アルゴリズムとデータ構造入門
1.手続きによる抽象の構築
1.3 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/06/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

1

Probabilistic Algorithms（確率的アルゴリズム）

- Carmichael numbers: 561, 1105, 1072, 2465,
  $a^{560} = (a^2\ a^{10}\ a^{16})^{20}$
  $a^2 \equiv 1 \bmod 3$, $a^{10} \equiv 1 \bmod 11$, $a^{16} \equiv 1 \bmod 17$
  $\Rightarrow\ a^{560} \equiv 1 \bmod 561 = 3 * 11 * 17$
- Fermat's testは、エラーの機会を任意に小さくできる。→ *probabilistic algorithm*
  必要条件のみ満足
- *Algorithm*: Wilson's test
  p is a prime precisely when $(p-1)! \equiv -1 \bmod p$
  必要十分条件

2

Discussion: Fermat's or Wilson's?

1. 単純な素数判定：
2. Fermat's test： *p* が素数なら
   $\forall a < p,\ a^{(p-1)} \equiv 1 \bmod p$
   if $(a, p)=1$
   $\forall a < p,\ a^p \equiv a \bmod p$
3. Wilson's test: *p* が素数である必要十分条件は
   $(p-1)! \equiv -1 \bmod p$

3

Fermat's Last Theorem

$$x^n + y^n = z^n$$

I have discovered a truly remarkable proof which this margin is too small to contain.

n>2 で *x, y, z* を満たす非負整

Euler's Conjecture

$$a^4 + b^4 + c^4 \neq d^4$$

が成立するだろう。

$$95800^4 + 217519^4 + 414560^4 = 422481^4$$

1987年発見

4

11月7日・本日のメニュー

高階手続きによる抽象化

- 1.3.1 Procedures as Arguments
- 1.3.2 Constructing Procedures Usin `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

5

Ex.1.32 Accumulation

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

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

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

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

Ex.1.32 Accumulation

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

12

Ex.1.32 Accumulation

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

13

11月7日・本日のメニュー

高階手続きによる抽象化

- 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

14

lambda: Anonymous procedure

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

15

Lambda as anonymous procedure

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

(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) )

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

Using let to create local variables

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

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

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

補助変数 a, bを使いたい

17

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 (($\langle v_1 \rangle$ $\langle e_1 \rangle$)
($\langle v_2 \rangle$ $\langle e_2 \rangle$)
…
($\langle v_n \rangle$ $\langle e_n \rangle$))
$\langle body \rangle$)

シンタックス・シュガー

18

scope of variables

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

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

Substition model

19

scope of variables（λを展開）

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

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

x = 7
x = 3
-> 33
x = 7 -> 40

20

scope of variables

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

x = 7
x = 3
-> 33
x = 7 -> 40

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

x = 5
x = 3
y = 7
-> 21

21

11月7日・本日のメニュー

高階手続きによる抽象化

- 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

22

1.3.3 Procedures as General Methods

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))))))
```

23

Finding roots of equations by the half-interval method

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

2点の値の符号が異なるかのチェックを行う

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

24

Finding fixed points of functions（不動点）

```
(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))
```

抽象化すると

xが不動点 $x = f(x)$ $f(x), f(f(x)), f(f(f(x))),$

25

Finding fixed points of functions（不動点）

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

$y = \cos y$

$y = \sin y + \cos y$

$y * y = x$ より $y = \frac{x}{y}$ と書くと、

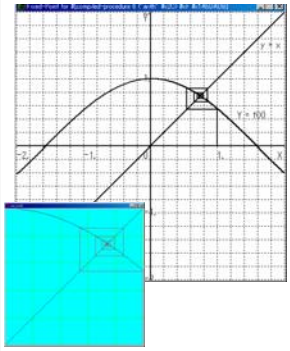
次の関数の不動点探索となる $y \mapsto \frac{x}{y}$

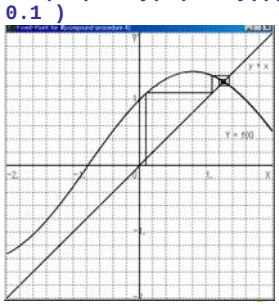
```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

26

Finding fixed points of functions（不動点）

```
(fixed-point cos 1.0) (fixed-point
                        (lambda (y)
                          (+ (sin y) (cos y)))
                        0.1 )
```

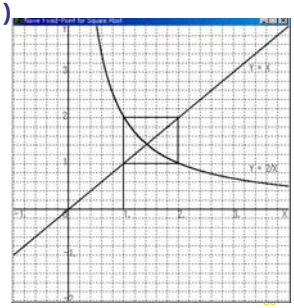




不動点が求まらない場合がある $\sqrt{x}$

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

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

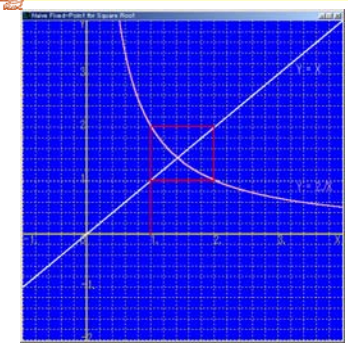


(sqrt 2) を実行すると

1 → 2 → 1

(y_1 → y_2 → y_1)

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

Average damping（平均緩和法）

One way to control such ocillations:

Redefine a new function

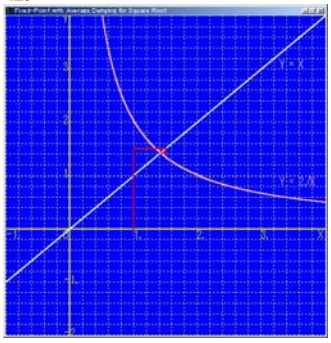
$$y \mapsto \frac{1}{2}\left(y + \frac{x}{y}\right)$$

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

Average damping（平均緩和法）

32

Fixed Point with Average Damping



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

Average damping 平均緩和法

33

11月7日・本日のメニュー

高階手続きによる抽象化

- 1.3.2 Constructing Procedures Using `Lambda'
- 1.3.3 Procedures as General Methods
- **Intermission**
- 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

34

プログラミングの進め方

1. プログラムファイルを作成
   - emacs･meadow･mule を使う
   - メモ帳を使う(改行コードが悪さをする)
2. tus2, tustk2, Edwin(MIT)でプログラムファイルを読み込む.
   (load “c:/my-tus/file.lsp”)
3. プログラムを動かす

35

emacs / meadow / mule

1. すべての入力は評価される
   - 単純な文字 ⇒ 自分が返されるself-evaluating)
2. コマンドは連想型
   - f(orward), b(ackward), e(nd), a(初め)
   - p(revious), n(ext)
   - d(delete), k(ill)
   - control-key(C-): 文字単位のコマンド
   - meta-key(M-) : 単語単位のコマンド(モードに依存)
   - control-meta-key(C-M-):S式単位のコマンド
3. Incremental search(C-s):逐次探索
4. C-x は拡張コマンド C-xC-s (save), C-xC-f (find)
5. ファイルの属性(ext)によりモード自動設定
6. タブで自動字下げ, 閉じ括弧で対応する開き括弧が点滅
7. M-x apropos で関連情報を検索するのがよい.

36

11月7日･本日のメニュー

高階手続きによる抽象化

- 1.3.2 Constructing Procedures Using `Lambda'
- 1.3.3 Procedures as General Methods
- **Intermission**
- 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

37

11月7日・本日のメニュー

- 1.3.2 Constructing Procedures Usin `Lambda'
- 1.3.3 Procedures as General Metho
- 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

38

1.3.4 Procedures as Returned Values

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

平均緩和法を不動点の観点から眺めると

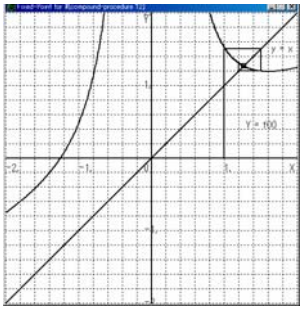
```
(define (average-damp f)
  (lambda (x) (average x (f x))))

((average-damp square) 10)

(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))
```

average-damp で統一的に捉えることが可能

39

Cubic-root の実行過程

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



40

Newton's method & differentiation

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

(define (cube x) (* x x x))
((deriv cube) 5)

(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))
```

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

ニュートン法

41

更なる抽象化・ first-class procedures

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

1st method

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

2nd method

```
(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

42

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.
- 手続きの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

43

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))
- (ddx square 3) ⇒ 6.00010000001205
- 我々はもっとスマートだった！導関数という考え方を採用
- (define (deriv f)
  (lambda (x)
  (/ (- (f (+ x dx)) (f x)) dx)))
- ((deriv square) 3) ⇒ 6.00010000001205
- ((deriv (deriv square)) 3)⇒ 1.99999998
- (define (new-ddx f x)
  ((deriv f) x))

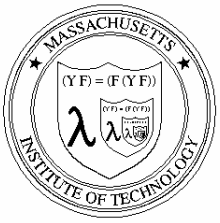
44

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

45

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) ?) )
```

46

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 100))
```

47

Fixed Point Operator F

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

再帰呼び出しに無名手続きを使いたい

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

詳しくは、Church numeralの項で説明。

48

宿題:11月13日午後5時締切

- 不動点
- 宿題は、次の4題:
- Ex.1.40, 1.41, 1.42, 1.43.

49