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
2.2.3 並び　2.3 記号データ

奥 乃 博
大学院情報学研究科 知能情報学専攻
知能メディア講座 音声メディア分野
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/06/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

- 12月5日は中間試験
- 範囲:11月28日までの講義, 教科書～P68

11月28日・本日のメニュー

データによる抽象化
- **2.2.3 Sequences as Conventional Interfaces**
- Exercises 2.33~43
- 2.3 Symbolic Data

4

2つの手続きの構成は似ている?!

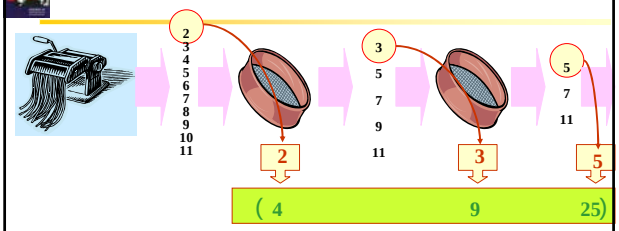
偶数のFibの和

```
(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) )
```

奇数の葉の二乗和

```
(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)

8

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))
              ))))
```

enumerate: Tree, leaves → filter: odd? → map: square → accumulate: +, 0

9

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) )
```

enumerate: integers → map: fib → filter: even? → accumulate: cons, ()

10

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))
    ⇒
```

11

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))
    ⇒
```

12

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))
    ⇒
```

13

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) )
```

⇒

14

Ex.1.32 Accumulation

```
(define (accumulate combiner null-value term a
    next b)
  (if (> a b)
      null-value
      (combiner (term a)
         (accumulate combiner null-value
               term (next a) next b ))))
```

```
(define (sum term a next b)
  (accumulate + 0 term a next b) )
(define (product term a next b)
  (accumulate * 1 term a next b) )
```

$$\sum_{i=a,next(i)}^{b} f(i)$$

```
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b)) ))
```

$$\prod_{i=a,next(i)}^{b} f(i)$$

17

Lisp/Scheme Programming十戒

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)

18

Lisp/Scheme Programming十戒

The Fifth Commandment

- When building a value with `+`, always use `0` for the value of the terminating line, for adding `0` does not change the value of an addition
- When building a value with `*`, always use `1` for the value of the terminating line, for multiplying by `1` does not change the value of a multiplication.
- When building a value with `cons`, always consider `()` for the value of terminating line.

The Sixth Commandment

Simplify only after the function is correct.

The Seventh Commandment

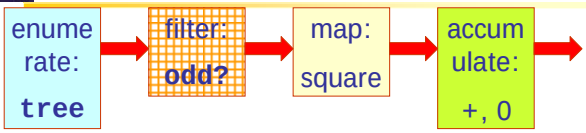
Recur on the *subparts* that are of the same nature

- On the sublists of a list.
- On the subexpressions of an arithmetic expression.

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

19

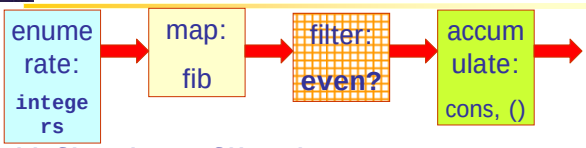
(Sum-odd-squares tree)




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

20

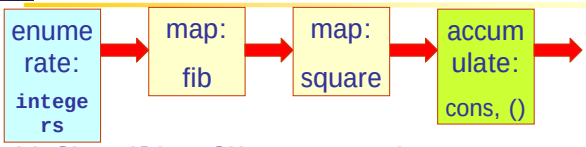
(even-fibs n)




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

21

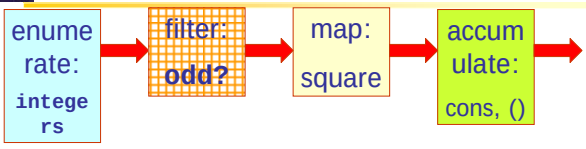
(list-fib-squares n)




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

22

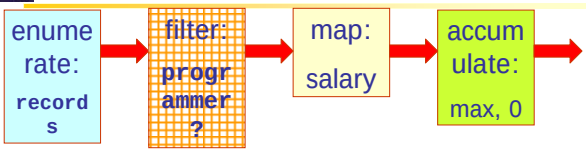
(product-of-squares-of-odd-elements seq)




```
(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) ) ⇒
```

23

(salary-of-highest-paid-programmer records)




```
(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

24

11月28日・本日のメニュー

データによる抽象化

- 2.2.3 Sequences as Conventional Interfaces
- **Exercises 2.33~43**
- 2.3 Symbolic Data

25

Ex2.33 Using accumulate

```
(define (my-map p sequence)
  (accumulate
     (lambda (x y) (cons (p x) y))
     nil
     sequence ))
(define (my-append seq1 seq2)
   (accumulate cons seq2 seq1) )
(define (my-length sequence)
   (accumulate
      (lambda (x y)
         (if (null? x) y (+ 1 y)) )
      0
      sequence ))
```

26

Ex2.34 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_n … 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
     coefficient-sequence ))
(horner-eval 2 (list 1 3 0 5 0 1))
```

⇒ 225

27

Ex2.37 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*) $n_{ij} = m_{ji}$

28

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)) )
(define (matrix-*-vector m v)
   (map (lambda () 補うこと) m) )
(define (transpose mat)
   (accumulate-n 補うこと 補うこと mat) )
(define (matrix-*-matrix m n)
   (let ((cols (transpose n)))
     (map 補うこと m) ))
```

$\sum_i v_i w_i$

$t_i = \sum_j m_{ij} v_j$

$n_{ij} = m_{ji}$

$p_{ij} = \sum_k m_{ik} n_{kj}$

29

accumulate-n

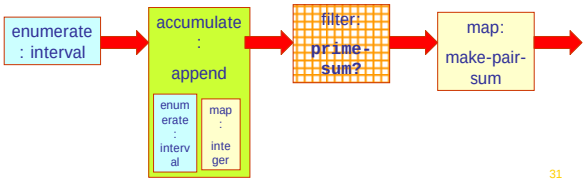
```
(accumulate-n + 0 '((1  2  3)
                    (4  5  6)
                    (7  8  9)
                    (10 11 12) ))

                ⇒ (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) )
       )))
```

30

写像の入れ子（nesting of mapping）

prime-sum-pairs 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



31

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)) )
```

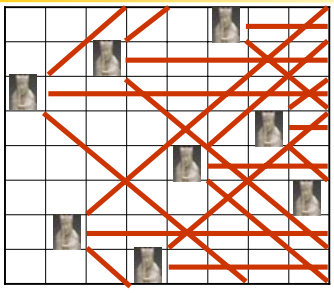
32

prime-sum-pairs は簡単

```
(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) ))))
```

33

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

34

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) )
```

行で取り合う

対角線

対角線

35

n-queen の手続き

```
(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 n) )
```

36

11月28日・本日のメニュー

データによる抽象化

- 2.2.3 Sequences as Conventional Interfaces
- Exercises 2.33~43
- **Intermission**
- **2.3 Symbolic Data**

38

Bugを最初に見つけたのは

- **Grace Murray Hopper (Dec. 9, 1906 – Jan. 1, 1992)**
- **1st women to receive a Ph.D in mathematics.**
- **Sep. 9, 1945. Mark II の Relay #70, Panel F で発見。**
- **bug はそれ以前から使用されていた。**
- **コンパイラ開発にも従事。**
- **NavyでCOBOL言語の検証ソフト開発**

40

A Bug (moth) in the Mark-II

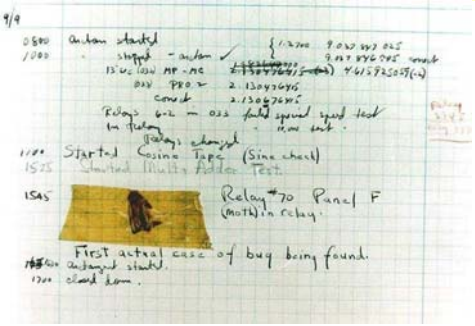


Photo #: NH 96566-KN (Color)
U.S. Naval Historical Center Photograph.

実験ノートをつけること。ルーズリーフはだめ

11月28日・本日のメニュー

データによる抽象化

- 2.2.3 Sequences as Conventional Interfaces
- Exercises 2.33~43
- **2.3 Symbolic Data**

48

2.3.1 Quotation

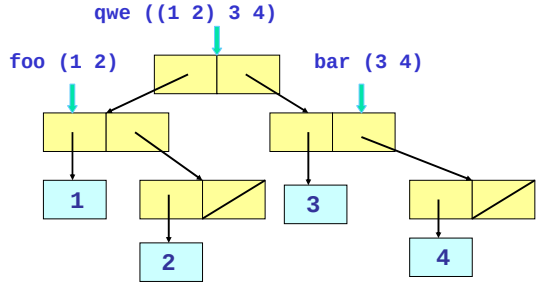
- 定数データの表現: quote（引用）'
- (define foo (list 'a 'b))
  ⇒
- eq? 2つの要素が同一か。コピーは eq? ではない！
- (define (memq item x)
  (cond ((null? x) false)
  ((eq? item (car x)) x)
  (else (memq item (cdr x)))))
- (memq 'banana '(pear banana prune))
  ⇒
- (memq '(a b) '(pear (a b) prune))
  ⇒
- (memq foo (list 'pear foo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

49

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))
```



50

append と reverse の例

- (append (list 1 2 3) ()) (1 2 3)
- (append (quote (1 2 3)) ()) (1 2 3)
- (append '(1 2 3) ()) (1 2 3)
- (append '(1 2 3) '(5 6 7)) (1 2 3 5 6 7)
- (append () '(a b c)) (a b c)

- (reverse '(1 2 3 4 5)) (5 4 3 2 1)
- (reverse '(ni ku i shi ku tsu u)) (u tsu ku shi i ku ni)
- (reverse '(に く い し く つ う)) (う つ く し い く に)

51

宿題はありません

- 12月2日は中間試験！
- 範囲:今日までの講義, 教科書～P68
- 全員満点を期待します.

D C!