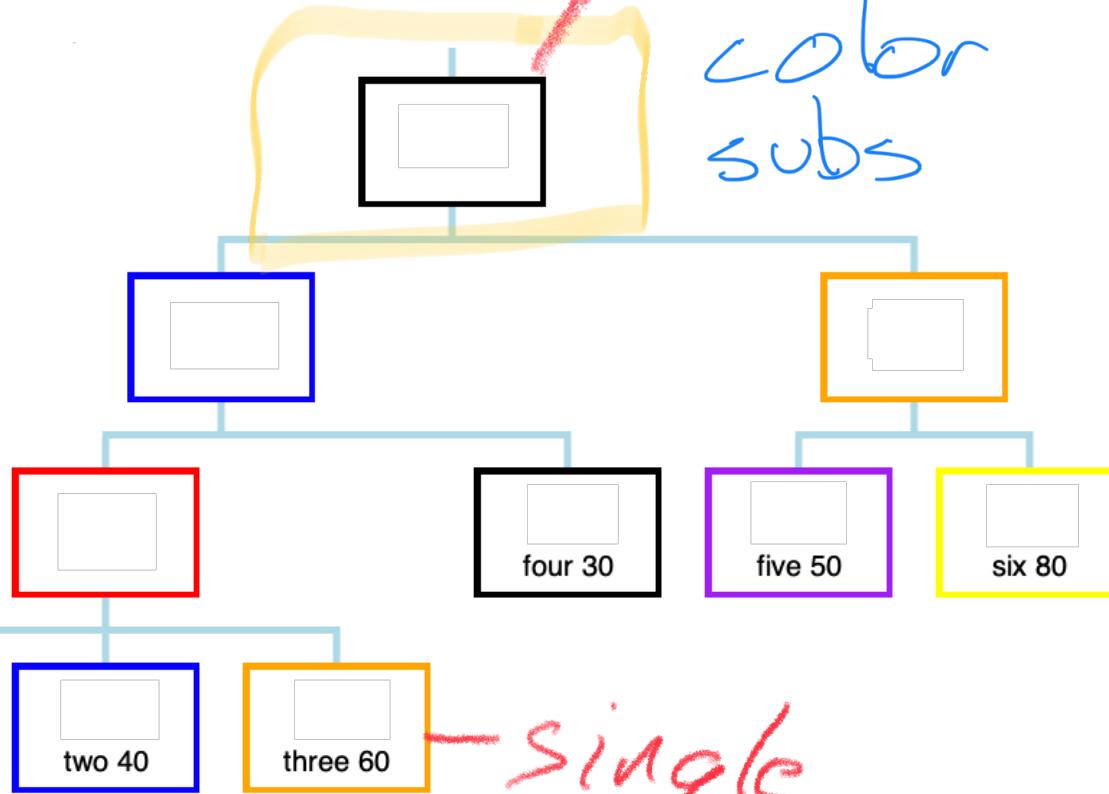


# TODAY

- Another day of reaping the rewards of the work you have done so far
- One important new idea – mutual reference / recursion
- The “hard parts” won’t actually be very hard
  - just trusting recursions as usual
- The easy parts will require learning some new details
  - minor housekeeping changes with @htdf organization

arbitrary  
arity  
tree

color  
lobe  
weight



```
;; Region is one of:  
;;   - (make-single String Natural Color)  
;;   - (make-group Color ListOfRegion)  
  
;; ListOfRegion is one of:  
;;   - empty  
;;   - (cons Region ListOfRegion)
```

```
(@HtDD Region ListOfRegion)
(define-struct single (label weight color))
(define-struct group (color subs))
;; Region is one of:
;; - (make-single String Natural Color)
;; - (make-group Color ListOfRegion)
;; interp.
;; an arbitrary-arity tree of regions
;; single regions have label, weight and color
;; groups have a color and a list of sub-regions
;;
;; weight is a unitless number indicating how much weight
;; the given single region contributes to whole tree

;; ListOfRegion is one of:
;; - empty
;; - (cons Region ListOfRegion)
;; interp. a list of regions
```

1; Question 1: [90 seconds]

1; How many arrows of any kind would you draw on the type comments?

1; A: 1    B: 2    C: 3    D: 4    E: 5

cycle

```
(@HtDD Region ListOfRegion)
(define-struct single (label weight color))
(define-struct group (color subs))
;; Region is one of:
;; - make-single String Natural Color
;; - make-group Color ListOfRegion)
;; interp.
;; an arbitrary-arity tree of regions
;; single regions have label, weight and color
;; groups have a color and a list of sub-regions
;;
;; weight is a unitless number indicating how much weight
;; the given single region contributes to whole tree
;;
;; ListOfRegion is one of:
;; - empty
;; - (cons Region ListOfRegion)
;; interp. a list of regions

;; Question 2, 3, 4: [30 each]
;;
;; Is this arrow:
;;
;; A: reference    B: self-reference    C: mutual reference
```

~~2 or more occurs flat makes a circle~~

```

(@template-origin Region)

(define (fn-for-region r)
  (cond [(single? r)
          (... (single-label r)
               (single-weight r)
               (single-color r))]
        [else
          (... (group-color r)
               (fn-for-lor (group-subs r)))]))

(@template-origin ListOfRegion)

(define (fn-for-lor lor)
  (cond [(empty? lor) (...)]
        [else
          (... (fn-for-region (first lor))
               (fn-for-lor (rest lor))))])

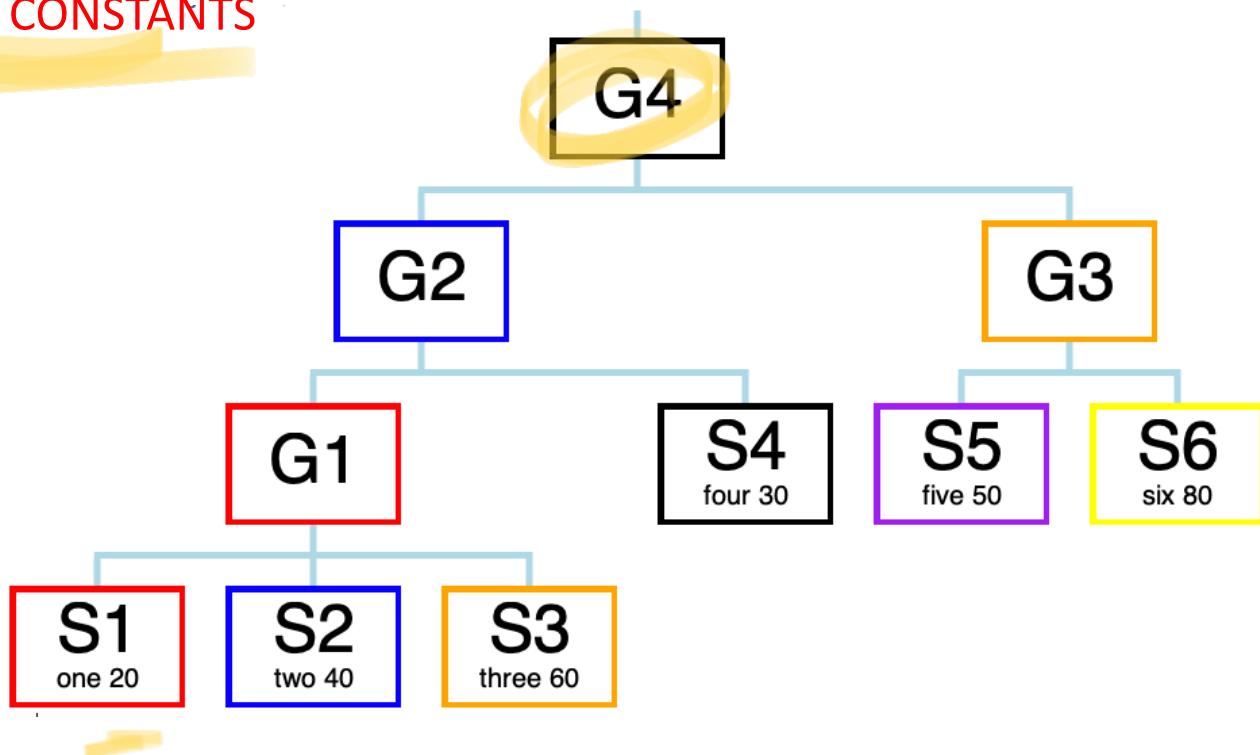
```

The diagram illustrates the concept of mutual reference between two functions. It features two code snippets: one for a single region and one for a list of regions. The first snippet, labeled 'NMR' (Non-Recursive Mutual Reference), defines a function 'fn-for-region' that either handles a single region or a group of regions. The second snippet, labeled 'NR' (Non-Recursive), defines a function 'fn-for-lor' that processes a list of regions. Red arrows point from the 'NMR' label to the 'fn-for-region' definition and from the 'NR' label to the 'fn-for-lor' definition, indicating the mutual dependency between the two functions.

mutual reference means  
these functions come in pairs  
(2 or more in general)

S1, S2, G4 etc. are the  
NAMES OF THE CONSTANTS

(find-r-r  
lFinder-r)  
"three" 64) → S3  
"apple" 64) →  
fail



```

(define (total-weight--region r)
  (cond [(single? r)
         (... (single-label r)
              (single-weight r)
              (single-color r))]
        [else
         (... (group-color r)
              (total-weight--lor (group-subs r)))]))

(define (total-weight--lor lor)
  (cond [(empty? lor) (...)]
        [else
         (+ (total-weight--region (first lor))
            (total-weight--lor (rest lor))))])

```

result will be?

total weight of subs

result will be?

tw of (first lor)

+ w (rest b1)

```
(define (all-with-color--region c r)
  (cond [(single? r)
         (... c
              (single-label r)
              (single-weight r)
              (single-color r))]
        [else
         (... c
              (group-color r)
              (all-with-color--lor c (group-subs r))))])
```

if color = list →  
→ empty?

result will be?

```
(define (all-with-color--lor c lor)
  (cond [(empty? lor) (... c)]
        [else
         (... c
              (all-with-color--region c (first lor))
              (all-with-color--lor c (rest lor))))]))
```

empty?

result will be?

result will be?





