CS 61A DISCUSSION 9

DELAYED EXPRESSIONS

Raymond Chan
Discussion 134
UC Berkeley Fall 16

AGENDA

- Announcements
- Iterables and Iterators
  - Generators
- Streams

ANNOUNCEMENTS

- Homework 11 due tonight.
- Scheme Project due 11/17.
  - Part 1 due tonight! (either EC or required, TBD)
  - Part 2 due 11/15.
- Homework 12 due 11/15.
- Lab 11 due Friday.
- Project party tonight.

ITERABLES AND ITERATORS

ITERABLES

- Iterables are container objects that be processed sequentially.
  - Lists, tuples, strings, dictionaries, ranges
- Call **iter** to obtain a new iterator for the iterable to process the elements.
- Can go through elements more than once.

ITERABLES AND ITERATORS

ITERATORS

- An iterator is an object that tracks the position in a sequence of values.
- It returns elements one at a time.
- Advance to the next element by calling **next**.
    - Eventually reach a StopIteration exception.
- Can only go through the elements once.
    - Can’t go to previous elements.
- Calling **iter** on an iterator will return itself.

ITERABLES AND ITERATORS

ITERATORS

```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)

>>> next(iterator1)

>>> next(iterator1)

>>> iterator2 = iter(iterable)
>>> next(iterator2)

>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```

iterator1

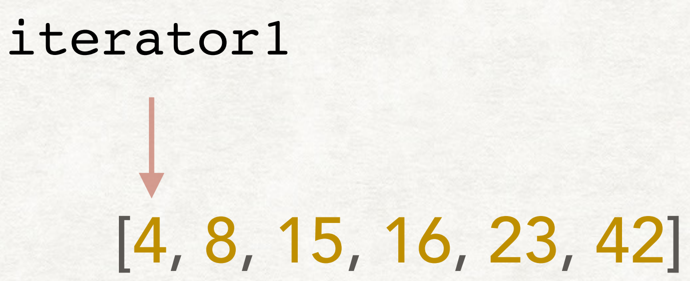
↓

[4, 8, 15, 16, 23, 42]

ITERABLES AND ITERATORS

ITERATORS

```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)

>>> next(iterator1)

>>> iterator2 = iter(iterable)
>>> next(iterator2)

>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```



ITERABLES AND ITERATORS

ITERATORS

```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)

>>> iterator2 = iter(iterable)
>>> next(iterator2)

>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```

iterator1

[4, 8, 15, 16, 23, 42]

ITERABLES AND ITERATORS

ITERATORS

```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)

>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```

`iterator1`

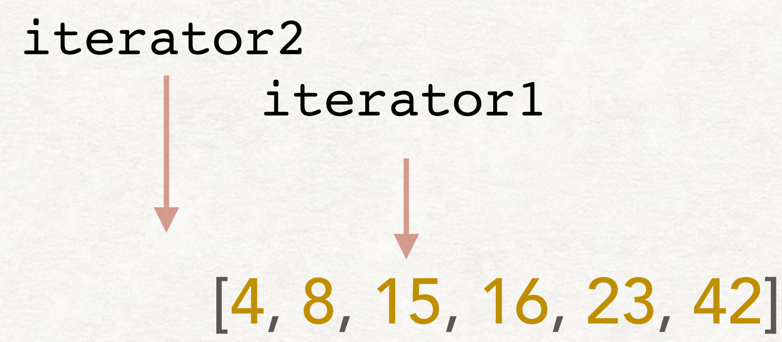
↓

[4, 8, 15, 16, 23, 42]

ITERABLES AND ITERATORS

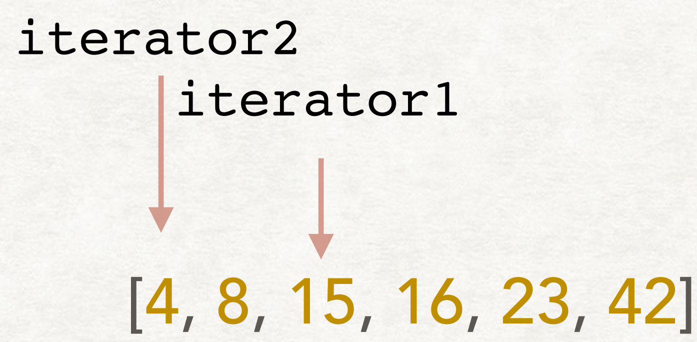
ITERATORS

```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)

>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```



ITERABLES AND ITERATORS

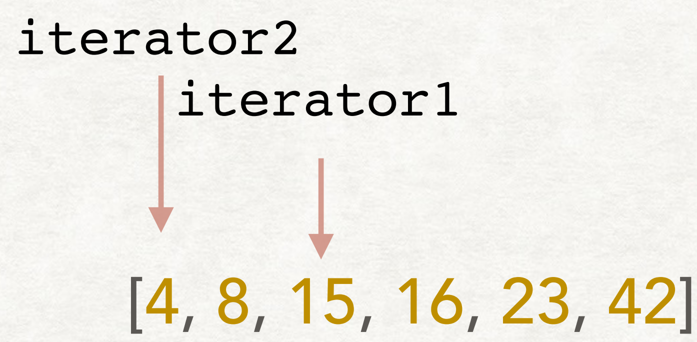
ITERATORS




```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)
4
>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```

ITERABLES AND ITERATORS

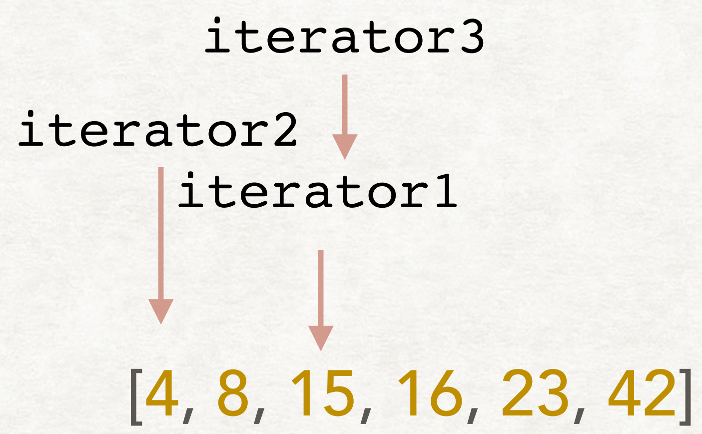
ITERATORS




```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)
4
>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```

ITERABLES AND ITERATORS

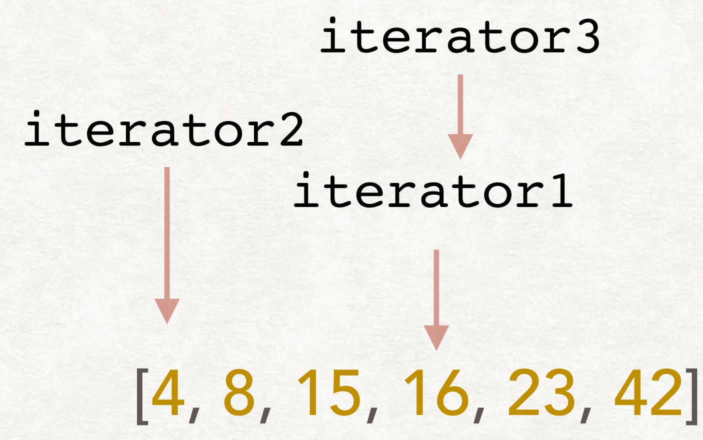
ITERATORS




```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)
4
>>> iterator3 = iter(iterator1)
>>> next(iterator3)

>>> next(iterator1)
```

ITERABLES AND ITERATORS

ITERATORS

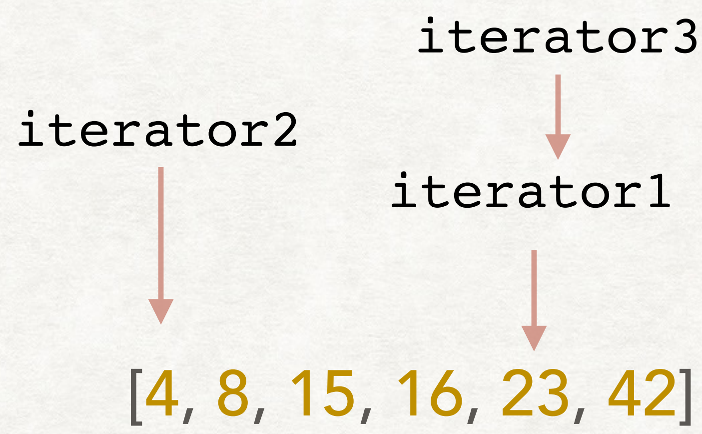



```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)
4
>>> iterator3 = iter(iterator1)
>>> next(iterator3)
16
>>> next(iterator1)
```

ITERABLES AND ITERATORS

ITERATORS

```
>>> iterable = [4, 8, 15, 16, 23, 42]
>>> iterator1 = iter(iterable)
>>> next(iterator1)
4
>>> next(iterator1)
8
>>> next(iterator1)
15
>>> iterator2 = iter(iterable)
>>> next(iterator2)
4
>>> iterator3 = iter(iterator1)
>>> next(iterator3)
16
>>> next(iterator1)
23
```



ITERABLES AND ITERATORS

ITERATORS

- A **for** loop calls **iter** on the iterable and continuously calls **next** on the iterator until a StopIteration Exception is caught.

```
for n in [1, 2, 3]:
   print(n)
```

```
iterator = iter([1, 2, 3])
try:
   while True:
      n = next(iterator)
      print(n)
except StopIteration:
   pass
```

ITERABLES AND ITERATORS

iterable	iterator
__iter__(self)	__iter__(self)
	__next__(self)

ITERABLES AND ITERATORS

GENERATORS

- A generator function uses a yield statement instead of return.
- Calling a generator function returns a *generator object*, a special kind of iterator.
- The yield tells Python we have a generator function.

ITERABLES AND ITERATORS

GENERATORS

- Each time we call **next** on the generator object, we executed until **yield**.
- At yield, we return the statement and *pauses the* frame.
- When we call **next**, we resume the frame and start from the line directly after yield until we hit another yield statement.
- There can be more than one yield.

ITERABLES AND ITERATORS

GENERATORS

```
def gen_naturals():
  current = 0
  while True:
    yield current
    current += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)

>>> next(gen)
```

ITERABLES AND ITERATORS

GENERATORS

```
def gen_naturals():
  current = 0
  while True:
    yield current
    current += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
```

ITERABLES AND ITERATORS

GENERATORS

```
def gen_naturals():
    current = 0
    while True:
        yield current
        current += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
```

ITERABLES AND ITERATORS

GENERATORS

```
def gen_naturals():
    current = 0
    while True:
        yield current
        current += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
```

ITERABLES AND ITERATORS

GENERATORS

```
def gen_naturals():
  current = 0
  while True:
    yield current
    current += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

ITERABLES AND ITERATORS

GENERATORS

```
def gen_naturals():
  current = 0
  while True:
    yield current
    current += 1

>>> gen = gen_naturals()
>>> gen
<generator object gen at ...>
>>> next(gen)
0
>>> next(gen)
1
```

```
def gen_naturals_limit(n):
  current = 0
  while current <= n:
    yield current
    current += 1


>>> gen_limit = gen_naturals_limit(2)
>>> next(gen_limit)
0
>>> next(gen_limit)
1
>>> next(gen_limit)
2
>>> next(gen_limit)
StopIteration Exception
```

Reaching the end of the frame raises StopIteration

ITERABLES AND ITERATORS

GENERATORS - YIELD FROM

- **yield from** takes in an iterable and yields each of the values from that iterable

```
square = lambda x: x*x
def many_squares(s):
    for x in s:
        yield square(x)
    yield from [square(x) for x in s]
    yield from map(square, s)

>>> list(many_squares([1, 2, 3]))
[1, 4, 9, 1, 4, 9, 1, 4, 9]
```

```
def f(s):
    yield from [square(x) for x in s]

>>> g = f([1, 2])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
StopIteration Exception
```

list takes in an iterable and calls next until a StopIteration

STREAMS

- Iterators and generators are *lazy or delayed* and can potentially represent infinite sequences.
- We only compute the next value when we ask for it.
- Scheme Lists cannot be infinite.

STREAMS

- The the second argument to **cons** is always evaluated.

```
> (define (naturals n)
      (cons n (naturals (+ n 1)))
> Maximum Recursion Depth Reached
```

STREAMS

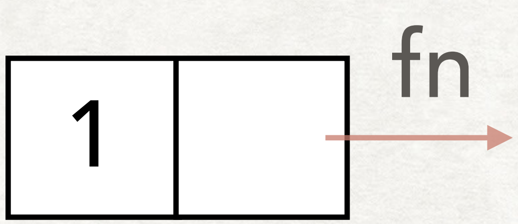
- Streams are lazy Scheme Lists.
- The rest of the list is not evaluated until you ask for it.
- Once you have asked for it once, it will save (memoize) the value so that it will not be evaluated again.
- Streams can be infinite or finite (ends with nil).

STREAMS

- **cons-stream** creates a pair where the second is a stream.
- **nil** is an empty stream.
- **car** returns the first element.
- **cdr-stream** *computes* and *returns* the rest of the stream.
- **cdr** will not calculate the next value.
  - Looks at second element of pair but does not evaluate.

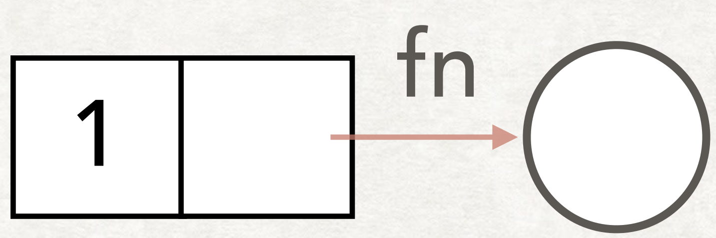
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
```



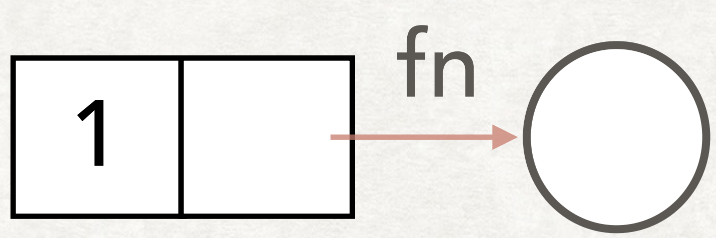
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
```



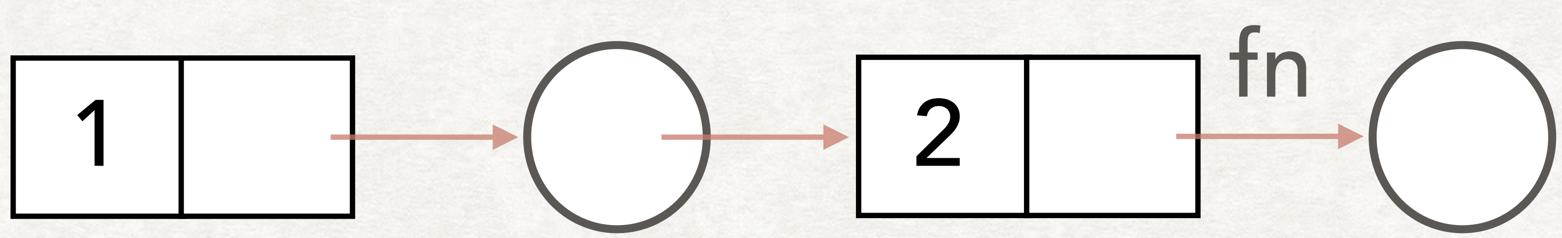
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
```



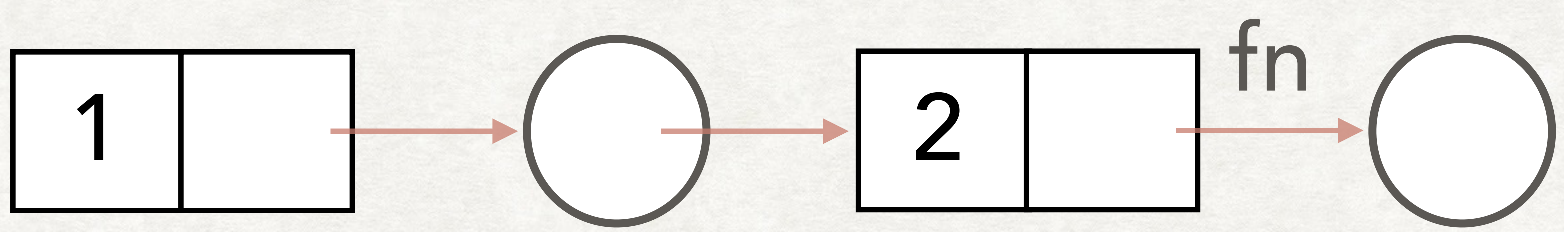
STREAMS

> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])



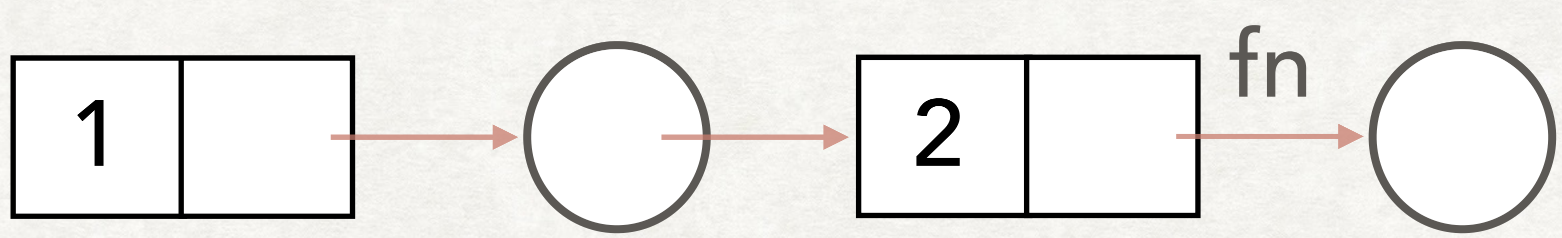
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
```



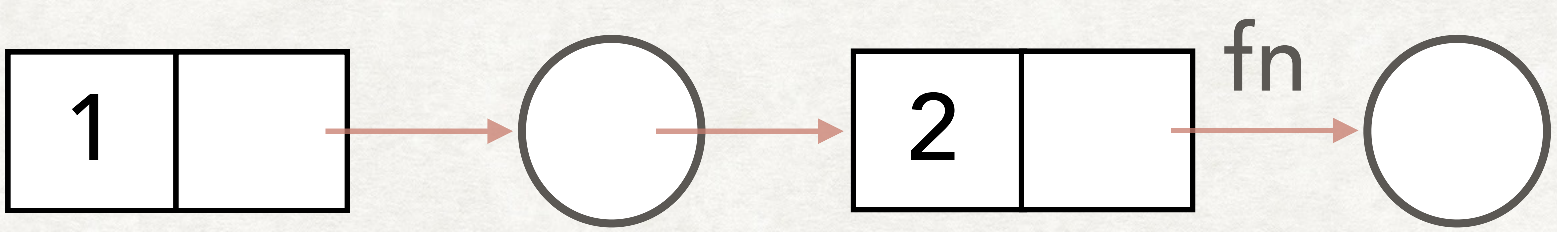
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
```



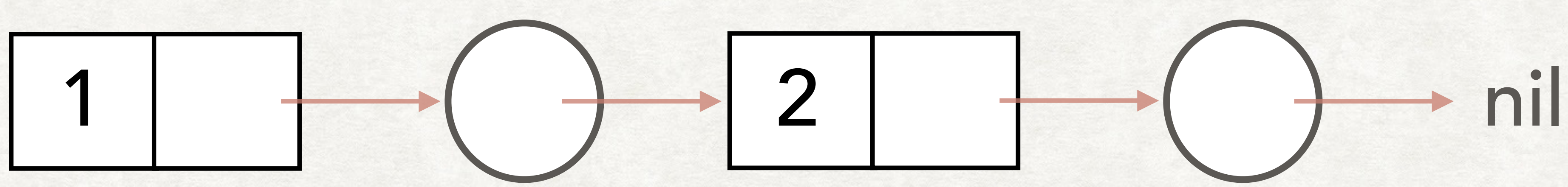
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
```



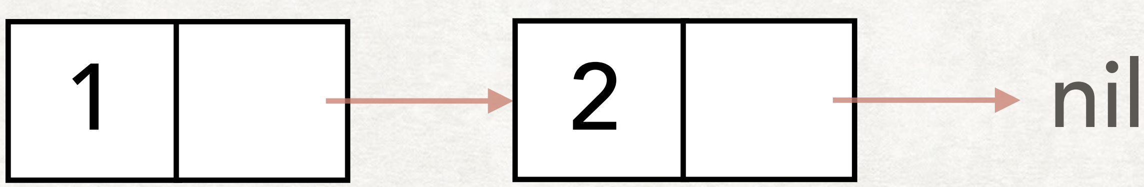
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
()
```



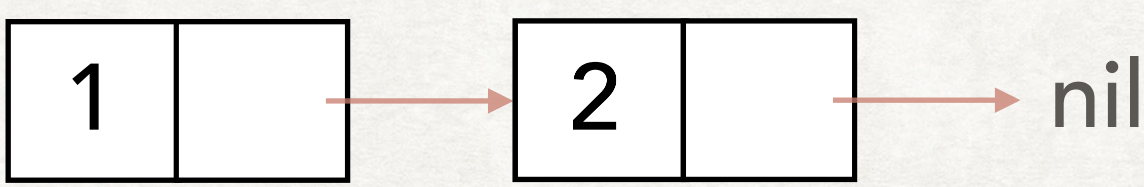
STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
()
> (cdr s)
```



STREAMS

```
> (define s (cons-stream 1 (cons-stream 2 nil)))
> s
(1 . #[promised (not forced)])
> (cdr-stream s)
(2 . #[promised (not forced)])
> s
(1 . #[promised (forced)])
> (cdr-stream (cdr-stream (cdr-stream s)))
()
> (cdr s)
#[promised (forced)]
```



STREAMS

Q1

```
> (define (has-even? s)
        (cond ((null? s) False)
              ((even? (car s)) True)
              (else (has-even? (cdr-streams)))))
> has-even?
> (define ones (cons-stream 1 ones))
ones
> (define twos (cons-stream 2 twos))
twos
> ones

> cdr ones)

> (cdr-stream ones)
```

```
> (cdr-stream (cdr-stream ones))

> (has-even? ones)

> (has-even? twos)
```

STREAMS

Q1

```
> (define (has-even? s)
        (cond ((null? s) False)
              ((even? (car s)) True)
              (else (has-even? (cdr-streams)))))
> has-even?
> (define ones (cons-stream 1 ones))
ones
> (define twos (cons-stream 2 twos))
twos
> ones
(1 . #[promise (not forced)])
> cdr ones)

> (cdr-stream ones)
```

```
> (cdr-stream (cdr-stream ones))

> (has-even? ones)

> (has-even? twos)
```

STREAMS

Q1

```
> (define (has-even? s)
        (cond ((null? s) False)
              ((even? (car s)) True)
              (else (has-even? (cdr-streams)))))
> has-even?
> (define ones (cons-stream 1 ones))
ones
> (define twos (cons-stream 2 twos))
twos
> ones
(1 . #[promise (not forced)])
> cdr ones)
#[promise (not forced)]
> (cdr-stream ones)
```

```
> (cdr-stream (cdr-stream ones))

> (has-even? ones)

> (has-even? twos)
```

STREAMS

Q1

```
> (define (has-even? s)
        (cond ((null? s) False)
              ((even? (car s)) True)
              (else (has-even? (cdr-streams)))))
> has-even?
> (define ones (cons-stream 1 ones))
ones
> (define twos (cons-stream 2 twos))
twos
> ones
(1 . #[promise (not forced)])
> cdr ones)
#[promise (not forced)]
> (cdr-stream ones)
(1 . #[promise (not forced)])
```

```
> (cdr-stream (cdr-stream ones))

> (has-even? ones)

> (has-even? twos)
```

STREAMS

Q1

```
> (define (has-even? s)
        (cond ((null? s) False)
              ((even? (car s)) True)
              (else (has-even? (cdr-streams)))))
> has-even?
> (define ones (cons-stream 1 ones))
ones
> (define twos (cons-stream 2 twos))
twos
> ones
(1 . #[promise (not forced)])
> cdr ones)
#[promise (not forced)]
> (cdr-stream ones)
(1 . #[promise (not forced)])
```

```
> (cdr-stream (cdr-stream ones))
(1 . #[promise (forced)])
> (has-even? ones)

> (has-even? twos)
```

STREAMS

Q1

```
> (define (has-even? s)
        (cond ((null? s) False)
              ((even? (car s)) True)
              (else (has-even? (cdr-streams)))))
> has-even?
> (define ones (cons-stream 1 ones))
ones
> (define twos (cons-stream 2 twos))
twos
> ones
(1 . #[promise (not forced)])
> cdr ones)
#[promise (not forced)]
> (cdr-stream ones)
(1 . #[promise (not forced)])
> (cdr-stream (cdr-stream ones))
(1 . #[promise (forced)])
> (has-even? ones)
# Runs forever
> (has-even? twos)
```

STREAMS

Q1

```
> (define (has-even? s)
        (cond ((null? s) False)
              ((even? (car s)) True)
              (else (has-even? (cdr-streams)))))
> has-even?
> (define ones (cons-stream 1 ones))
ones
> (define twos (cons-stream 2 twos))
twos
> ones
(1 . #[promise (not forced)])
> cdr ones)
#[promise (not forced)]
> (cdr-stream ones)
(1 . #[promise (not forced)])
```

```
> (cdr-stream (cdr-stream ones))
(1 . #[promise (forced)])
> (has-even? ones)
# Runs forever
> (has-even? twos)
True
```

RECAP

- Iterators goes over the elements of a sequence one at a time.
- Generators return generator objects that outputs at **yield** and passes the frame.
- Streams are lists such that the rest of the list is not calculated until we need it.

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

nat -> 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 …

(slice nat 4 12) -> (4 5 6 7 8 9 10 11 12)

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

nat -> 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 …

(slice nat 4 12) -> (4 5 6 7 8 9 10 11 12)

We need to reach the stream where we want to start using its values. 0, 1, 2, 3 is skipped. The starting point is the element at the start index. To count that number of elements, we can decrement start by 1, for each recursive call.

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

indices 4,   3,   2,   1,   0,
nat -> 0   1   2   3   4   5   6   7   8   9   10   11   12   13   14   15 …

(slice nat 4 12) -> (4 5 6 7 8 9 10 11 12)

We need to reach the stream where we want to start using its values. 0, 1, 2, 3 is skipped. The starting point is the element at the start index. To count that number of elements, we can decrement start by 1, for each recursive call.

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

indices 4, 3, 2, 1, 0,
nat -> 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 …

(slice nat 4 12) -> (4 5 6 7 8 9 10 11 12)

The end index is going to tell us when to end. After we reach 4, each time we include an element, we want to decrement the end index. We want to keep only the number of elements specified from the original start and end. So we want to end when end = original end - original start. But at a certain recursive call, original start and end are lost without using a new variable. So when we decremented start by 1 to reach the starting point, we can also decrement end by 1. Thus at 4, end will become the number of elements to keep. When it reaches 0, we know we have used up all the elements. Start index is not going to matter after we reached the starting point as long as we do not increase it beyond 0.

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

indices	4,12	3,11	2,10	1,9	0,8	0,7	0,6	0,5	0,4	0,3	0,2	0,1	0,0			
nat ->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15 …

(slice nat 4 12) -> (4 5 6 7 8 9 10 11)

The end index is going to tell us when to end. After we reach 4, each time we include an element, we want to decrement the end index. We want to keep only the number of elements specified from the original start and end. So we want to end when end = original end - original start. But at a certain recursive call, original start and end are lost without using a new variable. So when we decremented start by 1 to reach the starting point, we can also decrement end by 1. Thus at 4, end will become the number of elements to keep. When it reaches 0, we know we have used up all the elements. Start index is not going to matter after we reached the starting point as long as we do not increase it beyond 0.

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

```
(define (slice stream start end)
  (cond ((null? stream) nil)
        ((= end 0) nil)
        ((> start 0)
          (slice (cdr-stream (- start 1) (- end 1)))
        (else
          (cons (car stream)
            (slice (cdr-stream stream)
                   start
                   (- end 1))))))
```

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

A stream could be finite, and
thus we want to account for that case

```
(define (slice stream start end)
   (cond ((null? stream) nil)
            ((= end 0) nil)
            ((> start 0)
               (slice (cdr-stream (- start 1) (- end 1)))
            (else
               (cons (car stream)
                  (slice (cdr-stream stream)
                         start
                         (- end 1))))))
```

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

```
(define (slice stream start end)
   (cond ((null? stream) nil)
            ((= end 0) nil)
            ((> start 0)
               (slice (cdr-stream (- start 1) (- end 1)))
            (else
               (cons (car stream)
                  (slice (cdr-stream stream)
                           start
                           (- end 1))))))
```

A stream could be finite, and thus we want to account for that case

When end is 0, we know this is the first element that we want to exclude and thus end the list. (We keep all elements up to, but excluding the element indexed at end)

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

```
(define (slice stream start end)
   (cond ((null? stream) nil)
         ((= end 0) nil)
         ((> start 0)
           (slice (cdr-stream (- start 1) (- end 1)))
         (else
           (cons (car stream)
             (slice (cdr-stream stream)
                    start
                    (- end 1))))))
```

A stream could be finite, and thus we want to account for that case

When end is 0, we know this is the first element that we want to exclude and thus end the list. (We keep all elements up to, but excluding the element indexed at end)

Before start has reached 0 yet, we don't want to include any elements. Thus we only make the recursive call as slicing from the next element with both start and end decremented by 1 is the same as slicing from the current element with initial start and end.

```
(slice (2 3 4 …) 2 10)
-> (slice (3 4 …) 1 9)
     -> (slice (4 …) 0 8)
```

STREAMS

SLICE - NOTES

Slice a stream from a start index to an end index. Returns a Scheme List

```
(define (slice stream start end)
   (cond ((null? stream) nil)
            ((= end 0) nil)
            ((> start 0)
               (slice (cdr-stream (- start 1) (- end 1)))
            (else
               (cons (car stream)
                  (slice (cdr-stream stream)
                              start
                              (- end 1))))))
```

A stream could be finite, and thus we want to account for that case

When end is 0, we know this is the first element that we want to exclude and thus end the list. (We keep all elements up to, but excluding the element indexed at end)

Before start has reached 0 yet, we don't want to include any elements. Thus we only make the recursive call as slicing from the next element with both start and end decremented by 1 is the same as slicing from the current element with initial start and end.

```
(slice (2 3 4 …) 2 10)
-> (slice (3 4 …) 1 9)
     -> (slice (4 …) 0 8)
```

This is where we want to include the current element using cons to making our list. The second element of this pair will the be recursive call to slice from the next element with one less element to keep. Thus end is decremented by 1

```
(slice (4 5 6 …) 0 8)
-> (4 (slice (5 6…) 0, 7)
   -> (4 5 (slice (6 …) 0, 6)
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

First let's understand zip-with

```
(define (zip-with f xs ys)
   (if (or (null? xs) (null? ys)) nil
       (cons-stream (f (car xs) (car ys)
                    (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

zip-with takes in 2 streams and combines each corresponding element with function f.

(zip-with + (naturals 0) (naturals 0)) -> (- 2 4 6 …)

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

```
(define (zip-with f xs ys)
   (if (or (null? xs) (null? ys)) nil
       (cons-stream (f (car xs) (car ys)
                    (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

Lets define a factorial stream with zip-with. The recursive structure is multiplying the current index by the previous factorial value.
factorial(n) = n * factorial(n - 1)
Writing out the sequences of the indices and factorials:

(naturals 1)	1	2	3	4	5	6	7	8	10	11 …
factorials	1	1	2	6	24	120 …				

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

```
(define (zip-with f xs ys)
   (if (or (null? xs) (null? ys)) nil
       (cons-stream (f (car xs) (car ys)
                    (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

Lets define a factorial stream with zip-with. The recursive structure is multiplying the current index by the previous factorial value.
factorial(n) = n * factorial(n - 1)
Writing out the sequences of the indices and factorials:

```
(naturals 1)   1  2  3  4  5  6  7  8  10  11 …
               *  *  *
factorials     1  1  2  6  24  120 …
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

```
(define (zip-with f xs ys)
   (if (or (null? xs) (null? ys)) nil
       (cons-stream (f (car xs) (car ys)
                    (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

Lets define a factorial stream with zip-with. The recursive structure is multiplying the current index by the previous factorial value.
factorial(n) = n * factorial(n - 1)
Writing out the sequences of the indices and factorials:

```
(naturals 1)   1  2  3  4  5  6  7  8  10  11 …
               *  *  *
factorials     1  1  2  6  24  120 …
```

(define factorials (cons-stream 1 (zip-with * (naturals 1) factorials)))

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

```
(define (zip-with f xs ys)
   (if (or (null? xs) (null? ys)) nil
       (cons-stream (f (car xs) (car ys)
                        (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
(define factorials
   (cons-stream 1
          (zip-with * (naturals 1) factorials)))
```

We need start our factorials off with to have the value of 1. The rest of the elements are zipping together factorials with naturals starting at 1.

(naturals 1) 1 2 3 4 5 6 7 8 10 11 …

* * *

factorials 1 1 2 6 24 120 …

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys))
          (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
(define factorials
   (cons-stream 1
         (zip-with * (naturals 1) factorials)))
```

```
(naturals 1)   1   2   3   4 ...
                 *   *   *
factorials     1   1   2   6 ...
```

```
> (define factorials ...)
```

```
factorials   (1 (zip-with * (1 ...)         ))
factorials   (1 (zip-with * (1 ...) (1 ...) ))
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys))
          (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
(define factorials
   (cons-stream 1
         (zip-with * (naturals 1) factorials)))
```

```
(naturals 1)    1   2   3   4  ...
                *   *   *
factorials      1   1   2   6 ...
```

```
> (define factorials ...)
> (cdr-stream factorials)
```

```
factorials   (1 (zip-with * (1 ...)          ))
factorials   (1 (zip-with * (1 ...) (1 ...) ))
```

```
factorials   (1 (* 1 1) (zip-with * (2 ...)          ))
factorials   (1     1   (zip-with * (2 ...) (1 ...)  ))
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys)
          (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
(define factorials
   (cons-stream 1
         (zip-with * (naturals 1) factorials)))
```

```
(naturals 1)    1   2   3   4  …
                *   *   *
factorials      1   1   2   6 …
```

```
> (define factorials …)
> (cdr-stream factorials)
> (cdr-stream (cdr-stream
          (cdr-stream factorial)))
```

```
factorials   (1 (zip-with * (1 …)          ))
factorials   (1 (zip-with * (1 …) (1 …) ))

factorials   (1 (* 1 1) (zip-with * (2 …)          ))
factorials   (1     1   (zip-with * (2 …) (1 …)  ))

factorials   (1 1 (* 2 1) (zip-with + (3 …)          ))
factorials   (1 1     2   (zip-with + (3 …) (2 …)  ))
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys))
          (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
(define factorials
   (cons-stream 1
          (zip-with * (naturals 1) factorials)))
```

```
(naturals 1)   1   2   3   4  ...
               *   *   *
factorials     1   1   2   6 ...
```

```
> (define factorials ...)
> (cdr-stream factorials)
> (cdr-stream (cdr-stream
        (cdr-stream factorial)))
> (cdr-stream (cdr-stream
        (cdr-stream (cdr-stream factorial))))
```

```
factorials   (1 (zip-with * (1 ...)           ))
factorials   (1 (zip-with * (1 ...) (1 ...) ))

factorials   (1 (* 1 1) (zip-with * (2 ...)          ))
factorials   (1     1   (zip-with * (2 ...) (1 ...)  ))

factorials   (1 1 (* 2 1) (zip-with + (3 ...)          ))
factorials   (1 1     2   (zip-with + (3 ...) (2 ...)  ))

factorials   (1 1 2 (* 3 2) (zip-with + (4 ...)          ))
factorials   (1 1 2     6   (zip-with + (4 ...) (6 ...)  ))
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

```
(define (zip-with f xs ys)
   (if (or (null? xs) (null? ys)) nil
       (cons-stream (f (car xs) (car ys)
                         (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

Lets define a fibs stream with zip-with. The recursive structure is adding the last 2 fibonacci sequence together.
fib(n) = fib(n-1) + fib(n-2)
Writing out the sequences of 2 fibs

fibs 0 1 1 2 3 5 8 …

fibs 0 1 1 2 3 5 8 …

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys)
                   (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

Lets define a fibs stream with zip-with. The recursive structure is adding the last 2 fibonacci sequence together.
fib(n) = fib(n-1) + fib(n-2)
Writing out the sequences of 2 fibs

```
fibs        0  1  1  2  3  5  8 …
            +  +  +
fibs     0  1  1  2  3  5  8 …
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

Combine infinite streams together to form infinite streams of factorial numbers and the Fibonacci sequence.

```
(define (zip-with f xs ys)
   (if (or (null? xs) (null? ys)) nil
       (cons-stream (f (car xs) (car ys)
                    (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

Lets define a fibs stream with zip-with. The recursive structure is adding the last 2 fibonacci sequence together.
fib(n) = fib(n-1) + fib(n-2)
Writing out the sequences of 2 fibs

```
fibs         0  1  1  2  3  5  8 …
             +  +  +
fibs     0  1  1  2  3  5  8 …
```

Since the fibs sequence needs the first 2 elements to compute the first one. We start it off with 2 elements before recursively using zip-with.

```
(define fibs (cons-stream 0 (cons-stream 1 (zip-with fibs (cdr-stream fibs)))))
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

fibs (0 …)

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
     (cons-stream (f (car xs) (car ys))
        (zip-with f (cdr-stream xs) (cdr-stream ys)))))

(define fibs (cons-stream 0 (cons-stream 1
     (zip-with fibs (cdr-stream fibs)))))
```

```
fibs        0  1  1  2  3  5  8 …
            +  +  +
fibs     0  1  1  2  3  5  8 …
```

```
> (define fibs …)
```

STREAMS

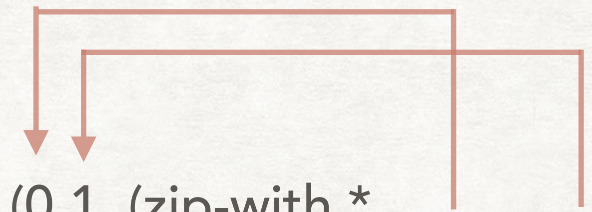
ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys)
          (zip-with f (cdr-stream xs) (cdr-stream ys)))))

(define fibs (cons-stream 0 (cons-stream 1
        (zip-with fibs (cdr-stream fibs)))))
```

```
fibs         0  1  1  2  3  5  8 …
             +  +  +
fibs      0  1  1  2  3  5  8 …
```

```
> (define fibs …)
> (cdr-stream fibs)
```

fibs (0 …)



fibs (0 1 (zip-with *))

fibs (0 1 (zip-with * (0 …) (1 …)))

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys))
         (zip-with f (cdr-stream xs) (cdr-stream ys)))))

 (define fibs (cons-stream 0 (cons-stream 1
        (zip-with fibs (cdr-stream fibs)))))
```

```
fibs         0   1   1   2   3   5   8 …
             +   +   +
fibs     0   1   1   2   3   5   8 …
```

```
> (define fibs …)
> (cdr-stream fibs)
> (cdr-stream (cdr-stream
        (cdr-stream fibs)))
```

```
fibs   (0 …)

fibs   (0 1  (zip-with *              ))
fibs   (0 1  (zip-with * (0 …) (1 …) ))

fibs   (0 1 (+ 0 1) (zip-with *              ))
fibs   (0 1     1   (zip-with * (1 …) (1 …)))
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys)
         (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
(define fibs (cons-stream 0 (cons-stream 1
        (zip-with fibs (cdr-stream fibs)))))
```

```
fibs        0  1  1  2  3  5  8 …
            +  +  +
fibs     0  1  1  2  3  5  8 …
```

```
> (define fibs …)
> (cdr-stream fibs)
> (cdr-stream (cdr-stream
          (cdr-stream fibs)))
> (cdr-stream (cdr-stream
          (cdr-stream (cdr-stream fibs))))
```

```
fibs   (0 …)

fibs   (0 1  (zip-with *                    ))
fibs   (0 1  (zip-with * (0 …) (1 …) ))

fibs   (0 1 (+ 0 1) (zip-with *               ))
fibs   (0 1      1   (zip-with * (1 …) (1 …)))

fibs  (0 1 1 (+ 1 1) (zip-with *              ))
fibs  (0 1 1    2    (zip-with * (1 …) (2 …)  ))
```

STREAMS

ZIP-WITH, FACTORIAL, FIBS - NOTES

```
(define (zip-with f xs ys)
  (if (or (null? xs) (null? ys)) nil
      (cons-stream (f (car xs) (car ys)
          (zip-with f (cdr-stream xs) (cdr-stream ys)))))
```

```
(define fibs (cons-stream 0 (cons-stream 1
        (zip-with fibs (cdr-stream fibs)))))
```

```
fibs          0   1   1   2   3   5   8 ...
              +   +   +
fibs      0   1   1   2   3   5   8 ...
```

```
> (define fibs ...)
> (cdr-stream fibs)
> (cdr-stream (cdr-stream
          (cdr-stream fibs)))
> (cdr-stream (cdr-stream
          (cdr-stream (cdr-stream fibs))))
> (cdr-stream (cdr-stream
          (cdr-stream
          (cdr-stream (cdr-stream fibs)))))
```

```
fibs   (0 ...)

fibs   (0 1  (zip-with *                 ))
fibs   (0 1  (zip-with * (0 ...) (1 ...) ))

fibs   (0 1 (+ 0 1) (zip-with *                  ))
fibs   (0 1    1    (zip-with * (1 ...) (1 ...)))

fibs  (0 1 1 (+ 1 1) (zip-with *                 ))
fibs  (0 1 1    2    (zip-with * (1 ...) (2 ...)  ))

fibs  (0 1 1 2 (+ 1 2) (zip-with *                 ))
fibs  (0 1 1 2    3    (zip-with * (2 ...) (3 ...) ))
```