

Sequences as conventional interfaces

A process can often be decomposed into a sequence of stages

Examples

- compiling
- finding words that are common to two text files

Common types of stages

- enumerate
- filter
- transduce
- accumulate

Sum of squares of odd leaves in a tree

- 1) Make list of all leaves [enumerate]
- 2) Extract the odd leaves from list [filter]
- 3) Make list of the squares [transduce]
- 4) Sum up the squares [accumulate]

A definition not based on stages

```
(define (sum-of-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree)
             (square tree)
             0))
        (else (+ (sum-of-odd-squares
                  (car tree))
                  (sum-of-odd-squares
                   (cdr tree))))))
```

Definition based on stages

```
(define (sum-of-odd-squares tree)
```

```
  (accumulate +
```

```
    0
```

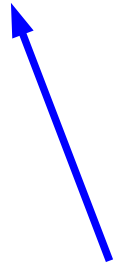
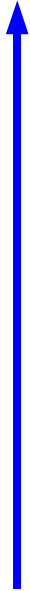
```
    (map square
```

```
      (filter odd?
```

```
        (enumerate-tree
```

```
          tree))))))
```

```
[accumulate] [transduce] [filter] [enumerate]
```



A general-purpose filter

```
(define (filter test list)
  (cond ((null? list) ())
        ((test (car list))
         (cons (car list)
               (filter test (cdr list))))
        (else (filter test (cdr list)))))

(filter even? (list 4 5 7 2 6 9 10 1)) -->
(4 2 6 10)
```

A general-purpose accumulator

```
(define (accumulate binary-op init-val list)
  (if (null? list)
      init-val
      (binary-op (car list)
                  (accumulate binary-op
                              init-val
                              (cdr list))))))
```

Accumulate examples

```
(accumulate + 0 (list 2 4 6 8)) --> 20
```

```
(accumulate * 1 (list 2 4 6 8)) --> 384
```

```
(accumulate cons () (list 2 4 6 8)) -->  
  (2 4 6 8)
```

```
(accumulate (lambda (x y) (if (< x y) y x))  
  0  
  (list 2 4 6 8)) --> 8
```

Enumerators are problem dependent

```
(define (enumerate-interval lo hi)
  (if (> lo hi)
      ()
      (cons lo (enumerate-interval (+ lo 1)
                                    hi)))))
```

```
(enumerate-interval 3 10) -->
(3 4 5 6 7 8 9 10)
```

Enumerating leaves

```
(define (enumerate-tree tree)
  (cond ((null? tree) ())
        ((not (pair? tree)) (list tree))
        (else
         (append (enumerate-tree (car tree))
                  (enumerate-tree
                   (cdr tree))))))
```

if $x \rightarrow ((1\ 3)\ 2\ (5\ (4\ 6)))$, then

```
(enumerate-tree x) --> (1 3 2 5 4 6)
```

Now we can do it!

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

if $x \rightarrow ((1\ 3)\ 2\ (5\ (4\ 6)))$, then

```
(sum-of-odd-squares x) --> 35
```

Why decompose into stages?

Because procedures that are decomposed into stages are easier to understand and to write

Data processing example

```
(define (salary-of-highest-paid-programmer
        records)
  (accumulate max
              0
              (map salary
                  (filter programmer?
                        records) ) ) )
```

Nested mappings

- maps are like loops, converting one list into another
- loops can be nested; so can maps
- Sometimes we want the results of inner lists appended together instead of returned as a list of lists

Example using nested maps

Find all triples $\langle i, j, i+j \rangle$ such that

- $1 \leq j$
- $j \leq i$
- $i \leq n$
- $i+j = \text{a perfect square}$

Strategy: find possible perfect squares, then find the ways that they can be written as sums of two integers

Decomposition of problem

- 1) Enumerate numbers from 1 through n
- 2) Make list of squares of these numbers
- 3) Save only the squares $\leq 2 \cdot n$
[$j \leq i \leq n$, so $i+j \leq n + n = 2n$]

Last stage

4) For each square that was saved, make a list of all the triples $\langle i, j, i+j \rangle$ such that $i+j =$ the square. Append these lists together rather than returning a list of lists of triples.

If $n = 7$, we want to get the list

$((3\ 1\ 4)\ (2\ 2\ 4)\ (8\ 1\ 9)\ (7\ 2\ 9)\ (6\ 3\ 9)\ (5\ 4\ 9))$

One possible solution

```
(define (square-sum-pairs n)
  (accumulate
    append
    ()
    (map (lambda (s)
          (map (lambda (j)
                (list (- s j) j s))
              (enumerate-interval
                1
                (* 0.5 s))))))
```

(continued)

```
(filter  
  (lambda (s) (<= s (* 2 n)))  
  (map square  
    (enumerate-interval 1 n))))))
```

A general-purpose procedure

```
(define (flatmap proc list)
  (accumulate append
    ()
    (map proc list)))
```

Another possible solution

```
(define (square-sum-pairs n)
  (flatmap (lambda (s)
            (map (lambda (j)
                  (list (- s j) j s))
                 (enumerate-interval
                    1
                    (* 0.5) )))
           (filter (lambda (s) (<= s (* 2 n)))
                  (map square
                     (enumerate-interval 1 n)))))
```

Computing permutations of a list of distinct numbers

Strategy: compute all the permutations that begin with the first element of list, all permutations that begin with second element of list, etc.

Permutation code

```
(define (permutations list)
  (if (null? list)
      (list ())
      (flatmap (lambda (x)
                 (map (lambda (p) (cons x p))
                      (permutations
                        (remove x list))))
               list)))

(define (remove item list)
  (filter (lambda (x) (not (= x item)))
          list))
```