アルゴリズムとデータ構造入門
2.3 記号データ (Symbolic Data)

奥 乃 博
大学院情報学研究科 知能情報学専攻音声メディア分野
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/10/IntroAlgDs/
okuno@i.kyoto-u.ac.jp
阿曽 慎平(M1) (10号館4階奥乃1研)
平澤 恭治(M1) (10号館4階奥乃1研)

試験は1月25(火)3限 8号館3階大会議室

1月11日・本日のメニュー

1. 2.4 Multiple Representations for Abstract Data
2. 2.4.1 Representations for Complex Numbers
3. 2.4.2 Tagged data
4. 2.4.3 Data-Directed Programming and Additivity
5. 2.5 Genetic Operation System

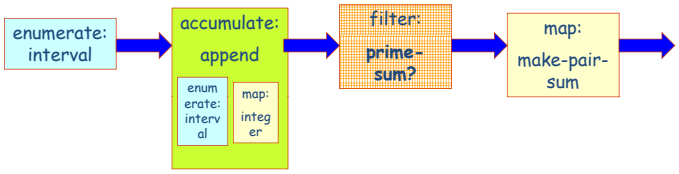
1月25日期末試験 2.2.3~授業

写像の入れ子(nesting of mapping)

1≤j<i≤n なる異なる正の整数 i,j に対して、i+jが素数となるものをすべて求める

n=6のとき

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
i+j	3	5	5	7	7	7	11



list of pairs of integers の作り方

```
(accumulate
   append
   nil
   (map
     (lambda (i)
       (map
         (lambda (j) (list i j))
         (enumerate-interval
                1 (- i 1) )))
     (enumerate-interval 1 n) ))
```

この呼び出しパターンを手続きとして定義

```
(define (flatmap proc seq)
  (accumulate append nil (map proc seq)) )
```


list of pairs of integers の作り方

```
(define (prime-sum? pair)
   (prime? (+ (car pair) (cadr pair)) )

(define (make-pair-sum pair)
   (list (car pair) (cadr pair)
         (+ (car pair) (cadr pair)) ))

(define (prime-sum-pairs n)
   (map make-pair-sum
        (filter prime-sum?
          (flatmap
             (lambda (i)
                (map (lambda (j) (list i j))
                     (enumerate-interval
                          1 (- i 1) )))
             (enumerate-interval 1 n) ))))
```


宿題は、ありません.

期末試験は
2.2.3~本日分+来週分
勉強をしっかりしてください.

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

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

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

複素数の演算

1. 虚数(imaginary part)

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

2. 加算(addition)

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

3. 乗算(multiplication)

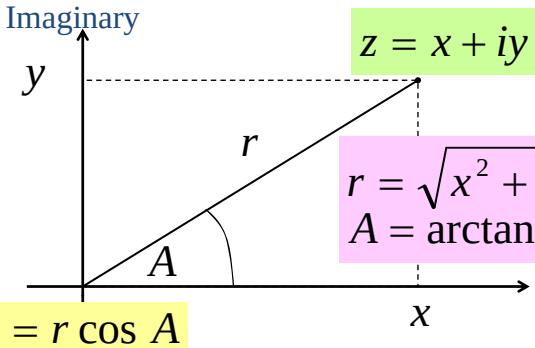
$$\text{Re}(z_1 \cdot z_2) = \text{Re}(z_1) \cdot \text{Re}(z_2) - \text{Im}(z_1) \cdot \text{Im}(z_2)$$
$$\text{Im}(z_1 \cdot z_2) = \text{Re}(z_1) \cdot \text{Im}(z_2) + \text{Im}(z_1) \cdot \text{Re}(z_2)$$
$$\text{Magnitude}(z_1 \cdot z_2) = \text{Magnitude}(z_1) \cdot \text{Magnitude}(z_2)$$
$$\text{Angle}(z_1 \cdot z_2) = \text{Angle}(z_1) + \text{Angle}(z_2)$$

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)) ))
```


複素数の2種類の表現法



$z = x + iy = re^{iA}$

$r = \sqrt{x^2 + y^2}$
$A = \arctan(y, x)$

$x = r\cos A$
$y = r\sin A$

複素数の2種類の表現法の実装

$z = x + iy = re^{iA}$

```
(make-from-real-imag
   (real-part z) (imag-part z) )
(make-from-mag-ang
   (magnitude z) (angle z) )
```

$x = r\cos A$
$y = r\sin A$

$r = \sqrt{x^2 + y^2}$
$A = \arctan(y, x)$

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))) )
```

複素数の表現法(続)

$$z = x + iy = re^{iA}$$

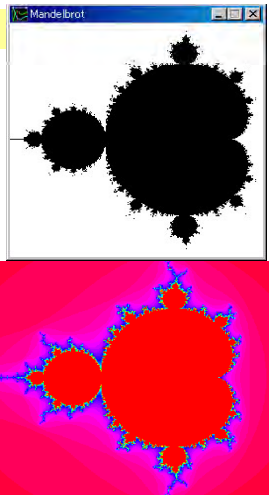
■ 選択子(selectors)

```
(define (real-part z)
  (* (magnitude z) (cos (angle z))) )
(define (imag-part z)
  (* (magnitude z) (sin (angle z))) )
(define (magnitude z) (car z))
(define (angle z) (cdr z))
```

■ 構築子(constructors)

```
(define (make-from-real-imag x y)
  (cons (sqrt (+ (square x) (square y)))
        (atan y x)) )
(define (make-from-mag-ang r a)
  (cons r a) )
```


図形言語に複素数を導入



$$z_{n+1} = z_n^{\,2} + C$$
$$z_0 = C$$

が収束する点$C = (x, y)$
Mandelbrot Set

右上の図はframe coordinate map未使用(直接点を描画)

http://mathworld.wolfram.com/MandelbrotSet.html

1月11日・本日のメニュー

1. 2.4 Multiple Representations for Abstract Data
2. 2.4.1 Representations for Complex Numbers
3. 2.4.2 Tagged data
4. 2.4.3 Data-Directed Programming and Additivity
5. 2.5 Genetic Operation System

Message passing

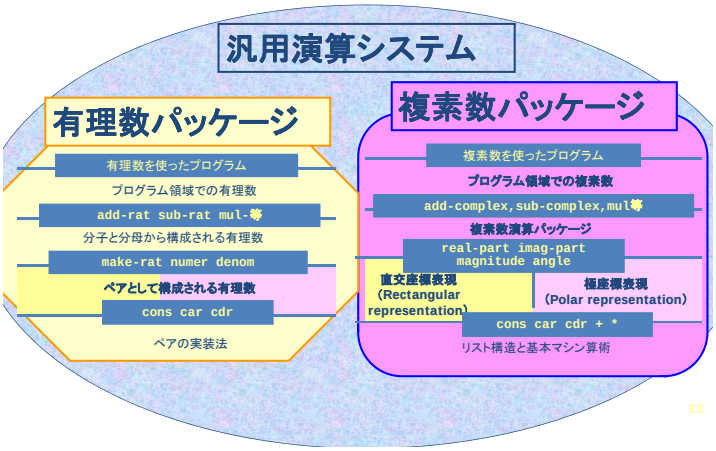
Church numeral と同じ発想

```
(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))
```


1月11日・本日のメニュー

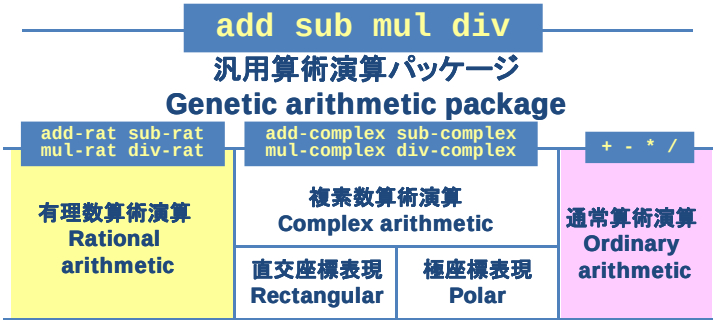
1. 2.4 Multiple Representations for Abstract Data
2. 2.4.1 Representations for Complex Numbers
3. 2.4.2 Tagged data
4. 2.4.3 Data-Directed Programming and Additivity
5. 2.5 Genetic Operation System

本節の目標: 統一システムの構築



汎用演算システムの構造

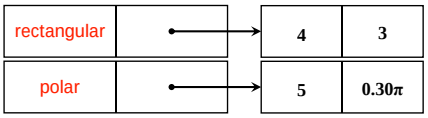
図形言語



汎用演算システム構築のアイデア

Complexで作成した2つのサブタイプ（部分型）
rectangular, polar の構築法を思い出そう

$4+3i$
$=5(\cos 0.30\pi + i\sin 0.30\pi)$
$=5e^{0.30\pi}$




```
(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 ))
```


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
- Generic operation（汎用算術演算）
- 引数のタイプ（型）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) )
```

1. データにはそのタイプを表現するタグ（tag）を付与
2. 演算に引数のタイプの組合せとその手続きを付与.

汎用算術演算システムの設計と課題

1. データにはそのタイプ（型）を表現するタグ（tag）を付与.
2. 演算に引数のタイプの組合せとその手続きを付与
3. 同じタイプ同士の手続きを定義

【課題1】 異なるタイプの組合わせへの対応
- 型階層（Tower of types）
- 強制型変換（coercion）

【課題2】 システム組込みデータタイプへの対応
- 実装ではタグは付けない.

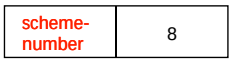
Ordinary number パッケージ

```
(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 )
```

演算に引数の型の組合せとその手続きを付与

Tagとデータ，演算の関係

```
(define (make-scheme-number n)
   ((get 'make 'scheme-number) n) )
(define foo (make-scheme-number 8))
```



が作成される.

- 中身を見る手続きは contents と type-tag
- 汎用演算（add sub mul div）には引数のタイプの組合せに対する手続きの対が並ぶ.

```
((scheme-number scheme-number) . 手続き)
 (rational rational) . 手続き)
 (complex complex) . 手続き)
 ...)
```

- put get の定義を思い出そう（連想リスト）

Scheme number パッケージの使用法

```
(define (make-scheme-number n)
   ((get 'make 'scheme-number) n) )
(define foo (make-scheme-number 8))
```


```
(define bar (make-scheme-number 3))
```


```
(add foo bar)は次と同じ
((get 'add '(scheme-number scheme-number))
  (contents foo) (contents bar) )
(+ 8 3)
```

Rational number パッケージ

```
(define (install-rational-package)
   ;; internal procedures
  (define (numer x) (car x))
  (define (denom x) (cdr x))
  (define (make-rat n d)
    (let ((g (gcd n d)))
       (cons (/ n g) (/ d g))))
  (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)) ))
```


Rational number パッケージ(続)

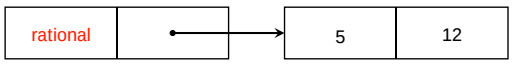
```
(define (install-rational-package)
   ;; internal procedures

  (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)

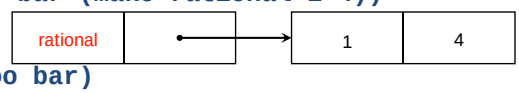
```
(define (install-rational-package)
  ;; interface to rest of the system
  (define (tag x) (attach-tag 'rational x))
  (put 'add '(rational rational)
       (lambda (x y) (tag (add-rat x y))))
  (put 'sub '(rational rational)
       (lambda (x y) (tag (sub-rat x y))))
  (put 'mul '(rational rational)
       (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 パッケージの使用法

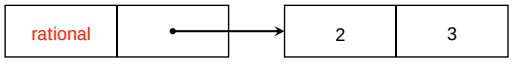
```
(define (make-rational n d)
   ((get 'make 'rational) n d)) )
(define foo (make-rational 5 12))
```




```
(define bar (make-rational 1 4))
```




```
(add foo bar)
((get 'add '(rational rational))
 (contents foo) (contents bar) )
(add-rat (contets foo) (contents bar))
```



Complex number パッケージ

Complexには, rectangular と polar というサブタイプ(部分型)があることを思い出そう.

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

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))))
```


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)) ))
```


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)) ))
```


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 )
```


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 ))
```

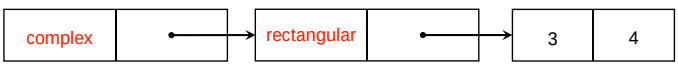
■ Complex number の genetic operations

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


Complex number パッケージの使用法

```
(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 ))

(define foo
   (make-complex-from-real-imag 3 4) )
```

3+4i



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。

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！

Ordinary number パッケージは変更なし

```
(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 )
```


1月11日・本日のメニュー

1. 2.4 Multiple Representations for Abstract Data
2.
3. 2.4.2 Tagged data
4. 2.4.3 Data-Additivity
5. 2.5 Genetic Operation System
6. 2.5.1 Generic Arithmetic Operations
7. 2.5.2 Combining Data of Different Types
8. 2.5.3 Example: Symbolic Algebra

1月12日・本日のメニュー

2.5 Genetic Operation System

1. 2.5.1 Generic Arithmetic Operations
2. 2.5.2 Combining Data of Different Types
3. 2.5.3 Example: Symbolic Algebra

2.5.2 Combining Data of Different Types

- 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
- 次の案はどうか？

```
;; to be included in the complex package
(define (add-complex-to-schemenum z x)
   (make-from-real-imag
      (+ (real-part z) x)
      (imag-part z)))
(put 'add '(complex scheme-number)
   (lambda (z x)
     (tag (add-complex-to-schemenum z x))
   ))
```

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。

```
(+ 3 3.1) ⇒ (+ 3.0 3.1)
(* 3+4i 2) ⇒ (+ 3+4i 2+0i)
```

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


Coercion(強制型変換)

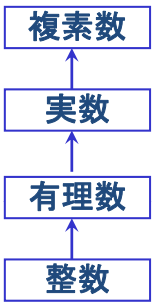
```
(define (apply-generic op . args)
 (let ((type-tags (map type-tag args)))
   (let ((proc (get op type-tags)))
     (if proc
        (apply proc (map contents args))
        (if (= (length args) 2)
            (let ((type1 (car type-tags))
                  (type2 (cadr type-tags))
                  (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) ))))))
```


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) ))))))
```


Hierarchies of types(型階層)

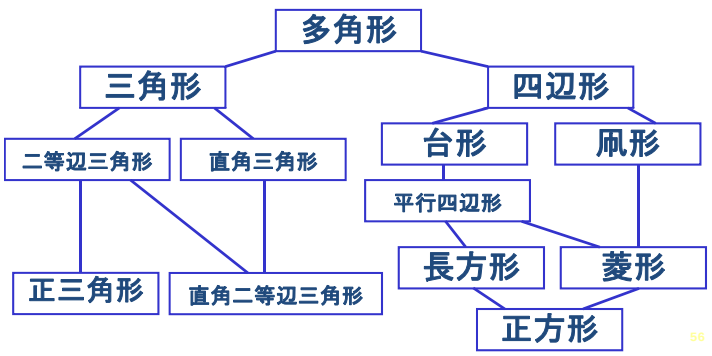
- Coercionではどの型へ変換するかが重要.
- 単純な場合: すぐ上位に変換



- *Tower of types* (型の塔)
- 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

Hierarchies of types(型階層)

複雑な場合: 両者に共通の上位に変換



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

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)**

西陣織(体験工房あり)

- http://kyoori.or.jp/

西陣織「紋意匠図」は西陣織を製織するための設計図、あるいは指図(さしず)ともいう。卦紙(けいがみ)という一種の方眼紙へ、現物の
ばして色別にかき直し
の桁の一コマが経(た
糸の一本にあたる。

紋紙への穴開け作業(ピアノマシン)

紋紙 • http://blog.goo.ne.jp/vitello/

1月18日・本日のメニュー

1. Internal Sorting(内部整列)
   1. 挿入ソート(insertion sort)
   2. クイックソート(quick sort)
2. vector (ベクタ)によるSorting
   1. ヒープソート(heap sort)
   2. バブルソート(bubble sort)
   3. マージソート(merge sort)
3. Searching (探索)
   1. 二分探索(binary search)
   2. ハッシュ法(hashing)

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.

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, 併合ソート)**

Javaによるデモ
http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/09/IntroAlgDs/

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.