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
2.3 記号表現
2.4 抽象データの複数の表現法

奥乃　博

大学院情報学研究科知能情報学専攻

http://winnie.kuis.kyoto-u.ac.jp/~okuno/Lecture/11/IntroAlgDs/

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

平山 直樹　if mod(学籍番号の下3桁, 3) ≡0

黄　 楊暘　if mod(学籍番号の下3桁, 3) ≡1

山口 雄紀　if mod(学籍番号の下3桁, 3) ≡2

1

12月11日・本日のメニュー

- 2.3.2 Symbolic Differentiation
- 2.3.3 Representing Sets
- 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

2

記号微分の拡張（1）

1. 差、商に拡張

```
(deriv '(- x y) 'x)
(deriv '(/ 3 x) 'x)
```

2. 冪乗に拡張

```
(deriv '(** x 3) 'x)
```

3. 2項演算子を多項演算子に拡張

```
(deriv '(+ (* 3 x) y (* x y)) 'x)
(deriv '(* x y (+ x 3)) 'x)
```

4

記号微分の拡張(2)

4. 2項演算子を多項演算子に拡張
   augend, multiplierの定義を変更するだけで
   `(deriv '(+ x (* x y) (** x 3)) 'x)`
   に対応できる。
5. 多項式の整理
   – 多項式を降冪あるいは昇冪の順に整列
   – 多項式を簡略化により整理

   2.5.3 記号代数(Symbolic Algebra)
6. 任意の関数が自由に付加できる微分システム

   2.5.3 Data-Directed Programming and Additivity

5

12月11日・本日のメニュー

2.3 Symbolic Data
2.3.2 Symbolic Differentiation
2.3.3 Representing Sets
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

6

集合(set)の表現

- 自然数の集合を定義してみよう
  1. {0, 1, 2, 3, …}
     外延的記法(extensional notation)
  2. *S = {n|0, n+1 if n ∈S}*
     内延的記法(intentional notation)
- 外延的記法での課題
  次の定義のどちらがよいか?
  1. {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, … }
  2. {0, 10, 20, 30, 2, 12, 22, 24, 4, 14, 24, … }

7

集合(set)の手続きと表現法

- 集合の手続き
  1. union-set　　S∪T
  2. intersection-set　　S∩T
  3. element-of-set?　　e∈T
  4. adjoin-set　　{e}∪S
- 集合の表現法の実装(implementation)
  1. 順序なし表現(*unordered list*)
     {30, 0, 20, 10, 22, 2, 12, 24, 34, … }
     (30 0 20 10 22 2 12 24 34 …)
  2. 順序付き表現(*ordered list*)
     {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, … }
     (0 2 4 6 8 10 12 14 16 18 …)

8

集合(set)のUnordered List表現

```
(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-set? x (cdr set)) ))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set) )))

(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) ))))
```

9

union-set の両者の違いは

```
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                (union-set (cdr s1) s2) )
        )))
(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (union-set (cdr s1)
                 (cons (car s1) s2)) ))))
```

(union-set '(1 2 3) '(a b c))の結果は?

10

集合のunordered list表現(続)

```
(define (intersection-set s1 s2)
  (cond ((or (null? s1) (null? s2)) ())
        ((element-of-set? (car s1) s2)
         (cons (car s1)
           (intersection-set
               (cdr s1) s2 )))
        (else (intersection-set
                   (cdr s1) s2 ))))
```

11

集合手続きの計算量 (#set=n)

```
(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (element-of-set? x (cdr set)) )))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set) )))

(define (union-set s1 s2)
  (cond ((null? s1) s2)
        ((element-of-set? (car s1) s2)
         (union-set (cdr s1) s2) )
        (else (cons (car s1)
                    (union-set (cdr s1) s2) ))))
```

Θ(n)

Θ(n)

#s1=n
#s2=m
Θ(mn)

12

intersection-setの計算量

```
(define (intersection-set s1 s2)
  (cond ((or (null? s1) (null? s2)) ())
        ((element-of-set? (car s1) s2)
         (cons (car s1)
           (intersection-set
               (cdr s1) s2 )))
        (else (intersection-set
                   (cdr s1) s2 ))))
```

計算のオーダは #s1=m1, #s2=m2 とすると、

$\Theta(n^2)$ n=max{m1,m2} (m1*m2 のオーダ)

13

集合(set)のOrdered List表現

```
(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((= x (car set)) #t)
        ((< x (car set)) #f)
        (else (element-of-set? x (cdr set)) ))
```

Θ(n) 平均的には n/2

```
(define (adjoin-set x set)
  (cond ((null? set) (list x))
        ((= x (car set)) set)
        ((< x (car set)) (cons x set))
        (else (cons (car set)
                    (adjoin-set x (cdr set))
              ))))
```

Θ(n) 平均的には n/2

14

集合(set)のOrdered List表現

```
(define (union-set s1 s2)
  (if (null? s1)
      s2
      (let ((x1 (car s1)) (x2 (car s2)))
        (cond ((= x1 x2)
               (cons x1
                     (union-set (cdr s1) (cdr s2))) )
              ((< x1 x2)
               (cons x1 (union-set (cdr s1) s2)) )
              (else
                (cons x2
                      (union-set s1 (cdr s2))
                 ))))))
```

計算のオーダは #s1=m1, #s2=m2 とすると、
Θ(n) n=max{m1,m2}(m1+m2 のオーダ)

15

集合のunordered list表現(続)

```
(define (intersection-set s1 s2)
  (if (or (null? s1) (null? s2))
      ()
      (let ((x1 (car s1)) (x2 (car s2)))
        (cond ((= x1 x2)
               (cons x1
                 (intersection-set (cdr s1) (cdr s2)) )
              ((< x1 x2)
               (intersection-set (cdr s1) s2) )
              (else
               (intersection-set s1 (cdr s2)) )))))
```

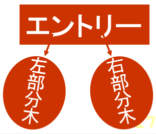
計算のオーダは #s1=m', #s2=m とすると、
Θ(n) n=max{m1,m2} (m+nのオーダ)

16

集合の二進木(binary tree)表現

- リスト構造(木)で集合を表現
- 設計方針
  - 順序付きリストのように制御してしないと，木の高さをhとすると、Θ(h^2)の計算量がかかる
  - 左部分木のエントリーはノードのそれより大きくない
  - 右部分木のエントリーはノードのそれより大きい
- ノードの表現法
  - 次のリストでノードを表現

    (エントリー　左部分木　右部分木)



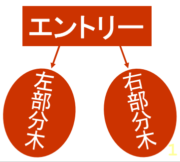
二進木(binary tree)表現の実装

構築子

```
(define (make-tree entry left right)
   (list entry left right) )
```

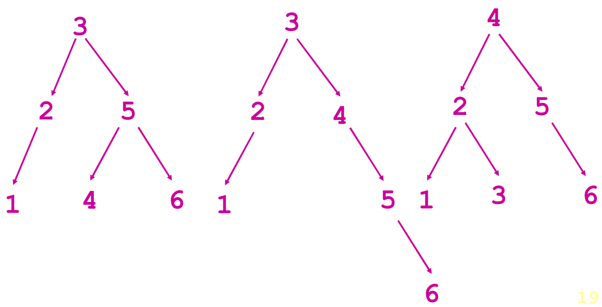
選択子

```
(define (entry tree) (car tree))
(define (left-branch tree) (cadr tree))
(define (right-branch tree)
   (caddr tree))
```



二進木表現の曖昧性

集合{1, 2, 3, 4, 5, 6} の二進木表現



集合(set)のbinary tree表現

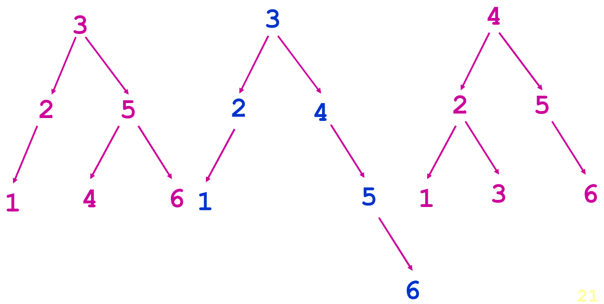
```
(define (element-of-set? x set)
  (cond ((null? set) #f)
        ((= x (entry set)) #t)
        ((< x (entry set))
         (element-of-set? x (left-branch set)) )
        (else
         (element-of-set? x (right-branch set)) )))

(define (adjoin-set x set)
  (cond ((null? set) (make-tree x () ()))
        ((= x (entry set)) set)
        ((< x (entry set))
         (make-tree (entry set)
                (adjoin-set x (left-branch set))
                (right-branch set) ))
        (else
          (make-tree (entry set)
                (left-branch set)
                (adjoin-set x (right-branch set)) ))))
```

20

adjoin-set の動き

3,2,1,5,4,6	3,4,2,5,6,1	4,2,1,5,6,3

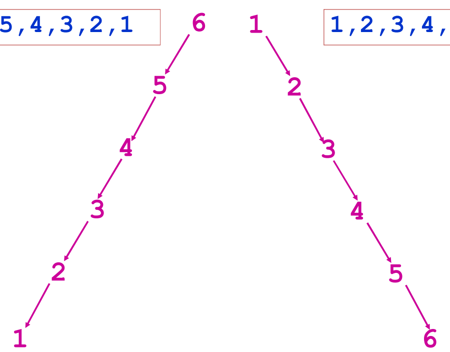


21

adjoin-set の動き

6,5,4,3,2,1

1,2,3,4,5,6



22

Tree⇒list 変換2種類

```
(define (tree->list-1 tree)
  (if (null? tree)
      ()
      (append (tree->list-1 (left-branch tree))
            (cons (entry tree)
                  (tree->list-1 (right-branch tree))))))

(define (tree->list-2 tree)
  (define (copy-to-list tree result-list)
    (if (null? tree)
        result-list
        (copy-to-list (left-branch tree)
               (cons (entry tree)
                     (copy-to-list (right-branch  tree)
                                   result-list )))))
  (copy-to-list tree '()) )
```

両者の違いは？
前順走査・間順走査・後順走査と走査順が違う（第2回）

23

balanced binary tree表現

```
(define (list->tree elements)
   (car (partial-tree elements (length elements))) )

(define (partial-tree elts n)
  (if (= n 0)
      (cons () elts)
      (let ((left-size (quotient (- n 1) 2)))
        (let ((left-result (partial-tree elts left-size)))
           (let ((left-tree (car left-result))
                 (non-left-elts (cdr left-result))
                 (right-size (- n (+ left-size 1))) )
             (let ((this-entry (car non-left-elts))
                   (right-result (partial-tree
                                       (cdr non-left-elts)
                                       right-size )))
               (let ((right-tree (car right-result))
                     (remaining-elts (cdr right-result)) )
                 (cons (make-tree this-entry
                                  left-tree
                                  right-tree )
                       remaining-elts ))))))))
```

24

balanced binary tree表現(改)

```
(define (list->tree elements)
   (car (partial-tree elements (length elements))) )

(define (partial-tree elts n)
  (if (= n 0)
      (cons () elts)
      (let* ((left-size (quotient (- n 1) 2))
             (left-result (partial-tree elts left-size))
             (left-tree (car left-result))
             (non-left-elts (cdr left-result))
             (right-size (- n (+ left-size 1)))
             (this-entry (car non-left-elts))
             (right-result
                (partial-tree (cdr non-left-elts) right-size) ))
             (right-tree (car right-result))
             (remaining-elts (cdr right-result)) )
        (cons
           (make-tree this-entry left-tree right-tree)
           remaining-elts ))))
```

25

Sets & information retrieval

```
(define (lookup given-key set-of-records)
   (cond ((null? set-of-records) #f)
         ((equal? given-key (key (car set-of-records)))
          (car  set-of-records) )
         (else (lookup given-key (cdr set-of-records))) ))
```

連想リスト(a-list, associative list)

((<属性> . <値のリスト>)
 (<attribute> . <value-list>)
 ...)

```
(define (assoc given-key set-of-records)
   (cond ((null? set-of-records) #f)
         ((equal? given-key (caar set-of-records))
          (cdar  set-of-records) )
         (else (assoc given-key (cdr set-of-records))) ))
```

26

簡単な情報検索

```
(define (lookup given-key set-of-records)
   (cond ((null? set-of-records) #f)
         ((equal? given-key (key (car set-of-records)))
          (car  set-of-records) )
         (else (lookup given-key (cdr set-of-records)))))
(define population
  '((China 1285.0 660.5 624.5)
    (India 1025.1 528.5 496.6)
    (USA    285.9 141.0 144.9)
    (Indonesia 214.8 107.8 107.1)
    (Brazil 172.6 85.2 87.4)
    (Pakistan 145.0 74.5 70.5)
    (Russia   144.7 67.7 77.0)
    (Bangradesh 140.4 72.3 68.0)
    (Japan    127.1 62.2 65.0)
    (Nigeria  116.9 59.0 58.0)
    (Mexico   100.4 49.6 50.7) ))
(lookup 'Japan population)
(assoc 'Japan population) = (cdr (lookup 'Japan population)
```

連想リスト(a-list, associative list)

((<属性> . <値のリスト>)
 (<attribute> . <value-list>)
 …
)

27

key の順序

1. 数
   1. 昇順(increasing order, ascending order) <
   2. 降順(decreasing order, descending order) >
2. 辞書式順序(lexicographical order)
   1. (string=? "PIE" "pie")
   2. (string-ci=? "PIE" "pie")
   3. string<?, string<=?, …
   4. char=?, char-ci=?, char>?, char>=?, …
3. alphanumeric order

28

ソーティングの応用

1. 本1冊に出てくる単語の頻度を求めよ。
2. Unix の pipe で処理
   次の1行のコマンドでできる。

```
tr `[ ¥t,.;:]*' `¥n' < file | 改行不可
tr '[A-Z]' '[a-z]' | sort | 改行不可
uniq -c | sort -r
```

3. www.gutenberg.org よりフルテキストを入手、
   - *Gulliver's Travel (Swift)*
     the 2894, of 1844, and 1755, to 1557,
     i 1311, a 1177, in 984, my 768, was 625
   - *TAO (Lao-Tsu)*
     the 675, and 373, to 345, of 335, is 290
     it 225, not 164, in 154, he 136, a 136

29

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

30

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）

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

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

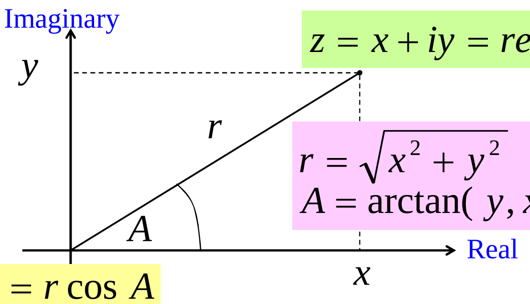
31

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



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

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

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

33

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) )
```

$x = r \cos A$

$y = r \sin A$

$r = \sqrt{x^2 + y^2}$

$A = \arctan(y, x)$

34

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) )
```

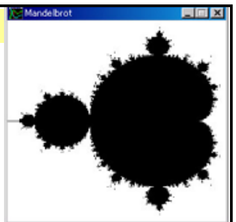
図形言語に複素数を導入

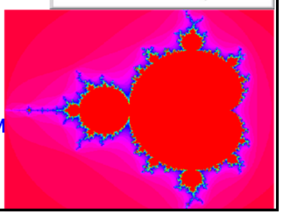
$$z_{n+1} = z_n^{\ 2} + C$$

$$z_0 = C$$

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

宿題:12月25日正午締切

1. 複素数システムを実装し，プログラムと実行例，および，それらの説明をレポート．（簡略化も行うこと）
2. 実行例を添付すること （簡略化の例も実行すること）
3. Program ファイルとレポート（pdf）を
   SICP-11@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!



38

12月11日・本日のメニュー

2.3 Symbolic Data
2.3.2 Symbolic Differentiation
2.3.3 Representing Sets
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

39

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))
```

40

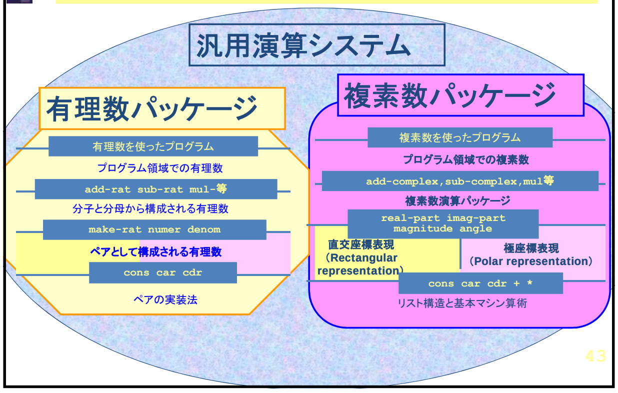
12月11日・本日のメニュー

NoStudent Left Behind

2.3 Symbolic Data
2.3.2 Symbolic Differentiation
2.3.3 Representing Sets
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

41

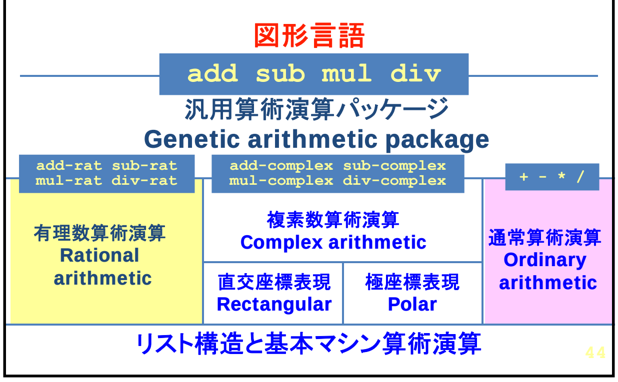
本節の目標： 統一システムの構築



43

汎用演算システムの構造

NoStudent Left Behind

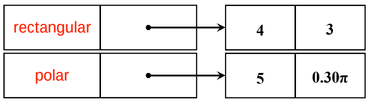


44

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 ))
```

45

西陣織(体験工房あり)

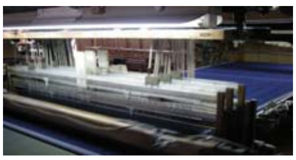
- http://kyoori.or.jp/

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、現物の図案を引きのばして色別にかき直したもの。方眼紙の桁の一コマが経(たて)糸、緯(よこ)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/



47