■ 連載

Haskell プログラミング

関数画家

和田 英一（IIJ 技術研究所）
wada@u-tokyo.ac.jp

入れ子構造の図と再帰プログラム

世の中には（計算機科学の分野には？）フラクタルのような入れ子の絵がよくある．一方，関数プログラムは再帰の専売特許みたいなものだから，Haskell でそういう絵が描きたくなるのもむべなるかなである．まただまし絵の M. C. Escher の絵にも入れ子構造のものがあり，プログラムしてみたくなる．

入れ子の絵といえば Hilbert 曲線が有名だ．そのほかにも Sierpinski 曲線，Koch 曲線，ドラゴン曲線などなどいろいろある．一方 Haskell ではないが，Scheme を使う Sussman たちの「計算機プログラムの構造と解釈」[1] に Henderson 流の図形言語の章があり，Escher の Square Limit を描くプログラムは面白かった．Paul Hudak の著した "The Haskell School of Expression" という Haskell 入門書はサブタイトルが Learning Functional Programming through Multimedia となっていて，たしかに絵がいろいろ描いてある（いささか意外な書名だが，序文によると Haskell Curry の両親 Samuel Silas Curry と Anna Baright Curry は 1879 年にボストン郊外 (Milton) に School of Elocution and Expression という学校を設立した．いまでは Curry College というそうだが，Expression は関数の基本ということで，この校名を借りたという）．

今回は描画に関連する関数プログラムの話をしたい．代表として Hilbert 曲線と Escher が 1964 年に発表した版画 Square Limit を選ぶ．これさえできればほかの図は演習問題にすぎない．

Hilbert 曲線

Hilbert 曲線は空間充填曲線の一種で，David Hilbert が 20 世紀の始め頃 Peano 曲線から思いついたものらしい（**図 -1** A）．

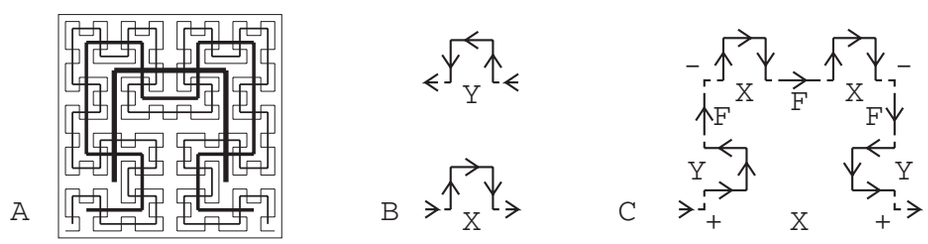

図-1

一番太い線の門構えのような形を 1 次の曲線とする．次の太さのΩ状のが 2 次で，よく見ると 1 次の曲線の端と角

のところにいろいろな向きで半分の大きさの 1 次の曲線があるのが分かる．同様にして 3 次の曲線の構造も理解できよう．ここには 4 次まで描いてある．

線の太さは無視し，外側の正方形の 1 辺の長さを単位とし，それぞれの次数の曲線の総延長を計算してみると，

次数	セグメント数	セグメント長	総延長
1	3	1/2	1.5
2	7	1/4	3.75
3	63	1/8	7.875
…			
n	$2^{2n}-1$	$1/2^n$	2^n-2^{-n}

であることが分かる．このように総延長が延びて空間を埋めていく．

ところで絵の書き方は図を見ながら考えても大体この辺に落ち着くが，物の本には Aristid Lindenmayer の L-system で説明するものが目につく．私は L-system をタートルグラフィクスを文脈自由文法で表したものと理解しているが，もっと深淵なものかもしれない．

L-system の Hilbert 曲線の生成規則は

```
X -> + Y F - X F X - F Y +
Y -> - X F + Y F Y + F X -
```

出発記号は X である．ここで + は移動方向を + の向きに 90 度回転する；- は − の向きに回転する；F は決まった長さだけ前進する指令だ．

なぜこれでよいかを図 -1 の B と C で考えよう．1 次の曲線を出発記号の X で描くのにタートル（かめ）は左から来て，上に凸の図を描いた後右へ去るものとする（**図 -1** B の下）．1 次だから下請けの X や Y は呼ばない．左から来て上を向きたいから + で左折（+ ターン），F で上向きの 1 画を描き，次の - で右折（− ターン），中央にある F で右向きの 1 画を描き再び右折，下向きの 1 画を描いて最後に左折し右を向く．

Y が逆に右から来て左回りに上に凸の 1 次の曲線を描き左へ去るのも容易に分かる（図 -1 B の上）．

2 次の X を見てみる（**図 -1** C）．左から来て左折，そこで右に倒れた 1 次の Y を描く．Y から出てくると上を向いているはずだから，F で上向きに 1 画を描く．次に左上の X を描くのだが，X はかめが左からくると期待しているから，- で右折しておく．そして X．右向きに出てくるからそのまま F X．まだ右向きだが，今度は下へ行きたいので右折．そして F で前進．Y を描いて下向きから右向きへと左折する．2 次より高い曲線も同様に描ける．

L-system ではこういう表現だが，当然下請けに入るための次数の制御が必要になる．とりあえずこれを確かめるべく，postscript で書くと（% から右はコメント）

```
/b 90 def                  % 回転角 b を 90 度と定義 (def)
/+ {/a a b add def} def    %+ 操作を定義　進行方向 a を b だけ増やす 反時計方向へ回転
/- {/a a b sub def} def    %- 操作を定義　進行方向 a を b だけ減らす 時計方向へ回転
/F {x y moveto /x x d a cos mul add def /y y d a sin mul add def
x y lineto} def            %x, y から x + d cos(a),y + d sin(a) まで移動

/X {n 0 gt                 %n>0 なら n-1 で X の定義を実行
{/n n 1 sub def + Y F - X F X - F Y + /n n 1 add def} if} def
/Y {n 0 gt                 %n>0 なら n-1 で Y の定義を実行
{/n n 1 sub def - X F + Y F Y + F X - /n n 1 add def} if} def
```

図 -1 A の Hilbert 曲線はこれをサブプログラムとして描いた．

Lindenmayer には 1980 年代に 1 度会った．遺伝学者 木村資生さんの話をしてくれたが，私は木村さんの名前を知っているだけなので，話は弾まなかった．その後間もなく Lindenmayer は他界した．

Haskell 版の Hilbert 曲線

このプログラムを以下のように Haskell に直した．Haskell には graphic library もあるらしいが，自分で自由になる postscript のプログラムを出力するようにした．つまり出力されるのは postscript のプログラムで，これをファイルにいったんためて実行するのでもよいが，直接に実行するにはプログラムの最後に起動するため `main = hilbert 256 5` と書いておき，

```
runhugs hilbert.hs | gv -
```

を実行する（gv は postscript の処理系）．`hilbert.hs` は以下の通り[☆1]．

```
import Numeric

left,right    :: (Int,Int) -> (Int,Int)
left  (dx,dy) = (-dy, dx)           -- 左折
right (dx,dy) = (dy, -dx)           -- 右折

f   :: (Int,Int) -> IO ()           -- F に対応する , rlineto は相対位置まで線を引く
f   (dx,dy) = putStrLn ((show dx) ++ " " ++ (show dy) ++  " rlineto")

x,y :: Int -> (Int,Int) -> IO ()    -- X,Y に対応
x 0 (_,_) = putStr ""               -- 0 次なら何もしない . 空を出力
x (n+1) (x0,y0) = do y n (x1,y1); f (x1,y1); x n (x0,y0); f (x0,y0)
                     x n (x0,y0); f (x3,y3); y n (x3,y3)
  where (x1,y1) = left (x0,y0); (x3,y3) = right (x0,y0)

y 0 (_,_) = putStr ""
y (n+1) (x0,y0) =do x n (x3,y3); f (x3,y3); y n (x0,y0); f (x0,y0)
                    y n (x0,y0); f (x1,y1); x n (x1,y1)
  where (x1,y1) = left (x0,y0); (x3,y3) = right (x0,y0)

hilbert :: Int -> Int -> IO ()      -- ドライバ
hilbert size n = do putStrLn ((show o) ++ " " ++ (show o) ++  " moveto")
                    x n (x0,y0)
                    putStrLn "stroke"
      where x0 = size `div` (2 ^ n) -- セグメント長 (= 移動距離 )
            y0 = 0
            o  = x0 `div` 2         -- 原点から出発位置までの距離

main = hilbert 256 5
```

簡単に説明すると最後の方，`hilbert` がドライバのプログラム．1 辺の長さ `size` と次数 `n` が引数．`x0` はセグメント長．正負が右と左に対応．`y0` は上下移動用．90 度の左右の回転でのセグメント長の変化はプログラムの上の方の `left, right` で定義する．

`f` は引数 `(dx,dy)` をとり，"dx dy rlineto" の文字列を出力する．`rlineto` の `r` は relative，つまり現在位置から `dx dy` だけ相対的に移動しながら線を引く命令だ．空白文字に注意．

`x`，`y` が L-system の `X`，`Y` に相当する．それぞれ次数と最初の向きを貰う．次数が 0 の場合は `if` 式ではなく，引数のパターンマッチで判定する．その場合は引数の値は使わないので，`(_,_)` と書く．何もしないわけだが，`IO ()` の型にするため空の文字列を書く．

[☆1] これらのプログラムは `http://www.sampou.org/haskell/ipsj/` から取ることができる．

もう一方は次数が (n + 1) の場合で，引数をこう書くと本体では (n - 1) の代りに n と書ける．下請けを何回も呼ぶとき便利だ．x は前述のように + y f - x f x - f y + と呼ぶ．その通りに書いてあるが，飛び込んで来たときの方向を x0, y0 とし，それに対し左向きを x1, y1，右向きを x3, y3 とする．気持が悪いという人もいるが，90 度と 270 度を 1 と 3 で表現している．式はこのようにセミコロン（;）で区切ってよい．

山下によるさらに関数プログラム的な解法はいつものプログラム掲載サイトを参照のこと．

Square Limit

Doug Hofstadter の "Gödel Escher Bach" を待つまでもなく，Escher の絵は Scientific American 誌の Martin Gardner のコラム，Mathematical Games などでいく度も紹介され，計算機科学者の関心を引きつけた．計算機さえあれば，こんなもの訳なく描けそうだとつい思う絵もある．それを手作業で描いたところが凄い．

果たしてパソコンの登場とともに Escher の絵への挑戦は増えた．私も Macintosh で実験したりしたが，さらに複雑な Squre Limit に挑戦したのが Peter Henderson である．Square Limit は 4 角形だが，Escher には円形を元にした Circle Limit が年代的にはずっと前に（1958 年前後に）I から IV まで 4 つある [2]．

Henderson は Square Limit の構造を調べ，そのためのプログラム Functional Geometry を考えた．最初の論文は 1982 年で，2002 年に改良版を発表した [3]．

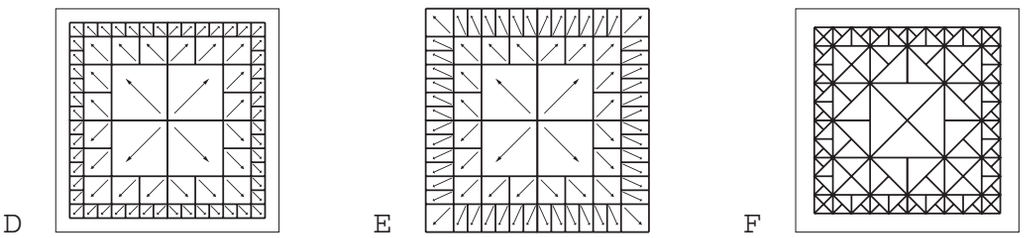

図-2

Square Limit は構造的には**図 -2** のようになっている．外方に向かい，同様な図が小さくなりながら並ぶ．D は Henderson の 1982 年の論文のもの，E は Sussman の教科書のもの，F は Henderson の 2002 年の論文の Fish 用のものである．私も持っているが Escher の作品集 [2] には，Square Limit の study として F と類似の図も掲載されている．Henderson はその図も参考にし，新方式を考えたらしい．

この面白さはペインタという関数が活躍することにある．それぞれの絵に対応するペインタは，それらの絵を描き出す場所 フレームを受け取るとそこに絵を描く関数として構成するが，そういうペインタを横に並べたペインタや回転するペインタをフレームとは無関係に次々と構築する．そのように構築した最終的なペインタにフレームを与えると，そのフレームに絵が描けるのである．

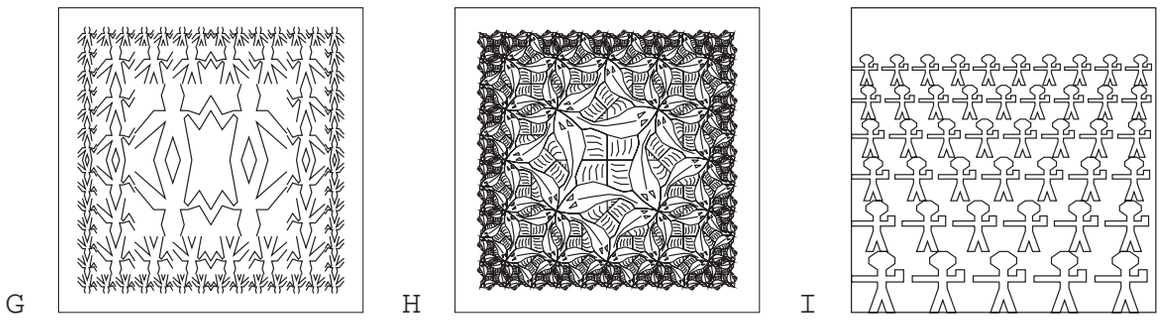

図-3

以下は**図 -3** の G を描くためのものだ．Sussman の教科書のを Haskell 風に書き直しだけだが，縁の構造を図 -2 の E から D に変えてある．wave は人がたのデータ．H と I は後述する．

```
import Numeric
type Vect = (Float, Float)          -- ベクタ型
type LineSegment = [Vect]           -- 線分
type Figure = [LineSegment]         -- 図形データは線分のリスト
type Frame = (Vect,Vect,Vect)       -- 最後の絵を描くフレーム
type Painter = (Frame -> IO ())     -- ペインタはフレームを貰い出力する
blan,wave :: Painter
blank =  \ frame -> putStr ""       -- 何も描かないペインタ

wave = segmentsToPainter 20 20      -- 図 -3 G の人がた
    [[(0,13),(3,8),(6,12),(7,11),(5,0)],[(8,0),(10,6),(12,0)],
     [(15,0),(12,10),(20,3)],[(20,7),(15,13),(12,13),(13,17),(12,20)],
     [(8,20),(7,17),(8,13),(6,13),(3,12),(0,17)]]

infixr 7 +~, -~                     -- 2 項演算子 +~,-~
infixr 8 *~                         -- 2 項演算子 *~
(+~), (-~) :: Vect -> Vect -> Vect
(x0,y0) +~ (x1,y1) = (x0+x1, y0+y1) -- ベクタ (x0,y0) に (x1,y1) を足す
(x0,y0) -~ (x1,y1) = (x0-x1, y0-y1) -- ベクタ (x0,y0) から (x1,y1) を引く
(*~) :: Float -> Vect ->Vect
a *~ (x,y) = (a*x, a*y)             -- ベクタ (x0,y0) を a 倍する

drawLine,drawLine' :: [Vect] -> String      --[(x,y),...] を結ぶ線を引く文字列を作る
drawLine ((x,y):xys) = show x ++ " " ++ show y ++ " moveto\n" ++ drawLine' xys
drawLine' [] = ""
drawLine' ((x,y):xys) = show x ++ " " ++ show y ++ " lineto\n" ++ drawLine' xys

segmentsToPainter :: Float -> Float -> Figure -> Painter
segmentsToPainter scale0 scale1 segs =      -- 線分の位置を正規化しペインタに
 \ frame -> putStr \$
            let toFrame (x,y) = frameCoodMap frame (x/scale0, y/scale1)
                drawSeg seg = drawLine (map toFrame seg)
             in
              concat (map drawSeg segs) ++ "stroke\n"

frameCoodMap :: Frame -> (Vect -> Vect)
frameCoodMap (org,edge0,edge1) =            -- フレームの枠内へ座標変換
 \ (x,y) -> org +~ x *~ edge0 +~ y *~ edge1

transformPainter :: Painter -> Vect -> Vect -> Vect -> Painter
transformPainter painter org edge0 edge1 =
 \ frame ->
    let m = frameCoodMap frame
        newOrg = m org
        newEdge0 = m edge0 -~ newOrg
        newEdge1 = m edge1 -~ newOrg
      in painter (newOrg, newEdge0, newEdge1)

rot,flipHoriz,flipVert :: Painter -> Painter
rot p = transformPainter p (1,0) (1,1) (0,0)        -- 左回転
flipHoriz p = transformPainter p (1,0) (0,0) (1,1)  -- 左右反転
flipVert p = transformPainter p (0,1) (1,1) (0,0)   -- 上下反転

above,beside :: Float -> Float -> Painter -> Painter -> Painter
```

```
above m n p q =                        -- p を q の上に m:n の高さで積む
 \ frame -> do transformPainter p (0,r) (1,r) (0,1) frame
               transformPainter q (0,0) (1,0) (0,r) frame
      where r = n/(m+n)
beside m n p q =                       -- p と q を m:n の幅で並べる
 \ frame -> do transformPainter p (0,0) (r,0) (0,1) frame
               transformPainter q (r,0) (1,0) (r,1) frame
      where r = m/(m+n)

infixr 3 </>                           -- 2 項演算子 </>
infixr 4 <->                           -- 2 項演算子 <->
(</>), (<->) :: Painter -> Painter -> Painter
(</>) = above 1 1                      -- p </> q  p を q の上に積む
(<->) = beside 1 1                     -- p <-> q  p と q を並べる

unitSquare :: Frame
unitSquare = ((128,16),(256,0),(0,256))      -- 絵を描くフレーム
```

型定義のあと，blank はフレームとして何を貰っても何もしないペインタである（空文字列を描く）．それ以外のペインタはたとえば図 -1 の 1 次と 2 次の Hilbert 曲線を描くペインタなら，左下 (0, 0)，右上 (w, h) を対角とする正方形内に描くつもりで座標のリスト（LineSegment）のリストを作り，それを (0, 0), (1, 1) 内に正規化するため，w, h とともに segmentsToPainter に渡す．

```
hilbert = segmentsToPainter 8 8
    [[(2,2),(2,6),(6,6),(6,2)],[(1,1),(3,1),(3,3),(1,3),(1,7),(3,7),(3,5),
      (5,5),(5,7),(7,7),(7,3),(5,3),(5,1),(7,1)]]
```

その先を見るため，(0, 0), (1, 1) に対角線を引く例を使おう．図形は `[[(0,0),(1,1)]]` で segmentsToPainter はこれをペインタに変える．フレームは最後に渡されるので，それを引数の関数 "\ frame -> putStr 文字列 " の形にする．この文字列は $ 以下で作られ，結局 "frameCoodMap frame (0,0) moveto frameCoodMap frame (1,1) lineto" となる．フレームは最後まで未知である．

frameCoodMap はフレームと座標を貰い，座標をフレームの枠にマップする．フレームは枠の (0, 0) に対応する位置 org と (1, 0), (0, 1) にそれぞれ対応する edge0, edge1 を貰い，新しい座標を返し，それに moveto や lineto を連結して postscript の命令にする．

対角線を描くペインタ p を左右逆転してみる．flipHoriz p とやるわけで，transformPainter が登場する．これはペインタと (0, 0), (1, 0), (0, 1) の各点がそれぞれ左右逆転で移動する先 (1, 0), (0, 0), (1, 1) をとり，フレームの座標がこの操作でどこへ移動するかを計算し，それをペインタへ渡す関数を構成する．なにしろ関数の形なので，いまどういう状態なのかは想像しなければならずややこしい．flipHoriz が分かれば，above や beside も理解が可能である．

cornerSplit から後は squareLimit を作るための道具である．この考え方を**図 -4** に示す．

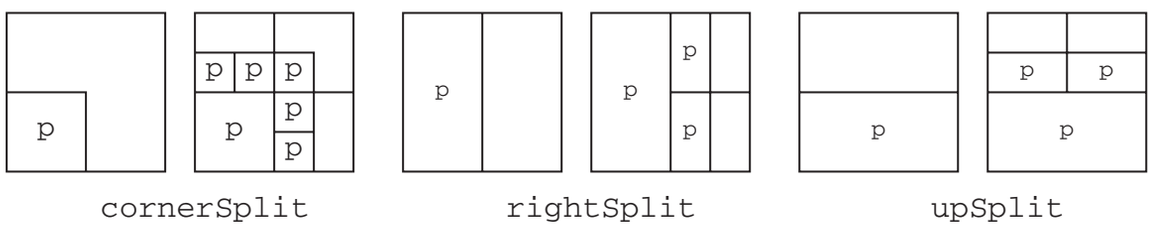

図-4

```
cornerSplit,rightSplit,upSplit,squareLimit:: Painter -> Int -> Painter
cornerSplit _ 0 = blank
cornerSplit p (n+1) = (topLeft </> p) <-> (corner </> bottomRight)
   where up = upSplit p n
         right = rightSplit p n
         topLeft = up <-> up
         bottomRight = right </> right
         corner = cornerSplit p n

rightSplit _ 0 = blank
rightSplit p (n+1) = p <-> (smaller </> smaller)
    where smaller = rightSplit p n

upSplit _ 0 = blank
upSplit p (n+1) = (smaller <-> smaller) </> p
    where smaller = upSplit p n

squareLimit p n = half </> (flipVert half)
   where half = (flipHoriz quarter) <-> quarter
         quarter = cornerSplit p n
main = squareLimit wave 3 unitSquare        -- 図 -3 の G
```

squareLimit は四隅に向かって対称なので，まず右上だけを構成し，最後に回転または反転しながら全体を作る．まず最後の squareLimit を見よう．最後の行の runhugs 用の main にあるように，squareLimit はあるペインタと次数を貰うとペインタが返り，それに unitSquare を与えるというのが使い方だ．定義は half を flipVert half の上に積むようになっている．2 項演算より単項演算が強いから，このかっこは実は不要だが，分かりやすいので使っている．ところで half は where 以下にあり，flipHoriz quarter と quarter を並べたものである．そして quarter は cornerSplit そのものだ．

図 -4 は 3 種類の Split について，左が最後に止まるとき ($n = 1$)，右は分岐中を示す．cornerSplit は $n = 1$ になると，2 番目の定義から左上が topLeft=up <-> up=upSplit p 0 <-> upSplit p 0=blank <-> blank，右下も同様で blank </> blank，右上が corner=cornerSplit p 0=blank で，結局 p が縦横それぞれ 1/2 になったものだけになり，停止する．そして G の図が描ける．最後に周囲に残った blank が図 -3 G の余白部分である．これは図 -2 D にある余白と同じである．

さかなの Square Limit

Henderson は 1982 年には**図 -5** の K の正方形を上下左右に 4 分した素材で Square Limit を構成したが，そもそも Escher は 1 匹のさかなから出発したに違いないとの反省で，2002 年には J の図をまず描いた．ここに掲げる図は必ずしも Henderson のデータと同じではないが，同様の精神で描いてある．周囲の太い線の黒丸で 4 分された各区間は相似形である．したがって斜辺中央の点を中心に 180 度回転した図とも合体できるし，K や L の図のようにも接続できる．下のプログラムで図形 fish は J の図を描く．対称性は最初の線分のデータから見て取れる．すなわち 2 行目の最後の方に (0, 0) があり，これはさかなの左のヒレの下，図の原点の黒丸に対応する．そこからデータを両方に読むと (−20, 20), (20, 20); (−16, 30), (30, 16);... のように続き，(0, 80), (80, 0) までが左上と右下の黒丸である．さらにデータを両方向に辿ると (5, 77) と (75, 3) が見えるがこれはそれぞれ足すと 80 になり，そういうデータがしばらく続く．先頭の (40, 60) に対応するのは最後から 3 つ目の (40, 20) で，その後の (32, 32), (40, 40) は太線から離れてさかなの輪郭を描いている．

J から K を作るため，角を中心に 45 度回転し，長さも $1/\sqrt{2}$ にする rot45 を定義した．また図を重ねることも必

要になり，over も用意した．

全体の構成は図 -2 の F に見る通りで，そのため 9 個の図をまとめる nonet も作った．beside や above が $m : n$ の比で分割できるように設計したのは実はこのためであった．K の図を 45 度回転すると破線が図 -2 F のパターンに一致する．完成図が図 -3 の H.

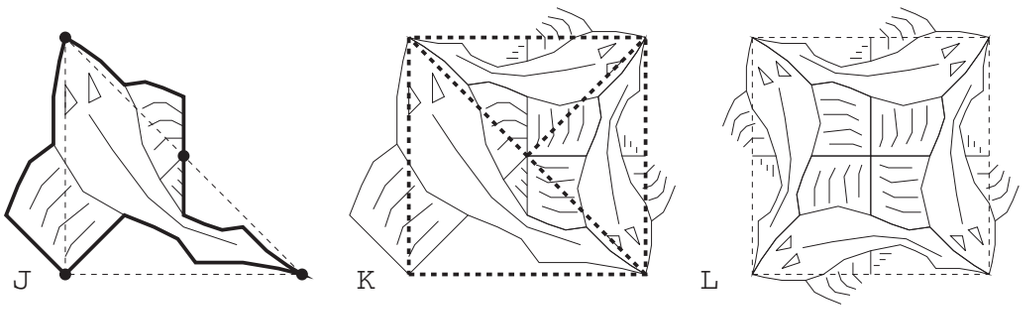

図-5

```
fish :: Painter
fish = segmentsToPainter 80 80
    [[(40,60),(33,63),(27,65),(20,64),(17,67),(12,72),(5,77),(0,80),(-2,72),
      (-4,60),(-4,50),(-4,44),(-12,38),(-16,30),(-20,20),(0,0),(20,20),(30,16),
      (38,12),(44,4),(50,4),(60,4),(72,2),(80,0),(75,3),(68,8),(63,13),(60,16),
      (53,15),(46,17),(40,20),(32,32),(40,40)],[(0,64),(0,54),(4,58),(0,64)],
     [(8,68),(8,58),(12,60),(8,68)],[(8,54),(16,42),(28,26),(40,16),(58,10)],
     [(-4,44),(6,28),(20,20)],[(-2,36),(-8,30),(-12,22)],
     [(2,30),(-6,22),(-8,16)],[(8,24),(-2,16),(-4,10)],[(10,18),(2,10),(0,6)],
     [(20,64),(24,56),(26,44),(32,32)],[(26,56),(30,58),(34,58),(40,54)],
     [(28,50),(32,52),(36,52),(40,50)],[(30,42),(34,46),(40,46)],
     [(38,36),(40,34)],[(38,32),(40,30)],[(36,30),(40,26)]]

rot45 :: Painter -> Painter                  -- 左上を中心に 45 度の回転
rot45 p = transformPainter p (0.5,0.5) (1,1) (0,1)
over :: Painter -> Painter -> Painter        -- 2 枚のペインタを重ねる
over p q = \ frame -> do transformPainter p (0,0) (1,0) (0,1) frame
                         transformPainter q (0,0) (1,0) (0,1) frame

fish2,u,t :: Painter
fish2 = flipHoriz (rot45 fish)
u = over (over fish2 (rot fish2))                    -- 図 -5 の L
         (over (rot(rot fish2)) (rot(rot(rot fish2))))
t = over fish (over fish2 (rot(rot(rot fish2))))     -- 図 -5 の K

nonet :: Painter -> Painter -> Painter -> Painter -> Painter ->
         Painter -> Painter -> Painter -> Painter -> Painter        --p q r
nonet p q r s t u v w x =                                           --s t u
  above 1 2 (beside 1 2 p (q <-> r))                                --v w x
            ((beside 1 2 s (t <-> u)) </> (beside 1 2 v (w <-> x)))

square, corner, side :: Int -> Painter
square n =
  nonet (corner n) (side n) (rot(rot(rot(corner n))))
        (rot(side n)) u (rot(rot(rot(side n))))
        (rot(corner n)) (rot(rot(side n))) (rot(rot(corner n)))

quartet :: Painter -> Painter -> Painter -> Painter -> Painter
quartet p q r s = (p <-> q) </> (r <-> s)
```

```
corner 0 = blank
corner (n+1) = quartet (corner n) (side n) (rot(side n)) u

side 0 = blank
side (n+1) = quartet (side n) (side n) (rot t) t

main = square 2 unitSquare          -- 図3のH
```

1970年頃のあるIFIPの会合の後，Hendersonは私をスポーツカーでWarwickからShakespeareの生地Stratford upon Avonまで送ってくれた．ゆるやかな丘陵を縫う田舎道を160km/hで飛ばすのには度肝を抜かされるも英国の田園の美しさを初めて堪能した．その後もレンタカーやバスで英国の地方を旅する機会はある．景観の美は相変わらぬが路傍を騎馬で行く少女や空地で洗濯物を干すジプシー家族は近年まったく見かけない．

Hendersonは Escherの本[2]を Les Beladyに貰ったと書いている．Beladyは昔日本IBMにいて数年前は三菱電機の米国ケンブリッジの研究所の会長を務めた親日家だ．ところでHendersonは図-3のIのような「関数プログラマの群れ(a crowd of functional programmers)」の絵を描いてごらんといっている．ここに絵があるくらいだから私はできている．みなさん，挑戦してください．解答は連載のプログラムのサイトに．

関数プログラマのデータは下記の通りである．

```
man :: Painter
man = segmentsToPainter 20 20
    [[(6,10),(0,10),(0,12),(6,12),(6,14),(4,16),(4,18),(6,20),(8,20),(10,18),
      (10,16),(8,14),(8,12),(10,12),(10,14),(12,14),(12,10),(8,10),(8,8),
      (10,0),(8,0),(7,4),(6,0),(4,0),(6,8),(6,10)]]
```


参考文献

1)ジェラルド・ジェイ・サスマン他: 計算機プログラムの構造と解釈第二版, ピアソン・エデュケーション(2000).

2) Locher, J. L. ed.: The World of M. C. Escher, Harry N. Abrams, Inc., Publishers (1971).

3) Henderson, P.: Functional Geometry, Higer Order and Symbolic Computation, Vol.15, pp.349-365 (2002). http://www.ecs.soton.ac.uk/~ph/papers/funcgeo2.pdf.

(平成17年8月2日受付)