Symbols

A `list-of-sym` program:

```
; eat-apples : 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:

; eat-apples : 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:

; eat-apples : 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

; filter-syms : (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

; filter-syms : (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

; filter-nums : (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

; filter-nums : (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:

; filter : ...
(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?

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

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

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

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

eat-apples \texttt{must return a list-of-sym, but by its signature,}
filter \texttt{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 **eat-apples**

\[; \; \text{eat-apples} : \; \text{list-of-sym} \rightarrow \text{list-of-sym} \]

**(define (eat-apples l)**

**(filter not-apple? l))**

\[; \; \text{not-apple?} : \; \text{sym} \rightarrow \text{bool} \]

**(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 \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 `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:

```
filter : 
((num -> bool) list-of-num 
 -> list-of-num)
OR
((sym -> bool) list-of-sym 
 -> 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}
filter :
  ((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}, \texttt{posns}, or \texttt{snakes}?
The True Signature of Filter

The real signature is

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

where \text{X} stands for any type

- The caller of \text{filter} gets to pick a type for \text{X}
- All \text{X}s in the signature must be replaced with the same type
The True Signature of Filter

The real signature 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 signature must be replaced with the same type

Data definitions need type variables, too:

\[
\begin{align*}
; & \text{A list-of-X is either} \\
; & - '() \\
; & - (\text{cons X list-of-X})
\end{align*}
\]
Using Filter

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

```scheme
(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) '()]
   [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 \texttt{feed-fish}:

\begin{verbatim}
; feed-fish : 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

\[
(define \ (map \ \text{CONV} \ l)
  \ (cond
    [(\text{empty?} \ l) \ '()]
    \else \ (\text{cons} \ \text{CONV} \ \text{first} \ l)
            \ (map \ \text{CONV} \ \text{rest} \ l)))]))
\]

; 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 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

  \[
  \text{map} : (X \rightarrow X) \text{ list-of-X} \rightarrow \text{list-of-X}
  \]
Posns and Distances

; 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

; 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:

\[
\begin{align*}
  &\text{(define (distances l)} \\
  &\quad \text{(map distance-to-0 l))}
\end{align*}
\]

The true signature of \text{map} is

\[
\text{map : (X -> Y) list-of-X -> list-of-Y}
\]

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

\begin{center}
\texttt{sum : list-of-num \rightarrow num}
\end{center}

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

How about \textit{sum}?

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

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

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

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

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

\[
(\text{cond}
\]

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

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

\[
(\text{COMB}
\]

\[
(\text{first} \ l)
\]

\[
(\text{combine-nums} \ (\text{rest} \ l) \ \text{base-n} \ \text{COMB}))]]))
\]
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 Foldr Function

; foldr : (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base 1)
  (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 '()' 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 '() 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 \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{verbatim}
; got-milk? : list-of-sym \to \text{bool}
(define (got-milk? l)
  (local [(define (is-milk? s)
             (symbol=? s 'milk))]
    (ormap is-milk? l)))
\end{verbatim}

\begin{verbatim}
; all-passed? : list-of-grade \to \text{bool}
(define (all-passed? l)
  (andmapmap passing-grade? l))
\end{verbatim}
What about Non-Lists?

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

; 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)))