Announcements
Efficient Sequence Processing
Sequence Operations
Sequence Operations

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

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

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

Space: $\Theta(1)$
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

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

sum_primes(1, 6)
```

Space: \( \Theta(1) \)
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

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

sum_primes(1, 6)
```

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

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

Space: \( \Theta(1) \)
Sequence Operations

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

Example: Sum all primes in an interval from \texttt{a} (inclusive) to \texttt{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)))
```

```
sum_primes(1, 6)
```

Space: $\Theta(1)$
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

sum_primes(1, 6)
```

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

```
sum_primes(1, 6)
```

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

```
sum_primes(1, 6)
```

\[\text{Space: } \Theta(1)\]
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)))
```

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

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

(Demo)
Streams
Streams are Lazy Scheme Lists
Streams are Lazy Scheme Lists

A stream is a list, but the rest of the list is computed only when needed:
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
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)) \rightarrow 1

(cdr (cons 1 2)) \rightarrow 2
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)) \rightarrow 1
(cdr (cons 1 2)) \rightarrow 2
(cons 1 (cons 2 nil))
Streams are Lazy Scheme Lists

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

\[
\begin{align*}
(car \ (cons \ 1 \ 2)) & \rightarrow 1 \\
(cdr \ (cons \ 1 \ 2)) & \rightarrow 2 \\
(\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{nil})) & \\
\end{align*}
\]
Streams are Lazy Scheme Lists

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

\[
\begin{align*}
(c\text{ar} \ (c\text{ons} \ 1 \ 2)) & \rightarrow 1 & (c\text{ar} \ (c\text{ons-}\text{stream} \ 1 \ 2)) & \rightarrow 1 \\
(c\text{dr} \ (c\text{ons} \ 1 \ 2)) & \rightarrow 2 & (c\text{dr-}\text{stream} \ (c\text{ons-}\text{stream} \ 1 \ 2)) & \rightarrow 2 \\
(c\text{ons} \ 1 \ (c\text{ons} \ 2 \ n\text{il})) &
\end{align*}
\]
Streams are Lazy Scheme Lists

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

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

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

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

Errors only occur when expressions are evaluated:
Streams are Lazy Scheme Lists

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

\[(\text{car} \ \text{cons} \ 1 \ 2) \rightarrow 1\] \hspace{1cm} \[(\text{car} \ \text{cons-stream} \ 1 \ 2) \rightarrow 1\]

\[(\text{cdr} \ \text{cons} \ 1 \ 2) \rightarrow 2\] \hspace{1cm} \[(\text{cdr-stream} \ \text{cons-stream} \ 1 \ 2) \rightarrow 2\]

\[(\text{cons} \ 1 \ \text{cons} \ 2 \ \text{nil})\] \hspace{1cm} \[(\text{cons-stream} \ 1 \ \text{cons-stream} \ 2 \ \text{nil})\]

Errors only occur when expressions are evaluated:

\[(\text{cons} \ 1 \ (/ \ 1 \ 0)) \rightarrow \text{ERROR}\]
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)) \rightarrow 1  \quad (car (cons-stream 1 2)) \rightarrow 1
\space
(cdr (cons 1 2)) \rightarrow 2  \quad (cdr-stream (cons-stream 1 2)) \rightarrow 2
\space
(cons 1 (cons 2 nil))  \quad (cons-stream 1 (cons-stream 2 nil))

Errors only occur when expressions are evaluated:

(cons 1 (/ 1 0)) \rightarrow ERROR
\space
(car (cons 1 (/ 1 0))) \rightarrow ERROR
Streams are Lazy Scheme Lists

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

\[(\text{car } (\text{cons } 1 \ 2)) \rightarrow 1\] \quad \(\text{car } (\text{cons-stream } 1 \ 2) \rightarrow 1\)

\[(\text{cdr } (\text{cons } 1 \ 2)) \rightarrow 2\] \quad \(\text{cdr-stream } (\text{cons-stream } 1 \ 2) \rightarrow 2\)

\[(\text{cons } 1 \ (\text{cons } 2 \ \text{nil}))\] \quad \(\text{cons-stream } 1 \ (\text{cons-stream } 2 \ \text{nil})\)

Errors only occur when expressions are evaluated:

\[(\text{cons } 1 \ (/ \ 1 \ 0)) \rightarrow \text{ERROR}\]

\[(\text{car } (\text{cons } 1 \ (/ \ 1 \ 0))) \rightarrow \text{ERROR}\]

\[(\text{cdr } (\text{cons } 1 \ (/ \ 1 \ 0))) \rightarrow \text{ERROR}\]
Streams are Lazy Scheme Lists

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

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

Errors only occur when expressions are evaluated:

\[
\begin{align*}
&\text{(cons 1 (/ 1 0))} \rightarrow \text{ERROR} & \text{(cons-stream 1 (/ 1 0))} \rightarrow (1 . #[delayed]) \\
&\text{(car (cons 1 (/ 1 0)))} \rightarrow \text{ERROR} \\
&\text{(cdr (cons 1 (/ 1 0)))} \rightarrow \text{ERROR}
\end{align*}
\]
Streams are Lazy Scheme Lists

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

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

Errors only occur when expressions are evaluated:

\[
\begin{align*}
\text{(cons 1 (/ 1 0))} & \rightarrow \text{ERROR} \\
\text{(car (cons 1 (/ 1 0)))} & \rightarrow \text{ERROR} \\
\text{(cdr (cons 1 (/ 1 0)))} & \rightarrow \text{ERROR}
\end{align*}
\]
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
(cdr (cons 1 2)) → 2
(cons 1 (cons 2 nil))

(car (cons-stream 1 2)) → 1
(cdr-stream (cons-stream 1 2)) → 2
(cons-stream 1 (cons-stream 2 nil))

Errors only occur when expressions are evaluated:

(cons 1 (/ 1 0)) → ERROR
(car (cons 1 (/ 1 0))) → ERROR
(cdr (cons 1 (/ 1 0))) → ERROR
(cons-stream 1 (/ 1 0)) → (1 . #[delayed])
(car (cons-stream 1 (/ 1 0))) → 1
(cdr-stream (cons-stream 1 (/ 1 0))) → ERROR
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 . #[delayed])
(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
Stream Ranges are Implicit

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

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

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

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

\[
(\text{define} \ \text{lots} \ (\text{range-stream} \ 1 \ 10000000000000000000))
\]

\[
\text{scm}\> (\text{car} \ \text{lots}) \\\n1
\]
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
```
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))))

(define lots (range-stream 1 10000000000000000000))

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

An integer stream is a stream of consecutive integers
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.
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))))
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))))
```
Stream Processing

(Demo)
Recursively Defined Streams
Recursively Defined Streams

The rest of a constant stream is the constant stream.
Recursively Defined Streams

The rest of a constant stream is the constant stream

\[(\text{define ones (cons-stream 1 ones)})\]
Recursively Defined Streams

The rest of a constant stream is the constant stream

```
(define ones (cons-stream 1 ones)) 1 1 1 1 1 1 ...
```
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

1 1 1 1 1 1 ...
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr
Recursively Defined Streams

The rest of a constant stream is the constant stream

\[
\text{define ones (cons-stream 1 ones)} \quad 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ ...
\]

Combine two streams by separating each into car and cdr

\[
\text{define (add-streams s t)}
\]
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

1 1 1 1 1 1 ...

Combine two streams by separating each into car and cdr

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

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

1 1 1 1 1 1 ...

Combine two streams by separating each into car and cdr

\[
\text{(define (add-streams s t)
  (cons-stream (+ (car s) (car t)))}
\]
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr

\[
(\text{define (add-streams } s \ t) \\
(\text{cons-stream} (+ (\text{car } s) (\text{car } t)) \\
(\text{add-streams} (\text{cdr-stream } s) \\
(\text{add-streams} (\text{cdr-stream } t))))))
\]
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

1 1 1 1 1 1 ...

Combine two streams by separating each into car and cdr

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

\[(\text{define} \ \text{ints} \ (\text{cons-stream} \ 1 \ (\text{add-streams} \ \text{ones} \ \text{ints})))\]
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr

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

\[
\text{(define ints (cons-stream 1 (add-streams ones ints))}
\]
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr

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

\[
\text{(define ints (\text{cons-stream} 1 (add-streams ones ints)))}
\]
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)))
```

1 1 1 1 1 1 1 ...
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr

\[
\text{(define (add-streams s t))}
\]

\[
\text{(cons-stream (+ (car s) (car t))}
\]

\[
\text{(add-streams (cdr-stream s)}
\]

\[
\text{(cdr-stream t))})
\]

\[
\text{(define ints (cons-stream 1 (add-streams ones ints)))}
\]
Recursively Defined Streams

The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr

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

\[
(\text{define ints} \ (\text{cons-stream} \ 1 \ (\text{add-streams} \ \text{ones} \ \text{ints})))
\]
Higher-Order Stream Functions
Higher-Order Functions on Streams

Implementations are identical, but change cons to cons-stream and change cdr to cdr-stream.
Higher-Order Functions on Streams

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

```
(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 start (car s))))
```
Higher-Order Functions on Streams

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

```scheme
(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 start (car s)))))
```
Higher-Order Functions on Streams

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

```scheme
(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))))
```

13
A Stream of Primes
A Stream of Primes

The stream of integers not divisible by any $k \leq n$ is:
A Stream of Primes

The stream of integers not divisible by any $k \leq n$ is:

- The stream of integers not divisible by any $k < n$
A Stream of Primes

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 \( n \)
A Stream of Primes

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
A Stream of Primes

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
A Stream of Primes

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
A Stream of Primes

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
A Stream of Primes

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

\[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\]
A Stream of Primes

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
A Stream of Primes

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
A Stream of Primes

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