Symbols

Our favorite \texttt{list-of-sym} program:

\begin{verbatim}
; eat-apples : list-of-sym \rightarrow 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)])]))
\end{verbatim}

- How about \texttt{eat-bananas}?
- How about \texttt{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:

```
; 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?
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?

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

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

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

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

\textit{eat-apples} must return a \textit{list-of-sym}, but by its contract, \textit{filter} might return a \textit{list-of-num}
The Contract 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 contract is too weak to define \textit{eat-apples}

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

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

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

The real contract is

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

where \( x \) stands for any type

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

Data definitions need type variables, too:

\[
\text{; A list-of-X is either}
\text{; \hspace{1cm} - empty}
\text{; \hspace{1cm} - (cons X empty)}
\]
Using Filter

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

New solution for HW 4 that works in Intermediate:

```
(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{inflate-by-4\%}: 

\begin{verbatim}
; inflate-by-4\% : list-of-num -> list-of-num
(define (inflate-by-4\% l)
  (cond
    [(empty? l) empty]
    [else (cons (* (first l) 1.04)
                     (inflate-by-4\% (rest l)))])
)
\end{verbatim}

Is there a built-in function to help?

\textbf{Yes:} \texttt{map}
(define (map CONV l)
  (cond
   [(empty? l) empty]
   [else (cons (CONV (first l))
               (map CONV (rest l)))])
)

(define (inflate-by-4% l)
  (local [(define (inflate-one n)
               (* n 1.04))]
    (map inflate-one l)))

; negate-colors : list-of-col -> list-of-col
(define (negate-colors l)
  (map negate-color))
The Contract for Map

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

- The \text{l} argument must be a list of \text{x}
- The \text{CONV} argument must accept each \text{x}
- If \text{CONV} returns a new \text{x} each time, then the contract for \text{map} is

\[
\text{map} : (X \to X) \text{ list-of-X } \to \text{ list-of-X}
\]
Another function from HW 4:

\[
; \text{distances} : \text{list-of-posn} \rightarrow \text{list-of-num} \\
(\text{define} \ (\text{distances} \ l) \\
\hspace{1em} (\text{cond} \\
\hspace{2em} [(\text{empty?} \ l) \ \text{empty}] \\
\hspace{2em} [(\text{cons?} \ l) \ (\text{cons} \ (\text{distance-to-0} \ (\text{first} \ l)) \\
\hspace{3em} (\text{distances} \ (\text{rest} \ l)))])]])
\]

The \text{distances} function looks just like \text{map}, except that \text{distances-to-0} is

\[
\text{posn} \rightarrow \text{num}
\]

not

\[
\text{posn} \rightarrow \text{posn}
\]
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

; modernize : list-of-pipe -> list-of-pipe
(define (modernize l)
  ; replaces 4 lines:
  (map modern-pipe l))

; modern-pipe : pipe -> pipe
...

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

; rob-car : car -> car
...
How about \textit{sum}?

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

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

But recall \textit{combine-nums}...

\begin{verbatim}
; 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)))])
\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)))]))

Useful for HW 5:

; total-blue : list-of-col -> num
(define (total-blue l)
  (local [(define (add-blue c n)
            (+ (color-blue c) n))]
    (foldr add-blue 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 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:

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

\[
\begin{align*}
; \text{all-passed?} &: \text{list-of-grade} \to \text{bool} \\
(\text{define} \ (\text{all-passed?} \ l) \\
 (\text{andmap} \ \text{passing-grade?} \ l))
\end{align*}
\]
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)))