MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes – Oct. 15, 1998

STREAMS

Contract for the stream constructor and the selectors:

```
(stream-car (cons-stream <x> <y>))  ==> <x>
(stream-cdr (cons-stream <x> <y>))  ==> <y>
```

A simple implementation of streams simply lifts the list abstraction and renames it.

```
(define the-empty-stream '())
(define stream-null? null?)
(define cons-stream cons)
(define stream-car car)
(define stream-cdr cdr)
```

Now we can do all the normal sorts of things we do with lists:

```
(define (add-streams s1 s2)
  (cond ((stream-null? s1) the-empty-stream)
        ((stream-null? s2) the-empty-stream)
        (else (cons-stream
               (+ (stream-car s1) (stream-car s2))
               (add-streams (stream-cdr s1)
                            (stream-cdr s2))))))

(define (stream-filter pred s)
  (cond ((stream-null? s) the-empty-stream)
        ((not (pred (stream-car s)))
         (stream-cdr s))
        (else (cons-stream (stream-car s)
                           (stream-filter pred
                                          (stream-cdr s))))))

(define (stream-ref s n)
  (if (= n 0)
    (stream-car s)
    (stream-ref (stream-cdr s) (- n 1))))

(define (stream-map proc s)
  (if (stream-null? s)
    the-empty-stream
    (cons-stream (proc (stream-car s))
                 (stream-map proc (stream-cdr s)))))

(define (stream-scale factor s)
  (stream-map (lambda (x) (* factor x))
              s))

(define (stream-for-each proc s)
  (if (stream-null? s)
    'done
    (begin (proc (stream-car s))
           (stream-for-each proc (stream-cdr s)))))
```

```
(define (display-stream s)
  (stream-for-each display-line s))

(define (display-line x)
  (newline)
  (display x))
```

We can make up streams of integers over some interval with:

```
(define (stream-enumerate-interval lo hi)
  (if (> lo hi)
    the-empty-stream
    (cons-stream lo
                 (stream-enumerate-interval (+ 1 lo)
                                            hi))))
```

And we can accumulate sums, products, or many intricate other things with:

```
(define (accumulate-stream combiner initial s)
  (if (stream-null? s)
    initial
    (combiner (stream-car s)
              (accumulate-stream combiner
                                 initial
                                 (stream-cdr s)))))
```

Now we can program complex things without making explicit reference to iteration:

```
(define (sum-odd-squares from to)
  (accumulate-stream
   +
   0
   (stream-map square
               (stream-filter odd?
                              (stream-enumerate-interval from to)))))
(define (integral f lo hi dx)
  (* dx
     (accumulate-stream
      +
      0
      (stream-map f
                  (stream-map (lambda (x) (+ lo x))
                              (stream-scale dx
                                            (stream-enumerate-interval
                                             0
                                             (ceiling (/ (- hi lo) dx)))))))))
```

But why did we need the stream abstraction? Why not just use lists? It turns out that with a very simple twist to the implementation of our abstraction we get streams *infinitely* more powerful than lists.

Suppose we have a special form `delay` such that `(delay` *exp*`)` does not evaluate *exp*, but rather returns an object that will later evaluate *exp* when that object is given to a procedure `force` as its argument.

Now we slightly modify our constructor and selectors. In particular we make `cons-stream` into a special form, so that its second argument does not get evaluated until we look at it with `stream-cdr`.

```
(cons-stream <a> <b>)  equivalent to (cons <a> (delay <b>))

(define (stream-car stream) (car stream)      ;; same as before
(define (stream-cdr stream) (force (cdr stream)))
```

It turns out that `delay` and `force` are not too outrageous.

```
(delay <exp>)  equivalent to (lambda () <exp>)

(define (force delayed-object)
  (delayed-object))
```

Now we can make infinite streams! As we don't look too far along them they won't be there and we won't have any problems.

```
(define (integers-starting-from n)
  (cons-stream n (integers-starting-from (+ n 1))))

(define integers (integers-starting-from 1)))
(define (fibgen a b)
  (cons-stream a (fibgen b (+ a b))))

(define fibs (fibgen 0 1))
(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens
  (stream-filter (lambda (x) (not (divisible? x 7)))
                 integers))
```

is an infinite stream of all integers not divisible by 7.

```
(stream-ref no-sevens 100)   ==> 117
```

We can manipulate it just like a finite sized object.

```
(define (sieve s)
  (cons-stream
   (stream-car s)
   (sieve (stream-filter
            (lambda (x)
               (not (divisible? x (stream-car s))))
            (stream-cdr s)))))

(define primes (sieve (integers-starting-from 2)))
```

Recursive definitions of infinite streams:

```
(define ones (cons-stream 1 ones))

(define integers (cons-stream 1 (add-streams ones integers)))

(define fibs
  (cons-stream 0
               (cons-stream 1
                            (add-streams (stream-cdr fibs)
                                         fibs))))
(define double (cons-stream 1 (stream-scale 2 double)))
```