MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Lecture Notes – Sept. 23, 1997

COMPOUND DATA

We first define some procedures for doing arithmetic with rational numbers using an abstraction for constructing and decomposing rational numbers.

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

;;; rational multiplication
(define (*rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
```

Lisp provides *pairs* constructed with `cons` whose components can be selected with `car` and `cdr` according to the following contract:

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

Now we can implement our abstraction for rational numbers:

```
;;; constructor
(define (make-rat n d) (cons n d))          ;or (define make-rat cons) !!
;;; selectors
(define (numer x) (car x))                  ;or (define numer car)
(define (denom x) (cdr x))                  ;or (define denom cdr)
```

We can easily make a change so that our rationals always are in lowest terms. Note that by using data abstraction we don't have to touch `+rat` or `*rat`.

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/d g))))
```

Suppose we knew we were going to create an extremely large data base (such as a census data base) and only 10% of the data would ever be looked at. We could save the work of `gcd` until we accessed the data by keeping the old `make-rat` and changing `numer` and `denom`.

```
(define make-rat cons)

(define (numer x)
  (/ (car x)
     (gcd (car x) (cdr x))))

(define (denom x)
  (/ (cdr x)
     (gcd (car x) (cdr x))))
```

Now let's consider points in the plane and line segments. The length of a line segment, assuming suitable abstractions, can be computed with:

```
(define (seglength seg)
  (let ((dx (- (xcor (end-pt seg))
               (xcor (start-pt seg))))
        (dy (- (ycor (end-pt seg))
               (ycor (start-pt seg)))))
    (sqrt (+ (square dx) (square dy)))))
```

A suitable abstraction for line segments might be:

```
(define (make-set p q) (cons p q))
(define (start-pt seg) (car seg))
(define (end-pt seg) (cdr seg))
```

and an abstraction for points in rectangular coordinates:

```
(define (make-point x y) (cons x y))
(define (xcor p) (car p))
(define (ycor p) (cdr p))
```

Now suppose we wish to switch everything to operate in polar coordinates. All we have to do is change our abstraction for points:

```
(define (make-point r ang) (cons r ang))
(define (xcor p) (* (car p) (cos (cdr p))))
(define (ycor p) (* (car p) (sin (cdr p))))
```

All that has to be true of `cons`, `car`, and `cdr` is that they satisfy the given contract. In most LISPs they are primitives, but since they fall below an abstraction barrier we could implement them any way we want to. Here is one way using *message passing*:

```
(define (cons a b)              ;note syntactic sugar
  (lambda (pick)
    (cond ((= pick 1) a)
          ((= pick 2) b))))

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

No data!! Here is an even more elegant way to implement them procedurally:

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

(define car (lambda (p) (p (lambda (a b) a))))
(define cdr (lambda (p) (p (lambda (a b) b))))
```