## Streams

## Announcements

## Efficient Sequence Processing

## Sequence Operations

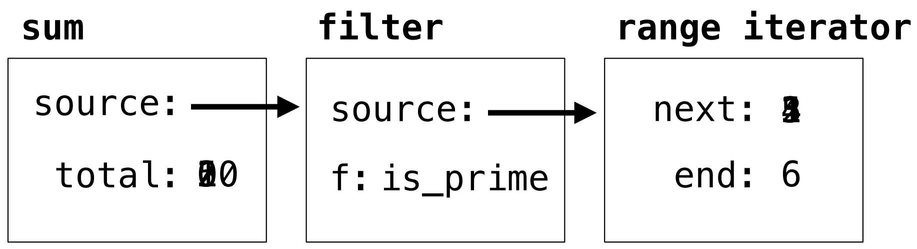
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
def sum_primes(a, b):
    total = 0
    x = a
    while x < b:
        if is_prime(x):
            total = total + x
        x = x + 1
    return total
```

```
def sum_primes(a, b):
    return sum(filter(is_prime, range(a, b)))

sum_primes(1, 6)
```



Space: $\Theta(1)$ $\Theta(1)$

(Demo)

## Streams

## Streams are Lazy Scheme Lists

A stream is a list, but the rest of the list is computed only when needed:

```
(car (cons 1 2)) -> 1                (car        (cons-stream 1 2)) -> 1

(cdr (cons 1 2)) -> 2                (cdr-stream (cons-stream 1 2)) -> 2

(cons 1 (cons 2 nil))                (cons-stream 1 (cons-stream 2 nil))
```

Errors only occur when expressions are evaluated:

```
(cons 1 (/ 1 0))       -> ERROR    (cons-stream 1 (/ 1 0))    -> (1 . #[promise (not forced)])

(car (cons 1 (/ 1 0))) -> ERROR    (car        (cons-stream 1 (/ 1 0))) -> 1

(cdr (cons 1 (/ 1 0))) -> ERROR    (cdr-stream (cons-stream 1 (/ 1 0))) -> ERROR
```

(Demo)

## Stream Ranges are Implicit

A stream can give on–demand access to each element in order

```
(define (range-stream a b)
  (if (>= a b)
      nil
      (cons-stream a (range-stream (+ a 1) b))))

(define lots (range-stream 1 10000000000000000000))

scm> (car lots)
1
scm> (car (cdr-stream lots))
2
scm> (car (cdr-stream (cdr-stream lots)))
3
```

## Infinite Streams

## Integer Stream

An integer stream is a stream of consecutive integers

The rest of the stream is not yet computed when the stream is created

```
(define (int-stream start)
  (cons-stream start (int-stream (+ start 1))))
```

(Demo)

## Stream Processing

(Demo)

## Recursively Defined Streams

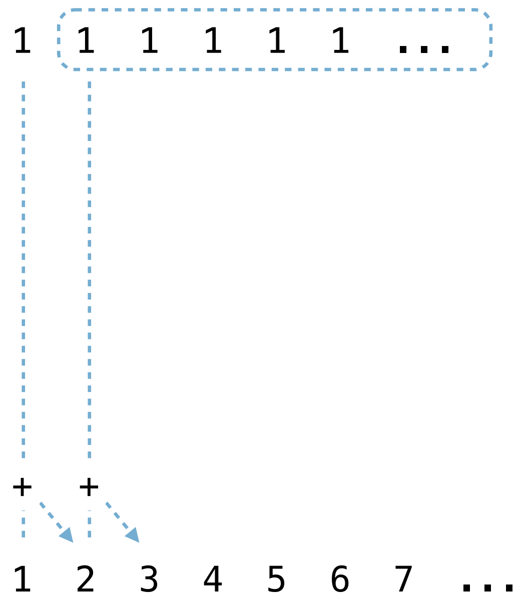
The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr

```
(define (add-streams s t)
  (cons-stream (+ (car s) (car t))
               (add-streams (cdr-stream s)
                            (cdr-stream t))))
```

```
(define ints (cons-stream 1 (add-streams ones ints)))
```



## Example: Repeats

```
(define a (cons-stream 1 (cons-stream 2 (cons-stream 3 a))))

(define (f s) (cons-stream (car s)
                           (cons-stream (car s)
                                        (f (cdr-stream s)))))

(define (g s) (cons-stream (car s)
                           (f (g (cdr-stream s)))))
```

What's (prefix a 8)?       ( 1 2 3 1 2 3 1 2 )

What's (prefix (f a) 8)?   ( 1 1 2 2 3 3 1 1 )

What's (prefix (g a) 8)?   ( 1 2 2 3 3 3 3 1 )

## Higher-Order Stream Functions

## Higher-Order Functions on Streams

Implementations are identical,
but change cons to cons–stream
and change cdr to cdr–stream

```
(define (map-stream f s)
  (if (null? s)
      nil
      (cons-stream (f (car s))
            (map-stream f
                (cdr-stream s)))))

(define (filter-stream f s)
  (if (null? s)
      nil
      (if (f (car s))
          (cons-stream (car s)
               (filter-stream f (cdr-stream s)))
          (filter-stream f (cdr-stream s)))))

(define (reduce-stream f s start)
  (if (null? s)
      start
      (reduce-stream f
              (cdr-stream s)
              (f start (car s)))))
```

:%s/\v(map|filter|reduce|cdr|cons)/\1-stream/g

## A Stream of Primes

For any prime k, any larger prime must not be divisible by k.

The stream of integers not divisible by any k <= n is:

- The stream of integers not divisible by any k < n
- Filtered to remove any element divisible by n

This recurrence is called the Sieve of Eratosthenes

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

(Demo)

## Promises

## Implementing Streams with Delay and Force

A promise is an expression, along with an environment in which to evaluate it

Delaying an expression creates a promise to evaluate it later in the current environment

Forcing a promise returns its value in the environment in which it was defined

```
scm> (define promise (let ((x 2)) (delay     (+ x 1)) ))
     (define promise (let ((x 2)) (lambda () (+ x 1)) ))

scm> (define x 5)

scm> (force promise)
3
```

```
(define-macro (delay expr)    `(lambda () ,expr))
(define       (force promise) (promise))
```

A stream is a list, but the rest of the list is computed only when **forced:**

```
scm> (define ones (cons-stream 1 ones))
(1 . #[promise (not forced)])
(1 . (lambda () ones))
```

```
(define-macro (cons-stream a b) `(cons ,a (delay ,b)))
(define       (cdr-stream s)     (force (cdr s)))
```