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))]))


(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)))]))
```

```
(@htdf all-greater?)
(@signature ListOfNumber Number -> Boolean)
;; produce true if every number in lon is greater than x.
(check-expect (all-greater? empty 0) true)
(check-expect (all-greater? (list 2 -3 -4) -6) true)
(check-expect (all-greater? (list -2 -3 -4) -3) false)

(@template-origin ListOfNumber)

(define (all-greater? lon x)
  (local [(define (>x? n) (> n x))]
    (andmap2 >x? lon)))
```

>x? is called a closure

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

```
(define (all-greater? lon x)
  (local [(define (>x? n) (> n x))]
    (andmap2 >x? lon)))

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

(local [(define (>x? n) (> n 3))]
  (andmap2 >x? (list 1 2 3 4)))
```

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

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

(andmap2 >x?_0 (list 1 2 3 4))
```

local, renames and lifts

result is unique copy of >x? that *closes over x*=3

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 map2)
;; given fn and (list n0 n1 ...) produce (list (fn n0) (fn n1) ...)
(@signature (X -> Y) (listof X) -> (listof Y))
(check-expect (map2 sqr empty) empty)
(check-expect (map2 sqr (list 2 4)) (list 4 16))
(check-expect (map2 sqrt (list 16 9)) (list 4 3))
(check-expect (map2 abs (list 2 -3 4)) (list 2 3 4))
(check-expect (map2 string-length (list "a" "bc" "d")) (list 1 2 1))

(@template-origin (listof X))

(define (map2 fn lox)
  (cond [(empty? lox) empty]
        [else
         (cons (fn (first lox))
               (map2 fn (rest lox)))]))
```

the built-in one is called map

```
(@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 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

built-in abstract functions

links -> language -> end of page

```
(map fn (list x0 x1 ... xn))     →  (list (fn x0) (fn x1) ... (fn xn))
```

result list has same # elements as argument list

```
(foldr fn y (list x0 x1 ... xn)) →  (fn x0 (fn x1 ... (fn xn y)))
```

folds argument list down onto result

```
(filter fn (list x0 x1 ... xn))  → <lst, with only xi for which fn is true>
```

result list has same elements, possibly fewer

```
(build-list n fn)                → (list (fn 0) (fn 1) ... (fn n-1))
```

result list of n elements

```
(andmap fn (list x0 x1 ... xn))  → (and (fn x0) (fn x1) ... (fn xn))

(ormap  fn (list x0 x1 ... xn))  → (or (fn x0) (fn x1) ... (fn xn))
```

built-in abstract functions

links -> language -> end of page

```
(@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) ...)
```

```
(@problem 1)
(@htdf circles)
(@signature (listof Natural) -> (listof Image))
;; produce list of solid blue circles of given radii
(check-expect (circles (list 3))
              (list (circle 3 "solid" "blue")))
(check-expect (circles (list 1 2 10))
              (list (circle 1 "solid" "blue")
                    (circle 2 "solid" "blue")
                    (circle 10 "solid" "blue")))

(define (circles lon) empty)
```

```
(@problem 1)
(@htdf circles)
(@signature (listof Natural) -> (listof Image))
;; produce list of solid blue circles of given radii
(check-expect (circles (list 3))
              (list (circle 3 "solid" "blue")))
(check-expect (circles (list 1 2 10))
              (list (circle 1 "solid" "blue")
                    (circle 2 "solid" "blue")
                    (circle 10 "solid" "blue")))

;(define (circles lon) empty)

;; Produces list of same length as argument; Every element of result
;; list is a function of corresponding element of argument list.
;; use map
(@template-origin use-abstract-fn)
```

```
(@problem 1)
(@htdf circles)
(@signature (listof Natural) -> (listof Image))
;; produce list of solid blue circles of given radii
(check-expect (circles (list 3))
              (list (circle 3 "solid" "blue")))
(check-expect (circles (list 1 2 10))
              (list (circle 1 "solid" "blue")
                    (circle 2 "solid" "blue")
                    (circle 10 "solid" "blue")))

;(define (circles lon) empty)

;; Produces list of same length as argument; Every element of result
;; list is a function of corresponding element of argument list.
;; use map
(@template-origin use-abstract-fn)



(define (circles lon)
  (local [(@signature Number -> Image)
          ;; produce one circle of given radius
          (@template-origin Number)
          (define (one-circle r)
            (... r))]

    (map one-circle lon)))
```

```
(build-list 3 number->string)                    ; -> (list "0" "1" "2")

(build-list 3 identity)                          ; -> (list 0 1 2)

(build-list 3 add1)                              ; -> (list 1 2 3)


(local [(define (one-image n)
          (text (number->string n) 40 "black"))]
  (build-list 3 one-image))                      ; -> (list 0 1 2)

(local [(define (produce-2 ignore) 2)]
  (build-list 10 produce-2))                     ;-> (list 2 2 2 2 2 2 2 2 2 2)
```