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
2.1　高階手続きによる抽象化

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/08/IntroAlgDs/

okuno@i.kyoto-u.ac.jp

TAの居室は10号館4階奥乃１研，２研

TAのページがオープン，質問箱もあります

http://winnie.kuis.kyoto-u.ac.jp/~fukubaya/AlgDsWiki/

11月18日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data

2.1 Introduction to Data Abstraction

2.1.1 Example: Arithmetic Operations for Rational Numbers

2.1.2 Abstraction Barriers

2.1.3 What Is Meant by Data?

2.1.4 Interval Arithmetic

「具体から抽象へは行けるが、抽象から具体へは行けない」

We can go from Concrete to Abstract, while we cannot from Abstract to Concrete.

Prof. Yotaro Hatamura

（畑村洋太郎『直観でわかる数学』岩波書店）

Let's Play JMC with your num.

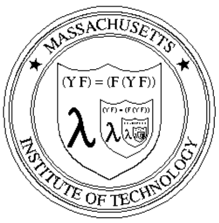
```
(define (jmc n)
  (if (> n 100)
      (- n 10)
     (jmc (jmc (+ n 11)))
  ))
```

- 各自、次の式を求めよ

(jmc (modulo 学籍番号 30))

5

補足： Fixed Point




```
(define (jmc n)
   (if (> n 100)
       (- n 10)
       (jmc (jmc (+ n 11))) ))

(fixed-point jmc 1) ⇒ ?
(Y F) = (F (Y F))   Y operator
                   (不動点となる手続きを作成)
(Y jmc) = (F (Y jmc))
        = (lambda (n)
            (if (> n 100) (- n 10) 91) )
```

6

第2章 データによる抽象の構築

- 第1章は**手続き抽象化**
  - 基本手続き
  - 合成手続き・手続き抽象化
  - 例： Σ, Π, accumulate, filtered-accumulate
- 第2章は**データ抽象化**
  - 基本データ構造(primitive data structure/object)
  - 合成データオブジェクト (compound data object)
- データ抽象化で手続きの意味(**semantics**)を拡張
  - 加算(＋)で**どのようなデータ構造も扱える**
  - 基本手続き： 整数＋整数、有理数＋有理数、実数＋実数
  - 合成手続き： 複素数＋複素数、行列＋行列

8

第2章　データ抽象化で学ぶこと

- **抽象化の壁**(abstraction barrier)の構築
  - データ構造の実装を外部から隠蔽(blackbox)
- **閉包**(closure)
  - 組み合わせを繰り返してもよい
- **従来型インタフェース**(conventional interface)
  - Sequence を手続き間インタフェースとして使用
  - ベルトコンベア、生産ライン、UNIXのパイプ
- **記号式**(symbolic expression)表現
- **汎用演算**(genetic operations)
- **データ主導プログラミング**(data-directed programming)

9

11月11日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data

2.1 Introduction to Data Abstraction

2.1.1 Example: Arithmetic Operations for Rational Numbers
2.1.2 Abstraction Barriers
2.1.3 What Is Meant by Data?
2.1.4 Interval Arithmetic

2.1 データ抽象化(data abstraction)

抽象データの4つの基本操作

1. **構成子(constructor)**
2. **選択子(selector)**
3. **述語(predicate)**
4. **入出力(input/ output)**

11

2.1.0 Integers(整数)

- 構成子(constructor)
  <n> ; <n> integer
- 選択子(selector)
  <n> ; <n> integer
- 述語(predicate)
  (integerp? <x>)
  (= <x> <y>)
- 入出力(input/output)
  <n> ; <n> integer

12

11月11日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data

2.1 Introduction to Data Abstraction

2.1.1 Example: Arithmetic Operations for Rational Numbers

2.1.2 Abstraction Barriers

2.1.3 What Is Meant by Data?

2.1.4 Interval Arithmetic

2.1.1 Rational Numbers(有理数)

- 構成子(constructor)
  (make-rat <n> <d>)
  <n> numenator(分子),
  <d> denominator (分母)
- 選択子(selector)
  (numer <x>)
  (denom <x>)
  <x> rational number
- 述語(predicate)
  (rational? <x>)
  (equal-rat? <x> <y>)
- 入出力(input/output)
  <n>/<d>

14

2.1.1 Rational Numbers（有理数）

- 加算（addition） $\frac{n_1}{d_1}+\frac{n_2}{d_2}=\frac{n_1d_2+n_2d_1}{d_1d_2}$
- 減算（subtraction） $\frac{n_1}{d_1}-\frac{n_2}{d_2}=\frac{n_1d_2-n_2d_1}{d_1d_2}$
- 乗算（multiplication） $\frac{n_1}{d_1}\times\frac{n_2}{d_2}=\frac{n_1n_2}{d_1d_2}$
- 除算（division） $\frac{n_1}{d_1}\div\frac{n_2}{d_2}=\frac{n_1d_2}{d_1n_2}$
- 述語 $n_1d_2=n_2d_1 \Rightarrow \frac{n_1}{d_1}=\frac{n_2}{d_2}$

15

Rational Number Operations

$\frac{n_1}{d_1}+\frac{n_2}{d_2}=\frac{n_1d_2+n_2d_1}{d_1d_2}$ $\frac{n_1}{d_1}-\frac{n_2}{d_2}=\frac{n_1d_2-n_2d_1}{d_1d_2}$

```
(define (add-rat x y)
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)) )
            (* (denom x) (denom y)) ))

(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)) )
            (* (denom x) (denom y)) ))
```

17

Rational Number Operations

$\frac{n_1}{d_1}\times\frac{n_2}{d_2}=\frac{n_1n_2}{d_1d_2}$ $\frac{n_1}{d_1}\div\frac{n_2}{d_2}=\frac{n_1d_2}{d_1n_2}$ $n_1d_2=n_2d_1 \Rightarrow \frac{n_1}{d_1}=\frac{n_2}{d_2}$

```
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y) )))

(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y) )))

(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x)) ))
```

19

Rational Number Representation

```
(define (make-rat n d) (cons n d))
```

n	d

ペア（pair）で表現

```
(define (numer x) (car x))

(define (denom x) (cdr x))

(define (print-rat x)
   (newline)
   (display (numer x))
   (display “/”)
   (display (denom x))
   x )
```

20

Rational Number Reduction（既約化）

```
(define (make-rat n d) (cons n d))
```

この表現は曖昧： e.g., 2/3, 4/6, 6/9

```
(define (make-rat n d)
   (let ((g (gcd n d)))
      (cons (/ n g) (/ d g)) ))
```

既約化： *reducing rational numbers to the lowest terms*

21

いつの時点で簡略化するべきか？

```
(define (make-rat n d)
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g)) ))
```

両者の長所・短所は？

```
(define (make-rat n d) (cond n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g) ))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g) ))
```

この違いは他のプログラムに影響を与えるか？ 22

11月11日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data
2.1 Introduction to Data Abstraction
2.1.1 Example: Arithmetic Operations for Rational Numbers
2.1.2 Abstraction Barriers
2.1.3 What Is Meant by Data?
2.1.4 Interval Arithmetic

既約化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`

```
(define (make-rat n d)
   (let ((g (gcd n d)))
      (cons (/ n g) (/ d g)) ))
```

ペアとして構成される有理数

```
(define (make-rat n d)
    (cond n d))
(define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g) ))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g) ))
```

`cons car cdr`

ペアの実装法

11月11日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data
2.1 Introduction to Data Abstraction
2.1.1 Example: Arithmetic Operations for Rational Numbers
2.1.2 Abstraction Barriers
2.1.3 What Is Meant by Data?
2.1.4 Interval Arithmetic

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

```
(numer x)        n
---------   =   ---
(denom x)        d
```

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)

28

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

29

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11日・本日のメニュ

データによる抽象化

2 Building Abstractions with Data

2.1 Introduction to Data Abstraction

2.1.1 Example: Arithmetic Operations for Rational Numbers

2.1.2 Abstraction Barriers

2.1.3 What Is Meant by Data?

2.1.4 Interval Arithmetic

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)))
```

自然数が0とf（後続関数）で定義

- (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))))))

33

Church NumeralsとTAO

『老子』第42節の

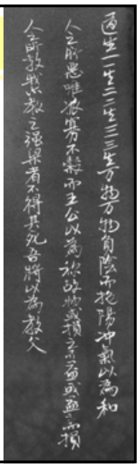
「道 (TAO) から一が生まれ, 一から二が生まれ, 二から三が生まれ, 三から万物が生まれ, 云々」

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

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.*}

113

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)) ))
```

38

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/

39

11月11日・本日のメニュー

データによる抽象化

2 Building Abstractions with Data
2.1 Introduction to Data Abstraction
2.1.1 Example: Arithmetic Operations for Rational Numbers
2.1.2 Abstraction Barriers
2.1.3 What Is Meant by Data?
2.1.4 Interval Arithmetic

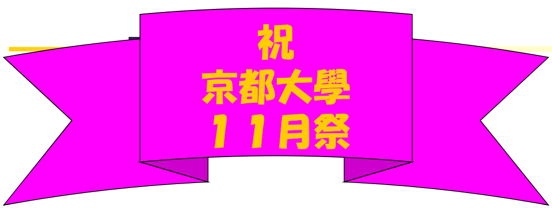
2.1.4 Interval Arithmetic

- Constructor
```
(define (make-interval a b) (cons a b))
```
- Selectors 等
```
 (define (upper-bound x)
 (define (lower-bound x)
 (define (equal-interval? x y)
 (define (sub-interval x y)
```
- Interval arithmetic は単位系変換で重要。
  - 1*in*≒2.54*cm*, 1*ft*≒30.48*cm*, 1*yd*≒0.914*m*, 1*mile*≒1.609*km*, 1*nautical mile*≒1.852*km*, 1*acre*≒4047m^2, 1 *UKgal*≒4.54*l*, 1*USgal*≒3.79*l*, 1*bbl*≒159*l*, **1*πsec*≒1 *nano-century* (10^{-7} *year*)** , 1 *light year*≒9.461×10^{12}km(**9.461Tkm**)
  - 1*oz*≒28.3*g*, 1*lb*≒0.454*Kg*, 1*ct*≒0.2*g*

41

2.1.4 Interval Arithmetic

```
(define (add-interval x y)
   (make-interval
     (+ (lower-bound x) (lower-bound y))
     (+ (upper-bound x) (upper-bound y)) ))
(define (mul-interval x y)
  (let ((p1 (* (lower-bound x) (lower-bound y)))
        (p2 (* (lower-bound x) (upper-bound y)))
        (p3 (* (upper-bound x) (lower-bound y)))
        (p4 (* (upper-bound x) (upper-bound y))))
     (make-interval (min p1 p2 p3 p4)
                    (max p1 p2 p3 p4))))
(define (div-interval x y)
   (mul-interval x
      (make-interval (/ 1.0 (upper-bound y))
                     (/ 1.0 (lower-bound y)) )))
```



宿題は、ありません.

よく遊び，よく学べ.