lib/basic/streams.ath

```
# Infinite streams here are represented as pairs of the form
# [x thunk], where x is the first element of the stream and
# thunk is a nullary closure able to generate the rest of
# the stream. An empty stream is represented by [].

(define empty-stream [])

(define stream-head first)

(define (stream-tail e) ((second e)))

(define empty-stream? null?)

(define (stream-cons x s)
  [x (lambda () s)])

(define (list->stream L)
  (match L
    ([] empty-stream)
    ((list-of x (some-list rest)) [x (lambda () (list->stream rest))])))

(define (stream-nth stream i)
  (check ((less? i 2) (stream-head stream))
         (else (stream-nth (stream-tail stream) (minus i 1)))))

(define (stream-map f s)
  (check ((empty-stream? s) s)
         (else [(f (stream-head s))
                (lambda () (stream-map f (stream-tail s)))])))

(define (stream-for-each s pred N)
  (letrec ((loop (lambda (s i)
                   (check ((|| (greater? i N) (empty-stream? s)) true)
                          ((pred (stream-head s)) (loop (stream-tail s) (plus i 1)))
                          (else false)))))
    (loop s 1)))

(define (stream-for-some s pred N)
  (letrec ((loop (lambda (s i)
                   (check ((|| (greater? i N) (empty-stream? s)) false)
                          ((pred (stream-head s)) true)
                          (else (loop (stream-tail s) (plus i 1)))))))
    (loop s 1)))


(define (stream-filter s pred)
  (check ((empty-stream? s) s)
         (else (let ((x (stream-head s)))
                 (check ((pred (stream-head s)) [x (lambda () (stream-filter (stream-tail s) pred))])
                        (else (stream-filter (stream-tail s) pred)))))))

# This is a friendly version of take: If the input stream is finite and
# has no more elements than n, then the entire stream is returned (as a list):

(define (stream-take stream n)
  (letrec ((loop (lambda (S i res)
                  (check ((leq? i 1) (check ((empty-stream? S) (rev res))
                                             (else (rev (add (stream-head S) res)))))
                         (else (check ((empty-stream? S) (rev res))
                                      (else (loop (stream-tail S) (minus i 1) (add (stream-head S) res)))))))))
    (check ((less? n 1) [])
           (else (loop stream n [])))))

# Interleaving two streams:

(define (weave-streams s1 s2)
  (check ((empty-stream? s1) s2)
         (else [(stream-head s1) (lambda () (weave-streams s2 (stream-tail s1)))])))
```

```

# Interleaving infinite streams raises questions of fairness and
# element distribution. The above version of weave always
# swaps orders on each call. The version below prefers drawing elements
# from the first stream with probability p, where 0 <= p <= 1.
# Thus, in the long run, (weave-streams-with-probability s1 s2 0.5)
# should give the same results as (weave-streams s1 s2):

(define (weave-streams-with-probability s1 s2 p)
  (check ((empty-stream? s1) s2)
         (else (let ((x (random-int 100)))
                 (check ((leq? x (times p 100))  [(stream-head s1) (lambda () (weave-streams-with-probability (stream-t
                        (else [(stream-head s2) (lambda () (weave-streams-with-probability s1 (stream-tail s2) p))]))))

# The procedure zip generates a stream representing the Cartesian product
# of input streams S1 and S2, either of which may be infinite. Moreover,
# it does this in a way that is fair and gives priority to left elements
# from both input streams. It walks through the (potentially infinite)
# two-dimensional matrix of all values from S1 and S2 in the style
# of Cantor's encoding of the rational numbers: by sweeping the matrix
# starting from the left upper-hand corners, then moving to the right,
# then down and left, and then back up and to the right again.
# The procedure does not need to keep track of positions (i,j) in
# the matrix: instead, it passes (and consumes) the matrix rows
# (each of which represents an infinite stream) dynamically as
# arguments. This leads to a quite efficient implementation.

(define (stream-zip S1 S2)
  (letrec ((getNext (lambda (front-streams back-streams first-stream)
                      (match front-streams
                        ([] (check ((empty-stream? first-stream)
                                    (match back-streams
                                      ([] empty-stream)
                                      (_ (getNext (rev back-streams) [] first-stream))))
                                 (else (let ((x (stream-head first-stream))
                                             (new-stream (stream-map (lambda (y) [x y]) S2)))
                                         (getNext (rev (add new-stream back-streams)) [] (stream-tail first-stream))
                        ((list-of stream-of-pairs more-streams)
                            (check ((empty-stream? stream-of-pairs)
                                    (getNext more-streams back-streams first-stream))
                                 (else (let ((pair (stream-head stream-of-pairs)))
                                         [pair (lambda ()
                                                 (getNext more-streams
                                                          (add (stream-tail stream-of-pairs) back-streams)
                                                           first-stream))]))))))))
     (check ((|| (empty-stream? S1) (empty-stream? S2)) empty-stream)
            (else (getNext [] [] S1)))))

(define (fair-weave stream-list)
  (letrec ((getNext (lambda (front-streams back-streams)
                      (match front-streams
                        ([] (match back-streams
                              ([] empty-stream)
                              (_ (getNext back-streams []))))
                        ((list-of (some-list S) (some-list more)) (check ((empty-stream? S) (getNext more back-streams
                                                        (else (let ((x (stream-head S)))
                                                                [x (lambda () (getNext more (add (stream-tail S) back-streams
      (match stream-list
        ([] empty-stream)
        (_ (getNext stream-list [])))))

(define (flatten-tuple L)
  (match L
    ([] [])
    ((list-of (as L' (list-of _ _)) more) (join (flatten-tuple L') (flatten-tuple more)))
    ((list-of x more) (add x (flatten-tuple more)))))

# The following is a generalization of stream-zip that can take any number
# of streams as input, packaged in a list:

```

```
(define (stream-zip* streams)
  (letrec ((loop (lambda (streams res)
                   (match streams
                     ([] res)
                     ((list-of (some-list S) (some-list more)) (loop more (stream-zip res S)))))))
     (match streams
       ([] empty-stream)
       ((list-of (some-list S) (some-list more))
          (stream-map flatten-tuple (loop more S))))))

(define (weave-streams* L)
                   (match L
                      ([] empty-stream)
                      ([(some-list s)] s)
                      (_ (let (([L1 L2] [(even-positions L) (odd-positions L)]))
                           (weave-streams (weave-streams* L1) (weave-streams* L2))))))


# Split an infinite stream in two roughly equal parts:

(define
  (stream-even-positions S)
     (check ((empty-stream? S) empty-stream)
            (else (stream-odd-positions (stream-tail S))))
  (stream-odd-positions S)
     (check ((empty-stream? S) empty-stream)
            (else [(stream-head S) (lambda () (stream-even-positions (stream-tail S)))])))

# Here L is an infinite stream of infinite streams:

(define (weave-streams** L)
  (check ((empty-stream? L) empty-stream)
         ((empty-stream? (stream-tail L)) (stream-head L))
         (else (let (([L1 L2] [(stream-even-positions L) (stream-odd-positions L)]))
                 (weave-streams (weave-streams** L1) (weave-streams** L2))))))


(define (all-from i) [i (lambda () (all-from (plus i 1)))])

(define (all-negative-integers-from i) [i (lambda () (all-negative-integers-from (minus i 1)))])

(define all-non-negative-integers (all-from 0))
(define all-positive-integers (all-from 1))
(define all-integers-less-than-or-equal-to-zero  (all-negative-integers-from 0))

(define all-integers (weave-streams-with-probability all-positive-integers all-integers-less-than-or-equal-to-zero 0.8

(define all-identifiers (stream-map (lambda (i)
                                       (string->id (join "x" (val->string i))))
                                    all-positive-integers))

(define non-negative-integers (all-from 0))

(define all-reals
   (stream-map (lambda (pair)
                 (match pair
                   ([(some-term i1) (some-term i2)] (string->num (join (val->string i1) "." (val->string i2))))))
               (stream-zip non-negative-integers non-negative-integers)))

(define all-numbers non-negative-integers)

(define (make-var n) (string->var (join "a" (symbol->string n))))

(define all-variables (stream-map make-var non-negative-integers))

(define (stream-append s1 s2)
  (check ((empty-stream? s1) s2)
         (else [(stream-head s1)
                (lambda () (stream-append (stream-tail s1) s2))])))
```

```
(define (stream-append* stream-list)
   (letrec ((loop (lambda (streams res)
                    (match streams
                      ([] res)
                      ((list-of (some-list stream) (some-list more)) (loop more (stream-append res stream)))))))
      (loop stream-list empty-stream)))

#(define s (stream-append* [all-numbers all-variables]))

(define st stream-take)

(define (stream-tail-k S k)
  (letrec ((loop (lambda (S i)
                   (check ((leq? i 0) S)
                          (else (check ((empty-stream? S) empty-stream)
                                       (else (loop (stream-tail S) (minus i 1)))))))))
     (loop S k)))

(define (stream-shuffle S k)
  (let ((chunk (st S k))
        (rest (stream-tail-k S k)))
    (stream-append (list->stream (rev chunk))
                   (check ((empty-stream? rest) empty-stream)
                          (else [(stream-head rest) (lambda () (stream-shuffle (stream-tail rest) k))])))))

## Flatten an infinite stream of (finite or infinite) streams into one infinite stream:

(define (stream-flatten* S)
  (check ((empty-stream? S) empty-stream)
        (else  (let ((S1 (stream-head S)))
                 (check ((empty-stream? S1) (stream-flatten* (stream-tail S)))
                        (else [(stream-head S1) (lambda () (stream-flatten* (stream-cons (stream-tail S1) (stream-tai
```