

# Streams

- Streams Interface
- Higher-Order Procedures
- Implementing Streams
- Infinitely Long Streams

# Sum of Odd Leaves

```
(define (sum-odd-squares tree)
  (if (leaf-node? tree)
      (if (odd? tree)
          (square tree)
          0)
      (+ (sum-odd-squares (left-branch tree))
         (sum-odd-squares (right-branch tree)))))
```

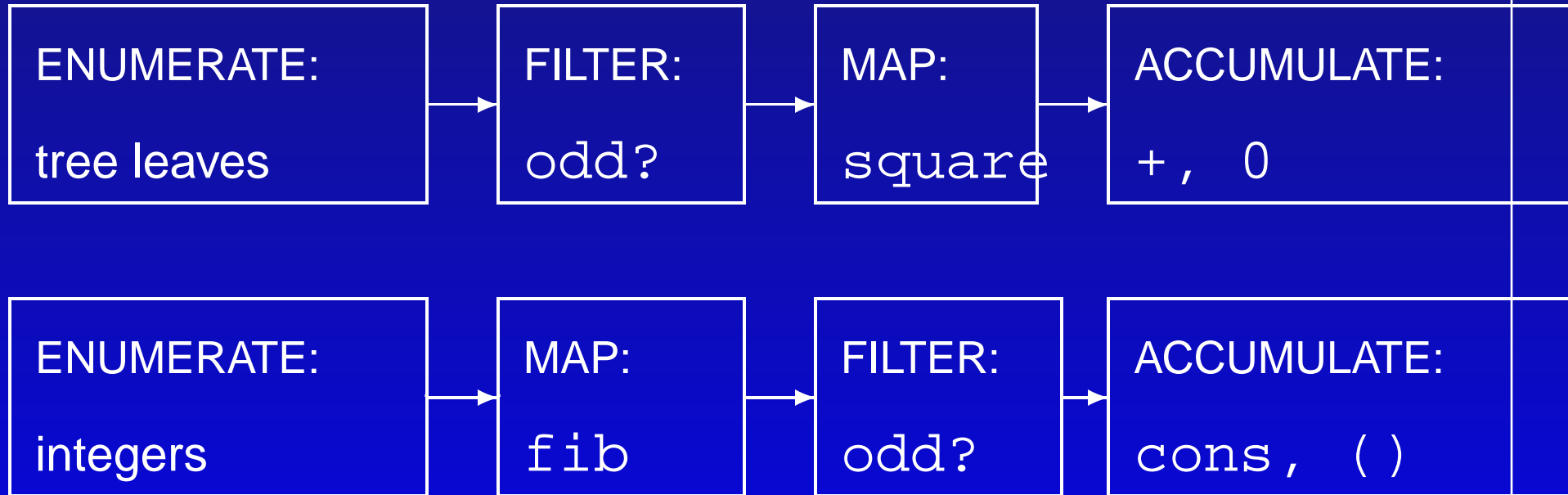
- enumerate the leaves of a tree
- filter them, selecting odd leaves
- squares selected leaves
- accumulates the results

# List of Fibonacci Numbers

```
(define (odd-fibs n)
  (define (next k)
    (if (> k n) '()
        (let ((f (fib k)))
          (if (odd? f)
              (cons f (next (1+ k)))
              (next (1+ k))))))
  (next 1))
```

- enumerate the integers from 1 to  $n$
- computes Fibonacci number for each integer
- filters them, leaving odd ones
- accumulates results into a list

# Similarities?



# Stream Operations

- Constructor `cons-stream`
- Selectors `stream-car` and `stream-cdr`
  - `(stream-car (cons-stream a b)) → a`
  - `(stream-cdr (cons-stream a b)) → b`

# Stream Operations

- Very similar to `cons`, `car`, and `cdr`
- Object called `the-empty-stream` (Similar to the empty list)
- Predicate `stream-null?` (Similar to `null?`)

# Two Enumerations

## Tree Enumeration

```
(define (enumerate-tree tree)
  (if (leaf-node? tree)
      (cons-stream tree the-empty-stream)
      (append-streams
         (enumerate-tree
          (left-branch tree))
         (enumerate-tree
          (right-branch tree)))))
```

# Two Enumerations

## Integer Enumeration

```
(define (enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low
                    (enumerate-interval
                     (1+ low) high))))
```

## Operations missing from the text:

```
(define leaf-node? number?)
```

```
(define left-branch car)
```

```
(define right-branch cdr)
```

## One of Two Maps

```
(define (map-square s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream
        (square (stream-car s))
        (map-square (stream-cdr s)))))
```

## One of Two Maps

```
(define (map-fib s)
  (if (stream-null? s)
      the-empty-stream
      (cons-stream
        (fib (stream-car s))
        (map-fib (stream-cdr s)))))
```

## One Filter

```
(define (filter-odd s)
  (cond ((stream-null? s) the-empty-stream)
        ((odd? (stream-car s))
         (cons-stream (stream-car s)
                       (filter-odd (stream-cdr s))))
        (else
         (filter-odd (stream-cdr s)))))
```

# Cooperative Exercises

- Write `accumulate-+`
- Write `accumulate-cons`

# Cooperative Exercises—Solutions

```
(define (accumulate+ s)
  (if (stream-null? s)
      0
      (+ (stream-car s)
         (accumulate+ (stream-cdr s)))))
```

```
(define (accumulate-cons s)
  (if (stream-null? s)
      '()
      (cons (stream-car s)
            (accumulate-cons
             (stream-cdr s)))))
```

# Rewriting Original Procedures with Streams

```
(define (sum-odd-squares tree)
  (accumulate-+
    (map-square
      (filter-odd
        (enumerate-tree tree))))))
```

```
(define (odd-fibs n)
  (accumulate-cons
    (filter-odd
      (map-fib
        (enumerate-interval 1 n))))))
```

# Reusing Procedures

```
(define (list-square-fibs n)
  (accumulate-cons
    (map-square
      (map-fib
        (enumerate-interval 1 n))))))
```

# Higher-Order Procedures (again)

- `accumulate+` and `accumulate-cons`
- `map-square` and `map-fib`
- Remember `map`?

# Accumulate as a Higher-Order Procedure

```
(define (accumulate combiner initial-value stream)
  (if (stream-null? stream)
      initial-value
      (combiner (stream-car stream)
                (accumulate combiner
                            initial-value
                            (stream-cdr stream))))))
```

# Filter as a Higher-Order Procedure

```
(define (stream-filter pred stream)
  (cond ((stream-null? stream) the-empty-stream)
        ((pred (stream-car stream))
         (cons-stream (stream-car stream)
                       (stream-filter pred
                                       (stream-cdr stream))))
        (else (stream-filter pred (stream-cdr stream)))))
```

# Cooperative Exercise

Write `stream-map` as a higher-order procedure.

# Stream-Map as a Higher-Order Procedure

```
(define (stream-map proc stream)
  (if (stream-null? stream)
      the-empty-stream
      (cons-stream (proc (stream-car stream))
                   (stream-map proc
                               (stream-cdr stream)))))
```

# Cooperative Exercise

Create a procedure to find the product of the squares of the odd elements in a stream.

## Cooperative Exercise—Solution

```
(define
  (product-of-squares-of-odd-elements s)
  (accumulate *
              1
              (stream-map square
                          (stream-filter odd? s))))
```

# Streams

- Streams Interface
- Higher-Order Procedures
- **Implementing Streams**
- Infinitely Long Streams

# Streams = Lists?

- Write a program to compute the sum of all prime numbers in an interval.
- Iterative Version

```
(define (sum-primes a b)
  (define (iter count accum)
    (cond ((> count b) accum)
          ((prime? count) (iter (1+ count)
                                 (+ count accum)))
          (else (iter (1+ count) accum))))
  (iter a 0))
```

# Streams = Lists?

- Streams Version

```
(define (sum-primes a b)
  (accumulate +
              0
              (stream-filter prime?
                              (enumerate-interval a b))))
```

- How are the the iterative and streams versions different?
- Would the streams version work well if streams were really lists?

## How bad can it get?

- Compute the second prime in the interval 10,000 to 1,000,000.

```
(stream-car  
  (stream-cdr  
    (stream-filter prime?  
      (enumerate-interval 10000 1000000))))))
```

- We have to store almost a million elements more than the equivalent iterative version if streams were really lists!

## A New Idea

- Construct elements of a stream only when required...
- Stream-Cdr of a stream is evaluated only when accessed by `stream-cdr` procedure
- Result is the interleaving of the construction with the use

# Differences between Streams and Lists

**Lists** `car` and `cdr` are evaluated at *construction* time

**Streams** `stream-cdr` is evaluated at *selection* time

- As a *data abstraction* they are the same!
- There are versions of Lisp which implement lists as streams.

## A New Special Form—`delay`

- `(delay <exp>)` returns a *delayed object*
  - We'll discuss the implementation of `delay` in a little bit.
- `force` performs evaluation on a delayed object

```
(define (force delayed-object)
  (delayed-object))
```

# Defining Stream Operations

- `cons-stream` is a special form
  - `(cons-stream a b)` is equivalent to `(cons a (delay b))`
  - Why is `cons-stream` a special form?

```
(define (stream-car stream) (car stream))
```

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

# Was our prime computation as bad as we thought?

```
(stream-car  
  (stream-cdr  
    (stream-filter prime?  
      (enumerate-interval 10000 1000000))))))
```

# Was our prime computation as bad as we thought?

- Remember:

```
(define (enumerate-interval low high)
  (if (> low high)
      the-empty-stream
      (cons-stream low
                   (enumerate-interval (1+ low) high))))
```

- Evaluated as:

```
(cons 10000
      (delay (enumerate-interval
              10001 1000000)))
```

## Implementing delay

- `(delay <exp>)` *could* be considered as syntactic sugar for `(lambda () <exp>)`.
- `(define s (enumerate-interval 1 3))`
- Take `(stream-car (stream-cdr s))`

```
(enumerate-interval 1 3)
```

```
(cons-stream 1 (enumerate-interval 2 3))
```

```
(cons 1 (delay (enumerate-interval 2 3)))
```

```
(cons 1 (lambda () (enumerate-interval 2 3)))
```

## Evaluation of (stream-cdr s)

```
(stream-cdr s)
(stream-cdr (cons 1 (lambda () (enumerate-interval 2 3))))
(force (cdr (cons 1 (lambda () (enumerate-interval 2 3)))))
(force (lambda () (enumerate-interval 2 3)))
((lambda () (enumerate-interval 2 3)))
(enumerate-interval 2 3)
(cons-stream 2 (enumerate-interval 3 3))
(cons-stream 2 (delay (enumerate-interval 3 3)))
(cons 2 (lambda () (enumerate-interval 3 3)))
```

## Evaluation of `(stream-cdr s)`

- `(stream-cdr s)` is a pair whose `car` is 2 and whose `cdr` is the procedure `(lambda () (enumerate-interval 3 3))`.
- Since `stream-car` is just `car`, `(stream-car (stream-cdr s))` is 2.

# Improved delay

- We don't want to force the same delayed object multiple times.
- How do we not force the same delayed object multiple times?
- MEMOIZATION (sounds strange, but there is no typo here)

# Improved delay

```
(define (memo-proc proc)
  (let ((already-run? nil) (result nil))
    (lambda ()
      (if (not already-run?)
          (sequence (set! result (proc))
                    (set! already-run? (not nil))
                    result)
          result))))
```

- Now define `delay` as `(memo-proc (lambda () <exp>))`

## Something to think about...

Use the environment model to evaluate:

```
(define (fact-3) (factorial 3))  
(fact-3)  
(fact-3)
```

and:

```
(define memoized-fact-3 (memo-proc fact-3))  
(memoized-fact-3)  
(memoized-fact-3)
```

# Streams

- Streams Interface
- Higher-Order Procedures
- Implementing Streams
- **Infinitely Long Streams**

# Infinitely Long Streams

- Delayed evaluation
  - Manipulate streams as complete entities
  - Only compute as much as we need to access
- Very long sequences can be represented as streams
- Infinite sequences can be represented as streams

# Infinitely Long Streams

- Constructor

`cons-stream`

- Selectors

`stream-car`, `stream-cdr`

- Higher order procedures

`stream-filter`, `stream-map`

- Memoization

# Infinitely Long Streams

- For instance

```
(define (integers-starting-from n)
  (cons-stream n
    (integers-starting-from (1+ n))))
```

```
(define integers
  (integers-starting-from 1))
```

- What is the `stream-car` of `integers`?
- What is the `stream-cdr` of `integers`?

# Infinitely Long Streams

- The “Sieve of Erasthones”

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

## Sieve, Continued

```
(define primes  
  (sieve (integers-starting-from 2)))
```

# Infinitely Long Streams

- The “Sieve of Erasthones”

```
(define primes
```

```
  (sieve (integers-starting-from 2)))
```

## Exercise

- The following procedure returns a list of all distinct symbols in a given list of symbols

```
(define (remove-duplicates lst)
  (cond ((null? lst) '())
        ((null? (memq (car lst) (cdr lst)))
         (cons (car lst)
               (remove-duplicates (cdr lst))))
        (else (remove-duplicates (cdr lst)))))
```

# Exercise

- Recall

```
(define (memq item x)
  (cond ((null? x) '())
        ((eq? item (car x)) x)
        (else (memq item (cdr x)))))
```

# Exercise

- Idea: to convert to streams,
  - `cons`  $\Rightarrow$  `cons-stream`
  - `car`  $\Rightarrow$  `stream-car`
  - `cdr`  $\Rightarrow$  `stream-cdr`
  - `null?`  $\Rightarrow$  `stream-null?`
  - `'()`  $\Rightarrow$  `the-empty-stream`

## Exercise

- What is wrong with

```
(define (remove-duplicates str)
  (cond ((stream-null? str) the-empty-stream)
        ((stream-null? (memq (stream-car str)
                              (stream-cdr str)))
         (cons-stream (stream-car str)
                       (remove-duplicates (stream-cdr str))))
        (else (remove-duplicates (stream-cdr str)))))
```

```
(define (memq item x)
  (cond ((null? x) '())
        ((eq? item (stream-car x)) x)
        (else (memq item (stream-cdr x)))))
```

# Exercise

- Complete

```
(define (remove-duplicates stream)
  (cons-stream
    < first - element >
    (stream-filter < predicate >
      < stream - to - filter >)))
```

# Defining Streams Implicitly

- What is this stream?

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

# Defining Streams Implicitly

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

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

# Defining Streams Implicitly

```
(define primes
  (cons-stream 2
    (stream-filter
      prime?
      (integers-starting-from 3))))

(define (prime? n)
  (define (iter ps)
    (cond ((> (square (stream-car ps)) n) t)
          ((divisible? n (stream-car ps)) nil)
          (else (iter (stream-cdr ps)))))
  (iter primes))
```

# Exercise

- Complete the following

```
(define integers
  (cons-stream 1
    (stream-map < expr > integers)))
```

## Exercise

- Use `filter` to define the stream of all integers that are not divisible by 2, 3, or 5.

## Exercise

- Enumerate, in ascending order, all positive integers with no prime factors other than 2, 3, or 5.
  - $S$  begins with 1
  - The elements of `(scale-stream 2 S)` are also elements of  $S$
  - The same is true for `(scale-stream 3 S)` and `(scale-stream 5 S)`

# Exercise

- These are all the elements of  $S$
- All we have to do is combine the elements from these four sources

```
(define S
  (cons-stream 1
    (merge < exp1 > < exp2 >)))
```

# Exercise

```
(define (merge s1 s2)
  (cond ((stream-null? s1) s2)
        ((stream-null? s2) s1)
        (else
         (let ((h1 (stream-car s1))
               (h2 (stream-car s2)))
           (cond ((< h1 h2)
                  (cons-stream h1
                                (merge (stream-cdr s1) s2)))
                 ((> h1 h2)
                  (cons-stream h2
                                (merge s1 (stream-cdr s2))))
                 (else
                  (cons-stream h1
                                (merge (stream-cdr s1)
                                        (stream-cdr s2))))))))))
```

# Streams as Signals

- Motivated streams as computational analogs of signals
- We can use streams to model signal processing in a very direct way
- One fundamental signal processing element is an *integrator*

$$S(t) = C + \int_0^t x(t)dt$$

# Streams as Signals

- Input stream  $x = (x_i)$
- Initial value  $C$
- Increment  $dt$
- Output value  $S = (S_i)$

$$S_i = C + \sum_{j=1}^i x_j dt$$

# Streams as Signals

- This leads naturally to

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

- Note the feedback loop!

# Streams and Delayed Evaluation

- Wait a minute!
- Isn't this just like  

```
(define balance (- balance amount))
```
- Why can streams be defined implicitly and other data not?

# Streams and Delayed Evaluation

- The interpreter can deal with implicit definition because of the `delay` that is incorporated into `cons-stream`
- In general, `delay` is crucial for using streams to model systems with loops
- We may, in fact, have to use `delay` explicitly

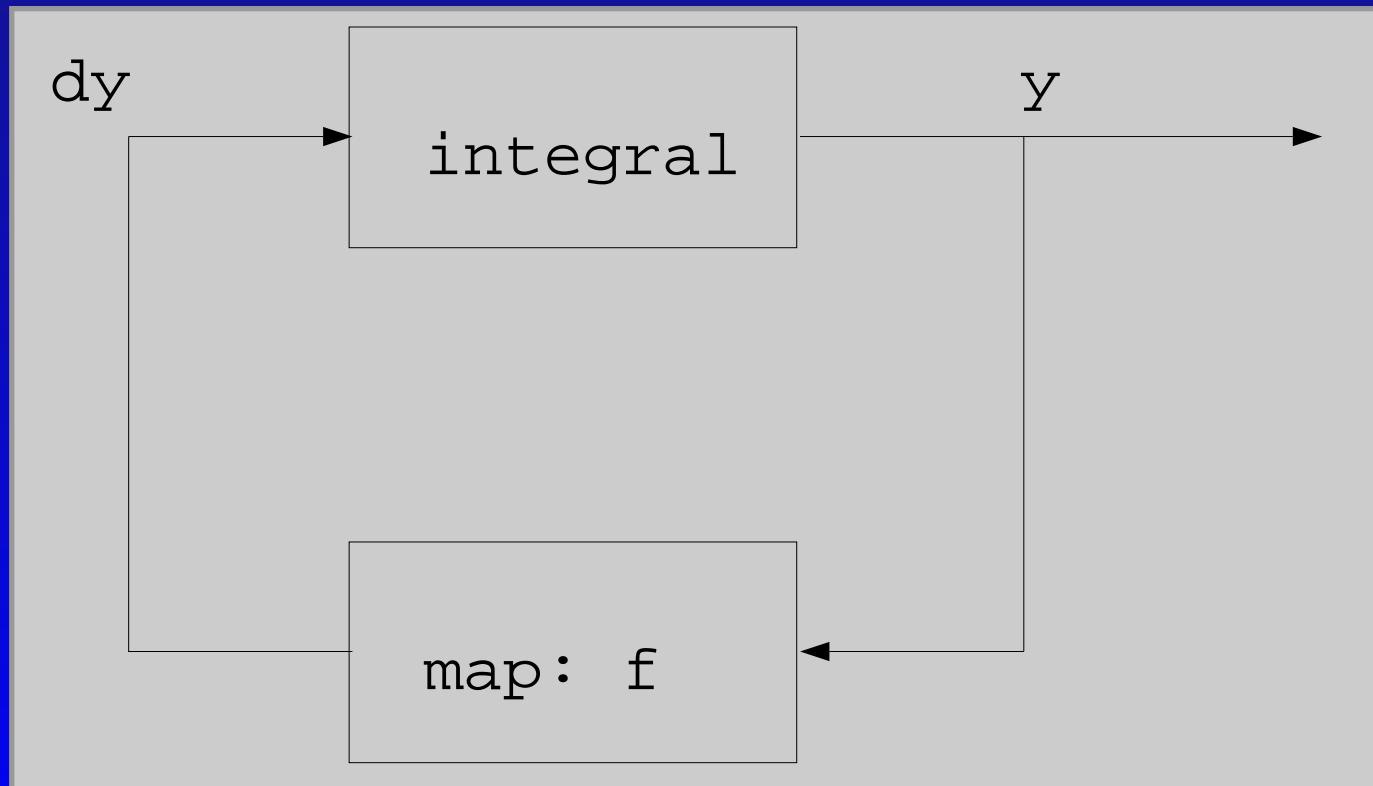
# Streams and Delayed Evaluation

- Suppose we want to solve

$$\frac{dy}{dt} = f(y)$$

where  $f$  is a given mathematical function

# Streams and Delayed Evaluation



# Streams and Delayed Evaluation

- What is wrong with

```
(define (solve f y-init dt)
  (define y (integral dy y-init dt))
  (define dy (stream-map f y))
  y)
```

# Streams and Delayed Evaluation

- Can we fix it with

```
(define (solve f y-init dt)
  (define dy (stream-map f y))
  (define y (integral dy y-init dt))
  y)
```

# Streams and Delayed Evaluation

- Explain

# Streams and Delayed Evaluation

- Let's think about `integral` more carefully
- We can generate part of the stream without knowing everything about the arguments (similar to `cons-stream`)
- That is, we can generate the stream-car of the stream (the initial value) without having to evaluate the integrand
- Redefine `integral` to have delayed integrand

# Streams and Delayed Evaluation

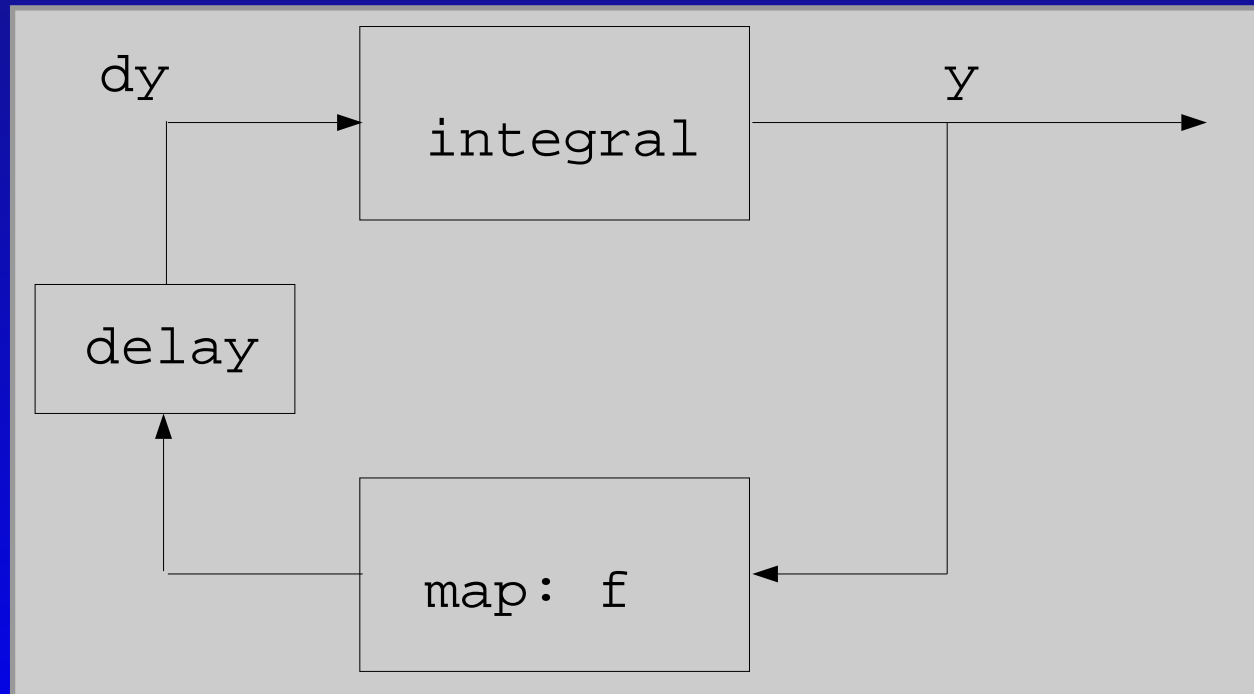
```
(define (integral delayed-integrand initial-value dt)
  (define int
    (cons-stream
      initial-value
      (let ((integrand (force delayed-integrand)))
        (add-streams (scale-stream dt integrand)
                     int))))
  int)
```

# Streams and Delayed Evaluation

- We can define `solve` as

```
(define (solve f y-init dt)
  (define y
    (integral (delay dy) y-init dt))
  (define dy (stream-map f y))
  y)
```

# Streams and Delayed Evaluation



# Nested Mappings over Infinite Streams

- Consider the following procedure which generates all pairs of elements from two streams

```
(define (pairs s1 s2)
  (collect (list i j)
            ((i s1)
             (j s2))))
```

# Nested Mappings over Infinite Streams

- This is equivalent to

```
(define (pairs s1 s2)
  (flatmap
    (lambda (i)
      (stream-map (lambda (j) (list i j)) S2))
    s1))
```

# Nested Mappings over Infinite Streams

- Problems because of `flatten`

```
(define (flatmap f s)
  (flatten (stream-map f s)))
```

```
(define (flatten stream)
  (accumulate append-streams
    the-empty-stream stream))
```

# Nested Mappings over Infinite Streams

- The order in which `flatten` collects elements to form the output stream
- Our use of `accumulate` in this context requires additional delayed evaluation to work properly

# Nested Mappings over Infinite Streams

- Consider

`(pairs integers integers)`

- `pairs` will run through all pairs of integers with  $i$  equal to 1
- We need to devise a better order of collection

## Nested Mappings over Infinite Streams

- Use `interleave` instead of `append-streams`

```
(define (interleave s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream
        (stream-car s1)
        (interleave s2
                    (stream-cdr s1))))))
```

```
(define (flatten stream)
  (accumulate interleave the-empty-stream s))
```

# Nested Mappings over Infinite Streams

- Use `interleave` instead of `append-streams`

```
(define (append-streams s1 s2)
  (if (stream-null? s1)
      s2
      (cons-stream
        (stream-car s1)
        (append-streams
          (stream-cdr s1) s2))))
```

# Nested Mappings over Infinite Streams

- Recall the definition of `accumulate`

```
(define (accumulate combiner initial-value stream)
  (if (stream-null? stream)
      initial-value
      (combiner (stream-car stream)
                (accumulate combiner
                            initial-value
                            (stream-cdr stream)))))
```

- What happens if `stream` is infinite?

# Nested Mappings over Infinite Streams

- Compare to `sieve`

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

# Nested Mappings over Infinite Streams

- We need a new version of `accumulate`

```
(define (accumulate-delayed combiner initial-value stream)
  (if (stream-null? stream)
      initial-value
      (combiner (stream-car stream)
                (delay
                 (accumulate-delayed
                  combiner
                  initial-value
                  (stream-cdr stream)))))))
```

## Nested Mappings over Infinite Streams

- We also need a new version of `interleave`

```
(define (interleave-delayed s1 delayed-s2)
  (if (stream-null? s1)
      (force delayed-s2)
      (cons-stream
        (stream-car s1)
        (interleave-delayed
          (force delayed-s2)
          (delay (stream-cdr s1)))))))
```

# Nested Mappings over Infinite Streams

- Final version of `flatten`

```
(define (flatten stream)
  (accumulate-delayed
    interleave-delayed
    the-empty-stream
    stream))
```

# Exercise

- Generate the stream of all positive integers that can be expressed as the sum of two cubes in two different ways.
- The first such number is 1,729.
- What is the second?

# Coda

- To model systems with local state
  - Assignment
  - Mutable data
- To model temporal systems
  - Streams

# Coda

- Recall

```
(define
  (make-simplified-withdraw balance)
  (lambda (amount)
    (set! balance
      (- balance amount))
    balance))
```

# Coda

- Consider

```
(define
  (stream-withdraw balance amount-stream)
  (cons-stream
    balance
    (stream-withdraw
      (- balance
          (stream-car amount-stream))
      (stream-cdr amount-stream))))
```

# Coda

- Do we need assignment after all?