# Symbols

A `list-of-sym` program:

```
; list-of-sym -> list-of-sym
(define (eat-apples l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define ate-rest (eat-apples (rest l)))]
      (cond
       [(symbol=? (first l) 'apple) ate-rest]
       [else (cons (first l) ate-rest)]))]))
```

# Symbols

A `list-of-sym` program:

```
; list-of-sym -> list-of-sym
(define (eat-apples l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define ate-rest (eat-apples (rest l)))]
      (cond
       [(symbol=? (first l) 'apple) ate-rest]
       [else (cons (first l) ate-rest)]))]))
```

- How about `eat-bananas`?
- How about `eat-non-apples`?

# Symbols

A `list-of-sym` program:

```
; list-of-sym -> list-of-sym
(define (eat-apples l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define ate-rest (eat-apples (rest l)))]
      (cond
       [(symbol=? (first l) 'apple) ate-rest]
       [else (cons (first l) ate-rest)]))]))
```

- How about `eat-bananas`?
- How about `eat-non-apples`?

We know where this leads...

# Filtering Symbols

```
; (sym -> bool) list-of-sym -> list-of-sym
(define (filter-syms PRED l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define r
              (filter-syms PRED (rest l)))]
      (cond
       [(PRED (first l))
        (cons (first l) r)]
       [else r]))]))
```

# Filtering Symbols

```
; (sym -> bool) list-of-sym -> list-of-sym
(define (filter-syms PRED l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define r
              (filter-syms PRED (rest l)))]
      (cond
       [(PRED (first l))
        (cons (first l) r)]
       [else r]))]))
```

This looks *really* familiar

# Last Time: Filtering Numbers

```
; (num -> bool) list-of-num -> list-of-num
(define (filter-nums PRED l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define r
              (filter-nums PRED (rest l)))]
      (cond
       [(PRED (first l))
        (cons (first l) r)]
       [else r]))]))
```

# Last Time: Filtering Numbers

```
; (num -> bool) list-of-num -> list-of-num
(define (filter-nums PRED l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define r
              (filter-nums PRED (rest l)))]
      (cond
       [(PRED (first l))
        (cons (first l) r)]
       [else r]))]))
```

How do we avoid cut and paste?

# Filtering Lists

We know this function will work for both number and symbol lists:

```
; ...
(define (filter PRED l)
  (cond
   [(empty? l) '()]
   [(cons? l)
    (local [(define r
              (filter PRED (rest l)))]
      (cond
       [(PRED (first l))
        (cons (first l) r)]
       [else r]))]))
```

But what is its signature?

# The Signature of Filter

How about this?

```
(num-OR-sym -> bool) list-of-num-OR-list-of-sym
-> list-of-num-OR-list-of-sym

 ; A num-OR-sym is either
 ;   - num
 ;   - sym

 ; A list-of-num-OR-list-of-sym is either
 ;   - list-of-num
 ;   - list-of-sym
```

# The Signature of Filter

How about this?

```
(num-OR-sym -> bool) list-of-num-OR-list-of-sym
-> list-of-num-OR-list-of-sym
```

This signature is too weak to define **eat-apples**

```
; list-of-sym -> list-of-sym
(define (eat-apples l)
  (filter not-apple? l))

; sym -> bool
(define (not-apple? s)
  (not (symbol=? s 'apple)))
```

**eat-apples** must return a **list-of-sym**, but by its signature, **filter** might return a **list-of-num**

# The Signature of Filter

How about this?

```
(num-OR-sym -> bool) list-of-num-OR-list-of-sym
-> list-of-num-OR-list-of-sym
```

This signature is too weak to define **eat-apples**

```
; list-of-sym -> list-of-sym
(define (eat-apples l)
  (filter not-apple? l))

; sym -> bool
(define (not-apple? s)
  (not (symbol=? s 'apple)))
```

**not-apple?** only works on symbols, but by its signature **filter** might give it a **num**

# The Signature of Filter

The reason **`filter`** works is that if we give it a **`list-of-sym`**, then it returns a **`list-of-sym`**

Also, if we give **`filter`** a **`list-of-sym`**, then it calls **`PRED`** with symbols only

# The Signature of Filter

The reason **`filter`** works is that if we give it a **`list-of-sym`**, then it returns a **`list-of-sym`**

Also, if we give **`filter`** a **`list-of-sym`**, then it calls **`PRED`** with symbols only

A better signature:

```
((num -> bool) list-of-num
 -> list-of-num)
OR
((sym -> bool) list-of-sym
 -> list-of-sym)
```

# The Signature of Filter

The reason **`filter`** works is that if we give it a **`list-of-sym`**, then it returns a **`list-of-sym`**

Also, if we give **`filter`** a **`list-of-sym`**, then it calls **`PRED`** with symbols only

A better signature:

```
((num -> bool) list-of-num
 -> list-of-num)
OR
((sym -> bool) list-of-sym
 -> list-of-sym)
```

But what about a list of **`image`**s, **`posn`**s, or **`snake`**s?

# The True Signature of Filter

The real signature is

```
((X -> bool) list-of-X -> list-of-X)
```

where `X` stands for any type

- The caller of `filter` gets to pick a type for `X`
- All `X`s in the signature must be replaced with the same type

# The True Signature of Filter

The real signature is

```
((X -> bool) list-of-X -> list-of-X)
```

where `X` stands for any type

- The caller of `filter` gets to pick a type for `X`
- All `X`s in the signature must be replaced with the same type

Data definitions need type variables, too:

```
; A list-of-X is either
;  - '()
;  - (cons X list-of-X)
```

# Using Filter

The `filter` function is so useful that it's built in

```
(define (eat-apples l)
  (local [(define (not-apple? s)
            (not (symbol=? s 'apple)))]
    (filter not-apple? l)))
```

# Looking for Other Built-In Functions

Recall **feed-fish**:

```
; list-of-num -> list-of-num
(define (feed-fish l)
  (cond
   [(empty? l) '()]
   [else (cons (+ 1 (first l))
               (feed-fish (rest l)))]))
```

Is there a built-in function to help?

# Looking for Other Built-In Functions

Recall **feed-fish**:

```
; list-of-num -> list-of-num
(define (feed-fish l)
  (cond
   [(empty? l) '()]
   [else (cons (+ 1 (first l))
               (feed-fish (rest l)))]))
```

Is there a built-in function to help?

Yes: **map**

# Using Map

```
(define (map CONV l)
  (cond
   [(empty? l) '()]
   [else (cons (CONV (first l))
               (map CONV (rest l)))]))

; list-of-num -> list-of-num
(define (feed-fish l)
  (local [(define (feed-one n)
            (+ n 1))]
    (map feed-one l)))

; list-of-animal -> list-of-animal
(define (feed-animals l)
  (map feed-animal l))
```

# The Signature for Map

```
(define (map CONV l)
  (cond
   [(empty? l) '()]
   [else (cons (CONV (first l))
               (map CONV (rest l)))]))
```

- The `l` argument must be a list of `X`
- The `CONV` argument must accept each `X`
- If `CONV` returns a new `X` each time, then the signature for `map` is

```
(X -> X) list-of-X -> list-of-X
```

# Posns and Distances

```
; list-of-posn -> list-of-num
(define (distances l)
  (cond
   [(empty? l) '()]
   [(cons? l) (cons (distance-to-0 (first l))
                    (distances (rest l)))]))
```

# Posns and Distances

```
; list-of-posn -> list-of-num
(define (distances l)
  (cond
   [(empty? l) '()]
   [(cons? l) (cons (distance-to-0 (first l))
                    (distances (rest l)))]))
```

The `distances` function looks just like `map`, except that `distances-to-0` is

```
posn -> num
```

not

```
posn -> posn
```

# The True Signature of Map

Despite the signature mismatch, this works:

```
(define (distances l)
  (map distance-to-0 l))
```

# The True Signature of Map

Despite the signature mismatch, this works:

```
(define (distances l)
  (map distance-to-0 l))
```

The true signature of `map` is

```
(X -> Y) list-of-X -> list-of-Y
```

The caller gets to pick both `X` and `Y` independently

# More Uses of Map

```
; list-of-posn -> list-of-posn
(define (rsvp l)
  ; replaces 4 lines:
  (map flip-posn l))

; posn -> posn
....
```

# More Uses of Map

```
; list-of-num -> list-of-num
(define (align-bricks lon)
  ; replaces 4 lines:
  (map round lon))
```

# More Uses of Map

```
; list-of-car -> list-of-car
(define (rob-train l)
  ; replaces 4 lines:
  (map rob-car l))

; car -> car
...
```

# Folding a List

How about **`sum`**?

```
list-of-num -> num
```

Doesn't return a list, so neither **`filter`** nor **`map`** help

# Folding a List

How about **sum**?

```
list-of-num -> num
```

Doesn't return a list, so neither **filter** nor **map** help

Abstracting over **sum** and **product** leads to **combine-nums**:

```
; list-of-num num (num num -> num) -> num
(define (combine-nums l base-n COMB)
  (cond
   [(empty? l) base-n]
   [(cons? l)
    (COMB
     (first l)
     (combine-nums (rest l) base-n COMB))]))
```

# The Foldr Function

```
; (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
          (foldr COMB base (rest l)))]))
```

# The Foldr Function

```
; (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
          (foldr COMB base (rest l)))]))
```

The **sum** and **product** functions become trivial:

```
(define (sum l) (foldr + 0 l))
(define (product l) (foldr * 1 l))
```

# The Foldr Function

```
; (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
          (foldr COMB base (rest l)))]))

  ; list-of-posn -> num
  (define (total-distance l)
    (local [(define (add-distance p n)
              (+ (distance-to-0 p) n))]
      (foldr add-distance 0 l)))
```

# The Foldr Function

```
; (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
          (foldr COMB base (rest l)))]))
```

In fact,

```
(define (map f l)
  (local [(define (comb i r)
            (cons (f i) r))]
    (foldr comb '() l)))
```

# The Foldr Function

```
; (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
          (foldr COMB base (rest l)))]))
```

Yes, `filter` too:

```
(define (filter f l)
  (local [(define (check i r)
            (cond
             [(f i) (cons i r)]
             [else r]))]
    (foldr check '() l)))
```

# The Source of Foldr

How can **`foldr`** be so powerful?

# The Source of Foldr

Template:

```
(define (func-for-loX l)
  (cond
   [(empty? l) ...]
   [(cons? l) ... (first l)
    ... (func-for-loX (rest l)) ...]))
```

Fold:

```
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
          (foldr COMB base (rest l)))]))
```

# Other Built-In List Functions

More specializations of **foldr**:

```
ormap : (X -> bool) list-of-X -> bool

andmap : (X -> bool) list-of-X -> bool
```

Examples:

```
; list-of-sym -> bool
(define (got-milk? l)
  (local [(define (is-milk? s)
            (symbol=? s 'milk))]
    (ormap is-milk? l)))

; list-of-grade -> bool
(define (all-passed? l)
  (andmap passing-grade? l))
```

# What about Non-Lists?

Since it's based on the template, the concept of fold is general

```
; (sym num sym Z Z -> Z) Z ftn -> Z
(define (fold-ftn COMB base ftn)
  (cond
   [(empty? ftn) base]
   [(child? ftn)
    (COMB (child-name ftn) (child-date ftn) (child-eyes ftn)
          (fold-ftn COMB BASE (child-father ftn))
          (fold-ftn COMB BASE (child-mother ftn)))]))

(define (count-persons ftn)
  (local [(define (add name date color c-f c-m)
            (+ 1 c-f c-m))]
    (fold-ftn add 0 ftn)))

(define (in-family? who ftn)
  (local [(define (here? name date color in-f? in-m?)
            (or (symbol=? name who) in-f? in-m?))]
    (fold-ftn here? #false ftn)))
```