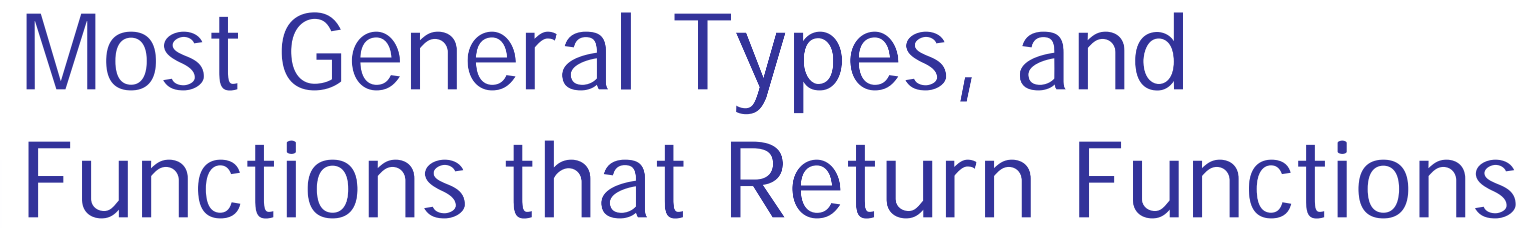

Most General Types, and Functions that Return Functions

What does most general type mean?

- Most general type makes sense when we
  - Consider introducing <u>only</u> variables
  - No "or"s, like
    - append: [X] [Y] -> [X or Y]
    - Also means no "any" ("number or symbol or …")
  - No new type constructors:
    - append: [X] [Y] -> [X followedBy Y]
- This leaves us with with [X] [X] -> [X].

Back to finding patterns

- We noticed this function:

```
;; below : lon number  ->  lon
;; to construct a list of those numbers
;; in alon that are below t
(define (below alon t)
   (cond [(empty? alon) empty]
         [else (cond [(< (first alon) t)
                      (cons (first alon) (below (rest alon) t))]
                     [else (below (rest alon) t)])]))
```

Back to finding patterns

- And we noticed this function too:

```
;; above : lon number  ->  lon
;; to construct a list of those numbers
;; in alon that are above t
(define (above alon t)
   (cond [(empty? alon) empty]
         [else (cond [(> (first alon) t)
                      (cons (first alon) (above (rest alon) t))]
                     [else (above (rest alon) t)])]))
```

Back to finding patterns

- We “abstracted” the pattern and wrote:

```
;; filter : comparison, lon number  ->  lon
;; to construct a list of those numbers n
;; in alon such that (test t n) is true
(define (filter test alon t)
   (cond [(empty? alon) empty]
            [else (cond [(test (first alon) t)
                               (cons (first alon) (filter test (rest alon) t))]
                              [else (filter test (rest alon) t)])]))
```

- But this test parameter is annoying, no?

Returning Functions

- We can can do something “easier”:

```
;; filter : comparison -> (lon number  ->  lon)
;; to construct a list of those numbers n in alon such that (test t n)
(define (filter test)
  (local ((define (new-fun alon t)
    (cond [(empty? alon) empty]
          [else (cond [(test (first alon) t)
                       (cons (first alon) (new-fun (rest alon) t))]
                      [else (new-fun (rest alon) t)])])))
      new-fun))
```

Returning Functions

- Now, instead of writing:

```
(define (above alon t) (filter > alon t))

(define (below alon t) (filter < alon t))
```

Returning Functions

- We write just:

```
(define above (filter >))

(define below (filter <))
```

- And that is how things get even cooler.

How does it really work?

```
(filter <)
= ;; by function application
  (local ((define (new-fun alon t)
   (cond [(empty? alon) empty]
         [else (cond [(< (first alon) t)
                      (cons (first alon) (new-fun (rest alon) t))]
                     [else (new-fun (rest alon) t)])]))))
      new-fun))
```

How does it really work?

```
(filter <)
= ;; by renaming of "new-fun" to "below"
  (local ((define (below alon t)
   (cond [(empty? alon) empty]
         [else (cond [((< (first alon) t)
                          (cons (first alon) (below (rest alon) t))]
                        [else (below (rest alon) t)])]))))
      below))
```

And we need one more rule

```
(define f
   (local ((define (f x) ...body... )
      f))
=
(define (f x) ...body... )
```

Postscript

- In class:
  - We illustrated this idea with (f x y) = x+y.
  - We emphasized that this is about changing the type of a function from:
    - A, B -> C
    - A -> (B -> C)
- To learn more about the relation between these too types, lookup "Currying", named after Haskell Curry.