Pair and List Abstractions

Pairs (Constructors, Accessors, Contract, Operations)

(cons <x> <y>) -> <Pair> ; cons: T1, T2 -> Pair
(car (cons <x> <y>)) = <x> ; car: Pair -> T1
(cdr (cons <x> <y>)) = <y> ; cdr: Pair -> T2

Lists

; List = T X (nil ∪ List)
(list <elem_1> <elem_2> ... <elem_n>) ==
  (cons <elem_1> (cons <elem_2 ... (cons <elem_n> nil))..)

Operations

(null? <elem>) -> #t if <elem> is nil (#f)
(pair? <item>) -> #t if <item> is a pair
(define (atom? x) (not (pair? x)))

Examples

(define 1-to-4 (list 1 2 3 4))

1-to-4 -> (1 2 3 4) ; NOTE: printed rep for list!
(car (cdr 1-to-4)) -> 2

(pair? (cdddr 1-to-4)) -> #t
(cdddr 1-to-4) -> (4)
Accessing nth item in a list

- For n=0, \texttt{list-ref} should return the \texttt{car} of the list
- Otherwise, \texttt{list-ref} should return the (n-1)st item of the \texttt{cdr} of the list.

\begin{verbatim}
; list-ref: List, Int -> T
(define (list-ref lst n)
  (if (= n 0)
      (car lst)
      (list-ref (cdr lst) (- n 1)))))

(list-ref 1-to-4 3) -> 3
\end{verbatim}
Copy a list

(define (copy lst)
  (if (null? lst)
      nil ; base case
      (cons (car lst)
            (copy (cdr lst))))); recursion

Append two lists

• If list1 is the empty list, then result is just list2
• Otherise, append the cdr of list1 with list2, and cons the first element of list1 onto the front of the result

(define (append list1 list2)
  (cond ((null? list1) list2) ; base case
        (else
         (cons (car list1)
               (append (cdr list1)))))); recursion

(define 5-to-7 (list 5 6 7))
(append 5-to-7 1-to-4) -> (5 6 7 1 2 3 4)
Square all items in a list

(define (square-em lst)
  (if (null? lst)
      nil
      (cons (square (car lst))
            (square-em (cdr lst)))))

(square-em 1-to-4) -> (1 4 9 16)

Cube all items in a list

(define (cube-em lst)
  (if (null? lst)
      nil
      (cons (cube (car lst))
            (cube-em (cdr lst)))))

(cube-em (square-em 1-to-4)) -> (1 64 729 4096)
Map a procedure over a list

(define (map proc lst)
  (if (null? lst)
      nil
      (cons (proc (car lst))
            (map proc (cdr lst))))

(map square 1-to-4) -> (1 4 9 16)

Example: Scale a list

(define (scale-list lst factor)
  (map (lambda (x) (* x factor))
       lst))

(scale-list 1-to-4 10) -> (10 20 30 40)
Pick odd elements out of a list

```
(define (odds lst)
  (cond ((null? lst) nil)
    ((odd? (car lst))
      (cons (car lst)
        (odds (cdr lst))))
    (else
      (odds (cdr lst)))))

(odds 1-to-4) -> (1 3)
```

Filter

```
(define (filter pred lst)
  (cond ((null? lst) nil)
    ((pred (car lst))
      (cons (car lst)
        (filter pred (cdr lst))))
    (else (filter pred (cdr lst))))

(filter odd? 1-to-4) -> (1 3)
(filter even? 1-to-4) -> (2 4)
```
Enumerate Integers

(define (integers-between low high)
  (if (> low high)
      nil
      (cons low (integers-between (+ low 1) high)))))

(integers-between 5 9) → (5 6 7 8 9)
Examples

(map fib (integers-between 10 20))) -> (55 89 ... 6765)

Fibonacci numbers:

\[
Fib(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
Fib(n-1) + Fib(n-2) & \text{otherwise}
\end{cases}
\]

(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))

(filter even? (map fib (integers-between 10 20)))

(map fib (filter even? (integers-between 10 20)))
Add up integers in list

\[
\text{(define (addem-up lst)} \\
\qquad \text{(if (null? lst)} \\
\qquad \qquad 0 \\
\qquad \qquad (+ (car lst) \\
\qquad \qquad \quad \text{(addem-up (cdr lst))}) \\
\qquad \text{))})
\]

Length of a list

\[
\text{(define (length lst)} \\
\qquad \text{(if (null? lst)} \\
\qquad \qquad 0 \\
\qquad \qquad (+ 1 \\
\qquad \qquad \quad \text{(length (cdr lst))}) \\
\qquad \text{))})
\]
Accumulation

(define (accumulate op init lst)
  (if (null? lst)
      init
      (op (car lst)
           (accumulate op init (cdr lst))))))

(define (addem-up lst)
  (accumulate + 0 lst))

Length as Accumulation

(define (length lst)
  (accumulate (lambda (x y) (+ 1 y))
              0
              lst))

Append as Accumulation

(define (append list1 list2)
  (accumulate cons list2 list1))
Using our Tools:

(accumulate
  * 1
  (map fib
    (filter even?
      (integers-between 10 20))))
Using our Tools:

(define (easy lo hi)
  (accum * 1
    (map fib
      (filter even?
        (integers-between lo hi)))))

Not Using our Tools:

(define (hard lo hi)
  (cond ((> lo hi) 1)
    ((even? lo) (* (fib lo)
        (hard (+ lo 1) hi)))
    (else (hard (+ lo 1) hi)))))
TREES

(define tree (list (list 1 2) (list 3 4)))

(length tree) ->

(countleaves tree) ->
Countleaves

- countleaves of the empty list is 0
- Countleaves of a tree is countleaves of the car of that tree plus countleaves of the cdr of that tree
- countleaves of a leaf is 1

(define countleaves tree)
  (cond ((null? tree) 0) ; base case
       ((atom? tree) 1)  ; base case
       (else (+ (countleaves (car tree)); tree-recursion
                  (countleaves (cdr tree))))))
Scaling a Tree

(define (scale-tree tree factor)
    (cond ((null? tree) nil)
          ((atom? tree) (* tree factor))
          (else
           (cons (scale-tree (car tree) factor)
                 (scale-tree (cdr tree) factor)))))

(scale-tree (list 1 (list 2 (list 3 4) 5) (list 6 7))
           10)
-> (10 (20 (30 40) 50) (60 70))

Scaling - As a sequence of sub-trees

(define (scale-tree tree factor)
    (map (lambda (sub-tree)
            (if (atom? sub-tree)
                (* sub-tree factor)
                (scale-tree sub-tree factor)))
         tree))
Tree Procedures

; Compute the sum of the squares of the odd leaves in a tree.
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((atom? tree)
         (if (odd? tree) (square tree) 0))
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))))))

; Construct a list of all the even Fibonacci numbers Fib(k) where k is <= n
(define (even-fibs n)
  (define (next k)
    (if (< k n)
      nil
      (let ((f (fib k)))
        (if (even? f)
          (cons f (next (+ k 1)))
          (next (+ k 1))))))
  (next 0))
sum-odd-squares

- enumerates the leaves of a tree
- filters them, selecting the odd ones
- squares each of the selected ones
- accumulates the results using +, starting with 0

even-fibs

- enumerates the integers from 0 to n
- computes the Fibonacci number for each integer
- filters them, selecting the even ones, and
- accumulates the results using cons, starting with the empty list (not necessary, but shows the similarity in structure to sum-odd-squares)
sum-odd-squares

enumerator-leaves

filter

odd?

square

+ 0

accumulate

sequences (list) of numbers

even-fibs

0 n

integers-between

map

fib

even?

cons nil

filter

accumulate
Enumerate-tree

(define (enumerate-tree tree)
  (cond ((null? tree) nil)
        ((atom? tree) (list tree))
        (else (append (enumerate-tree (car tree))
                       (enumerate-tree (cdr tree)))))))
Using Tree/List Tools

(define (sum-odd-squares tree)
  (accumulate +
    0
    (map square
      (filter odd?
        (enumerate-tree tree))))

(define (even-fibs n)
  (accumulate cons
    nil
    (filter even?
      (map fib
        (integers-between 0 n))))
Nested Enumerations [Optional]

Extend idea of enumeration - generate more complicated lists, e.g. ordered pairs of positive integers $i$ and $j$, where $1 \leq j < i \leq n$.

\[
\begin{align*}
\text{(map (lambda (i))}
\quad \text{(map (lambda (j) (list i j))}
\quad \text{(integers-between 1 (- i 1)))}
\quad \text{(integers-between 1 n))}
\end{align*}
\]

\[
\begin{align*}
\text{(define (order-pairs n)}
\quad \text{(accumulate append}
\quad \quad \text{nil}
\quad \quad \quad \quad \quad \text{(map (lambda (i))}
\quad \quad \quad \quad \quad \quad \text{(map (lambda (j) (list i j))}
\quad \quad \quad \quad \quad \quad \quad \text{(integers-between 1 (- i 1)))}
\quad \quad \quad \quad \quad \quad \text{(integers-between 1 n))}
\end{align*}
\]
Summary

- Conventional Interfaces
  - lists (arbitrary length)
  - trees (arbitrary depth & length)
  - sequences (infinite length!!)

- Library of standard components

- Modular, clear programs