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:

```scheme
(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-nnums PRED l)
  (cond
   [(empty? l) '()]  
   [(cons? l)
     (local [(define r
          (filter-nnums 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:

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

But what is its signature?
The Signature of Filter

How about this?

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

; 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 Signature of Filter

How about this?

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

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

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

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

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?

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

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

\[
; \ \text{list-of-sym} \rightarrow \text{list-of-sym} \\
(\text{define (eat-apples} \ l) \\
 \quad (\text{filter not-apple?} \ l))
\]

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

\textit{not-apple?} only works on symbols, but by its signature \textit{filter} might give it a \textit{num}
The Signature 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
The Signature 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 signature:

\[
((\texttt{num} \rightarrow \texttt{bool}) \ \texttt{list-of-num} \\
\rightarrow \ \texttt{list-of-num})
\]

OR

\[
((\texttt{sym} \rightarrow \texttt{bool}) \ \texttt{list-of-sym} \\
\rightarrow \ \texttt{list-of-sym})
\]
The Signature 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 signature:

\begin{verbatim}
  ((num -> bool) list-of-num
   -> list-of-num)

 OR

  ((sym -> bool) list-of-sym
   -> list-of-sym)
\end{verbatim}

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

The real signature is

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

where \(X\) stands for any type

- The caller of \texttt{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

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

where \(X\) stands for any type

- The caller of \textit{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:

\[
; \text{ A list-of-X is either} \\
; \quad - \; '() \\
; \quad - \; (\text{cons} \; X \; \text{list-of-X})
\]
Using Filter

The \texttt{filter} function is so useful that it's built in

\begin{verbatim}
(define (eat-apples l)
  (local [(define (not-apple? s)
         (not (symbol=? s 'apple)))]
  (filter not-apple? l)))
\end{verbatim}
Looking for Other Built-In Functions

Recall \texttt{feed-fish}:

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

Is there a built-in function to help?
Looking for Other Built-In Functions

Recall \textit{feed-fish}:

\begin{verbatim}
; list-of-num -> list-of-num
(define (feed-fish l)
  (cond
    [(empty? l) '()]
    [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) '()]  
   [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
  \[(X \rightarrow X) \ list-of-X \rightarrow list-of-X\]
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:

\[
\text{(define (distances l))}
\text{(map distance-to-0 l))}
\]

The true signature of \text{map} is

\[(X \rightarrow Y) \text{ list-of-X } \rightarrow \text{ list-of-Y}\]

The caller gets to pick both \text{X} and \text{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`?

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

Doesn’t return a list, so neither `filter` nor `map` help
Folding a List

How about \texttt{sum}?

\[
\text{list-of-num} \rightarrow \text{num}
\]

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}:

\[
; \text{list-of-num \ num} (\text{num \ num} \rightarrow \text{num}) \rightarrow \text{num}
\]

\[
(\text{define} \ (\text{combine-nums \ l \ base-n \ COMB})
\]

\[
(\text{cond}
[[(\text{empty? \ l}) \ \text{base-n}]

[[(\text{cons? \ l})

(\text{COMB}

(\text{first \ l})

(\text{combine-nums \ (rest \ l) \ base-n \ COMBuminum}))]))

\]
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 \texttt{foldr} be so powerful?
The Source of Foldr

Template:

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

Fold:

```lisp
(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 \texttt{foldr}:

\begin{align*}
\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}
\end{align*}

Examples:

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

\begin{align*}
; \text{ list-of-grade } \to \text{ bool} \\
(\text{define} \ (\text{all-passed?} \ l)) \\
& \quad (\text{andmap passing-grade?} \ l))
\end{align*}
What about Non-Lists?

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

```scheme
; (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)))
```