アルゴリズムとデータ構造入門
2.2.4 図形言語(Picture Language)

奥 乃 博

試験欠席者は
所定の手続きを

The Eighth Commandment
Use help function to abstract from representations.

The Ninth Commandment
Abstract common patterns with a new function.

The Tenth Commandment
Build functions to collect more than one value at a time.

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

12月12日・本日のメニュー

今日は必修課題の説明です.

1. 2.2.4 Picture Language(図形言語)
2. Space Padding Functions
3. Fractal (Self-Similarity)
   1. Hilbert curve
   2. Koch snowflake
   3. Sierpinski's Gasket
   4. Peano curve
4. Square limit variation
   解説:和田英一「関数画家」, Webにあり.
5. Circle limit

図形言語 Picture language とは

図形言語での学習目標

- 図形操作の抽象化
- 実際に図形言語で, フラクタル, 空間充填曲線が作成できる技術

図形言語の使い方（README.tustk）

```
% tustk2
> (load '/usr/local/lib/tustk/demos/picl)
> (load '/usr/local/lib/tustk/demos/picl-test)
> (make-canvas frm1)      ; 標準フレーム用キャンバスの作成
> ((square-limit wave 2)  frm1)
> (tk:update)             ; 上記実行で描画されなかった場合や
                          ; 再描画が必要なときに実行
> (clear)                 ; 描いた絵を消す
> ((square-limit wave 4) frm1)
> (clear)
> ((squash-inwards wave) frm1)
> (forget-canvas)         ; 現在のキャンバスを削除
> (make-canvas frm2)      ; 傾いたフレーム用キャンバスの作成
> ((square-limit wave 4) frm2)
> (output-canvas 'wave.ps) ; 絵をPostScriptファイルへ出力
```

図形言語の使い方（README.tustk）

- `*bg-color*` ; 背景色の定義
- `*line-color*` ; 描画線の定義
- `(set! *bg-color* <色>)` ; 色の設定を変更
- <色>；色名（blue, "red", ...）, RGB指定（"#RRGGBB"）
- 図形は painter で定義
  - 点 `(make-vect` *<x-coordinate>* *<y-coordinate>*`)`
  - 線（セグメント）`(make-segment` *<from-point>* *<to-point>*`)`
  - 線画 `(segments->painter` *<list-of-seguments>*`)`
  - 多角形（ポリゴン）`(vects->painter` *<list-of-points>*`)`
  - `(vects->painter` *<list-of-points>* *<smooth-or-not>* *<degree-of-smoothing>* *<filling-color>* `)`
  - `(pgm-file->painter` *<file-name>*`)` ; GIF/PPM/PGM

```
(define sicp (pgm-file->painter
              "/usr/local/lib/tustk/demos/sicp.ppm"))
((square-limit sicp 4) frm1)
```

8

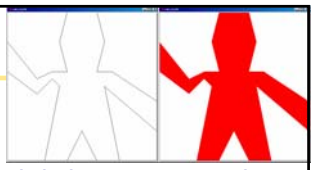
wave の定義

線を次々描いていく.

```
(define wave
  (segments->painter
    (list (make-segment (make-vect 0.25 0.00) (make-vect 0.35 0.50))
          (make-segment (make-vect 0.35 0.50) (make-vect 0.30 0.55))
          (make-segment (make-vect 0.30 0.55) (make-vect 0.20 0.45))
          (make-segment (make-vect 0.20 0.45) (make-vect 0.00 0.60))
          (make-segment (make-vect 0.00 0.80) (make-vect 0.20 0.55))
          (make-segment (make-vect 0.20 0.55) (make-vect 0.30 0.60))
          (make-segment (make-vect 0.30 0.60) (make-vect 0.40 0.60))
          (make-segment (make-vect 0.40 0.60) (make-vect 0.35 0.80))
          (make-segment (make-vect 0.35 0.80) (make-vect 0.40 1.00))
          (make-segment (make-vect 0.60 1.00) (make-vect 0.65 0.80))
          (make-segment (make-vect 0.65 0.80) (make-vect 0.60 0.60))
          (make-segment (make-vect 0.60 0.60) (make-vect 0.70 0.60))
          (make-segment (make-vect 0.70 0.60) (make-vect 1.00 0.40))
          (make-segment (make-vect 1.00 0.20) (make-vect 0.65 0.50))
          (make-segment (make-vect 0.65 0.50) (make-vect 0.75 0.00))
          (make-segment (make-vect 0.60 0.00) (make-vect 0.50 0.20))
          (make-segment (make-vect 0.50 0.20) (make-vect 0.40 0.00))
    )))
```

9

wave の変形



```
(define red-wave
  (vects->painter
    (list (make-vect 0.25 0.00) (make-vect 0.35 0.50)
          (make-vect 0.30 0.55) (make-vect 0.20 0.45)
          (make-vect 0.00 0.60) (make-vect 0.00 0.80)
          (make-vect 0.20 0.55) (make-vect 0.30 0.60)
          (make-vect 0.40 0.60) (make-vect 0.35 0.80)
          (make-vect 0.40 1.00) (make-vect 0.60 1.00)
          (make-vect 0.65 0.80) (make-vect 0.60 0.60)
          (make-vect 0.70 0.60) (make-vect 1.00 0.40)
          (make-vect 1.00 0.20) (make-vect 0.65 0.50)
          (make-vect 0.75 0.00) (make-vect 0.60 0.00)
          (make-vect 0.50 0.20) (make-vect 0.40 0.00) )
  #f 0 'red ))
```

点をつないで，ポリゴンを作成し，色を塗る.

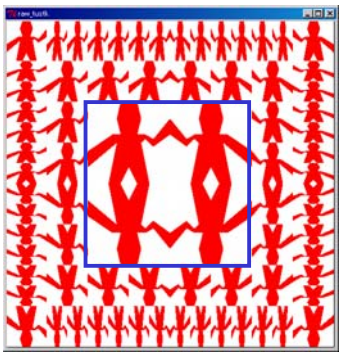
10

lambda の定義

```
(define red-letterlambda
  (vects->painter
    (list
      (make-vect .45 .60) (make-vect .25 .20)
      (make-vect .25 .20) (make-vect .20 .20)
      (make-vect .20 .20) (make-vect .20 .10)
      (make-vect .20 .10) (make-vect .30 .10)
      (make-vect .30 .10) (make-vect .50 .50)
      (make-vect .50 .50) (make-vect .70 .10)
      (make-vect .70 .10) (make-vect .80 .10)
      (make-vect .80 .10) (make-vect .80 .20)
      (make-vect .80 .20) (make-vect .75 .20)
      (make-vect .75 .20) (make-vect .40 .90)
      (make-vect .40 .90) (make-vect .30 .90)
      (make-vect .30 .90) (make-vect .30 .80)
      (make-vect .30 .80) (make-vect .35 .80)
      (make-vect .35 .80) (make-vect .45 .60) )
    #f 0 'red ))
```

12

square-limit *n*



これから

square-limit

を作る.

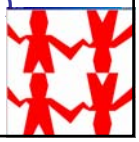
左はレベル2

まず，中央

14

flipped-pair（拡張版）

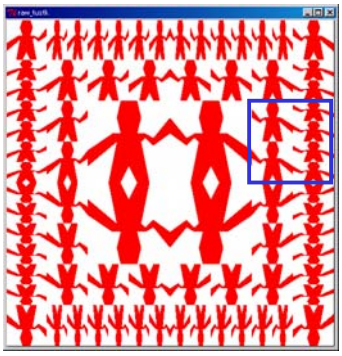
```
(define wave2
    (beside wave (flip-vert wave)) )
(define wave4 (below wave2 wave2))

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



こうすると

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

square-limit *n*



これから square-limit を作る.

左はレベル2

次は，右側

16

right-split *n*

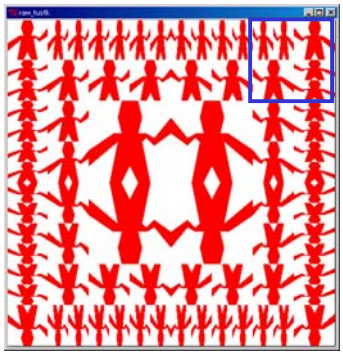
```
(define (right-split painter n)
   (if (= n 0)
       painter
       (let ((smaller
                (right-split painter (- n 1))) )
          (beside painter
                  (below smaller smaller) ))))
```

identity	right-split n-1
	right-split n-1

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

17

square-limit *n*



これから square-limit を作る.

左はレベル2

今度は，角

20

corner-split *n*

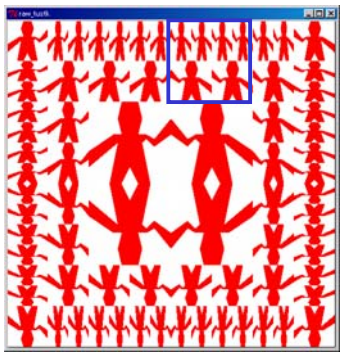
```
(define (corner-split painter n)
  (if (= n 0)
      painter
      (let ((up (up-split painter (- n 1)))
            (right (right-split painter (- n 1))) )
        (let ((top-left (beside up up))
              (bottom-right (below right right))
              (corner (corner-split painter (- n 1)))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

up-split n-1	up-split n-1	corner-split n-1
identity		right-split n-1
		right-split n-1

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

21

square-limit *n*



これから square-limit を作る.

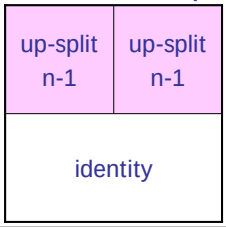
左はレベル2

最後は，上

24

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) ))))
```

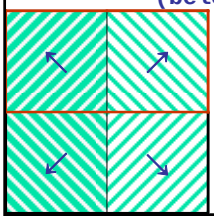


未定義の手続き
(below bottom top)
(beside left right)

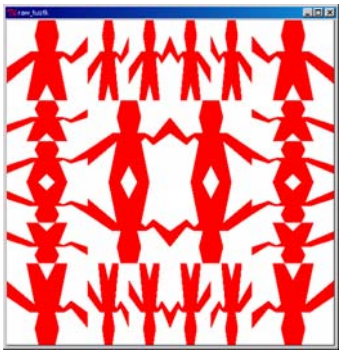
25

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))))
```



square-limit *n*

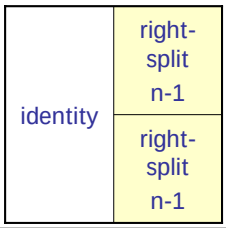


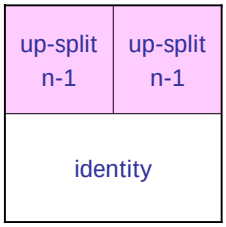
これで
square-limit
が完成.
左はレベル1

28

split群を抽象化

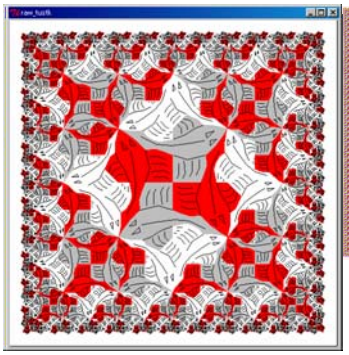
```
(define right-split (split beside below))
(define up-split (split below beside))
(define (split op1 op2)
   (op1 half (op2 quarter quarter)) ))
```





29

Escher's square-limit



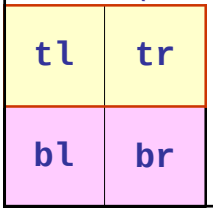
和田：関数画家、『情報処理』、Vol.46, No.10 (2005) 1163-1171

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

30

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) )))
```



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

34

flipped-pairsの別の定義

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

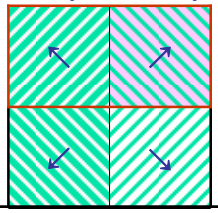
identity	flip-vert
identity	flip-vert

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

35

square-limit の別の定義

```
(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

36

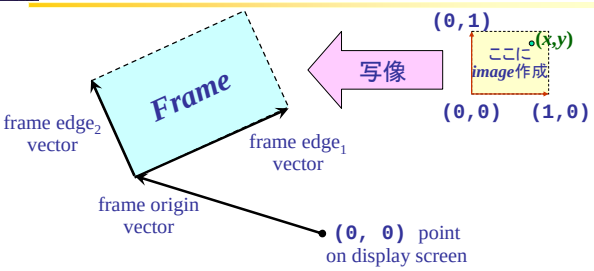
12月12日・本日のメニュー

今日は必修課題の説明です.

1. 2.2.4 Picture Language(図形言語)
   具体的な描画手続きを定義
2. Space Padding Functions
3. Fractal (Self-Similarity)
   1. Hilbert curve
   2. Koch snowflake
   3. Sierpinski's Gasket
   4. Peano curve
4. Square limit variation
   解説:和田英一「関数画家」, Webにあり.
5. Circle limit

37

Frame coordinate map



- coordinate: unit squareに作成
- frameに写像

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

38

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)`

39

Frames

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

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

それぞれに対する選択子を書け.

40

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 )))
```

(foreach *<procedure>* *<list-of-items>*)

42

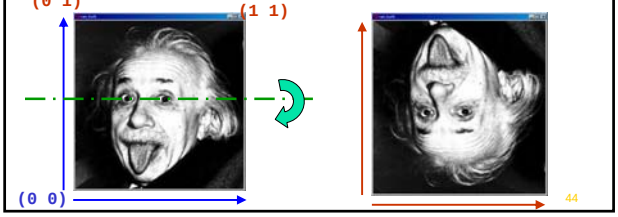
Transforming and combining painters

```
(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)))))))
```

43

Transforming and combining painters

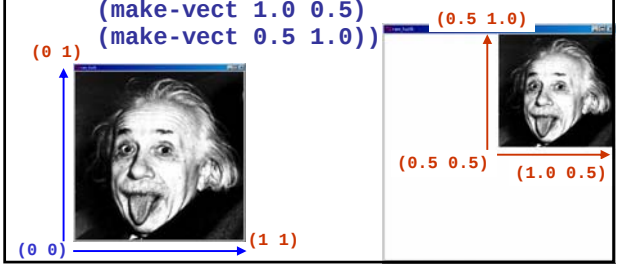
```
(define (flip-vert painter)
  (transform-painter
     painter
     (make-vect 0.0 1.0) ; new origin
     (make-vect 1.0 1.0) ; new end of edge1
     (make-vect 0.0 0.0))) ; new end of edge2
```



44

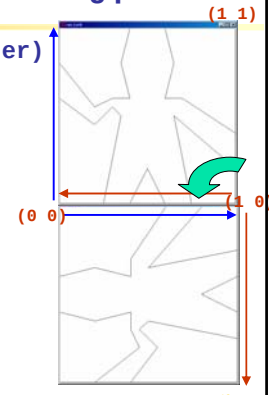
Transforming and combining painters

```
(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

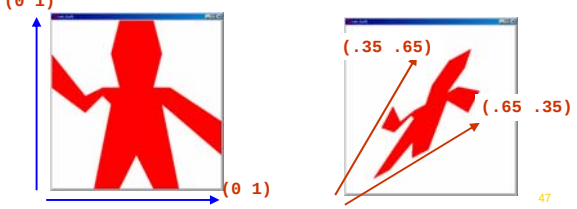
```
(define (rotate90 painter)
  (transform-painter
    painter
    (make-vect 1.0 0.0)
    (make-vect 1.0 1.0)
    (make-vect 0.0 0.0)
  ))
```



46

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)))
```



47

傾いたフレーム

```
(define frm2
    (make-frame
      (make-vect 0 0)
      (make-vect 1.0 0)
      (make-vect 0.5 0.5) )
```

48

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)))))
```

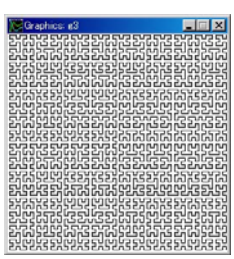
49

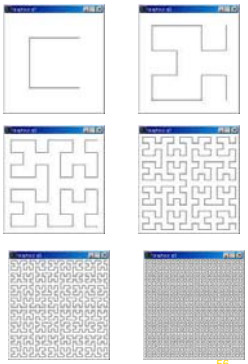
above m:n で分割

```
(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)))))
```

50

Hilbert curve の作成方法





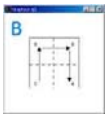
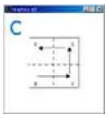
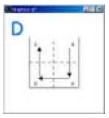
```
(hilbert 5)
```

56

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

58

Hilbert curve の手続き

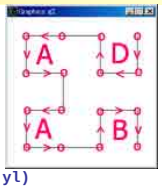
1. 各基本形に対して、レベル0ならコ型を書くための頂点のリストを求める。
2. さもなければ、分解形を再帰的に呼び出し、頂点を求める。
3. 求まった頂点リストから segment を求め `painter` を `vectors->segment` と `segments->painter`を使って作成する。

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

59

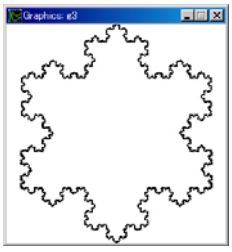
Hilbert curve の手続き

```
(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))) )
```

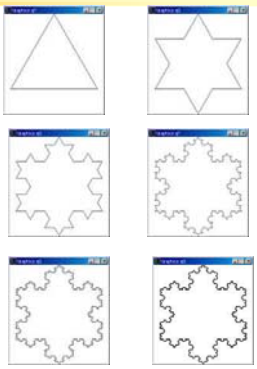


60

Koch snowflake の作成方法



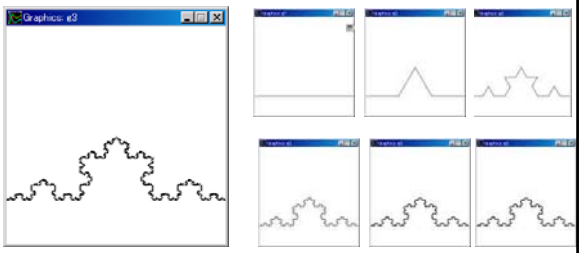
(koch 5)



61

Koch snowflake の作成方法

線分の分解



63

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>)
```

64

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))
        ))))
```

65

Koch snowflake の手続き（続）

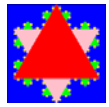
```
(define (koch n)
  (let* ((h (/ 0.75 0.86))
         (x0 (/ (- 1.0 h) 2))
         (x1 (- 1.0 x0)) )
    (segments->painter
      (vectors->segments
        (append
          (koch-line x0 0.25 x1 0.25 1 n)
          (koch-line x1 0.25 0.5 1.0 1 n)
          (koch-line 0.5 1.0 x0 0.25 1 n)
     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

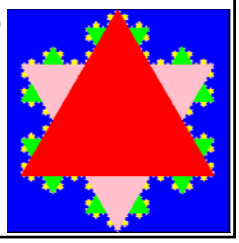
let* は let と違い、変数値対を順番に評価

66

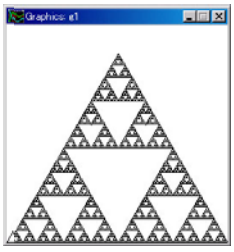
色つきKoch curve の手続き

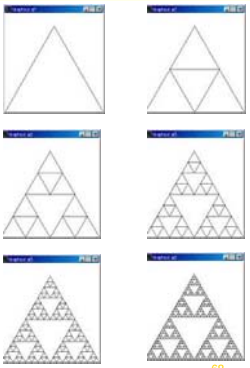
```
(define (koch-fill n . args)
  (let* ((h (/ 0.75 0.86))
         (x0 (/ (- 1.0 h) 2))
         (x1 (- 1.0 x0))
         (color (if (null? args) 'red (car args))) )
    (vects->painter
      (append (koch-line x0 0.25 x1 0.25 1 n)
        (koch-line x1 0.25 0.5 1.0 1 n)
        (koch-line 0.5 1.0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

68

Sierpinski's Gasket の手続き

```
(define (gasket 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 (gasket x0 y0 xm y0 (- i 1))
               (gasket xm y0 x1 y0 (- i 1))
               (list (make-vect x0 y0))
               (gasket xs ys xl ys (- i 1))
               (list (make-vect x0 y0) )))))

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

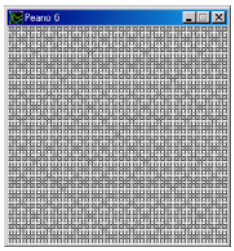
69

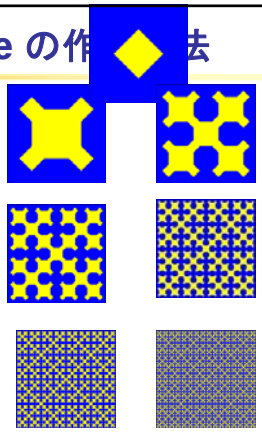
色つきSierpinski's Gasket 手続き

```
(define (gasket-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 (gasket-fill x0 y0 xm y0 (- i 1) color)
                (gasket-fill xm y0 x1 y0 (- i 1) color)
                (gasket-fill xs ys xl ys (- i 1) color) ))))

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

70

Peano Curve の作法



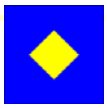


(peano 6)

71

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)) ))))
```



72

必修課題2：2月15日午後5時締切

1. 気の利いたpainterを1種類作れ.
2. それをsquare-limit等に適用.
3. 空間充填曲線を1種類作れ.
   (Hilbert curve, Peano curve, ...)
4. フラクタルを一種類作れ
   (Koch Snowflake, Sierpinsky's Gasket, ...)

- レポートは紙，またはファイルで.
- 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/
- 教えてもらった場合には，明記すること.
- 人と同じプログラム・作品（年度不問）は再提出.

73

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

宿題：12月18日午後5時締切

宿題は、次の計4問：

- **Ex. 2.37, 2.38, 2.39, 2.42**
- *Ex.2.44~52は*必修課題2-2*に包含*

- TAのレポート講評を見て復習すること.
- 教えてもらったときには明記すること.
- 人のレポートのコピーは厳禁（減点）

88