アルゴリズムとデータ構造入門
2.データによる抽象の構築
2 Building Abstractions with Data

奥 乃 博

The First Commandment
Always ask `null?` as the first question in expressing any function.

The Second Commandment
Use `cons` to build lists.

The Third Commandment
When building a list, describe the first typical element, and then `cons` it onto the natural recursion.

(Friedman, et al. "*The Little Schemer*", MIT Press)

11月22日・本日のメニュー

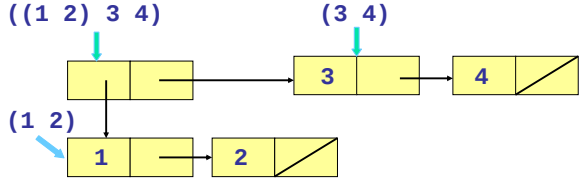
- 2 Building Abstractions with Data
- 2.2. Hierarchical Data and the Closure Property
  - 2.2.2 Hierarchical Structures
  - 2.2.3 Sequence as Conventional Interface
- 2.3 Symbolic Data
  - 2.3.1 Quotation

左上教科書表紙：http://mitpress.mit.edu/images/products/books/0262011530-f30.jpg

2.2.2 Hierarchical Structures

- **Tree（木）**

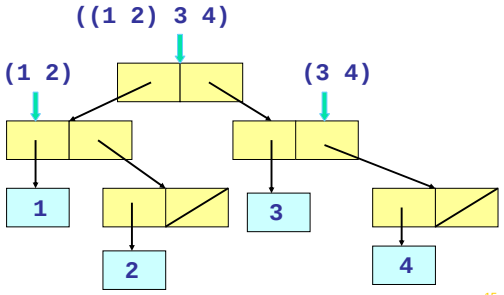
```
(cons (list 1 2) (list 3 4))
```



2.2.2 Hierarchical Structures

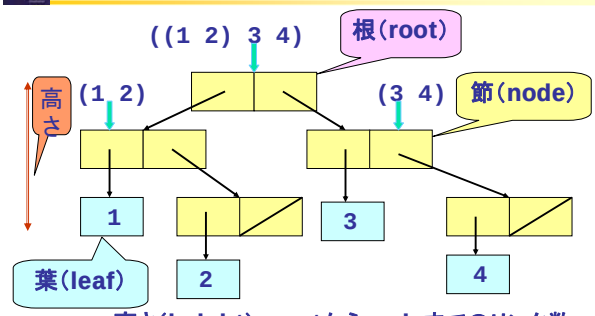
- Tree（木）と捉えると

```
(cons (list 1 2) (list 3 4))
```



15

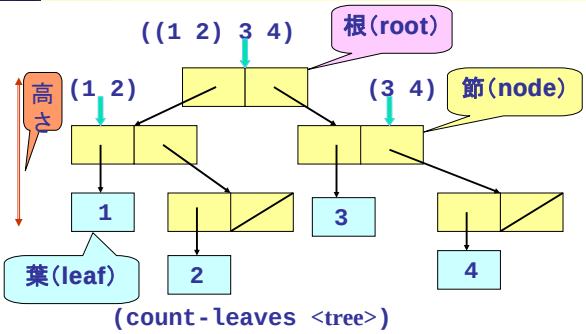
木の定義



高さ(height)：rootからnodeまでのリンク数
木の高さ：leafの高さの最大値

17

木とその上での演算




```
(count-leaves <tree>)
(max-height <tree>)
```

18

count-leaves · max-height

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x)) ))

(define (max-height x)
   (cond ((null? x) 0)
         ((not (pair? x)) 1)
         (else
           (+ 1 (max (max-height (car x))
                     (max-height (cdr x))
                )))))
```

19

木の写像

```
(define (scale-tree tree factor)
  (cond ((null? tree) nil)
        ((not (pair? tree)) (* tree factor))
        (else
          (cons (scale-tree (car tree) factor)
                (scale-tree (cdr tree) factor)
          ))))
```

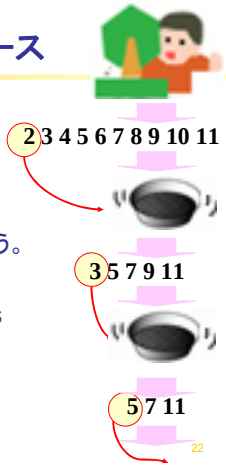
map を使用すると:

```
(define (scale-tree tree factor)
  (map (lambda (sub-tree)
         (if (pair? Sub-tree)
             (scale-tree sub-tree factor)
             (* sub-tree factor) ))
       tree ))
```

20

seq: 慣用インタフェース

- 処理間のインタフェース
- API（Application Program Interface）
- Parameter
- データ構造をインタフェースに使う。
- sequence を活用
- 例： The Sieve of Eratosthenes （エラトステネスの篩）



22

奇数の葉だけ2乗して和を取る

```
(define (count-leaves tree)
  (cond ((null? tree) 0)
        ((not (pair? tree)) 1)
        (else (+ (count-leaves (car tree))
                 (count-leaves (cdr tree)) ))
```

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

23

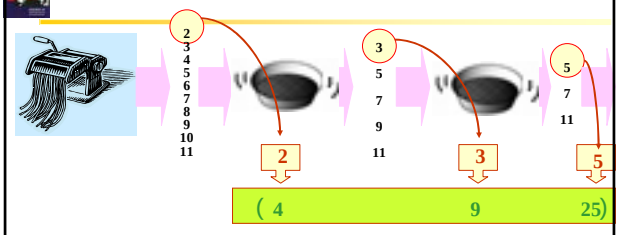
even-fibs 偶数のFibのリスト

```
(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) )
```

2つの手続きの
共通点は？

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

共通性の視点：素数の2乗を求める



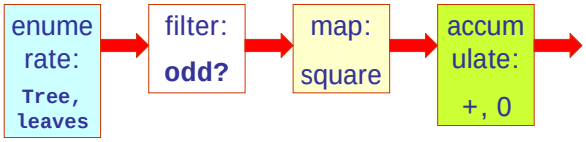
共通点を見る4つの基本手続き

- 数え上げ（enumerate）
- フィルタ（filter）
- 写像（map）
- 集約（accumulate）

25

4つの基本手続きから眺めると

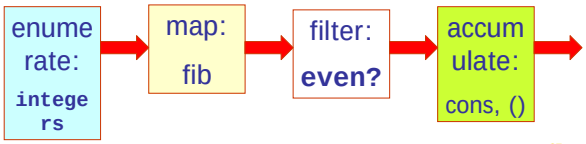
```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0) )
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))
              ))))
```



26

4つの基本手続きから眺めると

```
(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) )
```



27

4つの基本手続きをプログラム

```
(map square (list 1 2 3 4 5))
    ⇒

(define (filter predicate sequence)
  (cond ((null? sequence) nil)
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate
                       (cdr sequence)) ))
        (else (filter predicate
                      (cdr sequence))) ))

(filter odd? (list 1 2 3 4 5))
    ⇒
```

28

4つの基本手続きをプログラム(続)

```
(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial
                      (cdr sequence) ))))

(accumulate + 0 (list 1 2 3 4 5))
    ⇒
(accumulate * 1 (list 1 2 3 4 5))
    ⇒
(accumulate cons nil (list 1 2 3 4 5))
    ⇒
```

29

4つの基本手続きをプログラム(続)

整数の並びの数え上げ(enumerate): e.g. even-fibs

```
(define (enumerate-interval low high)
  (if (> low high)
      nil
      (cons low
           (enumerate-interval
                (+ low 1)
                 high ))))

(enumerate-interval 2 7)
    ⇒
(accumulate * 1 (enumerate-interval 1 5))
    ⇒
(accumulate + 0 (enumerate-interval 1 5))
    ⇒
```

30

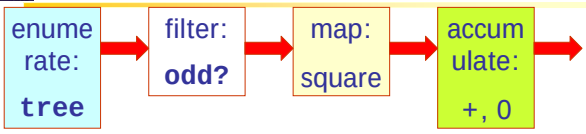
4つの基本手続きをプログラム(続)

木の数え上げ(enumerate): e.g. even-fibs

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

(enumerate-tree
    (list 1 (list 2 (list 3 4)) 5) )
    ⇒
```

31

4つの基本手続きを使ってプログラム




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

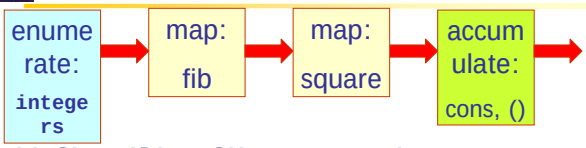
32

4つの基本手続きを使ってプログラム


```
(define (even-fibs n)
  (accumulate
     cons
     nil
     (filter
        even?
        (map
           fib
           (enumerate-interval 0 n) ))))
```

33

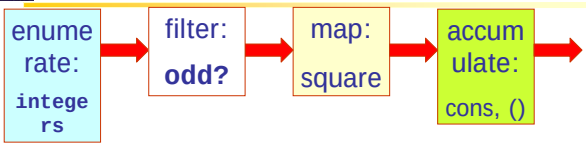
例題: list-fib-squares




```
(define (list-fib-squares n)
  (accumulate
     cons
     nil
     (map
        square
        (map
           fib
           (enumerate-interval 0 n) ))))
```

34

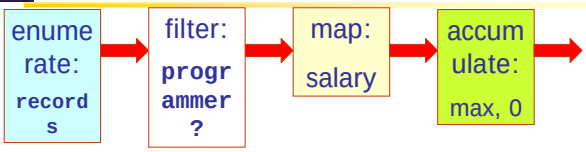
product-of-squares-of-odd-elements




```
(define (product-of-squares-of-odd-elements seq)
  (accumulate
     *
     1
     (map
        square
        (filter odd? seq) )))

(product-of-squares-of-odd-elements
    (list 1 2 3 4 5) ) ⇒
```

35

salary-of-highest-paid-programmer




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

データベース問い合わせ(query)言語の原型

36

11月22日・本日のメニュー

- 2 Building Abstractions with Data
- 2.2. Hierarchical Data and the Closure Property
  - 2.2.2 Hierarchical Structures
  - 2.2.3 Sequence as Conventional Interface
- **Intermission**
- 2.3 Symbolic Data
  - 2.3.1 Quotation

37

画像出所:
（一番上）http://www.greatbuildings.com/gbc/images/cid_1139983.150.jpg
(２番目)http://www.greatbuildings.com/gbc/images/cid_2161150.150.jpg
（3番目）http://www.greatbuildings.com/gbc/images/6a19666-Brooklyn_Bridge-s.150.jpg
（４枚目）http://www.physics.brown.edu/physics/demopages/Demo/waves/demo/tacoma.gif
(5枚目)http://www.civil.ibaraki.ac.jp/shmii/tacoma-narrows-bridge.JPG
（６枚目）http://www.vibrationdata.com/Resources/TB2B.JPG

Safety factor is *six times*.

- Suspension bridgesの設計の例
- John Roebling designed the Brooklyn Bridge which was built from 1869 to 1883.
- He designed the stiffness of the truss on the Brooklyn Bridge roadway to be *six times* what a normal calculation based on known static and dynamic load would have called for.
- *Galloping Gertie* of the Tacoma Narrows Bridge which tore itself apart in a windstorm in 1940, due to the nonlinearities in aerodynamic lift on suspension bridges modeled by the eddy spectrum.

38

画像出所:
(左側)http://www.nwrain.com/~newtsuit/recoveries/narrows/gg003.jpg
（右側）http://patmedia.net/kdnathan/Index/Gertie/Gertie%20Demo%20(unrestored).gif

Galloping Gertie ってなあに

- *Galloping Gertie* of the Tacoma Narrows Bridge which tore itself apart in a windstorm in 1940, due to the nonlinearities in aerodynamic lift on suspension bridges modeled by the eddy spectrum.

Horner's rule（Hornerの方法）

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ を計算するのに

$(\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0$ と変形する

coefficient-sequence: (a_5 … a_3 a_2 a_1 a_0)

```
(define (horner-eval x coefficient-
 sequence)
  (accumulate
     (lambda (this-coeff higher-term)
         (+ (* higher-term x) this-
 coeff) )
     0
```

41

行列(matrix)

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

```
((1 2 3 4)
 (4 5 6 6)
 (6 7 8 9) )
```

で表現

- (dot-product *v w*) $\sum_i v_i w_i$
- (matrix-*-vector *m v*) $t_i = \sum_j m_{ij} v_j$
- (matrix-*-matrix *m n*) $p_{ij} = \sum_k m_{ik} n_{kj}$
- (transpose *m*) $n_{ij} = m_{ji}$

42

行列(matrix)演算の実装

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 4 & 5 & 6 & 6 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

```
((1 2 3 4)
 (4 5 6 6)
 (6 7 8 9) )
```

で表現

```
(define (dot-product v w)
  (accumulate + 0 (map * v w)) )
```
$\sum_i v_i w_i$

```
(define (matrix-*-vector m v)
  (map (lambda () xxx) m) )
```
$t_i = \sum_j m_{ij} v_j$

```
(define (transpose mat)
  (accumulate-n xx xx mat) )
```
$n_{ij} = m_{ji}$

```
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map xxx m) ))
```
$p_{ij} = \sum_k m_{ik} n_{kj}$

43

accumulate-n

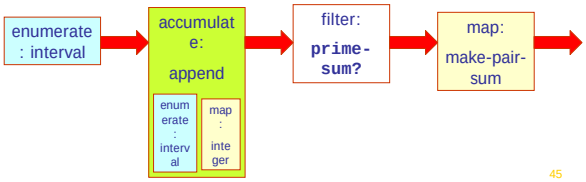
```
(accumulate-n + 0 '((1  2  3)
                    (4  5  6)
                    (7  8  9)
                    (10 11 12) ))

                ⇒ (22 26 30)

(define (accumulate-n op init seqs)
   (if (null? seqs)
       nil
       (cons (accumulate op init
                         (map car seqs) )
             (accumulate-n op init
                         (map cdr seqs) ))))
```

44

写像の入れ子（nesting of mapping）

1≦j<i≦n なる異なる正の整数 i,j に対して、i+jが素数となるものをすべて求める
n=6のとき

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
i+j	3	5	5	7	7	7	11



45

list of pairs of integers の作り方

```
(accumulate
   append
   nil
   (map
     (lambda (i)
       (map
         (lambda (j) (list i j))
         (enumerate-interval
                  1 (- i 1) )))
     (enumerate-interval 1 n) ))
```

この呼び出しパターンを手続きとして定義

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)) )
```

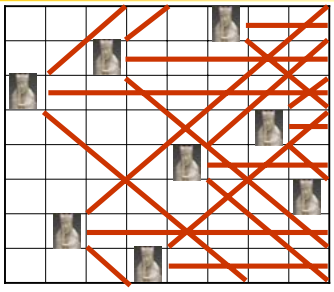
46

list of pairs of integers の作り方

```
(define (prime-sum? pair)
   (prime? (+ (car pair) (cadr pair)) )

(define (make-pair-sum pair)
   (list (car pair) (cadr pair)
         (+ (car pair) (cadr pair)) ))

(define (prime-sum-pairs n)
   (map make-pair-sum
        (filter prime-sum?
          (flatmap
             (lambda (i)
                (map (lambda (j) (list i j))
                     (enumerate-interval
                                1 (- i 1) )))
             (enumerate-interval 1 n) ))))
```

47

n-queens n人の女王の問題



8-queens puzzle

変種:すべての盤面をカバーする最小の女王の数は

- 女王は将棋の飛車角行
- お互いに取り合わないように配置

48

n-queens の作り方

```
(define (permutation s)
   (if (null? s)
       (list nil)
       (flatmap (lambda (x)
                  (map (lambda (p) (cons x p))
                       (permutation (remove x s)) ))
                s )))
(define (remove item sequence)
   (filter (lambda (x) (not (= x item))) sequence) )
(define (safe? k positions)
   (null?
     (filter
       (lambda (x)
          (not (or (= (cadr k) (cadr x))
                   (= (+ (car k) (cadr k))
                      (+ (car x) (cadr x)) )
                   (= (- (car k) (cadr k))
                      (- (car x) (cadr x)) ))))
       positions )))
(define (adjoin-position new k rest-of-q)
   (filter (lambda (x) (not (= x item))) sequence) )
```

49

list of pairs of integers の作り方

```
(define (queens n)
  (define (queen-cols k)
    (if (= k 0)
        (list empty-board)
        (filter
          (lambda (positions)
            (safe? k positions) )
          (flatmap
            (lambda (rest-of-q)
              (map (lambda (new-row)
                     (adjoin-position new-row
                                      k rest-of-q ))
                   (enumerate-interval 1 n) ))
            (queens-cols (- k 1)) ))))
  (queens-cols board-size) )
```

50

2.3.1 Quotation

- 定数データの表現: quote（引用）'
- (define foo (list 'a 'b))
  ⇒
- eq? 2つの要素が同一か。コピーは eq? ではない！
- (define (memq item x)
  (cond ((null? x) false)
  ((eq? item (car x)) x)
  (else (memq item (cdr x)))))
- (memq 'banana '(pear banana prune))
  ⇒
- (memq '(a b) '(pear (a b) prune))
  ⇒
- (memq foo (list 'pear foo 'prune))
  ⇒

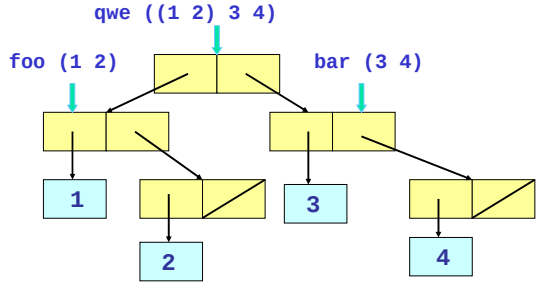
53

eq?

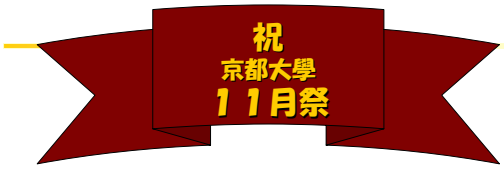
1. (eq? foo (list 1 2))
2. (eq? foo (car qwe))
3. (eq? bar (? qwe))

```
(define foo (list 1 2))
(define bar '(3 4))
(define qwe (cons foo bar))
```

qwe ((1 2) 3 4)



54



- 宿題は、次の5問:
- **Ex.2.21, 2.22, 2.28, 2.30, 2.32**
- (来週は:Ex.2.36, 2.37, 2.40, 2.42)

11月28日午後5時締切

55