



# Natural Numbers

---

or,

"All you ever need is compounds,  
variants, and recursion".



# Today's Lecture

---

- We've seen types like:
  - $[any] \rightarrow natural$                        $[num] \rightarrow num$
  - $[num] \rightarrow [num]$                        $[X], [X] \rightarrow [X]$
- Let's check out this guy:
  - $flatten : [[X]] \rightarrow [X]$
- And look at "lists without baggage"
  - Close look at "recursion over what?"



# Lists of different colors

---

- We've seen
  - [any] : the kitchen sink
  - [num] : sequences, n-D points, graph ...
    - distance-from-0 = (sqrt (sum (map-square l)))
- What about
  - [car], [course-info], [posn]
  - Could [[course-info]] be useful?



# Flatten

---

; flatten : [[X]] -> [X]

; Example : (flatten [ [1,2,3], [4], [5,6] ])

;           → [1,2,3,4,5,6]

(define (flatten x)

  (cond

    [(empty? x) empty]

    [else (append (first x)

              (flatten (rest x)))]))



# Make your own naturals

---

```
; A Natural is
;   - 'Zero
;   - (make-next nat) where nat is a Natural
(define-struct next (nat))
; Template:
; (define (f m)
;   (cond [(symbol? m) ...]
;         [else (... (f (next-nat m)))]))
```



# Importing them

---

```
; natIt  : number -> Natural
; purpose : convert numbers into Naturals
; example : 0 -> 'Zero, 1 -> (make-next 'zero),
;           2 -> (make-next (make-next 1)) ...
(define (natIt n)
  (cond [(= n 0) 'Zero]
        [else (make-next (natIt (- n 1)))]))
```



# Exporting them

---

```
; numIt  : Natural -> number
; purpose : convert a Natural to a number
; example : 0 <- 'Zero, 1 <- (make-next 1),
;           2 <- (make-next (make-next 1)) ...
(define (numIt n)
  (cond [(symbol? n) 0]
        [else (+ 1 (numIt (next-nat n)))]))
```



# Addition

---

```
; add : Natural, Natural -> Natural
(define (add n m)
  (cond
    [(symbol? n) m]
    [else (make-next (add (next-nat n)
                          m))]))
```

- Induction on n



# Multiplication

---

```
; mult : Natural, Natural -> Natural
(define (mult n m)
  (cond
    [(symbol? n) 'Zero]
    [else (add m (mult (next-nat n) m))]))
```

- Induction on n



# Exponentiation

---

`; exp : Natural, Natural -> Natural`

`(define (expo n m)`

`(cond`

`[(symbol? m) (make-next 'Zero)]`

`[else (mult n (expo n (next-nat m)))]))`

- Induction on m



# Greater or equal to

---

`; geq : Natural, Natural -> Natural`

`(define (geq n m) ; Induction on m`

`(cond`

`[(symbol? m) true]`

`[else (cond`

`[(symbol? n) false]`

`[else (geq (next-nat n)`

`(next-nat m))]]))`



# Subtraction

---

; minus : Natural, Natural -> Natural

(define (minus n m) ; Induction on m & n

(cond

[(symbol? m) n]

[(symbol? n) 'Zero]

[else (minus (next-nat n)

(next-nat m))]))



# Division

---

; divi : Natural, Natural -> Natural

(define (divi n m) ; Strong Induction on n

(cond

[(symbol? (next-nat m)) n]

[(not (geq n m)) 'Zero]

[else (make-next (divi (minus n m)  
m)))]))