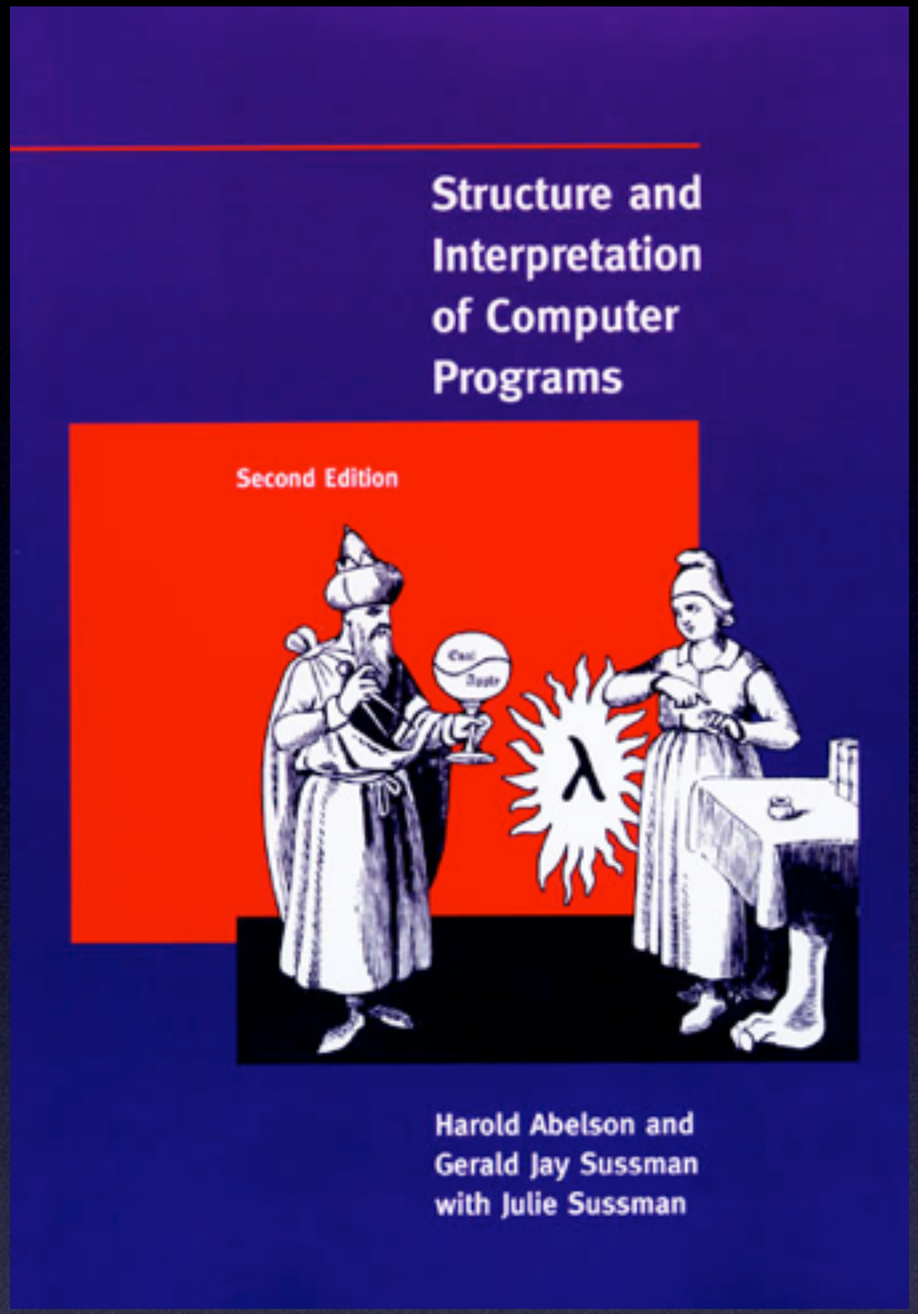


MCN 7105: Structure & Interpretation of Computer Programs

Chapter 2

Building Abstractions with Data

Status So Far

	data	procedures
primitive	X	X
combinations		X
abstraction		X

Now we treat compound data

Why Compound Data?

A rational number can be thought of as a numerator and a denominator (i.e. 2 numbers) or as “a rational number” (i.e. 1 number that happens to consist of 2 numbers)

Abstraction and Expressivity

We will introduce

- composition of data
- abstraction of data
- genericity of code
- programs as data (i.e. symbolic data)

Overal goal: structure our programs so that they can operate on abstract data

Illustration: Rational Numbers

Suppose we're writing a mathematical system.

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

Suppose we had these

Then this is possible!

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
```

Structuring Data in Scheme

```
(car (cons x y)) = x
(cdr (cons x y)) = y
```

```
> (define x (cons 1 2))
> (car x)
1
> (cdr x)
2
```

```
> (define x (cons 1 2))
> (define y (cons 3 4))
> (define z (cons x y))
> (car (car z))
1
> (car (cdr z))
3
```

Any structure can be made

Back to our Rational Numbers

4 alternative implementations

```scheme
(define make-rat cons)
(define numer car)
(define denom cdr)
```

```scheme
(define (make-rat n d)
  (cons n d))
(define (numer r)
  (car r))
(define (denom r)
  (cdr r))
```

```scheme
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
(define (numer r)
  (car r))
(define (denom r)
  (cdr r))
```

```scheme
(define (make-rat n d)
  (cons n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

Interludium: Textual Output

```
(display <expression>)
(newline)
```

```
(define (print-rat x)
  (display (numer x))
  (display "/")
  (display (denom x))
  (newline))
```

A body with multiple expressions

What is meant by Data ?

Procedural representation of cons cells

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- cons" m))))
  dispatch)

(define (car z)
  (z 0))
(define (cdr z)
  (z 1))
```

This curiosity will form the basis of object-oriented programming in Scheme (c.f. chapter 3)

Exercise

2.1 Our rational number implementation does not reduce rational numbers to lowest terms. Define a better version of make-rat that reduces rational numbers to the lowest terms.

2.2 Define a better version of make-rat that handles both positive and negative arguments. make-rat should normalize the sign so that if the rational number is positive, both the numerator and denominator are positive, and if the rational number is negative, only the numerator is negative.

Exercise

2.3 Here is an alternative procedural representation of pairs. For this representation, verify that (car (cons x y)) yields x for any objects x and y.

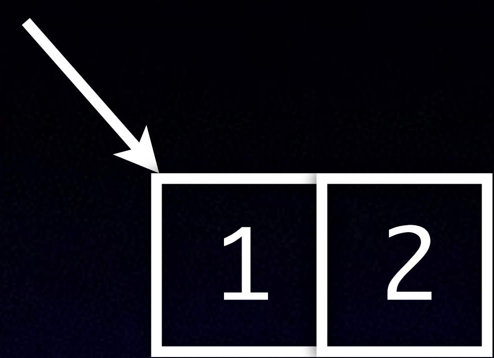
```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (p q) p)))

(define (cdr z)
  (z (lambda (p q) q)))
```

Representing Sequences

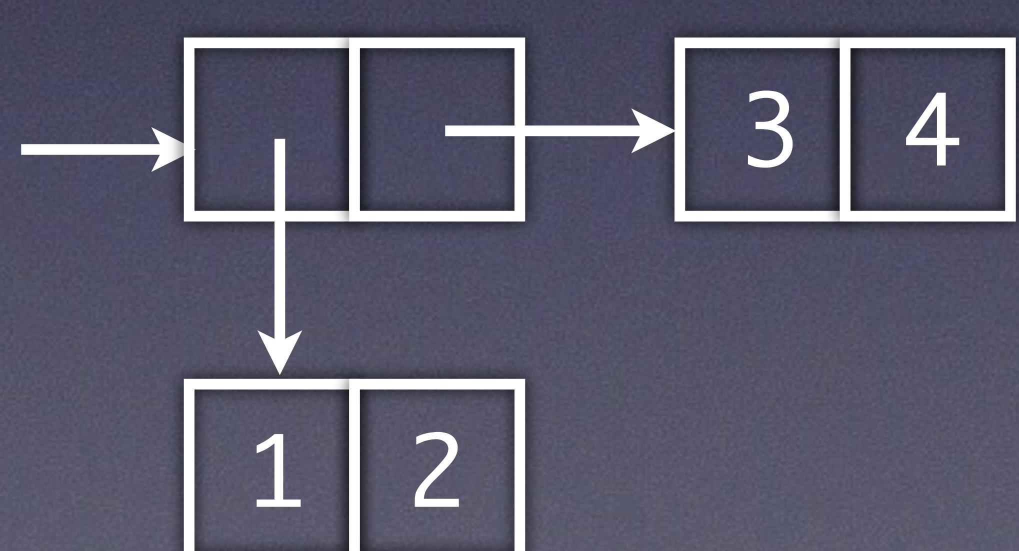
A pair's elements can be pairs again.

Closure property for pairs

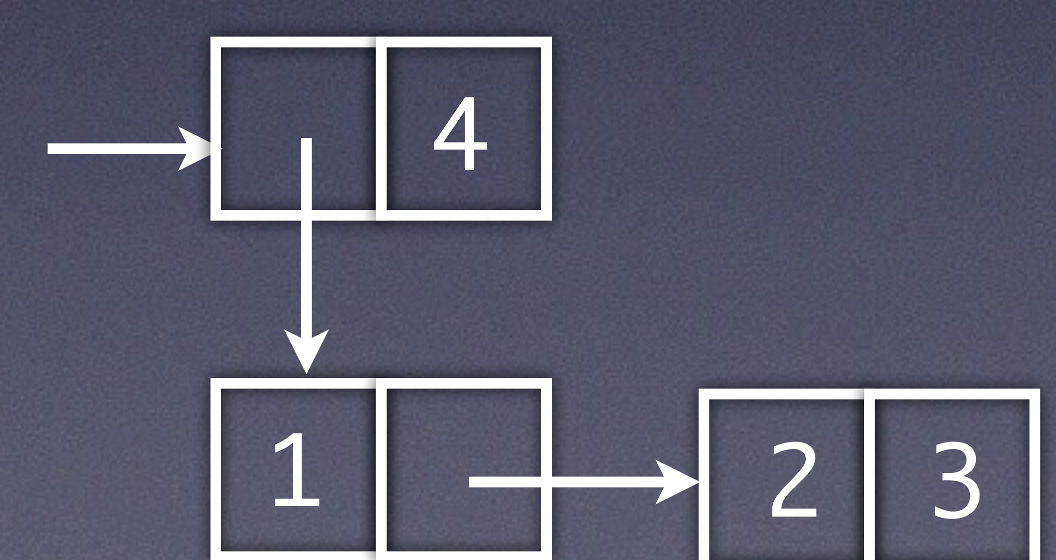
```
(cons 1 2)
```




```
(cons (cons 1 2)
      (cons 3 4))
```

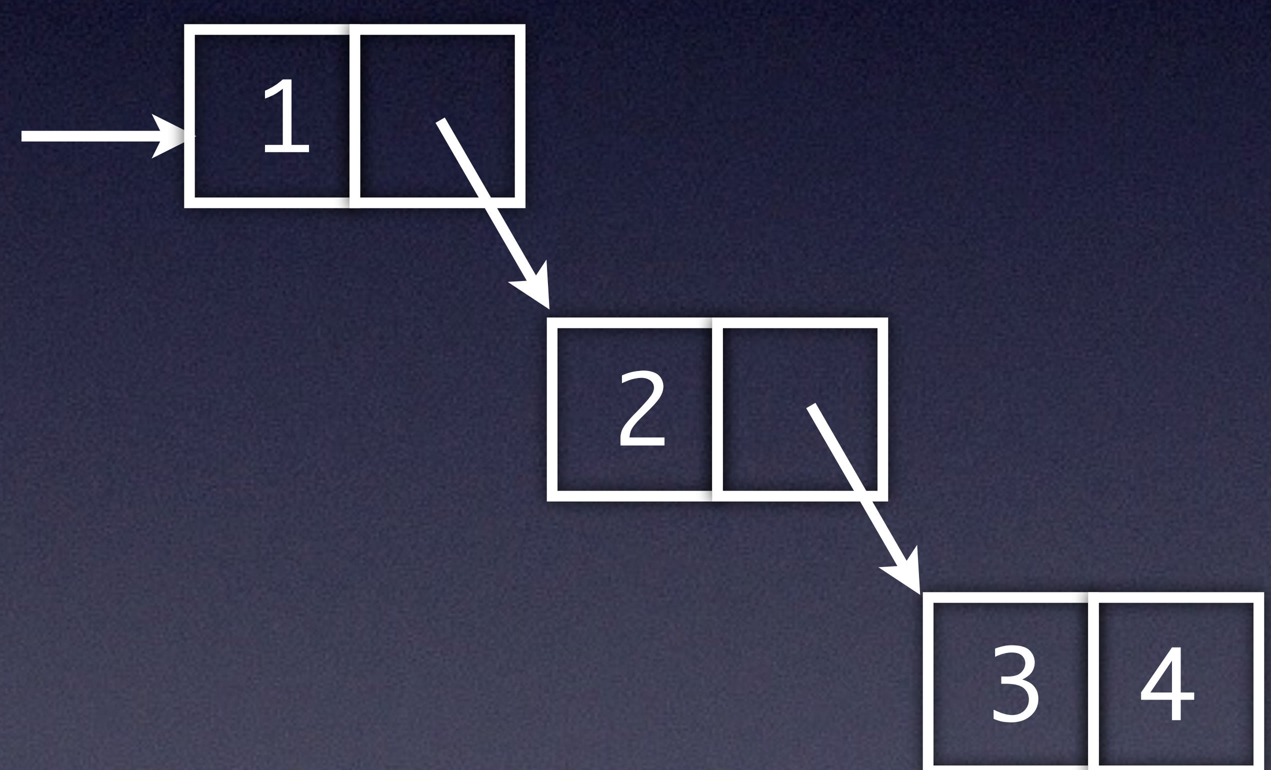



```
(cons (cons 1
            (cons 2 3))
      4)
```

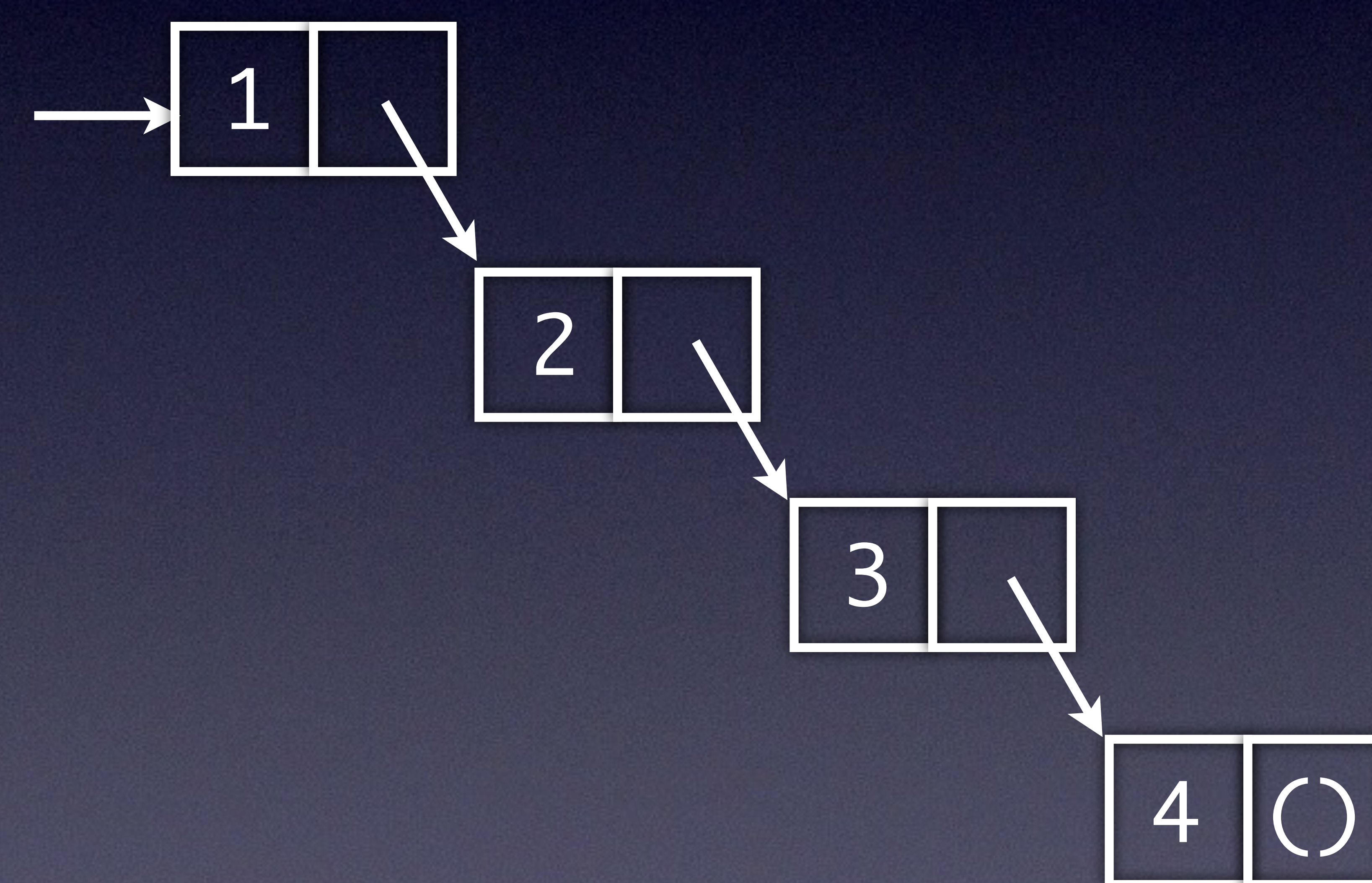


lists

A list is
- either the empty list ‘()
- any pair whose cdr is a list



not a list!



a list

For your convenience

```
> (list 1 2 3 4)
(1 2 3 4)
> (cons 1 (cons 2 (cons 3 (cons 4 '()))))
(1 2 3 4)
> (caddr (list 1 2 3 4))
3
> (caar (cons (cons 1 2) (cons 3 4)))
1
> (null? '())
#t
> (null? (list))
#t
> (null? (list 1))
#f
> (null? (cons 1 2))
#f
```

A Very Common Pitfall

```
> (1 2 3 4 5)
⊕ procedure application: expected procedure, given: 1; arguments
were: 2 3 4 5
>
```

Remember the evaluation rule for combinations

List Operations (1)

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

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

Recursive Process

```
> (define squares (list 1 4 9 16 25))
> (list-ref squares 3)
16
> (define odds (list 1 3 5 7))
> (length odds)
4
```

List Operations (2)

```
(define (length items)
  (define (length-iter a count)
    (if (null? a)
        count
        (length-iter (cdr a) (+ 1 count))))
  (length-iter items 0))

(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))
```

Iterative Process

```
> (append squares odds)
(1 4 9 16 25 1 3 5 7)
> (append odds squares)
(1 3 5 7 1 4 9 16 25)
```

Variadic Procedures

The procedures +, *, and list take arbitrary numbers of arguments. One way to define such procedures is to use define with *dotted-tail notation*.

the initial parameters (if any) will have as values the initial arguments, as usual.

```
(define (f x y . z) <body>)
```

the final parameter's value will be a list of any remaining arguments.

Mapping over lists

```
(define (map proc items)
  (if (null? items)
      '()
      (cons (proc (car items))
            (map proc (cdr items)))))
```

Abstraction!
(no cons,car,cdr)

```
> (map abs (list -10 2.5 -11.6 17))
(10 2.5 11.6 17)
> (map (lambda (x) (* x x))
       (list 1 2 3 4))
(1 4 9 16)
> (define (scale-list items factor)
    (map (lambda (x) (* x factor))
         items))
> (scale-list (list 1 2 3 4 5) 10)
(10 20 30 40 50)
```

Trees

A tree is a list whose elements are lists. The elements of the list are the branches of the tree.

Tree-recursive process

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

```
> (define x (cons (list 1 2) (list 3 4)))
> (length x)
3
> (count-leaves x)
4
> (length (list x x))
2
> (count-leaves (list x x))
8
```

Lists as Conventional Interfaces

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

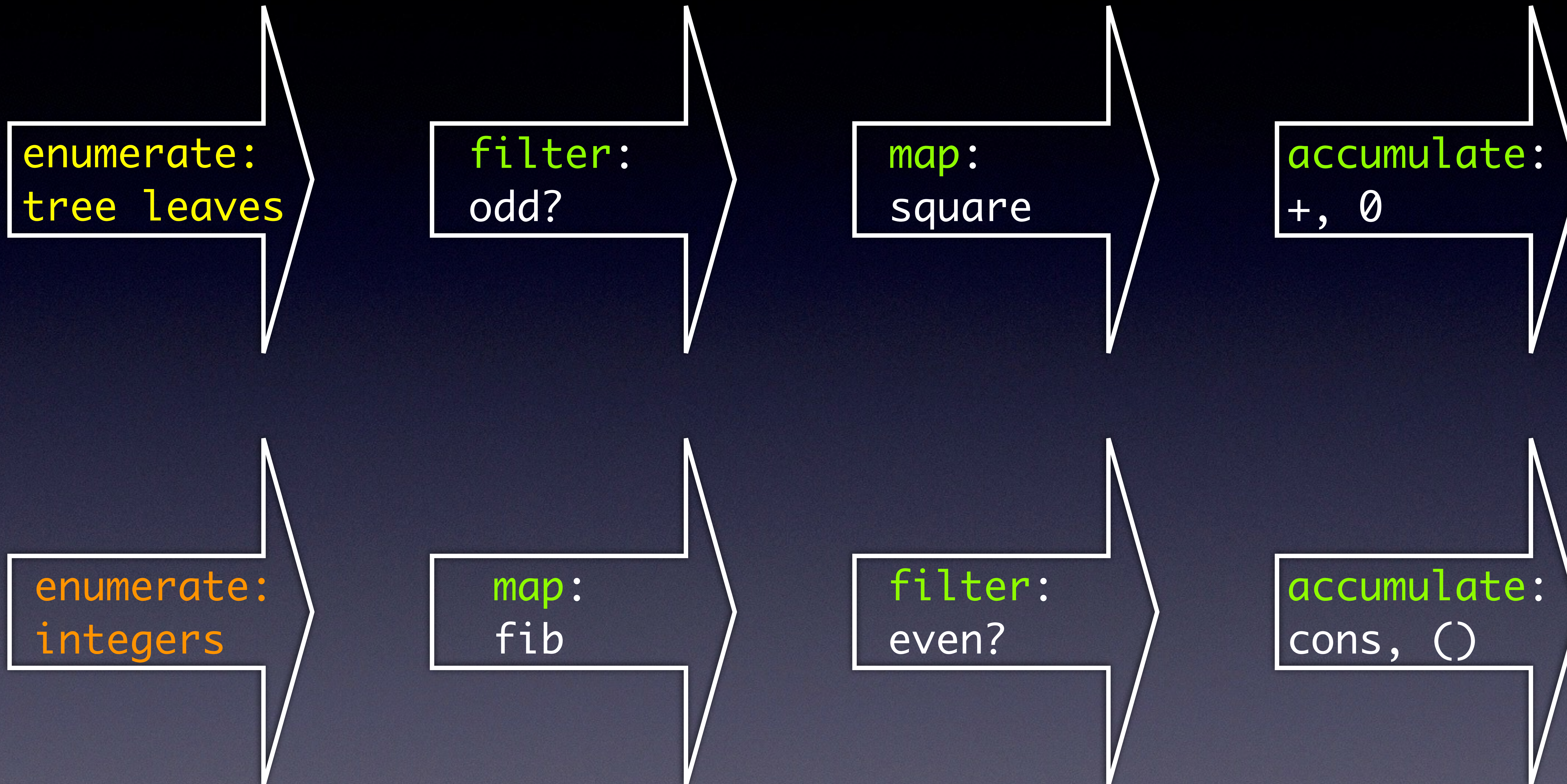
Compute the sum of the squares of the leaves that are odd

A list of all even fibonaccci numbers f_k with $k \leq n$

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

These procedures *seem* very different

A Signal-Processing Engineer's View



Let's make this structure explicit in the code...

Lists as Conventional Interfaces (I)

```
(define (filter predicate sequence)
  (cond ((null? sequence) '())
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))
```

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))
```

```
> (filter odd? (list 1 2 3 4 5))
(1 3 5)
> (accumulate + 0 (list 1 2 3 4 5))
15
> (accumulate * 1 (list 1 2 3 4 5))
120
> (accumulate cons '() (list 1 2 3 4 5))
(1 2 3 4 5)
```

Lists as Conventional Interfaces (2)

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

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

```
> (enumerate-interval 2 7)
(2 3 4 5 6 7)
> (enumerate-tree (list 1 (list 2 (list 3 4)) 5))
(1 2 3 4 5)
```

Lists as Conventional Interfaces (3)

```
(define (sum-odd-squares2 tree)
  (accumulate +
              0
              (map square
                   (filter odd?
                           (enumerate-tree tree)))))
(define (even-fibs2 n)
  (accumulate cons
              '()
              (filter even?
                      (map fib
                           (enumerate-interval 0 n)))))
```

More modular, more abstract

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

Construct a list of the squares of the first n+1 fibonacci numbers

And Reusable!

Nested Mappings

For some n, find all ordered pairs (i,j) where $1 \leq i < j \leq n$ and i+j prime.

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

Typically a problem for ‘nested loops’

Abstract it out

Nested Mappings (ctd.)

```scheme
(define (flatmap proc seq)
  (accumulate append '() (map proc seq)))

(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)))))
```

Another Example

Generate all permutations of a list L:
for each x in L: generate all permutations
of L-{x} and put x in front of each one.

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

(define (permutations s)
  (if (null? s)
      (list '())
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations (remove x s))))
               s)))
```

A Fundamental Twist in our Story

Until now: data = numbers, booleans, strings, pairs

Look at this expression:

```
(if #t
    4
    “test”)
```

It is a list of 4 elements (i.e. 4 pairs).

We know #t, we know 4 and we know “test”.

But what is if?

Answer: a symbol

A New Special Form: quote

Symbols are unevaluated identifiers

```
(quote number) = number
(quote string) = string
(quote boolean) = boolean
(quote (cons a b)) = (cons (quote a) (quote b))
(quote identifier) = identifier
```

Symbol

A common misconception: symbols are strings

Examples

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
> (list (quote a) (quote b))
(a b)
> (list (quote a) b)
(a 2)
> (quote 1)
1
> (quote (define a 1))
(define a 1)
> (car (quote (a b c)))
a
```

```
> (cdr (quote (a b c)))
(b c)
> (quote (car (quote (a b c))))
(car '(a b c))
> 'a
a
> '(a b c)
(a b c)
> '(1 2 3)
(1 2 3)
> (1 2 3)
⊕ procedure application: expected
procedure, given: 1; arguments were: 2 3
```

Shorthan

Remember the evaluation rule for combinations

eq?

```
> (eq? 1 2)
#f
> (eq? 'apple 'apple)
#t
> (eq? "apple" 'apple)
#f
> (eq? (cons 'apple 'pear) (cons 'apple 'pear))
#f
> (define c (cons 'apple 'pear))
> (eq? c c)
#t
```

Membership test procedure

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

```
> (memq 'apple '(pear banana prune))
#f
> (memq 'apple '(x (apple sauce) y apple pear))
(apple pear)
```

Symbol Manipulation: Example

We were taught one rule per expression type

$$\frac{\partial y}{\partial x} = 0 \text{ if } y \neq x$$

$$\frac{\partial y}{\partial x} = 1 \text{ if } y = x$$

$$\frac{\partial (f+g)}{\partial x} = \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x}$$

$$\frac{\partial (f.g)}{\partial x} = \frac{\partial f}{\partial x}.g + f.\frac{\partial g}{\partial x}$$

```
> (deriv '(+ x 3) 'x)
(+ 1 0)
> (deriv '(* x y) 'x)
(+ (* x 0) (* 1 y))
> (deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))
```

Representing Expressions

```scheme
(define (variable? e)
  (symbol? e))
(define (same-variable? v1 v2)
  (and (variable? v1) (variable? v2) (eq? v1 v2)))
```

```scheme
(define (make-sum a1 a2)
  (list '+ a1 a2))
(define (sum? x)
  (and (pair? x) (eq? (car x) '+)))
(define (addend s)
  (cadr s))
(define (augend s)
  (caddr s))
```

```scheme
(define (make-product m1 m2)
  (list '* m1 m2))
(define (product? x)
  (and (pair? x) (eq? (car x) '*)))
(define (multiplier p)
  (cadr p))
(define (multiplicand p)
  (caddr p))
```

Symbolic Derivation Rules

The rules in Scheme

```
(define (deriv exp var)
  (cond
    ((number? exp) 0)
    ((variable? exp)
     (if (same-variable? exp var) 1 0))
    ((sum? exp)
     (make-sum (deriv (addend exp) var)
               (deriv (augend exp) var)))
    ((product? exp)
     (make-sum
      (make-product (multiplier exp)
                    (deriv (multiplicand exp) var))
      (make-product (deriv (multiplier exp) var)
                    (multiplicand exp))))
    (else
     (error "unknown expression type -- deriv" exp))))
```

Reducing the Answers

```
> (deriv '(* (* x y) (+ x 3)) 'x)
(+ (* (* x y) (+ 1 0)) (* (+ (* x 0) (* 1 y)) (+ x 3)))
```

Can be simplified

```
(define (=number? exp num)
  (and (number? exp) (= exp num)))
(define (make-sum a1 a2)
  (cond ((=number? a1 0) a2)
        ((=number? a2 0) a1)
        ((and (number? a1) (number? a2) (+ a1 a2)))
        (else (list '+ a1 a2))))
(define (make-product m1 m2)
  (cond ((or (=number? m1 0) (=number? m2 0)) 0)
        ((=number? m1 1) m2)
        ((=number? m2 1) m1)
        ((and (number? m1) (number? m2)) (* m1 m2))
        (else (list '* m1 m2))))
```

c.f. rationals

```
> (deriv '(* (* x y) (+ x 3)) 'x)
(+ (* x y) (* y (+ x 3)))
```

much better

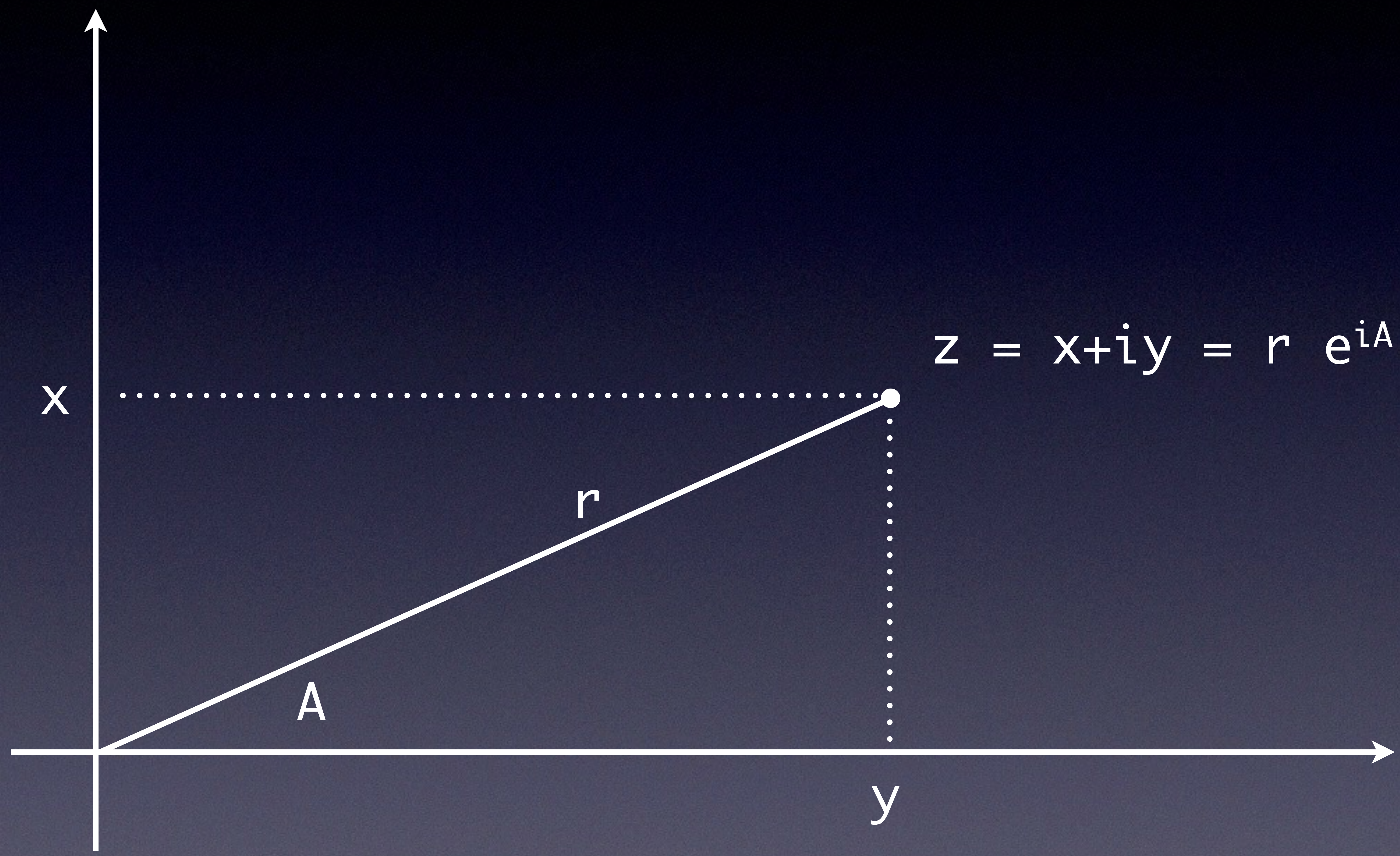
Status So Far

	data	procedures
primitive	X	X
combinations	X	X
abstraction		X

Now we treat data abstraction

let's build a math-package

We will need complex numbers



$z = x+iy = r\ e^{iA}$

There are 2 possible representations

Architecture of the System

Programs that use complex numbers

`add-complex, sub-complex, mul-complex, div-complex`

Complex-arithmetic package

Rectangular representation	Polar representation

List structure and primitive machine arithmetic

Make as much code as possible independent of the representation

Laws satisfied by a representation

```
(make-from-real-imag (real-part z) (imag-part z)) ⇔ z

(make-from-mag-ang (magnitude z) (angle z)) ⇔ z
```

Code Independent of Representation

Some code makes more sense in polar notation; other in rectangular

```
(define (add-complex z1 z2)
  (make-from-real-imag (+ (real-part z1) (real-part z2))
                       (+ (imag-part z1) (imag-part z2))))
(define (sub-complex z1 z2)
  (make-from-real-imag (- (real-part z1) (real-part z2))
                       (- (imag-part z1) (imag-part z2))))
(define (mul-complex z1 z2)
  (make-from-mag-ang (* (magnitude z1) (magnitude z2))
                     (+ (angle z1) (angle z2))))
(define (div-complex z1 z2)
  (make-from-mag-ang (/ (magnitude z1) (magnitude z2))
                     (- (angle z1) (angle z2))))
```

Two Possible Representations

Rectangular

```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z)) (square (imag-part z)))))
(define (angle z)
  (atan (imag-part z) (real-part z)))
(define (make-from-real-imag x y) (cons x y))
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))))
```

Polar

```
(define (real-part z)
  (* (magnitude z) (cos (angle z))))
(define (imag-part z)
  (* (magnitude z) (sin (angle z))))
(define (magnitude z) (car z))
(define (angle z) (cdr z))
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)))
(define (make-from-mag-ang r a) (cons r a))
```

Goal: Coexistence of Representations

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum -- TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum -- CONTENTS" datum)))
```

Tagging of Data

In other words: make code as generic as possible.

Applied to Complex numbers

```
(define (rectangular? z)
  (eq? (type-tag z) 'rectangular))
(define (polar? z)
  (eq? (type-tag z) 'polar))
```

Rectangular Representation (2nd)

```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z))
           (square (imag-part-rectangular z)))))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z)))
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)))
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
              (cons (* r (cos a)) (* r (sin a)))))
```

Tagging of Data

Polar Representation (2nd)

```
(define (real-part-polar z)
  (* (magnitude-polar z) (cos (angle-polar z))))
(define (imag-part-polar z)
  (* (magnitude-polar z) (sin (angle-polar z))))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
              (cons (sqrt (+ (square x) (square y)))
                    (atan y x))))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)))
```

Tagging of Data

Dispatching Towards Representations

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type -- REAL-PART" z))))
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Unknown type -- IMAG-PART" z))))
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Unknown type -- MAGNITUDE" z))))
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
```

Type-based Dispatching

etc...

Constructors Choose a Representation

“Reasonable” Choice

```
(define (make-from-real-imag x y)
  (make-from-real-imag-rectangular x y))
(define (make-from-mag-ang r a)
  (make-from-mag-ang-polar r a))
```

Summary

Programs that use complex numbers

`add-complex, sub-complex, mul-complex, div-complex`

Complex-arithmetic package

`real-part, imag-part`
`magnitude, angle`

Rectangular representation | Polar representation

List structure and primitive machine arithmetic

Adding a representation requires us to hand-code all these again!

Let's fix this!

What we want

Input Types

Operations

	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

Implementation

Let's make this table explicit. Adding a representation then merely requires us to extend the table.

Let Us Assume Table Operations

Notice the "state change"!

```
(put <op> <type> <item>)

(get <op> <type>)
```

Implementation:
Chapter 3

Rectangular Code

```scheme
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
             (square (imag-part z)))))
  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Packages can have the same name for local procedures!

Dispatch types are lists of arguments

Constructor-dispatch done on return type

Polar Code

```
(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z)
    (* (magnitude z) (cos (angle z))))
  (define (imag-part z)
    (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

Packages can have the same name for local procedures!

Dispatch types are lists of arguments

Constructor-dispatch done on return type

Applying An Operation

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -- APPLY-GENERIC"
            (list op type-tags))))))
```

Determine type of all arguments

If a corresponding operation exists, then apply it

```
(define (real-part z) (apply-generic 'real-part z))
(define (imag-part z) (apply-generic 'imag-part z))
(define (magnitude z) (apply-generic 'magnitude z))
(define (angle z) (apply-generic 'angle z))
```

Implementation of the 4 accessors

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular) x y))
(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

Implementation of constructors

Data-driven Programming

Read Extension in the Book

Layered application of the technique

Programs that use numbers

`add sub mul div`

Generic Arithmetic package

`add-rat sub-rat mul-rat div-rat`	`add-complex sub-complex mul-complex div-complex`		`+ - * /`
Rational Arithmetic	Complex Arithmetic		Ordinary Arithmetic
	Rectangular representation	Polar representation	

List structure and primitive machine arithmetic

What is the point of all this?

A generic function is a "set" of methods

Data-driven programming is the basis of the Common Lisp Object System (CLOS) and of the Clojure programming language.

```
(defgeneric add (a b))

(defmethod add ((a number) (b number))
   (+ a b))
(defmethod add ((a vector) (b number))
   (map 'vector (lambda (n) (+ n b)) a))

(defmethod add ((a vector) (b vector))
   (map 'vector #'+ a b))


(add 2 3)                    ; returns 5
(add #(1 2 3 4) 7)           ; returns #(8 9 10 11)
(add #(1 2 3 4) #(4 3 2 1)) ; returns #(5 5 5 5)
```

Every method declares dynamic types for the args

The "multiple dispatch" happens at runtime

This is similar (but very ≠) to Java/C++ overloading

Summary Chapters 1 and 2

	data	procedures
primitive	X	X
combinations	X	X
abstraction	X	X

We can now write and abstract over generic procedures that operate on abstractions over multiple representations of data combinations