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
工学部情報学科計算機科学コース
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/07/IntroAlgDs/
okuno@i.kyoto-u.ac.jp

TAの居室は10号館4階奥乃1研，2研
TAのページが運用中，質問箱もあります

11月20日・本日のメニュー

データによる抽象化

- **2.1.3 What Is Meant by Data?**
- 2.1.4 Extended Exercise: Interval Arithmetic
- 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

「具体から抽象へは行けるが、
抽象から具体へは行けない」
（畑村洋太郎『直観でわかる数学』岩波書店）

祝
京都大學チーム
ACM/ICPC 2007
アジア地区予選 優勝

日本チームの
アジア地区予選優勝は、
初快挙

2.1.3 データって何？ ペア(対、pair)再考

(make-rat n d) の満足すべき条件は

$$\frac{\text{(numer x)}}{\text{(denom x)}} = \frac{n}{d}$$

両者の長所・短所は？

1. cons, car, cdr を通常のセルで構築
2. 次の手続きで構築

```
(define (cons x y)
   (define (dispatch m)
      (cond ((= m 0) x)
            ((= m 1) y)
            (else (error "Argument not 0 or 1
                         -- CONS" m ))))
   dispatch )
(define (car z) (z 0))
(define (cdr z) (z 1))
```

(z 0) ⇒ car
(z 0) ⇒ cdr

ペア(対、pair)を手続きで実現

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0
                  or 1 -- CONS" m ))))
   dispatch)
(define (car z) (z 0))
(define (cdr z) (z 1))
```

- (define foo (cons 10 25))
- (car foo) ⇒
- (cdr foo)

5

もっとかっこよくペア(pair)を手続きで実現

```
(define (cons x y)
  (lambda (m) (m x y)) )
(define (car z)
  (z (lambda (p q) p)) )
(define (cdr z)
  (z (lambda (p q) q)) )
```

観測したらデータが得られる⇒量子コンピュータ風の計算

- (define foo (cons 10 25))
- (car foo) ⇒
  ((lambda (m) (m 10 25)) (lambda (p q) p))
  ⇒ ((lambda (p q) p) 10 25)
  ⇒ 10
- (cdr foo) ⇒
  ((lambda (m) (m 10 25)) (lambda (p q) q))
  ⇒ ((lambda (p q) q) 10 25)
  ⇒ 25

6

ペアの実装法を抽象化の壁から見ると

有理数を使ったプログラム

プログラム領域での有理数

`add-rat sub-rat mul-等`

分子と分母から構成される有理数

`make-rat numer denom`

ペアとして構成される有理数

`cons car cdr`

ペアの実装法

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

```
(define (cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0
                       or 1 -- CONS" m ))))
  dispatch)
(define (car z) (z 0))
(define (cdr z) (z 1))
```

11月20日・本日のメニュー

データによる抽象化

- 2.1.2 Abstraction Barriers
- **2.1.3 What Is Meant by Data?**
- 2.1.4 Extended Exercise: Interval Arithmetic

8

かっこよく自然数も手続きで実現

```
(define c0 (lambda (f) (lambda (x) x)))
(define (%succ c)
  (lambda (f) (lambda (x) (f ((c f) x))))
```

この自然数の表現を Church numerals （チャーチ数）という

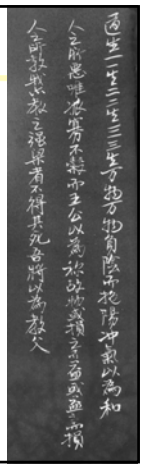
```
(define c1 (%succ c0))
   ⇒ (lambda (f) (lambda (x) (f ((c0 f) x))))
   ⇒ (lambda (f)
       (lambda (x)
         (f (((lambda (f) (lambda (x) x)) f) x)) ))
   ⇒ (lambda (f)
       (lambda (x)
         (f ((lambda (x) x) x)) ))
   ⇒ (lambda (f) (lambda (x) (f x)))

(define c1 (lambda (f) (lambda (x) (f x))))
(define c2 (lambda (f) (lambda (x) (f (f x)))))
(define c3 (lambda (f) (lambda (x) (f (f (f x))))))
```

自然数が0とf（後続関数）で定義

9

Church NumeralsとTAO



『老子』第42節の
「道 (TAO) から一が生まれ, 一から二が生まれ, 二から三が生まれ, 三から万物が生まれ, 云々」

Tao produced the one.
The one produced the two.
The two produced the three.
And the three produced the ten thousand things.
The ten thousand things carry the yin and embrace the yang, and through the blending of the material force they achieve harmony. *Tao-te Ching, 42, Lao Tzu.*

と符合するものです.
改良型Backus 記法が導入された Revised Report on the Algorithmic Language ALGOL 68 の113ページ にも上記の一節が引用されています.

INTREAL :: SIZETY integral ; SIZETY real.
SIZETY :: long LONGSETY ; short SHORTSETY ; EMPTY.
詳細は http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/04/IntroAlgDs/

Algol-68

Section80 van Wijngaarden, *et al.*

7.4. Well-formedness

{A mode is well formed if
(i) the elaboration of an **actual-declarer** specifying that mode is a finite action (i.e., any value of that mode can be stored in a finite memory) and
(ii) it is not strongly coercible from itself (since this would lead to ambiguities in coercion). }

7.4.1. Syntax

a) WHETHER (NOTION) **shields SAFE to SAFE**{73c} :
where (NOTION) is (PLAIN) or (NOTION) is (FLEXETY ROWS of) or (NOTION) is (union of) or (NOTION) is (void), WHETHER true.

b) WHETHER (PREF) **shields SAFE to yin SAFE**{73c} : WHETHER true.

c) WHETHER (**structured with**) **shields SAFE to yang SAFE**{73c} :
WHETHER true.

d) WHETHER (**procedure with**) **shields SAFE to yin yang SAFE**{73c} :
WHETHER true.

{As a by-product of mode equivalencing, modes are tested for well-formedness (7.3.1.c). All nonrecursive modes are well formed. For recursive modes, it is necessary that each cycle in each spelling of that mode (from '**MU definition of MODE**' to '**MU application**') passes through at least one '**HEAD**' which is yin, ensuring condition (i) and one (possibly the same) '**HEAD**' which is yang, ensuring condition (ii). Yin '**HEAD**'s are '**PREF**' and '**procedure with**'. Yang '**HEAD**'s are '**structured with**' and '**procedure with**'. The other '**HEAD**'s, including '**FLEXETY ROWS of**' and '**union of**', are neither yin nor yang. This means that the modes specified by **a**, **b** and **c** in

mode a = struct(int n, ref a next), b = struct(proc b next), c = proc(c)c

are all well formed. However, **mode d = [1 : 10] d, e = union(int, e)** is not a mode-declaration.}

{ *Tao produced the one.*
The one produced the two.
The two produced the three.
And the three produced the ten thousand things.
The ten thousand things carry the yin and embrace the yang, and through the blending of the material force they achieve harmony.
Tao-te Ching, 42, *Lao Tzu.*}

Operations on Church Numerals

```
(define c0 (lambda (f) (lambda (x) x)))
(define (%succ c)
  (lambda (f) (lambda (x) (f ((c f) x)))))
```

1～3の定義

```
(define c1 (lambda (f) (lambda (x) (f x))))
(define c2 (lambda (f) (lambda (x) (f (f x)))))
(define c3 (lambda (f) (lambda (x) (f (f (f x))))))
```

```
(define (%add n m)
  (lambda (f) (lambda (x) ((m f) ((n f) x)))))
(define (%multiply n m)
    (lambda (f) (lambda (x) ((n (m f)) x))) )
(define (%power n m)
    (lambda (f) (lambda (x) (((m n) f) x))) )
```

比較・減算は難しい、再帰呼び出しが必要。
再帰呼び出しのために無名手続きをYオペレータで使用。

Church Numerals の出力

- 実際の動きを見るために，入出力の関数を定義しましょう．

```
(define (c->n c)  ; 出力
    ((c (lambda (x) (+ 1 x))) 0) )
(define (n->c n)  ; 入力
   (if (> n 0)
       (%succ (n->c (- n 1)))
        c0 ))
```

- 上記の c->n はメモリを大量に消費し遅い。高速版は:

```
(define (c->n c) ((c 1+) 0))
```

- では実験．

```
1.(c->n (%add (n->c 5) (n->c 3)))
2.(c->n (%multiply (n->c 5) (n->c 3)))
3.(c->n (%power (n->c 5) (n->c 3)))
4.(c->n (%add (%power c2 c3)
              (%multiply c3 (n->c 4)) ))
```

13

減算・比較・Fixed Point Operator F

- 減算は難しい。まず、大小比較を定義する。
- 再帰呼び出しに無名手続きを使う必要がある。
- Y オペレータを使う。
  (Y F) = (F (Y F))

```
(define (Y F)
  (lambda (s)
    (F (lambda (x) (lambda (x) ((s s) x)))
       (lambda (s) (F (lambda (x) ((s s) x))))
    )))
```

詳細は Web ページにあります。
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/04/IntroAlgDs/

14

11月20日・本日のメニュー

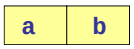
データによる抽象化

- 2.1.3 What Is Meant by 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

15

2.2 Hierachical Data and the Closure Property

- 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

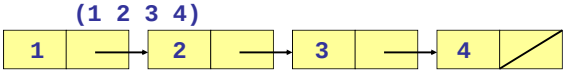
- <list> ::= <null> | (<element> . <element>)
- <element> ::= <name> | <number> | <list>

- Closure property (閉包性) of cons

16

2.2.1 Representing Sequences(表現)

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

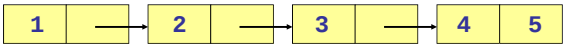


- (cons 1
    (cons 2
      (cons 3
        (cons 4 nil)
      )))
- (1 . (2 . (3 . (4 . nil))))
- (list 1 2 3 4)

17

Sequences表現の簡略化

- 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)

18

2.2.1 List operations（リスト演算）

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

19

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）

20

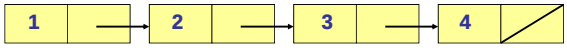
2.2.1 List operations（リスト演算）

- `(list-ref items n)`
  if `n=0`, `list-ref` is `car`
  otherwise, (n-1)st item of (cdr items)
-
```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1)) ))
```
-
```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items))) ))
```
- `cdr`ing down the list (cdr down)
- Tail recursion に注意

21

List-ref by cdring down

- (list-ref items n)
  if n=0, list-ref is car
  otherwise, (n-1)st item



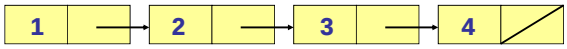

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1) )))
```

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

22

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！

23

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) )
```

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)

25

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) )
```

26

11月20日・本日のメニュー

データによる抽象化

- 2.1.3 What Is Meant by 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

27

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

28

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)

29

11月20日・本日のメニュー

データによる抽象化

- 2.1.3 What Is Meant by 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

30

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

31

2.2.1 リスト演算の例での quote

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

32

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)

33

cons up while cdring down

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

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

34

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)
      )))
```

35

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)) )))
```

36

cons up while cdring down

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

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

37

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)

38

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) )))
        )))
```

39

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)) ))
  ))
```

40

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>*`))`

41

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) )
```

42

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) ))
```

43

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！

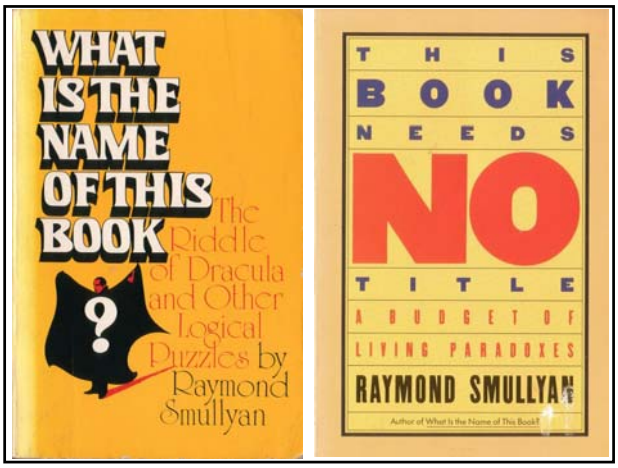
44

11月20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
- **Intermission**
- 2.2.2 Hierarchical Structures
- 2.2.3 Sequences as Conventional Interfaces

45



自己参照（Self-Reference）

1. What is the book of this book?
2. This book needs no title.
3. この文章は赤い色で書かれている.
4. この文章は17文字でできています.
5. 嘘つき村の住民は嘘つきです．嘘つき村の住民が「私は嘘はつきません」と言った.
6. 「クレタ人は嘘つきである」とクレタ人は言った.（新約聖書「テトスへの手紙」）

47

自己参照（Self-Reference）

嘘つき村の住民は皆嘘つきです．他の村の住民は嘘はつきません．嘘つき村の住民が「私は嘘はつきません」と言った.

- 発話が嘘（住民） ⇒
- 発話が本当 ⇒

ラッセルのパラドックス

- Aのすべての部分集合: 2^A

例: A={a, b, c}

2^A ={{}, {a},{b},{c},{a,b},{b,c},{c,a},{a,b,c}}

- すべての部分集合を含む集合Sを考える
- いずれが成立するか
  1. $S \in S$
  2. $S \notin S$

50

祝
京都大學
１１月祭

- 宿題は、ありません.
- よく遊び，よく学べ.