アルゴリズムとデータ構造入門
1.3 高階手続きによる抽象化
2.データによる抽象の構築
2.1データ抽象化入門

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/13/IntroAlgDs/
okuno@i.kyoto-u.ac.jp, okuno@nue.org

TAの居室は総合研究7号館4階418号室奥乃研
池宮 由楽(M1) 奥乃研・音楽情報処理G
坂東 宣昭(M1) 奥乃研・ロボット聴覚G
古川 孝太郎(M1) 奥乃研・ロボット聴覚G

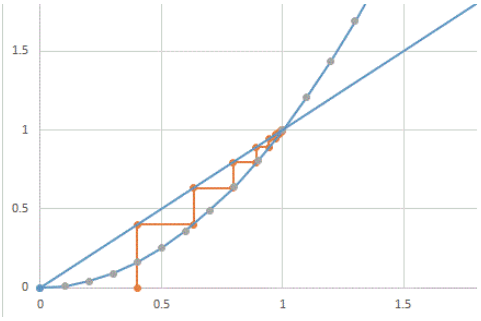
優秀レポート提出者

第4回
- 枡井啓貴 (4)
- 山田淳二 (4)
- 横山悠
- 遠藤ルッカス良
- 奥野僚介 (3)
- 延田和磨
- 伊藤博典
- 加田昇
- 高山勉 (2)

第5回
- 山田淳二 (5)
- 枡井啓貴 (5)
- 横山悠 (2)
- 三鼓悠
- 大家理和 (3)
- 紫藤佑介
- 柴田健太郎(2)
- 嶋隼輝
- 磯西市路 (3)

随意課題提出者

豊島 悠紀夫



11月12日・本日のメニュー

1.3 Formulating Abstractions with Higher-Order Procedures
1.3.1 Procedures as Arguments
1.3.2 Constructing Procedures Using 'Lambda'
1.3.3 Procedures as General Methods
1.3.4 Procedures as Returned Values
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

NoStudent Left Behind

1.3.1 Procedures as Arguments

NoStudent Left Behind

```
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b)) ))
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b)) ))
(define (cube x) (* x x x))
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b)) ))
```

$$\sum_{i=a}^{b} i$$

$$\sum_{i=a}^{b} i^3$$

$$\sum_{i=a,i+4}^{b} \frac{1}{i(i+2)}$$

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

手続きの共通パターンを抽象化
⇒ 高階手続き

汎用総和手続きを共通パターンを基に定義する

NoStudent Left Behind

```
(define (<name> a b)
  (if (> a b)
      0
      (+ (<term> a)
         (<name> (<next> a) b)) ))
(define (sum f a next b)
  (if (> a b)
      0
      (+ (f a)
         (sum f (next a) next b)) ))
(define (inc n) (+ n 1))
(define (cube x) (* x x x))
(define (sum-cubes a b)
  (sum cube a inc b) )
(define (identity x) x)
(define (sum-integers a b)
  (sum identity a inc b) )
```

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

$$\sum_{i=a,i+1}^{b} cube(i)$$

$$\sum_{i=a,i+1}^{b} i$$

Ex. 1.32 総和 (sum), 積 (product)を抽象化

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

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

```
(define (product term a next b)
  (if (> a b)
      1
      (* (term a)
         (product term (next a) next b)) )
```

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

```
(define (<combiner> <name> <term> a <next> b)
  (if (> a b)
      <null-value>
      (<combiner> (<term> a)
         (<name> <term> (<next> a) <next> b))
 ))
```

NoStudent Left Behind

accumulate ⇦ sum, product

NoStudent Left Behind

```
(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))
```

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

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

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

11月12日・本日のメニュー

1.3 Formulating Abstractions with Higher-Order Procedures
1.3.1 Procedures as Arguments
1.3.2 Constructing Procedures Using 'Lambda'
1.3.3 Procedures as General Methods
1.3.4 Procedures as Returned Values
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

NoStudent Left Behind

lambda: 無名(匿名)手続き

NoStudent Left Behind

```
(define (plus4 x) (+ x 4))
```

は次式と等価

```
(define plus4 (lambda (x) (+ x 4)))
```

ラムダ式の読み方

```
(lambda      (x)     (+   x     4))
```

the procedure of an argument x that adds x and 4

仮引数 (formal parameters)　本体 (body)

ラムダ式の適用

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

x = 1, y = 2, z = 3 を代入(置換)

```
(+ 1 2 (square 3))
(+ 1 2 9)
12
```

10

Lambda as anonymous procedure

NoStudent Left Behind

```
(lambda (x) (+ x 4))
```

無名(匿名)手続き

```
((lambda (x) (+ x 4)) 5)
```

手続き適用

```
(define (pi-sum a b)
  (define (pi-term x)
    (/ 1.0 (* x (+ x 2))) )
  (define (pi-next x)(+ x 4) )
  (sum pi-term a pi-next b) )
```

局所的な無駄な名前 pi-term, pi-next をなくす

```
(define (pi-sum a b)
  (sum (lambda (x) (/ 1.0 (* x (+ x 2))))
       a
       (lambda (x) (+ x 4))
       b ))
```

11

lambda: Anonymous procedure

NoStudent Left Behind

```
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1))) ))
```

は次の式と等価

```
(define fact
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact (- n 1))) )))
```

12

Using let to create local variables

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

$$a = 1+xy$$
$$b = 1-y$$
$$f(x,y) = xa^2 + yb + ab$$

補助変数 a, b を使いたい

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b) ))
  (f-helper
    (+ 1 (* x y))
    (- 1 y) ))
```

13

1.3.2 Local Variables with let

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b) ))
  (f-helper
    (+ 1 (* x y))
    (- 1 y) ))

(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)) )
    (+ (* x (square a))
       (* y b)
       (* a b) )))
```

```
(define (f x y)
  ((lambda (a b)
     (+ (* x (square a))
        (* y b)
        (* a b) ))
   (+ 1 (* x y))
   (- 1 y) ))
```

```
(let ((<v1> <e1>)
      (<v2> <e2>)
      ...
      (<vn> <en>) )
  <body> )
```

シンタックス・シュガー

14

scope of variables(有効範囲)

```
(let ((x 7))
  (+ (let ((x 3))
       (+ x (* x 10)) )
     x) )
```

Substition model

33

40

```
((lambda (x)
   (+ ((lambda (x)
        (+ x (* x 10)) )
       3 )
      x ))
 7 )
```

λ式に展開して考える

15

scope of variables(有効範囲)

NoStudent Left Behind

```
(let ((x 5))
  (let ((x 3)
        (y (+ x 2)) )
    (* x y) ))
```

Substition model

x=3, y=7

21

λ式に展開して考える

```
((lambda (x)
   ((lambda (x y)
      (* x y) )
    3 (+ x 2) ))
 5 )
```

16

let* は、変数を順番に評価

NoStudent Left Behind

```
(let ((x 5))
  (let* ((x 3)
         (y (+ x 2)) )
    (* x y) ))
```

Substition model

x=3, y=5

15

λ式に展開して考える

```
((lambda (x)
   ((lambda (x)
      ((lambda (y)
         (* x y) )
       (+ x 2) )
    3 )
 5 )
```

17

let は、変数を同時に評価

NoStudent Left Behind

```
(let ((x 5))
  (let ((x 3)
        (y (+ x 2)) )
    (* x y) ))
```

Substition model

x=3, y=7

21

λ式に展開して考える

```
((lambda (x)
   ((lambda (x y)
      (* x y) )
    3
    (+ x 2) )
 5 )
```

18

11月12日・本日のメニュー

1.3 Formulating Abstractions with Higher-Order Procedures
1.3.1 Procedures as Arguments
1.3.2 Constructing Procedures Using 'Lambda'
1.3.3 Procedures as General Methods
1.3.4 Procedures as Returned Values
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

NoStudent Left Behind

1.3.3 Procedures as General Methods

NoStudent Left Behind

Finding roots of equations by the half-interval method（区間二分法）

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negative? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
```

20

Finding roots of equations by the half-interval method

NoStudent Left Behind

2点の値の符号が異なるかのチェックを行う

```
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))

(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Values are not of opposite sign" a b)))
    ))
```

L:開始時の区間長、T:誤差許容度、
ステップ数: $\Theta(\log(L/T))$

21

Finding fixed points of functions(不動点)

抽象化すると

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

$f(x), f(f(x)), f(f(f(x))), \cdots$

xが不動点 $x = f(x)$

Finding fixed points of functions(不動点)

```
(fixed-point cos 1.0)
```

$y = \cos y$

```
(fixed-point
  (lambda (y) (+ (sin y) (cos y)))
  1.0 )
```

$y = \sin y + \cos y$

$y * y = x$より $y = \frac{x}{y}$ と書くと,

sqrtは次の関数の不動点探索となる

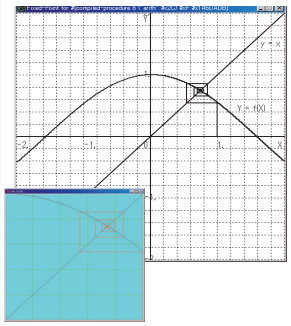
$y \mapsto \frac{x}{y}$

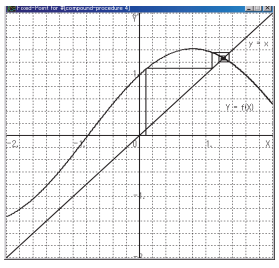
```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0 ))
```


Finding fixed points of functions(不動点)

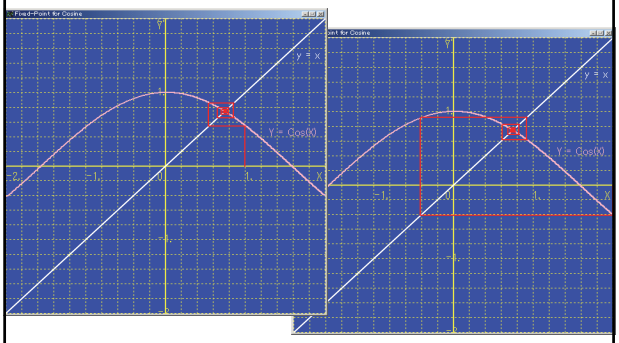
```
(fixed-point cos 1.0)
```

```
(fixed-point
  (lambda (y)
    (+ (sin y) (cos y)))
  0.1 )
```





(fixed-point cos 1.0) と (fixed-point cos 2.0)



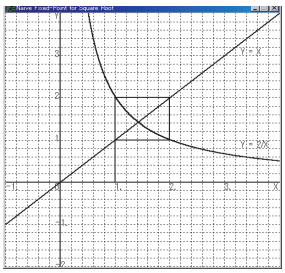
NoStudent Left Behind

不動点が求まらない場合がある

$\sqrt{x}$

```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

$$y \mapsto \frac{x}{y}$$



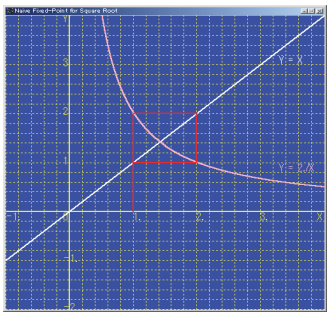
(sqrt 2)を実行すると

1 → 2 → 1

(y_1 → y_2 → y_1)

NoStudent Left Behind

Naïve Fixed Point (sqrt 2)



$$y \mapsto \frac{x}{y}$$

急いては事を仕損じる

アイデア倒れ

NoStudent Left Behind

Average damping（平均緩和法）

One way to control such ocillations:
Redefine a new function

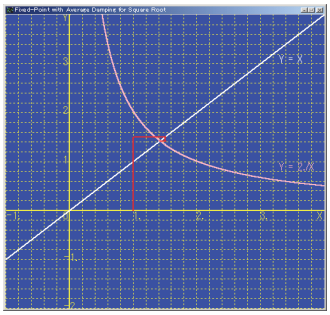
$$y \mapsto \frac{1}{2}\left(y + \frac{x}{y}\right)$$

```
(define (sqrt x)
  (fixed-point
      (lambda (y) (average y (/ x y)))
      1.0) )
```

Average damping（平均緩和法）

NoStudent Left Behind

Fixed Point with Average Damping



$$y \mapsto \frac{1}{2}\left(y + \frac{x}{y}\right)$$

Average damping 平均緩和法

NoStudent Left Behind

11月12日・本日のメニュー

1.3 Formulating Abstractions with Higher-Order Procedures
1.3.1 Procedures as Arguments
1.3.2 Constructing Procedures Using 'Lambda'
1.3.3 Procedures as General Methods
1.3.4 Procedures as Returned Values
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

NoStudent Left Behind

平均緩和法を不動点手続きから見直す

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```

$y \mapsto \frac{x}{y}$

平均緩和法を不動点手続きの観点から眺めると

$y \mapsto \frac{1}{2}\left(y + \frac{x}{y}\right)$

```
(define (average-damp f)
  (lambda (x) (average x (f x))))
((average-damp square) 10)
```

$\frac{1}{2}(x + x^2)$

average-damp で統一的に捉えることが可能

```
(define (sqrt x)
  (fixed-point
    (average-damp (lambda (y) (/ x y)))
    1.0))
(define (cube-root x)
  (fixed-point
    (average-damp (lambda (y) (/ x (square y))))
    1.0))
```

$y \mapsto \frac{x}{y^2}$ $y \mapsto \frac{1}{2}\left(y + \frac{x}{y^2}\right)$

31

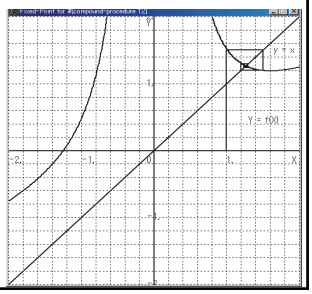
NoStudent Left Behind

Cubic-root の実行過程

NoStudent Left Behind

```
(define (cube-root x)
  (fixed-point
    (average-damp (lambda (y) (/ x (square y))))
    1.0 ))
```

$$y \mapsto \frac{1}{2}\left(y + \frac{x}{y^2}\right)$$



ニュートン法を不動点手続きから見直す

```
(define (deriv g)
    (lambda (x)(/ (- (g (+ x dx)) (g x)) dx)) )
(define dx 0.00001)
(define (cube x) (* x x x))
((deriv cube) 5)
```

ニュートン法

$$y = x - \frac{g(x)}{g'(x)}$$

```
(define (newton-transform g)
  (lambda (x)(- x (/ (g x) ((deriv g) x)))) )
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess) )
(define (sqrt x)
  (newtons-method (lambda (y) (- (square y) x))
                  1.0))
```

NoStudent Left Behind

更なる抽象化・first-class procedures

```
(define (fixed-point-of-transform g transform
   guess)
  (fixed-point (transform g) guess) )
```

1st method: 平均緩和法

```
(define (sqrt x)
  (fixed-point-of-transform
     (lambda (y) (/ x y))
     average-damp
     1.0 ))
```

2nd method:ニュートン法

```
(define (sqrt x)
  (fixed-point-of-transform
     (lambda (y) (- (square y) x))
     newton-transform
     1.0 ))
```

手続きの構築で何ら差別がない

First-class citizen （第1級市民）

第1級市民の“権利と特権”

- 変数で名前をつけることができる.
- 手続きへ引数として渡すことができる.
- 手続きを結果として返すことができる.
- データ構造の中に含めることができる.

Microsoft Longhorn will make RAW 'first class citizen.'
The Inquirer, Wed. Jun-8, 2005

手続き(関数)への演算： 導関数

```
(define dx 0.0001)
(define (ddx f x)
    (/ (- (f (+ x dx)) (f x)) dx) )
```

数値微分

```
(ddx square 3) ⇒ 6.00010000001205
```

我々はもっとスマート！　導関数という考え方を採用

```
(define (deriv f)
   (lambda (x)
      (/ (- (f (+ x dx)) (f x)) dx) ))
```

導関数

```
((deriv square) 3) ⇒ 6.00010000001205
((deriv (deriv square)) 3)⇒ 1.99999998
(define (new-ddx f x)
    ((deriv f) x) )
```

2次導関数もこの通り

手続き(関数)の合成: 高階導関数

この考え方を発展させ、高階導関数が構築できる

```
(define (compose f g)
  (lambda (x)
    (f (g x)) ))
```

```
(define 2nd-deriv (compose deriv deriv))
```

((2nd-deriv square) 3) ⇒ 1.9999999878

もちろん手続きの合成も

((compose square sqrt) 7) ⇒ 7.0

((2nd-deriv cos) pi) ⇒ 0.999999993922529

```
(define 3rd-deriv (compose deriv 2nd-deriv))
```

((3rd-deriv sin) pi) ⇒ 0.999999960615838

((4th-deriv cos) pi) ⇒ 1.11022302462516

37

Let's Play JMC with your number.

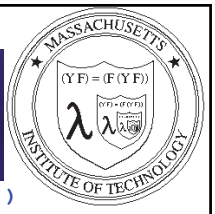
```
(define (jmc n)
  (if (> n 100)
      (- n 10)
      (jmc (jmc (+ n 11)))
  ))
```

各自、次の式を求めよ

(jmc (modulo 学籍番号 100))

38

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) ?) )
```

NoStudent Left Behind

訂正: Fixed Point Operator Y

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

(Y F) = (F (Y F))

再帰呼び出しに無名手続きを使いたい

```
(define (fact f)
  (lambda (n) (if (= n 0) 1 (* n (f (- n 1))))) )
```

((Y fact) 0) -> 0!
((Y fact) 10) -> 10!

(Y fact) は階乗のプログラムと等価

詳しくは，Church numeralの項で説明.

41

Rules of Thumb

1. Rule of 72 : $r \times y = 72$

   For y years with an interest rate of r % per year ⇒ roughly double.

2. *π seconds is a nanocentury.*

3. $1\ year = 3.155 \times 10^7\ seconds$

42

11月13日・本日のメニュー

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
2.2. Hierarchical Data and the Closure Property
2.2.1 Representing Sequences（並び）
2.3.1 Quote

43

Lyrical Lisp 登場

- 魔法言語 リリカル☆Lisp
- 『マンガで分かるLisp』好評連載中

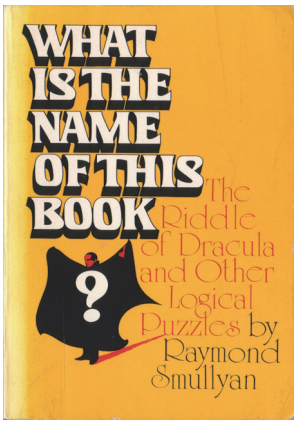
http://lyrical.bugyo.tk/

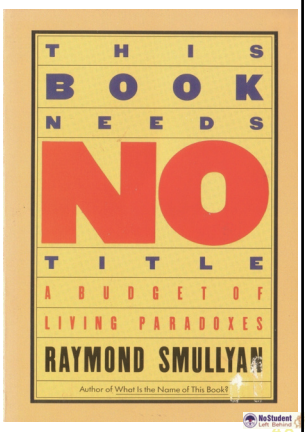
44

Interesting books

1. Douglas R. Hofstadter
2. Reymond Sumullyan
3. Doug Adams
4. Don E. Knuth
5. John H. Conway and Richard K. Guy

47





40

自己参照(Self-Reference)の例

NoStudent Left Behind

1. What is the name 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」)

49

自己参照(Self-Reference)

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

- 発話が嘘(住民)⇒
- 発話が本当⇒

51

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
  - $S \in S$
  - $S \notin S$

52

11月12日・本日のメニュー

1.3 Formulating Abstractions with Higher-Order Procedures
1.3.1 Procedures as Arguments
1.3.2 Constructing Procedures Using 'Lambda'
1.3.3 Procedures as General Methods
1.3.4 Procedures as Returned Values
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

NoStudent Left Behind

第2章 データによる抽象の構築

NoStudent Left Behind

- **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

56

「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)

57

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 (generic operations)
- データ主導プログラミング (data-directed programming)

58

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)

59

2.1.0　Integers(整数)

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

60

宿題：11月18日24時　締切

1. 教科書 **2-1-3 ~ 2-2-2** を読み，①想定質問，②想定質問の解答，③その説明を記述．1の課題の後ろに書くこと．
2. 練習問題2.17~2.28 をプログラムを書いて解きなさい．
3. 説明と出力結果について，レポートをlatexで作成し，pdf で提出すること．
4. プログラムファイルとレポート（pdf）を **SICP-6@zeus.kuis.kyoto-u.ac.jp** に送付

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

87