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
2.データによる抽象の構築
2.1データ抽象化入門

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/10/IntroAlgDs/
okuno@i.kyoto-u.ac.jp, okuno@nue.org
阿曽 慎平(M1) 奥乃研・音楽G (10号館4階奥乃1研)
平澤 恭治(M1) 奥乃研・ロボット聴覚G (10号館4階奥乃1研)

11月9日・本日のメニュー

- 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

12月21日 中間試験を実施

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))))))
```


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: $\Theta(\log(L/T))$

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))
```

xが不動点 $x = f(x)$

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

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，

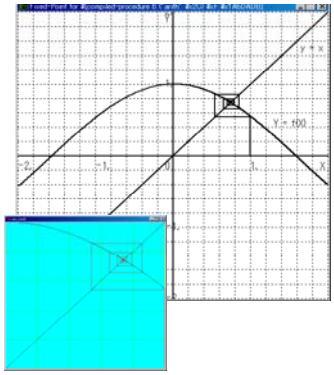
次の関数の不動点探索となる

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

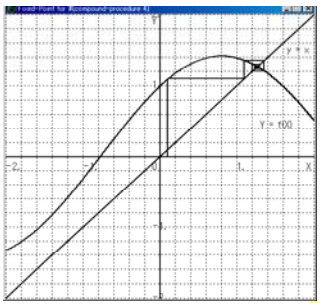
```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

Finding fixed points of functions(不動点)

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

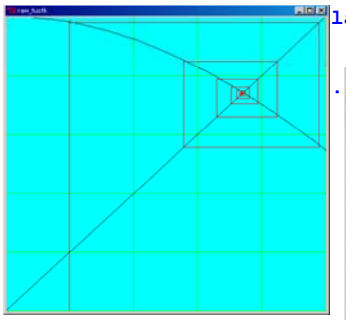


```
(fixed-point
  (lambda (y)
    (+ (sin y) (cos y)))
 0.1 )
```

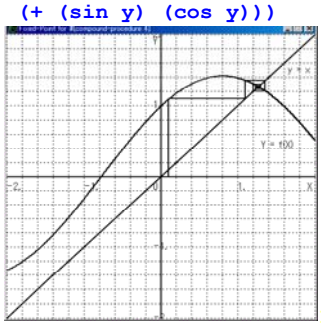


Finding fixed points of functions(不動点)

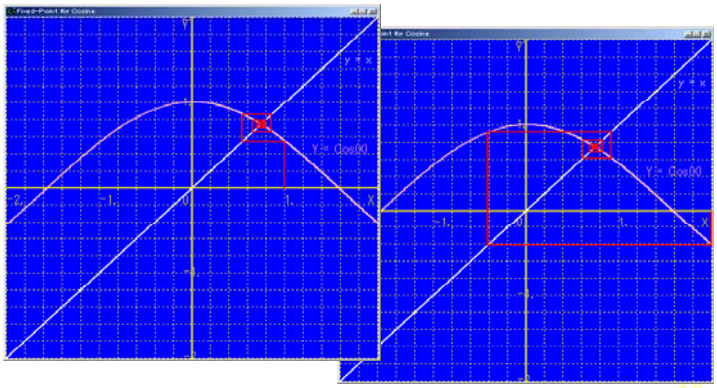
```
(fixed-point cos 0.2)
```



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



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



不動点が求まらない場合がある

$\sqrt{x}$

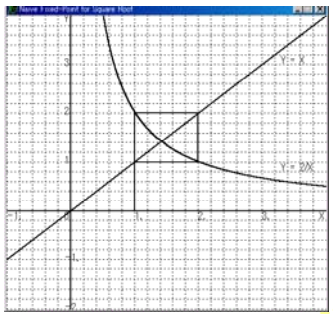
```
(define (sqrt x)
  (fixed-point (lambda (y) (/ x y))
               1.0))
```

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

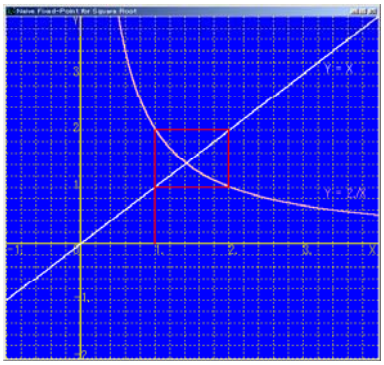
(sqrt 2)を実行すると

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）

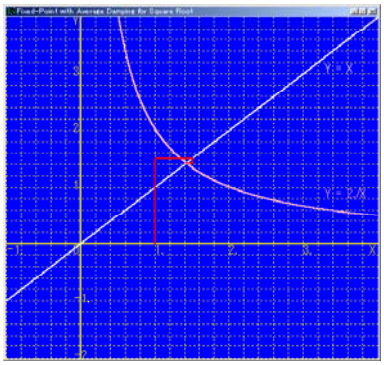
One way to control such ocillations:
Redefine a new function

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

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

Average damping（平均緩和法）

Fixed Point with Average Damping



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

Average damping 平均緩和法

11月9日・本日のメニュー

- 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

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)
```

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

```
(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))
```

$y \mapsto \frac{x}{y^2}$

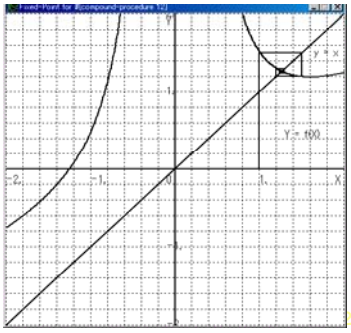
$y \mapsto \frac{1}{2}\left(y + \frac{x}{y^2}\right)$

average-damp で統一的に捉えることが可能

Cubic-root の実行過程

```
(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))
```

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)Z
       (/ (- (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

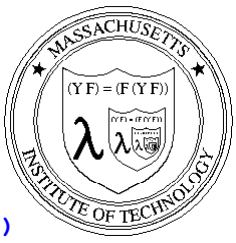
Let's Play JMC with your number.

```
(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))

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) ?) )
```


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。

「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）

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

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）

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）

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

2.1.1 Rational Numbers（有理数）

- 構成子（constructor）
  `(make-rat <n> <d>)`
  `<n>` numerator（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>`

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 $n_1d_2=n_2d_1 \Rightarrow \frac{n_1}{d_1}=\frac{n_2}{d_2}$

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

(define (sub-rat x y)
  (make-rat
```


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

(define (div-rat x y)
  (make-rat

(define (equal-rat? x y)
```


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 )
```


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*

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) (cons 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？

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

宿題:11月16日正午締切

1. 教科書練習問題 Ex.1.35, 1.41, 1.42, 1.43.
2. 実行例を添付すること
3. Program 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，その人の名前を明記すること．Webは出展を明記．(otherwise『同じ』回答は減点)

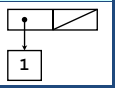
DON'T PANIC!

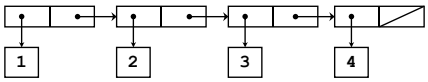


62

2.2 対(pairs)とシーケンス(sequences)

- 対(pair)
- シーケンス(並び，sequence)
- (cons 1 nil)
- (list 1)と同じ
- nilは空リスト
  - (list 1 2 3 4)
  - (cons 1
      (cons 2
        (cons 3
          (cons 4 nil))))

	(cons 1 2)	(cons 1 nil)
ドット記法(dotted notation)	(1 . 2)	(1 . nil) または (1)
箱ポインタ記法(box-and-pointer notation)		

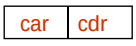


(1 2 3 4) あるいは
(1 . (2 . (3 . (4 . nil))))

66

リスト処理演算(list processing)

- (null *<expression>*) *<expression>* が nill か？
- (eq? *<e1>* *<e2>*) *<e1>*と*<e2>* が 同じオブジェクトか？
- (cons *<e1>* *<e2>*) *<e1>*と*<e2>* からpairを作成
- (car *<list>*) *<list>* の car を抽出
- (cdr *<list>*) *<list>* の cdr を抽出



(car (cons 1 2))	⇒ 1
(car (list 1 2 3 4))	⇒ 1
(cdr (cons 1 2))	⇒ 2
(cdr (cons 1 nil))	⇒ nil
(cdr (list 1))	⇒ nil
(cdr (list 1 2 3 4)))	⇒ (2 3 4)
(car (cdr (1 2 3 4)))	⇒ 2
(car nil)	⇒ error
(cdr nil)	⇒ error

67

リスト処理の練習問題

- (quote *<expression>*) *<expression>* を返す.
- ' *<expression>* *<expression>* を返す.
- (cons '1 '(2 3)) ⇒ (1 2 3)
- (cons '(1) '(2 3)) ⇒ ((1) 2 3)
- equal? を定義せよ.
  - (equal? 1 1) ⇒ #t
  - (equal? 3 '(1 2)) ⇒ #f
  - (equal? '(1 2) '(1 3)) ⇒ #f
  - (equal? 1 2) ⇒ #f
  - (equal? '(1 2) '(1 2)) ⇒ #t
  - (equal? '(1 2) '(1 3)) ⇒ #f
- member? を定義せよ.
  - (member 1 '(1 2)) ⇒ (1 2)
  - (member '(1 2) nil) ⇒ #f
  - (member '(1 2) '(1 2 (1 2) 3)) ⇒ ((1 2) 3)
  - (member 3 '(1 2)) ⇒ #f
- append を定義せよ.
  - (append '(1 2) '(3 4)) ⇒ (1 2 3 4)
  - (append '(1 2) nil) ⇒ (1 2)
  - (append 1 '(1 2)) ⇒ error
- reverse を定義せよ.
  - (reverse (list 1 2 3 4)) ⇒ (4 3 2 1)
  - (reverse '(1 2 3)) ⇒ (3 2 1)
  - (reverse nil) ⇒ nil

68

リスト処理の練習問題(自習用)

- but-last を定義せよ.
  - (but-last '(1 2 3 4 5)) ⇒ (1 2 3 4)
- assoc を定義せよ.
  - (assoc hgo '((f IP) (hgo IntroAlgDS) (yan ProLang)) ⇒ (hgo IntroAlgDS)
  - (assoc hgo '((ishi Gairon) (iso math) (tom HW)) ⇒ #f
- length を定義せよ.
  - (length '(1 2)) ⇒ 2
  - (length nil) ⇒ 0
  - (length '(1 2 3 5)) ⇒ 4
- copy を定義せよ.
  - (copy '(1 2)) ⇒ (1 2)
  - (copy '((1 . 2) . (3 . 4))) ⇒ ((1 . 2) 3 . 4)
- flatten を定義せよ.
  - (flatten '((1)(2 (3) 4) 5) ⇒ (1 2 3 4 5)
  - (flatten '((1 2 3))) ⇒ (1 2 3)
  - (flatten ' (1 2 3)) ⇒ (1 2 3)
  - (flatten nil) ⇒ nil

69