STREAMS

Contract for the stream constructor and the selectors:

\[
\begin{align*}
\text{stream-car (cons-stream } x \langle y \rangle) & \Rightarrow \langle x \rangle \\
\text{stream-cdr (cons-stream } x \langle y \rangle) & \Rightarrow \langle y \rangle
\end{align*}
\]

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

\[
\begin{align*}
\text{(define the-empty-stream '())} \\
\text{(define stream-null? null?)} \\
\text{(define cons-stream cons)} \\
\text{(define stream-car car)} \\
\text{(define stream-cdr cdr)}
\end{align*}
\]

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

\[
\begin{align*}
\text{(define (add-streams s1 s2)} \\
\text{\quad (cond ((stream-null? s1) the-empty-stream)} \\
\text{\quad \quad ((stream-null? s2) the-empty-stream)} \\
\text{\quad \quad (else cons-stream)} \\
\text{\quad \quad \quad (+ (stream-car s1) (stream-car s2))} \\
\text{\quad \quad \quad (add-streams (stream-cdr s1)} \\
\text{\quad \quad \quad \quad \quad (stream-cdr s2))))))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (stream-filter pred s)} \\
\text{\quad (cond ((stream-null? s) the-empty-stream)} \\
\text{\quad \quad ((not (pred (stream-car s)))} \\
\text{\quad \quad \quad (stream-cdr s))} \\
\text{\quad \quad \quad (else cons-stream (stream-car s)} \\
\text{\quad \quad \quad \quad \quad (stream-filter pred)} \\
\text{\quad \quad \quad \quad \quad \quad (stream-cdr s))))))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (stream-ref s n)} \\
\text{\quad (if (= n 0)} \\
\text{\quad \quad \quad (stream-car s) \quad (stream-ref (stream-cdr s) (- n 1)))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (stream-map proc s)} \\
\text{\quad (if (stream-null? s) \quad the-empty-stream)} \\
\text{\quad \quad \quad cons-stream (proc (stream-car s)) \quad (stream-map proc (stream-cdr s)))))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (stream-scale factor s)} \\
\text{\quad (stream-map (lambda (x) (* factor x)) \quad s))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (stream-for-each proc s)} \\
\text{\quad (if (stream-null? s) \quad 'done)} \\
\text{\quad \quad \quad \quad \quad \quad (begin (proc (stream-car s)) \quad (stream-for-each proc (stream-cdr s)))))}
\end{align*}
\]
(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) (+ 1 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.
It turns out that \texttt{delay} and \texttt{force} are not too outrageous.

It is not hard to see whether a number is divisible by 7. We can make an infinite stream of all integers not divisible by 7.

We can manipulate it just like a finite sized object.

Recursive definitions of infinite streams:

\begin{verbatim}
(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)))
\end{verbatim}