Data with Destiny:
Building Abstractions with Data

CS21b: Structure and Interpretation of Computer Programs
Spring 2013

*My other **car** is a **cdr**...*

Building Abstractions with Data

Look at the following:

```
(define (linear a b x y)
  (+ (* a x) (* b y)))

(define (fast-exp b n)
  (cond ((= n 0) 1)
        ((even? n)
         (fast-exp (square b) (/ n 2)))
        (else
         (* b (fast-exp b (- n 1))))))
```

```
(define (linear a b x y)
   (add (mul a x) (mul b y)))

(define (fast-exp b n)
   (cond ((= n 0) unit)
         ((even? n)
          (fast-exp (square b) (/ n 2)))
         (else
          (mul b (fast-exp b (- n 1))))))
```

Procedures on left assume arguments are integers, rationals, reals via use of `+, *`.

Those on right don't: given properly designed **`add, mul, square, unit,`** we could be working with complex numbers, matrices, polynomials...

Idea: "mix and match" lets us use the same "high level" procedures with an assortment of "low level" packages...

```
linear   etc.
=========================  abstraction barrier
reals     complex      polynomials
```

Issues:

* Separating use from implementation
* Mixing multiple implementations
* ***Packaging complex data***

Suppose God (or some computational deity) gave us, via [of course!] *(let ...)*

```
(make-complex r im)    (real c)
                       (imag c)
```

Now we can define

```
(define (add c1 c2)
  (make-complex (+ (real c1) (real c2))
                (+ (imag c1) (imag c2))))

(define (mul c1 c2)
  (make-complex (- (* (real c1) (real c2))
                   (* (imag c1) (imag c2)))
                (+ (* (real c1) (imag c2))
                   (* (imag c1) (real c2)))))
```

Can we do the same trick with rationals?

```
(make-rat n d)    (numerator r)
                  (denominator r)
```

Lunar lander:

```
(make-ship-state h v f)   (height ss)
                          (velocity ss)
                          (fuel ss)
```

But where to these *construction* and *selection* operators come from?

Scheme's mechanism for "making pairs"

```
(define x (cons 10 20))
;Value: x

x
;Value: (10 . 20)

(car x)
;Value: 10

(cdr x)
;Value: 20
```

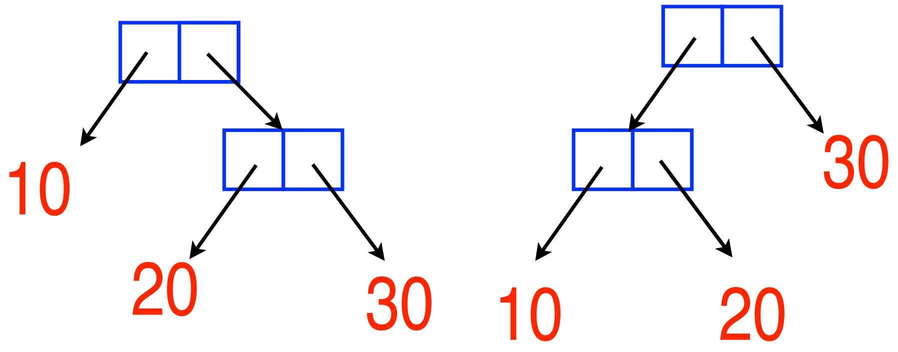
The arguments of cons can be "anything" (i.e., any first-class citizen, say **`(cons + *)`** for instance), including the result of another cons.

Box and pointer diagrams

`(define x (cons 10 (cons 20 30)))`

`(define y (cons (cons 10 20) 30))`

[Not the same!]



Atheism: making your own cons without a computational deity...

```
(define (cons x y) (lambda (m) (if m x y)))
;Value: cons

(define (car z) (z #t))
;Value: car

(define (cdr z) (z #f))
;Value: cdr

(car (cons 10 20))
;Value: 10

(cdr (cons 10 20))
;Value: 20
```

```
(define (cons x y)
  (define (cell m)
    (if m x y))
  cell)
```

Check using substitution model (this is part of the fascination with **lambda** -- it can do anything!)

```
(car (cons 10 20))
((lambda (z) (z #t)) (lambda (m) (if m 10 20)))
((lambda (m) (if m 10 20)) #t)
(if #t 10 20)
10

(cdr (cons 10 20))
((lambda (z) (z #f)) (lambda (m) (if m 10 20)))
((lambda (m) (if m 10 20)) #f)
(if #f 10 20)
20
```

The Meaning of Meaning -- Computer Style

Substitution model -- what we've got so far. What do **`cons, car, cdr`** mean? Do we need to know how they work?

`Cons-car-cdr` contract: [Axiomatic semantics]

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

An implementation can “keep the contract” any way it wants...

Dual representation: Are polar coordinates easier or harder to use than rectilinear coordinates?

"No implementation without representation!"

First, rectilinear coordinates:

Constructor: **`make-complex`**
Destructors: **`real imag`**

```
(define (add c1 c2)
  (make-complex (+ (real c1) (real c2))
                (+ (imag c1) (imag c2))))

(define (mul c1 c2)
  (make-complex (- (* real c1) (real c2))
                   (* (imag c1) (imag c2)))
                (+ (* real c1) (imag c2))
                   (* (imag c1) (real c2))))
```

Now, polar coordinates:

Constructors: **`make-complex`**
Destructors: **`norm angle`**

```
(define (add c1 c2) ...?...)

(define (mul c1 c2)
   (make-complex (* (norm c1) (norm c2))
                 (+ (angle c1) (angle c2))))
```

Why not instead the best of both worlds?

Constructors: **`make-rect make-polar`**
Destructors: **`real imag norm angle`**

```
(define (add c1 c2)
   (make-rect (+ (real c1) (real c2))
              (+ (imag c1) (imag c2))))

(define (mul c1 c2)
   (make-polar (* (norm c1) (norm c2))
               (+ (angle c1) (angle c2))))
```

Implementation 1:

```
(define make-rect cons)
(define real car)
(define imag cdr)

(define (make-polar r theta)
  (make-rect (* r (cos theta))
             (* r (sin theta))))

(define (norm c)
  (sqrt (+ (square (real c))
           (square (imag c)))))

(define (angle c)
  (arctan (/ (imag c) (real c))))
```

Implementation 2:

```
(define make-polar cons)
(define norm car)
(define angle cdr)

(define (make-rect x y)
  (make-polar (sqrt (+ (square x)
                       (square y)))
              (arctan (/ y x))))

(define (real c)
  (* (norm c) (cos (angle c))))
(define (imag c)
  (* (norm c) (sin (angle c))))
```

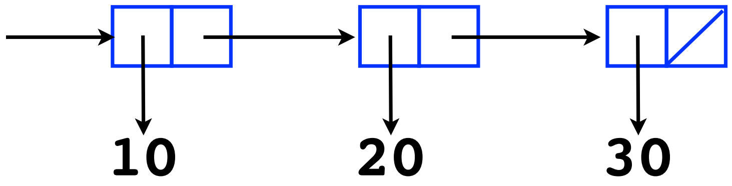
Does it make a difference which implementation you use?

Syntactic sugar: `list`

`(list x_1 x_2 ... x_n)` is syntactic sugar for

```
(cons x_1 (cons x_2 ...(cons x_n '()) ...))
```

see box and pointer diagram... `(list 10 20 30)`



`'()` [pronounced "nil"] -- the empty list, a special Scheme constant.

```
(null? '())
;Value: #t

(null? <anything else>)
;Value: #f
```

Recursion on lists:

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

(define (sum-list lst)
  (if (null? lst)
      0
      (+ (car lst) (sum-list (cdr lst)))))
```

Substitution model:

```
(length (cons 3 (cons 1 (cons 4 ‘()))))
(1+ (length (cons 1 (cons 4 ‘()))))
(1+ (1+ (length (cons 4 ‘()))))
(1+ (1+ (1+ (length ‘()))))
(1+ (1+ (1+ 0)))
...
3
```

Appending two lists:

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

Substitution model:

```
(append (list 1 2 3) (list 4 5 6 7))
(cons 1 (append (list 2 3) (list 4 5 6 7)))
(cons 1 (cons 2 (append (list 3) (list 4 5 6 7))))
(cons 1 (cons 2 (cons 3 (append ‘() (list 4 5 6 7)))))
(cons 1 (cons 2 (cons 3 (list 4 5 6 7))))
...
(list 1 2 3 4 5 6 7)
```

Reversing a list:

```
(define (reverse L)
  (if (null? L)
      ‘()
      (append (reverse (cdr L)) (list (car L)))))
```

Substitution model:

```
(reverse (list 1 2 3))
(append (reverse (list 2 3)) (list 1))
(append (append (reverse (list 3)) (list 2)) (list 1))
(append (append (append (reverse ‘())(list 3))
                                   (list 2)) (list 1))
(append (append (append ‘() (list 3)) (list 2)) (list 1))
(append (append (list 3) (list 2)) (list 1))
(append (list 3 2) (list 1))
(list 3 2 1)
```

Inductive definition of integers:

0 is an integer
if ***a*** is an integer, then `(1+ a)` is an integer
and that's all

Inductive definition of lists:

`'()` is a list
if ***a*** is anything, and ***`L`*** is a list, then **`(cons a L)`** is a list
and that's all

What are the constructors and destructors?

A menagerie of list-processing functions and higher-order procedures that implement them

```
(define (map fn L)
  (if (null? L)
      '()
      (cons (fn (car L))
            (map fn (cdr L)))))
;Value: map

(map square (list 1 2 3 4 5))
;Value: (1 4 9 16 25)

(define (filter pred lst)
  (if (null? lst)
      '()
      (if (pred (car lst))
          (cons (car lst) (filter pred (cdr lst)))
          (filter pred (cdr lst)))))
;Value: filter

(filter even? (list 0 1 2 3 4 5 6 7 8 9))
;Value: (0 2 4 6 8)

(define (append L1 L2)
  (if (null? L1)
      L2
      (cons (car L1) (append (cdr L1) L2))))
;Value: append

(append (list 0 1 2 3) (list 4 5 6 7))
;Value: (0 1 2 3 4 5 6 7)
```

Enumeration

```
(define (integers from to)
  (if (> from to)
      '()
      (cons from (integers (1+ from) to))))
;Value: integers

(integers 10 20)
;Value: (10 11 12 13 14 15 16 17 18 19 20)
```

By composing these procedures together, we begin to get a ***new idiom*** for writing programs...

```
(sum-list (filter even? (map square (integers 10 30))))
;Value: 4840

(map square (integers 10 30))
;Value: (100 121 144 169 196 225 256 289 324 361 400 441 484 529 576 625 676 729 784
841 900)

(filter even? (map square (integers 10 30)))
;Value: (100 144 196 256 324 400 484 576 676 784 900)
```

```
(define (tack x L)
  (if (null? L)
      (list x)
      (cons (car L) (tack x (cdr L)))))
;Value: tack

(tack 5 (list 10 11 12 13 14))
;Value: (10 11 12 13 14 5)

(define (reverse L)
  (if (null? L)
      '()
      (tack (car L) (reverse (cdr L)))))
;Value: reverse

(reverse (list 1 2 3 4 5))
;Value: (5 4 3 2 1)
```

Many of these procedures have a common pattern that can be abstracted...

```
(define (accumulate combiner null-value term L)
  (if (null? L)
      null-value
      (combiner (term (car L))
                (accumulate combiner null-value term (cdr L)))))
;Value: accumulate

(define (append L1 L2) (accumulate cons L2 (lambda (x) x) L1))
;Value: append

(define (map fn L) (accumulate cons '() fn L))
;Value: map

(define (reverse L) (accumulate tack '() (lambda (x) x) L))
;Value: reverse

(define (length L) (accumulate (lambda (x y) (1+ y)) 0 (lambda (x) 1) L))
;Value: length

(define (filter pred L) (accumulate ...))
;Value: filter

(define sum-list (accumulate + 0 (lambda (x) x) L))
;Value: sum-list
```

Lisping with Higher-Order Procedures

```
(define (map fn L)
  (if (null? L)
      '()
      (cons (fn (car L))
            (map fn (cdr L)))))
;Value: map

(define (integers from to)
  (if (> from to)
       '()
       (cons from (integers (1+ from) to))))
;Value: integers

(define (flatmap fn L)
  (if (null? L)
      '()
      (append (fn (car L))
              (flatmap fn (cdr L)))))
;Value: flatmap

(map (lambda (n) (integers n (+ n 5)))
     (list 10 20 30 40))
;Value: ((10 11 12 13 14 15) (20 21 22 23 24 25) (30 31 32 33 34 35) (40 41 42 43 44 45)

(flatmap (lambda (n) (integers n (+ n 5)))
        (list 10 20 30 40))
;Value: (10 11 12 13 14 15 20 21 22 23 24 25 30 31 32 33 34 35 40 41 42 43 44 45)
```

```
(define (insert x L)
  (if (null? L)
      (list (list x))
      (cons (cons x L)
            (map (lambda (M) (cons (car L) M))
                 (insert x (cdr L))))))
;Value: insert

(insert 0 (list 1 2 3 4 5))
;Value: ((0 1 2 3 4 5) (1 0 2 3 4 5) (1 2 0 3 4 5) (1 2 3 0 4 5) (1 2 3 4 0 5) (1 2
3 4 5 0))

(define (perms L)
  (if (null? L)
      (list '())
      (flatmap (lambda (M) (insert (car L) M))
               (perms (cdr L)))))
;Value: perms

(perms (list 2 3 4))
;Value:
((2 3 4) (3 2 4) (3 4 2)
 (2 4 3) (4 2 3) (4 3 2))

(perms (list 1 2 3 4))
;Value:
((1 2 3 4) (2 1 3 4) (2 3 1 4) (2 3 4 1)
 (1 3 2 4) (3 1 2 4) (3 2 1 4) (3 2 4 1)
 (1 3 4 2) (3 1 4 2) (3 4 1 2) (3 4 2 1)
 (1 2 4 3) (2 1 4 3) (2 4 1 3) (2 4 3 1)
 (1 4 2 3) (4 1 2 3) (4 2 1 3) (4 2 3 1)
 (1 4 3 2) (4 1 3 2) (4 3 1 2) (4 3 2 1))
```

Insertion Sort

```
(define (insert x L)
  (cond ((null? L) (list x))
        ((< x (car L)) (cons x L))
        (else (cons (car L) (insert x (cdr L))))))
;Value: insert

(insert 10 (list 3 6 9 12 15 18))
;Value 1: (3 6 9 10 12 15 18)

(define (isort L)
  (if (null? L)
      '()
      (insert (car L) (isort (cdr L)))))
;Value: isort

(isort (list 2 1 7 1 8 3 1 4 1 5 9 2 6))
;Value 2: (1 1 1 1 2 2 3 4 5 6 7 8 9)
```

Merge Sort

```
(define (odds L)
  (if (null? L)
      '()
      (cons (car L) (evens (cdr L)))))
;Value: odds

(define (evens L)
  (if (null? L)
      '()
      (odds (cdr L))))
;Value: evens

(odds (list 1 2 3 4 5 6 7 8 9 10))
;Value: (1 3 5 7 9)

(define (merge L1 L2)
  (cond ((null? L1) L2)
        ((null? L2) L1)
        ((< (car L1) (car L2)) (cons (car L1) (merge (cdr L1) L2)))
        (else (cons (car L2) (merge L1 (cdr L2))))))
;Value: merge

(merge (list 1 3 5 7 9) (list 2 4 6 8 10))
;Value: (1 2 3 4 5 6 7 8 9 10)

(define (msort L)
  (if (< (length L) 2)
      L
      (merge (msort (odds L)) (msort (evens L)))))
;Value: msort

(msort (list 3 1 4 1 5 9 2 6 2 7 1 8 2 8 1 4 1 4 2))
;Value: (1 1 1 1 1 2 2 2 2 3 4 4 4 5 6 7 8 8 9)
```

Why do odds, evens get used
to construct subfiles? Even sized subfiles...

Binary trees

Constructors

```
(define (leaf x) x)
;Value: leaf

(define (tree L R) (list L R))
;Value: tree
```

Destructors

```
(define (leafval L) L)
;Value: leafval

(define left car)
;Value: left

(define right cadr)
;Value: right
```

Discriminators

```
(define (leaf? T) (not (pair? T)))
;Value: leaf?

(define nonleaf? pair?)
;Value: nonleaf?

(leaf 3)
;Value: 3

(tree (leaf 3) (tree (leaf 10) (leaf 40)))
;Value: (3 (10 40))

(define (leaves T)
  (if (leaf? T)
      (list (leafval T))
      (append (leaves (left T)) (leaves (right T)))))
;Value: leaves

(leaves (tree (tree (leaf 21) (leaf 30)) (tree (leaf 14) (leaf 22))))
;Value: (21 30 14 22)
```