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
2.4 抽象データの多重表現

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

1月9日・本日のメニュー

データによる抽象化

- 2.4 Multiple Representations for Abstract Data
- 2.4.1 Representations for Complex Numbers
- 2.4.2 Tagged data
- 2.4.3 Data-Directed Programming and Additivity
- 2.5 Genetic Operation System
  - Coercion
  - Hierarchy of types
- 整列 (sorting)

4

複素数システムのデータ抽象化の壁

複素数を使ったプログラム

汎用演算

プログラム領域での複素数

add-complex,sub-complex,mul等

複素数演算パッケージ

情報隠蔽

real-part imag-part
magnitude angle

直交座標表現
(Rectangular representation)

極座標表現
(Polar representation)

cons car cdr

リスト構造と基本マシン算術

7

複素数の演算

1. 虚数(imaginary part)

$z = x + iy \qquad i^2 = -1$

2. 加算(addition)

$\mathrm{Real-part}(z_1 + z_2) = \mathrm{Real-part}(z_1) + \mathrm{Real-part}(z_2)$

$\mathrm{Imaginary-part}(z_1 + z_2) = \mathrm{Imaginary-part}(z_1) + \mathrm{Imaginary-part}(z_2)$

3. 乗算(multiplication)

$\mathrm{Re}(z_1 \cdot z_2) = \mathrm{Re}(z_1) \cdot \mathrm{Re}(z_2) - \mathrm{Im}(z_1) \cdot \mathrm{Im}(z_2)$

$\mathrm{Im}(z_1 \cdot z_2) = \mathrm{Re}(z_1) \cdot \mathrm{Im}(z_2) + \mathrm{Im}(z_1) \cdot \mathrm{Re}(z_2)$

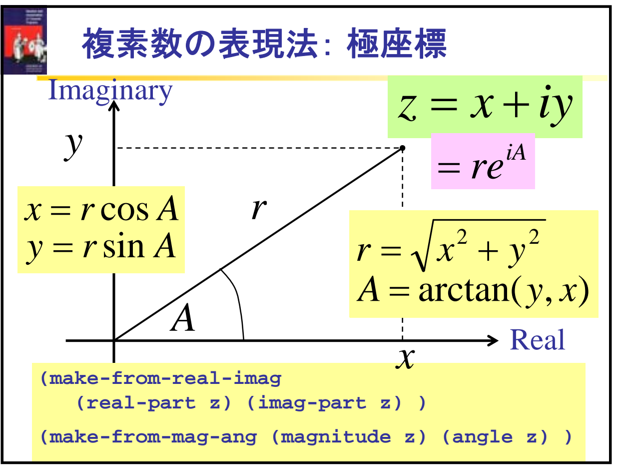
$\mathrm{Magnitude}(z_1 \cdot z_2) = \mathrm{Magnitude}(z_1) \cdot \mathrm{Magnitude}(z_2)$

$\mathrm{Angle}(z_1 \cdot z_2) = \mathrm{Angle}(z_1) + \mathrm{Angle}(z_2)$

複素数の四則演算 $z = x + iy = re^{iA}$

```
(define (add-complex z1 z2)
  (make-from-real-imag
    (+ (real-part z1) (real-part z2))
    (+ (imag-part z1) (imag-part z2)) ))
(define (sub-complex z1 z2)
  (make-from-real-imag
    (- (real-part z1) (real-part z2))
    (- (imag-part z1) (imag-part z2)) ))
(define (mul-complex z1 z2)
  (make-from-mag-ang
    (* (magnitude z1) (magnitude z2))
    (+ (angle z1) (angle z2)) ))
(define (div-complex z1 z2)
  (make-from-mag-ang
    (/ (magnitude z1) (magnitude z2))
    (- (angle z1) (angle z2)) ))
```

10



複素数の表現法 $z = x + iy = re^{iA}$

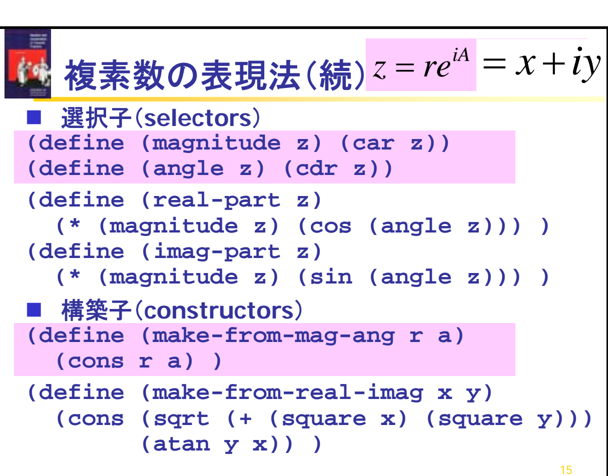
■ 選択子(selectors)

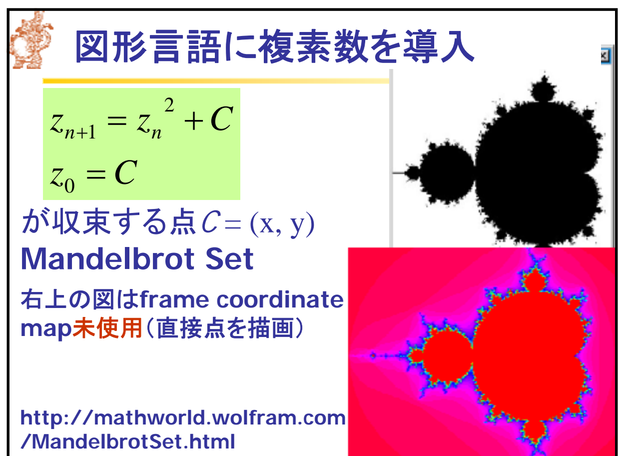
```
(define (real-part z) (car z))
(define (imag-part z) (cdr z))
(define (magnitude z)
  (sqrt (+ (square (real-part z)
            (square (imag-part z)) )))
(define (angle z)
  (atan (imag-part z) (real-part z)))
```

■ 構築子(constructors)

```
(define (make-from-real-imag x y)
  (cons x y) )
(define (make-from-mag-ang r a)
  (cons (* r (cos a)) (* r (sin a))) )
```

14





2.4.2 Tagged data（タグ付きデータ）

- データ抽象化の1つの観点
- *Principle of least commitment*
（最小責任の原則）
- 選択子と構築子を使用した抽象化の壁を使って、データオブジェクトの具体的な表現をできるだけ遅くし、システム設計における柔軟性を最大限にする。
- 本節ではさらに principle of least commitment を発展させる。

1. 表現法（選択子と構築子）の設計後でも、表現法の抽象化（曖昧性）を維持。
2. 直交座標と極座標が共用できる仕組みを考える。

タグ付きデータの実装法

- 手続きは `type-tag`（型タグ）で処理を区別する。
- `type-tag` はデータに付与されている。

```
(define (attach-tag type-tag contents)
  (cons type-tag contents))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      (error "Bad tagged datum -
              TYPE-TAG" datum)))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      (error "Bad tagged datum -
              CONTENTS" datum)))
```

22

座標のタグ付きデータの表現法

```
(define (rectangular? z)
   (eq? (type-tag z) 'rectangular))

(define (polar? z)
   (eq? (type-tag z) 'polar))
```

23

直交座標のタグ付きデータの表現法

```
(define (real-part-rectangular z) (car z))
(define (imag-part-rectangular z) (cdr z))
(define (magnitude-rectangular z)
  (sqrt (+ (square (real-part-rectangular z)
          (square (imag-part-rectangular z))
        )))
(define (angle-rectangular z)
  (atan (imag-part-rectangular z)
        (real-part-rectangular z) ))
(define (make-from-real-imag-rectangular x y)
  (attach-tag 'rectangular (cons x y)) )
(define (make-from-mag-ang-rectangular r a)
  (attach-tag 'rectangular
     (cons (* r (cos a)) (* r (sin a))) ))
```

25

極座標のタグ付きデータの表現法

```
(define (real-part-polar z)
  (* (magnitude-polar z)
     (cos (angle-polar z)) ))
(define (imag-part-polar z)
  (* (magnitude-polar z)
     (sin (angle-polar z)) ))
(define (magnitude-polar z) (car z))
(define (angle-polar z) (cdr z))
(define (make-from-real-imag-polar x y)
  (attach-tag 'polar
    (cons (sqrt (+ (square x) (square y)))
          (atan y x) )))
(define (make-from-mag-ang-polar r a)
  (attach-tag 'polar (cons r a)) )
```

27

タグ付きデータへの手続き

dispatching on type

```
(define (real-part z)
  (cond ((rectangular? z)
         (real-part-rectangular (contents z)))
        ((polar? z)
         (real-part-polar (contents z)))
        (else (error "Unknown type -
                     REAL-PART" z))))
(define (imag-part z)
  (cond ((rectangular? z)
         (imag-part-rectangular (contents z)))
        ((polar? z)
         (imag-part-polar (contents z)))
        (else (error "Unknown type -
                     IMAG-PART" z))))
```

28

タグ付きデータへの手続き（続）

dispatching on type

```
(define (magnitude z)
  (cond ((rectangular? z)
         (magnitude-rectangular (contents z)))
        ((polar? z)
         (magnitude-polar (contents z)))
        (else (error "Unknown type -
                      MAGNITUDE" z))))
(define (angle z)
  (cond ((rectangular? z)
         (angle-rectangular (contents z)))
        ((polar? z)
         (angle-polar (contents z)))
        (else (error "Unknown type -
                      ANGLE" z))))
```

29

タグ付きデータへの手続き（続々）

- 複素数の演算は不変
- 複素数用汎用演算（generic operation）使用の為

```
(define (add-complex z1 z2)
  (make-from-real-imag
     (+ (real-part z1) (real-part z2))
     (+ (imag-part z1) (imag-part z2)) ))

(define (mul-complex z1 z2)
  (make-from-mag-ang
     (* (magnitude z1) (magnitude z2))
     (+ (angle z1) (angle z2))))
```

- 抽象化の壁に注意
- Principle of least commitmentによる遅延

30

全システムの設計は

- 目的に合致した複素数表現を選ぶ
- 直交座標表現
  実数部と虚数部が分かっているとき
- 極座標表現
  半径と角度が分かっているとき

- 最終的に得られた複素数演算システムの構造は次のスライド
- type-tag の使用がポイント
- *dispatching on type* という技法

31

2.4.3 Data-Directed Programming and Additivity

- 型タグ(type-tag)の問題点
- 汎用手続き(real-part, imag-part, maginitude, angle)は、異なる表現をすべて知っておく必要がある。
- 例えば、複素数の新表現を作成したら

1. (new-rep? z) を定義
2. 各手続きに new-rep? に関係する処理を追加

```
 (define (real-part z)
    (cond ((rectangular? z) … )
          ((polar? z) … )
          ((new-rep? z) … )
          (else … )))
```

- **加法的(additivity)ではない。**

34

Data-Directed Programming (データ駆動型プログラミング)

- 加法的(additivity)なインタフェースとするために、表のようなデータを使用。

型 (type)

演算 operations	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

35

表の操作

TUT Scheme

- 表に演算名・型(type)でその処理法を`put`で付加
- 表から演算名・型(type)でその処理法を`get`で検索
- (put *<op>* *<type>* *<item>*)
  (putprop *<op>* *<type>* *<item>*)
  表に*<op>* *<type>*で索引をつけて*<item>*を登録
- (get *<op>* *<type>*) (getprop *<op>* *<type>*)
  表から*<op>* *<type>*の索引で検索し、あれば、*<item>*を抽出
- 演算に関連する情報*<item>*は、以下では、**手続き(ラムダ式)**
- *<type>*は、**引数の型のリスト**

36

put と get の動き

```
(define put putprop)
(define get getprop)
```

```
(put 'banana 'price 300)
(put 'banana 'color ' yellow)
(get 'banana 'price)
(put 'Kyoto 'Ja "kyouto")
(put 'University 'Ja "daigaku")
(get 'Kyoto 'Ja)
          -> "kyouto"
(map (lambda (x)  (get x 'Ja) )
     '(Kyoto University)  )
          -> ("kyouto" "daigaku")
(put 'University 'Ge "Universitate")
```

38

簡単な情報検索 by put & get

```
(define (lookup given-key set-of-records)
   (let ((result (get set-of-records given-key))
     (if (null? result) false result) ))
                                 総人口   男性人口  女性人口
(put 'population 'China '(1285.0 660.5 624.5))
(put 'population 'India '(1025.1 528.5 496.6))
(put 'population 'USA '(285.9 141.0 144.9))
(put 'population 'Indonesia '(214.8 107.8
107.1))
(put 'population 'Brazil '(172.6 85.2 87.4))
(put 'population 'Pakistan '(145.0 74.5 70.5))
(put 'population 'Russia '(144.7 67.7 77.0))
(put 'population 'Bangradesh '(140.4 72.3 68.0))
(put 'population 'Japan '(127.1 62.2 65.0))
(put 'population 'Nigeria '(116.9 59.0 58.0))
(put 'population 'Mexico '(100.4 49.6 50.7))
(lookup 'Japan 'population)
```

39

表の操作で使うデータ

- 演算に関連する情報*<item>*は、以下では、手続き(ラムダ式)
- *<type>*は、引数の型のリスト

```
(define (total-amount x n)
   (* n (get x 'price)) )
(put 'banana '(obj int) total-amount)
(put 'banana 'price 300)
((get 'banana '(obj int)) 'banana 10)
```

40

直交座標のタグ付きデータの表現法

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y)
    (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
             (square (imag-part z)))))
  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))
続く;; interface to the rest of the system
```

42

直交座標のタグ付きデータの表現法

```
(define (install-rectangular-package)
  ;; interface to the rest of the system
 (define (tag x)
   (attach-tag 'rectangular x) )
 (put 'real-part '(rectangular) real-part)
 (put 'imag-part '(rectangular) imag-part)
 (put 'magnitude '(rectangular) magnitude)
 (put 'angle '(rectangular) angle)
 (put 'make-from-real-imag 'rectangular
      (lambda (x y)
        (tag (make-from-real-imag x y))))
 (put 'make-from-mag-ang 'rectangular
      (lambda (r a)
        (tag (make-from-mag-ang r a))))
 'done)
```

43

直交座標のタグ付きデータの表現法

```
(define (install-rectangular-package)
  ;; internal procedures
  (define (real-part z) (car z))
  (define (imag-part z) (cdr z))
  (define (make-from-real-imag x y) (cons x y))
  (define (magnitude z)
    (sqrt (+ (square (real-part z))
             (square (imag-part z)))))
  (define (angle z)
    (atan (imag-part z) (real-part z)))
  (define (make-from-mag-ang r a)
    (cons (* r (cos a)) (* r (sin a))))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'rectangular x))
  (put 'real-part '(rectangular) real-part)
  (put 'imag-part '(rectangular) imag-part)
  (put 'magnitude '(rectangular) magnitude)
  (put 'angle '(rectangular) angle)
  (put 'make-from-real-imag 'rectangular
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'rectangular
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

44

極座標のタグ付きデータの表現法

```
(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a)
    (cons r a))
  (define (real-part z)
    (* (magnitude z) (cos (angle z))))
  (define (imag-part z)
    (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
続く;;  interface to the rest of the system
```

極座標のタグ付きデータの表現法

```
(define (install-rectangular-package)
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y)
         (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a)
          (tag (make-from-mag-ang r a)) ))
  'done)
```

極座標のタグ付きデータの表現法

```
(define (install-polar-package)
  ;; internal procedures
  (define (magnitude z) (car z))
  (define (angle z) (cdr z))
  (define (make-from-mag-ang r a) (cons r a))
  (define (real-part z)
    (* (magnitude z) (cos (angle z))))
  (define (imag-part z)
    (* (magnitude z) (sin (angle z))))
  (define (make-from-real-imag x y)
    (cons (sqrt (+ (square x) (square y)))
          (atan y x)))
  ;; interface to the rest of the system
  (define (tag x) (attach-tag 'polar x))
  (put 'real-part '(polar) real-part)
  (put 'imag-part '(polar) imag-part)
  (put 'magnitude '(polar) magnitude)
  (put 'angle '(polar) angle)
  (put 'make-from-real-imag 'polar
       (lambda (x y) (tag (make-from-real-imag x y))))
  (put 'make-from-mag-ang 'polar
       (lambda (r a) (tag (make-from-mag-ang r a))))
  'done)
```

generic operation の適用方法

```
(define (apply-generic op . args)
  (let ((type-tags (map type-tag args)))
    (let ((proc (get op type-tags)))
      (if proc
          (apply proc (map contents args))
          (error
            "No method for these types -
             APPLY-GENERIC"
            (list op type-tags))))))
  'done)
```

これでgeneric procedure を再定義する。

```
(define (real-part z)
     (apply-generic 'real-part z))
```

generic operation の適用方法

```
(define (real-part z)
   (apply-generic 'real-part z))

(define (imag-part z)
   (apply-generic 'imag-part z))

(define (magnitude z)
   (apply-generic 'magnitude z))

(define (angle z)
   (apply-generic 'angle z))
```

49

目的に合致した複素数表現を選ぶ

- 直交座標表現 if 実数部と虚数部が分かっているとき
- 極座標表現 if 半径と角度が分かっているとき

```
(define (make-from-real-imag x y)
  ((get 'make-from-real-imag 'rectangular)
   x y))

(define (make-from-mag-ang r a)
  ((get 'make-from-mag-ang 'polar) r a))
```

- ほうら、うまく目的に合致した複素数表現が選ばれて、その後、型に対応した手続きが自動的に選択されることがお分かりになったでしょう。

50

Symbolic differentiation

```
(define (deriv exp var)
  (cond ((number? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0) )
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var) ))
        ((product? exp)
         (make-sum
           (make-product (multiplier exp)
                (deriv (multiplicand exp) var) )
           (make-product
                (deriv (multiplier exp) var)
                (multiplicand exp) )))
        <more rules can be added here>
        (else (error "unknown expression type -
                      DERIV" exp ))))
```

51

Symbolic differentiation

```
(define (deriv exp var)
   (cond ((number? exp) 0)
         ((variable? exp)
          (if (same-variable? exp var)
              1
              0 ))
         (else
          ((get 'deriv (operator exp))
           (operands exp)
           var ))))

(define (operator exp) (car exp))
(define (operands exp) (cdr exp))
```

52

Data-Directed Programmingのポイント

- 表を行方向に分割：type-tagでdispatch

型（type）

演算 operations

	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

53

put と get の動き

- 演算に関連する情報<*item*>は、以下では、手続き(ラムダ式)

```
(define (total-amount x n)
   (* n (get x 'price)) )
(put 'banana '(obj int) total-amount)
(put 'banana 'price 300)

((get 'banana '(obj int)) 'banana 10)
```

- このプログラミングはさえない。
- 改善するのが message passing

54

Message Passing のポイント

- 表を行方向に分割: type-tagでdispatch
- 表を列方向に分割: データオブジェクトがdispatch

演算 operations / 型 (type)

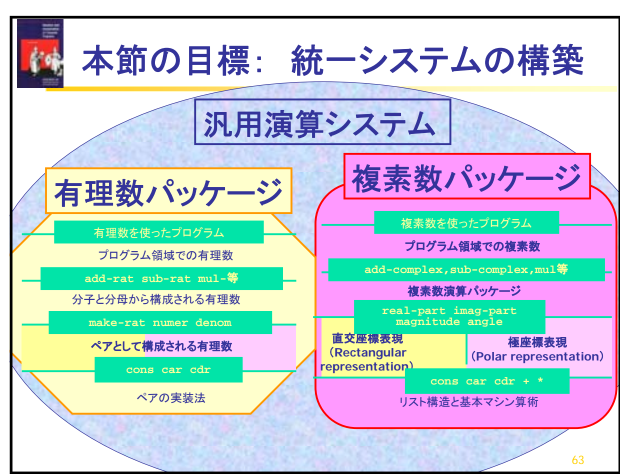
	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

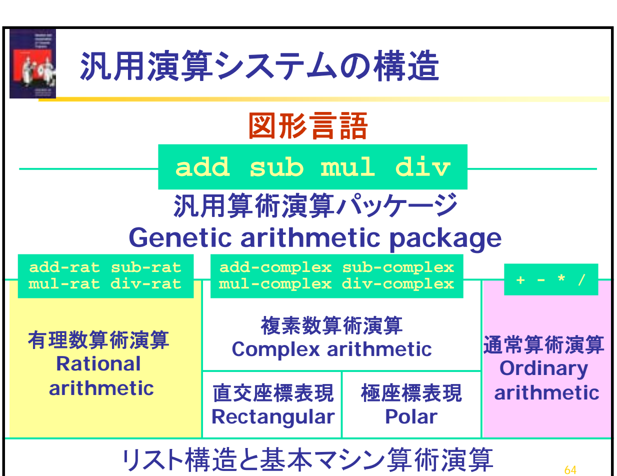
Message passing

Church numeralと同じ発想

```
(define (make-from-real-imag x y)
  (define (dispatch op)
    (cond ((eq? op 'real-part) x)
          ((eq? op 'imag-part) y)
          ((eq? op 'magnitude)
           (sqrt (+ (square x)
                    (square y) )))
          ((eq? op 'angle) (atan y x))
          (else
           (error "Unknown op –
            MAKE-FROM-REAL-IMAG" op))))
  dispatch)

(define (apply-generic op arg) (arg op))
```

56





2.5.1 汎用算術演算手続き

- add sub mul div だけで算術演算を記述する。
- 引数のタイプにより適切な演算を行う手続きを適用

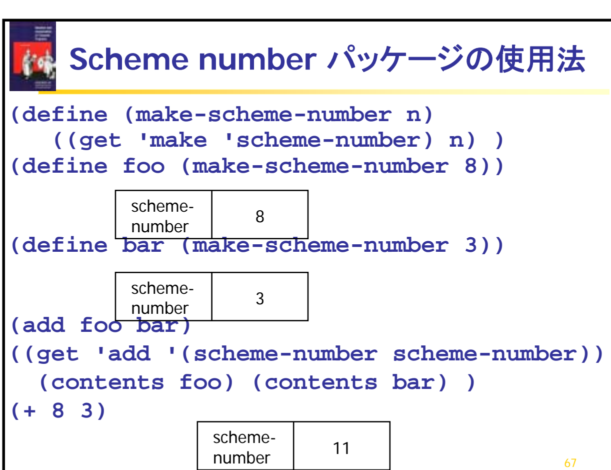
```
(define (add x y)
  (apply-generic 'add x y) )
(define (sub x y)
  (apply-generic 'sub x y) )
(define (mul x y)
  (apply-generic 'mul x y) )
(define (div x y)
  (apply-generic 'div x y) )
```

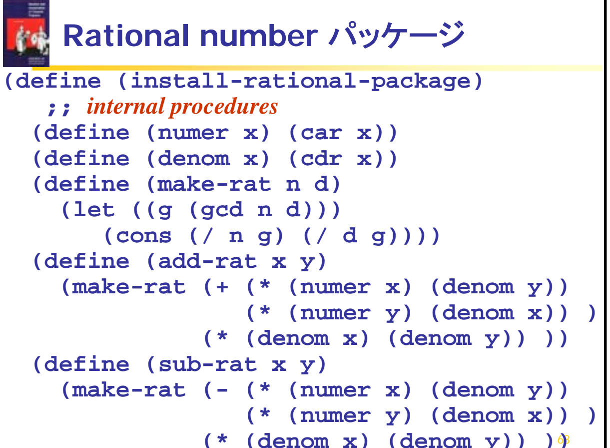
65

Ordinary number パッケージ

```scheme
(define (install-scheme-number-package)
  (define (tag x)
      (attach-tag 'scheme-number x))
  (put 'add '(scheme-number scheme-number)
       (lambda (x y) (tag (+ x y))))
  (put 'sub '(scheme-number scheme-number)
       (lambda (x y) (tag (- x y))))
  (put 'mul '(scheme-number scheme-number)
       (lambda (x y) (tag (* x y))))
  (put 'div '(scheme-number scheme-number)
       (lambda (x y) (tag (/ x y))))
  (put 'make 'scheme-number
       (lambda (x) (tag x))
 'done )
```

66





Rational number パッケージ(続)

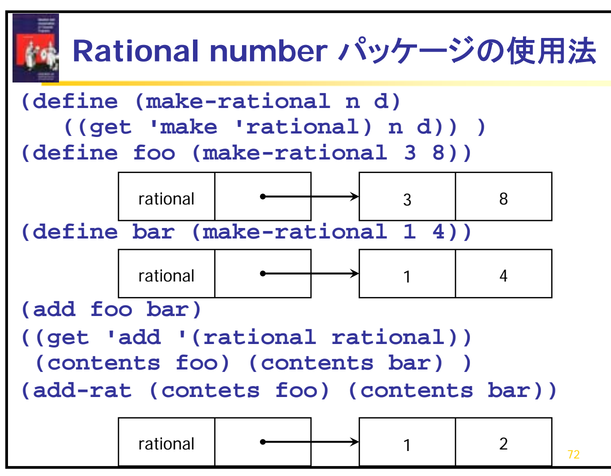
```
(define (install-rational-package)
   ;; internal procedures

  (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)) ))
```


Rational number パッケージ(続々)

```
(define (install-rational-package)
  ;; interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational)
       (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational)
       (lambda (x y) (tag (mul-rat x y))))
  (put 'div '(rational rational)
       (lambda (x y) (tag (div-rat x y))))
  (put 'make 'rational
       (lambda (n d) (tag (make-rat n d))))
  'done )
```


Rational number パッケージ


```
(define (install-rational-package)
  ;; interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational)
       (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational)
       (lambda (x y) (tag (mul-rat x y))))
  (put 'div '(rational rational)
       (lambda (x y) (tag (div-rat x y))))
  (put 'make 'rational
       (lambda (n d) (tag (make-rat n d))))
  'done )
```



Complex number パッケージ

```
(define(install-complex-package)
  ;; i imported procedures from rectangular and polar
  packages

   (define (make-from-real-imag x y)
      ((get 'make-from-real-imag
            'rectangular) x y ))
   (define (make-from-mag-ang r a)
      ((get 'make-from-mag-ang 'polar)
       r a ))
```

73

Complex number パッケージ(続)

```
(define(install-complex-package)
  ;; internal procedures
  (define (add-complex z1 z2)
     (make-from-real-imag
       (+ (real-part z1) (real-part z2))
       (+ (imag-part z1) (imag-part z2))))
  (define (sub-complex z1 z2)
     (make-from-real-imag
       (- (real-part z1) (real-part z2))
       (- (imag-part z1) (imag-part z2))))
  (define (mul-complex z1 z2)
     (make-from-mag-ang
       (* (magnitude z1) (magnitude z2))
       (+ (angle z1) (angle z2))))
```

74

Complex number パッケージ(続々)

```
(define(install-complex-package)
  ;; internal procedures

  (define (div-complex z1 z2)
     (make-from-mag-ang
       (/ (magnitude z1) (magnitude z2))
       (- (angle z1) (angle z2))))

  ;; internal procedures
  (define (tag z) (attach-tag 'complex z))
  (put 'add '(complex complex)
       (lambda (z1 z2)
          (tag (add-complex z1 z2)) ))
```

75

Complex number パッケージ(4)

```
(define(install-complex-package)
  ;; internal procedures

  (put 'sub '(complex complex)
       (lambda (z1 z2)
          (tag (sub-complex z1 z2)) ))
  (put 'mul '(complex complex)
       (lambda (z1 z2)
          (tag (mul-complex z1 z2)) ))
  (put 'div '(complex complex)
       (lambda (z1 z2)
          (tag (div-complex z1 z2)) ))
```

76

Complex number パッケージ(5)

```
(define(install-complex-package)
  ;; internal procedures

  (put 'make-from-real-imag 'complex
       (lambda (x y)
          (tag (make-from-real-imag x y))
       ))
  (put 'make-from-mag-ang 'complex
       (lambda (r a)
          (tag (make-from-mag-ang r a))))
  'done )
```

77

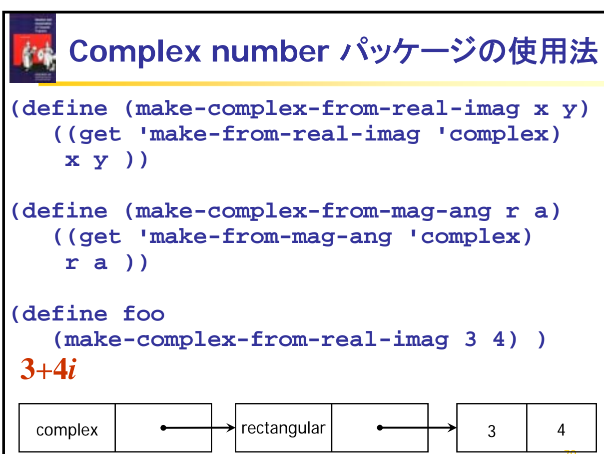
Ex.2.77 Complex number パッケージ

```
(define (make-complex-from-real-imag x y)
   ((get 'make-from-real-imag 'complex)
    x y ))

(define (make-complex-from-mag-ang r a)
   ((get 'make-from-mag-ang 'complex)
    r a ))
```

- Complex number の genetic operations

```
(put 'real-part '(complex) real-part)
(put 'imag-part '(complex) imag-part)
(put 'magnitude '(complex) magnitude)
(put 'angle '(complex) angle)
```

78



Ex.2.78 Ordinary number パッケージ

- Scheme-number を効率化したい
- 基本手続きは、内部でタイプチェックをしている
- symbol? number? pair? などを使用
- scheme-number では、タイプチェックをシステムに任せて、高速化したい。

80

scheme-number の効率化

```
(define (attach-tag type-tag contents)
  (if (eq? type-tag 'scheme-number)
      contents
      (cons type-tag contents) ))
(define (type-tag datum)
  (if (pair? datum)
      (car datum)
      'scheme-number ))
(define (contents datum)
  (if (pair? datum)
      (cdr datum)
      datum ))
```

- タイプ(型)チェックはシステムに任せた！

82

Coercion(強制型変換)

- 異なるタイプ同士に算術演算を拡張する。
- 引数のタイプにより適切な演算を行う手続きを適用
- 手続きを適用する前に、対応するタイプでない引数についてはそのタイプに変換する。

```
(define (scheme-number->complex n)
   (make-complex-from-real-imag
       (contents n) 0 ))
(put-coercion
    'scheme-number 'complex
    scheme-number->complex )
```

89

Coercion(強制型変換)

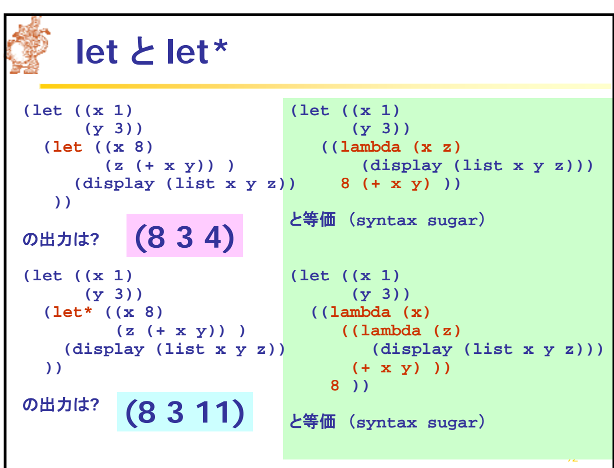
```
(define (apply-generic op . args)
 (let ((type-tags (map type-tag args)))
   (let ((proc (get op type-tags)))
     (if proc
        (apply proc (map contents args))
        (if (= (length args) 2)
            (let ((type1 (car type-tags))
                  (type2 (cadr type-tags))
                  (a1 (car args))
                  (a2 (cadr args)) )
              (let ((t1->t2 (get-coercion type1 type2))
                    (t2->t1 (get-coercion type2 type1)) )
                (cond (t1->t2
                       (apply-generic op (t1->t2 a1) a2))
                      (t2->t1
                       (apply-generic op a1 (t2->t1 a2)))
                      (else
                       (error "No method for these types"
                              (list op type-tags) )))))
            (error "No method for these types"
                   (list op type-tags) ))))))
```

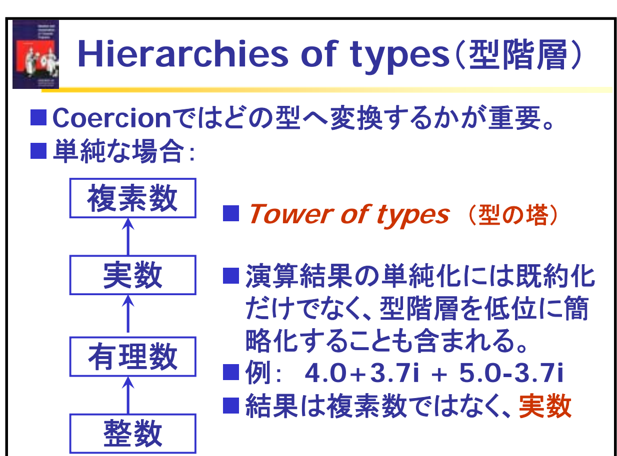
90

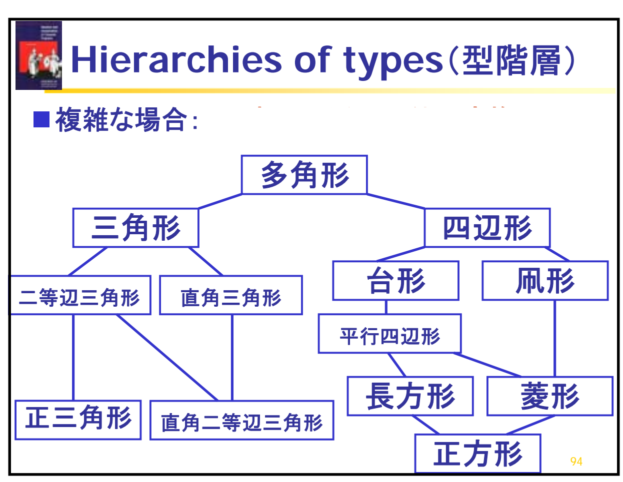
Coercion（強制型変換、改）

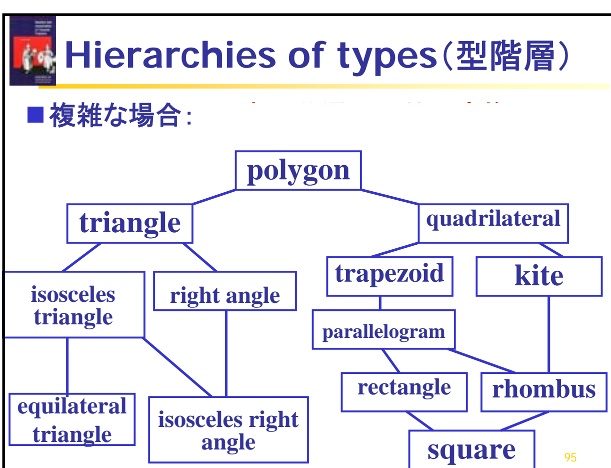
```
(define (apply-generic op . args)
 (let* ((type-tags (map type-tag args))
        (proc (get op type-tags)) )
   (if proc
       (apply proc (map contents args))
       (if (= (length args) 2)
           (let* ((type1 (car type-tags))
                  (type2 (cadr type-tags))
                  (a1 (car args))
                  (a2 (cadr args))
                  (t1->t2 (get-coercion type1 type2))
                  (t2->t1 (get-coercion type2 type1)) )
              (cond (t1->t2
                      (apply-generic op (t1->t2 a1) a2))
                    (t2->t1
                      (apply-generic op a1 (t2->t1 a2)))
                    (else
                      (error "No method for these types"
                             (list op type-tags) )))))
           (error "No method for these types"
                  (list op type-tags) ))))))
```

91









Inadequencies of hierachies

- 演算結果の単純化には既約化だけでなく、型階層を低位に簡略化することも含まれる。
- 例： 4.0+3.7i + 5.0-3.7i
- 結果は複素数ではなく、**実数**
- Ex2.83, 84 `raise` の設計
- Ex2.85 `drop` の設計

96

Abstraction barrier の効用

1. インタフェースの手続き(generic nameで)を定義しておけば、その手続きをどのように実装するかの決定は遅延できる。
2. インタフェースの実装はタイプにより規定。
3. 複数のタイプに対して手続きを適用する前には、対応するタイプに強制型変換を行う。
4. 同じインタフェースを複数のタイプで実装することも可能。

- 汎用演算(generic operations)
- 情報隠蔽(information hiding)

98

前田君の双曲幾何システム

reflection 2point-circle

双曲幾何パッケージ Hyperbolic Geometry

add sub mul div

汎用算術演算パッケージ
Genetic arithmetic package

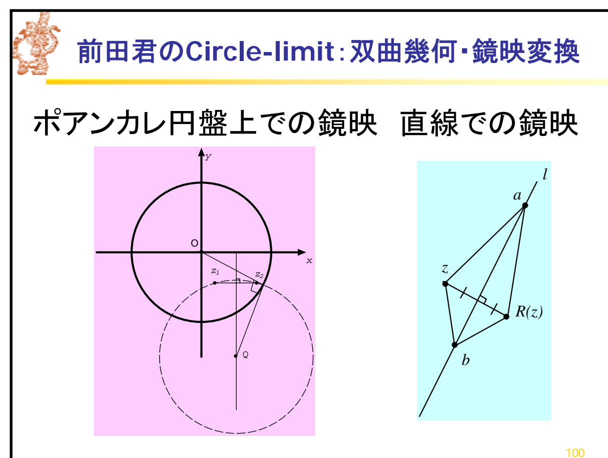
add-rat sub-rat mul-rat div-rat	add-complex sub-complex mul-complex div-complex		+ - * /
有理数算術演算 Rational arithmetic	複素数算術演算 Complex arithmetic		通常算術演算 Ordinary arithmetic
	直交座標表現 Rectangular	極座標表現 Polar	

リスト構造と基本マシン算術演算

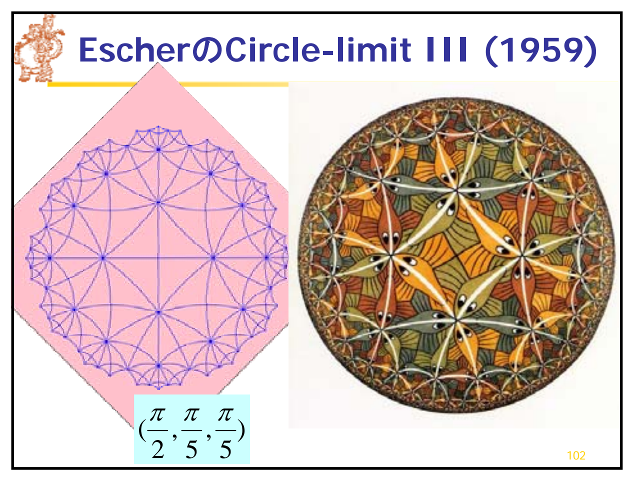
99

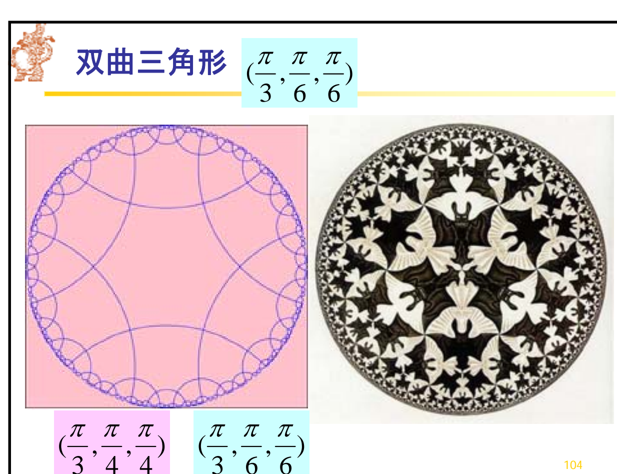
前田君のCircle-limit:双曲幾何・鏡映変換

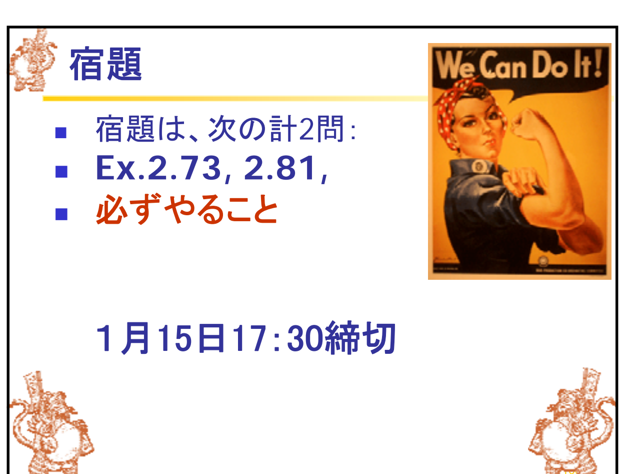
ポアンカレ円盤上での鏡映　直線での鏡映



100


EscherのCircle-limit III (1959)
$(\frac{\pi}{2},\frac{\pi}{5},\frac{\pi}{5})$
102


双曲三角形 $(\frac{\pi}{3},\frac{\pi}{6},\frac{\pi}{6})$
$(\frac{\pi}{3},\frac{\pi}{4},\frac{\pi}{4})$
$(\frac{\pi}{3},\frac{\pi}{6},\frac{\pi}{6})$
104


宿題
宿題は、次の計2問:
Ex.2.73, 2.81,
必ずやること
1月15日17:30締切
We Can Do It!

Sorting（整列）

- **内部整列（internal sorting）**
  - データはすべて主記憶上に置いて整列
  1. 作業領域を極力減らす。
  2. 比較回数を極力減らす。
- **外部整列（external sorting）**
  - 外部の記憶装置を用いて整列
  3. 主記憶と補助記憶との間でのデータ転送回数を極力減らす。

108

Internal Sorting（内部整列）

- 逐次入力型
  1. 挿入ソート（insertion sort）
  2. ヒープソート（heap sort）
- バッチ型
  1. クイックソート（quick sort）
  2. バブルソート（bubble sort）
- その他（外部整列と共通）
  1. マージソート（merge sort）

109

Internal Sorting（内部整列）

- 逐次入力型
  - **挿入ソート（insertion sort）**
  - ヒープソート（heap sort）
- バッチ型
  - **クイックソート（quick sort）**
  - バブルソート（bubble sort）
- その他（外部整列と共通）
  - マージソート（merge sort）

110

安定整列（stable sorting）

- 同じデータ間のもともとの順序が整列後も保存されている整列のこと。
- 基数ソート（radix sort）では重要な性質。
- 辞書式順序で整列で基数ソートが使われる。

111

挿入ソート（insert sort）

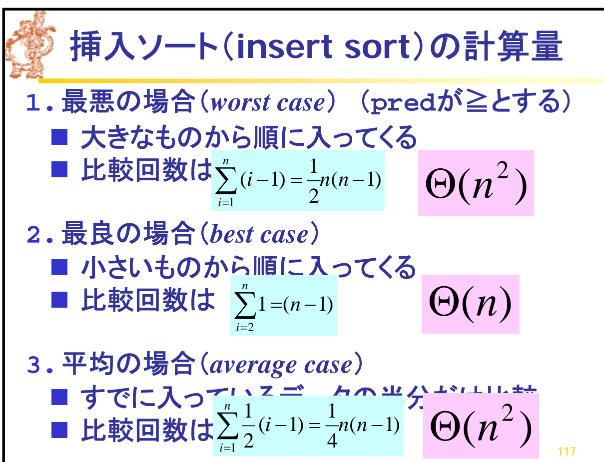
```
(define (insert-sort-pred pred records)
  (if (null? records)
      '()
      (insert-elem pred (car records)
         (insert-sort-pred pred (cdr records))
      )))
(define (insert-elem pred elem ordered-rec)
  (cond ((null? ordered-rec) (cons elem '()))
        ((pred elem (car ordered-rec))
         (cons elem ordered-rec) )
        (else
         (cons (car ordered-rec)
               (insert-elem pred elem
                      (cdr orderd-rec) )))))
(define (insert-sort records . args)
  (insert-sort-pred
     (if (null? args) > (car args))
     records ))
```

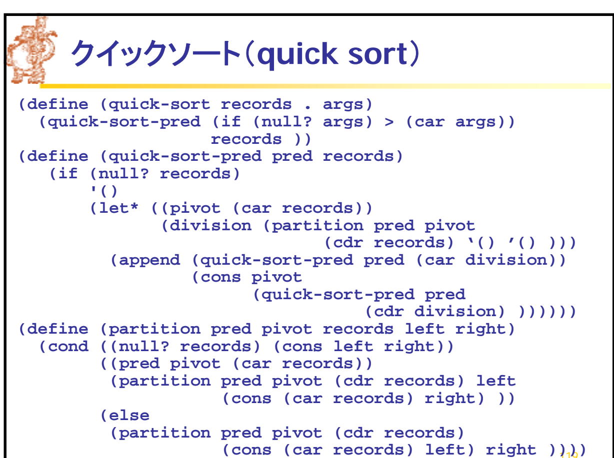
112

挿入ソート（insert sort）の実行トレース

```
(insert-sort-pred > '(2 3 1 6 4))
                      ↑
2
3  2
1
6  3  2  1
   4  3  2  1
```

113





quick sortの実行トレース(まとめ)

```
(quick-sort-pred > '(2 3 1 6 4))

                    2

     (3 6 4)                  (1)

        3                      1

 (6 4)        ()        ()        ()

    6

  ()     (4)

          4

     ()        ()
```

分割統治法

(divide and conquor)

120

quick sortの実行トレース1

```
1.  (quick-sort-pred > '(2 3 1 6 4))
2.  (partition > 2 '(3 1 6 4) '() '())
      ((3 6 4) (1))
3.  (append (quick-sort-pred > '(3 6 4))
        (cons 2 (quick-sort-pred > '(1)) )

3-1. (partition > 3 '(6 4) '() '())
        ((6 4) ())
3-2. (append (quick-sort-pred > '(6 4))
           (cons 3 ()) )
4-1.    (partition > 6 '(4) '() '())
              (() (4))
4-2.    (append ()
                (cons 6 (quick-sort-pred > '(4))) )
5-1.       (partition > 4 '() '() '())
                (() ())
4-3.    (append () (cons 6 (append () (cons 4 ())))))
              (6 4)
4-4.    (6 4)
```

121

quick sortの実行トレース2

```
3-1.   (partition > 3 '(6 4) '() '())
         ((6 4) ())
3-2.   (append (quick-sort-pred > '(6 4))
            (cons 3 ()) )
3-3.   (append '(6 4) (3))
            (6 4 3)
3-4. (append '(6 4 3)
             (cons 2 (quick-sort-pred > '(1)) )
4-1.    (partition > 1 '() '() '())
4-2. (append '(6 4 3)
             (cons 2 '(1)) )
     (6 4 3 2 1)
```

122

クイックソート（quick sort）の計算量

1．最悪の場合（*worst case*）（predが≧とする）
- 小さいものから順に入ってくる
- partitionでの走査回数は

$$\sum_{i=1}^{n}(n-i) = n^2 - \frac{1}{2}n(n+1) = \frac{1}{2}n(n-1)$$

$$\Theta(n^2)$$

2．最良の場合（*best case*）
- 分割がバランスしている
- partitionの呼ばれる回数は

$$\log n \qquad \Theta(n \log n)$$

3．平均の場合（*average case*）

123

クイックソート(quick sort)の計算量

3．平均の場合（*average case*）

- データはすべて異なる。あらゆる順列が等確率。
- 途中での分割でも同様の仮定が成立するとする。

■ n要素のquick sort に要する時間：$T(n)$ とする

■ $T(n) \le T(i) + T(n-i-1) + cn$ （c: 定数）

■ n要素がi要素とn-i-1要素に分割されたとすると

$T(n) \leqq T(i)+T(n-i-1)+cn$

124

クイックソート(quick sort)の計算量

■ n要素の分割、$(0,n-1), (1, n-2), \ldots, (n-1,0)$ が等確率で生ずるとすると、次の漸化式を得る

$$T(n) = cn + \frac{1}{n}\sum_{i=0}^{n-1} T(i) \qquad (n \ge 2)$$

$$T(1) = 0$$

$$T(0) = 1$$

■ 帰納法で証明すると

$$T(n) \approx 2n \log n \qquad \Theta(n \log n)$$

125