Symbols

A list-of-sym program:

; eat-apples : list-of-sym -> list-of-sym
(define (eat-apples l)
  (cond
   [(empty? l) empty]
   [(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

; filter-syms : (sym -> bool) list-of-sym
; -> list-of-sym
(define (filter-syms PRED l)
  (cond
   [(empty? l) empty]
   [(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

; filter-nums : (num -> bool) list-of-num
; -> list-of-num
(define (filter-nums PRED l)
  (cond
   [(empty? l) empty]
   [(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:

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

But what is its contract?
The Contract of Filter

How about this?

\[(\text{num-OR-sym} \to \text{bool}) \ \text{list-of-num-OR-list-of-sym} \to \text{list-of-num-OR-list-of-sym}\]

; 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 Contract of Filter

How about this?

\[(\text{num-OR-sym} \to \text{bool}) \ \text{list-of-num-OR-list-of-sym} \to \text{list-of-num-OR-list-of-sym}\]

This contract is too weak to define \textit{eat-apples}

\[
; \text{eat-apples} : \text{list-of-sym} \to \text{list-of-sym} \\
(\text{define} \ (\text{eat-apples} \ 1) \\
(\text{filter} \ \text{not-apple?} \ 1))
\]

\[
; \text{not-apple?} : \text{sym} \to \text{bool} \\
(\text{define} \ (\text{not-apple?} \ s) \\
(\text{not} \ (\text{symbol=?} \ s \ '\text{apple})))
\]

\textit{eat-apples} must return a \text{list-of-sym}, but by its contract, \textit{filter} might return a \text{list-of-num}
The Contract of Filter

How about this?

\[
\text{(num-OR-sym -> bool) list-of-num-OR-list-of-sym}
\rightarrow \text{list-of-num-OR-list-of-sym}
\]

This contract is too weak to define \text{eat-apples}

\[
\text{; eat-apples : list-of-sym -> list-of-sym}
\]
\[
\text{(define (eat-apples l)}
\text{\hspace{1em} (filter not-apple? l))}
\]

\[
\text{; not-apple? : sym -> bool}
\]
\[
\text{(define (not-apple? s)}
\text{\hspace{1em} (not (symbol=? s 'apple)))}
\]

\text{not-apple? only works on symbols, but by its contract filter might give it a num}
The Contract of Filter

The reason \texttt{filter} works is that if we give it a \texttt{list-of-sym}, then it returns a \texttt{list-of-sym}

Also, if we give \texttt{filter} a \texttt{list-of-sym}, then it calls \texttt{PRED} with symbols only

A better contract:

\begin{verbatim}
filter :
  ((num \rightarrow\ bool) list-of-num
  \rightarrow\ list-of-num)
\end{verbatim}

OR

\begin{verbatim}
[((sym \rightarrow\ bool) list-of-sym
  \rightarrow\ list-of-sym)
\end{verbatim}

But what about a list of \texttt{images}, \texttt{posns}, or \texttt{snakes}?
The True Contract of Filter

The real contract is

\[
\text{filter : } ((X \rightarrow \text{bool}) \text{ list-of-X } \rightarrow \text{ list-of-X})
\]

where \(X\) stands for any type

- The caller of \text{filter} gets to pick a type for \(X\)
- All \(X\)s in the contract must be replaced with the same type

Data definitions need type variables, too:

\[
; \ A \text{ list-of-X is either}
; \quad \text{- empty}
; \quad \text{- (cons X list-of-X)}
\]
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 \texttt{feed-fish}:

\begin{verbatim}
; feed-fish : list-of-num -> list-of-num
(define (feed-fish l)
  (cond
    [(empty? l) empty]
    [else (cons (+ 1 (first l))
      (feed-fish (rest l)))]))
\end{verbatim}

Is there a built-in function to help?

\textbf{Yes: map}
Using Map

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

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

; feed-animals : list-of-animal -> list-of-animal
(define (feed-animals l)
  (map feed-animal l))
```
The Contract for Map

(define (map CONV l)
  (cond
   [(empty? l) empty]
   [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 contract for map is

  map : (X -> X) list-of-X -> list-of-X
Posns and Distances

; distances : list-of-posn -> list-of-num
(define (distances l)
    (cond
        [(empty? l) empty]
        [(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 Contract of Map

Despite the contract mismatch, this works:

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

The true contract of `map` is

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

The caller gets to pick both `X` and `Y` independently
More Uses of Map

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

; flip-posn : posn -> posn

....
More Uses of Map

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

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

; rob-car : car -> car
...
Folding a List

How about \texttt{sum}?

\texttt{sum : list-of-num \rightarrow num}

Doesn’t return a list, so neither \texttt{filter} nor \texttt{map} help

Abstracting over \texttt{sum} and \texttt{product} leads to \texttt{combine-nums}:

\begin{verbatim}
; combine-nums : list-of-num num
; (num num \rightarrow num) \rightarrow 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)))])
\end{verbatim}
The Foldr Function

; foldr : (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

; foldr : (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)))]))

; total-distance : 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

; foldr : (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 empty l)))
The Foldr Function

; foldr : (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 empty 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 1)) ...]])

Fold:

(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
      (foldr COMB base (rest 1)))]))
Other Built-In List Functions

More specializations of `foldr`:

\[
\text{ormap} : (X \to \text{bool}) \ \text{list-of-X} \to \text{bool}
\]
\[
\text{andmap} : (X \to \text{bool}) \ \text{list-of-X} \to \text{bool}
\]

Examples:

\[
\text{; got-milk? : list-of-sym} \to \text{bool}
\]
\[
(\text{define} \ (\text{got-milk?} \ l)
\quad (\text{local} \ [(\text{define} \ (\text{is-milk?} \ s)
\quad \quad (\text{symbol=?} \ s \ '\text{milk}))]
\quad (\text{ormap} \ \text{is-milk?} \ l)))))
\]

\[
\text{; all-passed? : list-of-grade} \to \text{bool}
\]
\[
(\text{define} \ (\text{all-passed?} \ l)
\quad (\text{andmap} \ \text{passing-grade?} \ l))
\)
What about Non-Lists?

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

```scheme
; fold-ftn : (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)))
```