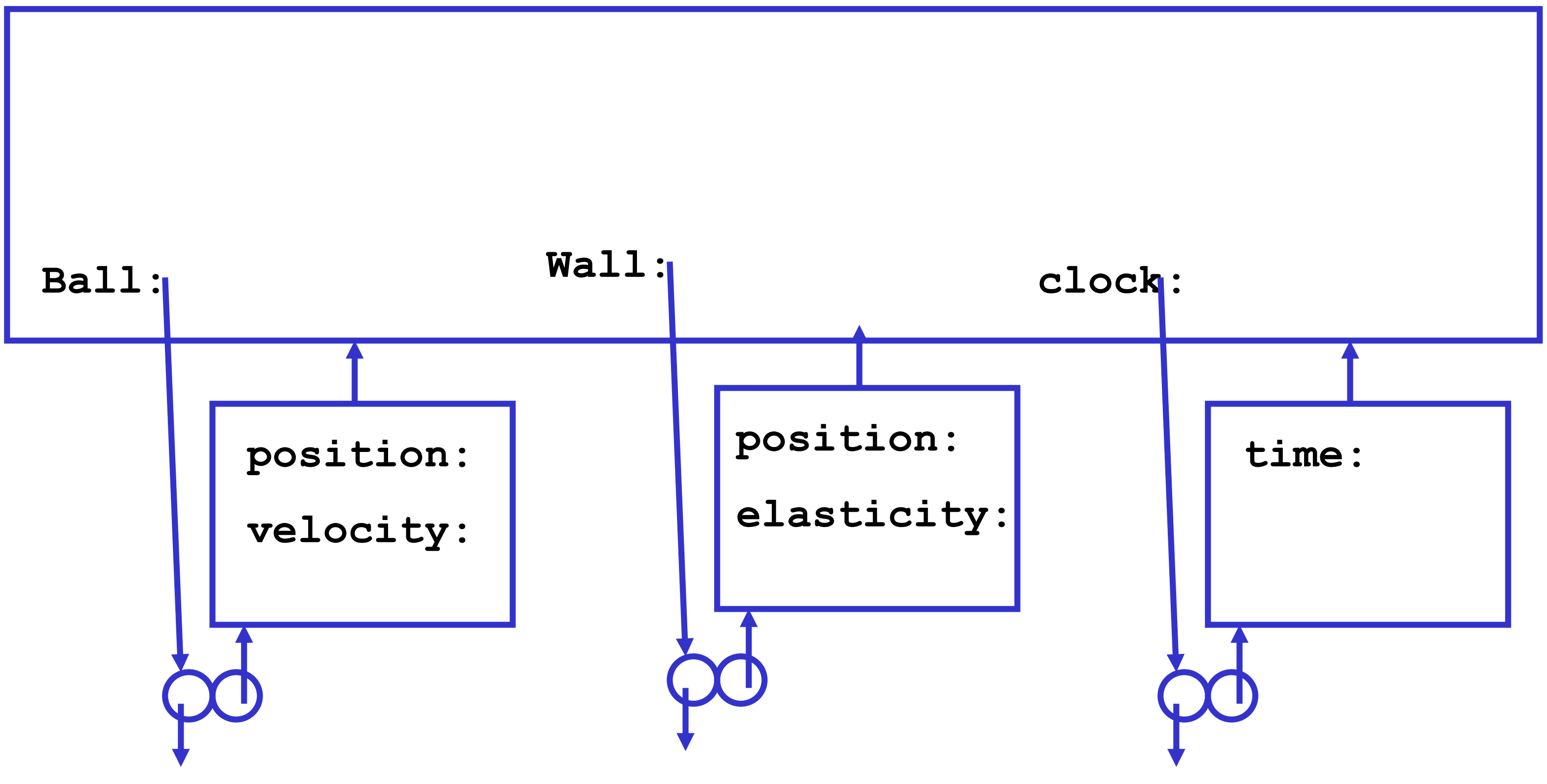
# 6.001 SICP
# Streams – the lazy way to infinity, and beyond…

Beyond Scheme – designing language variants:

- Streams – an alternative programming style

# Streams – motivation

- Imagine simulating the motion of a ball bouncing against a wall
  - Use state variables, clock, equations of motion to update



# Streams – motivation

- State of the simulation captured in instantaneous values of state variables

| | | |
|---|---|---|
| `Clock: 1` | `Ball: (x1 y1)` | `Wall: e1` |
| `Clock: 2` | `Ball: (x2 y2)` | `Wall: e2` |
| `Clock: 3` | `Ball: (x3 y3)` | `Wall: e2` |
| `Clock: 4` | `Ball: (x4 y4)` | `Wall: e2` |
| `Clock: 5` | `Ball: (x5 y5)` | `Wall: e3` |
| | ... | |

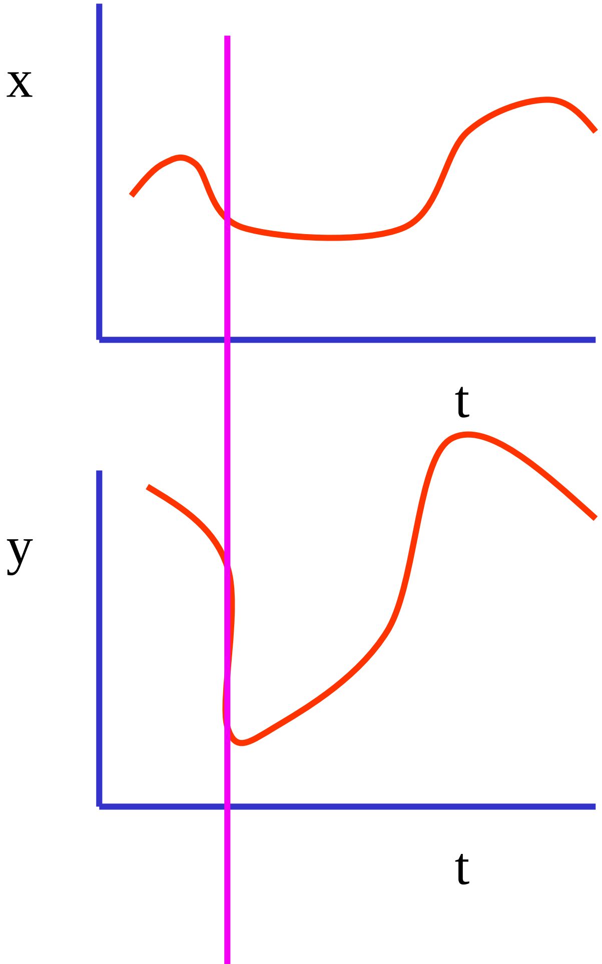
# Streams – motivation

- Another view of the same informaton

| Clock: | Ball: | Wall: |
|---|---|---|
| 1 | (x1 y1) | e1 |
| 2 | (x2 y2) | e2 |
| 3 | (x3 y3) | e2 |
| 4 | (x4 y4) | e2 |
| 5 | (x5 y5) | e3 |
| … | … | … |

# Streams – Basic Idea

- Have each object output a continuous stream of information
- State of the simulation captured in the history (or stream) of values



# Demo

# Remember our Lazy Language?

- Normal (Lazy) Order Evaluation:
  - go ahead and apply operator with unevaluated argument subexpressions
  - evaluate a subexpression only when value is *needed*
    - to print
    - by primitive procedure (that is, primitive procedures are "*strict*" in their arguments)
    - on branching decisions
    - a few other cases
- Memoization -- keep track of value after expression is evaluated

- Compromise approach: **give programmer control between normal and applicative order.**

# Variable Declarations: `lazy` and `lazy-memo`

- Handle lazy and lazy-memo extensions in an upward-compatible fashion.;

```
(lambda (a (b lazy) c (d lazy-memo)) ...)
```

  - "a", "c" are normal variables (evaluated before procedure application
  - "b" is lazy; it gets (re)-evaluated each time its value is actually needed
  - "d" is lazy-memo; it gets evaluated the first time its value is needed, and then that value is returned again any other time it is needed again.

# The lazy way to streams

- Use cons

```
(define (cons-stream x (y lazy-memo))
   (cons x y))
(define stream-car car)
(define stream-cdr cdr)
```

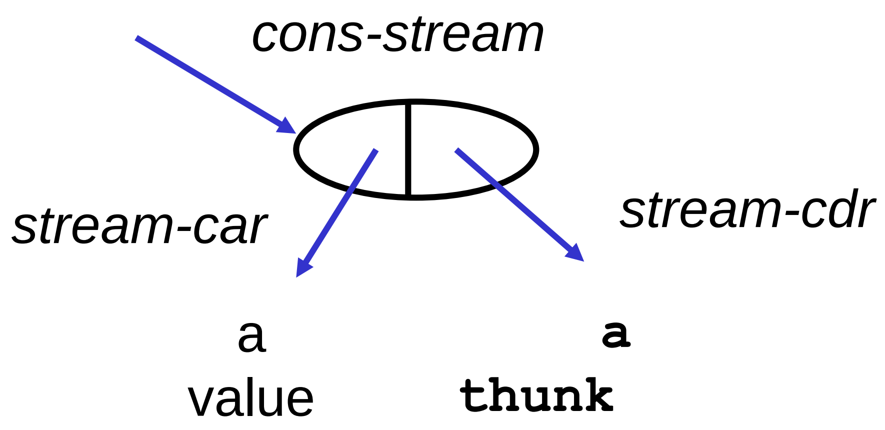
- Or, users could implement a *stream abstraction*:

```
(define (cons-stream x (y lazy-memo))
  (lambda (msg)
    (cond ((eq? msg 'stream-car) x)
          ((eq? msg 'stream-cdr) y)
          (else (error "unknown stream msg" msg)))))

(define (stream-car s) (s 'stream-car))
(define (stream-cdr s) (s 'stream-cdr))
```

# Stream Object

- A pair-like object, except the cdr part is *lazy* (not evaluated until needed):




- Example

```
(define x (cons-stream 99 (/ 1 0)))
(stream-car x) => 99
(stream-cdr x) => error – divide by zero
```

**Because stream-cdr is same as cdr, this is a primitive procedure application, hence forces evaluation**

# Decoupling computation from description

- Can separate order of events in computer from apparent order of events in procedure description

```
(list-ref
  (filter (lambda (x) (prime? x))
          (enumerate-interval 1 100000000))
  100)
```

Creates 100K elements

Creates 1M elements

```
(define (stream-interval a b)
   (if (> a b)
       the-empty-stream
       (cons-stream a (stream-interval (+ a 1) b))))
```

```
(stream-ref
   (stream-filter (lambda (x) (prime? x))
                  (stream-interval 1 100000000))
   100)
```

# Stream-filter

```
(define (stream-filter pred str)
    (if (pred (stream-car str))
        (cons-stream (stream-car str)
                     (stream-filter pred
                                    (stream-cdr str)))
        (stream-filter pred
                       (stream-cdr str))))
```

# Decoupling Order of Evaluation

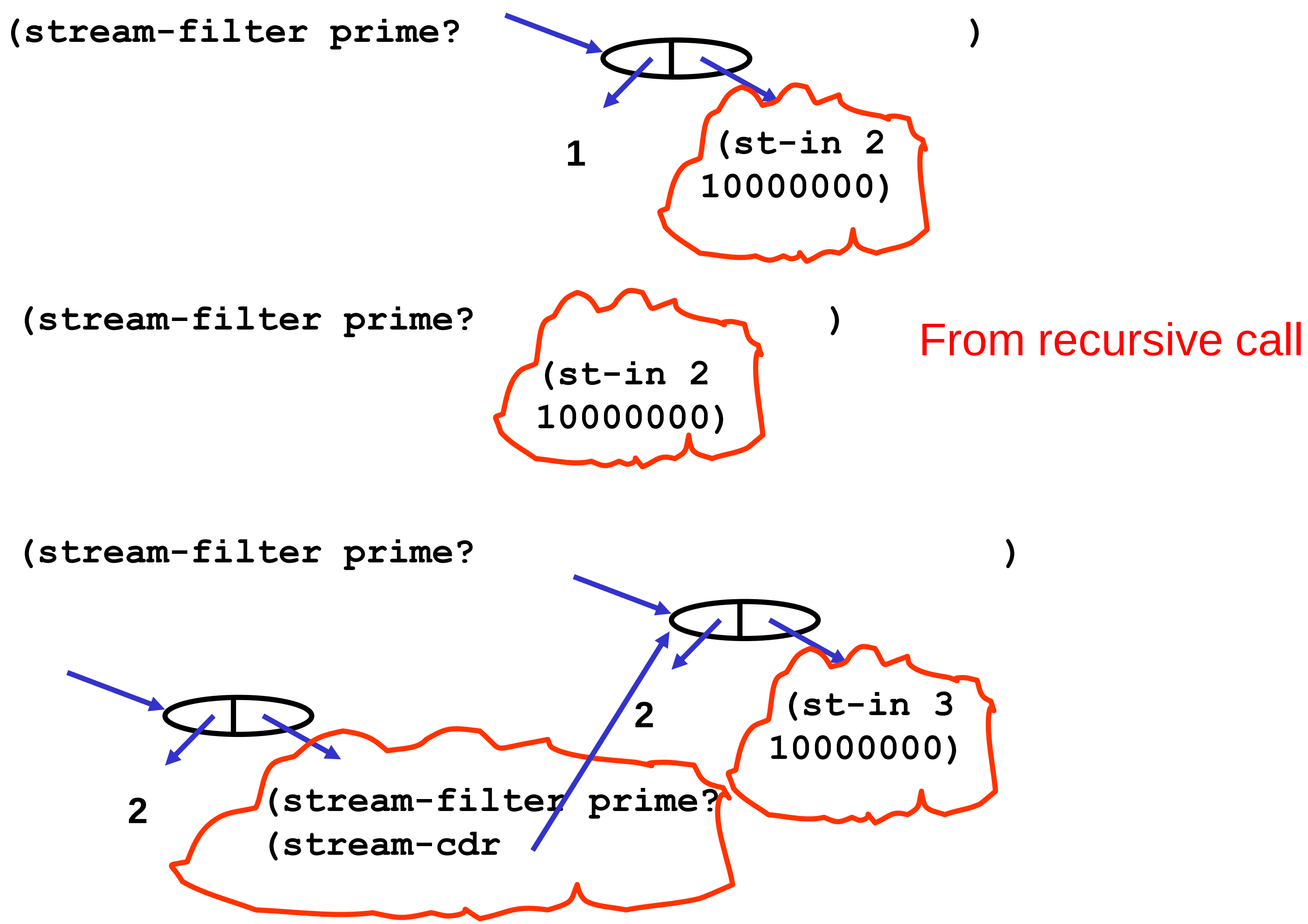
```
(stream-ref
   (stream-filter (lambda (x) (prime? x))
                  (stream-interval 2 100000000))
   100)
```

Creates 1 element, plus a promise

Creates 1 element, plus a promise

# Decoupling Order of Evaluation

```
(stream-filter prime? (str-in 1 100000000))
```

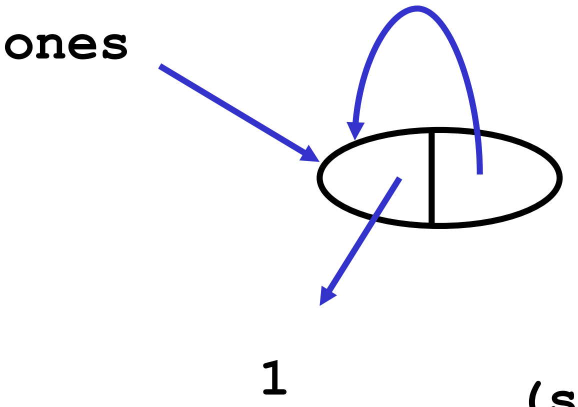


# One Possibility: Infinite Data Structures!

- Some very interesting behavior

```
(define ones (cons 1 ones))

(define ones (cons-stream 1 ones))
(stream-car (stream-cdr ones)) => 1
```




The infinite stream of 1's!

**ones: 1 1 1 1 1 1 ....**

```
(stream-ref ones 1) ➔1
(stream-ref ones 1000) ➔1
(stream-ref ones 1000000) ➔ 1
```

# Finite list procs turn into infinite stream procs

```
(define (add-streams s1 s2)
  (cond ((empty-stream? s1) the-empty-stream)
        ((empty-stream? s2) the-empty-stream)
        (else (cons-stream
                (+ (stream-car s1) (stream-car s2))
                (add-streams (stream-cdr s1)
                             (stream-cdr s2))))))
```

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

```
ones:   1 1 1 1 1 1 ....
ints:   1 2 3 ...
```

```
add-streams ones
            ints
```

```
add-streams (str-cdr ones)
            (str-cdr ints)
```

# Finding all the primes

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
| 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 |
| 41 | 42 | 43 | 44 | 45 | 46 | 47 | 48 | 49 | 50 |
| 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 |
| 61 | 62 | 63 | 64 | 65 | 66 | 67 | 68 | 69 | 60 |
| 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

# Using a sieve

```
(define (sieve str)
  (cons-stream
    (stream-car str)
    (sieve (stream-filter
             (lambda (x)
               (not (divisible? x (stream-car str))))
             (stream-cdr str)))))

(define primes
    (sieve (stream-cdr ints)))
```

( 2 sieve (filter ints 2) )

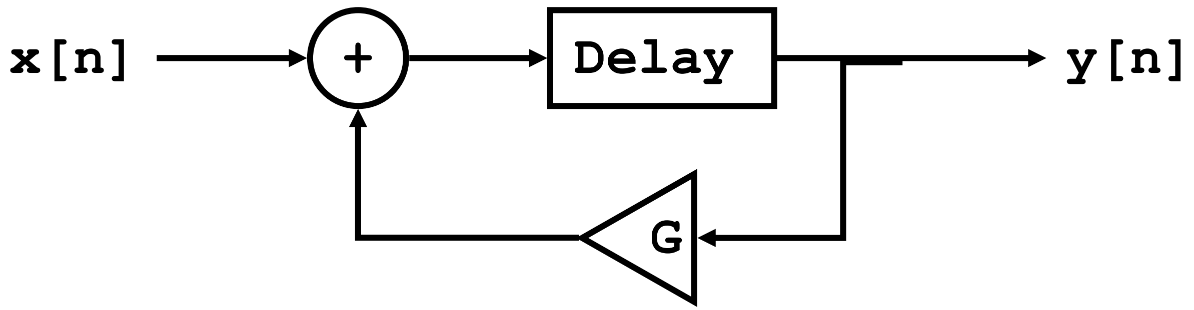
(2 3 sieve (filter

sieve (filter ints 2)

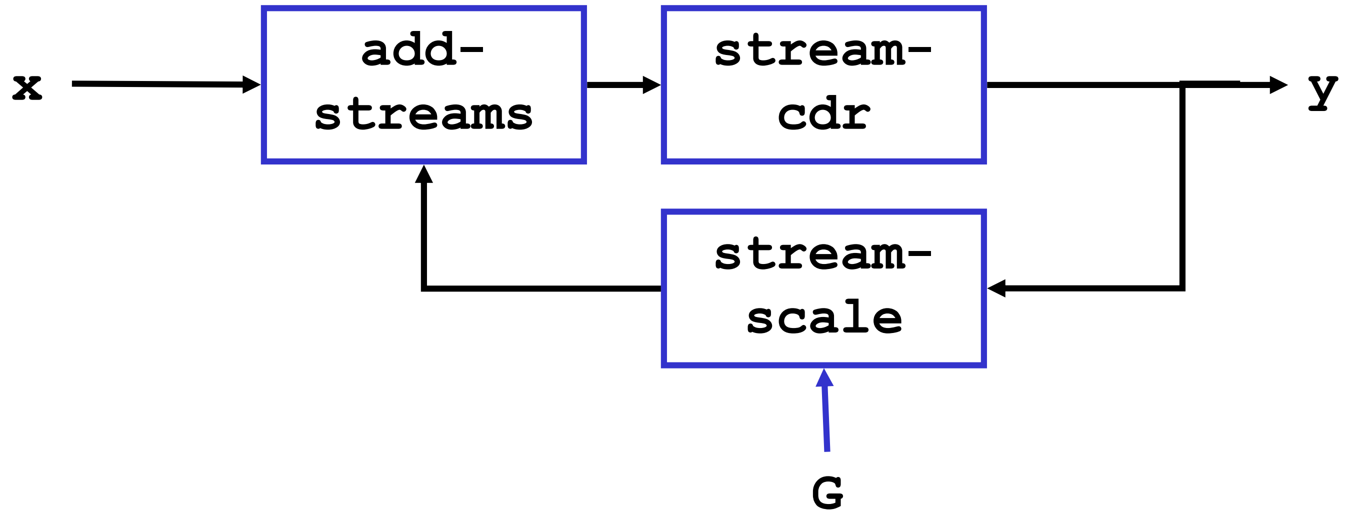
3))

# Streams Programming

- Signal processing:



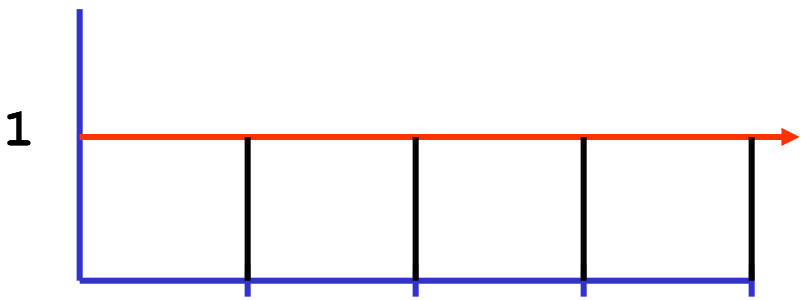

- Streams model:



# Integration as an example

```
(define (integral integrand init dt)
   (define int
       (cons-stream
          init
          (add-streams (stream-scale dt integrand)
                       int)))
   int)
```

```
(integral ones 0 2)
 =>    0  2  4  6  8
Ones:  1  1  1  1  1
Scale  2  2  2  2  2
```



# An example: power series

$g(x) = g(0) + x\, g'(0) + x^2/2\, g''(0) + x^3/3!\, g'''(0) + \ldots$

For example:

$\cos(x) = 1 - x^2/2 + x^4/24 - \ldots$

$\sin(x) = x - x^3/6 + x^5/120 - \ldots$

# An example: power series

**Think about this in stages, as a stream of values**

```
(define (powers x)
   (cons-stream 1
                (scale-stream x (powers x))))
```

$\Rightarrow$ **1 x $x^2$ $x^3$ …**

Think of (powers x) as giving all the powers of x starting at 1, then whole expression gives all the powers starting at x

```
(define facts
   (cons-stream 1
          (mult-streams (stream-cdr ints) facts)))
```

**=> 1 2 6 24 …**

Think of facts as stream whose nth element is n!, then multiplying these two streams together gives a stream whose nth element is (n+1)!

# An example: power series

```
(define (series-approx coeffs)
   (lambda (x)
     (mult-streams
        (div-streams (powers x) (cons-stream 1 facts))
        coeffs)))
```

$g(x) = g(0) + x\,g'(0) + x^2/2\;g''(0) + x^3/3!\;g'''(0) + \ldots$

```
(define (stream-accum str)
  (cons-stream (stream-car str)
               (add-streams (stream-accum str)
                            (stream-cdr str))))
```

$\Rightarrow g(0)$

$\Rightarrow g(0) + x\,g'(0)$

$\Rightarrow g(0) + x\,g'(0) + x^2/2\;g''(0)$

$\Rightarrow g(0) + x\,g'(0) + x^2/2\;g''(0) + x^3/3!\;g'''(0)$

# An example: power series

```
(define (power-series g)
  (lambda (x)
    (stream-accum ((series-approx g) x))))

(define sine-coeffs
  (cons-stream 0
    (cons-stream 1
      (cons-stream 0
        (cons-stream –1 sine-coeffs)))))

(define cos-coeffs (stream-cdr sine-coeffs))

(define (sine-approx x)
   ((power-series sine-coeffs) x))
(define (cos-approx x)
   ((power-series cos-coeffs) x))
```

# Using streams to decouple computation

- Here is our old SQRT program

```
(define (sqrt x)
   (define (try guess)
      (if (good-enough? Guess)
          guess
          (try (improve guess))))
   (define (improve guess)
      (average guess (/ x guess)))
   (define (good-enough? Guess)
      (close? (square guess) x))
   (try 1))
```

- Unfortunately, it intertwines stages of computation

# Using streams to decouple computation

- So let's pull apart the idea of generating estimates of a sqrt from the idea of testing those estimates

```
(define (sqrt-improve guess x)
  (average guess (/ x guess)))
(define (sqrt-stream x)
  (cons-stream
     1.0
     (stream-map (lambda (g) (sqrt-improve g x))
                 (sqrt-stream x))))
(print-stream (sqrt-stream 2))

1.0  1.5  1.4166666666666665  1.4142156862745097
          1.4142135623745899  1.414213562373095
          1.414213562373095
```

**Note how fast it converges!**

# Using streams to decouple computation

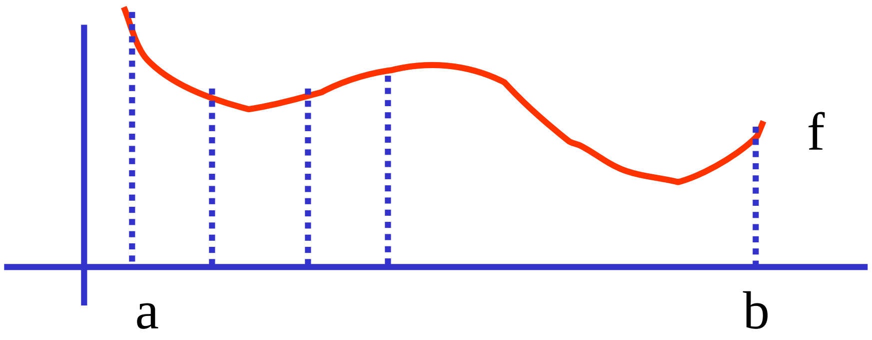
- That was the generate part, here is the test part…

```
(define (stream-limit s tol)
  (define (iter s)
     (let ((f1 (stream-car s))
           (f2 (stream-car (stream-cdr s))))
       (if (close-enough? f1 f2 tol)
           f2
           (iter (stream-cdr s)))))
   (iter s))

(stream-limit (sqrt-stream 2) 1.0e-5)
;Value: 1.414213562373095
```

- This reformulates the computation into two distinct stages: generate estimates and test them.

# Do the same trick with integration




```
(trapezoid f 0 4 0.1)
```

```
(define (trapezoid f a b h)
  (let ((dx (* (- b a) h))
        (n (/ 1 h)))
    (define (iter j sum)
         (if (>= j n)
             sum
             (iter (+ j 1) (+ sum (f (+ a (* j dx)))))))
    (* dx (iter 1 (+ (/ (f a) 2)
                     (/ (f b) 2))))))
```

# Do the same trick with integration

```
(define (witch x) (/ 4 (+ 1 (* x x))))

(trapezoid witch 0 1 0.1)

;Value: 3.1399259889071587

(trapezoid witch 0 1 0.01)

;Value: 3.141575986923129
```

- So this gives us a good approximation to pi, but quality of approximation depends on choice of trapezoid size.  What happens if we let h → 0??

# Accelerating a decoupled computation

```
(define (keep-halving R h)
   (cons-stream
     (R h)
     (keep-halving R (/ h 2))))

(print-stream
   (keep-halving
     (lambda (h) (trapezoid witch 0 1 h))
     0.1))
3.13992598890715
3.14117598695412
3.14148848692361
3.14156661192313
3.14158614317312
3.14159102598562
3.14159224668875
3.14159255186453
3.14159262815847
3.14159265723195
```

Convergence – getting about 1 new digit each time, but each line takes twice as much work as the previous one!!

```
(stream-limit (keep-halving
                 (lambda (h) (trapezoid witch 0 1 h))
                  .5)
              1.0e-9)
;Value: 3.14159265343456 - takes 65,549 evaluations of witch
```

# Summary

- Lazy evaluation – control over evaluation models
  - Convert entire language to normal order
  - Upward compatible extension
    - lazy & lazy-memo parameter declarations

- Streams programming:
  a powerful way to structure and think about computation