Assignment 6: Streams — CS351—Fall 2008

Due 23:59 Sun 30-Nov-2008. Email *one text file* containing *all* your solutions to: `barak+cs351-hw5@cs.nuim.ie`. (This file should be loadable into Scheme, meaning essay question answers should be in comments.)

In this assignment we will use streams using the API defined in class, including the defined functions: `stream-car`, `stream-cdr`, `stream-map`, `stream-cons`, `stream-zip`, `stream-take`. You may also use other stream-related definitions from the lecture notes.

Solution: For expository simplicity, all solution code below assumes all streams to be infinite, so no account is taken of the possibility of a null stream.

1. Define `stream-circulate` which takes a non-empty list as an argument and returns a stream consisting of the elements of the given list, repeated over and over. Examples:
`(stream-take 10 (stream-circulate '(a b c)))` ⇒ `(a b c a b c a b c a)`
`(stream-take 10 (stream-circulate '(1)))` ⇒ `(1 1 1 1 1 1 1 1 1 1)`

   Solution:

```
(define stream-circulate
  (lambda (lis)

    ;; aux makes a stream with elements on init-lis first, followed
    ;; by a circular stream of the elements on lis
    (define aux
      (lambda (init-lis lis)
        (if (null? init-lis)
            (aux lis lis)
            (stream-cons (car init-lis)
                         (lambda ()
                           (aux (cdr init-lis) lis))))))

    (aux lis lis)))
```

2. Define `stream-uniq` which takes a stream and returns a stream containing the same sequence as the stream it was passed, except that adjacent identical elements are compressed out. Example:
`(stream-take 10 (stream-uniq (stream-circulate '(a a b b b c c c d d d d d d d d d a))))` =
`(a b c d a b c d a b)`

   Solution: One strategy is to look at the first two elements, and skip the 1st if it is equal to the 2nd.

```
(define stream-uniq
  (lambda (stream)
    (let ((x1 (stream-car stream))
          (d (stream-cdr stream)))
      (let ((x2 (stream-car d)))
        (if (equal? x1 x2)
            (stream-uniq d)
            (stream-cons x1
                         (lambda ()
                           (stream-uniq d)))))))))
```

   Another strategy is to write an auxiliary function to do the skipping,

```
;;; skips all initial elements equal to x
(define stream-skip-xs
  (lambda (stream x)
    (let ((a (stream-car stream)))
      (if (equal? a x)
          (stream-skip-xs (stream-cdr stream) x)
          (stream-cdr stream)))))

(define stream-uniq
  (lambda (stream)
    (lambda (s)
      (cond ((eq? s 'car) (stream-car stream))
            ((eq? s 'cdr) (stream-uniq (stream-skip-x stream (stream-car stream))))
            (else (error "invalid invocation"))))))
```

3. Define `stream-zip-with` which is a higher-order function taking a binary function and two streams, and returning a stream of the results of applying the given function to corresponding successive elements of the two streams. Example:
`(stream-take 10 (stream-zip-with / stream-of-1s (stream-cdr stream-of-naturals)))`
$\Rightarrow$ `(1 1/2 1/3 1/4 1/5 1/6 1/7 1/8 1/9 1/10)`

Solution:

```
;;; This is just like stream-map in the lecture notes,
;;; but with two streams instead of just one.
;;; Or, exactly like stream-+, but abstracted over the +.

(define stream-zip-with
  (lambda (f stream1 stream2)
    (lambda (s)
      (cond ((eq? s 'car) (f (stream-car stream1)
                             (stream-car stream2))) ; first elt of stream
            ((eq? s 'cdr) (stream-zip-with
                           f
                           (stream-cdr stream1)
                           (stream-cdr stream2))) ; remainder of stream
            (else (error "invalid invocation"))))))
```

4. We define a *sorted stream* to be a stream of numbers which are in strictly increasing numeric order.
   (a) Define `sorted-member?` which takes a number and a sorted stream and returns `#t` or `#f` depending on whether the given number is or is not an element of the stream. Examples:
   `(sorted-member? 13 stream-of-fibbs)` $\Rightarrow$ `#t`
   `(sorted-member? 14 stream-of-fibbs)` $\Rightarrow$ `#f`
   `(sorted-member? 573147844013817084100 stream-of-fibbs)` $\Rightarrow$ `#f`
   `(sorted-member? 573147844013817084101 stream-of-fibbs)` $\Rightarrow$ `#t`

   Solution:
   ```
   (define sorted-member?
     (lambda (x stream)
       (let ((e (stream-car stream)))
         (or (= x e)
             (and (< x e)
                  (sorted-member x (stream-cdr stream)))))))
   ```

   (b) Define `sorted-merge` which takes two sorted streams and returns a sorted stream which contains all, and only, the elements of the two streams it was passed. Example:

```
(stream-take 55
             (sorted-merge stream-of-fibbs
                           (stream-map (lambda (x) (* x 10))
                                       stream-of-fibbs)))
```

⇒

```
(1 1 2 3 5 8 10 10 13 20 21 30 34 50 55 80 89 130 144 210 233 340 377
 550 610 890 987 1440 1597 2330 2584 3770 4181 6100 6765 9870 10946
 15970 17711 25840 28657 41810 46368 67650 75025 109460 121393 177110
 196418 286570 317811 463680 514229 750250 832040)
```

Solution:

```
(define sorted-merge
  (lambda (s1 s2)
    (let ((e1 (stream-car s1))
          (e2 (stream-car s2)))
      (if (< e1 e2)
          (stream-cons e1
                       (lambda ()
                         (sorted-merge (stream-cdr s1) s2)))
          (stream-cons e2
                       (lambda ()
                         (sorted-merge s1 (stream-cdr s2))))))))
```

5. Consider the set T of s-expressions representing binary trees over `x`, defined by the following: `x` $\in T$ and $(t_1\ \ t_2) \in T$ when $t_1 \in T$ and $t_2 \in T$. In other words, the symbol `x` is in T, and any two-element list both of whose elements are in T is also in T. By this definition, the following are in T:

```
x
(x x)
(((x (x x)) x) x)
((((x x) x) x) (((x (x x)) x) (((((x x) (x x)) (x x)) x) x)))
```

define `all-tees` which is a stream containing all, and only, elements of T.

Solution:

```
;;; Many ways of doing this; machinery below makes it straightforward

(define all-tees
  (stream-cons 'x
               (lambda ()
                 (stream-map-all-pairs list all-tees all-tees))))
```

6. *(For extra credit.)* Define `stream-map-all-pairs` which takes a binary function and two streams, and returns a stream containing the result of applying the given function to all pairs of elements consisting of one element of the 1st stream and one element of the 2nd stream. Order is arbitrary. E.g., (`stream-map-all-pairs` $f\ \ q\ \ r$) $\Rightarrow s$ where the stream q contains elements $q_0, q_1, \ldots$, the stream r contains elements $r_0, r_1 \ldots$, and the stream s contains elements $f(q_0, r_0), f(q_1, r_0), f(q_1, r_1), f(q_2, r_0), f(q_2, r_1), \ldots$, in some arbitrary order.

Solution:

```
;;; Strategy: we build two streams, one the maps the first
;;; elements of the first stream with *all* elements of the
;;; second stream, and the other that maps *all* remaining
;;; elements of the first stream with *all* the elements of
;;; the second stream.  Then we interleave their elements.
```

```
(define stream-map-all-pairs
  (lambda (f s1 s2)
    (let ((e1 (stream-car s1)))
      (stream-interleave
       (lambda () (stream-map (lambda (e2) (f e1 e2)) s2))
       (lambda () (stream-map-all-pairs f (stream-cdr s1) s2))))))

;;; This interleaves two streams, passed as thunks

(define stream-interleave
  (lambda (s1-thunk s2-thunk)
    (lambda (s)
      (cond ((eq? s 'car)
             (stream-car (s1-thunk)))
            ((eq? s 'cdr)
             (stream-interleave s2-thunk
                                (lambda () (stream-cdr (s1-thunk)))))
            (else (error "invalid invocation"))))))
```

7. *(Optional)* If you encountered any problems with the assignment, or have any comments on it, or other comments or suggestions, I would appreciate hearing them. As practice for actual work, where weekly reports are not unusual, please embody these in a brief report.

Solution:

This is the best class ever. My only suggestion: longer harder assignments. And more of them!

Honor Code: You may discuss these with others, but please write your answers by yourself and without reference to communal notes. In other words, your answers should be *from your own head.*