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
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/10/IntroAlgDs/
okuno@i.kyoto-u.ac.jp, okuno@nue.org
阿曽 慎平(M1) 奥乃研・音楽G (10号館4階奥乃1研)
平澤 恭治(M1) 奥乃研・ロボット聴覚G (10号館4階奥乃1研)

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.

11月30日・本日のメニュー

2 Building Abstractions with Data
- 2.2. Hierarchical Data and the Closure Property(階層データ構造と閉包性)
- リスト処理の復習
- 2.2.1 Representing Sequences(並び)
- 2.2.2 Hierarchical Structures
- 2.2.3 Sequences as Conventional Interfaces

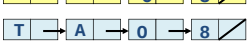
12月21日 中間試験を実施

いきなり復習だ

リスト処理手続きを一挙に学ぶ.
1. equal?
2. length
3. copy, deep-copy
4. append, reverse, deep-reverse
5. flatten, fringe
6. butlast
7. member?, assoc

いきなり復習: equal?

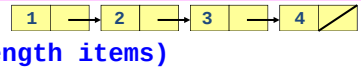
```
(equal? items1 items2)
  if items1 is eq? to items2, return the value of eq?
  otherwise if items1 is non-pair,
          return the value of items1 is eq? to items2
  otherwise
    if the 1st of items1 is equal? to the 1st of items2,
      return the value of the rest of items1 is equal? to
      that of items2
```




```
(define (equal? items1 items2)
   (cond ((eq? items1 items2))
         ((not (pair? items1)) (eq? items1 items2))
         ((not (pair? items2)) (eq? items1 items2))
         ((equal? (car items1) (car items2))
          (equal? (cdr items1) (cdr items2)) ))
```


いきなり復習: length

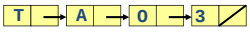
```
(length items)
    if items is null?, length is 0
    otherwise, 1 + length of the rest
```




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


いきなり復習: copy

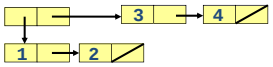
```
(copy items)
  if items is null? or non-pair, return it
  otherwise
    cons the 1st of items and copy of the rest of items
```




```
(define (copy items)
   (cond ((null? items) items)
         ((not (pair? items)) items)
         (else (cons (car items)
                     (copy (cdr items)) ))))
```

いきなり復習: deep-copy

(deep-copy items)
  if items is null? or non-pair, return it
  otherwise cons copy of the 1st of items
      and copy of the rest of items




```
(define (deep-copy items)
   (cond ((null? items) items)
         ((not (pair? items)) items)
         (else (cons (deep-copy (car items))
                     (deep-copy (cdr items))
               ))))
```


いきなり復習: append

(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)
          )))
```

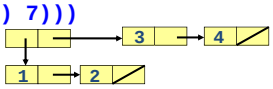

いきなり復習: reverse

(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)) )))
```


いきなり復習: deep-reverse

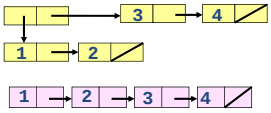
```
(reverse '(1 (2 3) 4 ((5 6) 7)))
      (((5 6) 7) 4 (2 3) 1)
(deep-reverse '(1 (2 3) 4 ((5 6) 7)))
      ((7 (6 5)) 4 (3 2) 1)
```




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


いきなり復習: fringe, flatten

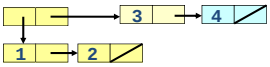
```
(fringe '(1 (2 3) 4 ((5 6) 7)))
      (1 2 3 4 5 6 7)
```




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


いきなり復習: last, but-last

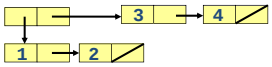
```
(last '(1 (2 3) 4 ((5 6) 7)))
      (((5 6) 7))
(but-last '(1 (2 3) 4 ((5 6) 7)))
      (1 (2 3) 4)
```




```
(define (last tree)
   (if (null? tree)
       nil
       (list (car (reverse tree))) ))

(define (but-last tree)
   (if (or (null? tree) (null? (cdr tree)))
       nil
       (reverse (cdr (reverse tree)) ))
```

いきなり復習: member

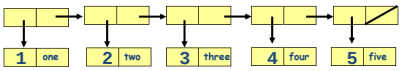
```
(member x items)
  if items is null?, member is false
  otherwise if x is the 1st of items, member is items
  otherwise x is member of x in the rest of items
```




```
(define (member x items)
   (cond ((null? items) nil)
         ((equal? x (car items)) items)
         (else (member x (cdr items))) ))
```


いきなり復習: assoc

```
(assoc x alist)
 if alist is null?, assoc is false
  otherwise if the 1st of items is pair? and x is its car,
          assoc is 1st of alist
  otherwise x is member of x in the rest of items
```




```
(define (assoc x alist)
   (cond ((null? alist) nil)
         ((and (pair? (car alist))
               (equal? x (caar alist)) )
          (car alist) )
         (else (assoc x (cdr items))) ))
```


いきなり復習: listの各要素への演算

```
(define (absolute-list items)
    (if (null? items)
        nil
        (cons (abs (car items))
              (absolute-list (cdr items)) )))

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

(absolute-list (list -10 2.5 -11.6 17))
   ⇒ (10 2.5 11.6 17)

(scale-list (list -10 2.5 -11.6 17) 10)
   ⇒ (-100 25 -116 170)
```


listの各要素への演算の抽象化

```
(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) )
(define (scale-absolute-list items factor)
    (map (compose (lambda (x) (* x factor)) abs) items) )
```


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) )
```


11月30日・本日のメニュー

2 Building Abstractions with Data

- 2.2. Hierarchical Data and the Closure Property(階層データ構造と閉包性)
- リスト処理の復習
- **2.2.1 Representing Sequences(並び)**
- 2.2.2 Hierarchical Structures
- 2.2.3 Sequences as Conventional Interfaces

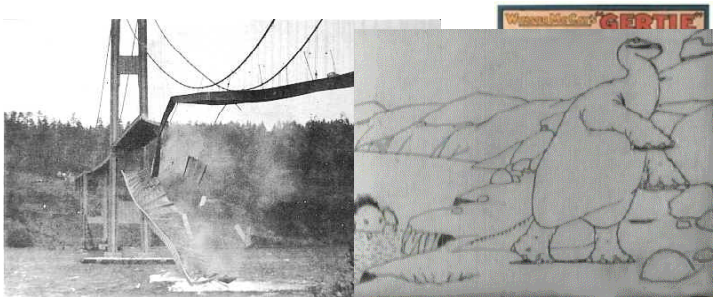
Safety factor is six times.

- 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.

51

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.



11月30日・本日のメニュー

2 Building Abstractions with Data

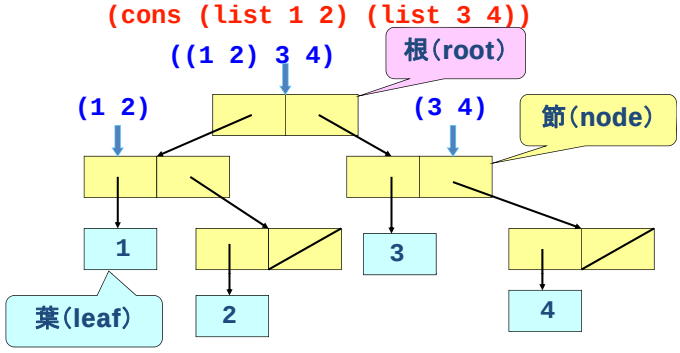
- 2.2. Hierarchical Data and the Closure Property（階層データ構造と閉包性）
- 2.2.1 Representing Sequences（並び）
- **2.2.2 Hierarchical Structures**
- 2.2.3 Sequences as Conventional Interfaces

54

2.2.2 Hierarchical Structures

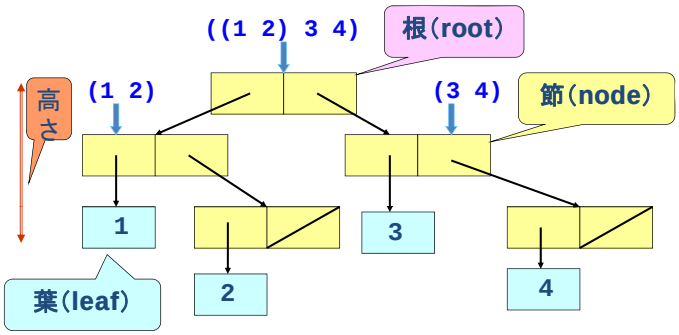
Tree（木）と捉えると

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



56

木の定義

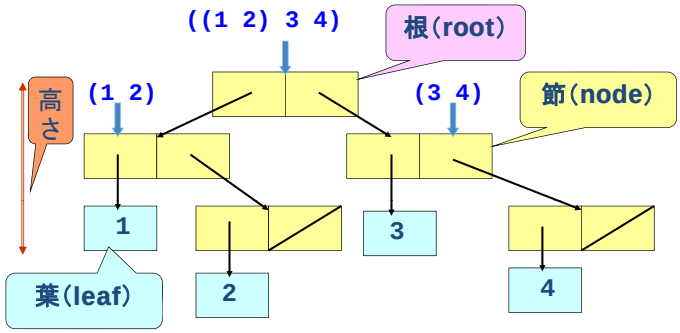


高さ(height)： rootからnodeまでのリンク数

木の高さ： leafの高さの最大値

57

木とその上での演算




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

58

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))
               ))))
```


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))
           )))))
```


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 ))
```


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 )))))
```

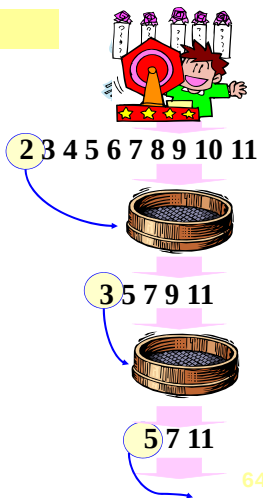

11月30日・本日のメニュー

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)
- 2.2.2 Hierarchical Structures
- 2.2.3 **Sequences as Conventional Interfaces**

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)



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)) ))
   ))
```


even-fibs 偶数のfibのリスト

偶数のfibの和

2つの手続きの共通点は？

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


Ex.1.32 Accumulationを思い出そう （復習）

```
(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 ))))
```

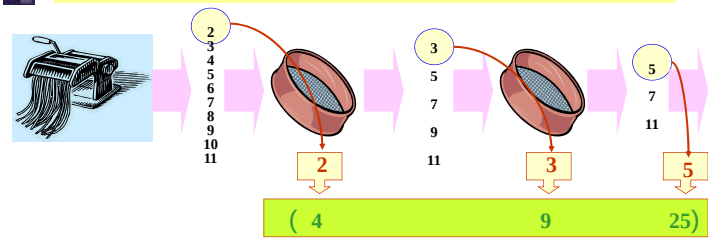
```
(define (sum term a next b)
  (accumulate + 0 term a next b) )
(define (product term a next b)
  (accumulate * 1 term a next b) )
```

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

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

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

共通性の視点：素数の2乗を求める

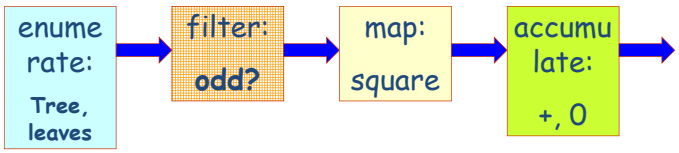


共通点を見る4つの基本手続き

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

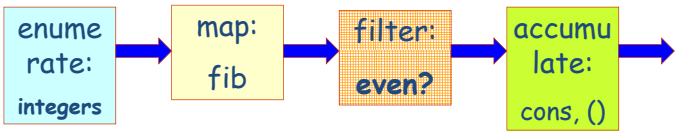
4つの基本手続きから眺めると

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



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) )
```



4つの基本手続きをプログラム: filter

```
(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))
    ⇒
```


4つの基本手続きをプログラム: accumulate

```
(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))
    ⇒
```


4つの基本手続きをプログラム: enumerate

整数の並びの数え上げ(enumerate): e.g. even-fibs

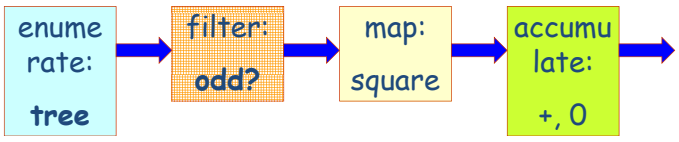
```
(define (enumerate-interval low high)
   (if (> low high)
      nil
      (cons low
           (enumerate-interval
                (+ low 1)
               high ))))

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


4つの基本手続きをプログラム: enumerate(続)

木の数え上げ(enumerate): e.g. even-fibs

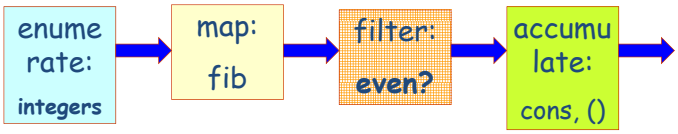
```
(define (enumerate-tree tree)
   (cond ((null? tree) nil)
         ((not (pair? tree)) (list tree))
         (else (append
                (enumerate-tree (car tree))
                (enumerate-tree (cdr tree))
               ))))

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


4つの基本手続きを使ってプログラム



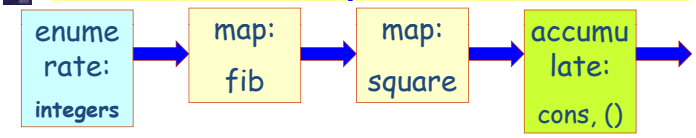

```
(define (sum-odd-squares tree)
   (accumulate
     +
     0
     (map
        square
        (filter
           odd?
           (enumerate-tree tree) ))))
```


4つの基本手続きを使ってプログラム



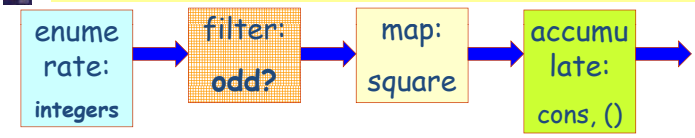

```
(define (even-fibs n)
   (accumulate
     cons
     nil
     (filter
        even?
        (map
           fib
           (enumerate-interval 0 n) ))))
```

例題: list-fib-squares



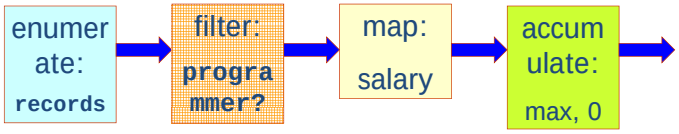

```
(define (list-fib-squares n)
   (accumulate
     cons
     nil
     (map
        square
        (map
           fib
           (enumerate-interval 0 n) ))))
```


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) ) ⇒ 225
```


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

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 ))
```


宿題:12月7日正午締切

1. Ex.2.33, 2.34, 2.37
2. Program, プログラムの解説, 実行例をまとめたレポート(pdf) を

   SICP-8@zeus.kuis.kyoto-u.ac.jp に送付

- 友達に教えてもらったら, その人の名前を明記すること. Webは出展を明記. (otherwise『同じ』回答は減点)

DON'T PANIC!

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)) )
```