アルゴリズムとデータ構造入門

2.2.4 図形言語

2.2.4 Picture Language

奥 乃 博

The First Commandment

When recurring on a list of atoms, `lat`, ask two questions about it: `(null? lat)` and else.
When recurring on a number, `n`, ask two questions about it: `(zero? n)` and else.
When recurring on a list of S-expressions, `l`, ask three questions about it : `(null? l)`, `(atom? (car l))`, and else.

The Fourth Commandment

Always change at least one argument while recurring. When recurring on a list of atoms, `lat`, user `(cdr lat)`. When recurring on a number, `n`, use `(sub1 n)`. And when recurring on a list of S-expressions, `l`, use `(car l)` and `(cdr l)` if neither `(null? l)` nor `(atom? (car l))` are true.

It must be changed to be closer to termination. The changing argument must be tested in the termination condition:
when using `cdr`, test termination with `null?` and
when using `sub1`, test termination with `zero?`.

(Friedman, et al. "*The Little Schemer*", MIT Press)

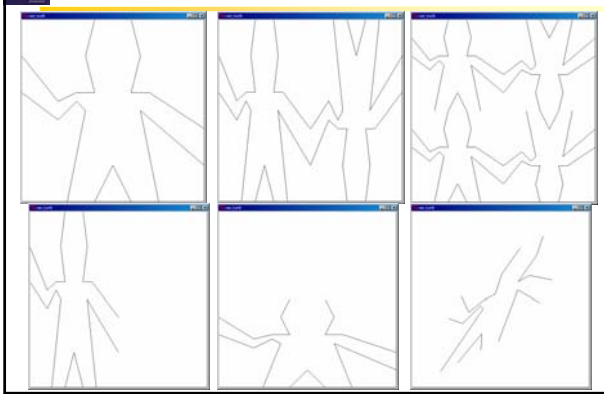
1

11月29日・本日のメニュー

- OCWに昨年度の講義資料公開
- 2 Building Abstractions with Data
  - 2.2.4 Picture Language
  - Space Padding Functions
  - Fractal (Self-Similarity)
- Hilbert curve
- Koch snowflake
- Sierpinski's Gasket
- Peano curve
- Square limit variation

2

図形言語 Picture language とは



flipped-pair

```
(define wave2 (beside wave (flip-vert
 wave)))
(define wave4 (below wave2 wave2))

(define (flipped-pairs painter)
       (let ((painter2 (beside painter
  (flip-vert painter))))
         (below painter2 painter2)))
```

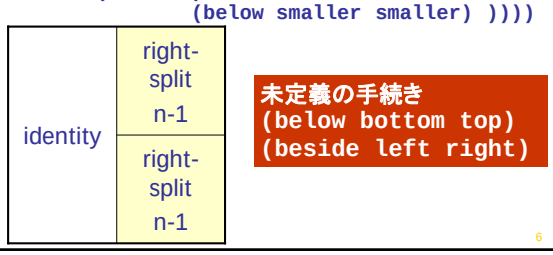
こうすると

```
(define wave4 (flipped-pairs wave))
```

5

right-split *n*

```
(define (right-split painter n)
   (if (= n 0)
       painter
       (let ((smaller
              (right-split painter (- n 1))) )
          (beside painter
                  (below smaller smaller) ))))
```

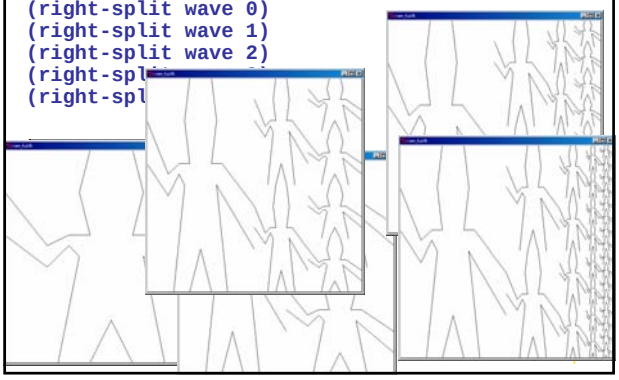


未定義の手続き
(below bottom top)
(beside left right)

6

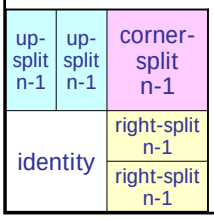
right-split *n* の動き

```
(right-split wave 0)
(right-split wave 1)
(right-split wave 2)
(right-spl
(right-spl
```



corner-split *n*

```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))) )
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1))) )
          (beside (below painter top-left)
                  (below bottom-right corner) )))))
```

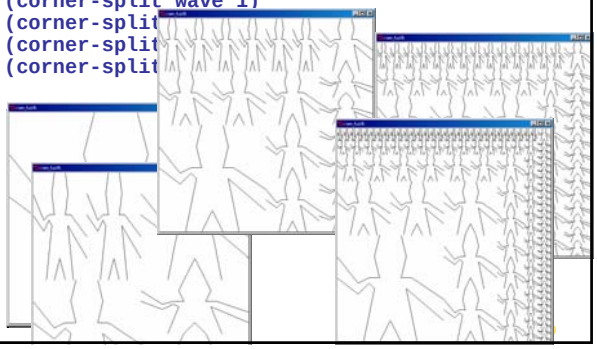


未定義の手続き
(below bottom top)
(beside left right)

9

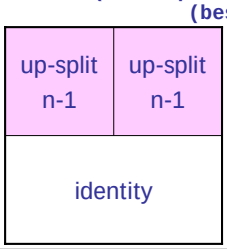
corner-split *n* の動き

```
(corner-split wave 0)
(corner-split wave 1)
(corner-split
(corner-split
(corner-split
```



Ex.2.44 up-split

```
(define (up-split painter n)
  (if (= n 0)
      painter
      (let ((smaller
             (up-split painter (- n 1))) )
        (below painter
               (beside smaller smaller) ))))
```

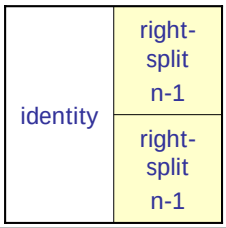


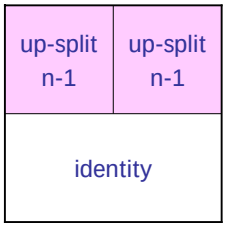
未定義の手続き
(below bottom top)
(beside left right)

12

Ex.2.45 right-split

```
(define right-split (split beside below))
(define up-split (split below beside))
(define (split op1 op2)
   (op1 half (op2 quarter quarter)) ))
```

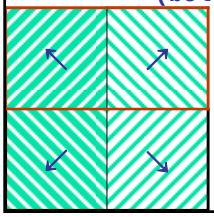




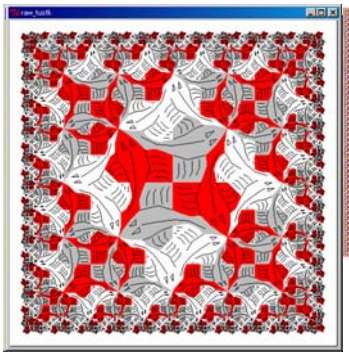
13

square-limit *n*

```
(define (square-limit painter n)
   (let ((quarter
          (corner-split painter n)) )
      (let ((half (beside
                   (flip-horiz quarter)
                   quarter )))
         (below (flip-vert half)
                half))))
```



Escher's square-limit



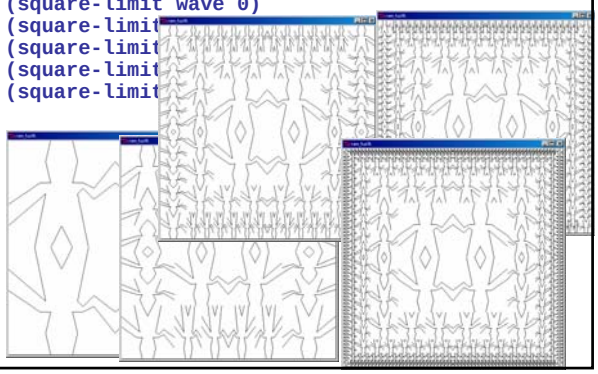
和田：関数画家、『情報処理』、Vol.46, No.10 (2005) 1163-1171

http://www.ecs.soton.ac.uk/~ph/papers/funcgeo2.pdf

16

square-limit *n* の動き

```
(square-limit wave 0)
(square-limit
(square-limit
(square-limit
(square-limit
```



Higher-order operations

```
(define (square-of-four tl tr bl br)
   (lambda (painter)
      (let ((top (beside (tl painter)
                         (tr painter)) )
            (bottom (beside (bl painter)
                            (br painter) )))
         (below bottom top) )))
```

tl	tr
bl	br

未定義の手続き
(below bottom top)
(beside left right)

20

flipped-pairs

```
(define (flipped-pairs painter)
   (let ((combine4
          (square-of-four
               identity flip-vert
               identity flip-vert )))
      (combine4 painter) ))
```

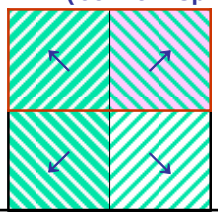
identity	flip-vert
identity	flip-vert

未定義の手続き
(below bottom top)
(beside left right)

21

flipped-pairs

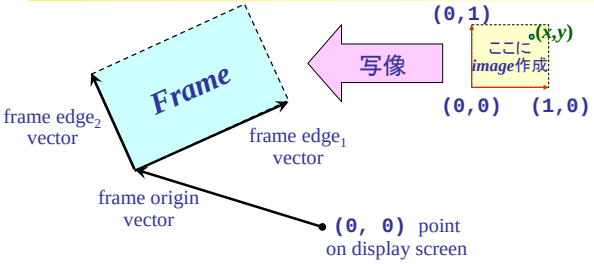
```
(define (square-limit painter n)
  (let ((combine4
         (square-of-four
             flip-horiz identity
             rotate180 flip-vert )))
    (combine4
       (corner-split painter n)) ))
```



flip-horiz	identity
rotate180	flip-vert

22

Frame coordinate map



- coordinate: unit squareに作成
- frameに写像

$$\overrightarrow{\text{Origin}}(\text{Frame}) + x \cdot \overrightarrow{\text{Edge}_1}(\text{Frame}) + y \cdot \overrightarrow{\text{Edge}_2}(\text{Frame})$$

24

Frames

```
(define (frame-coord-map frame)
  (lambda (v)
    (add-vect
      (origin-frame frame)
      (add-vect (scale-vect (xcor-vect v)
                    (edge1-frame frame) )
                (scale-vect (ycor-vect v)
                    (edge2-frame frame) )
      ))))

    ((frame-coord-map a-frame)
     (make-vect 0 0) )
```

の返す値: (origin-frame a-frame)

25

Frames

```
(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2) )

(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)) )
```

26

Painters

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
     (lambda (segment)
       (draw-line
        ((frame-coord-map frame)
         (start-segment segment) )
        ((frame-coord-map frame)
         (end-segment segment) )))
     segment-list )))
```

28

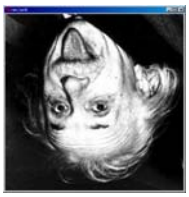
Transforming and combining painters

```
(define (transform-painter painter
                origin corner1 corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
         (make-frame new-origin
            (sub-vect (m corner1)
                      new-origin)
            (sub-vect (m corner2)
                      new-origin)))))))
```

29

Transforming and combining painters

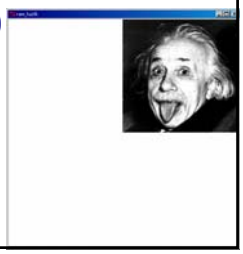
```
(define (flip-vert painter)
  (transform-painter
     painter
     (make-vect 0.0 1.0) ;  new origin
     (make-vect 1.0 1.0) ;  new end of edge1
     (make-vect 0.0 0.0))) ; new end of edge2
```



30

Transforming and combining painters

```
(define (shrink-to-upper-right painter)
  (transform-painter
      painter
     (make-vect 0.5 0.5)
     (make-vect 1.0 0.5)
     (make-vect 0.5 1.0))
```



Transforming and combining painters

```
(define (rotate90 painter)
  (transform-painter
      painter
     (make-vect 1.0 0.0)
     (make-vect 1.0 1.0)
     (make-vect 0.0 0.0)))
```

32

Transforming and combining painters

```
(define (squash-inwards painter)
  (transform-painter
      painter
      (make-vect 0.0 0.0)
      (make-vect 0.65 0.35)
      (make-vect 0.35 0.65)))
```

33

beside

```
(define (beside painter1 painter2)
  (let ((split-point (make-vect 0.5 0.0)))
    (let ((paint-left
           (transform-painter
                painter1
                (make-vect 0.0 0.0)
                split-point
                (make-vect 0.0 1.0)))
          (paint-right
           (transform-painter
                 painter2
                 split-point
                 (make-vect 1.0 0.0)
                 (make-vect 0.5 1.0))))
      (lambda (frame)
        (paint-left frame)
        (paint-right frame)))))
```

35

beside

```
(define (above painter1 painter2 . l)
  (let* ((m (if (null? l) 1 (car l)))
         (n (if (or (null? l) (null? (cdr l)))
                1 (cadr l)))
         (r (/ n (+ m n)))
         (split-point (make-vect 0.0 r)) )
    (let ((paint-lower
           (transform-painter painter2
                              (make-vect 0.0 0.0)
                              (make-vect 1.0 0.0)
                              split-point))
          (paint-upper
           (transform-painter painter1
                              split-point
                              (make-vect 1.0 r)
                              (make-vect 0.0 1.0))))
      (lambda (frame)
        (paint-lower frame)
        (paint-upper frame)))))
```

36

frame coordination map

```
(define (frame-coord-map frame)
  (lambda (v)
    (add-vect
     (origin-frame frame)
     (add-vect (scale-vect (xcor-vect v)
                           (edge1-frame frame))
               (scale-vect (ycor-vect v)
                           (edge2-frame frame)) ))))
;: ((frame-coord-map a-frame) (make-vect 0 0))
;: (origin-frame a-frame)

(define (make-frame origin edge1 edge2)
  (list origin edge1 edge2))
(define (make-frame origin edge1 edge2)
  (cons origin (cons edge1 edge2)))
```

37

描画のための基本手続き

```
(define (segments->painter segment-list)
  (lambda (frame)
    (for-each
     (lambda (segment)
       (draw-line ((frame-coord-map frame)
                   (start-segment segment))
                  ((frame-coord-map frame)
                   (end-segment segment)) ))
     segment-list )))

(define (transform-painter painter origin corner1
  corner2)
  (lambda (frame)
    (let ((m (frame-coord-map frame)))
      (let ((new-origin (m origin)))
        (painter
         (make-frame new-origin
              (sub-vect (m corner1) new-origin)
              (sub-vect (m corner2) new-origin) ))))))
```

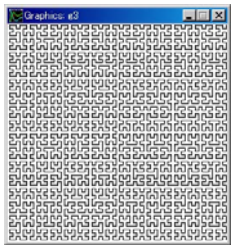
38

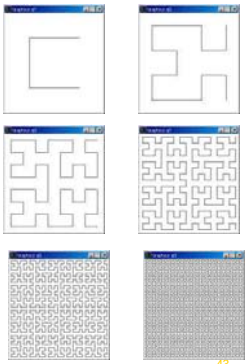
11月29日・本日のメニュー

- 2 Building Abstractions with Data
- 2.2.4 Picture Language
- Space Padding Functions
- Fractal (Self-Similarity)
- Hilbert curve
- Koch snowflake
- Sierpinski's Gasket
- Peano curve
- Square limit variation

41

Hilbert curve の作成方法





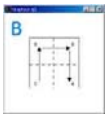
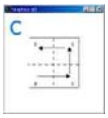
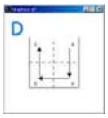
```
(hilbert 5)
```

43

Hilbert curve の作成方法

4つの基本形

1. 基本形A: D⇒A⇒A⇒B
2. 基本形B: C⇒B⇒B⇒A
3. 基本形C: B⇒C⇒C⇒D
4. 基本形D: A⇒D⇒D⇒C

基本形A	基本形B	基本形C	基本形D
			
分解形A	分解形B	分解形C	分解形D

45

Hilbert curve の手続き

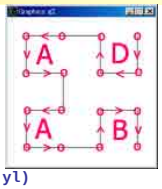
1. 各基本形に対して、レベル0ならコ型を書くための頂点のリストを求める。
2. さもなければ、分解形を再帰的に呼び出し、頂点を求める。
3. 求まった頂点リストから segment を求め `painter` を `vectors->segment` と `segments->painter`を使って作成する。

```
(vectors->segment <list of vectors>)
(segments->painter <list of segments>)
```

46

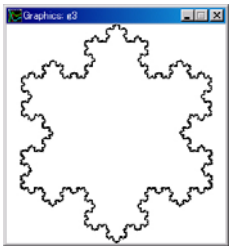
Hilbert curve の手続き

```
(define (hilbert-a x0 y0 x1 y1 i)
  (let ((xs (/ (+ (* 3.0 x0) x1) 4.0))
        (ys (/ (+ (* 3.0 y0) y1) 4.0))
        (xm (/ (+ x0 x1) 2.0))
        (ym (/ (+ y0 y1) 2.0))
        (xl (/ (+ x0 (* 3.0 x1)) 4.0))
        (yl (/ (+ y0 (* 3.0 y1)) 4.0)) )
    (if (= i 0)
        (list (make-vect xl yl) (make-vect xs yl)
              (make-vect xs ys) (make-vect xl ys) )
        (append (hilbert-d xm ym x1 y1 (- i 1))
                (hilbert-a x0 ym xm y1 (- i 1))
                (hilbert-a x0 y0 xm ym (- i 1))
                (hilbert-b xm y0 x1 ym (- i 1)) ))))

(define (hilbert n)
  (segments->painter
    (vectors->segments (hilbert-a 0.0 0.0 1.0 1.0 n))) )
```

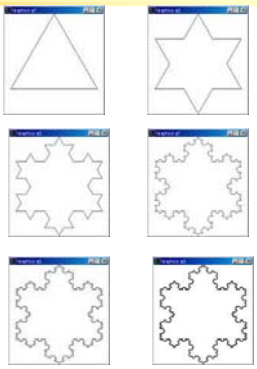


47

Koch snowflake の作成方法



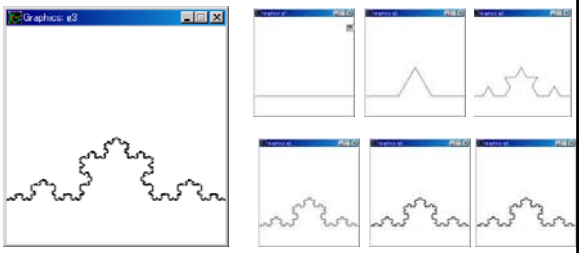
(koch 5)



48

Koch snowflake の作成方法

線分の分解



49

Koch snowflake の手続き

1. 各線分に対して、レベル0なら、三角形の頂点リストを求める。
2. さもなければ、分解形を再帰的に呼び出し、頂点を求める。
3. 求まった頂点リストから segment を求め painter を vectors->segment と segments->painterを使って作成する。

```
(vectors->segment <list of vectors>)
(segments->painter <list of segments>)
```

50

Koch snowflake の手続き（続）

```
(define (koch-line x0 y0 x1 y1 r i)
  (if (= i 0)
      (list (make-vect x0 y0) (make-vect x1 y1))
      (let* ((r1 (/ r 3.0))
             (x3 (/ (- x1 x0) 3.0))
             (y3 (/ (- y1 y0) 3.0))
             (xs (/ (+ (* 2.0 x0) x1) 3.0))
             (ys (/ (+ (* 2.0 y0) y1) 3.0))
             (xl (/ (+ x0 (* 2.0 x1)) 3.0))
             (yl (/ (+ y0 (* 2.0 y1)) 3.0))
             (xm (+ (* 0.5 x3) (* 0.866 y3) xs))
             (ym (+ (* 0.5 y3) (* -0.866 x3) ys)) )
        (append (koch-line x0 y0 xs ys r1 (- i 1))
                (koch-line xs ys xm ym r1 (- i 1))
                (koch-line xm ym xl yl r1 (- i 1))
                (koch-line xl yl x1 y1 r1 (- i 1))
        ))))
```

51

Koch snowflake の手続き（続）

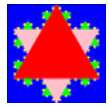
```
(define (koch n)
  (let* ((h (/ 0.75 0.86))
         (x0 (/ (- 1.0 h) 2))
         (x1 (- 1.0 x0)) )
    (segments->painter
      (vectors->segments
        (append
          (koch-line x0 0.25 x1 0.25 1 n)
          (koch-line x1 0.25 0.5 1.0 1 n)
          (koch-line 0.5 1.0 x0 0.25 1 n)
        )))))
```

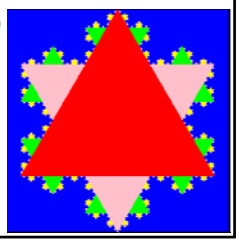
let* は let と違い、変数値対を順番に評価

52

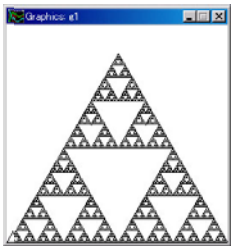
Koch curve の手続き

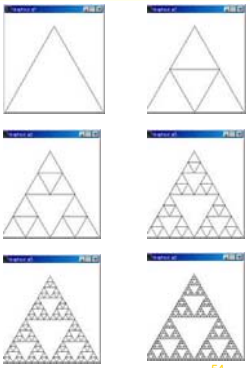
```
(define (koch-fill n . args)
  (let* ((h (/ 0.75 0.86))
         (x0 (/ (- 1.0 h) 2))
         (x1 (- 1.0 x0))
         (color (if (null? args) 'red (car args))) )
    (vects->painter
      (append (koch-line x0 0.25 x1 0.25 1 n)
        (koch-line x1 0.25 0.5 1.0 1 n)
        (koch-line 0.5 1.0 x0 0.25 1 n))
      #f 0 color )))

(define (fun-koch x)
   ((koch-fill 5 'pink) x)
   ((koch-fill 4 'while) x)
   ((koch-fill 3 'yellow) x)
   ((koch-fill 2 'green) x)
   ((koch-fill 1 'pink) x)
   ((koch-fill 0 'red) x)
  )
```





Sierpinski's Gasket の作成方法





```
(sierpiski_6)
```

54

Sierpinski's Gasket の手続き

```
(define (gachet x0 y0 x1 y1 i)
  (let* ((xm (/ (+ x0 x1) 2.0))
         (ym (+ (* (- x1 x0) 0.866) y0))
         (xs (/ (+ (* 3.0 x0) x1) 4.0))
         (xl (/ (+ (* 3.0 x1) x0) 4.0))
         (ys (+ (* (- x1 x0) 0.433) y0)) )
    (if (= i 0)
       (list (make-vect x0 y0) (make-vect x1 y1)
             (make-vect (/ (+ x0 x1) 2.0) ym)
             (make-vect x0 y0) )
       (append (gachet x0 y0 xm y0 (- i 1))
               (gachet xm y0 x1 y0 (- i 1))
               (list (make-vect x0 y0))
               (gachet xs ys xl ys (- i 1))
               (list (make-vect x0 y0) )))))

(define (sierpenski n)
  (segments->painter
    (vectors->segments (gachet 0.0 0.0 1.0 0.0 n)) ))
```

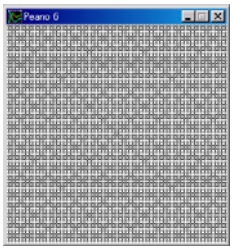
55

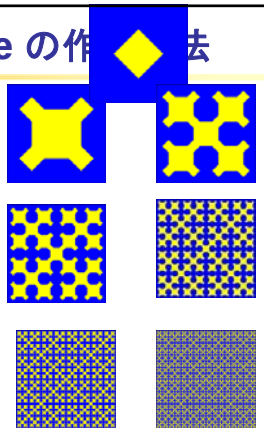
Sierpinski's Gasket の手続き

```
(define (gachet-fill x0 y0 x1 y1 i color)
  (let* ((xm (/ (+ x0 x1) 2.0))
         (ym (+ (* (- x1 x0) 0.866) y0))
         (xs (/ (+ (* 3.0 x0) x1) 4.0))
         (xl (/ (+ (* 3.0 x1) x0) 4.0))
         (ys (+ (* (- x1 x0) 0.433) y0)) )
    (if (= i 0)
        (list (vects->painter
                (list (make-vect x0 y0)
                      (make-vect x1 y1)
                      (make-vect xm ym) )
                #f 0 color))
        (append (gachet-fill x0 y0 xm y0 (- i 1) color)
                (gachet-fill xm y0 x1 y0 (- i 1) color)
                (gachet-fill xs ys xl ys (- i 1) color) ))))

(define (sierpenski-fill n . args)
  (let ((color (if (null? args) 'red (car args))))
    (do ((i (gachet-fill 0.0 0.0 1.0 0.0 n color) (cdr i)))
        ((null? i))
      (i frm1) )))
```

56

Peano Curve の作法



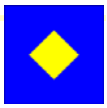


(peano 6)

57

Peano Curve の手続き

```
(define (peano-a x0 y0 x1 y1 i)
  (append (peano-a-1 x0 y0 x1 y1 i)
    (peano-a-2 x0 y0 x1 y1 i) ))

(define (peano-a-1 x0 y0 x1 y1 i)
  (let ((xs (/ (+ (* 3.0 x0) x1) 4.0)) (ys (/ (+ (* 3.0 y0) y1) 4.0))
        (xm (/ (+ x0 x1) 2.0)) (ym (/ (+ y0 y1) 2.0))
        (xl (/ (+ x0 (* 3.0 x1)) 4.0)) (yl (/ (+ y0 (* 3.0 y1)) 4.0)) )
    (if (= i 0)
        (list (make-vect xm yl) (make-vect xs ym))
        (append (peano-a-1 xm ym x1 y1 (- i 1))
                (peano-d-1 x0 ym xm y1 (- i 1))
                (peano-d-2 x0 ym xm y1 (- i 1))
                (peano-a-1 x0 y0 xm ym (- i 1)) ))))

(define (peano-a-2 x0 y0 x1 y1 i)
  (let ((xs (/ (+ (* 3.0 x0) x1) 4.0)) (ys (/ (+ (* 3.0 y0) y1) 4.0))
        (xm (/ (+ x0 x1) 2.0)) (ym (/ (+ y0 y1) 2.0))
        (xl (/ (+ x0 (* 3.0 x1)) 4.0)) (yl (/ (+ y0 (* 3.0 y1)) 4.0)) )
    (if (= i 0)
        (list (make-vect xm ys) (make-vect xl ym))
        (append (peano-a-2 x0 y0 xm ym (- i 1))
                (peano-b-1 xm y0 x1 ym (- i 1))
                (peano-b-2 xm y0 x1 ym (- i 1))
                (peano-a-2 xm ym x1 y1 (- i 1)) ))))
```



58

必修課題3：2月15日午後5時締切

1. 気の利いたpainterを1種類作れ。
2. 空間充填曲線を1種類作れ。
   （Hilbert curve, Peano curve, ...）
3. フラクタルを一種類作れ。
   （Koch Snowflake, Sierpinsky's Gasket, ...）

プログラムはメイルで okuno@i.kyoto-u.ac.jp
例は：http://winnie.kuis.kyoto-u.ac.jp/ にあり

随意課題3：3月15日午後5時締切

circle-limit を作成せよ。

プログラムはメイルで
okuno@i.kyoto-u.ac.jp

祝 成功
京都大學
１１月祭

- 宿題は、次の計9問：
- **Ex.2.36, 2.37, 2.40, 2.42**
- **Ex.2.44～52　のうち5題選択**

11月28日午後5時締切