Streams

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)

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

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

Space: Constant
Also Constant

Errors only occur when expressions are evaluated:

```scheme
(car range-stream 1 nil) → 1
(car (cdr-stream (range-stream 1 0))) → 1
(car (cdr-stream (range-stream 1 nil))) → ERROR
```

Infinite Streams

As an integer stream is a stream of consecutive integers

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

```scheme
(define int-stream 1)
```

Streams

Streams are Lazy Scheme Lists

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

```scheme
(car lots) → 1
(car (cdr-stream lots)) → 2
(car (cdr-stream (cdr-stream (cdr-stream lots)))) → ERROR
```

Infinite Streams

As an integer stream is a stream of consecutive integers

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

```scheme
(define int-stream start)
```

Announcements

Efficient Sequence Processing

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

```scheme
(define int-stream start)
```

Infinite Streams

As an integer stream is a stream of consecutive integers

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

```scheme
(define int-stream start)
```
Stream Processing

Recursively Defined Streams

The rest of a constant stream is the constant stream:

(define ones (cons-stream 1 ones))

(ones)

1 1 1 1 1 1 ...

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))))

(ints)

2 3 4 5 6 7 ...

1 + 2

Higher-Order Stream Functions

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

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

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

(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 (car s) start))))

Higher-Order Functions on Streams

Implementations are identical, but change cons to cons-stream and change cdr to cdr-stream:

(map)

(filter)

(reduce)

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, 5, 7, 11, 13

(Sie)