abstraction

- “never write the same thing twice”
  - that’s too strong, but…
- Good programmers hate to see the same code twice
  - makes for more code
  - makes for more bugs
  - makes for more inconsistencies
- When we see a pattern

  →we want to develop an abstraction of the pattern
  - abstraction is a verb and a noun

```
(define (all-greater? lon x)
  (cond [(empty? lon) true]
        [else
         (and (> (first lon) x)
              (all-greater? (rest lon) x))]))
```

```
(define (all-positive? lon)
  (cond [(empty? lon) true]
        [else
         (and (positive? (first lon))
              (all-positive? (rest lon)))]))
```

```
(define (all-greater? lon x)
  (cond [(empty? lon) true]
        [else
         (and (> (first lon) x)                    ;try to make this one arg fn
              (all-greater? (rest lon) x))])))     ;ignore recursive call

(define (all-greater? lon x)
  (local [(define (>x? n) (> n x))]
    (cond [(empty? lon) true]
          [else
           (and (>x? (first lon))
                (all-greater? (rest lon) x))])))

(define (all-positive? lon)
  (cond [(empty? lon) true]
        [else
         (and (positive? (first lon))
              (all-positive? (rest lon)))]))
```

3 new “type expressions”

```
(listof <any-type>)          ;means ListOfAnyType
                             ;no need to do the data definition

X, Y, Z ...                  ;type parameters
                             ;one letter, starting at X
                             ;can be any type, BUT all X in a
                             ;given signature are same type

(<type> ... -> <type>)       ;a function signature
```

```
(@htdf filter2)
(@signature (X -> Boolean) (listof X) -> (listof X))
;; produce list of only those elements of lst for which p produces true
(check-expect (filter2 zero?     (list))           (list))
(check-expect (filter2 positive? (list 1 -2 3 -4)) (list 1 3))
(check-expect (filter2 negative? (list 1 -2 3 -4)) (list -2 -4))
(check-expect (filter2 empty?    (list (list 1 2) empty (list 3 4) empty))
              (list empty empty))

(@template-origin (listof X))

(define (filter2 p lox)
  (cond [(empty? lox) empty]
        [else
         (if (p (first lox))
             (cons (first lox)
                   (filter2 p (rest lox)))
             (filter2 p (rest lox)))]))
```

the built-in one is called filter

```
(@htdf all-greater-than)
(@signature Number (listof Number) -> (listof Number))
;; produce list of all elements of lon > than n
(check-expect (all-greater-than 3 empty) empty)
(check-expect (all-greater-than 3 (list 1 4 2 5)) (list 4 5))

(@template-origin use-abstract-fn)

(define (all-greater-than n lon)
  (local [(define (>n? x) (> x n))]
    (filter2 >n? lon)))
```

>n? is called a closure

it “closes over” the parameters and local defines in it’s scope

```
(define (all-greater-than n lon)
  (local [(define (>n? x) (> x n))]
    (filter2 >n? lon)))

(all-greater-than 3 (list 1 2 3 4))

(local [(define (>n? x) (> x 3))]
  (filter2 >n? (list 1 2 3 4)))

(define (>n?_0 x) (> x 3))

(filter2 >n?_0 (list 1 2 3 4))
```

function call replaces n and lon w 3 and (list 1 2 3)

local, renames and lifts
makes a unique copy of >n? that closes over n=3

```
(@htdf foldr2)
(@signature (X Y -> Y) Y (listof X) -> Y)
;; from fn b (list x0 x1...) produce (fn x0 (fn x1 ... b))
(check-expect (foldr2 + 0 (list 1 2 3)) 6)
(check-expect (foldr2 * 1 (list 2 3 4)) 24)
(check-expect (local [(define (+to-string s y)
                        (string-append (number->string s) y))]
                (foldr2 +to-string "" (list 1 37 65)))
              "13765")
(check-expect (foldr2 string-append "" (list "foo" "bar" "baz"))
              "foobarbaz")

(define (foldr2 fn b lox)
  (cond [(empty? lox) b]
        [else
         (fn (first lox)
             (foldr2 fn b (rest lox)))]))
```

the built-in one is called foldr

```
(@signature Natural (Natural -> X) -> (listof X))
;; produces (list (f 0) ... (f (- n 1)))
(define (build-list n f) ...)

(@signature (X -> boolean) (listof X) -> (listof X))
;; produce a list from all those items on lox for which p holds
(define (filter p lox) ...)

(@signature (X -> Y) (listof X) -> (listof Y))
;; produce a list by applying f to each item on lox
;; that is, (map f (list x-1 ... x-n)) = (list (f x-1) ... (f x-n))
(define (map f lox) ...)

(@signature (X -> boolean) (listof X) -> Boolean)
;; produce true if p produces true for every element of lox
(define (andmap p lox) ...)

(@signature (X -> boolean) (listof X) -> Boolean)
;; produce true if p produces true for some element of lox
(define (ormap p lox) ...)

(@signature (X Y -> Y) Y (listof X) -> Y)
;; (foldr f base (list x-1 ... x-n)) = (f x-1 ... (f x-n base))
(define (foldr f base lox) ...)

(@signature (X Y -> Y) Y (listof X) -> Y)
;; (foldl f base (list x-1 ... x-n)) = (f x-n ... (f x-1 base))
(define (foldl f base lox) ...)
```

links -> language -> end of page