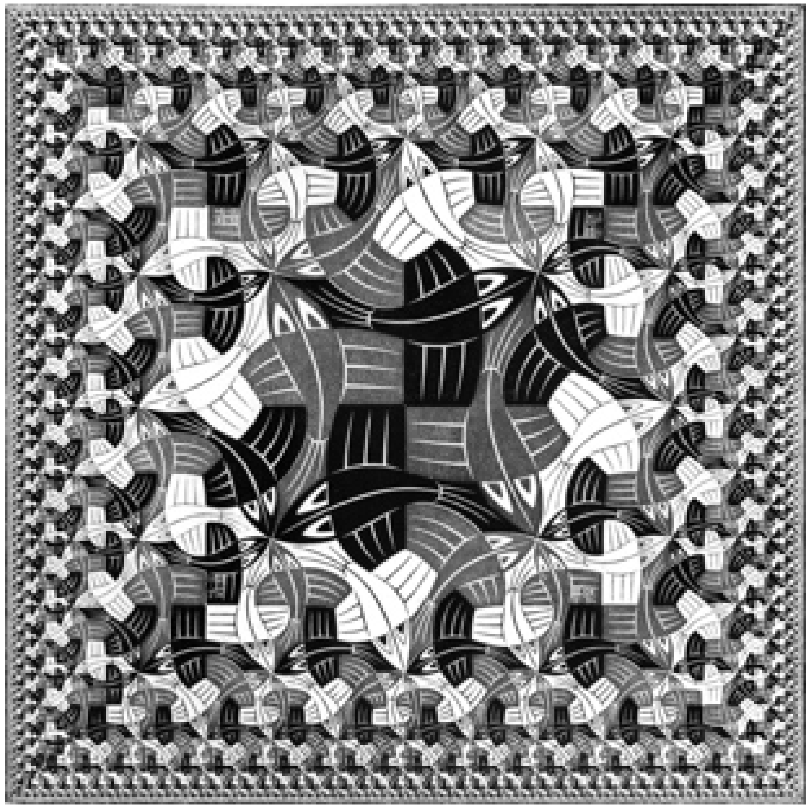
A PostScript to Functional Geometry

@einarwh



=>

A PostScript to Functional Geometry

Functional Geometry

Peter Henderson
Department of Electronics and Computer Science
University of Southampton
Southampton, SO17 1BJ, UK
p.henderson@ecs.soton.ac.uk
http://www.ecs.soton.ac.uk/˜ph

October, 2002

Abstract. An algebra of pictures is described that is sufficiently powerful to denote the structure of a well-known Escher woodcut, Square Limit. A decomposition of the picture that is reasonably faithful to Escher's original design is given. This illustrates how a suitably chosen algebraic specification can be both a clear description and a practical implementation method. It also allows us to address some of the criteria that make a good algebraic description.

Keywords: Functional programming, graphics, geometry, algebraic style, architecture, specification.

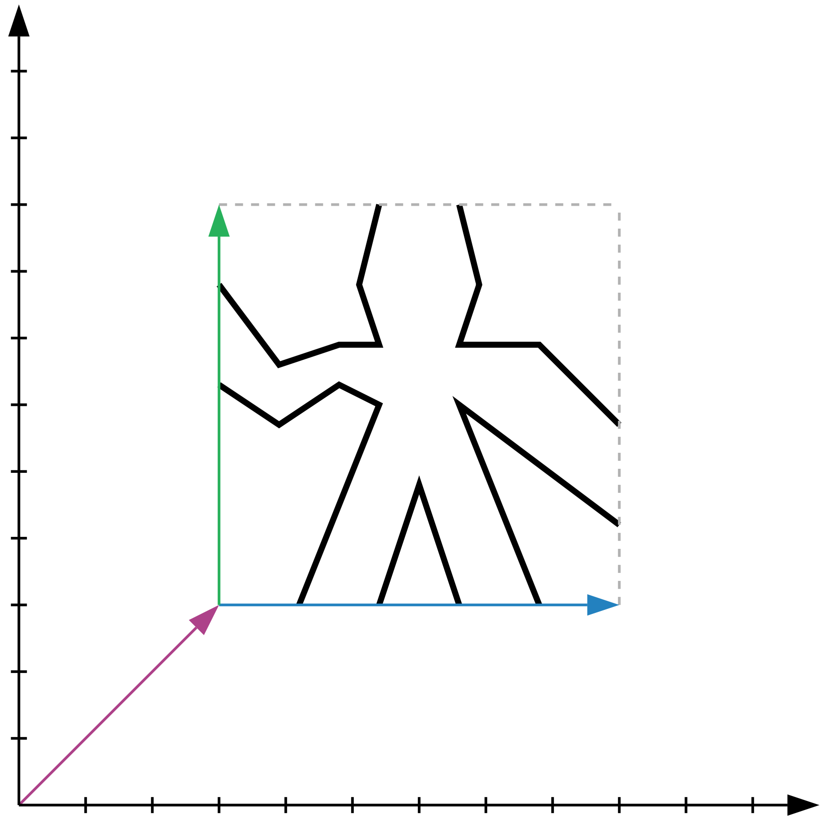
A picture is an example of a complex object that can be described in terms of its parts.

Let us define a picture as a function which takes three arguments, each being two-space vectors and returns a set of graphical objects to be rendered on the output device.

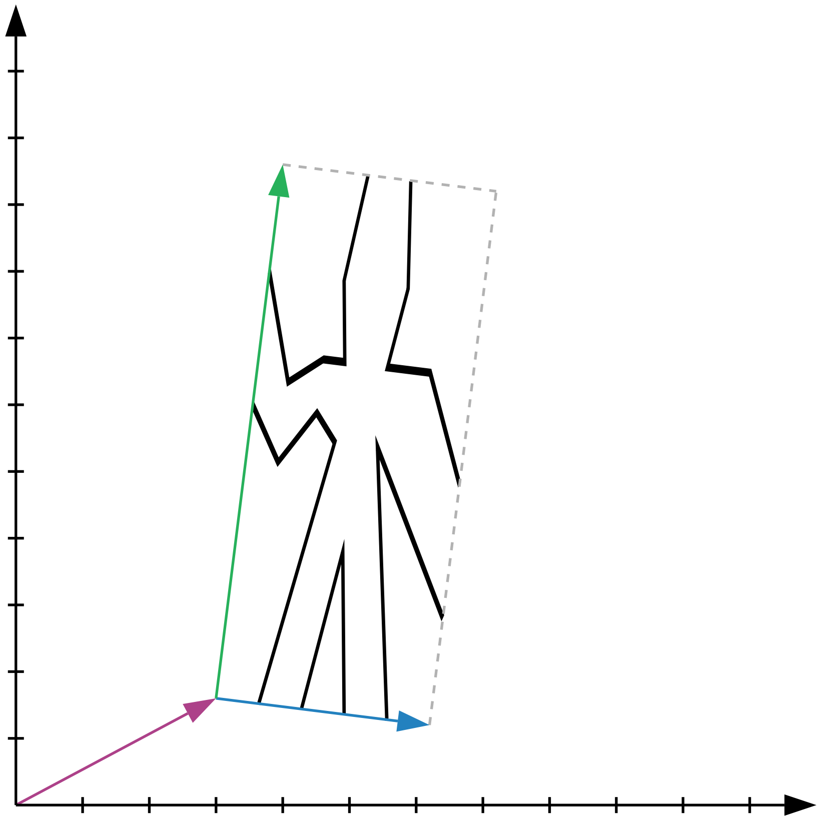
```
type Box = { a : Vector
             b : Vector
             c : Vector }

type Picture = Box -> Rendering
type Box = { a : Vector
             b : Vector
             c : Vector }

type Picture = Box -> Rendering
```

PSEUDOCODE

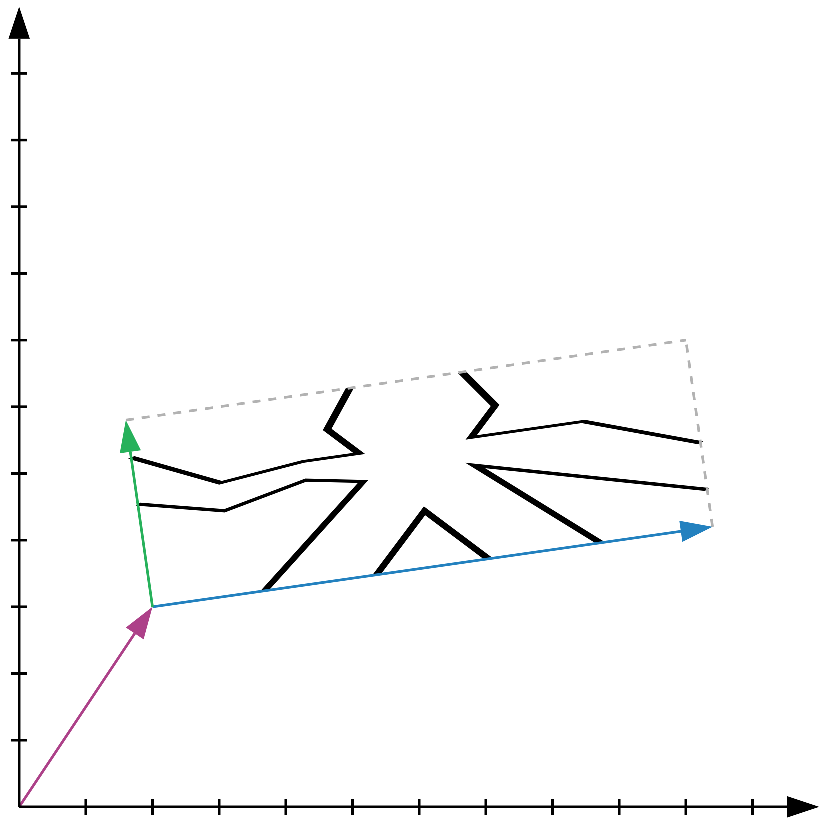
george



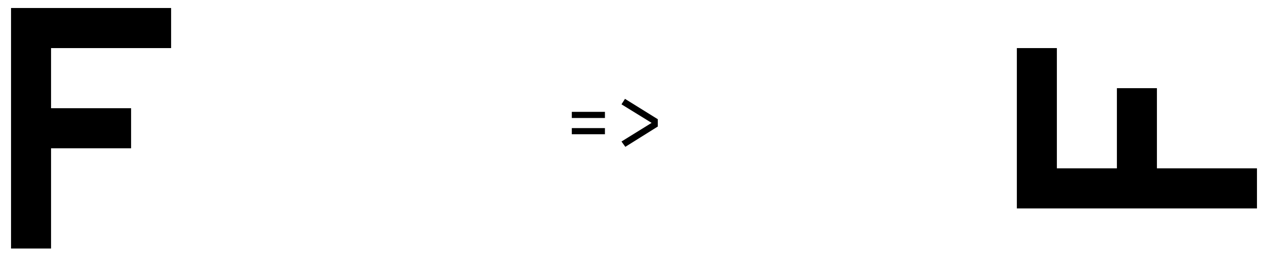
also george



still george

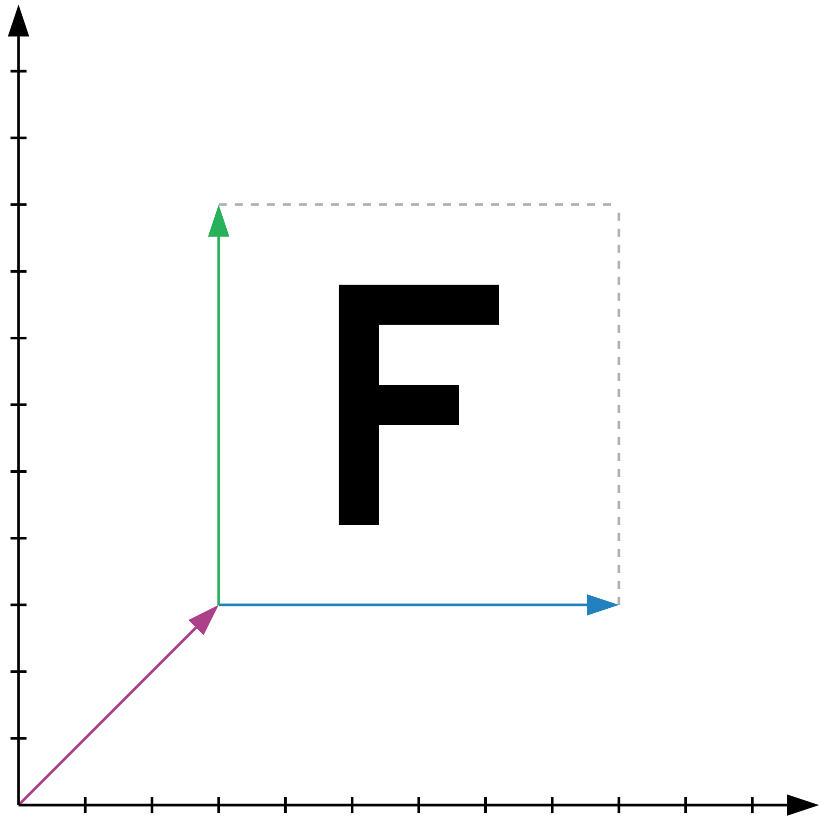


turn

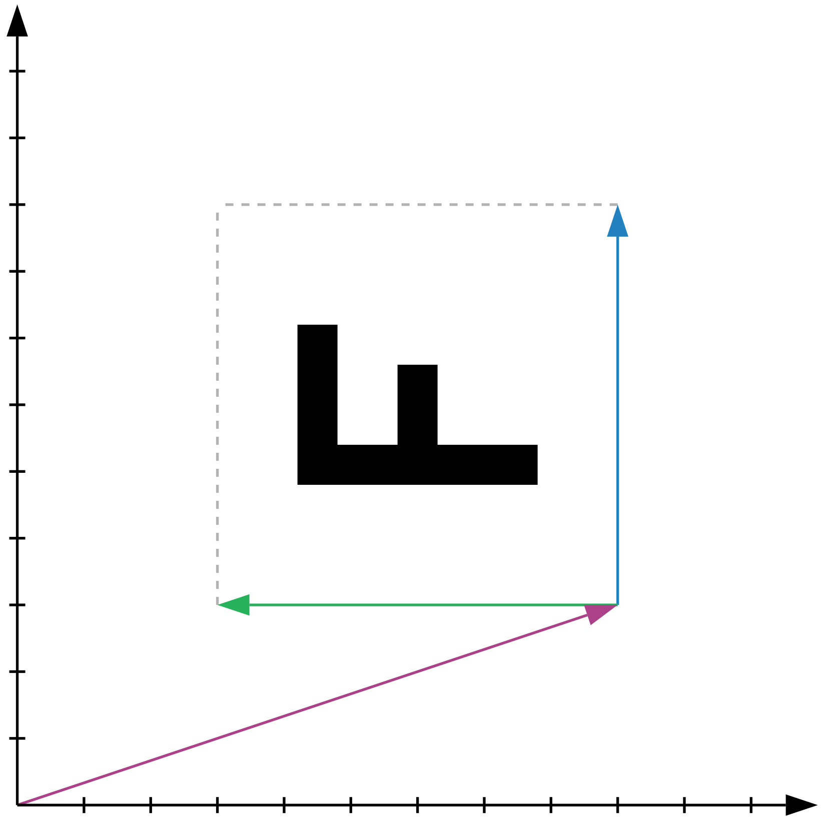


```
turnBox : Box -> Box
turnBox { a, b, c } = { a = add a b
                      , b = c
                      , c = neg b }

turn : Picture -> Picture
turn p = turnBox >> p
```

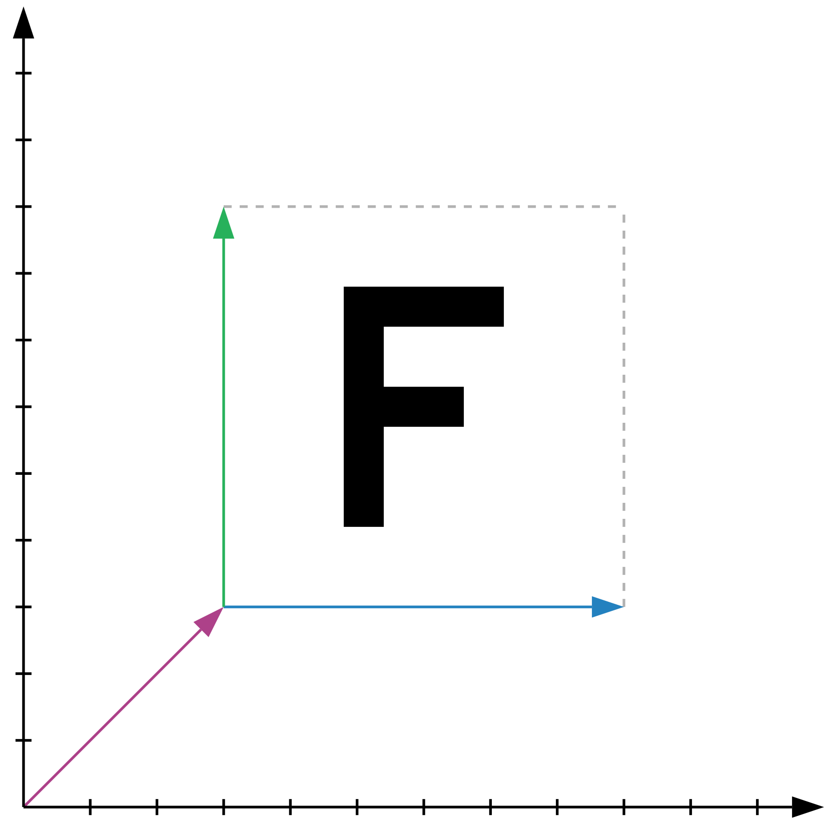
turn



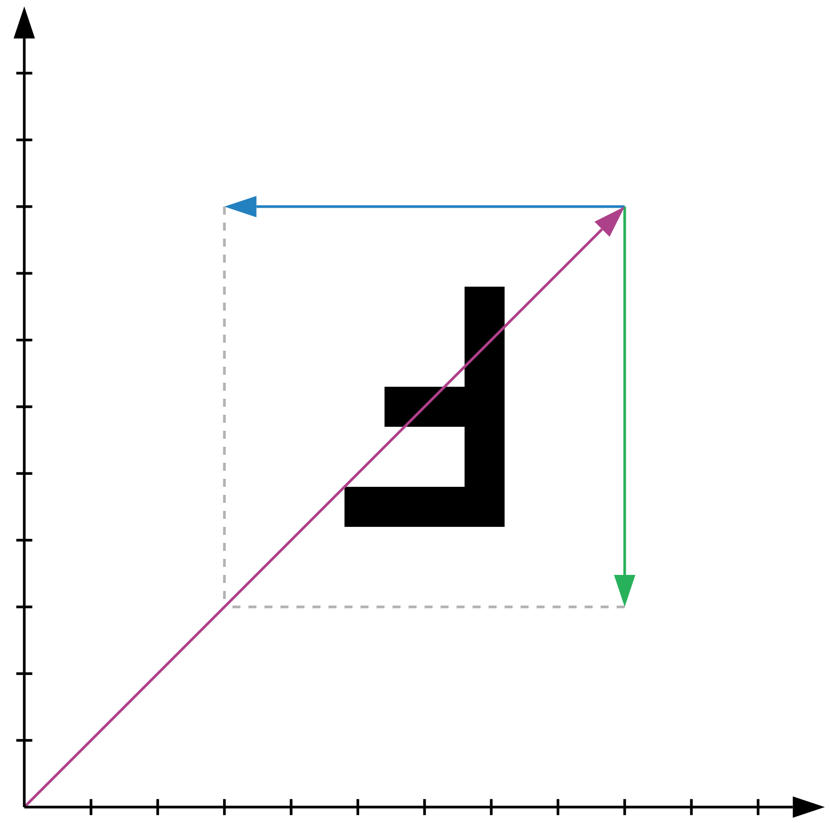
=>



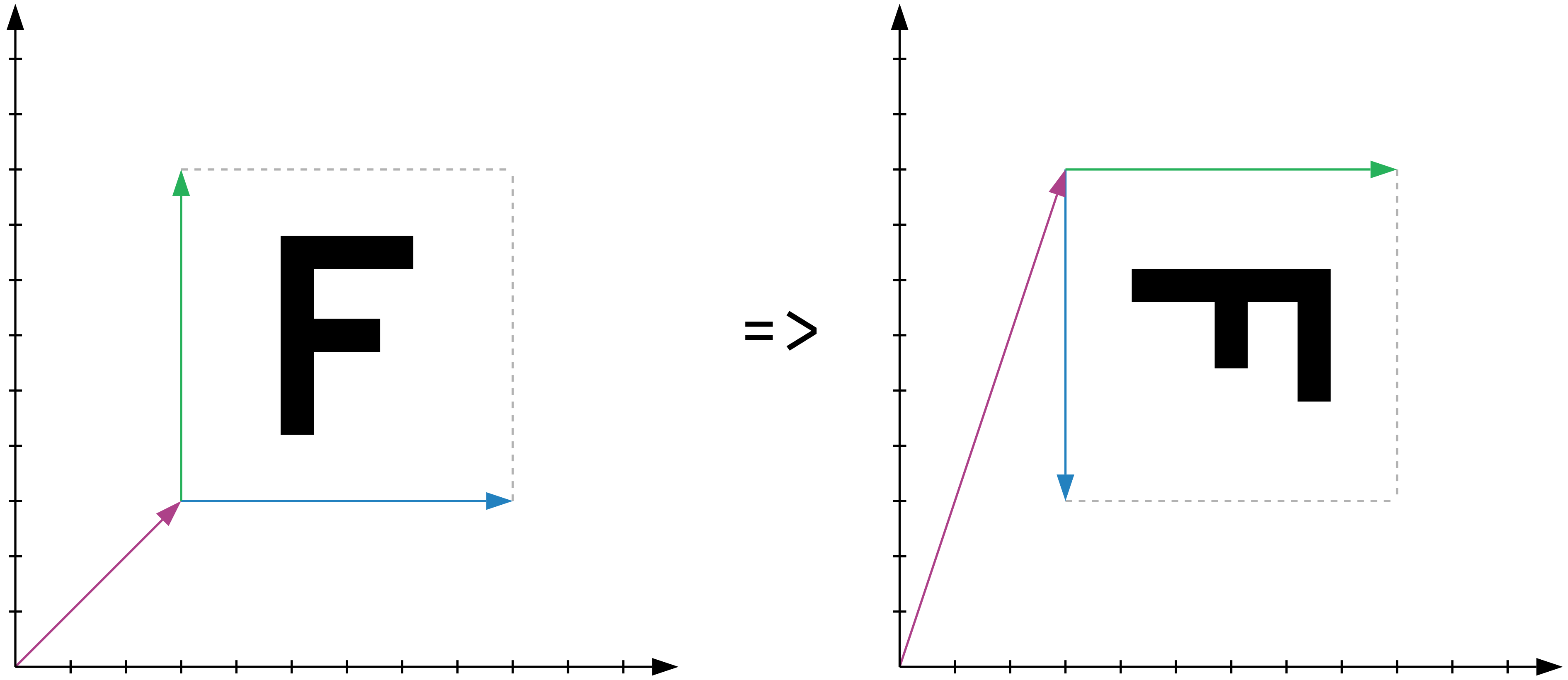
turn >> turn



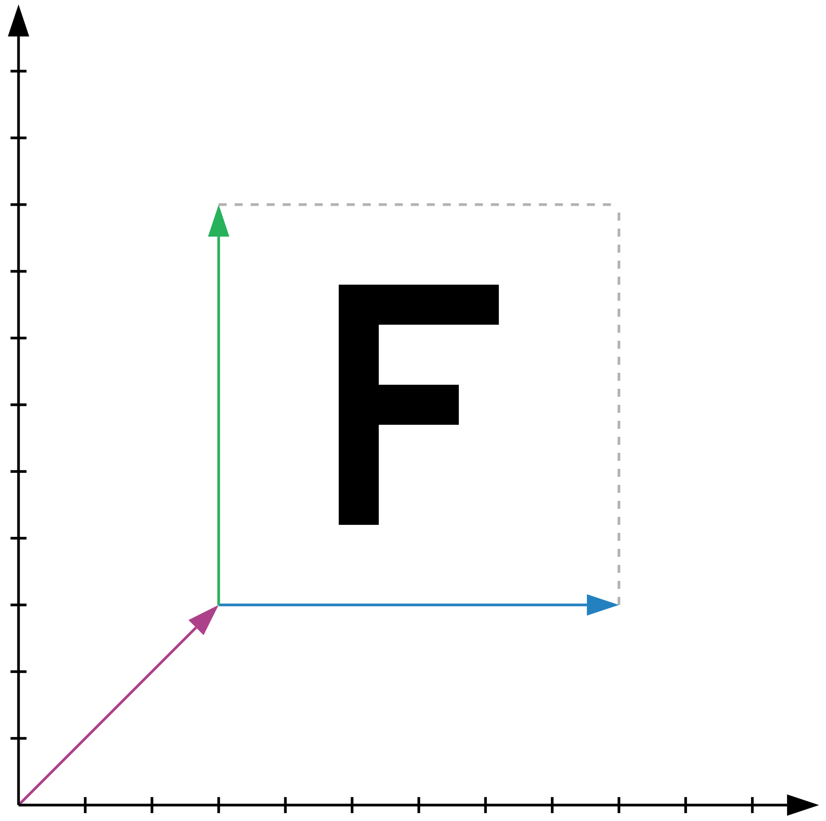
=>



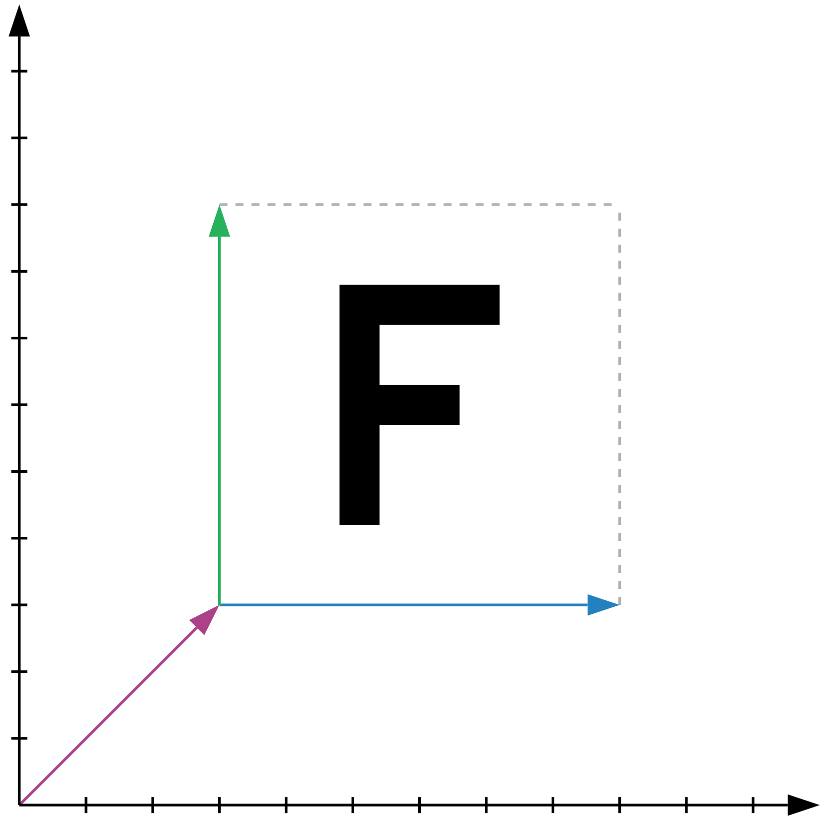
turn >> turn >> turn



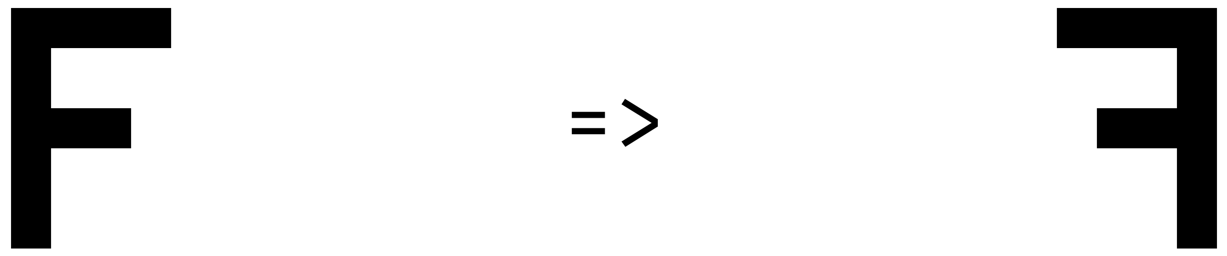
```
turn >> turn >> turn >> turn
```



=>

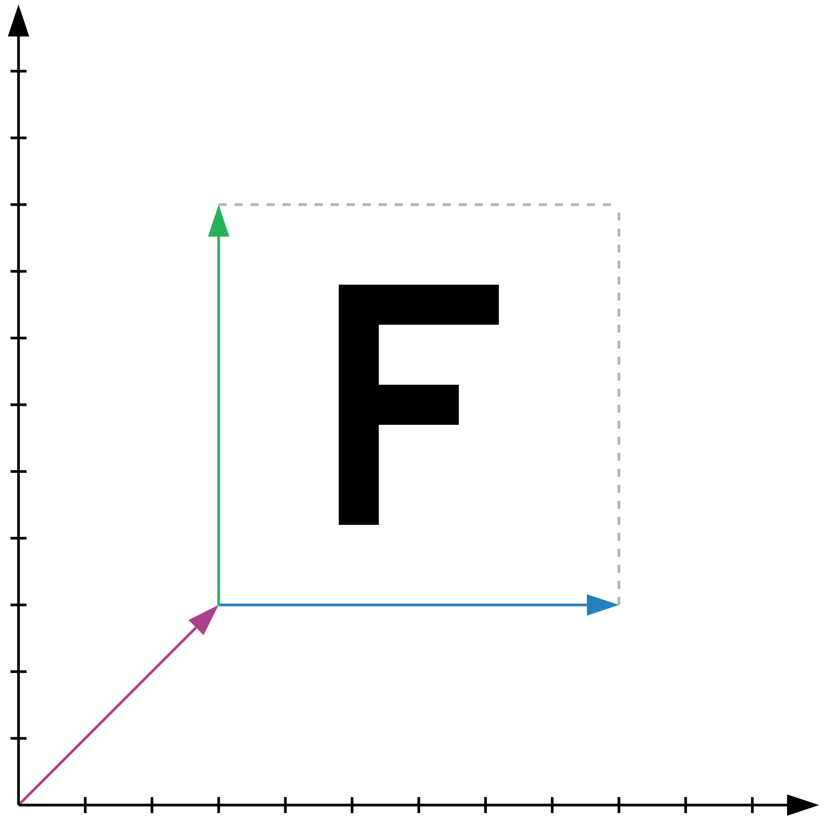


flip

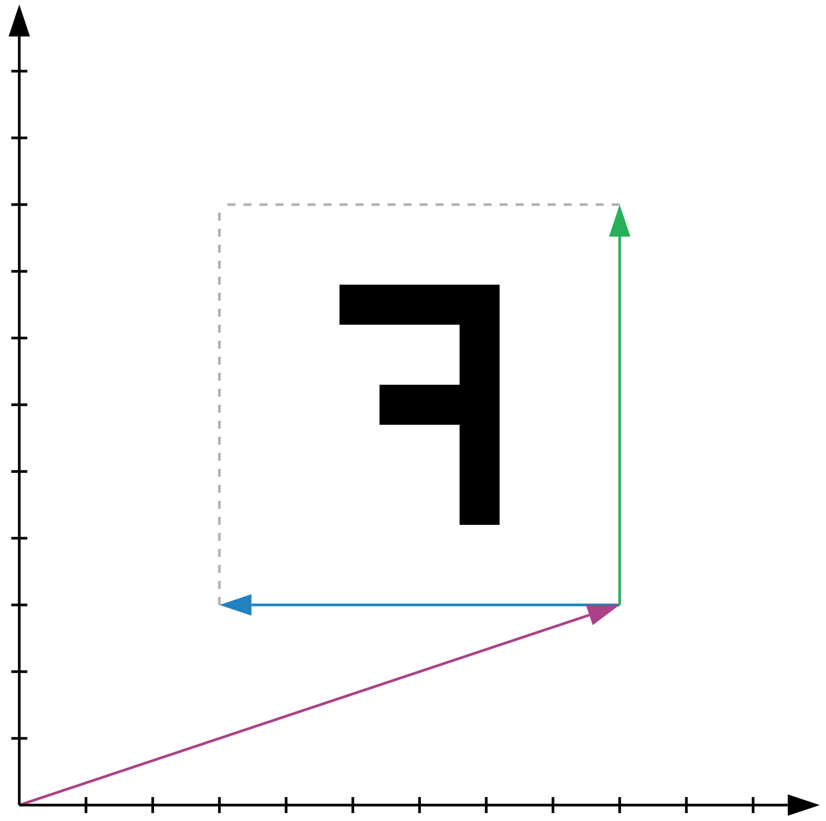


```
flipBox : Box -> Box
flipBox { a, b, c } = { a = add a b
                      , b = neg b
                      , c = c }

flip : Picture -> Picture
flip p = flipBox >> p
```

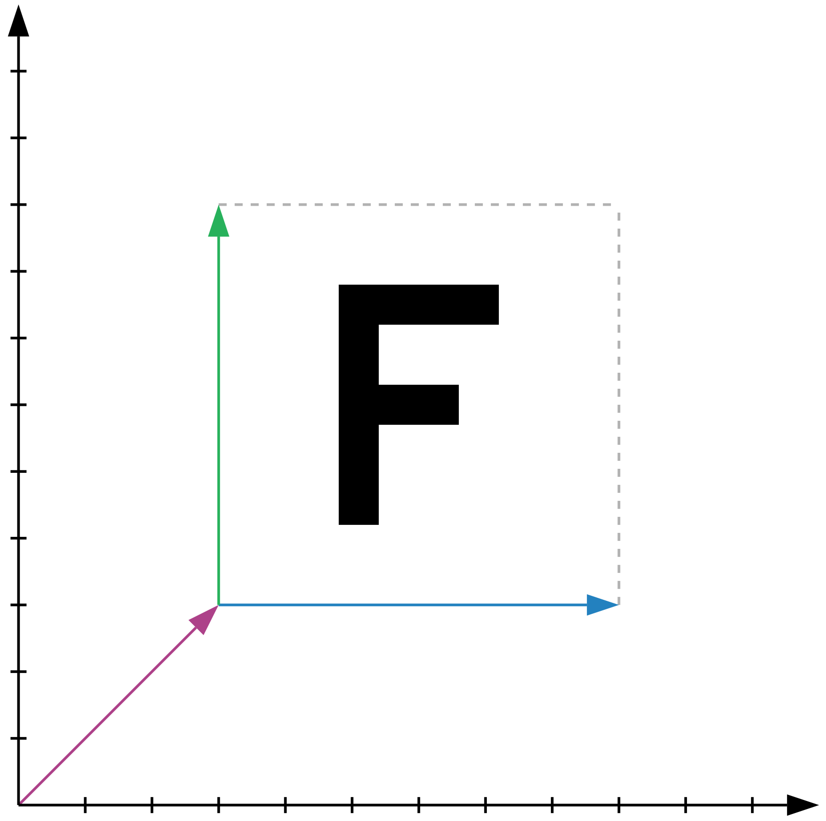
flip



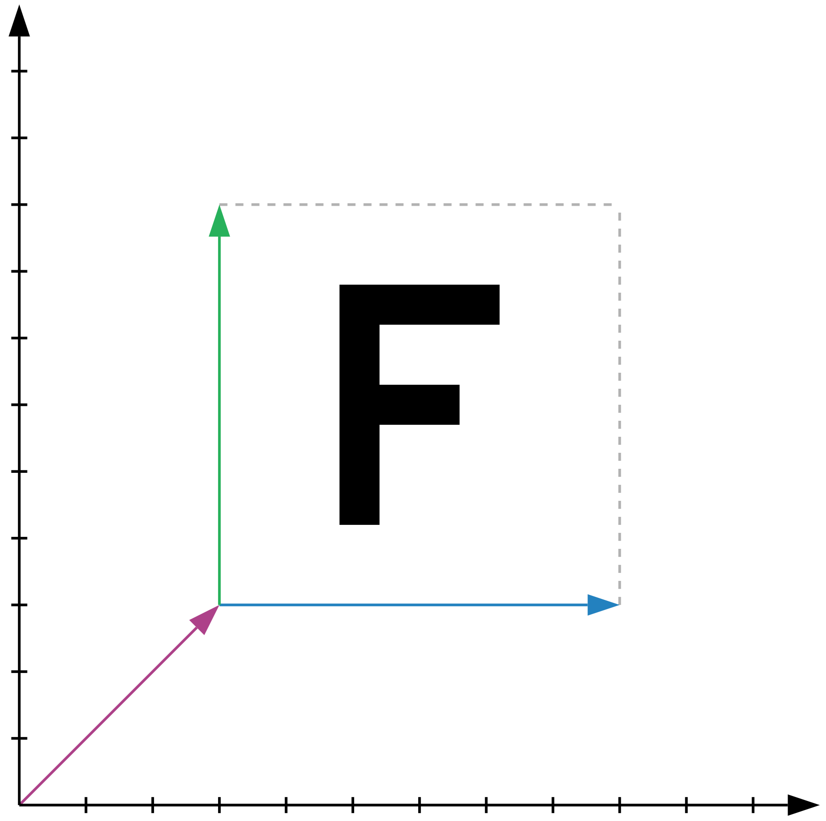
=>



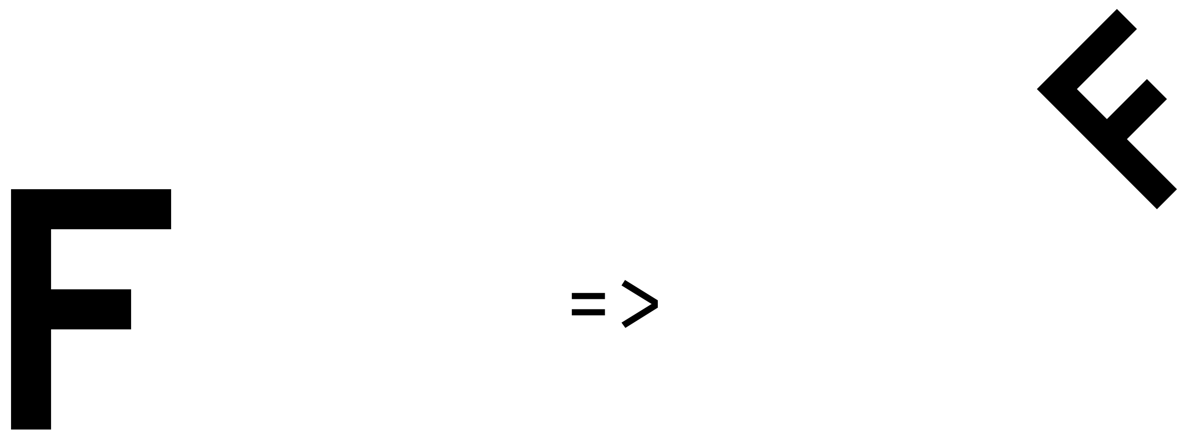
`flip >> flip`



=>

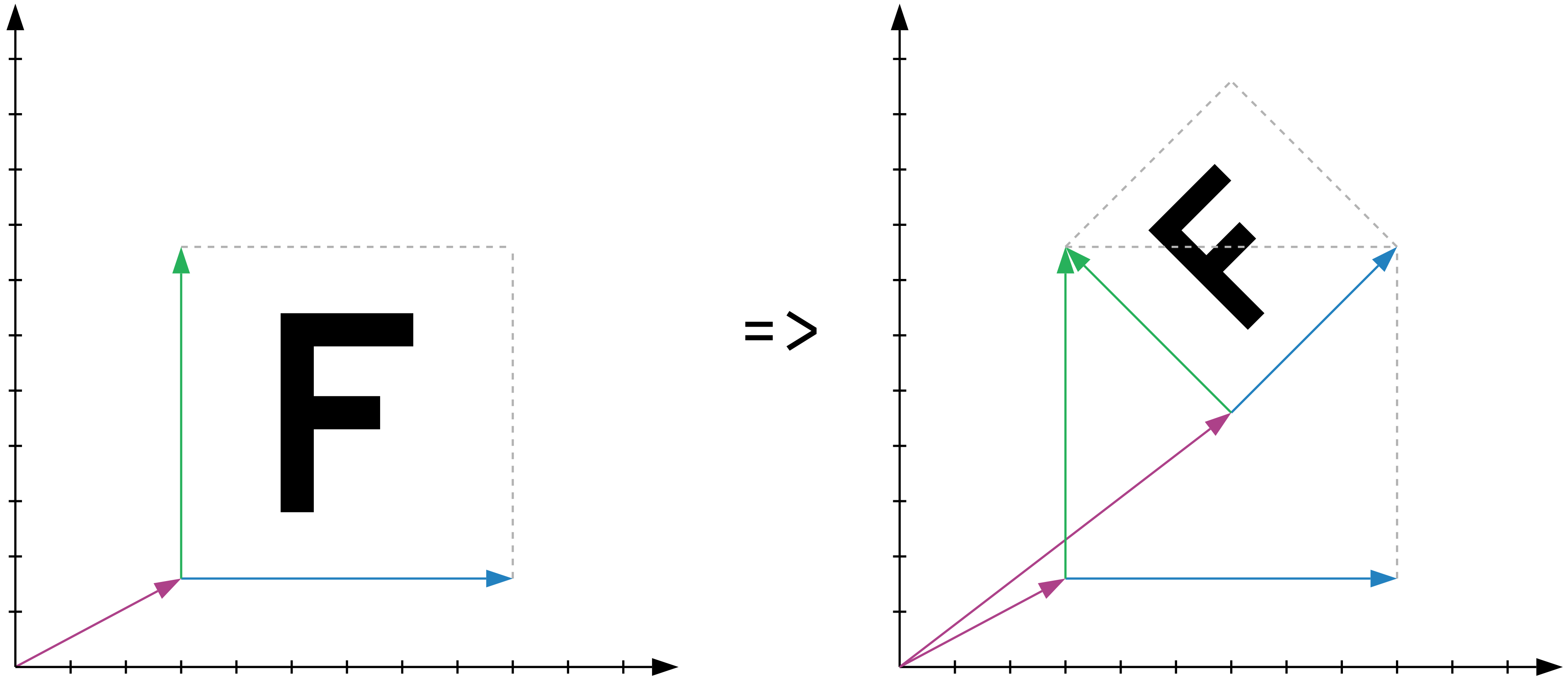


toss

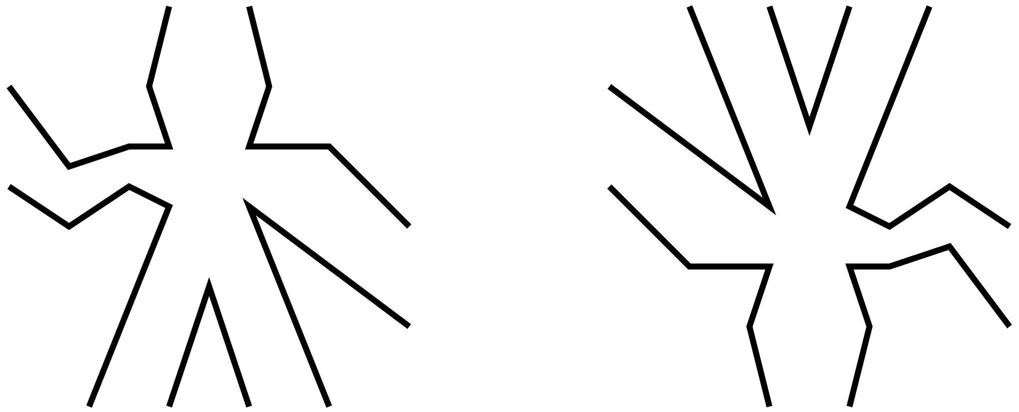


```
tossBox : Box -> Box
tossBox { a, b, c } =
    { a = add a (scale 0.5 (add b c))
    , b = scale 0.5 (add b c)
    , c = scale 0.5 (sub c b) }

toss : Picture -> Picture
toss p = tossBox >> p
```

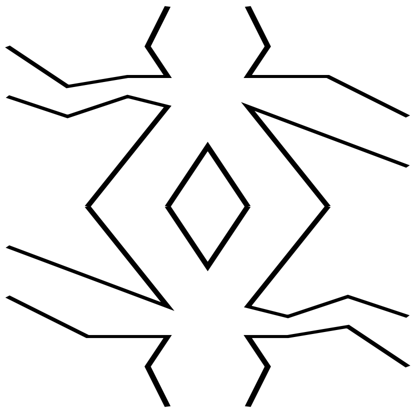
toss



```
above george ((turn >> turn) george)
```

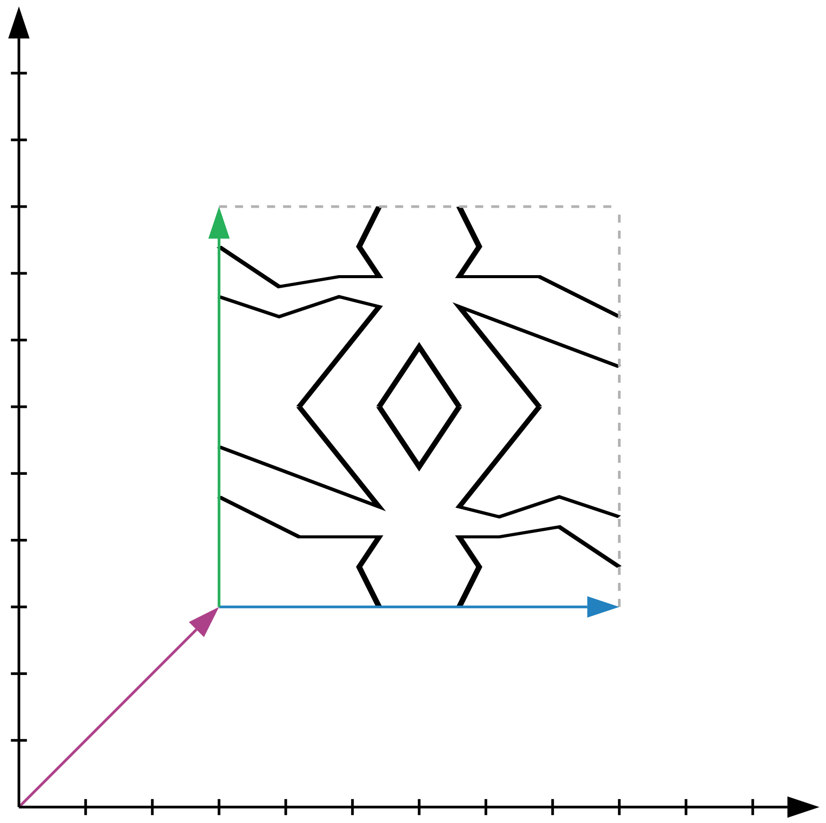


=>

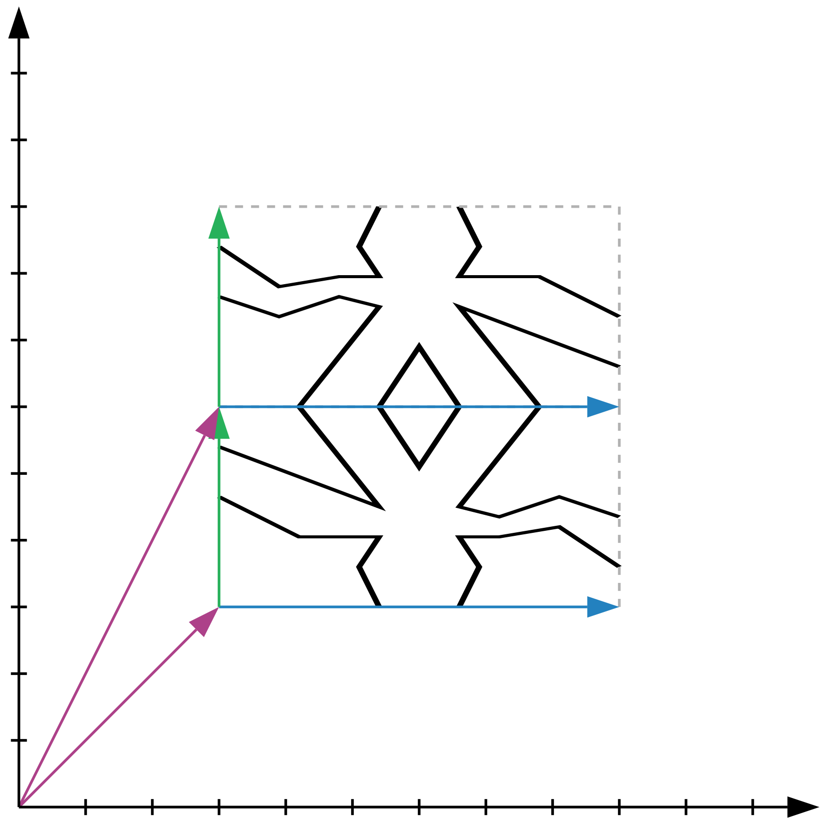


```
aboveRatio : Int -> Int -> Pic -> Pic -> Pic
aboveRatio m n p1 p2 =
    \box ->
        let
            f = m / (m + n)
            (b1, b2) = splitVertically f box
        in
            (p1 b1) ++ (p2 b2)

above : Pic -> Pic -> Pic
above p1 p2 = aboveRatio 1 1
```

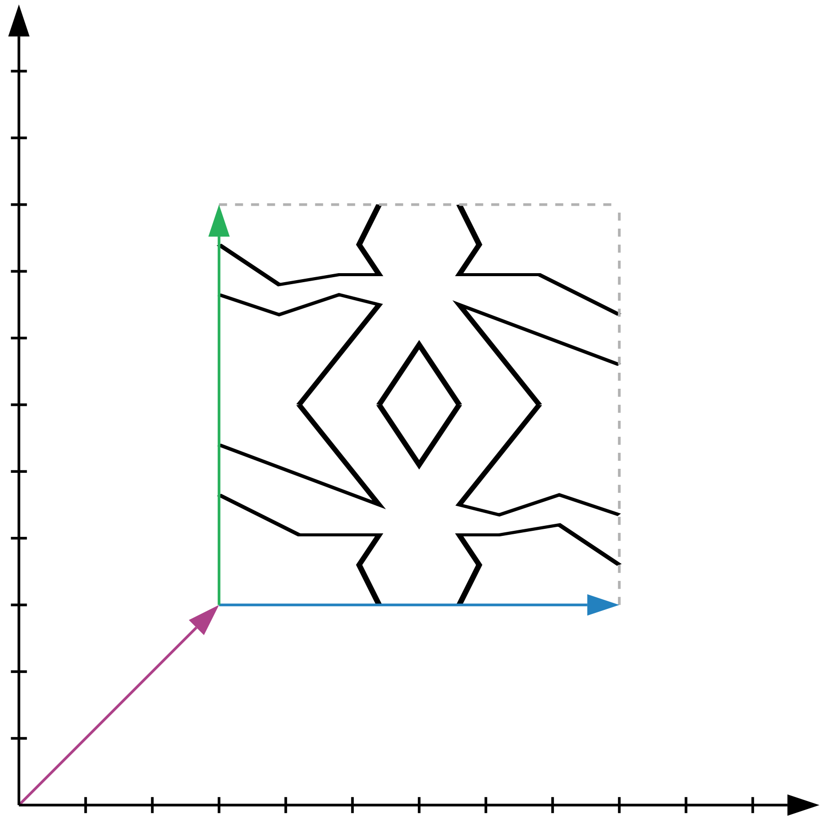
above george ((turn >> turn) george)



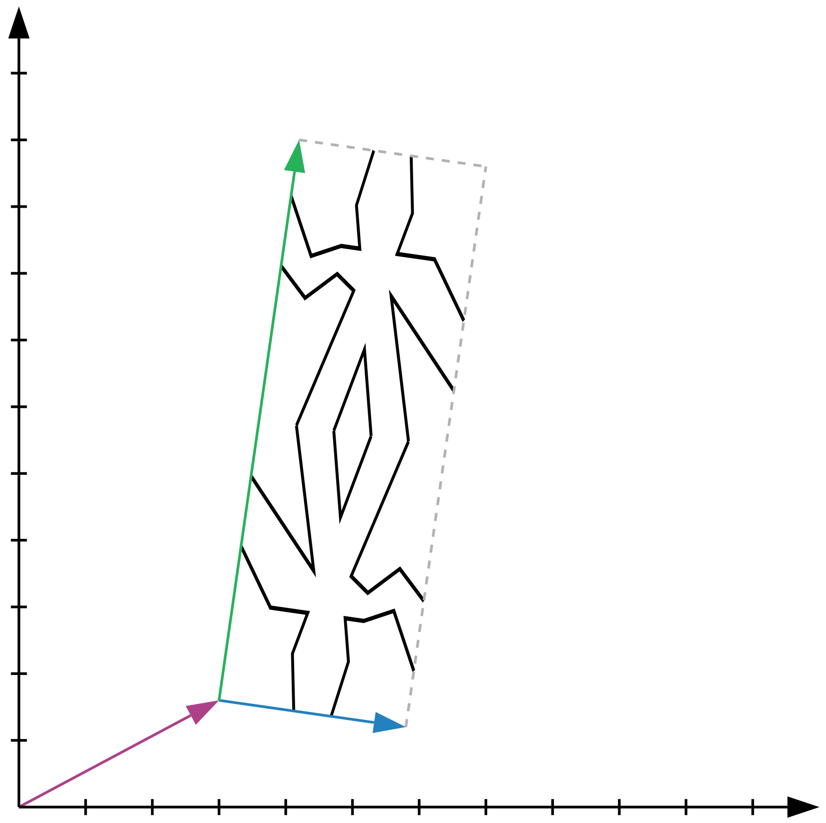
`above george ((turn >> turn) george)`



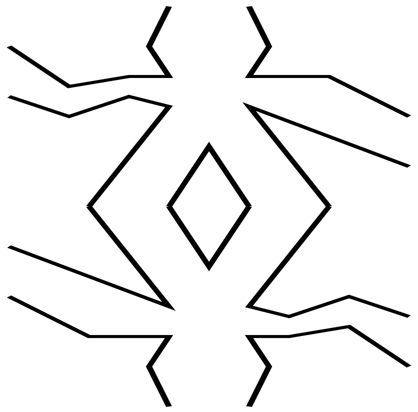
mirrorgeorge

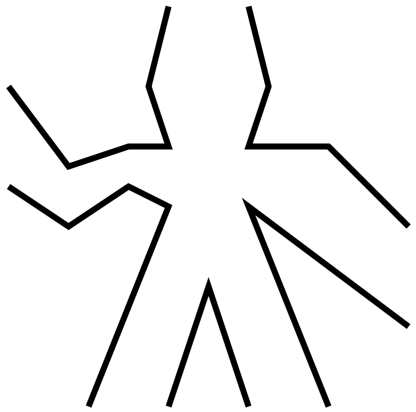


mirrorgeorge

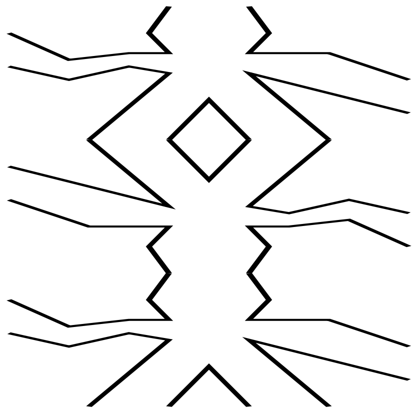


```
aboveRatio 2 1 mirrorgeorge george
```

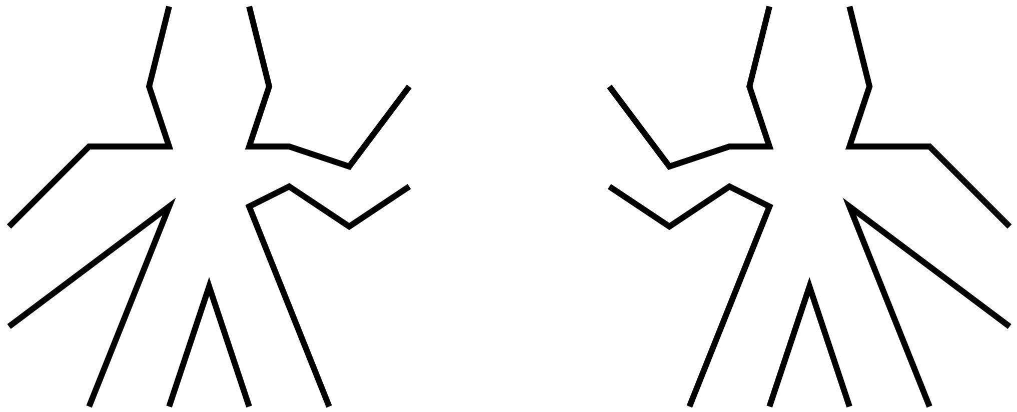




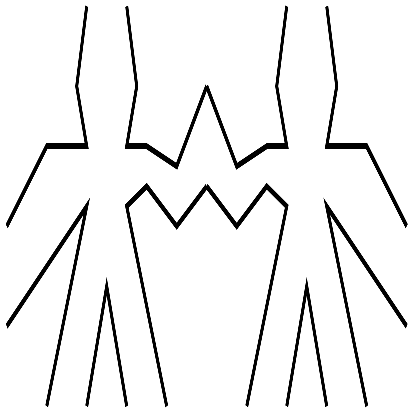
=>



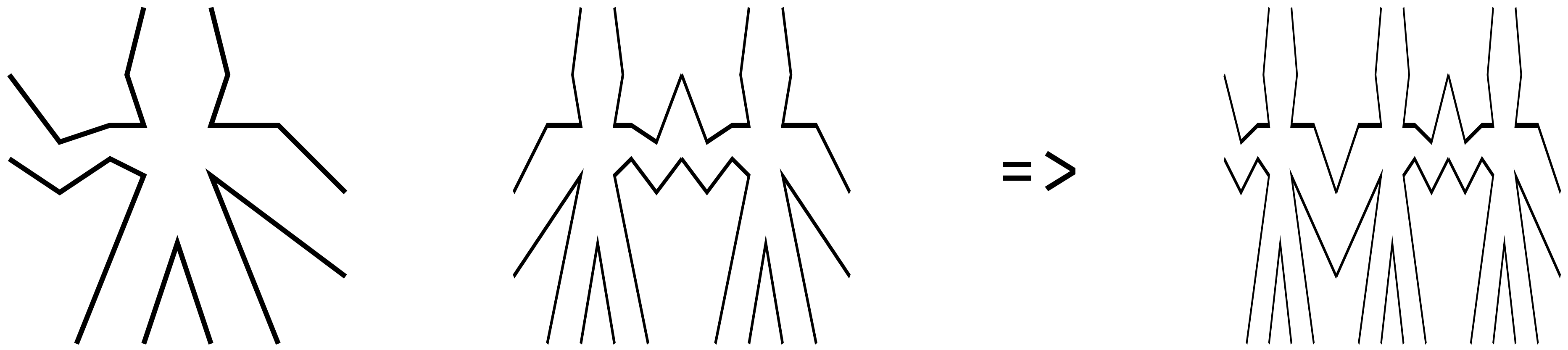
```
beside (flip george) george
```



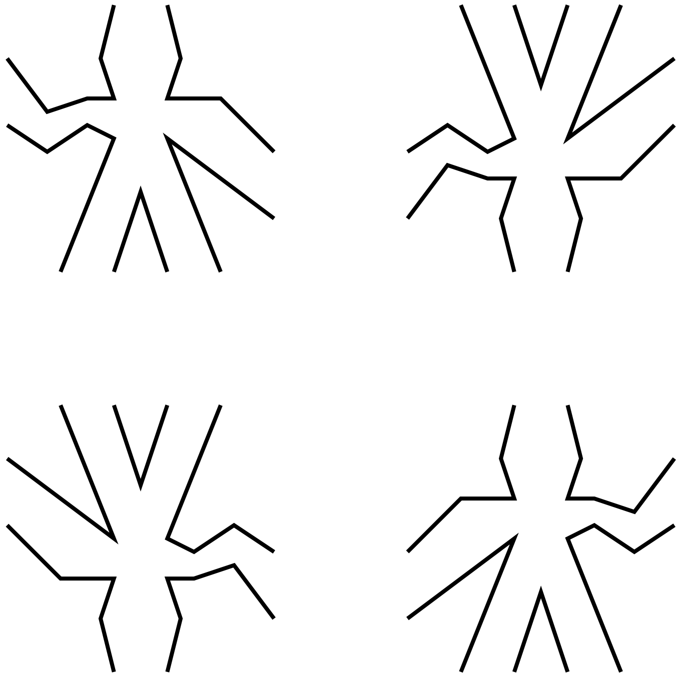
=>



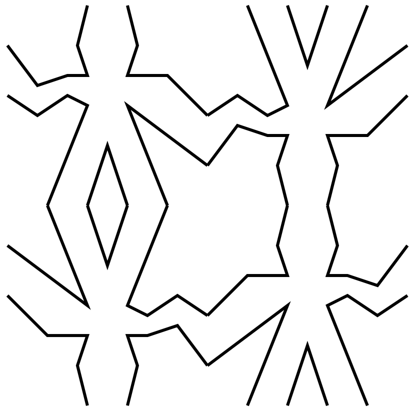
```
besideRatio 1 2 george twingeorge
```



```
quartet g1 g2 g3 g4
```

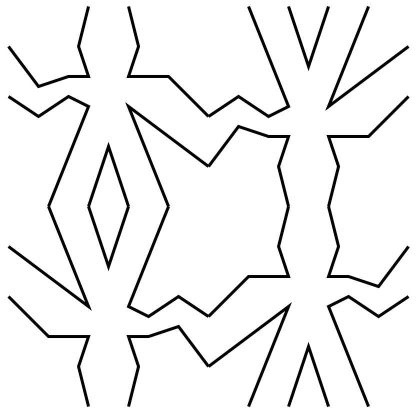


=>

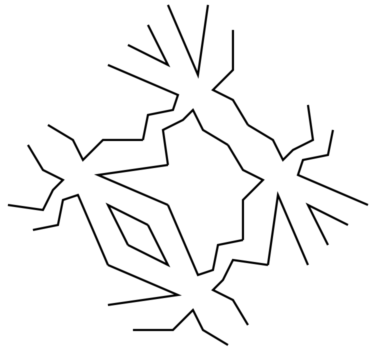


```
quartet : P -> P -> P -> P -> P
quartet nw ne sw se =
    above (beside nw ne)
          (beside sw se)
```

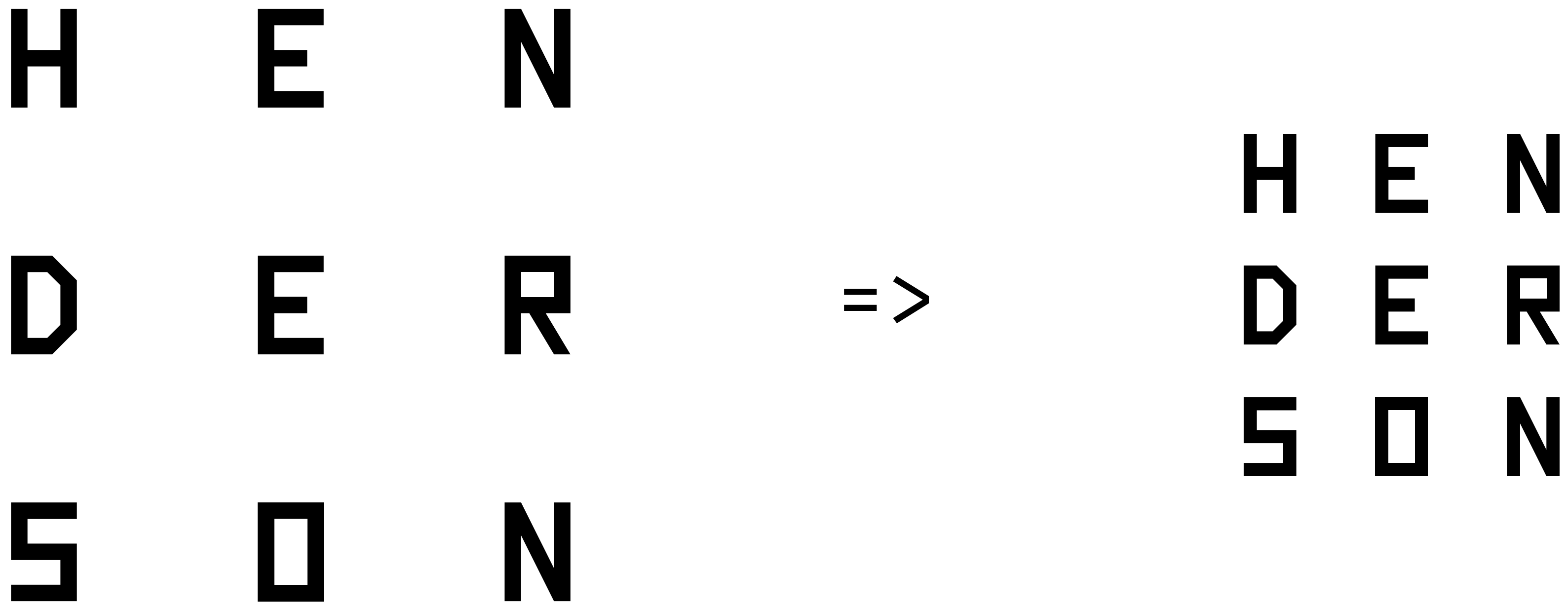
toss



=>

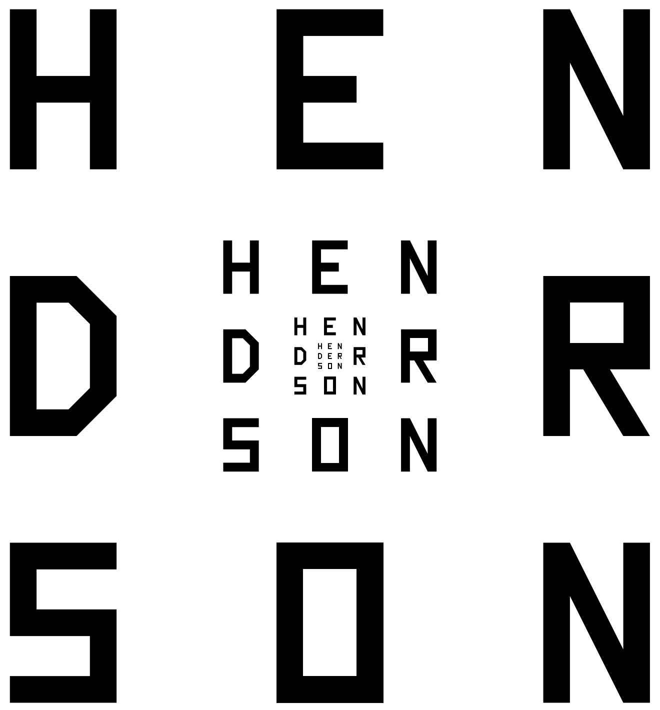


```
nonet h e n d e r s o n
```

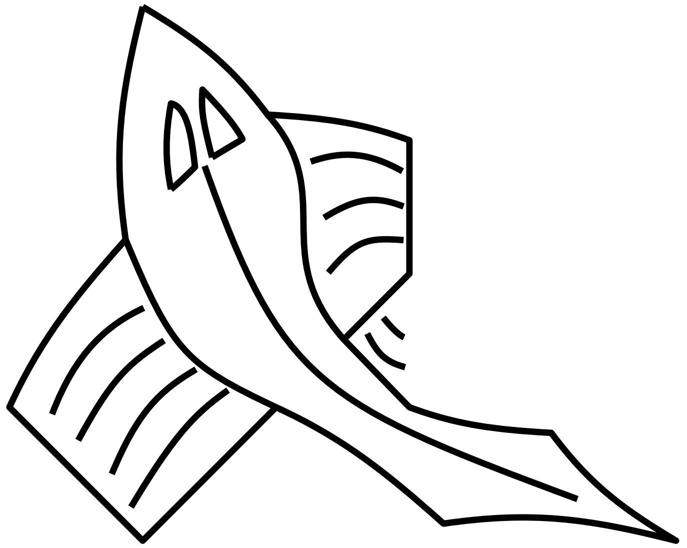


```
nonet : P -> P -> P -> P -> P -> P -> P -> P -> P -> P
    let
        row w m e = besideRatio 1 2 w (beside m e)
        col n m s = aboveRatio 1 2 n (above m s)
    in
        col (row nw nm ne)
            (row mw mm me)
            (row sw sm se)
```

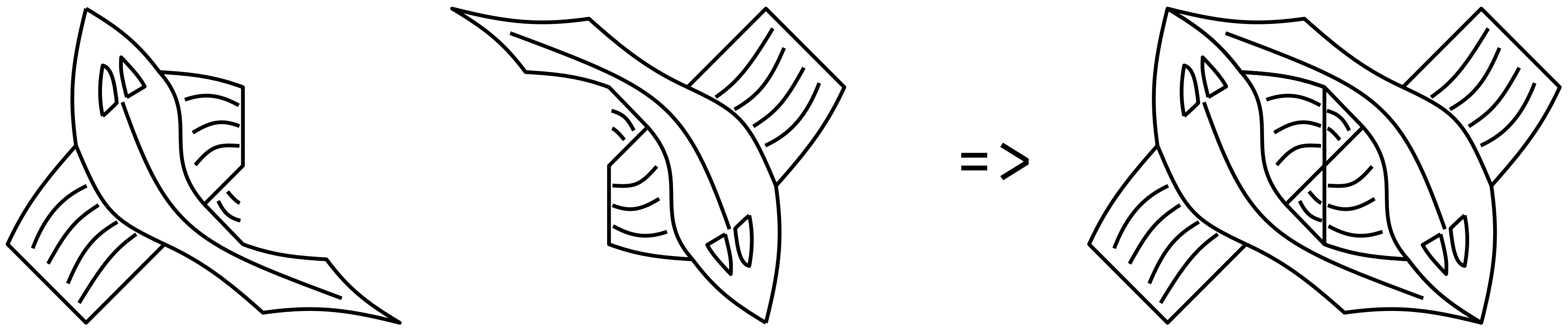
nonets are just pictures



a fish picture

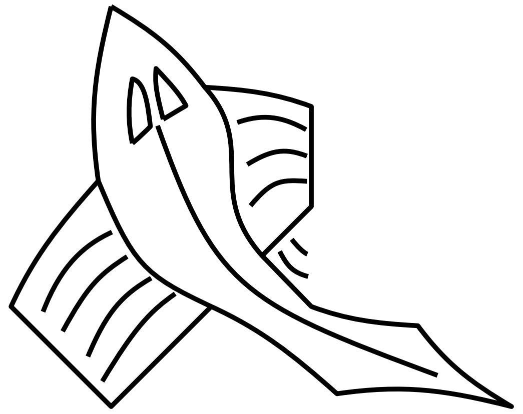


```
over fish ((turn >> turn) fish)
```

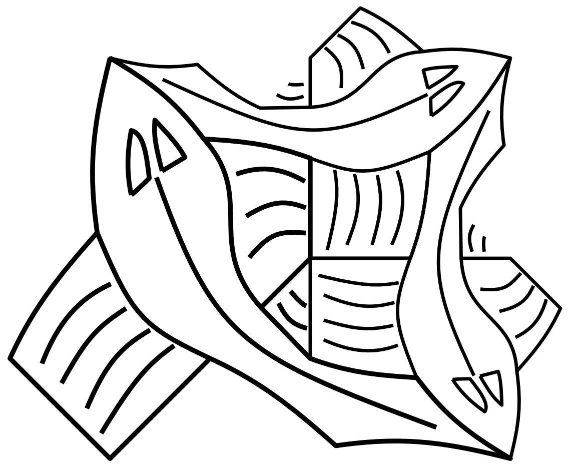


```
over : Pic -> Pic -> Pic
over p1 p2
    \box -> p1 box ++ p2 box
```

ttile

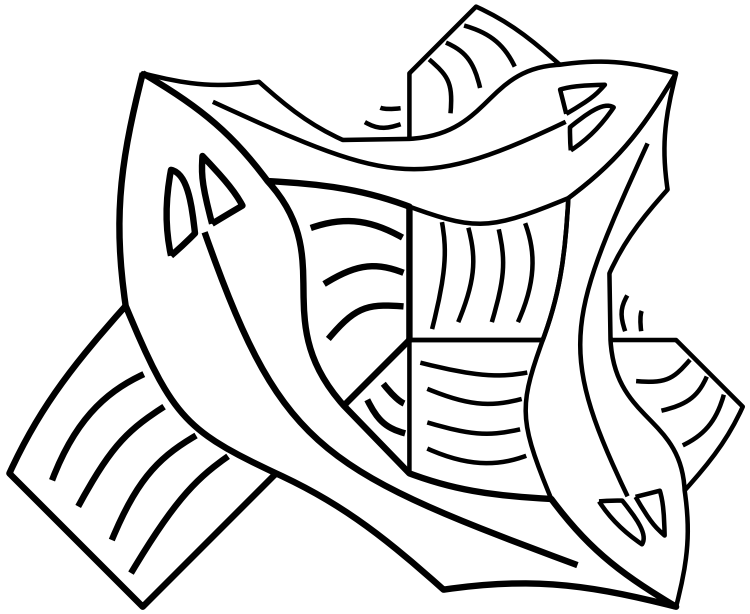


=>

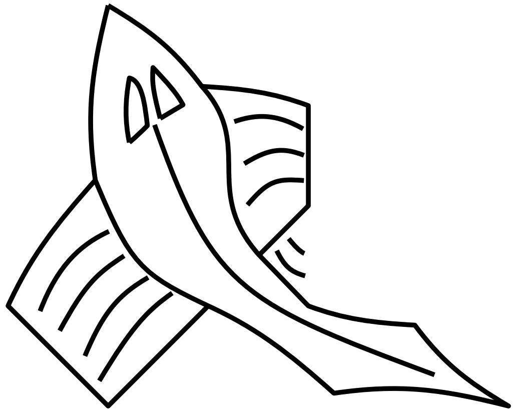


```
ttile : Picture -> Picture
ttile p =
    let
        pn = (toss >> flip) p
        pe = (turn >> turn >> turn) p
    in
        over p (over pn pe)
```

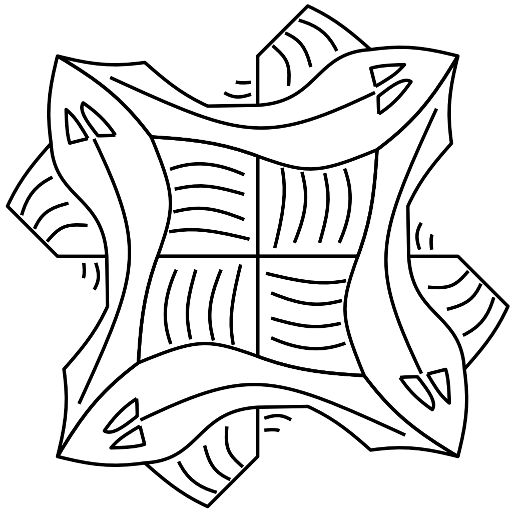
ttile



utile

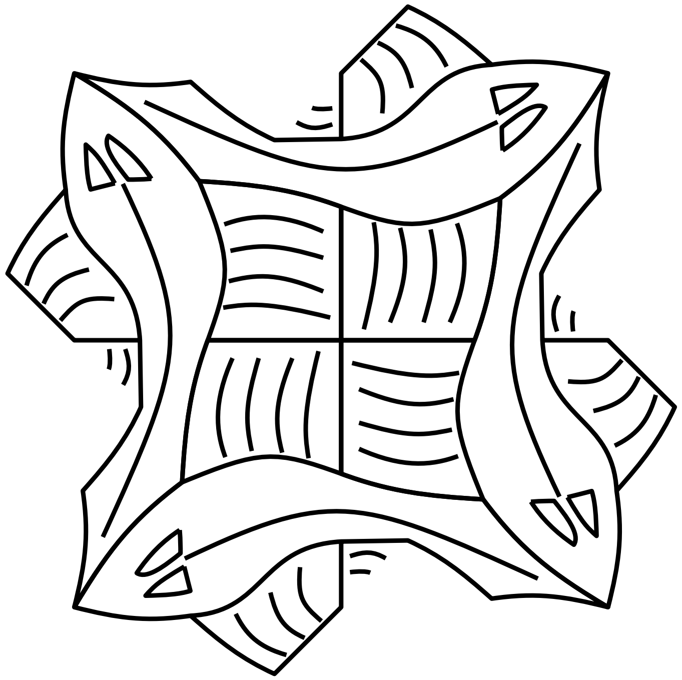


=>

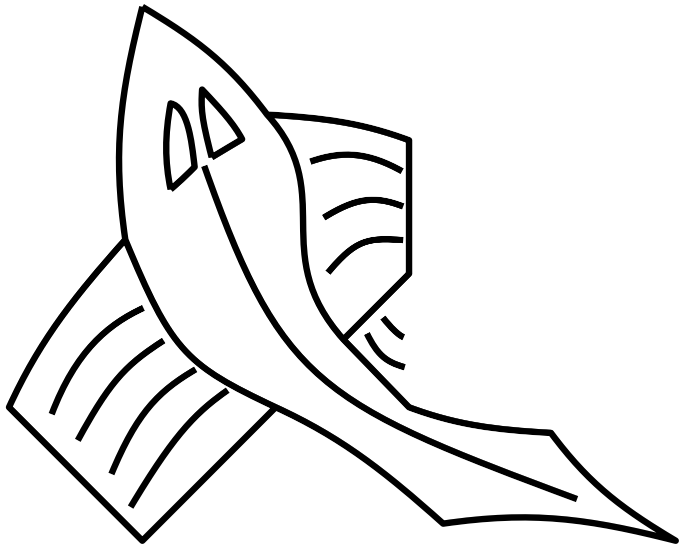


```
utile : Picture -> Picture
utile p =
    let
        pn = (toss >> flip) p
        pw = turn pn
        ps = turn pw
        pe = turn ps
    in
        over pn (over pw (over ps pe))
```

utile

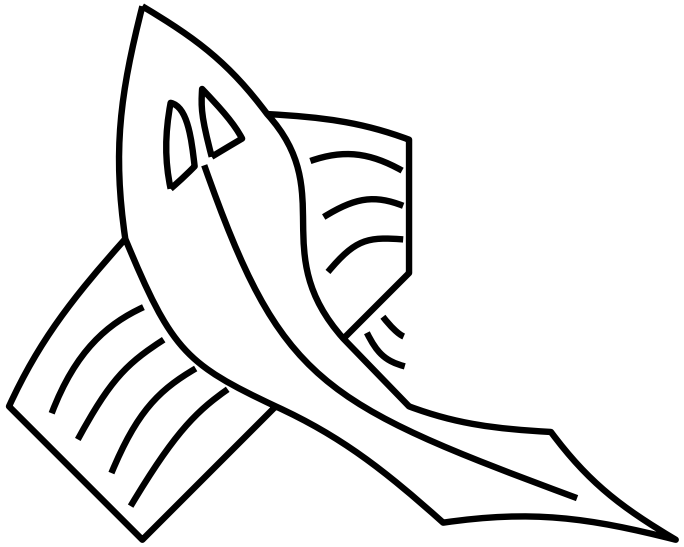


side 0

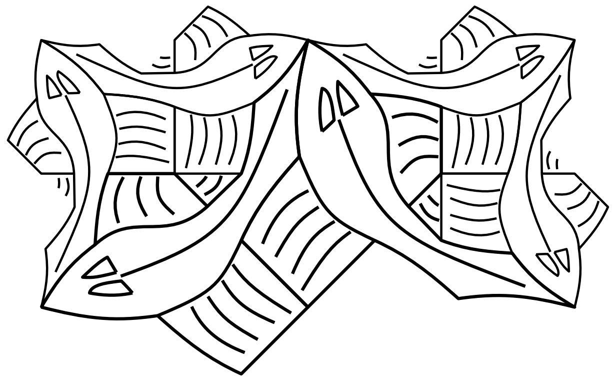


=>

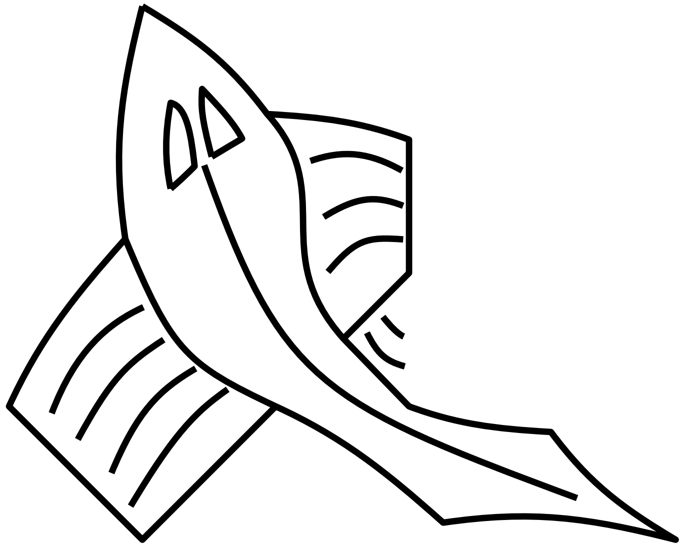
side 1



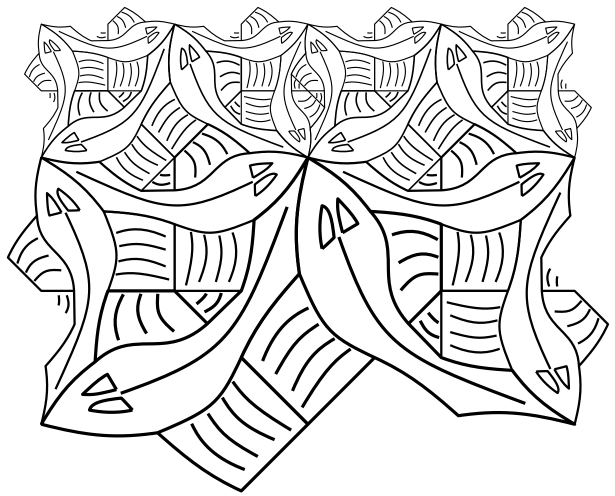
=>



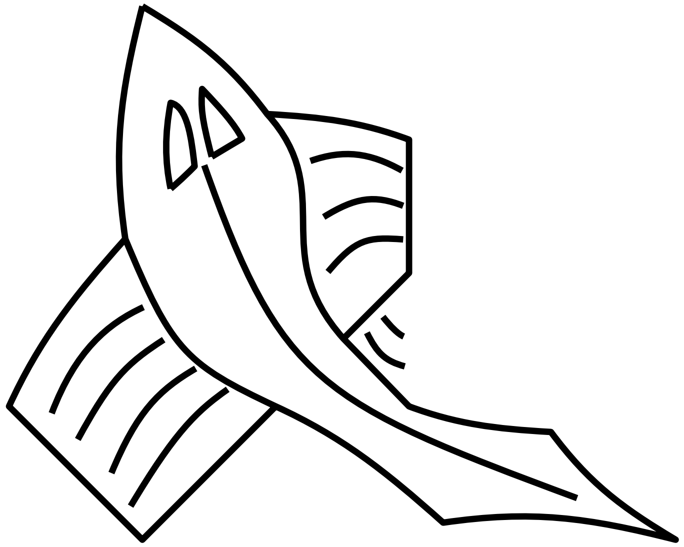
side 2



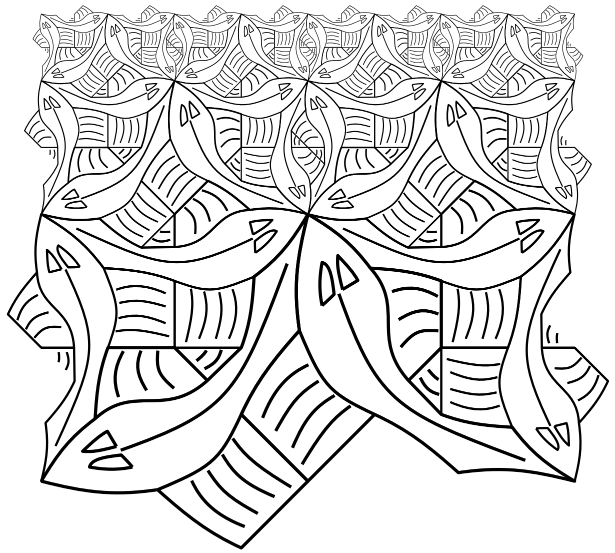
=>



side 3

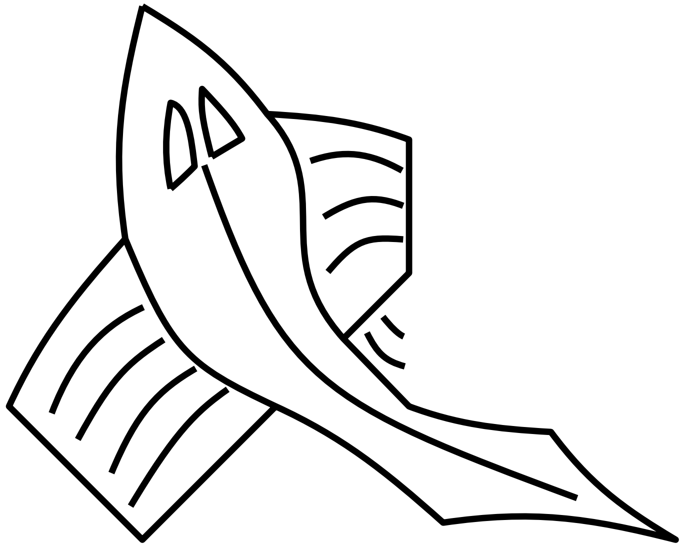


=>



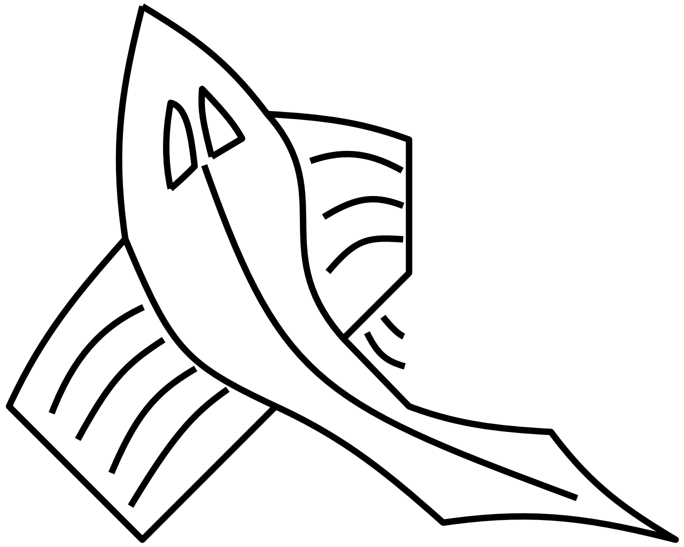
```
side : Int -> Picture -> Picture
side n p =
    if n <= 0 then blank
    else
        let
            s = side (n - 1) p
            t = ttile p
        in
            quartet s s (turn t) t
```

corner 0

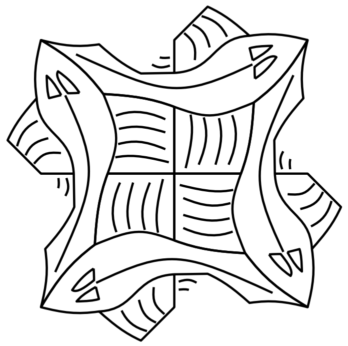


=>

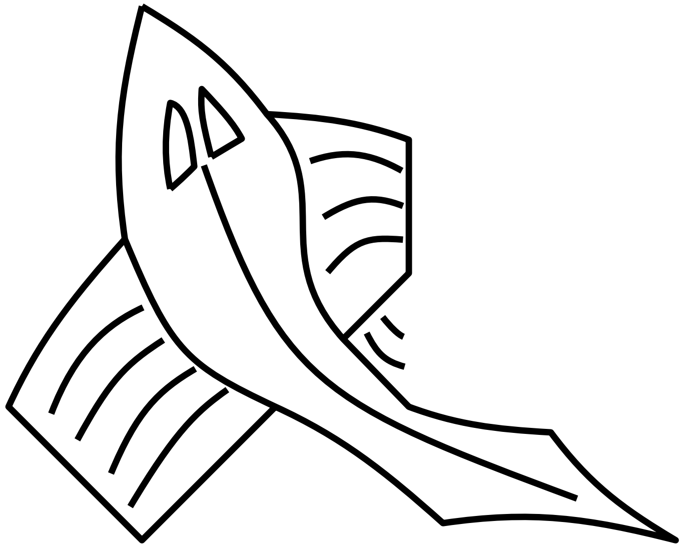
corner 1



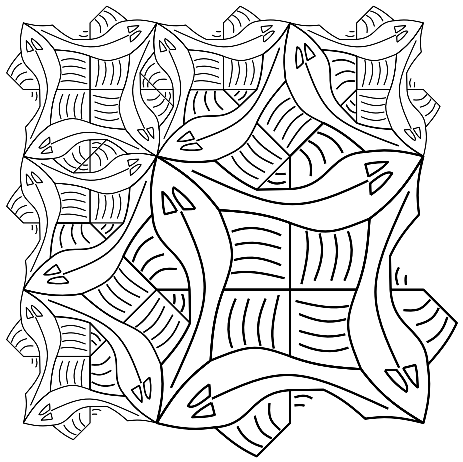
=>



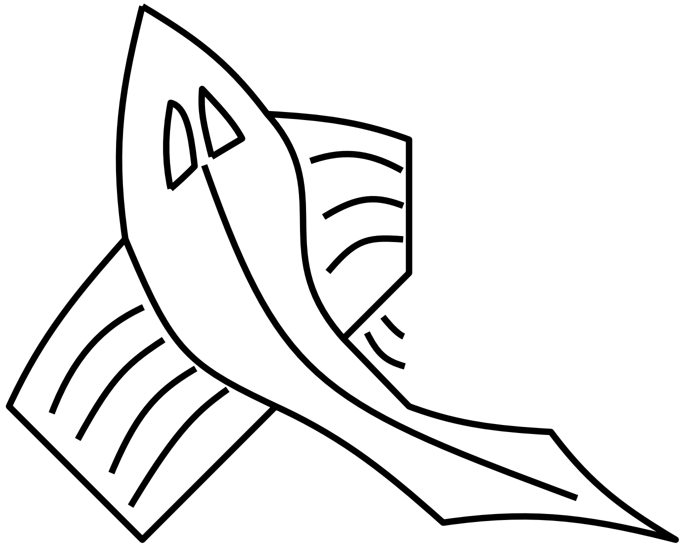
corner 2



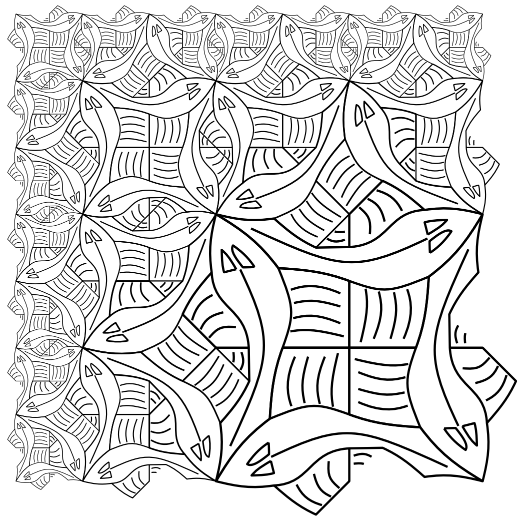
=>



corner 3

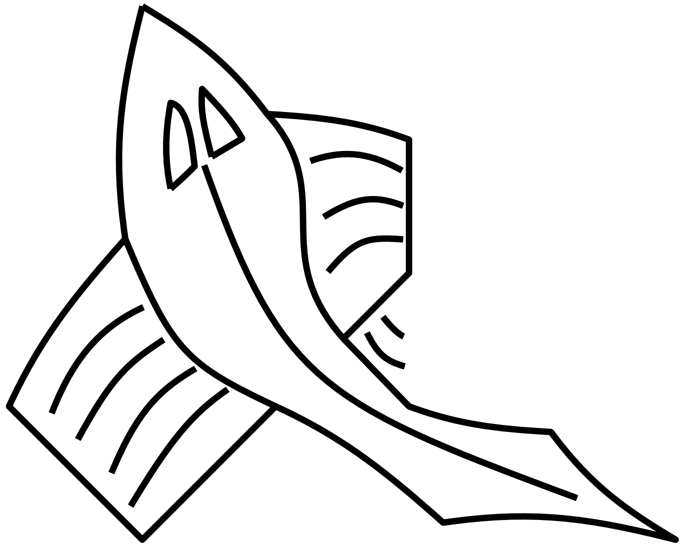


=>

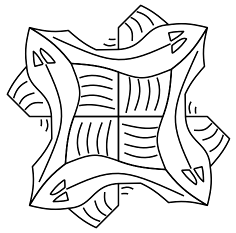


```
corner : Int -> Picture -> Picture
corner n p =
    if n <= 0 then blank
    else
        let
            c = corner (n - 1) p
            s = side (n - 1) p
        in
            quartet c s (turn s) (utile p)
```

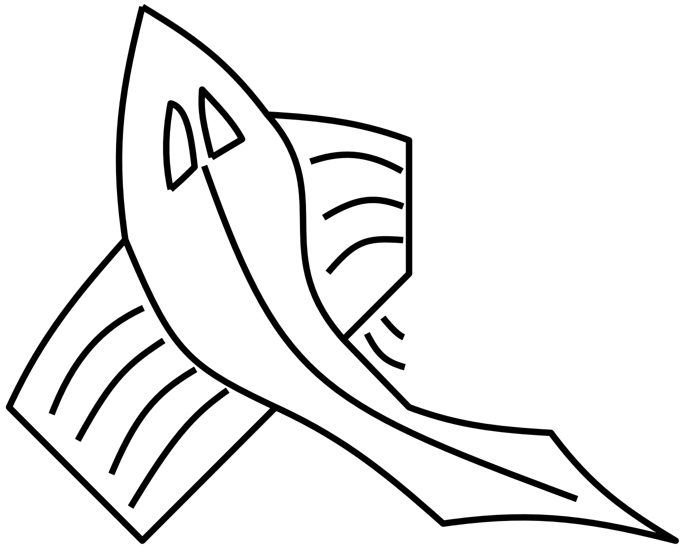
square-limit 0



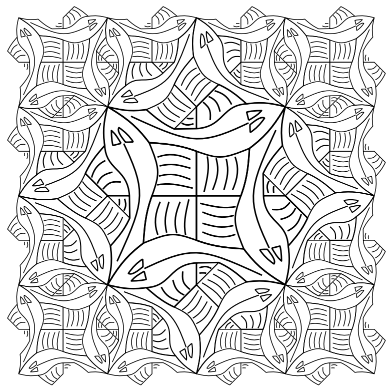
=>



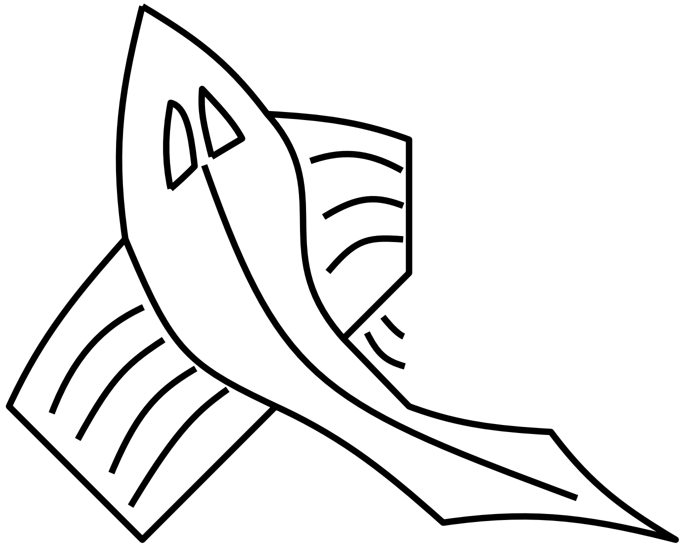
square-limit 1



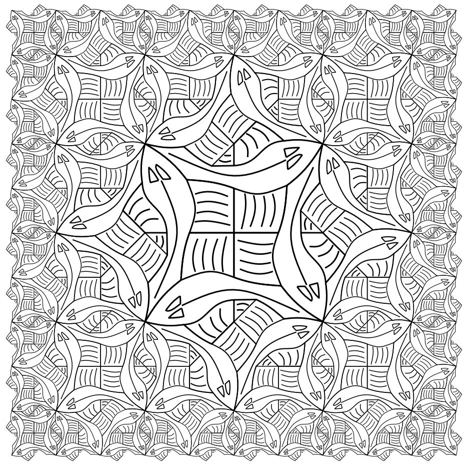
=>



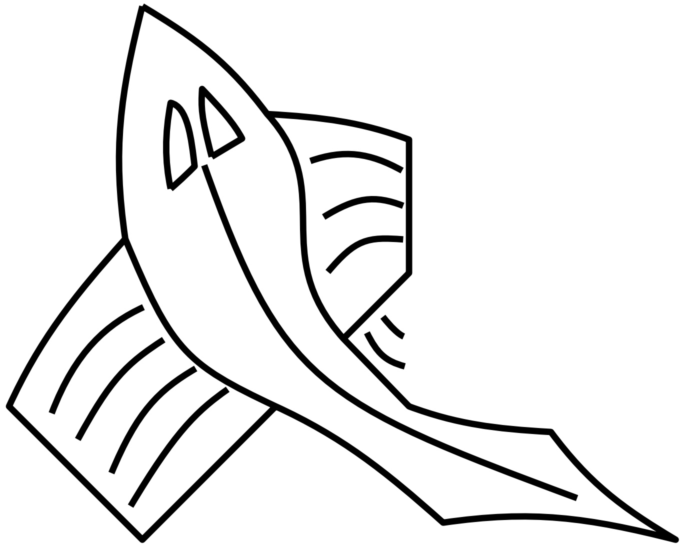
square-limit 2



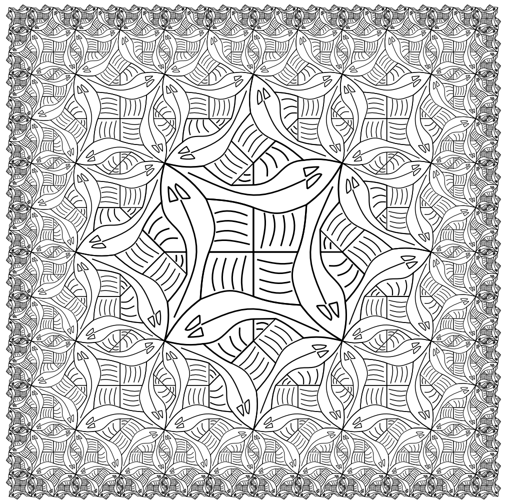
=>



square-limit 3

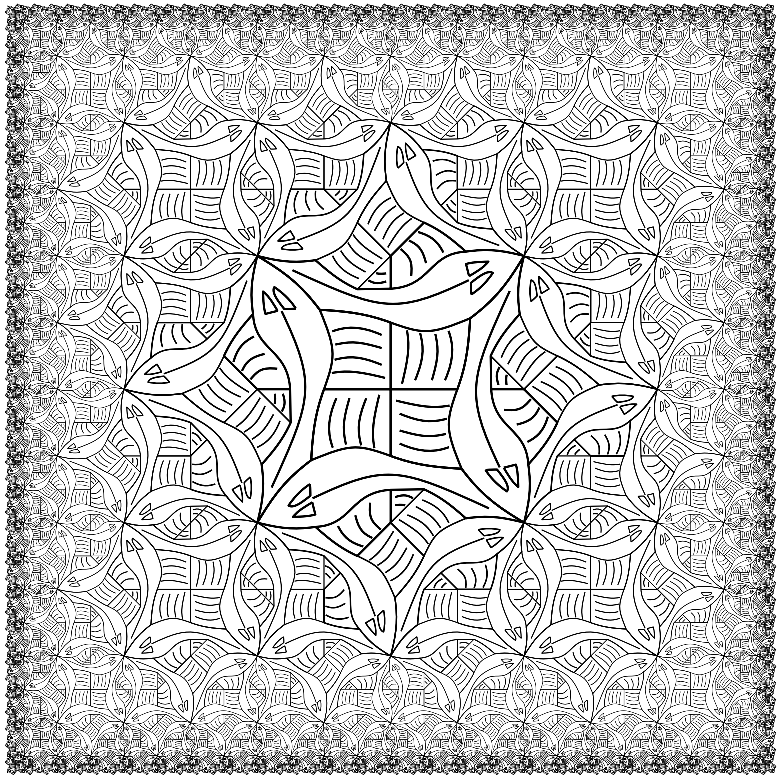


=>



```
squareLimit : Int -> Picture -> Picture
squareLimit n p =
    let
        mm = utile p
        nw = corner n p
        sw = turn nw
        se = turn sw
        ne = turn se
        nm = side n p
        mw = turn nm
        sm = turn mw
        me = turn sm
    in
        nonet nw nm ne mw mm me sw sm se
```

Henderson's square limit



A picture needs to be rendered on a printer or a screen by a device that expects to be given a sequence of commands.

Programming that sequence of commands directly is much harder than having an application generate the commands automatically from the simpler, denotational description.

The pictures were drawn by a Java program which generated PostScript commands directly. The Java was written in a functional style so that the definitions which were executed were exactly as they appear in the paper.

The pictures were drawn by a PostScript program which generated PostScript commands directly.
The PostScript was written in a functional style so that the definitions which were executed were not unlike as they appear in the paper.

It probably is true that PostScript is not everyone's first choice as a programming language. But let's put that premise behind us, and assume that you need (or want) to write a program in the PostScript language.