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 += x
        x += 1
    return total
```

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

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)
```

Infinite Streams

Streams

Stream Ranges are Implicit

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

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

```scheme
(define lots (range-stream 1 10000000000000000000))
```

```
1
2
3
```
Stream Processing

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 2)))))

Stream Processing

Recurseivly 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 s (cons-stream 1 (cons-stream 2 (cons-stream 3 s))))

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

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

Higher-Order Functions on Streams

(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))))))

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

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 \(k \leq n\) 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.

\[
2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
\]
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)

(force promise) 3
```

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

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

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

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```

```
(define x promise (not forced))
```