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)

```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: \( \Theta(1) \)  \( \Theta(1) \)

(Demo)
Streams
Streams are Lazy Scheme Lists

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

\[
\begin{align*}
    \text{car (cons 1 2)} & \rightarrow 1 & \text{car (cons-stream 1 2)} & \rightarrow 1 \\
    \text{cdr (cons 1 2)} & \rightarrow 2 & \text{cdr-stream (cons-stream 1 2)} & \rightarrow 2 \\
    \text{cons 1 (cons 2 nil)} & \rightarrow \text{cons-stream 1 (cons-stream 2 nil)}
\end{align*}
\]

Errors only occur when expressions are evaluated:

\[
\begin{align*}
    \text{cons 1 (/ 1 0)} & \rightarrow \text{ERROR} & \text{cons-stream 1 (/ 1 0)} & \rightarrow (1 . \#\text{[promise (not forced)]}) \\
    \text{car (cons 1 (/ 1 0))} & \rightarrow \text{ERROR} & \text{car (cons-stream 1 (/ 1 0))} & \rightarrow 1 \\
    \text{cdr (cons 1 (/ 1 0))} & \rightarrow \text{ERROR} & \text{cdr-stream (cons-stream 1 (/ 1 0))} & \rightarrow \text{ERROR}
\end{align*}
\]

(Demo)
Stream Ranges are Implicit

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

\[
\text{(define (range-stream a b)}
\text{  (if (>= a b)}
\text{    nil}
\text{    (cons-stream a (range-stream (+ a 1) b))))}
\]

\[
\text{(define lots (range-stream 1 100000000000000000000))}
\]

\[
\text{scm> (car lots)}
\text{1}
\text{scm> (car (cdr-stream lots))}
\text{2}
\text{scm> (car (cdr-stream (cdr-stream lots))}  \text{3}
\]
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

\[
\textbf{(define} \ \textit{ones} \ \textbf{(cons-stream} \ 1 \ \textit{ones}))
\]

Combine two streams by separating each into car and cdr

\[
\textbf{(define} \ \textbf{(add-streams} \ s \ t) \\
\quad \textbf{(cons-stream} \ (+ \ \textbf{(car} \ s)) \ \textbf{(car} \ t)) \\
\quad \textbf{(add-streams} \ \textbf{(cdr-stream} \ s) \\
\quad \textbf{(cdr-stream} \ t)))
\]

\[
\textbf{(define} \ \textit{ints} \ \textbf{(cons-stream} \ 1 \ \textbf{(add-streams} \ \textit{ones} \ \textit{ints})))
\]
Example: Repeats

```scheme
(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)?   ( __ __ __ __ __ __ __ __ __ __ __ __ __ __ )
What's (prefix (f a) 8)? ( __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ )
What's (prefix (g a) 8)? ( __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ _ _ _ _ _ _ )
Higher-Order Stream Functions
Higher-Order Functions on Streams

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

\[
\text{(define (map-stream f s)}
\begin{align*}
&\quad \text{if (null? s)} \\
&\quad \quad \text{nil} \\
&\quad \quad \text{(cons-stream (f (car s))} \\
&\quad \quad \quad \text{(map-stream f} \\
&\quad \quad \quad \quad \text{(cdr-stream s))))
\end{align*}
\]

\[
\text{(define (filter-stream f s)}
\begin{align*}
&\quad \text{if (null? s)} \\
&\quad \quad \text{nil} \\
&\quad \quad \text{(if (f (car s))} \\
&\quad \quad \quad \text{(cons-stream (car s)} \\
&\quad \quad \quad \quad \text{(filter-stream (edm f) (cdr-stream s))} \\
&\quad \quad \quad \quad \text{(filter-stream (edm f) (cdr-stream s)))})
\end{align*}
\]

\[
\text{(define (reduce-stream f start)}
\begin{align*}
&\quad \text{if (null? s)} \\
&\quad \quad \text{start} \\
&\quad \quad \text{(reduce-stream f} \\
&\quad \quad \quad \text{(cdr-stream s)} \\
&\quad \quad \quad \quad \text{(f start (car s)))})
\end{align*}
\]
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
(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

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

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

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