Most General Types, and Functions that Return Functions
What does most general type mean?

- Most general type makes sense when we consider introducing only variables.
- No "or"s, like:
  - append: \([X \, [Y] \rightarrow [X \ or \ Y]\)
- Also means no "any" ("number or symbol or …")
- No new type constructors:
  - append: \([X \, [Y] \rightarrow [X \ followedBy \ Y]\)
- This leaves us with \([X \, [X] \rightarrow [X].\)
We noticed this function:

;; below : Ion number  ->  Ion
;; 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:

```scheme
;; above : Ion number -> Ion
;; to construct a list of those numbers
;; in alon that are above t
(define (above alon t)
  (cond [(empty? alon) empty]
        [(> (first alon) t)
         (cons (first alon) (above (rest alon) t))]
        [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 do something “easier”:

```scheme
;; 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:

\[
\text{(define (above alon t) (filter > alon t))}
\]

\[
\text{(define (below alon t) (filter < alon t))}
\]
Returning Functions

- We write just:

  \[
  \text{(define above (filter >))}
  \]

  \[
  \text{(define below (filter <))}
  \]

- 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

\[
\text{(define } f \\
\text{ (local ((define (f x) \ldots body\ldots ) \ \\
\text{ f)))} \\
= \\
\text{(define (f x) \ldots body\ldots )}
\]
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:
    - \(A, B \rightarrow C\)
    - \(A \rightarrow (B \rightarrow C)\)
  - To learn more about the relation between these two types, lookup "Currying", named after Haskell Curry.