61A Lecture 31

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)

```
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)

```
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)

```
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)

```
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

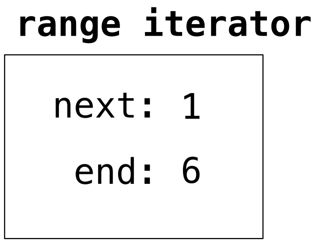
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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)
```



range iterator

next: 1

end: 6

Space: $\Theta(1)$

Sequence Operations

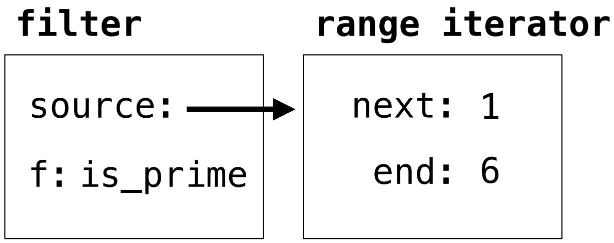
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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

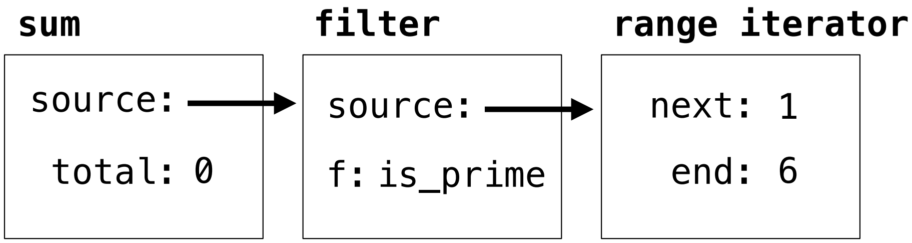
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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

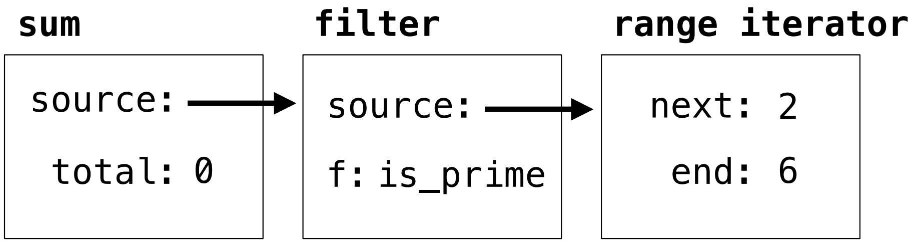
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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)
```



sum

source:

total: 0

filter

source:

f: is_prime

range iterator

next: 2

end: 6

Space: $\Theta(1)$

Sequence Operations

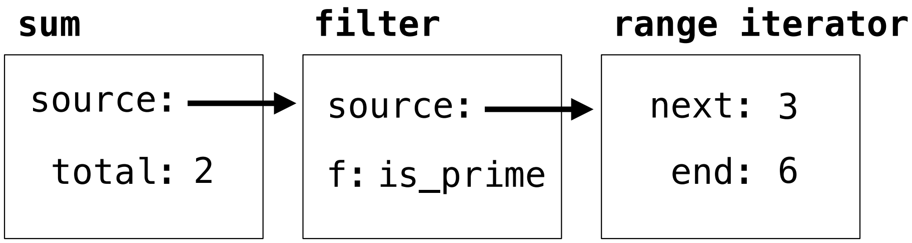
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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)
```



sum

source:

total: 2

filter

source:

f: is_prime

range iterator

next: 3

end: 6

Space: $\Theta(1)$

Sequence Operations

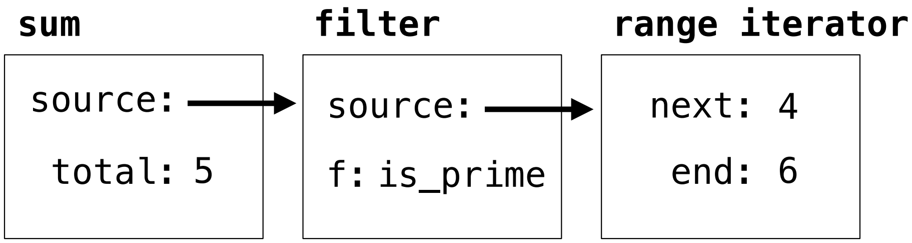
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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)
```


sum

source:

total: 5

filter

source:

f: is_prime

range iterator

next: 4

end: 6

Space: $\Theta(1)$

Sequence Operations

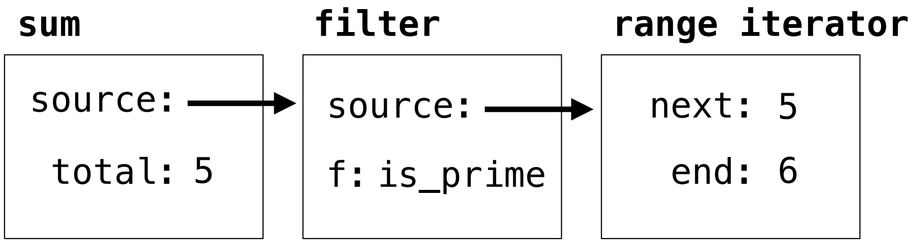
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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)
```



sum

source:

total: 5

filter

source:

f: is_prime

range iterator

next: 5

end: 6

Space: $\Theta(1)$

Sequence Operations

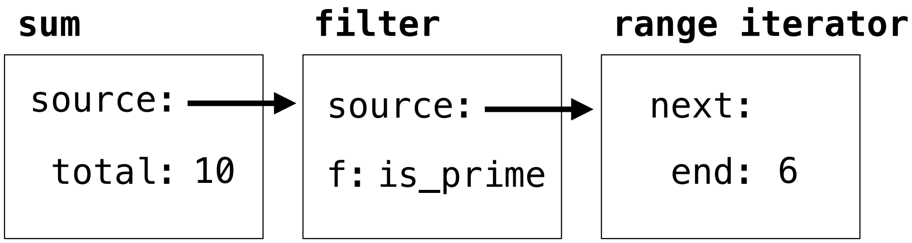
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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)
```



sum

source:

total: 10

filter

source:

f: is_prime

range iterator

next:

end: 6

Space: $\Theta(1)$

Sequence Operations

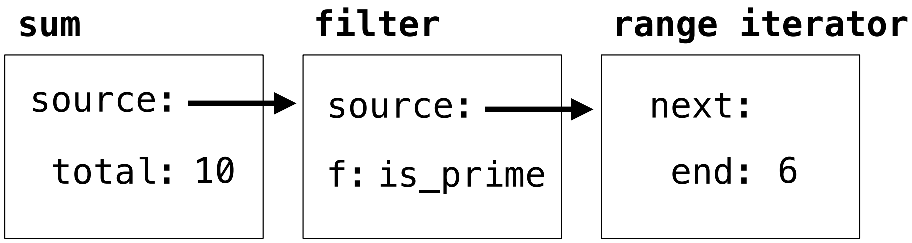
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

```
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)
```


sum

source:

total: 10

filter

source:

f: is_prime

range iterator

next:

end: 6

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

Sequence Operations

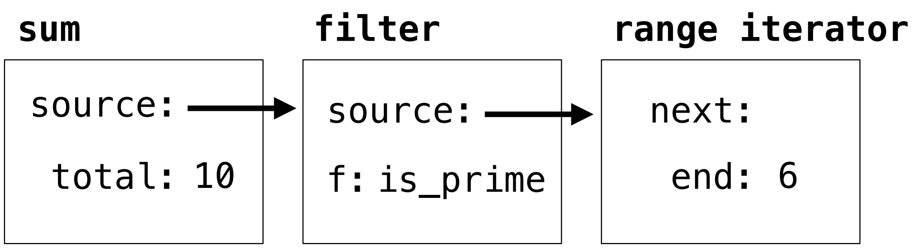
Map, filter, and reduce express sequence manipulation using compact expressions

Example: Sum all primes in an interval from **a** (inclusive) to **b** (exclusive)

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

(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)) -> 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

(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:

```
(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))
```

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:

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
```

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

(car (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:

```
(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

(car (cons 1 (/ 1 0))) -> ERROR

(cdr (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:

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

(car (cons 1 (/ 1 0))) -> ERROR

(cdr (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:

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

(car (cons 1 (/ 1 0))) -> ERROR    (car        (cons-stream 1 (/ 1 0))) -> 1

(cdr (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:

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

(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
```

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 . #[promise (not forced)])

(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

```
(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

```
(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

```
(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 (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

```
(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))))
```

(Demo)

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

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

Recursively Defined Streams

The rest of a constant stream is the constant stream

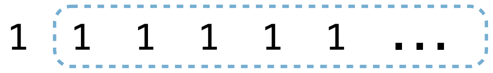
```
(define ones (cons-stream 1 ones))
```

1 1 1 1 1 1 ...

Recursively Defined Streams

The rest of a constant stream is the constant stream

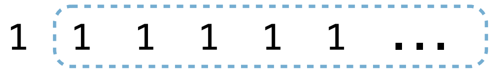
```
(define ones (cons-stream 1 ones))
```



Recursively Defined Streams

The rest of a constant stream is the constant stream

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



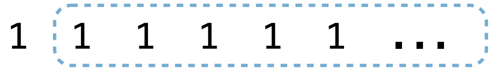
1 1 1 1 1 1 ...

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))
```



1 1 1 1 1 1 ...

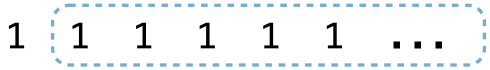
Combine two streams by separating each into car and cdr

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

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 ...

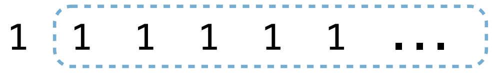
Combine two streams by separating each into car and cdr

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

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 ...

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))))
```

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 ...

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)))
```

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 ...

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

Recursively Defined Streams

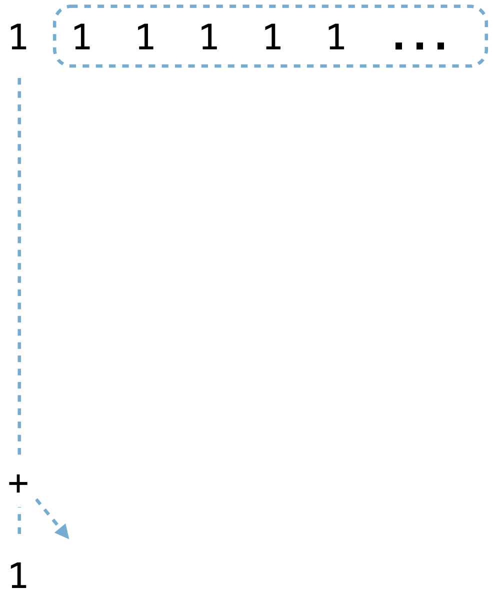
The rest of a constant stream is the constant stream

```
(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

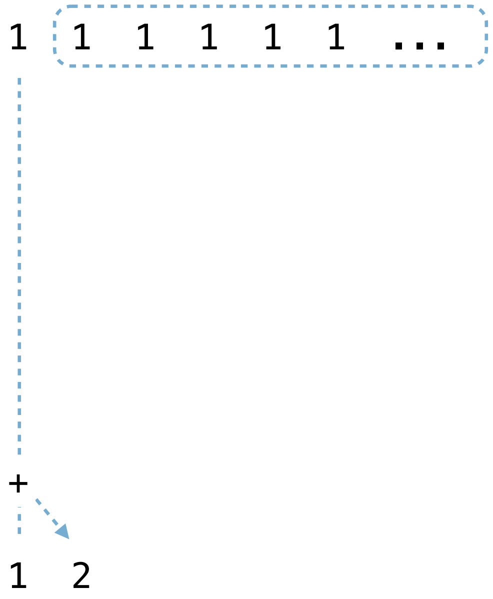
The rest of a constant stream is the constant stream

```
(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 2

Recursively Defined Streams

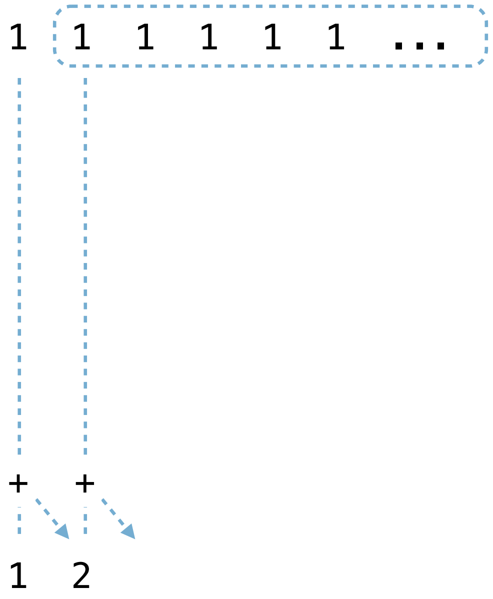
The rest of a constant stream is the constant stream

```
(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)))
```



Recursively Defined Streams

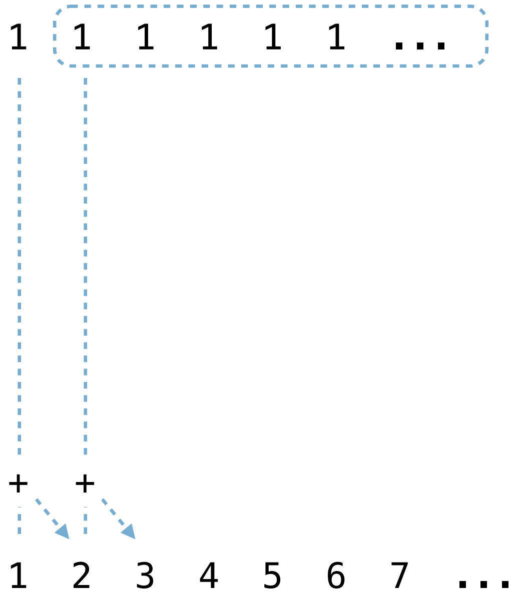
The rest of a constant stream is the constant stream

```
(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)))
```



Example: Repeats

Example: Repeats

```
(define a (cons-stream 1 (cons-stream 2 (cons-stream 3 a))))
```

Example: Repeats

```
(define a (cons-stream 1 (cons-stream 2 (cons-stream 3 a))))
```

```
What's (prefix 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)))))
```

What's (prefix 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)))))
```

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

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

```
(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)))))
```

```
                           1   2   3
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)))))
```

```
                            1    2    3    1    2    3    1    2
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)?      ( 1  2  3  1  2  3  1  2 )

What's (prefix (f a) 8)?  ( 1  __ __ __ __ __ __ __ )

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)?      ( 1  2  3  1  2  3  1  2 )

What's (prefix (f a) 8)?  ( 1  1  __ __ __ __ __ __ )

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)))))
```

```
                             1    2    3    1    2    3    1    2
What's (prefix a 8)?       ( __   __   __   __   __   __   __   __ )

                             1    1    2
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)))))
```

```
                           1    2    3    1    2    3    1    2
What's (prefix a 8)?     ( __   __   __   __   __   __   __   __ )

                           1    1    2    2
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)?       ( 1   2   3   1   2   3   1   2  )

What's (prefix (f a) 8)?   ( 1   1   2   2   3   3   1   1  )

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)?       ( 1  2  3  1  2  3  1  2 )

What's (prefix (f a) 8)?   ( 1  1  2  2  3  3  1  1 )

What's (prefix (g a) 8)?   ( 1  __ __ __ __ __ __ __ )
```

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)?      ( 1  2  3  1  2  3  1  2 )

What's (prefix (f a) 8)?  ( 1  1  2  2  3  3  1  1 )

What's (prefix (g a) 8)?  ( 1  2  2  __ __ __ __ __ )
```

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)?      ( _1_ _2_ _3_ _1_ _2_ _3_ _1_ _2_ )

What's (prefix (f a) 8)?  ( _1_ _1_ _2_ _2_ _3_ _3_ _1_ _1_ )

What's (prefix (g a) 8)?  ( _1_ _2_ _2_ _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)? (1 2 3 1 2 3 1 2)

What's (prefix (f a) 8)? (1 1 2 2 3 3 1 1)

What's (prefix (g a) 8)? (1 2 2 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)?       ( 1  2  3  1  2  3  1  2 )

What's (prefix (f a) 8)?   ( 1  1  2  2  3  3  1  1 )

What's (prefix (g a) 8)?   ( 1  2  2  3  3  3  3  1 )
```

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

```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
```

```
(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
                 (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)))))
```

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 <= 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 <= 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 <= 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 <= 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 <= 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 <= 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 <= 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 <= 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 <= 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 <= 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 <= 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

```
scm> (define promise (let ((x 2)) (delay     (+ x 1)) ))


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

```
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

```
scm> (define promise (let ((x 2)) (delay     (+ x 1)) ))

scm> (define x 5)

scm> (force promise)
3
```

```
(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)) ))
     (define promise (let ((x 2)) (lambda () (+ x 1)) ))

scm> (define x 5)

scm> (force promise)
3
```

```
(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)) ))
     (define promise (let ((x 2)) (lambda () (+ x 1)) ))

scm> (define x 5)

scm> (force promise)
3
```

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

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
```

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

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
```

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

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
```

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

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
```

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

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

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