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　階層データ構造と閉包性

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/09/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

次回は図形言語の説明をします．
必修課題のために必ず出席すること．

12月15日・本日のメニュー

データによる抽象化
2 Building Abstractions with Data
復習2.2. Hierarchical Data and the Closure Property（階層データ構造と閉包性）
2.2.1 Representing Sequences（並び）
2.3.1 Quote
2.2.2 Hierarchical Structures
2.2.3 Sequences as Conventional Interfaces

Lisp/Scheme Programming十戒

1. **The First Commandment**
   Always ask `null?` *as the first question in expressing any function.*
2. **The Second Commandment**
   Use `cons` *to build lists.*
3. **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）

3

数・cons (リスト) の生成関数

データ型	単位元	生成関数
自然数	0 （零）	successor
リスト	nil, () （空リスト）	cons
文字列	“” （空文字列）	string-append

Closure property（閉包性）of 自然数, cons, 文字列

データ型	単位元	演算	終了チェック
数	0	+	zero?
数	1	×	zero?
リスト	nil	cons, car, cdr	null?
文字列	“”	string-append	null-string?

TUT 無, MIT Scheme有

4

The Fourth Commandment

Always change at least one argument while recurring.
When recurring on a list of atoms, `lat`, use `(cdr lat)`.

When recurring on a number, `n`, use `(sub1 n)`.

教科書では `(- n 1)`

When recurring on a list of S-expressions, `l`, use `(car l)` and `(cdr l)`, if neither `(null? l)` and `(atom? (car l))` are true.

教科書では `(not (pair? (car l)))`

It must be changed to be closer to termination. The changing argument must be tested in the termination condition:

- when using `cdr`, test termination with `null?` and
- when using `sub1`, test termination with `zero?`.

5

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.

6

Lisp/Scheme Programming十戒

6. The Sixth Commandment
Simplify only after the function is correct.

7. 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.

8. The Eighth Commandment
Use help function to abstract from representations.

9. The Ninth Commandment
Abstract common patterns with a new function.

10. The Tenth Commandment
Build functions to collect more than one value at a time.

2.2 Hierachical Data and the Closure Property (閉包性)

- Pair (cell) 対(セル)
  (cons a b)
- Box-and-pointer notation

- List structure(Backus-Naur Form, BNF記法)

::= は定義　　| は代替

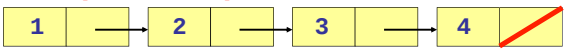
- <list> ::= <null> | (<element> . <element>)
- <element> ::= <name> | <number> | <list>

- **Closure property (閉包性) of cons**

2.2.1 Representing Sequences (シーケンス表現)

Sequence (列・並び) 1, 2, 3, 4

(1 2 3 4)




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

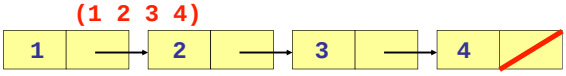
```
(1 . (2 . (3 . (4 . nil))))
```

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

Sequences表現の簡略化

Sequence（列・並び）の表現の簡略化

(1 2 3 4)



1. (xxx . nil) ⇒ (xxx)
2. (xxx . (yyy ･･･))⇒(xxx yyy ･･･)

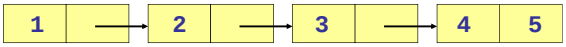
```
  (1 . (2 . (3 . (4 . nil))))
⇒ (1 . (2 . (3 . (4))))
⇒ (1 2 . (3 . (4)))
⇒ (1 2 3 . (4))
⇒ (1 2 3 4)
```

10

Sequences表現の簡略化（その2）

Sequence（列・並び）の表現の簡略化

(1 2 3 4 . 5)



1. (xxx . nil) ⇒ (xxx)
2. (xxx . (yyy ･･･))⇒(xxx yyy ･･･)

```
  (1 . (2 . (3 . (4 . 5))))
⇒ (1 2 . (3 . (4 . 5)))
⇒ (1 2 3 . (4 . 5))
⇒ (1 2 3 4 . 5)
```

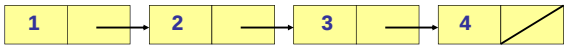
11

list-ref by cdring down

(list-ref items n)

if n=0, list-ref is car

otherwise, (n-1)st item of the rest



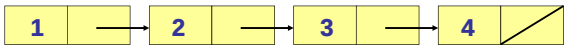

```
(define (list-ref items n)
   (if (= n 0)
       (car items)
       (list-ref (cdr items) (- n 1) )))
```

- cdring down the list (cdr down)
- Tail recursion に注意（自動的に iteration に変換）

12

length by cdring down

(length items)
if items is null?, length is 0
otherwise, 1 + length of the rest




```
(define (length items)
    (if (null? items)
        0
        (+ 1 (length (cdr items))) ))
```

- (+ (length (cdr items)) 1) との違い！

13

length : recursion and iteration versions

```
(define (length items)

(define (length-iter items)
```

14

cons up while cdring down

```
(define (append list1 list2) … )
   例 (append () '(a b c))
          (a b c)
   例 (append '(1 2) '(a b c))
          (1 2 a b c)
(define (reverse items) … )
   例 (reverse ())
          ()
   例 (reverse '(1 2 3 4 5))
          (5 4 3 2 1)
```

' は quote

18

12月15日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data

2.2. Hierarchical Data and the Closure Property（階層データ構造と閉包性）

2.2.1 Representing Sequences（並び）

2.3.1 Quote

2.2.2 Hierarchical Structures

2.2.3 Sequences as Conventional Interfaces

2.3.1 Quotation

- 定数データの表現: quote（引用）'
- (define foo (list 'a 'b))
  ⇒ (a b)
- (define foo '(a b)) とほぼ同じ

20

2.2.1 リスト演算の例での quote

```
(define (list-ref items n) … )
   1. (list-ref 0 (list 0 1 2))
   2. (list-ref 0 '(0 1 2))
   3.     1          （1.，2. ともにどちらも同じ）
   4. (list-ref 2 (list 0 1 2)
   5. (list-ref 2 '(0 1 2)
   6.     3
   7. (list-ref 5 (list 0 1 2)
   8. (list-ref 5 '(0 1 2)
   9.     ()
(define (length items) … )
   1. (length (list 1 2 3))
   2. (length '(1 2 3))
   3.     3
```

21

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)
```

22

append by consing up while cdring down

```
(append list1 list2)
  if list1 is null?, append is list2
  otherwise, cons the 1st item of list1 and
   append of the rest of list1 and list2

(define (append list1 list2)
    (if (null? list1)
         list2
        (cons (car list1)
              (append (cdr list1) list2)
        )))
```

24

reverse by consing up while cdring down

```
(reverse items)
  if items is null?, reverse is nil
  otherwise, append reverse of the rest of
   items and list of the 1st item of items

(define (reverse items)
    (if (null? items)
        nil
        (append (reverse (cdr items))
                (list (car items)) )))
```

25

Ex2.27 deep-reverse

```
(reverse '(1 (2 3) 4 ((5 6) 7)))
      (((5 6) 7) 4 (2 3) 1)
(deep-reverse '(1 (2 3) 4 ((5 6) 7)))
      ((7 (6 5)) 4 (3 2) 1)
(define (deep-reverse tree)
    (cond ((null? tree) nil)
          ((not (pair? tree)) tree)
          (t (append
                (deep-reverse (cdr tree))
                (list (deep-reverse
                            (car tree) )))
          )))
```

27

Ex2.28 fringe

```
(fringe '(1 (2 3) 4 ((5 6) 7)))
      (1 2 3 4 5 6 7)
(define (fringe tree)
    (cond ((null? tree) nil)
          ((not (pair? tree))
           (list tree) )
          (t (append
                (fringe (car tree))
                (fringe (cdr tree)) ))
    ))
```

28

Formal parameterの指定

```
(define (f x y . z) <body>)
  例    (f 1 2 3 4 5 6)
  x ⇐ 1, y ⇐ 2, z ⇐ (3 4 5 6)
(define (g . w) <body>)
  例    (g 1 2 3 4 5 6)
  w ⇐ (1 2 3 4 5 6)

(define f (lambda (x y . z)
                  <body> ))
(define g (lambda w <body>))
```

29

Arguments with dotted-tail notation

```
(define (f x y . z)
     <body> )

(define (sum . items)
   (define (iter items result)
      (if (null? items)
          result
          (iter (cdr items)
                (+ result (car items))
          )))
   (iter items 0) )

(define (sum . items)
   (define (recur items)
      (if (null? items)
          0
          (+ (car items) (recur (cdr items))) ))
   (recur items) )
```

30

Apply transformation to each element

```
(define (scale-list items factor)
    (if (null? items)
        nil
        (cons (* (car items) factor)
              (scale-list (cdr items) factor) )))
```

⇓

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

```
(map abs (list -10 2.5 -11.6 17))
  ⇒ (10 2.5 11.6 17)
(define (scale-list items factor)
    (map (lambda (x) (* x factor)) items) ))
```

31

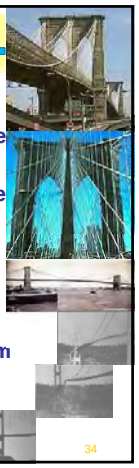
Apply transformation to each element

```
(map (lambda (x y) (+ x (* 2 y)))
     (list 1 2 3)
     (list 4 5 6) )
  ⇒ (9 12 15)
```

- 本当の map はもっと強力！

32

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.

34

12月15日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data

2.2. Hierarchical Data and the Closure Property（階層データ構造と閉包性）

2.2.1 Representing Sequences（並び）

2.3.1 Quote

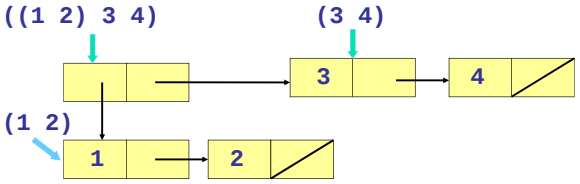
2.2.2 Hierarchical Structures

2.2.3 Sequences as Conventional Interfaces

2.2.2 Hierarchical Structures

- Tree（木）

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

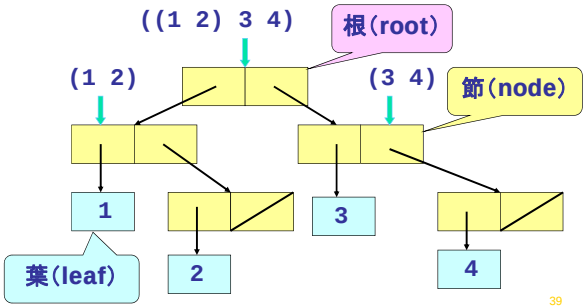


38

2.2.2 Hierarchical Structures

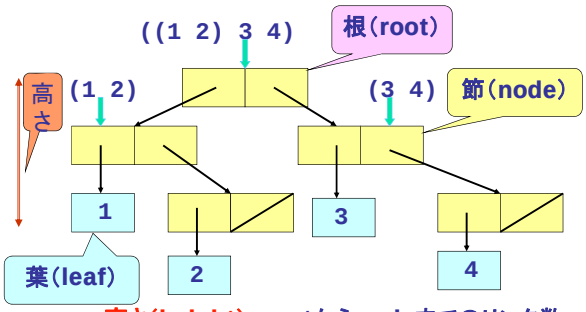
Tree（木）と捉えると

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



39

木の定義

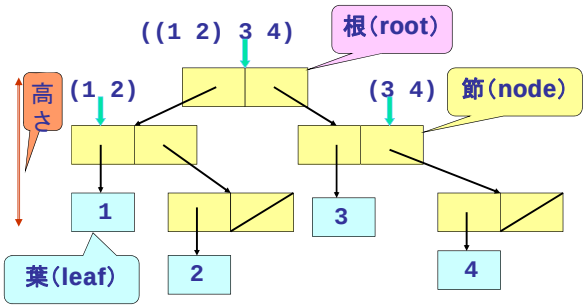


高さ(height): rootからnodeまでのリンク数

木の高さ:leafの高さの最大値

42

木とその上での演算



(count-leaves <tree>)

(max-height <tree>)

43

count-leaves

(count-leaves x)

If x is null?, count-leaves is 0

else if x is a leaf (not pair?), count-leaves is 1

otherwise add count-leaves of the car of x and count-leaves of the cdr of x

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

44

max-height

(max-height x)

If x is null?, max-height is 0

else if x is a leaf (not pair?), max-height is 1

otherwise add 1 to max of max-height of the car of x and max-height of the cdr of x

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

45

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))
                )))))
```

46

木の写像（leafに倍数をかける）

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

木をたどって手続きを適用する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 ))
```

48

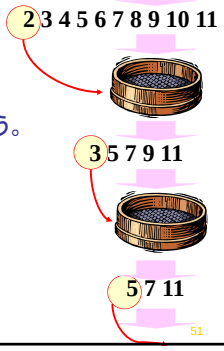
Ex2.32 powerset

```
A = (1 2 3)
2^A = (() (3) (2) (2 3) (1) (1 3) (1 2)
        (1 2 3) )
(define (powerset a)
  (if (null? a)
      (list nil)
      (let ((rest (powerset (cdr a)))
        (append
           rest
           (map
             (lambda (x)
               (append (list (car a)) x))
             rest )))))
```

49

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での受け渡し
- データ構造をインタフェースに使う。
- sequence を活用
- 例： 素数を求めるための The Sieve of Eratosthenes（エラトステネスの篩）



51

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)) ))
```

を基に，奇数の葉だけ2乗して和を取る

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

52

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) )
```

偶数のfibの和

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))
               ))))
```

奇数の葉の二乗和

Ex.1.32 Accumulationを思い出そう

```
(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) )
```

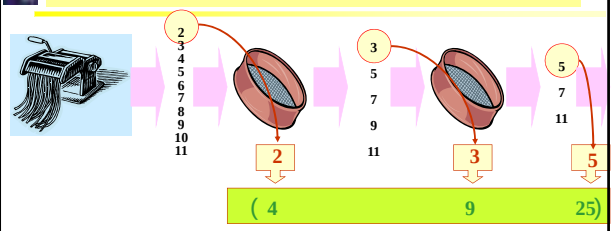
$$\sum_{i=a,next(i)}^{b} f(i)$$

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

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

54

共通性の視点：素数の2乗を求める



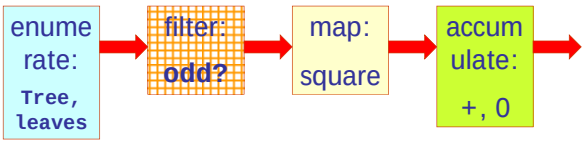
共通点を見る4つの基本手続き

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

55

4つの基本手続きから眺めると

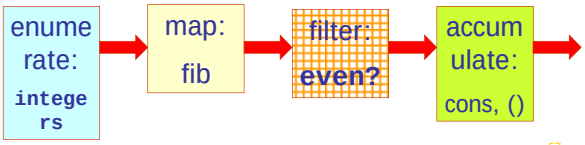
```
(define (sum-odd-squares tree)
  (cond ((null? tree) 0)
        ((not (pair? tree))
         (if (odd? tree) (square tree) 0) )
        (else (+ (sum-odd-squares (car tree))
                 (sum-odd-squares (cdr tree))
              ))))
```



56

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) )
```



57

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))
    ⇒
```

58

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))
    ⇒
```

60

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))
    ⇒
```

62

4つの基本手続きをプログラム(続)

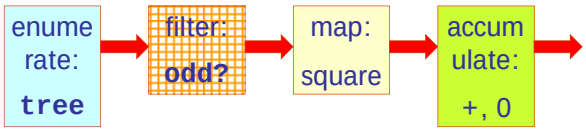
木の数え上げ(enumerate): e.g. even-fibs

```
(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) )
    ⇒
```

64

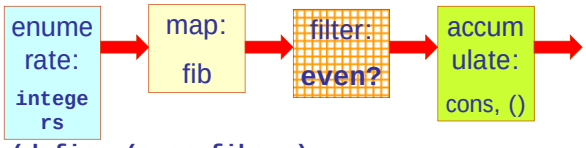
4つの基本手続きを使ってプログラム




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

66

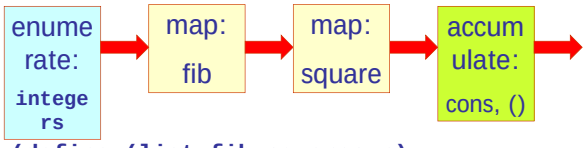
4つの基本手続きを使ってプログラム




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

67

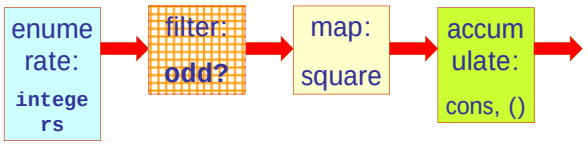
例題: list-fib-squares




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

68

product-of-squares-of-odd-elements

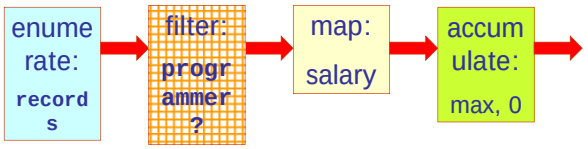



```
(define (product-of-squares-of-odd-elements seq)
  (accumulate
     *
     1
     (map
        square
        (filter odd? seq) )))

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

69

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

70

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 ))
```

71

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

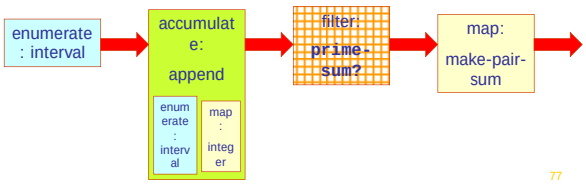
73

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



77

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)) )
```

78

list of pairs of integers の作り方

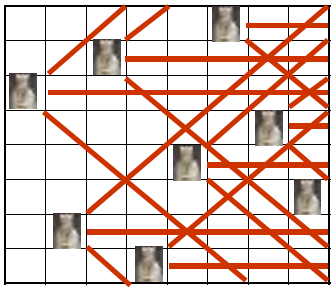
```
(define (prime-sum? pair)
   (prime? (+ (car pair) (cadr pair)) )

(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) ))))
```

79

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

80

n-queens の作り方：数え上げ

```
(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) )
```

81

n-queens の本体

```
(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) )
```

queen-cols returns the sequence of all ways to place queens in the first k columns of the board.

82

宿題：12月22日正午締切

- 4つの基本手続の考え方を習得
- 宿題は、次の3題：
  Ex.2.33, 2.34, 2.35.
- 随意宿題　2.42 (n-queens)