Streams

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)

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

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

Space:

Filter:

<table>
<thead>
<tr>
<th>Source</th>
<th>Filter</th>
<th>Range Iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>total: 1</td>
<td>f: is_prime</td>
<td>next: 6</td>
</tr>
</tbody>
</table>

Streams

Streams are Lazy Scheme Lists

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

```scheme
(define (range-stream a b)
  (if (>= a b)
      nil
      (cons-stream a
                   (range-stream (+ a 1) b)))))
(define lots (range-stream 1 10000000000000000000))
```

Errors only occur when expressions are evaluated:

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

Stream Ranges are Implicit

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

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

```scheme
(define lots (range 0 10000000000000000000))
```

Infinite Streams

<table>
<thead>
<tr>
<th>Source</th>
<th>Filter</th>
<th>Range Iterator</th>
</tr>
</thead>
<tbody>
<tr>
<td>total: 1</td>
<td>f: is_prime</td>
<td>next: 6</td>
</tr>
</tbody>
</table>

(Demo)
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

\[ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, \ldots \]
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:

```
(define promise (let ((x 2)) (lambda () (+ x 3))))
```

```
(define x 5)
```

```
(define (force promise) (promise))
```

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

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

```
(1 . (lambda () ones))
```

```
(1 . #[(promise (not forced))])
```