Lists of Structures

30 September 2020
Recap: Lists of Numbers
Representing an arbitrary number of fish in an aquarium

Here’s a general strategy:

For 0 fish, use 

If you have a package and a new fish, put them together

To combine many fish, start with 
and add fish one at a time.
If all we’re interested in representing is how much each fish weighs, an aquarium can be represented as a list of numbers.

(Really, it would be a list of non-negative numbers.)
A ListOfNumbers is either
- '()
- (cons Number ListOfNumbers)

A recursive data definition requires at least one base case and at least one self-reference case.

'() is the base case

(cons Number ListOfNumbers) is the self-reference case (also called the recursive case).
A ListOfNumbers is either
- '()
- (cons Number ListOfNumbers)

A recursive data definition requires at least one base case and at least one self-reference case.

'() is the base case

(cons Number ListOfNumbers) is the self-reference case (also called the recursive case).
A ListOfNumbers is either
- '()
- (cons Number ListOfNumbers)

A recursive data definition requires at least one base case and at least one self-reference case.

'() is the **base case**

(cons **Number** ListOfNumbers) is the **self-reference case** (also called the recursive case).
For a list of numbers – or any list:

You can check if the list is empty using the `empty?` predicate.
You can get the `first` element.
You can get the `rest` of the list, which is itself a list.
(define L1 (cons "Hello"
             (cons "World" '())))

How can I get the second value in a list, e.g.,
"World" in L1?

(first (rest L1)) → "World"
;; A ListOfNumbers is either
;; - '()
;; - (cons Number ListOfNumbers)

;; ListOfNumbers -> ...
;; Template: Remind ourselves how to use the data
(define (lon-template lon)
  (cond [(empty? lon) ...]
        [(cons? lon)
         (... (first lon)
             (... (lon-template (rest lon))
                 ...))])))
;;; A ListOfNumbers is either
;;; - '()
;;; - (cons Number ListOfNumbers)

;;
;;; ListOfNumbers -> ...
;;; Template: Remind ourselves how to use the data
(define (lon-template lon)
  (cond [(empty? lon) ...]
        [(cons? lon)
         (... (first lon)
            (... (lon-template (rest lon))
             ...)])))
A ListOfNumbers is either
- '()
- (cons Number ListOfNumbers)

Template: Remind ourselves how to use the data
(define (lon-template lon)
  (cond [(empty? lon) ...]
        [(cons? lon)
          (... (first lon)
               (... (lon-template (rest lon))
                ...))]))
When writing a recursive function, trust the natural recursion will produce the right result for the rest of the list.

Don’t think “what am I going to do with the rest of the list?”
Think “what am I going to do with the result of the natural recursion?”
Design a function `feed-all` that feeds every fish in an aquarium 1 lb of food.
;; feed-all : ListOfNumbers -> ListOfNumbers
;; Feed every fish in lon 1lb of food
(check-expect (feed-all '()) '())
(check-expect (feed-all (cons 7 '()))) (cons 8 '()))
(check-expect
  (feed-all (cons 7 (cons 9 (cons 11 (cons 17 '()))))))
  (cons 8 (cons 10 (cons 12 (cons 18 '())))))))
(define (feed-all lon)
  (cond [(empty? lon) '()]
    [(cons? lon)
      (cons (+ 1 (first lon))
        (feed-all (rest lon)))]))
Design a function `large-fish`, which removes every fish that’s less than 5 lbs from an aquarium.
;; large-fish : ListOfNumbers → ListOfNumbers
;; Keep only fish that are >= 5 lbs
(check-expect (large-fish '()) '())
(check-expect (large-fish (cons 4 '())) '())
(check-expect (large-fish (cons 7 '())) (cons 7 '()))
(check-expect (large-fish (cons 4 (cons 7 '()))) (cons 7 '()))
(check-expect (large-fish (cons 7 (cons 4 '()))) (cons 7 '()))
;; large-fish : ListOfNumbers -> ListOfNumbers
;; Keep only fish that are >= 5 lbs
(check-expect (large-fish '()) '())
...
(check-expect (large-fish (cons 7 (cons 4 '())))
  (cons 7 '()))
(define (large-fish lon)
  (cond [[(empty? lon) ...]
        [(cons? lon)
          (... (first lon)
               ... (large-fish (rest lon)) ...)])
;; large-fish : ListOfNumbers -> ListOfNumbers
;; Keep only fish that are >= 5 lbs
(check-expect (large-fish '()) '())
...
(check-expect (large-fish (cons 7 (cons 4 '())))
  (cons 7 '()))
(define (large-fish lon)
  (cond 
    [(empty? lon) '()]
    [(cons? lon)
     (if (>= (first lon) 5)
      (cons (first lon)
        (large-fish (rest lon)))
      (large-fish (rest lon)))]))

When you have nested conditionals – cond or if – ask yourself if you can simplify.
;; large-fish : ListOfNumbers -> ListOfNumbers
;; Keep only fish that are >= 5 lbs
(check-expect (large-fish '()) '())
...
(check-expect (large-fish (cons 7 (cons 4 '())))
  (cons 7 '()))

(define (large-fish lon)
  (cond [(empty? lon) '()]
   [(<= (first lon) 5)
    (cons (first lon)
      (large-fish (rest lon)))]
   [else (large-fish (rest lon))])))
Lists of Positions
A ListOfPosns is either
- '() 
- (cons Posn ListOfPosns)
A `ListOfPosns` is either
- '()
- `(cons Posn ListOfPosns)`
A ListOfPosns is either
- '()  
- (cons Posn ListOfPosns)
A ListOfPosns is either
- '()
- (cons Posn ListOfPosns)

A Posn is
(make-posn Number Number)
A ListOfPosns is either
- '()
- (cons Posn ListOfPosns)

A Posn is
(\texttt{(make-posn Number Number)})

\texttt{(define (lop-template l)}
\begin{verbatim}
 (cond [(empty? l) ...]
       [(cons? l) ...]))
\end{verbatim}
A ListOfPosns is either
- '()
- (cons Posn ListOfPosns)

A Posn is
(make-posn Number Number)

ListOfPosns -> ...
(define (lop-template l)
  (cond [(empty? l) ...]
        [(cons? l)
         (... (first l)
             ... (rest l) ...)])
)
A ListOfPosns is either
- '(())
- (cons Posn ListOfPosns)

A Posn is
(make-posn Number Number)

ListOfPosns -> ...
(define (lop-template l)
  (cond [(empty? l) ...]
        [(cons? l)
          (... (first l)
               ... (lop-template (rest l))
               ...)]))
A ListOfPosns is either
- '(())
- (cons Posn ListOfPosns)

A Posn is
(make-posn Number Number)

ListOfPosns -> ...
(define (lop-template l)
 (cond [(empty? l) ...]
 [(!cons? l)
  (... (posn-template (first l))
  ... (lop-template (rest l))
  ...)])
)

Posn -> ...
(define (posn-template p)
  (... (posn-x p) (posn-y p)))
A ListOfPosns is either
- '()  
- (cons Posn ListOfPosn)

A Posn is
- (make-posn Number Number)

ListOfPosns -> ...
(define (lop-template l)
  (cond [[(empty? l) ...]]
        [[(cons? l)
          (... (posn-template (first l))
          ... (lop-template (rest l))
          ...)])])

Posn -> ...
(define (posn-template p)
  (... (posn-x p) (posn-y p)))
Design the function \texttt{flip-posns}, which flips the $x$ and $y$ parts of every \texttt{Posn} in a list of \texttt{Posns}.
;; flip-posns : ListOfPosns -> ListOfPosns
;; Flip every Posn over the diagonal
(check-expect (flip-posns '()) '())
(check-expect (flip-posns (cons (make-posn 6 5) '()))
  (cons (make-posn 5 6) '()))
(check-expect (flip-posns (cons (make-posn 8 6)
  (cons (make-posn 25 2)
    '())))
  (cons (make-posn 6 8)
    (cons (make-posn 2 25)
      '()))))
;; flip-posns : ListOfPosns -> ListOfPosns
;; Flip every Posn over the diagonal
(check-expect (flip-posns '()) '())
(check-expect (flip-posns (cons (make-posn 6 5) '()))
   (cons (make-posn 5 6) '()))
(check-expect (flip-posns (cons (make-posn 8 6)
   (cons (make-posn 25 2)
       '())))
   (cons (make-posn 6 8)
       (cons (make-posn 2 25)
       '())))

(define (flip-posns lop)
  (cond [(empty? lop) ...
        [(cons? lop)
          (... (flip-posn (first lop))
          (... (flip-posns (rest lop)) ...)])])

;; flip-posn : Posn -> ...
;; Flip Posn p
(define (flip-posn p)
  (... (posn-x p) (posn-y p)))
;; flip-posns : ListOfPosns -> ListOfPosns
;; Flip every Posn over the diagonal
(check-expect (flip-posns '()) '())
(check-expect (flip-posns (cons (make-posn 6 5) '()))
  (cons (make-posn 5 6) '()))
(check-expect (flip-posns (cons (make-posn 8 6)
  (cons (make-posn 25 2)
    '())))
  (cons (make-posn 6 8)
    (cons (make-posn 2 25)
      '())))

(define (flip-posns lop)
  (cond [(empty? lop) '()]
    [(cons? lop)
      (cons (flip-posn (first lop))
        (flip-posns (rest lop)))]))

;; flip-posn : Posn -> ...
;; Flip Posn p
(define (flip-posn p)
  (... (posn-x p) (posn-y p)))
;; flip-posns : ListOfPosns -> ListOfPosns
;; Flip every Posn over the diagonal
(check-expect (flip-posns '()) '())
(check-expect (flip-posns (cons (make-posn 6 5) '()))
      (cons (make-posn 5 6) '()))
(check-expect (flip-posns (cons (make-posn 8 6)  
                                     (cons (make-posn 25 2)  
                                         '()))))
      (cons (make-posn 6 8)  
            (cons (make-posn 2 25)  
                 '()))))

(define (flip-posns lop)
  (cond [(empty? lop) '()]  
         [(cons? lop)
          (cons (flip-posn (first lop))  
                (flip-posns (rest lop)))]))

;; flip-posn : Posn -> Posn
;; Flip Posn p
(define (flip-posn p)
  (make-posn (posn-y p) (posn-x p)))
Lists of Lists of Numbers
;; A ListOfLoN is either
;;  - '()
;;  - (cons ListOfNumbers ListOfLoN)
;; A ListOfLoN is either
;; - '()  Self-reference
;; - (cons ListOfNumbers ListOfLoN)
A ListOfLoN is either
- '()
- (cons ListOfNumbers ListOfLoN)

A ListOfNumbers is either
- '()
- (cons Number ListOfNumbers)
- (cons ListOfNumbers ListOfLoN)

Self-reference
A ListOfLoN is either
- '()
- (cons ListOfNumbers ListOfLoN)

A ListOfNumbers is either
- '()
- (cons Number ListOfNumbers)
Implement the function `sums`, which takes a list of list-of-numbers and produces a list of sums.
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
(check-expect (sums (cons '() '())) (cons 0 '()))
(check-expect (sums (cons (cons 60 (cons 80 '()))
                     (cons (cons 70 (cons 40 '()))
                     (cons (cons 50 '())
                     '()))))
     (cons 140 (cons 110 (cons 50 '()))))
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
(check-expect (sums (cons '() '())) (cons 0 '()))
(check-expect (sums (cons (cons 60 (cons 80 '()))
          (cons (cons 70 (cons 40 '()))
          (cons (cons 50 '())
               '()))))
          (cons 140 (cons 110 (cons 50 '(()))))
(define (sums lolon)
  (cond [[(empty? lolon) ...]
          [(cons? lolon)
           (... (first lolon)
                ... (sums (rest lolon)))]))
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
(check-expect (sums (cons '() '())) (cons 0 '()))
(check-expect (sums (cons (cons 60 (cons 80 '()))
                      (cons (cons 70 (cons 40 '()))
                      (cons (cons 50 '())
                          '()))))
                      (cons 140 (cons 110 (cons 50 '()))))
(define (sums lolon)
  (cond [(empty? lolon) ...]
        [(cons? lolon)
         ([... (first lolon)
           (... (sums (rest lolon)))])])
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
(check-expect (sums (cons '() '())) (cons 0 '()))
(check-expect (sums (cons (cons 60 (cons 80 '()))
  (cons (cons 70 (cons 40 '()))
  (cons (cons 50 '())
   '()))))
  (cons 140 (cons 110 (cons 50 '()))))

(define (sums lolon)
  (cond [(empty? lolon) ...
     [(cons? lolon)
       (... (sum (first lolon))
       ... (sums (rest lolon)))]))

;; sum : ListOfNumbers -> Number
;; Add up the numbers in lon
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
(check-expect (sums (cons '() '())) (cons 0 '()))
(check-expect (sums (cons (cons 60 (cons 80 '())))
   (cons (cons 70 (cons 40 '())))
   (cons (cons 50 '()))
   '()))))
   (cons 140 (cons 110 (cons 50 '()))))

(define (sums lolon)
  (cond [(empty? lolon) ...
         [(cons? lolon)
          (... (sum (first lolon))
               ... (sums (rest lolon)))]))

;; sum : ListOfNumbers -> Number
;; Add up the numbers in lon
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
(check-expect (sums (cons '() '())) (cons 0 '()))
(check-expect (sums (cons (cons 60 (cons 80 '()))
  (cons (cons 70 (cons 40 '()))
  (cons (cons 50 '())
   '()))))
  (cons 140 (cons 110 (cons 50 '()))))
(define (sums lolon)
  (cond [(empty? lolon) ...
    [(cons? lolon)
      (... (sum (first lolon))
        ... (sums (rest lolon)))]))

;; sum : ListOfNumbers -> Number
;; Add up the numbers in lon

This is an entry in our wishlist; we’ll get back to writing it when we finish sums.
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
(check-expect (sums (cons '() '())) (cons 0 '()))
(check-expect (sums (cons (cons 60 (cons 80 '()))
     (cons (cons 70 (cons 40 '()))
     (cons (cons 50 '()))
     '())))
     (cons 140 (cons 110 (cons 50 '()))))

(define (sums lolon)
  (cond [[(empty? lolon) '()]
         [(cons? lolon)
          (cons (sum (first lolon))
          (sums (rest lolon)))]])

;; sum : ListOfNumbers -> Number
;; Add up the numbers in lon
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
...

(define (sums lolon)
  (cond [(empty? lolon) '()]
        [(cons? lolon)
         (cons (sum (first lolon))
               (sums (rest lolon)))]))

;; sum : ListOfNumbers -> Number
;; Add up the numbers in lon
(check-expect (sum '()) 0)
(check-expect (sum (cons 10 (cons 20 '()))) 30)
;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
...

(define (sums lolon)
  (cond [(empty? lolon) '()]
        [(cons? lolon)
         (cons (sum (first lolon))
               (sums (rest lolon)))]))

;; sum : ListOfNumbers -> Number
;; Add up the numbers in lon
(check-expect (sum '()) 0)
(check-expect (sum (cons 10 (cons 20 '()))) 30)
(define (sum lon)
  (cond [(empty? lon) ...]
        [(cons? lon)
         (... (first lon)
              ... (sum (rest lon)))])

;; sums : ListOfLoNs -> ListOfNumbers
;; Sum all the lists in lolon
(check-expect (sums '()) '())
...

(define (sums lolon)
  (cond [(empty? lolon) '()
           [(cons? lolon)
            (cons (sum (first lolon))
                 (sums (rest lolon)))]])

;; sum : ListOfNumbers -> Number
;; Add up the numbers in lon
(check-expect (sum '()) 0)
(check-expect (sum (cons 10 (cons 20 '()))) 30)
(define (sum lon)
  (cond [(empty? lon) 0]
        [(cons? lon)
         (+ (first lon)
            (sum (rest lon)))]))
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith
Laney Strange