# Today’s topic: Abstraction

- Compound Data

- *Data* Abstractions:
    - Isolate *use* of data abstraction from details of *implementation*

- Relationship between data abstraction and procedures that operate on it

# Compound data

- Need a way of (procedure for) gluing data elements together into a unit that can be treated as a simple data element
- Need ways of (procedures for) getting the pieces back out
- Need a contract between the “glue” and the “unglue”

• Ideally want the result of this “gluing” to have the property of **closure:**

  - “the result obtained by creating a compound data structure can itself be treated as a primitive object and thus be input to the creation of another compound object”

# Pairs (cons cells)

- **(cons <x-exp> <y-exp>) ==> <P>**
  - Where **<x-exp>** evaluates to a value **<x-val>**, and **<y-exp>** evaluates to a value **<y-val>**
  - **Returns a** pair **<P>** whose car-part is **<x-val>** and whose cdr-part is **<y-val>**
- **(car <P>) ==> <x-val>**
  - Returns the car-part of the pair **<P>**
- **(cdr <P>) ==> <y-val>**
  - Returns the cdr-part of the pair **<P>**

# Pairs Are A Data Abstraction

- Constructor

```
; cons: A,B -> A X B

; cons: A,B -> Pair<A,B>
  (cons <x> <y>) ==> <P>
```

- Accessors

```
; car: Pair<A,B> -> A
  (car <P>) ==> <x>

; cdr: Pair<A,B> -> B
  (cdr <P>) ==> <y>
```

- Contract

```
; (car (cons <a> <b> )) ➔ <a>
; (cdr (cons <a> <b> )) ➔ <b>
```

- Operations

```
; pair? anytype -> boolean
  (pair? <z>)
    ==> #t if <z> evaluates to a pair, else #f
```

# Pair Abstraction

- Once we build a pair, we can treat it as if it were a primitive (e.g. the same way we treat a number)
- Pairs have the property of closure, meaning we can use a pair anywhere we would expect to use a primitive data element :
  - *(cons (cons 1 2) 3)*

# Elements of a Data Abstraction

## -- Pair Abstaction --

1. Constructor

```
;cons: A, B → Pair<A,B>; A & B = anytype
(cons <x> <y>) → <p>
```

2. Accessors

```
(car <p>)        ; car: Pair<A,B> → A
(cdr <p>)        ; cdr: Pair<A,B> → B
```

3. Contract

```
(car (cons <x> <y>)) → <x>
(cdr (cons <x> <y>)) → <y>
```

4. Operations

```
; pair?: anytype → boolean
(pair? <p>)
```

5. Abstraction Barrier

IGNORANCE NEED TO KNOW

6. Concrete Representation & Implementation

```
Could have alternative implementations!
```

# Rational Number Abstraction

- A rational number is a ratio n/d
- a/b + c/d = (ad + bc)/bd
  - 2/3 + 1/4 = (2*4 + 3*1)/12 = 11/12
- a/b * c/d = (ac)/(bd)
  - 2/3 * 1/3 = 2/9

# Rational Number Abstraction

1. Constructor

```
; make-rat: integer, integer -> Rat
(make-rat <n> <d>) -> <r>
```

2. Accessors

```
; numer, denom: Rat -> integer
(numer <r>)
(denom <r>)
```

3. Contract

```
(numer (make-rat <n> <d>)) ==> <n>
(denom (make-rat <n> <d>)) ==> <d>
```

4. Operations

```
(print-rat <r>) prints rat
(+rat x y) ; +rat: Rat, Rat -> Rat
(*rat x y) ; *rat: Rat, Rat -> Rat
```

5. Abstraction Barrier

```
Say nothing about implementation!
```

# Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = Pair<integer,integer>
(define (make-rat n d) (cons  __ __))
(define (numer r) (____ r))
(define (denom r) (____ r))
```

# Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = Pair<integer,integer>
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

# Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = List
(define (make-rat n d) (list ___ ___))
(define (numer r) (____ r))
(define (denom r) (____ r))
```

# Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = List
(define (make-rat n d) (list  n  d  ))
(define (numer r) (car  r))
(define (denom r) (cadr r))
```

# Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = List
(define (make-rat n d) (list ___ ___))
(define (numer r) (____ r))
(define (denom r) (____ r))
```

# Rational Number Abstraction

1. Constructor
2. Accessors
3. Contract
4. Operations
5. Abstraction Barrier

6. Concrete Representation & Implementation

```
; Rat = List
(define (make-rat n d) (list  d  n  ))
(define (numer r) (cadr  r))
(define (denom r) (car r))
```

# Additional Rational Number Operations

```
; +rat: Rat, Rat -> Rat
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))

; *rat: Rat, Rat -> Rat
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

# Using our system

- `(define one-half (make-rat 1 2))`
- `(define three-fourths (make-rat 3 4))`
- `(define new (+rat one-half three-fourths))`

```
(numer new) → 10
(denom new) → 8
```

**Oops – should be 5/4 not 10/8!!**

# “Rationalizing” Implementation

```
(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

Strategy: remove common factors when access numer and denom

```
(define (numer r)
    (/ (car r) (gcd (car r) (cdr r))))

(define (denom r)
    (/ (cdr r) (gcd (car r) (cdr r))))

(define (make-rat n d)
  (cons n d))
```

# Alternative "Rationalizing" Implementation

- Strategy: remove common factors when create a rational number

```
(define (numer r) (car r))

(define (denom r) (cdr r))

(define (make-rat n d)
    (cons (/ n (gcd n d))
          (/ d (gcd n d))))

(define (gcd a b)
  (if (= b 0)
    a
    (gcd b (remainder a b))))
```

Either implementation is fine – most importantly no other code has to change if I switch from one to the other!!

# Alternative +rat Operations

```
(define (+rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

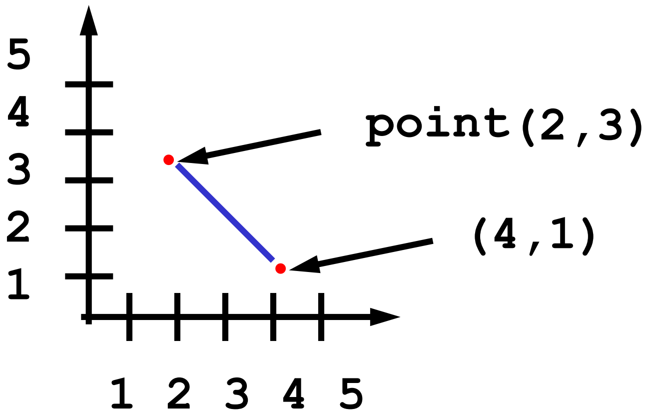
```
(define (+rat x y)
  (cons (+ (* (car x) (cdr y))
           (* (car y) (cdr x)))
        (* (cdr x) (cdr y))))
```

# Lessons learned

- Valuable to build strong abstractions
  - Hide details behind names of accessors and constructors
  - Rely on closure of underlying implementation
- Enables user to change implementation without having to change procedures that use abstraction
- Data abstractions tend to have procedures whose structure mimics their inherent structure

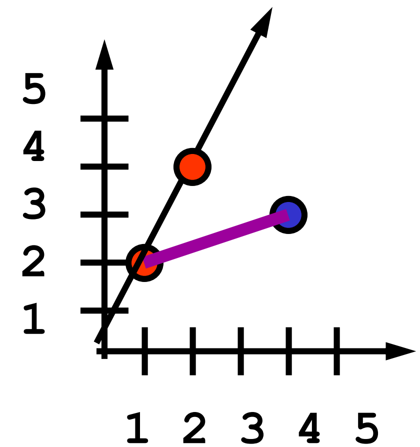
# *Building* Additional Data Abstractions

```
(define (make-point x y)
  (cons x y))

(define (point-x point)
  (car point))

(define (point-y point)
  (cdr point))

(define P1 (make-point 2 3))
(define P2 (make-point 4 1))

(define (make-seg pt1 pt2)
  (cons pt1 pt2))

(define (start-point seg)
   (car seg))

(define S1 (make-seg P1 P2))
```



# *Using* Data Abstractions

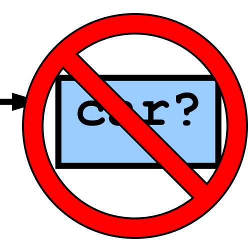
```
(define p1 (make-point 1 2))
(define p2 (make-point 4 3))
(define s1 (make-seg p1 p2))
```




```
(define stretch-point
   (lambda (pt scale)
      (make-point
         (* scale (point-x pt))
         (* scale (point-y pt)))))
```

Constructor

Selector



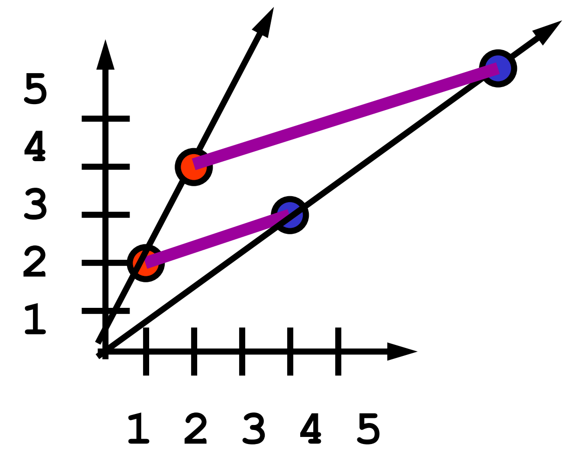
```
(stretch-point p1 2) → (2 . 4)

p1 → (1 . 2)
```

**I now have a contract for stretch-point – given a point as input, it returns a point as output – and it doesn't care about how points are created!!**

# *Using* Data Abstractions

- Generalize to other structures

```
(define stretch-seg
  (lambda (seg sc)
   (make-seg (stretch-point (start-pt seg) sc)
             (stretch-point (end-pt seg) sc))))
```




```
(define seg-length
  (lambda (seg)
    (sqrt (+ (square (- (point-x (start-point seg))
                        (point-x (end-point seg))))
             (square (- (point-y (start-point seg))
                        (point-y (end-point seg))))))))
```

Selector for point

Selector for segment

**Once again, I have a contract – given a segment as input, it returns a segment as output – and it doesn't care about how segments (or points) are created!!**

# Grouping together larger collections

- Suppose we want to group together a set of points. Here is one way

```
(cons (cons (cons (cons p1 p2)
                  (cons p3 p4))
            (cons (cons p5 p6)
                  (cons p7 p8)))
      p9)
```

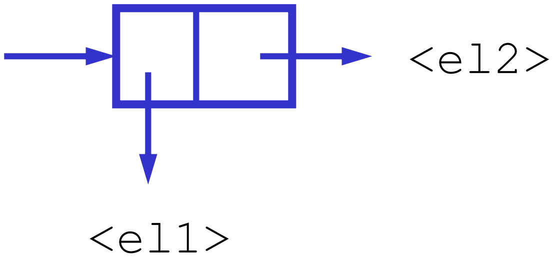
- **UGH!!** How do we get out the parts to manipulate them?

# Conventional interfaces -- Lists

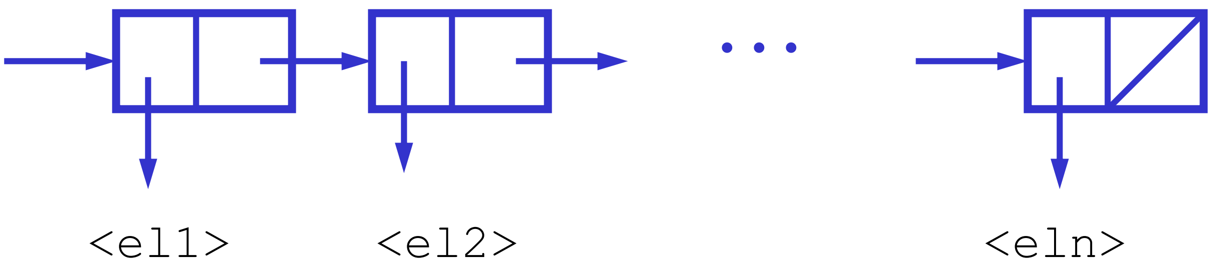
- A list is a data object that can hold an arbitrary number of ordered items.
- More formally, a list is a sequence of pairs with the following properties:
  - Car-part of a pair in sequence – holds an item
  - Cdr-part of a pair in sequence – holds a pointer to cdr of list
  - Terminates in an empty-list `'()` – signals no more pairs, or end of list
- Note that lists are closed under operations of **`cons`** and **`cdr`**.

# Conventional Interfaces -- Lists

**(cons <el1> <el2>)**




**(list <el1> <el2> ... <eln>)**




**(list 1 2 3 4)** ➔ **(1 2 3 4)**

Predicate

**(null? <z>)**

**==> #t** if **<z>** evaluates to empty list

# Types – compound data

- Pair<A,B>
  - A compound data structure formed by a cons pair, in which the car element is of type A, and the second of type B: e.g. (cons 1 2) has type Pair<number, number>
- List<A>=Pair<A, List<A> or ‘()>
  - A compound data structure that is recursively defined as a pair, whose car element is of type A, and whose second element is either a list of type A or the empty list.
    - E.g. (list 1 2 3) has type List<number>; while (list 1 “string” 3) has type List<number | string>

# Examples

```
25                                   ; Number
3.45                                 ; Number
“this is a string”                   ; String
(> a b)                              ; Boolean
(cons 1 3)                           ; Pair<Number, Number>
(list 1 2 3)                         ; List<Number>
(cons “foo” (cons “bar” ‘())) ; List<String>
```

# … to be really careful

- For today we are going to create different constructors and selectors for a list, to distinguish from pairs …
  - **`(define car car)`**
  - **`(define cdr cdr)`**
  - **`(define cons cons)`**
- These abstractions inherit closure from the underlying abstractions

# Common patterns of data manipulation

- Have seen common patterns of procedures
- When applied to data structures, often see common patterns of procedures as well
  - Procedure pattern reflects recursive nature of data structure
  - Both procedure and data structure rely on
    - Closure of data structure
    - Induction to ensure correct kind of result returned

# cons'ing up a list

- Motivation:

```
(define 1thru4 (lambda() (list 1 2 3 4)))

(define (2thru7) (list 2 3 4 5 6 7))

...
```

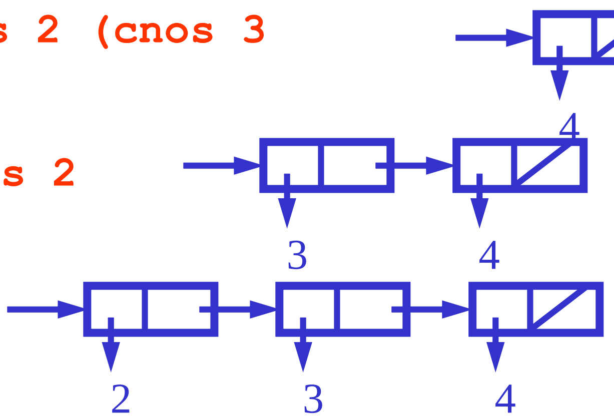
# cons'ing up a list

```
(define (enumerate-interval from to)
  (if (> from to)
      `()
      (cons from (enumerate-interval (+ 1 from) to))))
```

```
(e-i 2 4)
(if (> 2 4) `() (cons 2 (e-i (+ 1 2) 4)))
(if #f `() (cons 2 (e-i 3 4)))
(cons 2 (e-i 3 4))
(cons 2 (cons 3 (e-i 4 4)))
(cons 2 (cons 3 (cons 4 (e-i 5 4))))
(cons 2 (cons 3 (cons 4 `())))
```

(cons 2 (cnos 3 ))

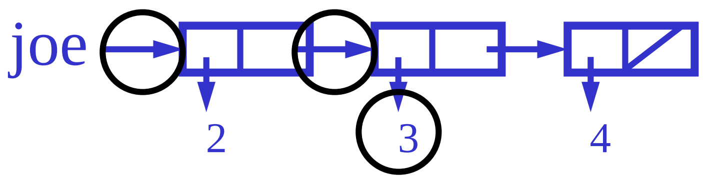
(cons 2 )




==> (2 3 4)

# cdr'ing down a list

```
(define (list-ref lst n)
  (if (= n 0)
      (car lst)
      (list-ref (cdr lst)
                (- n 1))))
```




```
(list-ref joe 1)
```

**Note how induction ensures that code is correct – relies on closure property of data structure**

```
(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

# Cdr’ing and Cons’ing Examples

```
(define (copy lst)
  (if (null? lst)                ; test
      `()                        ; base case
      (cons (car lst)            ; recursion
            (copy (cdr lst)))))
```

```
(append (list 1 2) (list 3 4))
==> (1 2 3 4)
```

Strategy: “copy” list1 onto front of list2.

```
(define (append list1 list2)
  (cond ((null? list1) list2)   ; base
        (else
          (cons (car list1)      ; recursion
                (append (cdr list1) list2)))))
```

# Mapping over Lists

```
(define group (list p1 p2 … p9))

(define stretch-group
  (lambda (gp sc)
    (if (null? gp)
        `()
        (cons (stretch-point (car gp) sc)
                  (stretch-group (cdr gp) sc)))))
```

**`stretch-group`** separates operations on points from operations on the group

Walks (cdr's) down the list, creates a new point, cons'es up a new list of points.

# Mapping over Lists

```
(define (square-list lst)
  (if (null? lst)
      `()
      (cons (square (car lst))
            (square-list (cdr lst)))))

(define (double-list lst)
  (if (null? lst)
      `()
      (cons (* 2 (car lst))
            (double-list (cdr lst)))))
```

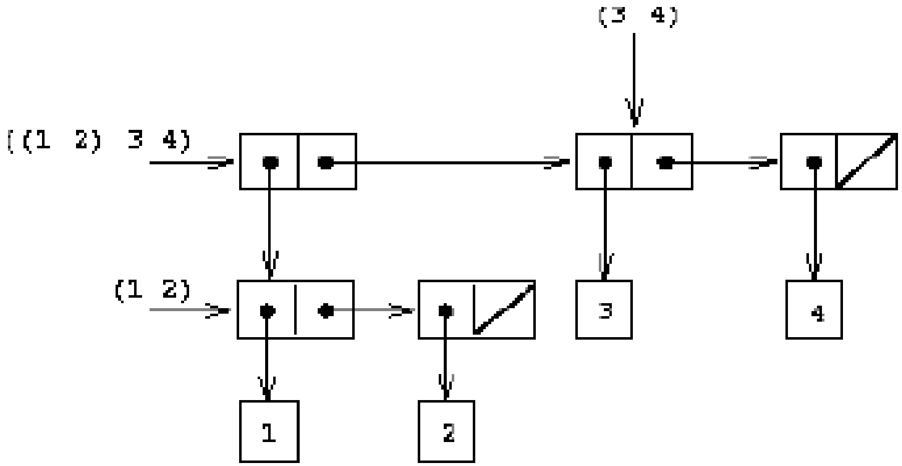
```
(define (MAP proc lst)
  (if (null? lst)
      `()
      (cons (proc (car lst))
            (map proc (cdr lst)))))
```

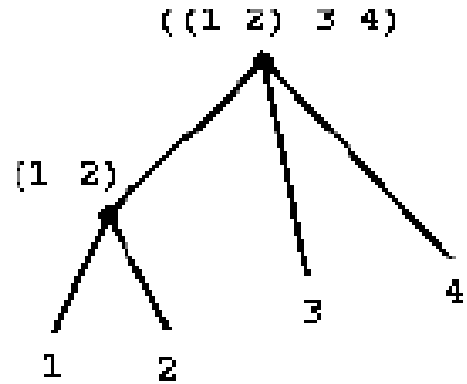
Transforms a list to a list, replacing each value by the procedure applied to that value

```
(define (square-list lst)
  (map square lst))
(square-list (list 1 2 3 4)) → ?

(define (double-list lst)
  (map (lambda (x) (* 2 x)) lst))
```

# Hierarchical Structures

(define x (cons (list 1 2)  (list 3 4)))




(length x) → 3
(count-leaves x) → 4
(list x x) → (((1 2) 3 4) ((1 2) 3 4)))
(length x) → 2
(count-leaves (list x x)) → 8

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

# Mapping over Trees

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else (cons (scale-tree (car tree) factor)
                    (scale-tree (cdr tree) factor)))))

(define (scale-tree tree factor)
  (map (lambda (sub-tree)
         (if (pair? sub-tree)
             (scale-tree sub-tree factor)
             (* sub-tree factor)))
  tree))
```

# Sequences as a Conventional Interfaces

Consider the following two different procedures

```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
             (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))

(define (even-fibs n)
  (define (next k)
      (if (> k n)
          nil
          (let ((f (fib k)))
              (if (even? f) (cons f (next (+ k 1))) (next (+ k 1))))))
  (next 0))
```

# Sequences as a Conventional Interfaces

The car program

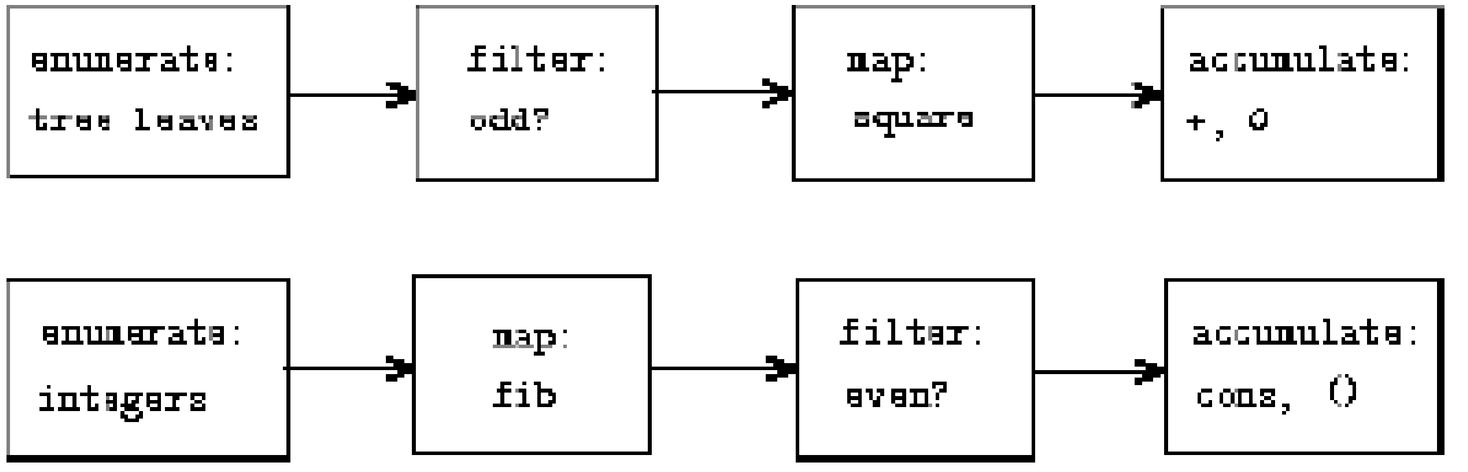
- ✓enumerates the leaves of a tree;
- ✓filters them, selecting the odd ones;
- ✓squares each of the selected ones; and
- ✓accumulates the results using +, starting with 0.

The second program

- ✓enumerates the integers from 0 to *n;*
- ✓computes the Fibonacci number for each integer;
- ✓filters them, selecting the even ones; and
- ✓accumulates the results using cons, starting with the empty list.

# Sequences as a Conventional Interfaces

- Similarity of two procedures – signal processing approach



# Filtering a List

```
(map square (list 1 2 3 4 5))
(1 4 9 16 25)

(define (filter pred lst)
  (cond ((null? lst) `())
        ((pred (car lst))
         (cons (car lst)
               (filter pred (cdr lst))))
        (else (filter pred (cdr lst)))))

(filter odd? (list 1 2 3 4 5 6))
;Value: (1 3 5)
```

# Accumulating Results

```
(define (add-up lst)
  (if (null? lst)
      0
      (+ (car lst)
         (add-up (cdr lst)))))

(define (mult-all lst)
  (if (null? lst)
      1
      (* (car lst)
         (mult-all (cdr lst)))))

(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (accumulate op init (cdr lst)))))

(define (add-up lst)
  (accumulate + 0 lst))
```

# Sequence Operations

```
(define (enumerate-interval low high)
  (if (> low high)
          nil
          (cons low (enumerate-interval (+ low 1) high))))
```

(enumerate-interval 2 7) → *(2 3 4 5 6 7)*

```
(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (list tree))
        (else (append (enumerate-tree (car tree))
                      (enumerate-tree (cdr tree))))))
```

(enumerate-tree (list 1 (list 2 (list 3 4)) 5)) →*(1 2 3 4 5)*

# Sequence Operations

```
(define (even-fibs n)
   (accumulate  cons
                        nil
                        (filter even?
                                (map fib
                                        (enumerate-interval 0 n)))))

(define (list-fib-squares n)
   (accumulate   cons
                        nil
                        (map square
                                    (map fib
                                            (enumerate-interval 0 n)))))
(list-fib-squares 10)
```

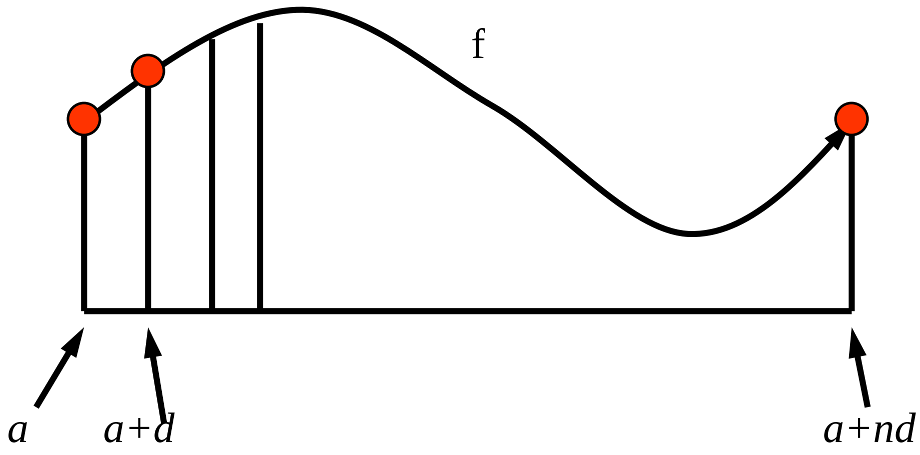
*(0 1 1 4 9 25 64 169 441 1156 3025)*

# Sequence Operations

```
(define (sum-odd-squares tree)
  (accumulate  +
               0
               (map square
                    (filter odd?
                            (enumerate-tree tree)))))


(define (salary-of-highest-paid-programmer records)
  (accumulate  max
               0
               (map salary
                  (filter programmer? records))))
```

# Using common patterns over data structures

- We can more compactly capture our earlier ideas about common patterns using these general procedures.
- Suppose we want to compute a particular kind of summation:




$$\sum_{i=0}^{n} f(a+i\delta) = f(a) + f(a+\delta) + f(a+2\delta) + \ldots + f(a+n\delta)$$

# Using common patterns over data structures

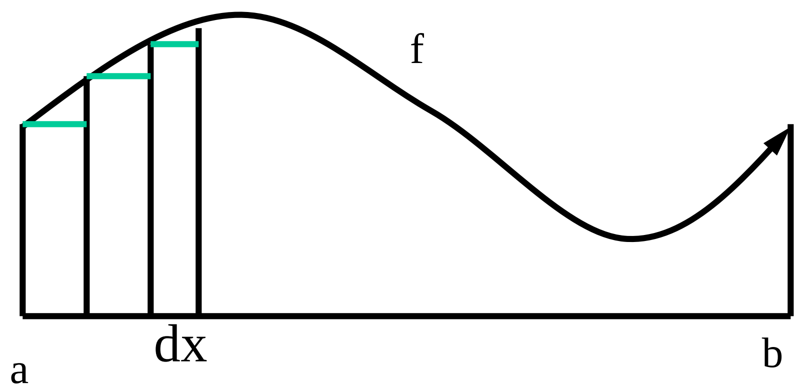
```
(define (generate-interval a b)
  (if (> a b)
      '()
      (cons a (generate-interval (+ 1 a) b))))
(generate-interval 0 6) → ?

(define (sum f start inc terms)
  (accumulate +
              0
              (map(lambda(delta)(f(+ start (* delta inc))))
                  (generate-interval 0 terms))))
```

$$\sum_{i=0}^{n} f(a + i\delta)$$

# Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))




```
(define (integral f a b n)
   (let ((delta (/ (- b a) n)))
      (* delta (sum f a delta n))))
```

# Computing Integrals

```
(define (integral f a b n)
   (let ((delta (/ (- b a) n)))
      (* (sum f a delta n) delta)))
```

$$\int_0^a \frac{1}{1+x^2}dx = ?$$

```
(define atan (lambda (a)
   (integral (lambda (x) (/ 1 (+ 1 (square x)))) 0 a)))
```

# Nested Mapping

- Given a positive integer *n, find all ordered pairs of* distinct positive integers *i and j, where 1< j< i< n, such that i + j is prime*
- *n = 6*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $i$ | 2 | 3 | 4 | 4 | 5 | 6 | 6 |
| $j$ | 1 | 2 | 1 | 3 | 2 | 1 | 5 |
| $i + j$ | 3 | 5 | 5 | 7 | 7 | 7 | 11 |

1. We map along the sequence (enumerate-interval 1 n)
2. For each *i in this sequence, we map along the sequence* (enumerate-interval 1 (- i 1)).
3. For each *j in this latter sequence, we generate the pair* (list i j)

# Nested Mapping

```
(accumulate append
            nil
            (map (lambda (i)
                   (map (lambda (j) (list i j))
                        (enumerate-interval 1 (- i 1))))
                 (enumerate-interval 1 n)))

(define (flatmap proc seq)
  (accumulate append nil (map proc seq)))
```

# Nested Mapping

```
(define (prime-sum? pair)
   (prime? (+ (car pair) (cadr pair))))

(define (make-pair-sum pair)
   (list (car pair) (cadr pair) (+ (car pair) (cadr pair))))

(define (prime-sum-pairs n)
   (map make-pair-sum
            (filter prime-sum?
                  (flatmap
                      (lambda (i)
                            (map (lambda (j) (list i j))
                                  (enumerate-interval 1 (- i 1))))
                      (enumerate-interval 1 n)))))
```

# Nested Mapping

```
(define (permutations s)
  (if (null? s) ; empty set?
      (list nil) ; sequence containing empty set
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))

(define (remove item sequence)
  (filter (lambda (x) (not (= x item)))
      sequence))
```

# Lessons learned

- There are conventional ways of grouping elements together into compound data structures.
- The procedures that manipulate these data structures tend to have a form that mimics the actual data structure.
- Compound data structures rely on an inductive format in much the same way recursive procedures do.  We can often deduce properties of compound data structures in analogy to our analysis of recursive procedures by using induction.