MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science

6.891 Spring 2006
Problem Set 6

Issued: Wed. 22 March 2006   Due: Wed. 12 April 2006

Reading:

Online MIT/GNU Scheme Documentation,
Section 12.4 Continuations
Section 2.3 Dynamic Binding

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

Code: conspire.scm, try-two-ways.scm, samefringe.scm

Notice: This problem set is not due until 12 April. We need more time to prepare material to present in class, and you now have a term project to work on, so we will lighten up on the problem sets.

Fun with Control Structures

Scheme provides the ability for a programmer to get the continuation of an expression. Continuations are one of the most powerful, and the most dangerous, tools of a programmer. But most other languages do not support the use of first-class continuations. (Some languages that have first-class continuations are SML, Ruby, and Smalltalk.)

Explicit continuations may be powerful and sometimes useful, but they are rarely necessary. Let's look at a variety of ways to solve a famous problem, the samefringe problem.

The fringe of a tree is the ordered list of terminal leaves of the tree encountered when the tree is traversed in a 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 s)
  (cond ((pair? s)
         (append (fringe (car s))
                 (fringe (cdr s))))
        ((null? s) '())
        (else (list s))))
```

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)
|#
```

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

Problem 6.1:

What is the worst-case time order of growth of this procedure? 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? Is the depth of the tree relevant? So make sure you explain your answer clearly.

Here is a nicer procedure that computes the fringe, without any nasty recopying.

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

So the samefringe problem appears really simple:

```
(define (samefringe t1 t2)
  (equal? (fringe t1) (fringe t2)))
```

Indeed, this works.

```
#|
(samefringe '((a b) c ((d)) e (f ((g h))))
            '(a b c ((d) e) (f (g (h)))))
;Value: #t

(samefringe '((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 as needed to compare them. One way to do this is with "lazy evaluation", (as in streams). This method only requires examining as much of the input trees as is necessary to decide that two fringes are not the same:

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

      (define (lazy-samefringe t1 t2)
        (let lp ((f1 (lazy-fringe t1))
                 (f2 (lazy-fringe t2)))
          (cond ((and (empty-stream? f1) (empty-stream? f2)) #t)
                ((or (empty-stream? f1) (empty-stream? f2)) #f)
                ((eq? (head f1) (head f2))
                 (lp (tail f1) (tail f2)))
                (else #f))))

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

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

Problem 6.2:

This implementation of lazy fringe has the same problem of recopying the resulting stream that our original fringe program had recopying the resulting list. Write a stream program that eliminates this recopying using a technique similar to what we did above.

An alternative incremental idea is to make coroutines that generate the fringes, using an explicit continuation argument and an assignment. Notice that in the following code we invent a special object *done*. Because it is a newly consed list, this object is only eq? to itself and it cannot be confused with any other object. This is a very common device for making unique objects.

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

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

```
(define (coroutine-samefringe t1 t2)
  (let ((f1 (fringe-generator t1))
        (f2 (fringe-generator t2)))
    (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 that in this code there is an assignment. This makes it possible for the procedures f1 and f2 that are the result of calling fringe generator to have different results each time that they are called. The assignment is what makes the fringe generator have a local state.

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

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

Problem 6.3:

Why is it necessary to have the expression "(lambda () (next))" rather than just "next" 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 proc)
  (let ((my-state) (his-state))
    (define (return value)
      (call-with-current-continuation
       (lambda (k)
         (set! my-state
               (lambda ()
                 (call-with-current-continuation
                  (lambda (his)
                    (set! his-state his)
                    (k unspecific)))))
         (his-state value))))
    (set! my-state (proc return))
    (lambda ()
      (call-with-current-continuation
       (lambda (his)
         (set! his-state his)
         (my-state))))))
```

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

```
(define (samefringe t1 t2)
  (let ((f1 (make-coroutine (fringe-gen t1)))
        (f2 (make-coroutine (fringe-gen t2))))
    (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 ((fringe-gen tree) return)
  (define (lp tree)
    (cond ((pair? tree)
           (lp (car tree))
           (lp (cdr tree)))
          ((null? tree))
          (else
           (return tree))))
  (lambda ()
    (lp tree)
    (return *done*)))
```

Problem 6.4:

Suppose I accidentally left out the return, so that the last line of the fringe generator were just *done* rather than (return *done*). What behavior would I get? Why? (I actually made this mistake. It took me about 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 a 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 samefringe program as follows:

```
(define (samefringe t1 t2)
  (let ((p1 (make-pipe)) (p2 (make-pipe)))
    (let ((thread1
           (conspire:make-thread
            conspire:runnable
            (lambda ()
              (fringe-gen t1 (pipe-writer p1)))))
          (thread2
           (conspire:make-thread
            conspire:runnable
            (lambda ()
              (fringe-gen t2 (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 (fringe-gen tree return)
  (define (lp tree)
    (cond ((pair? tree)
           (lp (car tree))
           (lp (cdr tree)))
          ((null? tree))
          (else
           (return tree))))
  (lp tree)
  (return *done*))
```

Problem 6.5:

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 (samefringe t1 t2)
        (let ((f1 (make-threaded-filter (fringe-gen t1)))
              (f2 (make-threaded-filter (fringe-gen t2))))
          (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 ((fringe-gen tree) return)
        (define (lp tree)
          (cond ((pair? tree)
                 (lp (car tree))
                 (lp (cdr tree)))
                ((null? tree))
                (else
                 (return tree))))
        (lp tree)
        (return *done*))
```

Problem 6.6:

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