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

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

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       filter       range iterator
source:    source:     next: 1  
total: 0  f: is_prime  end: 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 `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)))
```

```
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)))
```

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)))
```

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)) → 1

(cdr (cons 1 2)) → 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:

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

```
(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))
```
Streams are Lazy Scheme Lists

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

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

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

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

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:

(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
Streams are Lazy Scheme Lists

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

\[(\text{car} \ (\text{cons} \ 1 \ 2)) \rightarrow 1\]
\[(\text{cdr} \ (\text{cons} \ 1 \ 2)) \rightarrow 2\]
\[(\text{cons} \ 1 \ (\text{cons} \ 2 \ \text{nil}))\]

Errors only occur when expressions are evaluated:

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

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

(car (cons-stream 1 2)) \rightarrow 1
(cdr-stream (cons-stream 1 2)) \rightarrow 2
(cons-stream 1 (cons-stream 2 nil))

Errors only occur when expressions are evaluated:

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

(cons-stream 1 (/ 1 0)) \rightarrow (1 . #[promise (not forced)])
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:

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

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

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

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

Errors only occur when expressions are evaluated:

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

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

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

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

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

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

Errors only occur when expressions are evaluated:

\[(\text{cons} \ 1 \ (/ \ 1 \ 0)) \to \text{ERROR}\]  \[(\text{cons-stream} \ 1 \ (/ \ 1 \ 0)) \to (1 . \#\{\text{promise (not forced)}\})\]

\[(\text{car} \ (\text{cons} \ 1 \ (/ \ 1 \ 0))) \to \text{ERROR}\]  \[(\text{car} \ (\text{cons-stream} \ 1 \ (/ \ 1 \ 0))) \to 1\]

\[(\text{cdr} \ (\text{cons} \ 1 \ (/ \ 1 \ 0))) \to \text{ERROR}\]  \[(\text{cdr-stream} \ (\text{cons-stream} \ 1 \ (/ \ 1 \ 0))) \to \text{ERROR}\]  

(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

```
(define (range-stream a b)
  (if (>= a b)
      nil
      (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

```
(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
```
Stream Ranges are Implicit

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

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

\[
(\text{define lots (range-stream } 1 \ 10000000000000000000000000000000))
\]

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

\[
\text{(define (range-stream a b)}
\begin{align*}
& \quad \text{(if (>= a b)} \\
& \quad \quad \text{nil} \\
& \quad \quad (\text{cons-stream a (range-stream (+ a 1) b)})
\end{align*}
\]

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

```
scm> (car lots)
1
scm> (car (cdr-stream lots))
2
scm> (car (cdr-stream (cdr-stream lots)))
3
```
Infinite Streams
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.

\[
\text{(define (int-stream start)}
\text{(cons-stream start (int-stream (+ start 1)))))}
\]
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.

\[
\text{(define (int-stream start)} \\
\quad \text{(cons-stream start (int-stream (+ start 1))))}
\]
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

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

The rest of a constant stream is the constant stream

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

\[
\text{ones} = \text{cons-stream} \ 1 \ \text{ones}
\]

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

The rest of a constant stream is the constant stream

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

Combine two streams by separating each into car and cdr

```
(define (add-streams s t)
```

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

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

\[
\]

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

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

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

1 1 1 1 1 1 ...

Combine two streams by separating each into car and cdr

```scheme
(define (add-streams s t)
  (cons-stream (+ (car s) (car t))
    (add-streams (cdr-stream s) (cdr-stream t))))
```

```scheme
(define ints (cons-stream 1 (add-streams ones ints)))
```

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

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

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

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

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

```
(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)))
```

\[
\begin{array}{c}
1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & \ldots
\end{array}
\]
Example: Repeats
Example: Repeats

\[(\text{define} \ a \ (\text{cons-stream} \ 1 \ (\text{cons-stream} \ 2 \ (\text{cons-stream} \ 3 \ a))))\]
Example: Repeats

(\text{define\ a\ (cons-stream\ 1\ (cons-stream\ 2\ (cons-stream\ 3\ a))))}

What's (prefix a 8)? ( __ __ __ __ __ __ __ __ __ )
Example: Repeats

\[
\text{(define } a \text{ (cons-stream 1 (cons-stream 2 (cons-stream 3 a))))}
\]

\[
\text{(define } (f \ s) \text{ (cons-stream (car } s) \text{ (cons-stream (car } s) \text{ (f (cdr-stream } s))))}
\]

What's (prefix a 8)?

\[
( \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_)
\]
Example: Repeats

\[
(\text{define } a \ (\text{cons-stream } 1 \ (\text{cons-stream } 2 \ (\text{cons-stream } 3 \ a))))
\]

\[
(\text{define } (f s) \ (\text{cons-stream } (\text{car } s) \\
\quad \ (\text{cons-stream } (\text{car } s) \\
\quad \quad \ (f \ (\text{cdr-stream } s))))))
\]

What's (prefix a 8)? \hspace{1cm} ( _ _ _ _ _ _ _ _ _ _ )

What's (prefix (f a) 8)? \hspace{1cm} ( _ _ _ _ _ _ _ _ _ _ )
Example: Repeats

\[
\text{(define } a \ (\text{cons-stream} \ 1 \ (\text{cons-stream} \ 2 \ (\text{cons-stream} \ 3 \ a))))
\]

\[
\text{(define } f \ s \ (\text{cons-stream} \ \text{car} \ s)
\ \text{(cons-stream} \ \text{car} \ s)
\ \text{(f} \ \text{cdr-stream} \ s)))))
\]

\[
\text{(define } g \ s \ (\text{cons-stream} \ \text{car} \ s)
\ \text{f} \ \text{(g} \ \text{cdr-stream} \ s)))))
\]

What's (prefix a 8)? ( __ __ __ __ __ __ __ __ __ __)

What's (prefix (f a) 8)? ( __ __ __ __ __ __ __ __ __ __)
Example: Repeats

(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)? ( __ __ __ __ __ __ __ )
Example: Repeats

\[
\text{(define } a \text{ (cons-stream 1 (cons-stream 2 (cons-stream 3 a))))}
\]

\[
\text{(define } f s \text{ (cons-stream (car s)}
\text{ (cons-stream (car s)}
\text{ (f (cdr-stream s)))))
\]

\[
\text{(define } g s \text{ (cons-stream (car s)}
\text{ (f (g (cdr-stream s)))))
\]

What's (prefix a 8)? ( _ _ _ _ _ _ _ _ _ _ )

What's (prefix (f a) 8)? ( _ _ _ _ _ _ _ _ _ _ )

What's (prefix (g a) 8)? ( _ _ _ _ _ _ _ _ _ _ )
Example: Repeats

```
(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)? ( __ __ __ __ __ __ __ __)
Example: Repeats

\[
\text{define } a \ (\text{cons-stream} \ 1 \ (\text{cons-stream} \ 2 \ (\text{cons-stream} \ 3 \ a)))
\]

\[
\text{define } (f \ s) \ (\text{cons-stream} \ (\text{car} \ s) \\
\quad (\text{cons-stream} \ (\text{car} \ s) \\
\quad (\text{f} \ (\text{cdr-stream} \ s))))
\]

\[
\text{define } (g \ s) \ (\text{cons-stream} \ (\text{car} \ s) \\
\quad (\text{f} \ (g \ (\text{cdr-stream} \ s))))
\]

What's \((\text{prefix } a \ 8)\)? \(\begin{array}{cccccccc}
1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 \\
\end{array}\)

What's \((\text{prefix } (f \ a) \ 8)\)? \(\begin{array}{cccccccc}
1 & & & & & & & \\
\end{array}\)

What's \((\text{prefix } (g \ a) \ 8)\)? \(\begin{array}{cccccccc}
& & & & & & & \\
\end{array}\)
Example: Repeats

```
(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)?   ( 1 1 __ __ __ __ __ __ )

What's (prefix (g a) 8)?   ( __ __ __ __ __ __ __ __ )
```
Example: Repeats

\[
\begin{align*}
(\text{define } a & \ (\text{cons-stream } 1 \ (\text{cons-stream } 2 \ (\text{cons-stream } 3 \ a)))) \\
(\text{define } f \ s & \ (\text{cons-stream } (\text{car } s) \\
& \ (\text{cons-stream } (\text{car } s) \ (f \ (\text{cdr-stream } s)))))) \\
(\text{define } g \ s & \ (\text{cons-stream } (\text{car } s) \ (f \ (g \ (\text{cdr-stream } s))))) \\
\text{What's } (\text{prefix } a \ 8) & \ ? \ ( \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 ) \\
\text{What's } (\text{prefix } f \ a \ 8) & \ ? \ ( \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ 1 \ 1 \ 2 \ \_ \ \_ \ \_ \ \_ ) \\
\text{What's } (\text{prefix } g \ a \ 8) & \ ? \ ( \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ \_ \ 1 \ 2 \ 3 \ \_ \ \_ \ \_ )
\end{align*}
\]
Example: Repeats

(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)?  ( 1  __  __  2  __  __  __  __ )

What's (prefix (g a) 8)?  ( __  __  __  __  __  __  __  __ )
Example: Repeats

\[
\text{define } a \ (\text{cons-stream } 1 \ (\text{cons-stream } 2 \ (\text{cons-stream } 3 \ a)))
\]

\[
\text{define } (f \ s) \ (\text{cons-stream } \text{car } s)
\quad (\text{cons-stream } \text{car } s)
\quad (f \ (\text{cdr-stream } s)))
\]

\[
\text{define } (g \ s) \ (\text{cons-stream } \text{car } s)
\quad (f \ (g \ (\text{cdr-stream } s)))
\]

What's \((\text{prefix } a \ 8)\)? \(\text{ ( ___ ___ ___ ___ ___ ___ ___ ___ )}\)

What's \((\text{prefix } (f \ a) \ 8)\)? \(\text{ ( 1 1 2 2 3 3 1 1 )}\)

What's \((\text{prefix } (g \ a) \ 8)\)? \(\text{ ( ___ ___ ___ ___ ___ ___ ___ ___ )}\)
Example: Repeats

\[
(\text{define } a (\text{cons-stream } 1 (\text{cons-stream } 2 (\text{cons-stream } 3 a))))
\]

\[
(\text{define } (f \ s) (\text{cons-stream } (\text{car } s)\\
\phantom{(f \ s)} (\text{cons-stream } (\text{car } s)\\
\phantom{(f \ s)} (f (\text{cdr-stream } s)))))
\]

\[
(\text{define } (g \ s) (\text{cons-stream } (\text{car } s)\\
\phantom{(g \ s)} (f (g (\text{cdr-stream } s)))))
\]

What's (prefix a 8)? \hspace{1cm} ( 1 2 3 1 2 3 1 2 )

What's (prefix (f a) 8)? \hspace{1cm} ( 1 1 2 2 3 3 1 1 )

What's (prefix (g a) 8)? \hspace{1cm} ( 1 )
Example: Repeats

\(\text{define } a \ (\text{cons-stream } 1 \ (\text{cons-stream } 2 \ (\text{cons-stream } 3 \ a)))\)

\(\text{define } (f \ s) \ (\text{cons-stream } \ (\text{car } s)\)
\qquad \ (\text{cons-stream } \ (\text{car } s) \ (f \ (\text{cdr-stream } s)))\))

\(\text{define } (g \ s) \ (\text{cons-stream } \ (\text{car } s) \ (f \ (g \ (\text{cdr-stream } s))))\))

What's (prefix a 8)? \(\text{ ( } 1 \ 2 \ 3 \ 1 \ 2 \ 3 \ 1 \ 2 \text{ )}\)

What's (prefix (f a) 8)? \(\text{ ( } 1 \ 1 \ 2 \ 2 \ 3 \ 3 \ 1 \ 1 \text{ )}\)

What's (prefix (g a) 8)? \(\text{ ( } 1 \ 2 \ 2 \text{ )}\)
Example: Repeats

\[
\text{(define } a \text{ (cons-stream } 1 \text{ (cons-stream } 2 \text{ (cons-stream } 3 \text{ a))))}
\]

\[
\text{(define } f \text{ s} \text{ (cons-stream } \text{ (car } s) \text{ (cons-stream } \text{ (car } s) \text{ (f } \text{ (cdr-stream } s)))})
\]

\[
\text{(define } g \text{ s} \text{ (cons-stream } \text{ (car } s) \text{ (f } \text{ (g } \text{ (cdr-stream } s)))})
\]

What's (prefix a 8)?
\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 \\
\end{array}
\]

What's (prefix (f a) 8)?
\[
\begin{array}{cccccccc}
1 & 2 & 2 & 3 & 3 & 1 & 1 \\
\end{array}
\]

What's (prefix (g a) 8)?
\[
\begin{array}{cccccccc}
1 & 2 & 2 & 3 & 3 & \_ & \_ \\
\end{array}
\]
Example: Repeats

(\texttt{define \texttt{a} (cons-stream \texttt{1} (cons-stream \texttt{2} (cons-stream \texttt{3} \texttt{a})))})

(\texttt{define \texttt{(f s)} (cons-stream (car s))
\hspace{2em} (cons-stream (car s)
\hspace{4em} (f (cdr-stream s)))))

(\texttt{define \texttt{(g s)} (cons-stream (car s))
\hspace{2em} (f (g (cdr-stream s)))))

What's (prefix \texttt{a} 8)? \hspace{2em} ( __ 1 2 3 1 2 3 1 2 )

What's (prefix (f \texttt{a}) 8)? \hspace{2em} ( 1 1 2 2 3 3 1 1 )

What's (prefix (g \texttt{a}) 8)? \hspace{2em} ( 1 2 2 3 3 3 3 3 )
Example: Repeats

```
(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.
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

```
(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

```
(define (map-stream f s)
  (if (null? s)
      nil
      (cons-stream (f (car s))
                    (map-stream f
ddr-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
ddr-stream s)
      (f start (car s)))))
```
A Stream of Primes
A Stream of Primes

For any prime $k$, any larger prime must not be divisible by $k$. 
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:
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 $k < n$
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 $n$
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
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

$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13$
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

$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13$
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
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.

$$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13$$
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

$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13$
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

$2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13$
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

(Demo)
Promises
Implementing Streams with Delay and Force
Implementing Streams with Delay and Force

A promise is an expression, along with an environment in which to evaluate it.
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.
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.
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> (define promise (let ((x 2)) (delay (+ x 1))) )
```

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

```sml
(define promise (let ((x 2)) (delay (+ x 1))))
```

```sml
(force promise)
```

3
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> (define promise (let ((x 2)) (delay (+ x 1))) )

scm> (define x 5)
scm> (force promise)
3
```
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.

```sce
(define promise (let ((x 2)) (delay (+ x 1))))
```

```sce
(define x 5)
(define-macro (delay expr) `(lambda () ,expr))
(define (force promise) (promise))
```

```sce
(force promise)
```

3
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.

```scheme
(define promise (begin (define x 2) (delay (+ x 1))))
(define promise (begin (define x 2) (lambda () (+ x 1))))
(define x 5)
(force promise)
```

```scheme
(define-macro (delay expr) '(lambda () ,expr))
(define (force promise) promise)
```
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> (define promise (let ((x 2)) (delay (+ x 1))))
(scm> (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:
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> (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> (define ones (cons-stream 1 ones))
```
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> (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> (define ones (cons-stream 1 ones))
(1 . #[promise (not forced)])
```
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> (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> (define ones (cons-stream 1 ones))
(1 . #[promise (not forced)])
```

```
(define-macro (cons-stream a b) `(cons ,a (delay ,b)))
(define (cdr-stream s) (force (cdr s)))
```
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> (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> (define ones (cons-stream 1 ones))

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

scm> (define-macro (cons-stream a b) `(cons ,a (delay ,b)))

(1 . (lambda () ones))

(define-macro (cons-stream a b) `(cons ,a (delay ,b)))

(define (cdr-stream s) (force (cdr s)))
```