Scheme Request for Implementation 41: Streams

Philip L. Bewig

Streams, sometimes called lazy lists, are a sequential data structure containing elements computed only on demand. A stream is either null or is a pair with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again.

Streams without memoization were first described by Peter Landin in 1965. Memoization became accepted as an essential feature of streams about a decade later. Today, streams are the signature data type of functional programming languages such as Haskell.

This Scheme Request for Implementation describes two libraries for operating on streams: a canonical set of stream primitives and a set of procedures and syntax derived from those primitives that permits convenient expression of stream operations. They rely on facilities provided by R6RS, including libraries, records, and error reporting. To load both stream libraries, say:

```
(import (streams))
```

1. Streams

Harold Abelson and Gerald Jay Sussman discuss streams at length, giving a strong justification for their use. The streams they provide are represented as a `cons` pair with a promise to return a stream in its `cdr`; for instance, a stream with elements the first three counting numbers is represented conceptually as `(cons 1 (delay (cons 2 (delay (cons 3 (delay '()))))))`. Philip Wadler, Walid Taha and David MacQueen describe such streams as *odd* because, regardless of their length, the parity of the number of constructors (`delay`, `cons`, `'()`) in the stream is odd.

The streams provided here differ from those of Abelson and Sussman, being represented as promises that contain a `cons` pair with a stream in its `cdr`; for instance, the stream with elements the first three counting numbers is represented conceptually as `(delay (cons 1 (delay (cons 2 (delay (cons 3 (delay '()))))))))`; this is an *even* stream because the parity of the number of constructors in the stream is even.

Even streams are more complex than odd streams in both definition and usage, but they offer a strong benefit: they fix the off-by-one error of odd streams. Wadler, Taha and MacQueen show, for instance, that an expression like `(stream->list 4 (stream-map / (stream-from 4 -1)))` evaluates to `(1/4 1/3 1/2 1)` using even streams but fails with a divide-by-zero error using odd streams, because the next element in the stream, which will be `1/0`, is evaluated before it is accessed. This extra bit of laziness is not just an interesting oddity; it is vitally critical in many circumstances, as will become apparent below.

When used effectively, the primary benefit of streams is improved modularity. Consider a process that takes a sequence of items, operating on each in turn. If the operation is complex, it may be useful to split it into two or more procedures in which the partially-processed sequence is an intermediate result. If that sequence is stored as a list, the entire intermediate result must reside in memory all at once; however, if the intermediate result is stored as a stream, it can be generated piecemeal, using only as much memory as required by a single item. This leads to a programming style that uses many small operators, each operating on the sequence of items as a whole, similar to a pipeline of unix commands.

In addition to improved modularity, streams permit a clear exposition of backtracking algorithms using the “stream of successes” technique, and they can be used to model generators and co-routines. The implicit memoization of streams makes them useful for building persistent data structures, and the laziness of streams permits some multi-pass algorithms to be executed in a single pass. Savvy programmers use streams to enhance their programs in countless ways.

There is an obvious space/time trade-off between lists and streams; lists take more space, but streams take more time (to see why, look at all the type conversions in the implementation of the stream primitives). Streams are appropriate when the sequence is truly infinite, when the space savings are needed, or when they offer a clearer exposition of the algorithms that operate on the sequence.

2. The `(streams primitive)` library

The `(streams primitive)` library provides two mutually-recursive abstract data types: An object of the `stream` abstract data type is a promise that, when forced, is either `stream-null` or is an object of type stream-pair. An object of the stream-pair abstract data type contains a stream-car and a stream-cdr, which must be a stream. The essential feature of streams is the systematic suspensions of the recursive promises between the two data types.

```
α stream
    :: (promise stream-null)
    |  (promise (α stream-pair))

α stream-pair
    :: (promise α) × (promise (α stream))
```

The object stored in the stream-car of a stream-pair is a promise that is forced the first time the stream-car is accessed; its value is cached in case it is needed again. The object may have any type, and different stream elements may have different types. If the stream-car is never accessed, the object stored there is never evaluated. Likewise, the stream-cdr is a promise to return a stream, and is only forced on demand.

This library provides eight operators: constructors for stream-null and stream-pairs, type recognizers for streams and the two kinds of streams, accessors for both fields of a stream-pair, and a lambda that creates procedures that return streams.

stream-null **constructor**
Stream-null is a promise that, when forced, is a single object, distinguishable from all other objects, that represents the null stream. Stream-null is immutable and unique.

(stream-cons *object stream*) **constructor**
Stream-cons is a macro that accepts an *object* and a *stream* and creates a newly-allocated stream containing a promise that, when forced, is a stream-pair with the *object* in its stream-car and the *stream* in its stream-cdr. Stream-cons must be syntactic, not procedural, because neither *object* nor *stream* is evaluated when stream-cons is called. Since *stream* is not evaluated, when the stream-pair is created, it is not an error to call stream-cons with a *stream* that is not of type stream; however, doing so will cause an error later when the stream-cdr of the stream-pair is accessed. Once created, a stream-pair is immutable; there is no stream-set-car! or stream-set-cdr! that modifies an existing stream-pair. There is no dotted-pair or improper stream as with lists.

(stream? *object*) **recognizer**
Stream? is a procedure that takes an *object* and returns #t if the *object* is a stream and #f otherwise. If *object* is a stream, stream? does not force its promise. If (stream? obj) is #t, then one of (stream-null? obj) and (stream-pair? obj) will be #t and the other will be #f; if (stream? obj) is #f, both (stream-null? obj) and (stream-pair? obj) will be #f.

(stream-null? *object*) **recognizer**
Stream-null? is a procedure that takes an *object* and returns #t if the *object* is the distinguished null stream and #f otherwise. If *object* is a stream, stream-null? must force its promise in order to distinguish stream-null from stream-pair.

(stream-pair? *object*) **recognizer**
Stream-pair? is a procedure that takes an *object* and returns #t if the *object* is a stream-pair constructed by stream-cons and #f otherwise. If *object* is a stream, stream-pair? must force its promise in order to distinguish stream-null from stream-pair.

(stream-car *stream*) **accessor**
Stream-car is a procedure that takes a *stream* and returns the object stored in the stream-car of the *stream*. Stream-car signals an error if the object passed to it is not a stream-pair. Calling stream-car causes the object stored there to be evaluated if it has not yet been; the object's value is cached in case it is needed again.

(stream-cdr *stream*) **accessor**
Stream-cdr is a procedure that takes a *stream* and returns the stream stored in the stream-cdr of the *stream*. Stream-cdr signals an error if the object passed to it is not a stream-pair. Calling stream-cdr does not force the promise containing the stream stored in the stream-cdr of the *stream*.

(stream-lambda *args body*) **lambda**
Stream-lambda creates a procedure that returns a promise to evaluate the *body* of the procedure. The last *body* expression to be evaluated must yield a stream. As with normal lambda, *args* may be a single variable name, in which case all the formal arguments are collected into a single list, or a list of variable names, which may be null if there are no arguments, proper if there are an exact number of arguments, or dotted if a fixed number of arguments is to be followed by zero or more arguments collected into a list. *Body* must contain at least one expression, and may contain internal definitions preceding any expressions to be evaluated.

```
(define strm123
  (stream-cons 1
    (stream-cons 2
      (stream-cons 3
        stream-null))))

(stream-car strm123) ⇒ 1
(stream-car (stream-cdr strm123) ⇒ 2

(stream-pair?
  (stream-cdr
    (stream-cons (/ 1 0) stream-null))) ⇒ #f
```

```
(stream? (list 1 2 3)) ⇒ #f

(define iter
  (stream-lambda (f x)
    (stream-cons x (iter f (f x)))))

(define nats (iter (lambda (x) (+ x 1)) 0))

(stream-car (stream-cdr nats)) ⇒ 1

(define stream-add
  (stream-lambda (s1 s2)
    (stream-cons
      (+ (stream-car s1) (stream-car s2))
      (stream-add (stream-cdr s1)
                  (stream-cdr s2)))))

(define evens (stream-add nats nats))

(stream-car evens) ⇒ 0

(stream-car (stream-cdr evens)) ⇒ 2

(stream-car (stream-cdr (stream-cdr evens))) ⇒ 4
```

3. The `(streams derived)` library

The `(streams derived)` library provides useful procedures and syntax that depend on the primitives defined above. In the operator templates given below, an ellipsis `...` indicates zero or more repetitions of the preceding subexpression and square brackets `[…]` indicate optional elements. In the type annotations given below, square brackets `[…]` refer to lists, curly braces `{…}` refer to streams, and `nat` refers to exact non-negative integers.

`(define-stream (`*name args*`)` *body*`)` **syntax**

`Define-stream` creates a procedure that returns a stream, and may appear anywhere a normal `define` may appear, including as an internal definition, and may have internal definitions of its own, including other `define-stream`s. The defined procedure takes arguments in the same way as `stream-lambda`. `Define-stream` is syntactic sugar on `stream-lambda`; see also `stream-let`, which is also a sugaring of `stream-lambda`.

A simple version of `stream-map` that takes only a single input stream calls itself recursively:

```
(define-stream (stream-map proc strm)
  (if (stream-null? strm)
      stream-null
      (stream-cons
        (proc (stream-car strm))
        (stream-map proc (stream-cdr strm))))))
```

`(list->stream` *list-of-objects*`)` **procedure**

`[α] → {α}`

`List->stream` takes a list of objects and returns a newly-allocated stream containing in its elements the objects in the list. Since the objects are given in a list, they are evaluated when `list->stream` is called, before the stream is created. If the list of objects is null, as in `(list->stream '())`, the null stream is returned. See also `stream`.

```
(define strm123 (list->stream '(1 2 3)))

; fails with divide-by-zero error
(define s (list->stream (list 1 (/ 1 0) -1)))
```

`(port->stream [`*port*`])` **procedure**

`port → {char}`

`Port->stream` takes a *port* and returns a newly-allocated stream containing in its elements the characters on the port. If *port* is not given it defaults to the current input port. The returned stream has finite length and is terminated by `stream-null`.

It looks like one use of `port->stream` would be this:

```
(define s ;wrong!
  (with-input-from-file filename
    (lambda () (port->stream))))
```

But that fails, because `with-input-from-file` is eager, and closes the input port prematurely, before the first character is read. To read a file into a stream, say:

```
(define-stream (file->stream filename)
  (let ((p (open-input-file filename)))
    (stream-let loop ((c (read-char p)))
      (if (eof-object? c)
          (begin (close-input-port p)
                 stream-null)
          (stream-cons c
            (loop (read-char p)))))))
```

`(stream` *object* `...)` **syntax**

`Stream` is syntax that takes zero or more *object*s and creates a newly-allocated stream containing in its elements the *object*s, in order. Since `stream` is syntactic, the *object*s are evaluated when they are accessed, not when the stream is created. If no *object*s are given, as in `(stream)`, the null stream is returned. See also `list->stream`.

```
(define strm123 (stream 1 2 3))

; (/ 1 0) not evaluated when stream is created
(define s (stream 1 (/ 1 0) -1))
```

`(stream->list [`*n*`]` *stream*`)` **procedure**

`nat × {α} → [α]`

`Stream->list` takes a natural number *n* and a *stream* and returns a newly-allocated list containing in its elements the first *n* items in the *stream*. If the *stream* has less than *n* items all the items in the *stream* will be included in the returned list. If *n* is not given it defaults to infinity, which means that unless *stream* is finite `stream->list` will never return.

```
(stream->list 10
  (stream-map (lambda (x) (* x x))
    (stream-from 0)))
  ⇒ (0 1 4 9 16 25 36 49 64 81)
```

(stream-append *stream* ...) **procedure**

`{α} ... → {α}`

`Stream-append` returns a newly-allocated stream containing in its elements those elements contained in its input *stream*s, in order of input. If any of the input *stream*s is infinite, no elements of any of the succeeding input *stream*s will appear in the output stream; thus, if `x` is infinite, `(stream-append x y)` ≡ `x`. See also `stream-concat`.

Quicksort can be used to sort a stream, using `stream-append` to build the output; the sort is lazy; so if only the beginning of the output stream is needed, the end of the stream is never sorted.

```
(define-stream (qsort lt? strm)
  (if (stream-null? strm)
      stream-null
      (let ((x (stream-car strm))
            (xs (stream-cdr strm)))
        (stream-append
          (qsort lt?
            (stream-filter
              (lambda (u) (lt? u x))
              xs))
          (stream x)
          (qsort lt?
            (stream-filter
              (lambda (u) (not (lt? u x)))
              xs))))))
```

When used in tail position as in `qsort`, `stream-append` does not suffer the poor performance of `append` on lists. The list version of `append` requires re-traversal of all its list arguments except the last each time it is called. But `stream-append` is different. Each recursive call to `stream-append` is suspended; when it is later forced, the preceding elements of the result have already been traversed, so tail-recursive loops that produce streams are efficient even when each element is appended to the end of the result stream. This also implies that during traversal of the result only one promise needs to be kept in memory at a time.

(stream-concat *stream*) **procedure**

`{{α}} ... → {α}`

`Stream-concat` takes a *stream* consisting of one or more streams and returns a newly-allocated stream containing all the elements of the input streams. If any of the streams in the input *stream* is infinite, any remaining streams in the input *stream* will never appear in the output stream. See also `stream-append`.

```
(stream->list
  (stream-concat
    (stream
      (stream 1 2) (stream) (stream 3 2 1))))
  ⇒ (1 2 3 2 1)
```

The permutations of a finite stream can be determined by interleaving each element of the stream in all possible positions within each permutation of the other elements of the stream. `Interleave` returns a stream of streams with *x* inserted in each possible position of *yy*:

```
(define-stream (interleave x yy)
  (stream-match yy
    (() (stream (stream x)))
    ((y . ys)
      (stream-append
        (stream (stream-cons x yy))
        (stream-map
          (lambda (z) (stream-cons y z))
          (interleave x ys))))))

(define-stream (perms xs)
  (if (stream-null? xs)
      (stream (stream))
      (stream-concat
        (stream-map
          (lambda (ys)
            (interleave (stream-car xs) ys))
          (perms (stream-cdr xs))))))
```

(stream-constant *object* ...) **procedure**

`α ... → {α}`

`Stream-constant` takes one or more *object*s and returns a newly-allocated stream containing in its elements the *object*s, repeating the *object*s in succession forever.

```
(stream-constant 1) ⇒ 1 1 1 ...

(stream-constant #t #f) ⇒ #t #f #t #f #t #f ...
```

(stream-drop *n* *stream*) **procedure**

`nat × {α} → {α}`

`Stream-drop` returns the suffix of the input *stream* that starts at the next element after the first *n* elements. The output stream shares structure with the input *stream*; thus, promises forced in one instance of the stream are also forced in the other instance of the stream. If the input *stream* has less than *n* elements, `stream-drop` returns the null stream. See also `stream-take`.

```
(define (stream-split n strm)
  (values (stream-take n strm)
          (stream-drop n strm)))
```

(stream-drop-while *pred?* *stream*) **procedure**

`(α → boolean) × {α} → {α}`

`Stream-drop-while` returns the suffix of the input *stream* that starts at the first element *x* for which `(pred? x)` is `#f`. The output stream shares structure with the input *stream*. See also `stream-take-while`.

`Stream-unique` creates a new stream that retains only the first of any sub-sequences of repeated elements.

```
(define-stream (stream-unique eql? strm)
  (if (stream-null? strm)
      stream-null
      (stream-cons (stream-car strm)
        (stream-unique eql?
          (stream-drop-while
            (lambda (x)
              (eql? (stream-car strm) x))
            strm)))))
```

`(stream-filter` *pred? stream*`)` **procedure**

`(α → boolean) × {α} → {α}`

`Stream-filter` returns a newly-allocated stream that contains only those elements *x* of the input *stream* for which (*pred? x*) is non-`#f`.

```
(stream-filter odd? (stream-from 0))
   ⇒ 1 3 5 7 9 ...
```

`(stream-fold` *proc base stream*`)` **procedure**

`(α × β → α) × α × {β} → α`

`Stream-fold` applies a binary *proc*edure to *base* and the first element of *stream* to compute a new *base*, then applies the *proc*edure to the new *base* and the next element of *stream* to compute a succeeding *base*, and so on, accumulating a value that is finally returned as the value of `stream-fold` when the end of the *stream* is reached. *Stream* must be finite, or `stream-fold` will enter an infinite loop. See also `stream-scan`, which is similar to `stream-fold`, but useful for infinite streams. For readers familiar with other functional languages, this is a left-fold; there is no corresponding right-fold, since right-fold relies on finite streams that are fully-evaluated, at which time they may as well be converted to a list.

`Stream-fold` is often used to summarize a stream in a single value, for instance, to compute the maximum element of a stream.

```
(define (stream-maximum lt? strm)
  (stream-fold
    (lambda (x y) (if (lt? x y) y x))
    (stream-car strm)
    (stream-cdr strm)))
```

Sometimes, it is useful to have `stream-fold` defined only on non-null streams:

```
(define (stream-fold-one proc strm)
  (stream-fold proc
    (stream-car strm)
    (stream-cdr strm)))
```

`Stream-minimum` can then be defined as:

```
(define (stream-minimum lt? strm)
  (stream-fold-one
    (lambda (x y) (if (lt? x y) x y))
    strm))
```

`Stream-fold` can also be used to build a stream:

```
(define-stream (isort lt? strm)
    (define-stream (insert strm x)
      (stream-match strm
        (() (stream x))
        ((y . ys)
          (if (lt? y x)
              (stream-cons y (insert ys x))
              (stream-cons x strm)))))
    (stream-fold insert stream-null strm))
```

`(stream-for-each` *proc stream* `...)` **procedure**

`(α × β × ...) × {α} × {β} ...`

`Stream-for-each` applies a *proc*edure element-wise to corresponding elements of the input *stream*s for its side-effects; it returns nothing. `Stream-for-each` stops as soon as any of its input *stream*s is exhausted.

The following procedure displays the contents of a file:

```
(define (display-file filename)
  (stream-for-each display
    (file->stream filename)))
```

`(stream-from` *first* `[`*step*`])` **procedure**

`number × number → {number}`

`Stream-from` creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *step*. If *step* is not given it defaults to `1`. *First* and *step* may be of any numeric type. `Stream-from` is frequently useful as a generator in `stream-of` expressions. See also `stream-range` for a similar procedure that creates finite streams.

`Stream-from` could be implemented as `(stream-iterate (lambda (x) (+ x` *step*`))` *first*`)`.

```
(define nats (stream-from 0)) ⇒ 0 1 2 ...
```

```
(define odds (stream-from 1 2)) ⇒ 1 3 5 ...
```

`(stream-iterate proc base)` **procedure**

`(α → α) × α → {α}`

`Stream-iterate` creates a newly-allocated stream containing *base* in its first element and applies *proc* to each element in turn to determine the succeeding element. See also `stream-unfold` and `stream-unfolds`.

```
(stream-iterate (lambda (x) (+ x 1)) 0)
  ⇒ 0 1 2 3 4 ...
```

```
(stream-iterate (lambda (x) (* x 2)) 1)
  ⇒ 1 2 4 8 16 ...
```

Given a *seed* between `0` and 2^{32}, exclusive, the following expression creates a stream of pseudo-random integers between `0` and 2^{32}, exclusive, beginning with *seed*, using the method described by Stephen Park and Keith Miller:

```
(stream-iterate
  (lambda (x) (modulo (* x 16807) 2147483647))
  seed)
```

Successive values of the continued fraction shown below approach the value of the "golden ratio" φ ≈ 1.618:

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\cdots}}}}}}$$

The fractions can be calculated by the stream

```
(stream-iterate (lambda (x) (+ 1 (/ x))) 1)
```

`(stream-length` *stream*`)` **procedure**

`{α} → nat`

`Stream-length` takes an input *stream* and returns the number of elements in the *stream*; it does not evaluate its elements. `Stream-length` may only be used on finite streams; it enters an infinite loop with infinite streams.

```
(stream-length strm123) ⇒ 3
```

`(stream-let` *tag* `((`*var expr*`) ...)` *body*`)` **syntax**

`Stream-let` creates a local scope that binds each *var*iable to the value of its corresponding *expr*ession. It additionally binds *tag* to a procedure which takes the bound *var*iables as arguments and *body* as its defining expressions, binding the *tag* with `stream-lambda`. *Tag* is in scope within *body*, and may be called recursively. When the expanded expression defined by the `stream-let` is evaluated, `stream-let` evaluates the expressions in its *body* in an environment containing the newly-bound *va*riables, returning the value of the last expression evaluated, which must yield a stream.

`Stream-let` provides syntactic sugar on `stream-lambda`, in the same manner as normal `let` provides syntactic sugar on normal `lambda`. However, unlike normal `let`, the *tag* is required, not optional, because unnamed `stream-let` is meaningless.

`Stream-member` returns the first stream-pair of the input `strm` with a `stream-car` *x* that satisfies `(eql? obj` *x*`)`, or the null stream if *x* is not present in `strm`.

```
(define-stream (stream-member eql? obj strm)
  (stream-let loop ((strm strm))
    (cond ((stream-null? strm) strm)
          ((eql? obj (stream-car strm)) strm)
          (else (loop (stream-cdr strm))))))
```

`(stream-map` *proc stream* `...)` **procedure**

`(α × β ... → ω) × {α} × {β} ... → {ω}`

`Stream-map` applies a *proc*edure element-wise to corresponding elements of the input *stream*s, returning a newly-allocated stream containing elements that are the results of those *proc*edure applications. The output stream has as many elements as the minimum-length input *stream*, and may be infinite.

```
(define (square x) (* x x))

(stream-map square (stream 9 3)) ⇒ 81 9

(define (sigma f m n)
  (stream-fold + 0
    (stream-map f (stream-range m (+ n 1)))))

(sigma square 1 100) ⇒ 338350
```

In some functional languages, `stream-map` takes only a single input stream, and `stream-zipwith` provides a companion function that takes multiple input streams.

`(stream-match` *stream clause* `...)` **syntax**

`Stream-match` provides the syntax of pattern-matching for streams. The input *stream* is an expression that evaluates to a stream. Clauses are of the form `(`*pattern* `[`*fender*`]` *expr*`)`, consisting of a *pattern* that matches a stream of a particular shape, an optional *fender* that must succeed if the pattern is to match, and an *expr*ession that is evaluated if the *pattern* matches. There are four types of *pattern*s:

- `()` — Matches the null stream.
- `(`pat_0 pat_1 `...)` — Matches a finite stream with length exactly equal to the number of pattern elements.
- `(`pat_0 pat_1 `... .` pat_{rest}`)` — Matches an infinite stream, or a finite stream with length at least as great as the number of pattern elements before the literal dot.
- *pat* — Matches an entire stream. Should always appear last in the list of clauses; it's not an error to appear elsewhere, but subsequent clauses could never match.

Each pattern element pat_i may be either:

- An identifier — Matches any stream element. Additionally, the value of the stream element is bound to the variable named by the identifier, which is in scope in the *fender* and *expr*ession of the corresponding *clause*. Each identifier in a single pattern must be unique.
- A literal underscore — Matches any stream element, but creates no bindings.

The *pattern*s are tested in order, left-to-right, until a matching pattern is found; if *fender* is present, it must evaluate as non-`#f` for the match to be successful. Pattern variables are bound in the corresponding *fender* and *expr*ession. Once the matching pattern is found, the corresponding *expr*ession is evaluated and returned as the result of the match. An error is signaled if no pattern matches the input *stream*.

`Stream-match` is often used to distinguish null streams from non-null streams, binding `head` and `tail`:

```
(define (len strm)
  (stream-match strm
    (() 0)
    ((head . tail) (+ 1 (len tail)))))
```

Fenders can test the common case where two stream elements must be identical; the `else` pattern is an identifier bound to the entire stream, not a keyword as in `cond`.

```
(stream-match strm
  ((x y . _) (equal? x y) 'ok)
  (else 'error))
```

A more complex example uses two nested matchers to match two different stream arguments; `(stream-merge lt? .` *strms*`)` stably merges two or more streams ordered by the `lt?` predicate:

```
(define-stream (stream-merge lt? . strms)
  (define-stream (merge xx yy)
    (stream-match xx (() yy) ((x . xs)
      (stream-match yy (() xx) ((y . ys)
        (if (lt? y x)
            (stream-cons y (merge xx ys))
            (stream-cons x (merge xs yy))))))))
  (stream-let loop ((strms strms))
    (cond ((null? strms) stream-null)
          ((null? (cdr strms)) (car strms))
          (else (merge (car strms)
                       (apply stream-merge lt?
                         (cdr strms)))))))
```

(stream-of *expr clause* ...) **syntax**

`Stream-of` provides the syntax of stream comprehensions, which generate streams by means of looping expressions. The result is a stream of objects of the type returned by *expr*. There are four types of clauses:

- (*var* in *stream-expr*) — Loop over the elements of *stream-expr*, in order from the start of the stream, binding each element of the stream in turn to *var*. `Stream-from` and `stream-range` are frequently useful as generators for *stream-expr*.

- (*var* is *expr*) — Bind *var* to the value obtained by evaluating *expr*.

- (pred? *expr*) — Include in the output stream only those elements *x* for which (pred? *x*) is non-#f.

The scope of *var*iables bound in the stream comprehension is the clauses to the right of the binding clause (but not the binding clause itself) plus the result expression.

When two or more generators are present, the loops are processed as if they are nested from left to right; that is, the rightmost generator varies fastest. A consequence of this is that only the first generator may be infinite and all subsequent generators must be finite. If no generators are present, the result of a stream comprehension is a stream containing the result expression; thus, `(stream-of 1)` produces a finite stream containing only the element `1`.

```
(stream-of (* x x)
  (x in (stream-range 0 10))
  (even? x))
  ⇒ 0 4 16 36 64

(stream-of (list a b)
  (a in (stream-range 1 4))
  (b in (stream-range 1 3)))
  ⇒ (1 1) (1 2) (2 1) (2 2) (3 1) (3 2)

(stream-of (list i j)
  (i in (stream-range 1 5))
  (j in (stream-range (+ i 1) 5)))
  ⇒ (1 2) (1 3) (1 4) (2 3) (2 4) (3 4)
```

(stream-range *first past* [*step*]) **procedure**

number × number × number → {number}

`Stream-range` creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *step*. The stream is finite and ends before *past*, which is not an element of the stream. If *step* is not given it defaults to `1` if *first* is less than *past* and `-1` otherwise. *First*, *past* and *step* may be of any numeric type. `Stream-range` is frequently useful as a generator in `stream-of` expressions. See also `stream-from` for a similar procedure that creates infinite streams.

```
(stream-range 0 10) ⇒ 0 1 2 3 4 5 6 7 8 9

(stream-range 0 10 2) ⇒ 0 2 4 6 8
```

Successive elements of the stream are calculated by adding *step* to *first*, so if any of *first*, *past* or *step* are inexact, the length of the output stream may differ from (ceiling (- (/ (- *past first*) *step*) 1).

(stream-ref *stream n*) **procedure**

{α} × nat → α

`Stream-ref` returns the *n*th element of *stream*, counting from zero. An error is signaled if *n* is greater than or equal to the length of *stream*.

```
(define (fact n)
  (stream-ref
    (stream-scan * 1 (stream-from 1))
    n))
```

(stream-reverse *stream*) **procedure**

{α} → {α}

`Stream-reverse` returns a newly-allocated stream containing the elements of the input *stream* but in reverse order. `Stream-reverse` may only be used with finite streams; it enters an infinite loop with infinite streams. `Stream-reverse` does not force evaluation of the elements of the stream.

```
> (define s (stream 1 (/ 1 0) -1))
> (define r (stream-reverse s))
> (stream-ref r 0)
-1
> (stream-ref r 2)
1
> (stream-ref r 1)
error: division by zero
```

(stream-scan *proc base stream*) **procedure**

(α × β → α) × α × {β} → {α}

`Stream-scan` accumulates the partial folds of an input *stream* into a newly-allocated output stream. The output stream is the *base* followed by (stream-fold proc base (stream-take *i* stream)) for each of the first *i* elements of *stream*.

```
(stream-scan + 0 (stream-from 1))
  ⇒ (stream 0 1 3 6 10 15 ...)

(stream-scan * 1 (stream-from 1))
  ⇒ (stream 1 1 2 6 24 120 ...)
```

(stream-take *n stream*) **procedure**

nat × {α} → {α}

`Stream-take` takes a non-negative integer *n* and a *stream* and returns a newly-allocated stream containing the first *n* elements of the input *stream*. If the input *stream* has less than *n* elements, so does the output stream. See also `stream-drop`.

Mergesort splits a stream into two equal-length pieces, sorts them recursively and merges the results:

```
(define-stream (msort lt? strm)
  (let* ((n (quotient (stream-length strm) 2))
         (ts (stream-take n strm))
         (ds (stream-drop n strm)))
    (if (zero? n)
        strm
        (stream-merge lt?
          (msort < ts) (msort < ds)))))
```

`(stream-take-while` *pred? stream*`)` **procedure**

`(α → boolean) × {α} → {α}`

`Stream-take-while` takes a predicate and a *stream* and returns a newly-allocated stream containing those elements *x* that form the maximal prefix of the input *stream* for which `(pred?` *x*`)` is non-`#f`. See also `stream-drop-while`.

```
(stream-car
  (stream-reverse
    (stream-take-while
      (lambda (x) (< x 1000))
      primes))) ⇒ 997
```

`(stream-unfold` *map pred? gen base*`)` **procedure**

`(α → β) × (α → boolean) × (α → α) × α → {β}`

`Stream-unfold` is the fundamental recursive stream constructor. It constructs a stream by repeatedly applying *gen* to successive values of *base*, in the manner of `stream-iterate`, then applying *map* to each of the values so generated, appending each of the mapped values to the output stream as long as `(`*pred? base*`)` is non-`#f`. See also `stream-iterate` and `stream-unfolds`.

The expression below creates the finite stream `0 1 4 9 16 25 36 49 64 81`. Initially the `base` is `0`, which is less than `10`, so *map* squares the *base* and the mapped value becomes the first element of the output stream. Then *gen* increments the *base* by `1`, so it becomes `1`; this is less than `10`, so *map* squares the new `base` and `1` becomes the second element of the output stream. And so on, until the *base* becomes `10`, when *pred?* stops the recursion and `stream-null` ends the output stream.

```
(stream-unfold
  (lambda (x) (expt x 2)) ; map
  (lambda (x) (< x 10))   ; pred?
  (lambda (x) (+ x 1))    ; gen
  0)                      ; base
```

`(stream-unfolds` *proc seed*`)` **procedure**

`(α → (values α × β ...)) × α → (values {β} ...)`

`Stream-unfolds` returns *n* newly-allocated streams containing those elements produced by successive calls to the generator *proc*, which takes the current *seed* as its argument and returns *n*+1 values

`(`*proc seed*`)` → *seed* $result_0$ `...` $result_{n-1}$

where the returned *seed* is the input *seed* to the next call to the generator and $result_i$ indicates how to produce the next element of the i^{th} result stream:

- `(`*value*`)` — *value* is the next car of the result stream
- `#f` — no value produced by this iteration of the generator *proc* for the result stream
- `()` — the end of the result stream

It may require multiple calls of *proc* to produce the next element of any particular result stream. See also `stream-iterate` and `stream-unfold`.

`Stream-unfolds` is especially useful when writing expressions that return multiple streams. For instance, `(stream-partition` *pred? strm*`)` is equivalent to

```
(values
  (stream-filter pred? strm)
  (stream-filter
    (lambda (x) (not (pred? x))) strm))
```

but only tests *pred?* once for each element of *strm*.

```
(define (stream-partition pred? strm)
  (stream-unfolds
    (lambda (s)
      (if (stream-null? s)
          (values s '() '())
          (let ((a (stream-car s))
                (d (stream-cdr s)))
            (if (pred? a)
                (values d (list a) #f)
                (values d #f (list a))))))
    strm))
```

```
(call-with-values
  (lambda ()
    (stream-partition odd?
      (stream-range 1 6)))
  (lambda (odds evens)
    (list (stream->list odds)
          (stream->list evens))))
  ⇒ ((1 3 5) (2 4))
```

`(stream-zip` *stream* `...)` **procedure**

`{α} × {β} × ... → {[α β ...]}`

`Stream-zip` takes one or more input *stream*s and returns a newly-allocated stream in which each element is a list (not a stream) of the corresponding elements of the input *streams*. The output stream is as long as the shortest input *stream*, if any of the input *streams* is finite, or is infinite if all the input *stream*s are infinite.

A common use of `stream-zip` is to add an index to a stream, as in `(stream-finds` *eql? obj strm*`)`, which returns all the zero-based indices in *strm* at which *obj* appears; `(stream-find` *eql? obj strm*`)` returns the first such index, or `#f` if *obj* is not in *strm*.

```
(define-stream (stream-finds eql? obj strm)
  (stream-of (car x)
    (x in (stream-zip (stream-from 0) strm))
    (eql? obj (cadr x))))
```

```
(define (stream-find eql? obj strm)
  (stream-car
    (stream-append
      (stream-finds eql? obj strm)
      (stream #f))))
```

```
(stream-find char=? #\l
  (list->stream
    (string->list "hello"))) ⇒ 2
```

```
(stream-find char=? #\l
  (list->stream
    (string->list "goodbye"))) ⇒ #f
```

`Stream-find` is not as inefficient as it looks; although it calls `stream-finds`, which finds all matching indices, the matches are computed lazily, and only the first match is needed for `stream-find`.

4. Utilities

Streams, being the signature structured data type of functional programming languages, find useful expression in conjunction with higher-order functions. Some of these higher-order functions, and their relationship to streams, are described below.

The identity and constant procedures are frequently useful as the recursive base for maps and folds; **`(identity`** ***obj*****`)`** always returns *obj*, and **`(const`** ***obj*****`)`** creates a procedure that takes any number of arguments and always returns the same *obj*, no matter its arguments:

```
(define (identity obj) obj)
```

```
(define (const obj) (lambda x obj))
```

Many of the stream procedures take a unary predicate that accepts an element of a stream and returns a boolean. Procedure **`(negate`** ***pred?*****`)`** takes a unary predicate and returns a new unary predicate that, when called, returns the opposite boolean value as the original predicate.

```
(define (negate pred?)
  (lambda (x) (not (pred? x))))
```

`Negate` is useful for procedures like `stream-take-while` that take a predicate, allowing them to be used in the opposite direction from which they were written; for instance, with the predicate reversed, `stream-take-while` becomes `stream-take-until`. `Stream-remove` is the opposite of `stream-filter`:

```
(define-stream (stream-remove pred? strm)
  (stream-filter (negate pred?) strm))
```

A section is a procedure which has been partially applied to some of its arguments; for instance, `(double` *x*`)`, which returns twice its argument, is a partial application of the multiply operator to the number `2`. Sections come in two kinds: left sections partially apply arguments starting from the left, and right sections partially apply arguments starting from the right. Procedure **`(lsec`** ***proc args*** **`...)`** takes a *proc*edure and some prefix of its arguments and returns a new procedure in which those arguments are partially applied. Procedure **`(rsec`** ***proc args*** **`...)`** takes a *proc*edure and some reversed suffix of its arguments and returns a new procedure in which those arguments are partially applied.

```
(define (lsec proc . args)
  (lambda x (apply proc (append args x))))
```

```
(define (rsec proc . args)
  (lambda x (apply proc (reverse
    (append (reverse args) (reverse x))))))
```

Since most of the stream procedures take a stream as their last (right-most) argument, left sections are particularly useful in conjunction with streams.

```
(define stream-sum (lsec stream-fold + 0))
```

Function composition creates a new function by partially applying multiple functions, one after the other. In the simplest case there are only two functions, `f` and `g`, composed as `((compose f g)` *x*`)` ≡ `(f (g` *x*`))`; the composition can be bound to create a new function, as in `(define fg (compose f g))`. Procedure **`(compose`** ***proc*** **`...)`** takes one or more *proc*edures and returns a new procedure that performs the same action as the individual procedures would if called in succession.

```
(define (compose . fns)
  (let comp ((fns fns))
    (cond
      ((null? fns) 'error)
      ((null? (cdr fns)) (car fns))
      (else
        (lambda args
          (call-with-values
            (lambda ()
              (apply
                (comp (cdr fns))
                args))
            (car fns)))))))
```

`Compose` works with sections to create succinct but highly expressive procedure definitions. The expression to compute the squares of the integers from `1` to `10` given above at `stream-unfold` could be written by composing `stream-map`, `stream-take-while`, and `stream-iterate`:

```
((compose
  (lsec stream-map (rsec expt 2))
  (lsec stream-take-while (negate (rsec > 10)))
  (lsec stream-iterate (rsec + 1)))
 1)
```

5. Examples

The examples below show a few of the myriad ways streams can be exploited, as well as a few ways they can trip the unwary user. All the examples are drawn from published sources; it is instructive to compare the Scheme versions to the originals in other languages.

5.1. Infinite streams

As a simple illustration of infinite streams, consider this definition of the natural numbers:

```
(define nats
  (stream-cons 0
    (stream-map add1 nats)))
```

The recursion works because it is offset by one from the initial `stream-cons`. Another sequence that uses the offset trick is this definition of the fibonacci numbers:

```
(define fibs
  (stream-cons 1
    (stream-cons 1
      (stream-map +
        fibs
        (stream-cdr fibs)))))
```

Yet another sequence that uses the same offset trick is the Hamming numbers, named for the mathematician and computer scientist Richard Hamming, defined as all numbers that have no prime factors greater than 5; in other words, Hamming numbers are all numbers expressible as $2^i \cdot 3^j \cdot 5^k$, where i, j and k are non-negative integers. The Hamming sequence starts with `1 2 3 4 5 6 8 9 10 12` and is computed starting with `1`, taking `2`, `3` and `5` times all the previous elements with `stream-map`, then merging sub-streams and eliminating duplicates.

```
(define hamming
  (stream-cons 1
    (stream-unique =
      (stream-merge <
        (stream-map (lsec * 2) hamming)
        (stream-map (lsec * 3) hamming)
        (stream-map (lsec * 5) hamming)))))
```

It is possible to have an infinite stream of infinite streams. Consider the definition of `power-table`:

```
(define power-table
  (stream-of
    (stream-of (expt m n)
      (m in (stream-from 1)))
      (n in (stream-from 2))))
```

which evaluates to an infinite stream of infinite streams:

```
(stream
  (stream 1 4 9 16 25 ...)
  (stream 1 8 27 64 125 ...)
  (stream 1 16 81 256 625 ...)
  ...)
```

But even though it is impossible to display `power-table` in its entirety, it is possible to select just part of it:

```
(stream->list 10 (stream-ref power-table 1))
  ⇒ (1 8 27 64 125 216 343 512 729 1000)
```

This example clearly shows that the elements of a stream are computed lazily, as they are needed; `(stream-ref power-table 0)` is not computed, even when its successor is displayed, since computing it would enter an infinite loop.

Chris Reade shows how to calculate the stream of prime numbers according to the sieve of Eratosthenes, using a method that eliminates multiples of the sifting base with addition rather than division:

```
(define primes (let ()
  (define-stream (next base mult strm)
    (let ((first (stream-car strm))
          (rest (stream-cdr strm)))
      (cond ((< first mult)
              (stream-cons first
                (next base mult rest)))
            ((< mult first)
              (next base (+ base mult) strm))
            (else (next base
                    (+ base mult) rest)))))
  (define-stream (sift base strm)
    (next base (+ base base) strm))
  (define-stream (sieve strm)
    (let ((first (stream-car strm))
          (rest (stream-cdr strm)))
      (stream-cons first
        (sieve (sift first rest)))))
  (sieve (stream-from 2))))
```

A final example of infinite streams is a functional pearl from Jeremy Gibbons, David Lester and Richard Bird that enumerates the positive rational numbers without duplicates:

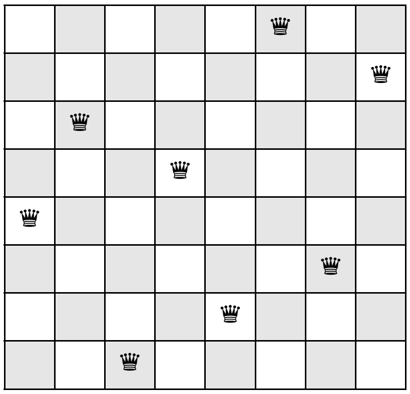
```
(define rats
  (stream-iterate
    (lambda (x)
      (let* ((n (floor x)) (y (- x n)))
        (/ (- n -1 y))))
    1))
```

5.2. Backtracking via the stream of successes

Philip Wadler describes the *stream of successes* technique that uses streams to perform backtracking search. The basic idea is that each procedure returns a stream of possible results, so that its caller can decide which result it wants; an empty stream signals failure, and causes backtracking to a previous choice point. The stream of successes technique is useful because the program is written as if to simply enumerate all possible solutions; no backtracking is explicit in the code.

The Eight Queens puzzle, which asks for a placement of eight queens on a chessboard so that none of them attack any other, is an example of a problem that can be solved using the stream of successes technique. The algorithm is to place a queen in the first column of a chessboard; any column is satisfactory. Then a queen is placed in the second column, in any position not held in check by the queen in the first column. Then a queen is placed in the third column, in any position not held in check by the queens in the first two columns. And so on, until all eight queens have been placed. If at any point there is no legal placement for the next queen, backtrack to a different legal position for the previous queens, and try again.

The chessboard is represented as a stream of length *m*, where there are queens in the first *m* columns, each position in the stream representing the rank on which the queen appears in that column. For example, stream `4 6 1 5 2 8 3 7` represents the following chessboard:



Two queens at column *i* row *j* and column *m* row *n* check each other if their columns *i* and *m* are the same, or if their rows *j* and *n* are the same, or if they are on the same diagonal with $i + j = m + n$ or $i - j = m - n$. There is no need to test the columns, because the placement algorithm enforces that they differ, so the `check?` procedure tests if two queens hold each other in check.

```
(define (check? i j m n)
  (or (= j n)
      (= (+ i j) (+ m n))
      (= (- i j) (- m n))))
```

The algorithm walks through the columns, extending position *p* by adding a new queen in row *n* with `(stream-append p (stream n))`. `Safe?` tests if it is safe to do so, using the utility procedure `stream-and`.

```
(define (stream-and strm)
  (let loop ((strm strm))
    (cond ((stream-null? strm) #t)
          ((not (stream-car strm)) #f)
          (else (loop (stream-cdr strm))))))

(define (safe? p n)
  (let* ((len (stream-length p))
         (m (+ len 1)))
    (stream-and
      (stream-of
        (not (check? (car ij) (cadr ij) m n))
          (ij in (stream-zip
                  (stream-range 1 m)
                  p))))))
```

Procedure `(queens m)` returns all the ways that queens can safely be placed in the first *m* columns.

```
(define (queens m)
  (if (zero? m)
      (stream (stream))
      (stream-of (stream-append p (stream n))
        (p in (queens (- m 1)))
        (n in (stream-range 1 9))
        (safe? p n))))
```

To see the first solution to the Eight Queens problem, say

```
(stream->list (stream-car (queens 8)))
```

To see all 92 solutions, say

```
(stream->list
  (stream-map stream->list
    (queens 8)))
```

There is no explicit backtracking in the code. The `stream-of` expression in `queens` returns all possible streams that satisfy `safe?`; implicit backtracking occurs in the recursive call to `queens`.

5.3 Generators and co-routines

It is possible to model generators and co-routines using streams. Consider the task, due to Carl Hewitt, of determining if two trees have the same sequence of leaves:

```
(same-fringe? = '(1 (2 3)) '((1 2) 3)) ⇒ #t

(same-fringe? = '(1 2 3) '(1 (3 2))) ⇒ #f
```

The simplest solution is to flatten both trees into lists and compare them element-by-element:

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((pair? (car tree))
          (append (flatten (car tree))
                  (flatten (cdr tree))))
        (else (cons (car tree)
                    (flatten (cdr tree))))))

(define (same-fringe? eql? tree1 tree2)
  (let loop ((t1 (flatten tree1))
             (t2 (flatten tree2)))
    (cond ((and (null? t1) (null? t2)) #t)
          ((or (null? t1) (null? t2)) #f)
          ((not (eql? (car t1) (car t2))) #f)
          (else (loop (cdr t1) (cdr t2))))))
```

That works, but requires time to flatten both trees and space to store the flattened versions; if the trees are large, that can be a lot of time and space, and if the fringes differ, much of that time and space is wasted.

Hewitt used a generator to flatten the trees one element at a time, storing only the current elements of the trees and the machines needed to continue flattening them, so `same-fringe?` could stop early if the trees differ. Dorai Sitaram presents both the generator solution and a co-routine solution, which both involve tricky calls to `call-with-current-continuation` and careful coding to keep them synchronized.

An alternate solution flattens the two trees to streams instead of lists, which accomplishes the same savings of time and space, and involves code that looks little different than the list solution presented above:

```
(define-stream (flatten tree)
  (cond ((null? tree) stream-null)
        ((pair? (car tree))
          (stream-append
            (flatten (car tree))
            (flatten (cdr tree))))
        (else (stream-cons
                (car tree)
                (flatten (cdr tree))))))
```

```
(define (same-fringe? eql? tree1 tree2)
  (let loop ((t1 (flatten tree1))
             (t2 (flatten tree2)))
    (cond ((and (stream-null? t1)
                (stream-null? t2)) #t)
          ((or  (stream-null? t1)
                (stream-null? t2)) #f)
          ((not (eql? (stream-car t1)
                      (stream-car t2))) #f)
          (else (loop (stream-cdr t1)
                      (stream-cdr t2))))))
```

Note that streams, a data structure, replace generators or co-routines, which are control structures, providing a fine example of how lazy streams enhance modularity.

5.4. A pipeline of procedures

Another way in which streams promote modularity is enabling the use of many small procedures that are easily composed into larger programs, in the style of unix pipelines, where streams are important because they allow a large dataset to be processed one item at a time. Bird and Wadler provide the example of a text formatter. Their example uses right-folds:

```
(define (stream-fold-right f base strm)
  (if (stream-null? strm)
      base
      (f (stream-car strm)
         (stream-fold-right f base
           (stream-cdr strm)))))

(define (stream-fold-right-one f strm)
  (stream-match strm
  ((x) x)
  ((x . xs)
    (f x (stream-fold-right-one f xs)))))
```

Bird and Wadler define text as a stream of characters, and develop a standard package for operating on text, which they derive mathematically (this assumes the line-separator character is a single `#\newline`):

```
(define (breakon a)
  (stream-lambda (x xss)
    (if (equal? a x)
        (stream-append (stream (stream)) xss)
        (stream-append
          (stream (stream-append
              (stream x) (stream-car xss)))
          (stream-cdr xss)))))

(define-stream (lines strm)
  (stream-fold-right
    (breakon #\newline)
    (stream (stream))
    strm))

(define-stream (words strm)
  (stream-filter stream-pair?
    (stream-fold-right
      (breakon #\space)
      (stream (stream))
      strm)))

(define-stream (paras strm)
  (stream-filter stream-pair?
    (stream-fold-right
      (breakon stream-null)
      (stream (stream))
      strm)))

(define (insert a)
  (stream-lambda (xs ys)
    (stream-append xs (stream a) ys)))

(define unlines
  (lsec stream-fold-right-one
    (insert #\newline)))

(define unwords
  (lsec stream-fold-right-one
    (insert #\space)))

(define unparas
  (lsec stream-fold-right-one
    (insert stream-null)))
```

These versatile procedures can be composed to count words, lines and paragraphs; the `normalize` procedure squeezes out multiple spaces and blank lines:

```
(define countlines
  (compose stream-length lines))

(define countwords
  (compose stream-length
           stream-concat
           (lsec stream-map words)
           lines))

(define countparas
  (compose stream-length paras lines))

(define parse
  (compose (lsec stream-map
             (lsec stream-map words))
           paras
           lines))

(define unparse
  (compose unlines
           unparas
           (lsec stream-map
             (lsec stream-map unwords))))

(define normalize (compose unparse parse))
```

More useful than normalization is text-filling, which packs as many words onto each line as will fit.

```
(define (greedy m ws)
  (- (stream-length
       (stream-take-while (rsec <= m)
         (stream-scan
           (lambda (n word)
             (+ n (stream-length word) 1))
           -1
           ws))) 1))

(define-stream (fill m ws)
  (if (stream-null? ws)
      stream-null
      (let* ((n (greedy m ws))
             (fstline (stream-take n ws))
             (rstwrds (stream-drop n ws)))
        (stream-append
          (stream fstline)
          (fill m rstwrds)))))

(define linewords
  (compose stream-concat
           (lsec stream-map words)))

(define textparas
  (compose (lsec stream-map linewords)
           paras
           lines))
```

```
(define (filltext m strm)
  (unparse
    (stream-map (lsec fill m)
      (textparas strm))))
```

To display *filename* in lines of *n* characters, say:

```
(stream-for-each display
  (filltext n (file->stream filename)))
```

Though each operator performs only a single task, they can be composed powerfully and expressively. The alternative is to build a single monolithic procedure for each task, which would be harder and involve repetitive code. Streams ensure procedures are called as needed.

5.5. Persistent data

Queues are one of the fundamental data structures of computer science. In functional languages, queues are commonly implemented using two lists, with the front half of the queue in one list, where the head of the queue can be accessed easily, and the rear half of the queue in reverse order in another list, where new items can easily be added to the end of a queue. The standard form of such a queue holds that the front list can only be null if the rear list is also null:

```
(define queue-null (cons '() '()))

(define (queue-null? obj)
  (and (pair? obj) (null? (car obj))))

(define (queue-check f r)
  (if (null? f)
      (cons (reverse r) '())
      (cons f r)))

(define (queue-snoc q x)
  (queue-check (car q) (cons x (cdr q))))

(define (queue-head q)
  (if (null? (car q))
      (error "empty queue: head")
      (car (car q))))

(define (queue-tail q)
  (if (null? (car q))
      (error "empty-head: tail")
      (queue-check (cdr (car q)) (cdr q))))
```

This queue operates in amortized constant time per operation, with two conses per element, one when it is added to the rear list, and another when the rear list is reversed to become the front list. `Queue-snoc` and `queue-head` operate in constant time; `queue-tail` operates in worst-case linear time when the front list is empty.

Chris Okasaki points out that, if the queue is used persistently, its time-complexity rises from linear to quadratic since each persistent copy of the queue requires its own linear-time access. The problem can be fixed by implementing the front and rear parts of the queue as streams, rather than lists, and rotating one element from rear to front whenever the rear list is larger than the front list:

```
(define queue-null
  (cons stream-null stream-null))

(define (queue-null? x)
  (and (pair? x) (stream-null (car x))))

(define (queue-check f r)
  (if (< (stream-length r) (stream-length f))
      (cons f r)
      (cons (stream-append f (stream-reverse r))
            stream-null)))

(define (queue-snoc q x)
  (queue-check (car q) (stream-cons x (cdr q))))

(define (queue-head q)
  (if (stream-null? (car q))
      (error "empty queue: head")
      (stream-car (car q))))

(define (queue-tail q)
  (if (stream-null? (car q))
      (error "empty queue: tail")
      (queue-check (stream-cdr (car q))
                   (cdr q))))
```

Memoization solves the persistence problem; once a queue element has moved from rear to front, it need never be moved again in subsequent traversals of the queue. Thus, the linear time-complexity to access all elements in the queue, persistently, is restored.

5.6. Reducing two passes to one

The final example is a lazy dictionary, where definitions and uses may occur in any order; in particular, uses may precede their corresponding definitions. This is a common problem. Many programming languages allow procedures to be used before they are defined. Macro processors must collect definitions and emit uses of text in order. An assembler needs to know the address that a linker will subsequently give to variables. The usual method is to make two passes over the data, collecting the definitions on the first pass and emitting the uses on the second pass. But Chris Reade shows how streams allow the dictionary to be built lazily, so that only a single pass is needed. Consider a stream of requests:

```
(define requests
  (stream
    '(get 3)
    '(put 1 "a")    ; use follows definition
    '(put 3 "c")    ; use precedes definition
    '(get 1)
    '(get 2)
    '(put 2 "b")    ; use precedes definition
    '(put 4 "d")))  ; unused definition
```

We want a procedure that will display `cab`, which is the result of `(get 3)`, `(get 1)`, and `(get 2)`, in order. We first separate the request stream into `gets` and `puts`:

```
(define (get? obj) (eq? (car obj) 'get))

(define-stream (gets strm)
  (stream-map cadr (stream-filter get? strm)))

(define-stream (puts strm)
  (stream-map cdr  (stream-remove get? strm)))
```

Now, `run-dict` inserts each element of the `puts` stream into a lazy dictionary, represented as a stream of key/value pairs (an association stream), then looks up each element of the `gets` stream with `stream-assoc`:

```
(define-stream (run-dict requests)
  (let ((dict (build-dict (puts requests))))
    (stream-map (rsec stream-assoc dict)
      (gets requests))))

(define (stream-assoc key dict)
    (cond ((stream-null? dict) #f)
          ((equal? key (car (stream-car dict)))
            (stream-car dict))
          (else (stream-assoc key
                  (stream-cdr dict)))))
```

`Dict` is created in the `let`, but nothing is initially added to it. Each time `stream-assoc` performs a lookup, enough of `dict` is built to satisfy the lookup, but no more. We are assuming that each item is defined once and only once. All that is left is to define the procedure that inserts new items into the dictionary, lazily:

```
(define-stream (build-dict puts)
  (if (stream-null? puts)
      stream-null
      (stream-cons
        (stream-car puts)
        (build-dict (stream-cdr puts)))))
```

Now we can run the requests and print the result:

```
(stream-for-each display
  (stream-map cadr (run-dict requests)))
```

The `(put 4 "d")` definition is never added to the dictionary because it is never needed.

5.7. Pitfalls

Programming with streams, or any lazy evaluator, can be tricky, even for programmers experienced in the genre. Programming with streams is even worse in Scheme than in a purely functional language, because, though the streams are lazy, the surrounding Scheme expressions in which they are embedded are eager. The impedance between lazy and eager can occasionally lead to astonishing results. Thirty-two years ago, William Burge warned:

> Some care must be taken when a stream is produced to make sure that its elements are not really a list in disguise, in other words, to make sure that the stream elements are not materialized too soon.

For example, a simple version of `stream-map` that returns a stream built by applying a unary procedure to the elements of an input stream could be defined like this:

```
(define-stream (stream-map proc strm) ;wrong!
  (let loop ((strm strm))
    (if (stream-null? strm)
        stream-null
        (stream-cons
          (proc (stream-car strm))
          (loop (stream-cdr strm))))))
```

That looks right. It properly wraps the procedure in `stream-lambda`, and the two legs of the `if` both return streams, so it type-checks. But it fails because the named `let` binds `loop` to a procedure using normal `lambda` rather than `stream-lambda`, so even though the first element of the result stream is lazy, subsequent elements are eager. `Stream-map` can be written using `stream-let`:

```
(define-stream (stream-map proc strm)
  (stream-let loop ((strm strm))
    (if (stream-null? strm)
        stream-null
        (stream-cons
          (proc (stream-car strm))
          (loop (stream-cdr strm))))))
```

Here, `stream-let` assures that each element of the result stream is properly delayed, because each is subject to the `stream-lambda` that is implicit in `stream-let`, so the result is truly a stream, not a "list in disguise." Another version of this procedure was given previously at the description of `define-stream`.

Another common problem occurs when a stream-valued procedure requires the next stream element in its definition. Consider this definition of `stream-unique`:

```
(define-stream (stream-unique eql? strm) ;wrong!
  (stream-match strm
    (() strm)
    ((_) strm)
    ((a b . _)
      (if (eql? a b)
          (stream-unique eql?
            (stream-cdr strm))
          (stream-cons a
            (stream-unique eql?
              (stream-cdr strm)))))))
```

The `(a b . _)` pattern requires the value of the next stream element after the one being considered. Thus, to compute the n^{th} element of the stream, one must know the $n+1^{st}$ element, and to compute the $n+1^{st}$ element, one must know the $n+2^{nd}$ element, and to compute…. The correct version, given above in the description of `stream-drop-while`, only needs the current stream element.

A similar problem occurs when the stream expression uses the previous element to compute the current element:

```
(define (nat n)
  (stream-ref
    (stream-let loop ((s (stream 0)))
      (stream-cons (stream-car s)
        (loop (stream (add1 (stream-car s))))))
    n))
```

This program traverses the stream of natural numbers, building the stream as it goes. The definition is correct; `(nat 15)` evaluates to `15`. But it needlessly uses unbounded space because each stream element holds the value of the prior stream element in the binding to `s`.

When traversing a stream, it is easy to write the expression in such a way that evaluation requires unbounded space, even when that is not strictly necessary. During the discussion of SRFI-40, Joe Marshall created this infamous procedure:

```
(define (times3 n)
  (stream-ref
    (stream-filter
      (lambda (x)
        (zero? (modulo x n)))
      (stream-from 0))
    3))
```

`(times3 5)` evaluates to `15` and `(times3 #e1e9)` evaluates to three billion, though it takes a while. In either case, `times3` should operate in bounded space, since each iteration mutates the promise that holds the next value. But it is easy to write `times3` so that it does not operate in bounded space, as the follies of SRFI-40 showed. The common problem is that some element of the stream (often the first element) is bound outside the expression that is computing the stream, so it holds the head of the stream, which holds the second element, and so on. In addition to testing the programmer, this procedure tests the stream primitives (it caught several errors during development) and also tests the underlying Scheme system (it found a bug in one implementation).

Laziness is no defense against an infinite loop; for instance, the expression below never returns, because the `odd?` predicate never finds an odd stream element.

```
(stream-null?
  (stream-filter odd?
    (stream-from 0 2)))
```

Ultimately, streams are defined as promises, which are implemented as thunks (lambda with no arguments). Since a stream is a procedure, comparisons such as `eq?`, `eqv?` and `equal?` are not meaningful when applied to streams. For instance, the expression `(define s ((stream-lambda () stream-null)))` defines `s` as the null stream, and `(stream-null? s)` is `#t`, but `(eq? s stream-null)` is `#f`. To determine if two streams are equal, it is necessary to evaluate the elements in their common prefixes, reporting `#f` if two elements ever differ and `#t` if both streams are exhausted at the same time.

```
(define (stream-equal? eql? xs ys)
  (cond ((and (stream-null? xs)
              (stream-null? ys)) #t)
        ((or (stream-null? xs)
             (stream-null? ys)) #f)
        ((not (eql? (stream-car xs)
                    (stream-car ys))) #f)
        (else (stream-equal? eql?
                (stream-cdr xs)
                (stream-cdr ys)))))
```

It is generally not a good idea to mix lazy streams with eager side-effects, because the order in which stream elements are evaluated determines the order in which the side-effects occur. For a simple example, consider this side-effecting version of `strm123`:

```
(define strm123-with-side-effects
  (stream-cons (begin (display "one") 1)
    (stream-cons (begin (display "two") 2)
      (stream-cons (begin (display "three") 3)
        stream-null))))
```

The stream has elements `1 2 3`. But depending on the order in which stream elements are accessed, `"one"`, `"two"` and `"three"` could be printed in any order.

Since the performance of streams can be very poor, normal (eager) lists should be preferred to streams unless there is some compelling reason to the contrary. For instance, computing pythagorean triples with streams

```
(stream-ref
  (stream-of (list a b c)
    (n in (stream-from 1))
    (a in (stream-range 1 n))
    (b in (stream-range a n))
    (c is (- n a b))
    (= (+ (* a a) (* b b)) (* c c)))
  50)
```

is about two orders of magnitude slower than the equivalent expression using loops.

```
(do ((n 1 (+ n 1))) ((> n 228))
  (do ((a 1 (+ a 1))) ((> a n))
    (do ((b a (+ b 1))) ((> b n))
      (let ((c (- n a b)))
        (if (= (+ (* a a) (* b b)) (* c c))
            (display (list a b c)))))))
```

6. Implementation

Bird and Wadler describe streams as either null or a pair with a stream in the tail:

```
α list :: null | α * α list
```

That works in a purely functional language such as Miranda or Haskell because the entire language is lazy. In an eager language like ML or Scheme, of course, it's just a normal, eager list.

Using ML, Wadler, Taha and MacQueen give the type of even streams as:

```
datatype 'a stream_
  = Nil_
  | Cons_ of 'a * 'a stream
withtype 'a stream
  = 'a stream_ susp;
```

Their `susp` type is similar to Scheme's `promise` type. Since Scheme conflates the notions of record and type (the only way to create a new type disjoint from all other types is to create a record), it is necessary to distribute the suspension through the two constructors of the stream data type:

```
α stream
  :: (promise stream-null)
  |  (promise (α stream-pair))

α stream-pair
  :: α × (α stream)
```

That type captures the systematic suspension of recursive promises that is the essence of "streamness." But it doesn't quite work, because Scheme is eager rather than lazy, and both the car and the cdr of the stream are evaluated too early. So the final type of streams delays both the car and the cdr of the stream-pair:

```
α stream
  :: (promise stream-null)
  |  (promise (α stream-pair))

α stream-pair
  :: (promise α) × (promise (α stream))
```

The two outer promises, in the `stream` type, provide streams without memoization. The two inner promises, in the `stream-pair` type, add the memoization that is characteristic of streams in modern functional languages.

Lists provide seven primitive operations: the two constructors `'()` and `cons`, the type predicates `list?`, `null?` and `pair?`, and the accessors `car` and `cdr` for pairs. All other list operations can be derived from those primitives.

It would seem that the same set of primitives could apply to streams, but in fact one additional primitive is required. André van Tonder describes the reason in his discussion of the promise data type. The promises of R6RS are inadequate to support iterative algorithms because each time a promise is called iteratively it binds the old promise in the closure that defines the new promise (so the old promise can be forced later, if requested). However, in the case of iteration, the old promise becomes unreachable, so instead of creating a new promise that binds the old promise within, it is better to mutate the promise; that way, no space is wasted by the old promise.

Van Tonder describes this new promise type, and provides a recipe for its use: all constructors are wrapped with `delay`, all accessors are wrapped with `force`, and all function bodies are wrapped with `lazy`. Given the seven primitives above, the first two parts of van Tonder's recipe are simple: the two constructors `stream-null` and `stream-pair` hide `delay`, and the two accessors `stream-car` and `stream-cdr` hide `force` (`stream-null?` and `stream-pair?` also hide `force`, so they can distinguish the two constructors of the stream type).

Although the new promise type prevents a space leak, it creates a new problem: there is no place to hide the `lazy` that is the third part of van Tonder's recipe. SRFI-40 solved this problem by exposing it (actually, it exposed `delay`, which was incorrect). But that violates good software engineering by preventing the stream data type from being fully abstract. The solution of SRFI-41 is to create a new primitive, `stream-lambda`, that returns a function that hides `lazy`.

Besides hiding `lazy` and making the types work out correctly, `stream-lambda` is obvious and easy-to-use for competent Scheme programmers, especially when augmented with the syntactic sugar of `define-stream` and named `stream-let`. The alternative of exposing `stream-lazy` would be less clear and harder to use.

One of the hardest tasks when writing any program library is to decide what to include and, more importantly, what to exclude. One important guideline is minimalism, since once an operator enters a library it must remain forever: *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher*.

Since streams are substantially slower than lists (the stream primitives require numerous type conversions, and list operations in most Scheme implementations are heavily optimized), most programmers will use streams only when the sequence of elements is truly infinite (such as mathematical series) or when there is some clear advantage of laziness (such as reducing the number of passes though a large data set). Thus, the library is biased toward functions that work with infinite streams left-to-right. In particular, there is no right-fold; if you need to materialize an entire stream, it's best to use a list.

The complete implementation is given in the appendices.

Acknowledgements

Jos Koot sharpened my thinking during many e-mail discussions, suggested several discussion points in the text, and contributed the final version of `stream-match`. Michael Sperber and Abdulaziz Ghuloum gave advice on R6RS.

References

Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts. Second edition, 1996. `mitpress.mit.edu/sicp`. The classic text on computer science. Section 3.5 includes extensive discussion of odd streams.

Anne L. Bewig. "Golden Ratio" (personal communication). Homework for the high school course *Calculus*. Teaching my daughter how to calculate the 200th element of a continued fraction was a moment of sheer joy in the development of the stream libraries.

Philip L. Bewig. *Scheme Request for Implementation 40: A Library of Streams*. August, 2004. `srfi.schemers.org/srfi-40`. Describes an implementation of the `stream` data type.

Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988. The classic text on functional programming. Even streams are discussed in the context of purely functional programming.

William H. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1975. An early text on functional programming, and still one of the best, though the terminology is dated. Discusses even streams in Section 3.10.

Jeremy Gibbons, David Lester and Richard Bird, "Functional Pearl: Enumerating the Rationals," under consideration for publication in *Journal of Functional Programming*. `http://web.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/rationals.pdf`. Discusses a series of expressions that enumerate the rational numbers without duplicates.

Carl Hewitt. "Viewing control structures as patterns of passing messages," in *Journal of Artificial Intelligence*, Volume 8, Number 3 (June, 1977), pp 323-364. Also published as Artificial Intelligence Memo 410 by the Massachusetts Institute of Technology, `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-410.pdf`. Describes the Actor message-passing system; one of the examples used is the `same-fringe?` problem.

Peter J. Landin. "A correspondence between ALGOL 60 and Church's lambda-notation: Part I," *Communications of the ACM*, Volume 8, Number 2, February 1965., pages 89–101. The seminal description of streams.

Joe Marshall. "Stream problem redux", from *Usenet comp.lang.scheme*, June 28, 2002. `groups.google.com/group/comp.lang.scheme/msg/db4b4a1f33e3eea8`. The original post on `comp.lang.scheme` that describes the `times3` problem.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 2003. Revised version of Okasaki's thesis *Purely Functional Data Structures*, Carnegie-Mellon University, 1996, `www.cs.cmu.edu/~rwh/theses/okasaki.pdf`. Provides a strong defense of laziness, and describes several data structures that exploit laziness, including streams and queues.

Stephen K. Park and Keith W. Miller. "Random number generators: good ones are hard to find," *Communications of the ACM*, Volume 31, Issue 10 (October 1988), pages 1192–1201. Describes a minimal standard random number generator.

Simon Peyton-Jones, et al, editors. *Haskell 98: Haskell 98 Language and Libraries: The Revised Report*. December 2002. `www.haskell.org/onlinereport`. Haskell is the prototypical purely functional language, and includes even streams, which it calls lists, as its fundamental structured data type.

Chris Reade. *Elements of Functional Programming*. Addison-Wesley, April 1989. A textbook on functional programming.

Antoine de Saint-Exupéry. Chapter III "L'Avion" of *Terre des Hommes*. 1939. "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Dorai Sitaram. *Teach Yourself Scheme in Fixnum Days*. `www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`. A useful introduction to Scheme; includes generator and co-routine solutions to the `same-fringe?` problem.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton von Straaten, editors. *Revised[6] Report on the Algorithmic Language Scheme*. September 26, 2007. `www.r6rs.org`. The standard definition of the Scheme programming language.

André van Tonder. *Scheme Request for Implementation 45: Primitives for Expressing Iterative Lazy Algorithms*. `srfi.schemers.org/srfi-45`. April, 2004. Describes the problems inherent in the `promise` data type of R5RS (also present in R6RS), and provides the alternate `promise` data type used in the stream primitives.

Philip Wadler. "How to replace failure by a list of successes," in *Proceedings of the conference on functional programming languages and computer architecture*, Nancy, France, 1985, pages 113–128. Describes the "list of successes" technique for implementing backtracking algorithms using streams.

Philip Wadler, Walid Taha, and David MacQueen, "How to add laziness to a strict language without even being odd." 1998 ACM SIGPLAN Workshop on ML, pp. 24ff. `homepages.inf.ed.ac.uk/wadler/papers/lazyinstrict/lazyinstrict.ps`. Describes odd and even styles of lazy evaluation, and shows how to add lazy evaluation to the strict functional language SML.

All cited web pages visited during September 2007.

Appendix 1: Implementation of `(streams primitive)`

```
(library (streams primitive)

  (export stream-null stream-cons stream? stream-null? stream-pair?
          stream-car stream-cdr stream-lambda)

  (import (rnrs) (rnrs mutable-pairs))

  (define-record-type (stream-type make-stream stream?)
    (fields (mutable box stream-promise stream-promise!)))
```

```
(define-syntax stream-lazy
  (syntax-rules ()
    ((stream-lazy expr)
      (make-stream
        (cons 'lazy (lambda () expr))))))

(define (stream-eager expr)
  (make-stream
    (cons 'eager expr)))

(define-syntax stream-delay
  (syntax-rules ()
    ((stream-delay expr)
      (stream-lazy (stream-eager expr)))))

(define (stream-force promise)
  (let ((content (stream-promise promise)))
    (case (car content)
      ((eager) (cdr content))
      ((lazy)  (let* ((promise* ((cdr content)))
                      (content  (stream-promise promise)))
                 (if (not (eqv? (car content) 'eager))
                     (begin (set-car! content (car (stream-promise promise*)))
                            (set-cdr! content (cdr (stream-promise promise*)))
                            (stream-promise! promise* content)))
                 (stream-force promise))))))

(define stream-null (stream-delay (cons 'stream 'null)))

(define-record-type (stream-pare-type make-stream-pare stream-pare?)
  (fields (immutable kar stream-kar) (immutable kdr stream-kdr)))

(define (stream-pair? obj)
  (and (stream? obj) (stream-pare? (stream-force obj))))

(define (stream-null? obj)
  (and (stream? obj)
       (eqv? (stream-force obj)
             (stream-force stream-null))))

(define-syntax stream-cons
  (syntax-rules ()
    ((stream-cons obj strm)
      (stream-eager (make-stream-pare (stream-delay obj) (stream-lazy strm))))))

(define (stream-car strm)
  (cond ((not (stream? strm)) (error 'stream-car "non-stream"))
        ((stream-null? strm) (error 'stream-car "null stream"))
        (else (stream-force (stream-kar (stream-force strm))))))

(define (stream-cdr strm)
  (cond ((not (stream? strm)) (error 'stream-cdr "non-stream"))
        ((stream-null? strm) (error 'stream-cdr "null stream"))
        (else (stream-kdr (stream-force strm)))))

(define-syntax stream-lambda
  (syntax-rules ()
    ((stream-lambda formals body0 body1 ...)
      (lambda formals (stream-lazy (let () body0 body1 ...))))))
```

Appendix 2: Implementation of `(streams derived)`

```
(library (streams derived)

  (export define-stream list->stream port->stream stream stream->list
          stream-append stream-concat stream-constant stream-drop
          stream-drop-while stream-filter stream-fold stream-for-each stream-from
          stream-iterate stream-length stream-let stream-map stream-match _
          stream-of stream-range stream-ref stream-reverse stream-scan stream-take
          stream-take-while stream-unfold stream-unfolds stream-zip)

  (import (rnrs) (streams primitive))
```

```
(define-syntax define-stream
  (syntax-rules ()
    ((define-stream (name . formal) body0 body1 ...)
      (define name (stream-lambda formal body0 body1 ...)))))

(define (list->stream objs)
  (define list->stream
    (stream-lambda (objs)
      (if (null? objs)
          stream-null
          (stream-cons (car objs) (list->stream (cdr objs))))))
  (if (not (list? objs))
      (error 'list->stream "non-list argument")
      (list->stream objs)))

(define (port->stream . port)
  (define port->stream
    (stream-lambda (p)
      (let ((c (read-char p)))
        (if (eof-object? c)
            stream-null
            (stream-cons c (port->stream p))))))
  (let ((p (if (null? port) (current-input-port) (car port))))
    (if (not (input-port? p))
        (error 'port->stream "non-input-port argument")
        (port->stream p))))

(define-syntax stream
  (syntax-rules ()
    ((stream) stream-null)
    ((stream x y ...) (stream-cons x (stream y ...)))))

(define (stream->list . args)
  (let ((n (if (= 1 (length args)) #f (car args)))
        (strm (if (= 1 (length args)) (car args) (cadr args))))
    (cond ((not (stream? strm)) (error 'stream->list "non-stream argument"))
          ((and n (not (integer? n))) (error 'stream->list "non-integer count"))
          ((and n (negative? n)) (error 'stream->list "negative count"))
          (else (let loop ((n (if n n -1)) (strm strm))
                  (if (or (zero? n) (stream-null? strm))
                      '()
                      (cons (stream-car strm) (loop (- n 1) (stream-cdr strm)))))))))

(define (stream-append . strms)
  (define stream-append
    (stream-lambda (strms)
      (cond ((null? (cdr strms)) (car strms))
            ((stream-null? (car strms)) (stream-append (cdr strms)))
            (else (stream-cons (stream-car (car strms))
                               (stream-append (cons (stream-cdr (car strms)) (cdr strms))))))))
  (cond ((null? strms) stream-null)
        ((exists (lambda (x) (not (stream? x))) strms)
          (error 'stream-append "non-stream argument"))
        (else (stream-append strms))))

(define (stream-concat strms)
  (define stream-concat
    (stream-lambda (strms)
      (cond ((stream-null? strms) stream-null)
            ((not (stream? (stream-car strms)))
              (error 'stream-concat "non-stream object in input stream"))
            ((stream-null? (stream-car strms))
              (stream-concat (stream-cdr strms)))
            (else (stream-cons
                    (stream-car (stream-car strms))
                    (stream-concat
                      (stream-cons (stream-cdr (stream-car strms)) (stream-cdr strms))))))))
  (if (not (stream? strms))
      (error 'stream-concat "non-stream argument")
      (stream-concat strms)))
```

```
(define stream-constant
  (stream-lambda objs
    (cond ((null? objs) stream-null)
          ((null? (cdr objs)) (stream-cons (car objs) (stream-constant (car objs))))
          (else (stream-cons (car objs)
                             (apply stream-constant (append (cdr objs) (list (car objs)))))))))

(define (stream-drop n strm)
  (define stream-drop
    (stream-lambda (n strm)
      (if (or (zero? n) (stream-null? strm))
          strm
          (stream-drop (- n 1) (stream-cdr strm)))))
  (cond ((not (integer? n)) (error 'stream-drop "non-integer argument"))
        ((negative? n) (error 'stream-drop "negative argument"))
        ((not (stream? strm)) (error 'stream-drop "non-stream argument"))
        (else (stream-drop n strm))))

(define (stream-drop-while pred? strm)
  (define stream-drop-while
    (stream-lambda (strm)
      (if (and (stream-pair? strm) (pred? (stream-car strm)))
          (stream-drop-while (stream-cdr strm))
          strm)))
  (cond ((not (procedure? pred?)) (error 'stream-drop-while "non-procedural argument"))
        ((not (stream? strm)) (error 'stream-drop-while "non-stream argument"))
        (else (stream-drop-while strm))))

(define (stream-filter pred? strm)
  (define stream-filter
    (stream-lambda (strm)
      (cond ((stream-null? strm) stream-null)
            ((pred? (stream-car strm))
              (stream-cons (stream-car strm) (stream-filter (stream-cdr strm))))
            (else (stream-filter (stream-cdr strm))))))
  (cond ((not (procedure? pred?)) (error 'stream-filter "non-procedural argument"))
        ((not (stream? strm)) (error 'stream-filter "non-stream argument"))
        (else (stream-filter strm))))

(define (stream-fold proc base strm)
  (cond ((not (procedure? proc)) (error 'stream-fold "non-procedural argument"))
        ((not (stream? strm)) (error 'stream-fold "non-stream argument"))
        (else (let loop ((base base) (strm strm))
                (if (stream-null? strm)
                    base
                    (loop (proc base (stream-car strm)) (stream-cdr strm)))))))

(define (stream-for-each proc . strms)
  (define (stream-for-each strms)
    (if (not (exists stream-null? strms))
        (begin (apply proc (map stream-car strms))
               (stream-for-each (map stream-cdr strms)))))
  (cond ((not (procedure? proc)) (error 'stream-for-each "non-procedural argument"))
        ((null? strms) (error 'stream-for-each "no stream arguments"))
        ((exists (lambda (x) (not (stream? x))) strms)
          (error 'stream-for-each "non-stream argument"))
        (else (stream-for-each strms))))

(define (stream-from first . step)
  (define stream-from
    (stream-lambda (first delta)
      (stream-cons first (stream-from (+ first delta) delta))))
  (let ((delta (if (null? step) 1 (car step))))
    (cond ((not (number? first)) (error 'stream-from "non-numeric starting number"))
          ((not (number? delta)) (error 'stream-from "non-numeric step size"))
          (else (stream-from first delta)))))

(define (stream-iterate proc base)
  (define stream-iterate
    (stream-lambda (base)
      (stream-cons base (stream-iterate (proc base)))))
  (if (not (procedure? proc))
      (error 'stream-iterate "non-procedural argument")
      (stream-iterate base)))
```

```
(define (stream-length strm)
  (if (not (stream? strm))
      (error 'stream-length "non-stream argument")
      (let loop ((len 0) (strm strm))
        (if (stream-null? strm)
            len
            (loop (+ len 1) (stream-cdr strm))))))

(define-syntax stream-let
  (syntax-rules ()
    ((stream-let tag ((name val) ...) body1 body2 ...)
     ((letrec ((tag (stream-lambda (name ...) body1 body2 ...))) tag) val ...))))

(define (stream-map proc . strms)
  (define stream-map
    (stream-lambda (strms)
      (if (exists stream-null? strms)
          stream-null
          (stream-cons (apply proc (map stream-car strms))
                       (stream-map (map stream-cdr strms))))))
  (cond ((not (procedure? proc)) (error 'stream-map "non-procedural argument"))
        ((null? strms) (error 'stream-map "no stream arguments"))
        ((exists (lambda (x) (not (stream? x))) strms)
          (error 'stream-map "non-stream argument"))
        (else (stream-map strms))))

(define-syntax stream-match
  (syntax-rules ()
    ((stream-match strm-expr clause ...)
      (let ((strm strm-expr))
        (cond
          ((not (stream? strm)) (error 'stream-match "non-stream argument"))
          ((stream-match-test strm clause) => car) ...
          (else (error 'stream-match "pattern failure"))))))

(define-syntax stream-match-test
  (syntax-rules ()
    ((stream-match-test strm (pattern fender expr))
      (stream-match-pattern strm pattern () (and fender (list expr))))
    ((stream-match-test strm (pattern expr))
      (stream-match-pattern strm pattern () (list expr)))))

(define-syntax stream-match-pattern
  (lambda (x)
    (define (wildcard? x)
      (and (identifier? x)
           (free-identifier=? x (syntax _))))
    (syntax-case x ()
      ((stream-match-pattern strm () (binding ...) body)
        (syntax (and (stream-null? strm) (let (binding ...) body))))
      ((stream-match-pattern strm (w? . rest) (binding ...) body)
        (wildcard? #'w?)
        (syntax (and (stream-pair? strm)
                     (let ((strm (stream-cdr strm)))
                       (stream-match-pattern strm rest (binding ...) body)))))
      ((stream-match-pattern strm (var . rest) (binding ...) body)
        (syntax (and (stream-pair? strm)
                     (let ((temp (stream-car strm)) (strm (stream-cdr strm)))
                       (stream-match-pattern strm rest ((var temp) binding ...) body)))))
      ((stream-match-pattern strm w? (binding ...) body)
        (wildcard? #'w?)
        (syntax (let (binding ...) body)))
      ((stream-match-pattern strm var (binding ...) body)
        (syntax (let ((var strm) binding ...) body))))))

(define-syntax stream-of
  (syntax-rules ()
    ((_ expr rest ...)
      (stream-of-aux expr stream-null rest ...))))

(define-syntax stream-of-aux
  (syntax-rules (in is)
    ((stream-of-aux expr base)
      (stream-cons expr base))
```

```
    ((stream-of-aux expr base (var in stream) rest ...)
      (stream-let loop ((strm stream))
        (if (stream-null? strm)
            base
            (let ((var (stream-car strm)))
              (stream-of-aux expr (loop (stream-cdr strm)) rest ...)))))
    ((stream-of-aux expr base (var is exp) rest ...)
      (let ((var exp)) (stream-of-aux expr base rest ...)))
    ((stream-of-aux expr base pred? rest ...)
      (if pred? (stream-of-aux expr base rest ...) base))))

(define (stream-range first past . step)
  (define stream-range
    (stream-lambda (first past delta lt?)
      (if (lt? first past)
          (stream-cons first (stream-range (+ first delta) past delta lt?))
          stream-null)))
  (cond ((not (number? first)) (error 'stream-range "non-numeric starting number"))
        ((not (number? past)) (error 'stream-range "non-numeric ending number"))
        (else (let ((delta (cond ((pair? step) (car step)) ((< first past) 1) (else -1))))
                (if (not (number? delta))
                    (error 'stream-range "non-numeric step size")
                    (let ((lt? (if (< 0 delta) < >)))
                      (stream-range first past delta lt?)))))))

(define (stream-ref strm n)
  (cond ((not (stream? strm)) (error 'stream-ref "non-stream argument"))
        ((not (integer? n)) (error 'stream-ref "non-integer argument"))
        ((negative? n) (error 'stream-ref "negative argument"))
        (else (let loop ((strm strm) (n n))
                (cond ((stream-null? strm) (error 'stream-ref "beyond end of stream"))
                      ((zero? n) (stream-car strm))
                      (else (loop (stream-cdr strm) (- n 1))))))))

(define (stream-reverse strm)
  (define stream-reverse
    (stream-lambda (strm rev)
      (if (stream-null? strm)
          rev
          (stream-reverse (stream-cdr strm) (stream-cons (stream-car strm) rev)))))
  (if (not (stream? strm))
      (error 'stream-reverse "non-stream argument")
      (stream-reverse strm stream-null)))

(define (stream-scan proc base strm)
  (define stream-scan
    (stream-lambda (base strm)
      (if (stream-null? strm)
          (stream base)
          (stream-cons base (stream-scan (proc base (stream-car strm)) (stream-cdr strm))))))
  (cond ((not (procedure? proc)) (error 'stream-scan "non-procedural argument"))
        ((not (stream? strm)) (error 'stream-scan "non-stream argument"))
        (else (stream-scan base strm))))

(define (stream-take n strm)
  (define stream-take
    (stream-lambda (n strm)
      (if (or (stream-null? strm) (zero? n))
          stream-null
          (stream-cons (stream-car strm) (stream-take (- n 1) (stream-cdr strm))))))
  (cond ((not (stream? strm)) (error 'stream-take "non-stream argument"))
        ((not (integer? n)) (error 'stream-take "non-integer argument"))
        ((negative? n) (error 'stream-take "negative argument"))
        (else (stream-take n strm))))

(define (stream-take-while pred? strm)
  (define stream-take-while
    (stream-lambda (strm)
      (cond ((stream-null? strm) stream-null)
            ((pred? (stream-car strm))
              (stream-cons (stream-car strm) (stream-take-while (stream-cdr strm))))
            (else stream-null))))
  (cond ((not (stream? strm)) (error 'stream-take-while "non-stream argument"))
        ((not (procedure? pred?)) (error 'stream-take-while "non-procedural argument"))
        (else (stream-take-while strm))))
```

```
(define (stream-unfold mapper pred? generator base)
  (define stream-unfold
    (stream-lambda (base)
      (if (pred? base)
          (stream-cons (mapper base) (stream-unfold (generator base)))
          stream-null)))
  (cond ((not (procedure? mapper)) (error 'stream-unfold "non-procedural mapper"))
        ((not (procedure? pred?)) (error 'stream-unfold "non-procedural pred?"))
        ((not (procedure? generator)) (error 'stream-unfold "non-procedural generator"))
        (else (stream-unfold base))))

(define (stream-unfolds gen seed)
  (define (len-values gen seed)
    (call-with-values
      (lambda () (gen seed))
      (lambda vs (- (length vs) 1))))
  (define unfold-result-stream
    (stream-lambda (gen seed)
      (call-with-values
        (lambda () (gen seed))
        (lambda (next . results)
          (stream-cons results (unfold-result-stream gen next))))))
  (define result-stream->output-stream
    (stream-lambda (result-stream i)
      (let ((result (list-ref (stream-car result-stream) (- i 1))))
        (cond ((pair? result)
                (stream-cons
                  (car result)
                  (result-stream->output-stream (stream-cdr result-stream) i)))
              ((not result)
                (result-stream->output-stream (stream-cdr result-stream) i))
              ((null? result) stream-null)
              (else (error 'stream-unfolds "can't happen"))))))
  (define (result-stream->output-streams result-stream)
    (let loop ((i (len-values gen seed)) (outputs '()))
      (if (zero? i)
          (apply values outputs)
          (loop (- i 1) (cons (result-stream->output-stream result-stream i) outputs)))))
  (if (not (procedure? gen))
      (error 'stream-unfolds "non-procedural argument")
      (result-stream->output-streams (unfold-result-stream gen seed))))

(define (stream-zip . strms)
  (define stream-zip
    (stream-lambda (strms)
      (if (exists stream-null? strms)
          stream-null
          (stream-cons (map stream-car strms) (stream-zip (map stream-cdr strms))))))
  (cond ((null? strms) (error 'stream-zip "no stream arguments"))
        ((exists (lambda (x) (not (stream? x))) strms)
          (error 'stream-zip "non-stream argument"))
        (else (stream-zip strms))))
```

Appendix 3: Implementation of `(streams)`

```
(library (streams)

  (export stream-null stream-cons stream? stream-null? stream-pair? stream-car
          stream-cdr stream-lambda define-stream list->stream port->stream stream
          stream->list stream-append stream-concat stream-constant stream-drop
          stream-drop-while stream-filter stream-fold stream-for-each stream-from
          stream-iterate stream-length stream-let stream-map stream-match _
          stream-of stream-range stream-ref stream-reverse stream-scan stream-take
          stream-take-while stream-unfold stream-unfolds stream-zip)

  (import (streams primitive) (streams derived)))
```