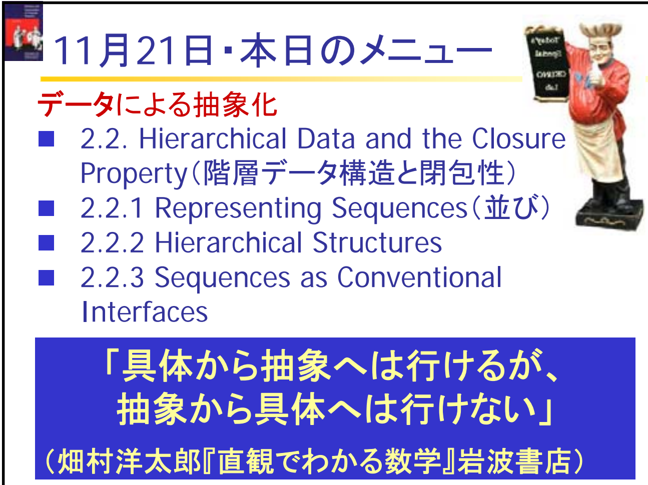
アルゴリズムとデータ構造入門

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/06/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

- 12月5日は中間試験
- 範囲は11月28日までの講義＆教科書



The road to wisdom?

Don Knuth

http://www-cs-faculty.stanford.edu/~knuth/graphics.html

The road to wisdom? Well it's plain & simple to express: Err and err and err again but less and less and less. PIET HEIN

訂正・補足

- Power Set of A: 2^A
- TUT-Scheme (TUS) の時間測定
  - (time <*form*>)
  - (time (factorial 1000))
  - (time (null? (factorial 10000)))
- TUT-Scheme (TUS) の乱数 (random)はない.
  - http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/05/IntroAlgDs/random.lsp
  - tus2, tustk2 ではあり.
- cygwin上のTUS: 改行が nl (¥n) でないといけない
  - ファイルの改行が "cr nl" か"nl"だけかをチェッ

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

- 基本データ構造: Pair (cell) 対(セル)
  (cons a b)
- Box-and-pointer notation

a	b

- 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 ⇒ リスト

6

2.2.1 Representing Sequences(表現)

- Sequence（列・並び） 1, 2, 3, 4
  (1 2 3 4)

1	→	2	→	3	→	4	/

- ```
  (cons 1
    (cons 2
      (cons 3
        (cons 4 nil)
      )))
  ```
  構築子を使うと
- `(1 . (2 . (3 . (4 . nil))))` 入出力による表現
- `(list 1 2 3 4)`

7

Sequences表現の簡略化

- Sequence（列・並び） の表現の簡略化
  (1 2 3 4 . 5)

1	→	2	→	3	→	4	5

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

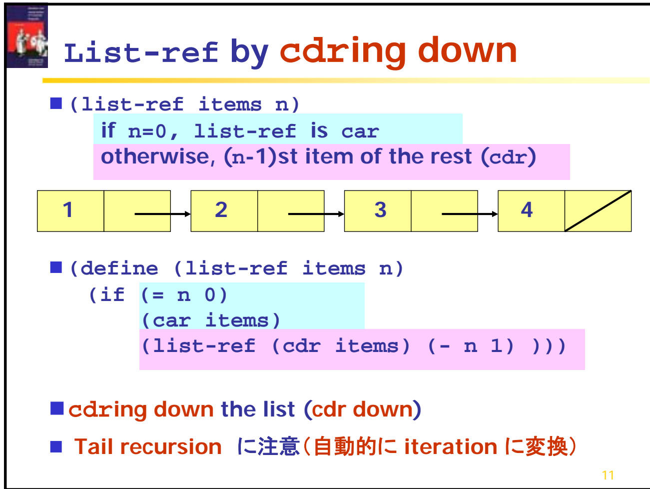
```
  (1 . (2 . (3 . (4 . 5))))
⇒ (1 2 . (3 . (4 . 5)))
⇒ (1 2 3 . (4 . 5))
⇒ (1 2 3 4 . 5)
```

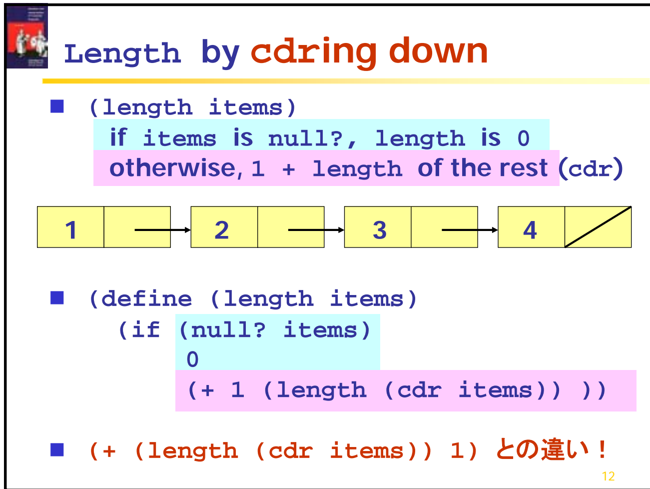
8

2.2.1 List operations(リスト演算)

- (define (list-ref items n) …)
  - (list-ref (list 0 1 2) 0)
  - 0
  - (list-ref (list 0 1 2) 2)
  - 2
  - (list-ref (list 0 1 2) 5)
  - ()
- (define (length items) …)
  - (length ())
  - 0
  - (length (list 1 2 3))
  - 3

9
```





length : recursion and iteration versions

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

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

13

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)

14

Ex2.19 `cc` change of coins

```
■ (define us-coins (list 50 25 10 5 1))
■ (define uk-coins (list 100 50 20 10 5 2 1 0.5))
■ (define (cc amount coin-values)
    (cond ((= amount 0) 1)
          ((or (< amount 0) (no-more? coin-values))
           0 )
          (else
            (+ (cc amount
                   (except-first-demonination coin-values))
               (cc (- amount
                      (first-denomination coin-values) )
                   coin-values )))))
■  (define (except-first-denomination coins)
      (cdr coins) )
■  (define (first-denomination coins)
      (car coins) )
■  (define (no-more? coins)
      (null? coins) )
```

15

11月21日・本日のメニュー

データによる抽象化
- 2.2. Hierarchical Data and the Closure Property
- **2.2.1 Representing Sequences**
- **2.3.1 Quote**
- 2.2.2 Hierarchical Structures
- 2.2.3 Sequences as Conventional Interfaces

16

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

17

2.2.1 リスト演算の例での quote

- (define (list-ref items n) …)
  - (list-ref (list 0 1 2) 0)
  - (list-ref '(0 1 2) 0)
  - 0
  - (list-ref (list 0 1 2) 1)
  - (list-ref '(0 1 2) 1)
  - 1
  - (list-ref (list 0 1 2) 5)
  - (list-ref '(0 1 2) 5)
  - ()
- (define (length items) …)
  - (length (list 1 2 3))
  - (length '(1 2 3))
  - 3

18

cons up while cdring down

- (define (append list1 list2) …)
  例 (append () (list 1 2 3))
  (1 2 3)
  例 (append () (list '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

19

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)

21

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)
      )))
```

23

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)) )))
```

24

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)

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) )))
        )))
```

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)) ))
  ))
```

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>))

Arguments with dotted-tail notation

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

iterative procedure

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

recursive procedure

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


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)
```

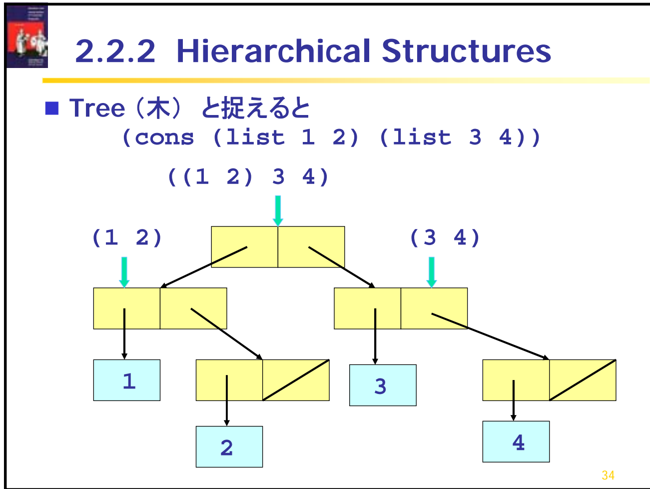
```
(define (scale-list items factor)
  (map (lambda (x) (* x factor)) items) ))
```

Apply transformation to each element

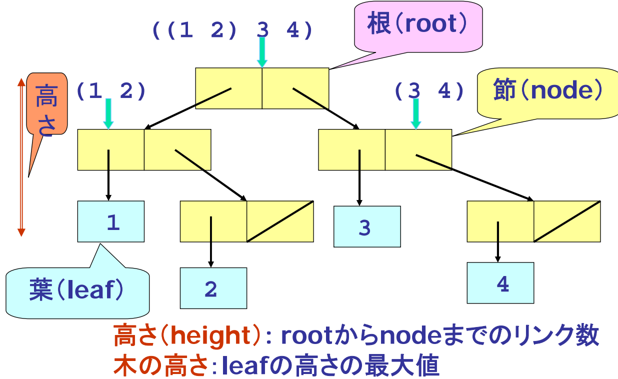
- 本当の map はもっと強力！

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

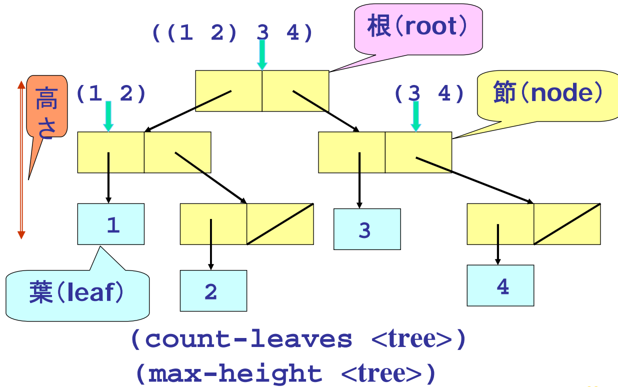
32



木の定義



木とその上での演算



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))
                )))))
```

40

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))
                )))))
```

41

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 ))
```

43

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 )))))
```


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)

11月21日・本日のメニュー

データによる抽象化

- 2.2. Hierarchical Data and the Closure Property
- 2.2.1 Representing Sequences
- 2.2.2 Hierarchical Structures
- **Intermission**
- 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.

47

11月21日・本日のメニュー

データによる抽象化

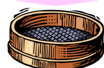
- 2.2. Hierarchical Data and the Closure Property
- 2.2.1 Representing Sequences
- 2.2.2 Hierarchical Structures
- Intermission
- **2.2.3 Sequences as Conventional Interfaces**

50

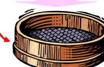
seq: 慣用インタフェース

- 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）

2 3 4 5 6 7 8 9 10 11



3 5 7 9 11



5 7 11

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)) ))
  ))
```

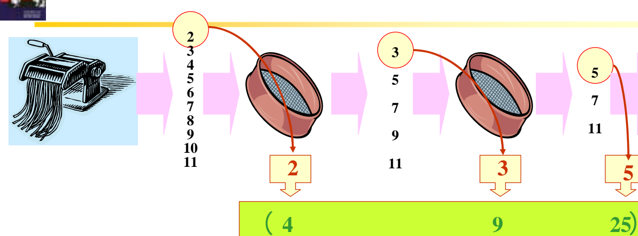

even-fibs 偶数のFibのリスト

```
(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)

祝
京都大學
１１月祭

- 宿題は、ありません.
- よく遊び，よく学べ.