Compounds and Varieties
**Compound data**
- To model "course", "car", "house", etc
- One built-in example: *posn*
- Declared with convenient syntax
  - You write: `(define-struct my-posn (x y z))`
  - You get: `make-my-posn`, `my-posn-x`, ...
  - These operators are called *constructs and destructors*
- Functions will often
  - Deconstruct (or *consume*) a compound value
  - Compute on "the payload"
  - Construct a (or *produce*) compound value
Today's Goal

- More on basic compound types
  - And why "not all that shines is gold".

- Varieties
  - What if we like both oranges and apples?
  - At the end of the last lab, we saw that compounds can have "categories" of data.

- Should order of evaluation matter?
  - In math, it don't
  - In many programming languages, it do
  - But it don't have to (remember that!)
More on compound data

- The posn type we say is just
  - (define-struct posn (x y))
- Then we can say
  - (make-posn 3 4)
- But we can also say
  - (define p (make-posn 'ringo 'george))
- What about (distance-from-0 p)?
"Not all that shines is gold"

- We need to be clear about what we put in compound data types
  - The design recipe needs to be modified:
    - A posn is a structure
      
      (make-posn x y)
      
      where x and y are ?

- Such restrictions are crucial for contracts to be meaningful → recipe
Varieties

- You can test the type of a value:
  - number?, boolean?, symbol?, struct?
- There is even special support for structs
  - You write: (define-struct my-posn (x y z))
  - You get: my-posn?
- Now we can talk about sets (types) that contain more than one type of thing.
We know each of these, individually
- (define-struct triangle (base height))
- (define-struct square (side))

Now we can talk about shape as either
- (triangle b h) where b and h are numbers, or
- (square s) where s is a number

Functions that use shape must follow a template
Template (for deconstruction)

; f : shape -> ...

(define (f x)
  (cond
   [(triangle? x) (...(triangle-base x) ...
                    ... (triangle-height x) ...)]
   [(square? x) (...(square-side x) ...)])
)
Instance of template

; area : shape -> number
(define (area x)
  (cond [(triangle? x)
    (* 0.5 (* (triangle-base x)
      (triangle-height x)))]
    [(square? x)
    (sqr (square-side x))]))
Evaluation order

- Take the mathematical \((1+2)/(4-1)\)
  - One way of simplifying it is:
    - \(= 3/(4-1) = 3/3 = 1\)
  - Another way is:
    - \(= (1+2)/3 = 3/3 = 1\)
- "All roads lead to Rome".
- Order don't matter. A.k.a. *confluence*. 
Evaluation order

- Consider the Scheme "functions"
  - (switch-on 'green) = true
    - on success, and draws the green bulb
  - (switch-off 'red) = true
    - on success, and erases the red bulb

- What does the following command do?
  - (and (switch-on 'red) (switch-off 'red))
Effects

- Things like
  - Input
  - Output
  - Interaction (interleaved Input/Output)
  - Exceptions/errors

- 6.2 is a bit distracting, but "OK"
- All can be done "nicely" (Moggi'91)