can be compared without generating all of the pairs of elements first.

We can abstract this control structure, using underlying continuations.  Now things get very complicated.  This is a real brain teaser!  Here, a procedure that is to be used as a coroutine takes an argument: return.  Its value is a procedure that can be called to return a value from the coroutine.

The coroutine-manager procedure make-coroutine takes todo, a procedure that describes the computation to be done.  Todo gets a return procedure that it can call to deliver an incremental result.  The coroutine-manager uses continuations to save the future of the todo, when it calls the return procedure, so that the todo can be resumed at the result of the return.

```
(define (make-coroutine todo)          ;todo=(lambda (return) ...)
  (let ((todo-future #f))
    (lambda (supplicant)               ;supplicant=(lambda (value) ...)
      (let ((resume-point supplicant))
        (define (return value)
          (set! resume-point           ;gets (resume-point value)
                (call-with-current-continuation
                 (lambda (k)
                   (set! todo-future k)
                   (resume-point value)))))
        (if todo-future                ;is there a saved future?
            (todo-future supplicant)
            (todo return))))))         ;initial call
```

The next-value procedure creates a supplicant so that when the next value is obtained, the value is given to the supplicant.  It is amazing that we can use underlying Scheme continuations to provide the supplicants needed!

```
(define next-value
  call-with-current-continuation)
```

We can try this out with simple lists first.  Here is an appropriate list iterator that will deliver an element of a list each time it is poked with next-value:

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

(define (done? x) (eq? x *done*))

(define (list-iterator list)
  (make-coroutine
   (lambda (return)
     (for-each return list)
     (return *done*))))
```

```
It seems to work pretty well:
#|
(let ((foo (list-iterator '(1 2 3))))
  (let lp ((v (next-value foo)))
    (if (not (done? v))
        (begin (pp v)
               (lp (next-value foo)))
        'done)))
1
2
3
;Value: done

(let ((foo (list-iterator '(1 2 3)))
      (bar (list-iterator '(a b c))))
  (let lp ((v (next-value foo))
           (w (next-value bar)))
    (pp (list v w))
    (if (and (not (done? v)) (not (done? w)))
        (lp (next-value foo)
            (next-value bar))
        'done)))
(1 a)
(2 b)
(3 c)
((*done*) (*done*))
;Value: done
|#

-------------
Problem 8.4:

Unfortunately, the program fails in a strange way if we do not return
*done* using (return *done*).  Indeed, if we erroneously write

(define (list-iterator list)
  (make-coroutine
   (lambda (return)
     (for-each return list)
     *done*)))

we will get the following strange behavior:

(let ((foo (list-iterator '(1 2 3)))
      (bar (list-iterator '(a b c))))
  (let lp ((v (next-value foo)) (w (next-value bar)))
    (pp (list v w))
    (if (and (not (done? v)) (not (done? w)))
        (lp (next-value foo) (next-value bar))
        'done)))
(1 a)
(2 b)
(3 c)
((*done*) a)
;Value: done

Figure this out.  How do we get an asymmetric result here?  Where did
the "a" come from?  Write an explanation of this weird behavior.
-------------
```

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

```
(define (fringe-generator tree)
  (make-coroutine
   (lambda (return)
     (define (walk subtree)
       (cond ((pair? subtree)
              (walk (car subtree))
              (walk (cdr subtree)))
             ((null? subtree) 'goop)
             (else (return subtree))))
     (walk tree)
     (return *done*))))

(define (same-fringe? tree1 tree2)
  (let ((f1 (fringe-generator tree1))
        (f2 (fringe-generator tree2)))
    (let lp ((x1 (next-value f1))
             (x2 (next-value f2)))
      ;; (pp (list x1 x2))
      (cond ((and (done? x1) (done? x2))
             #t)
            ((or (done? x1) (done? x2))
             #f)
            ((eq? x1 x2)
             (lp (next-value f1)
                 (next-value f2)))
            (else #f)))))

#|
(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

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

## 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.
*****GJS:**** Say something about interrupt holes, for example between a test and use of the result. (fix is double-checked locks)
-------------

**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.**
**-------------**