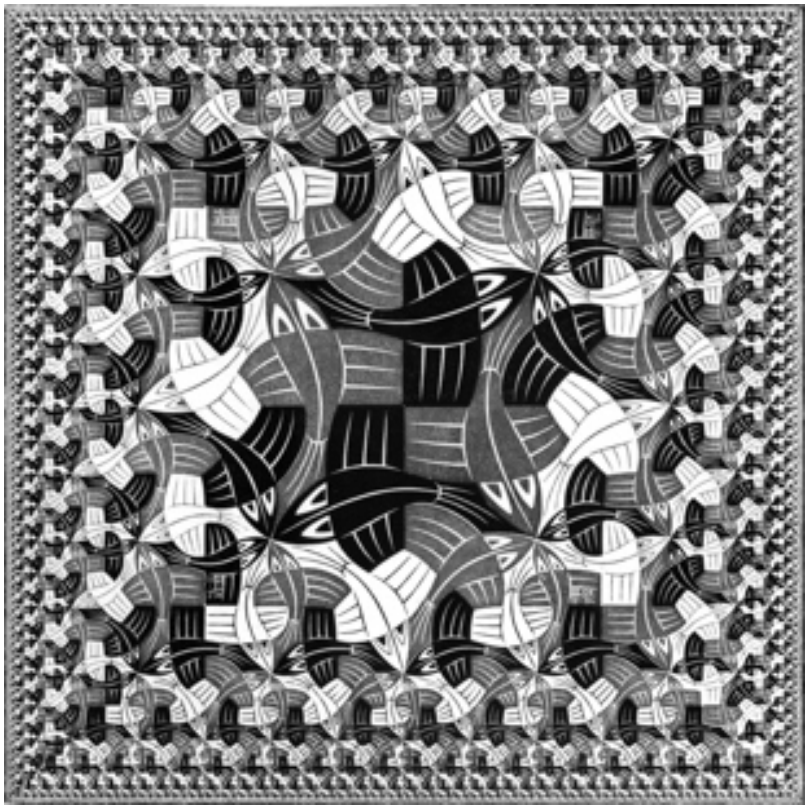


Functional Geometry

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.

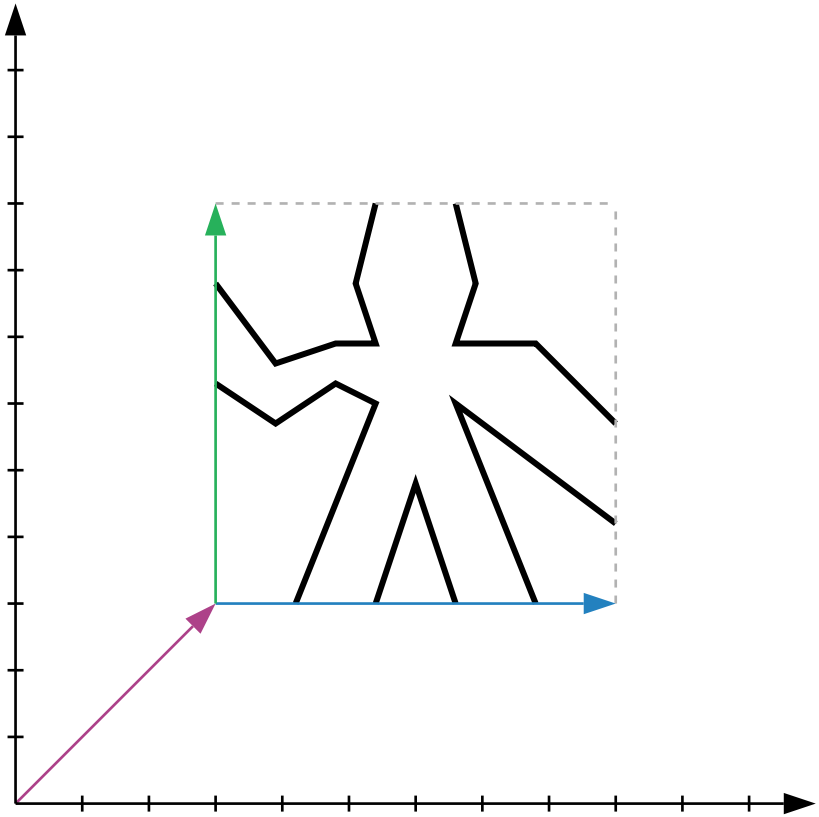
A picture is an example
of a complex object that
can be described in terms
of its parts.

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.

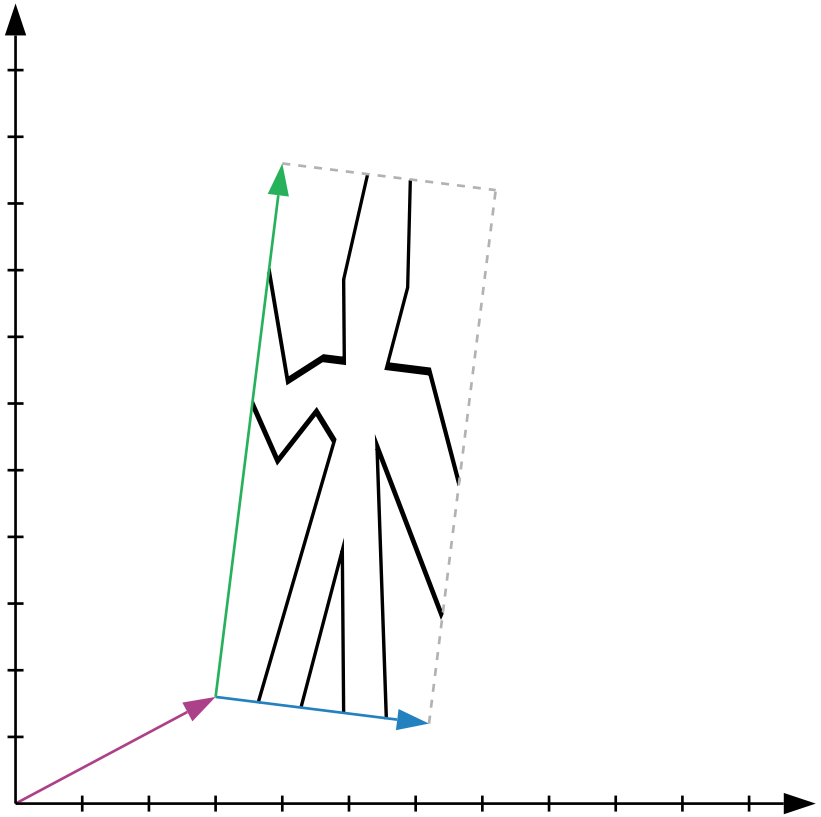
```
type Box = { a : Vector
           , b : Vector
           , c : Vector }

type Picture = Box -> Rendering
```

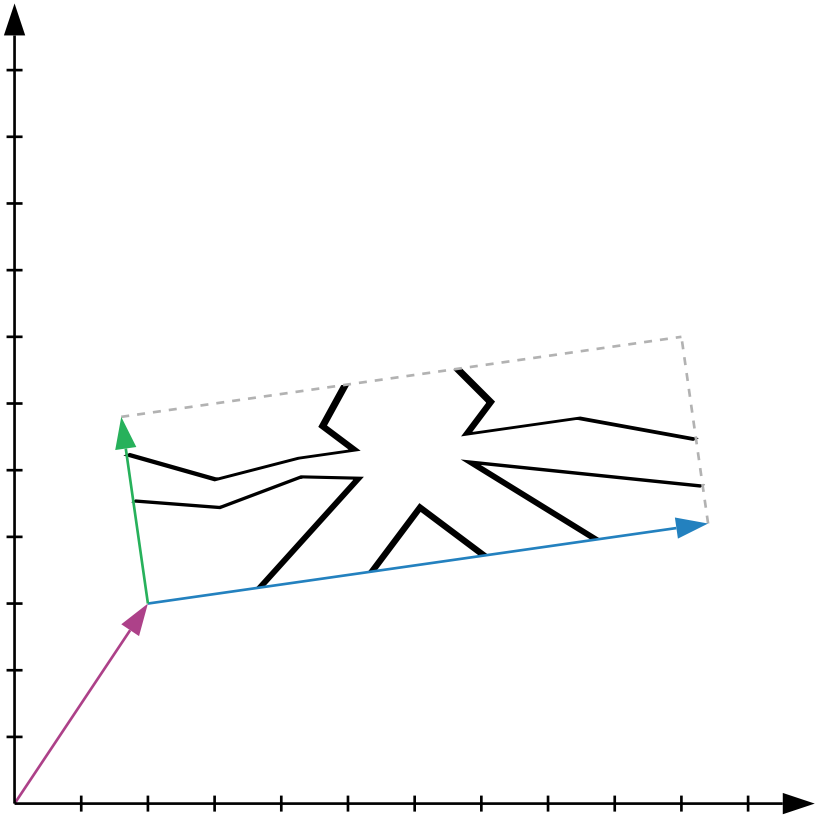
george



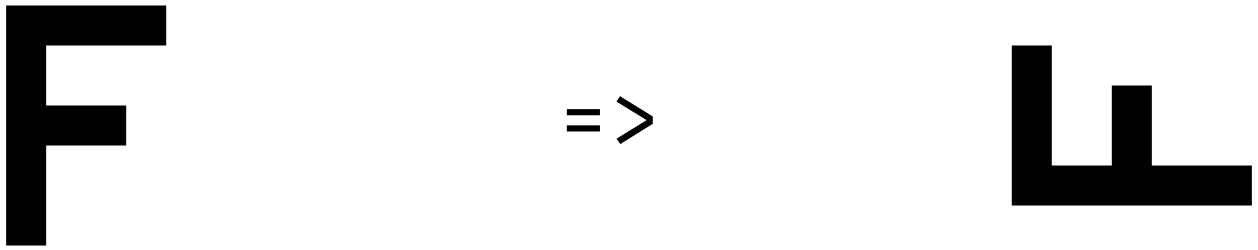
also george



still george



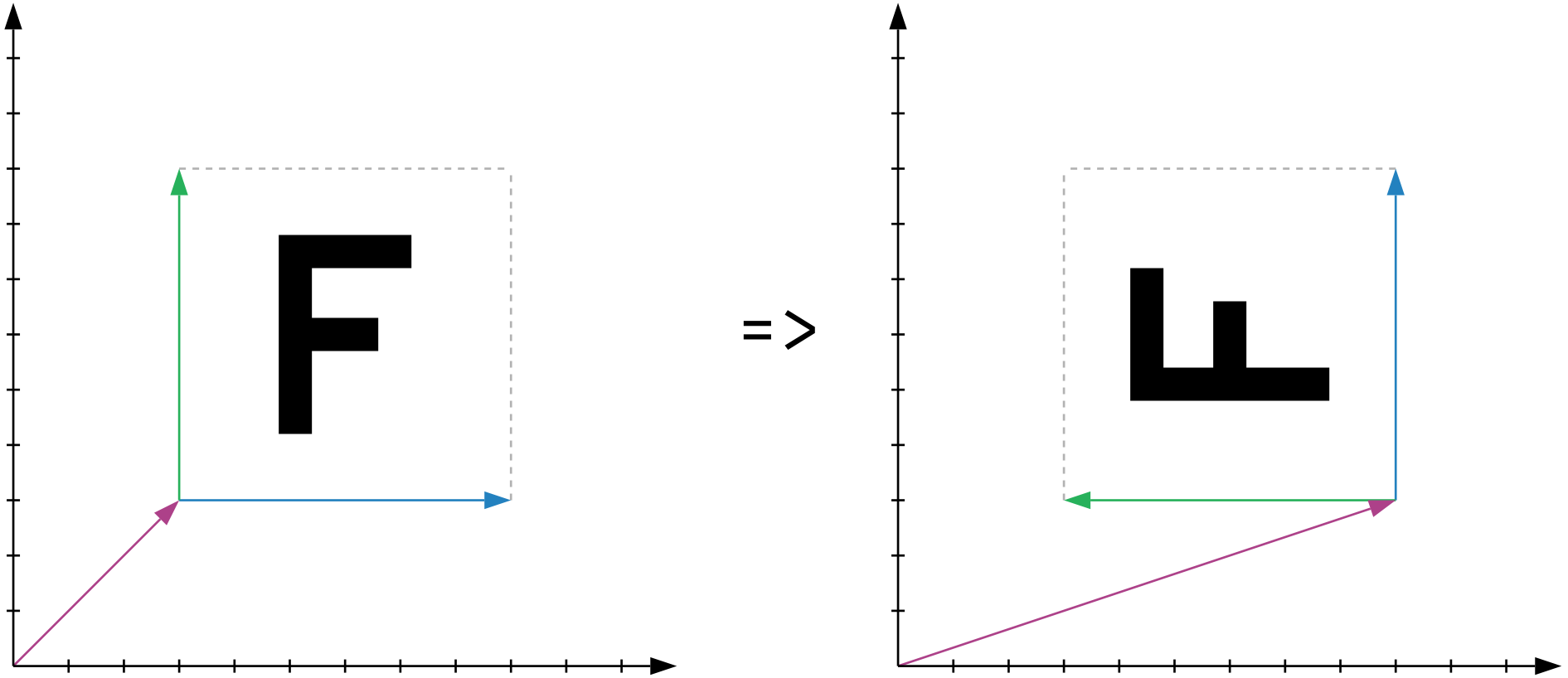
turn



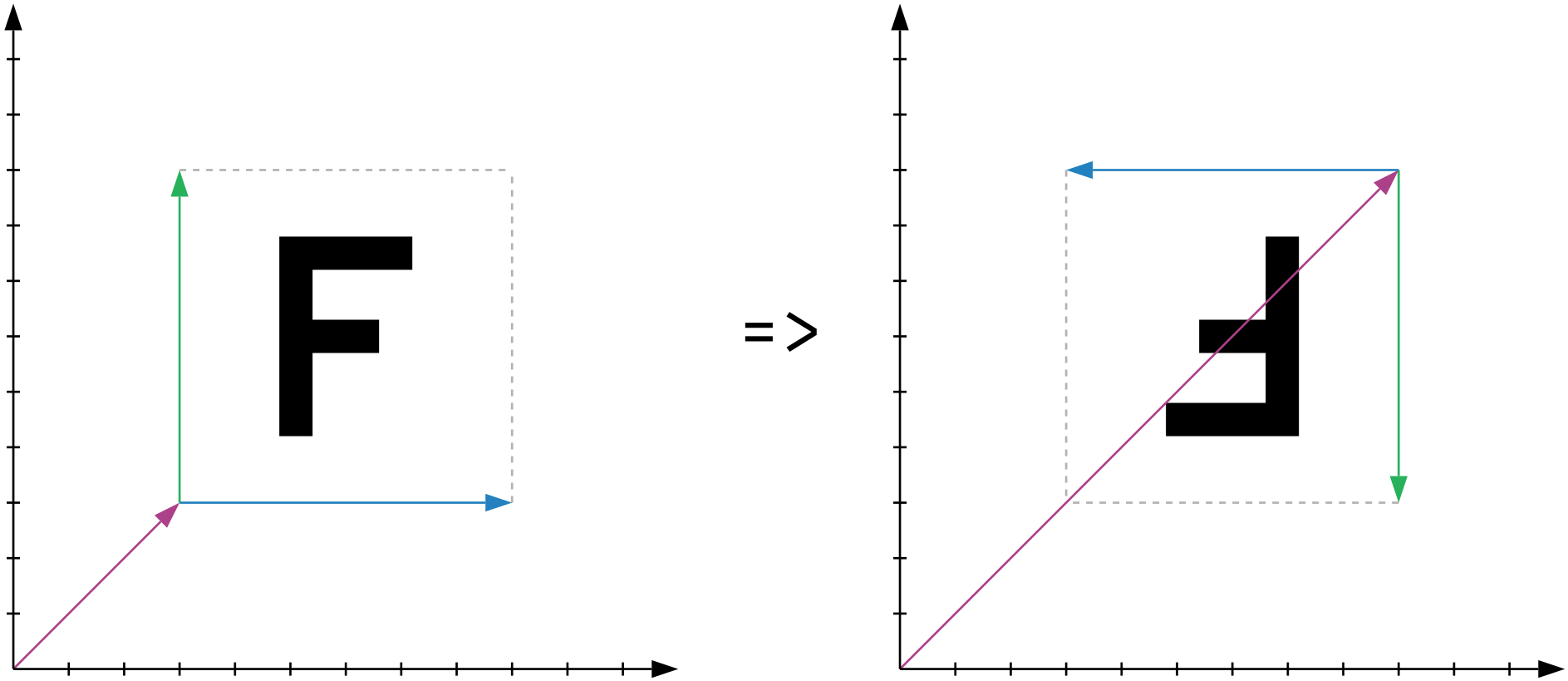
```
turnBox : Box -> Box
turnBox { a, b, c } = { a = add a b
                      , b = c
                      , c = neg b }

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

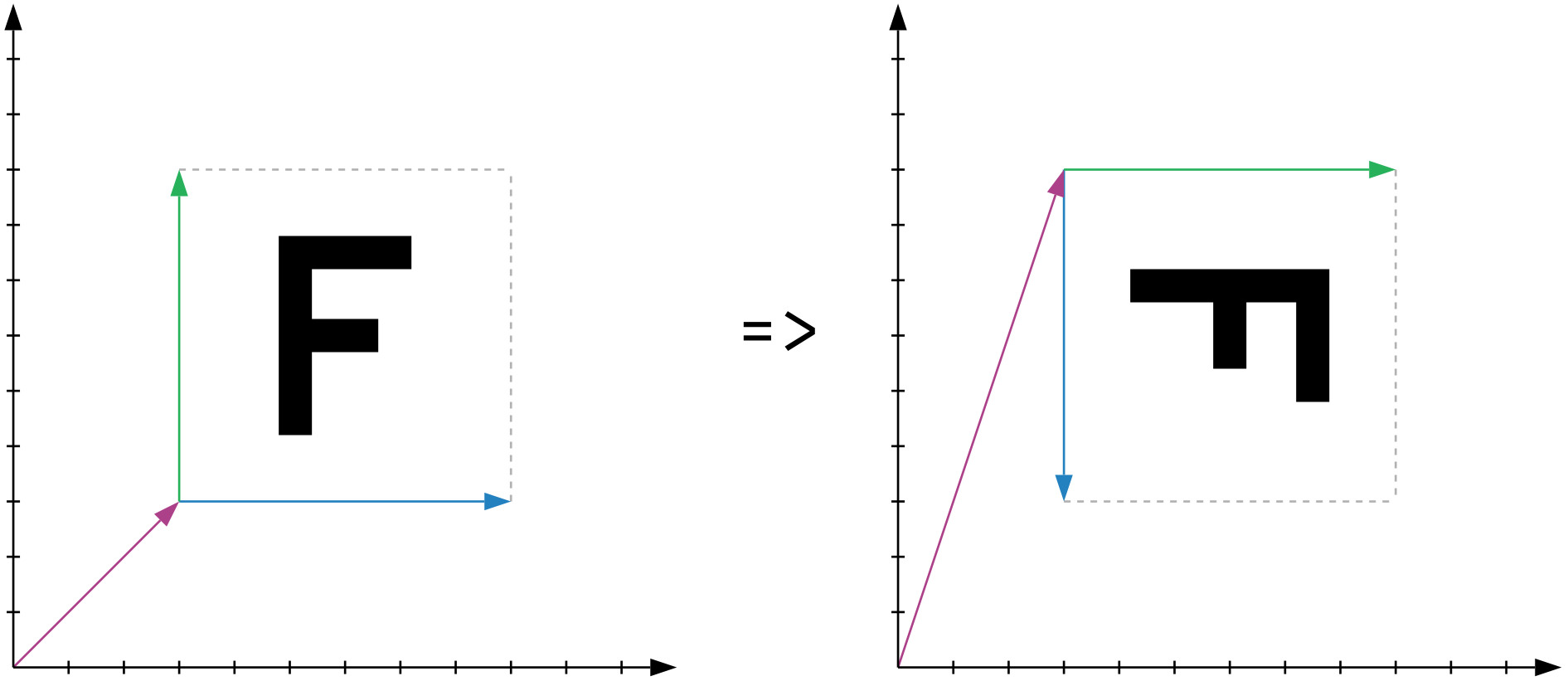
turn



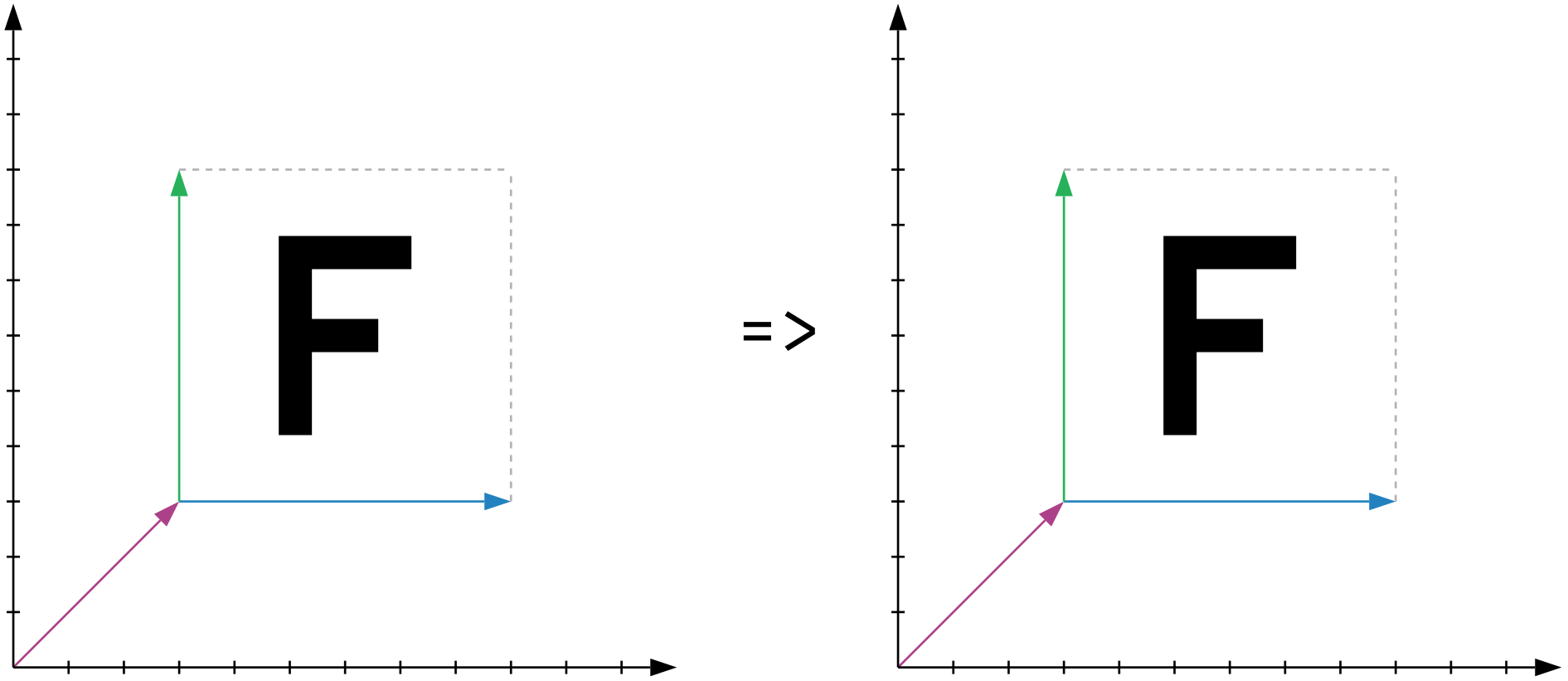
turn >> turn



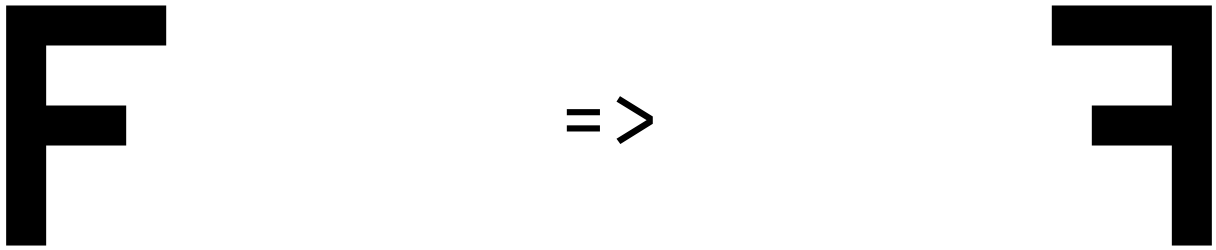
```
turn >> turn >> turn
```



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



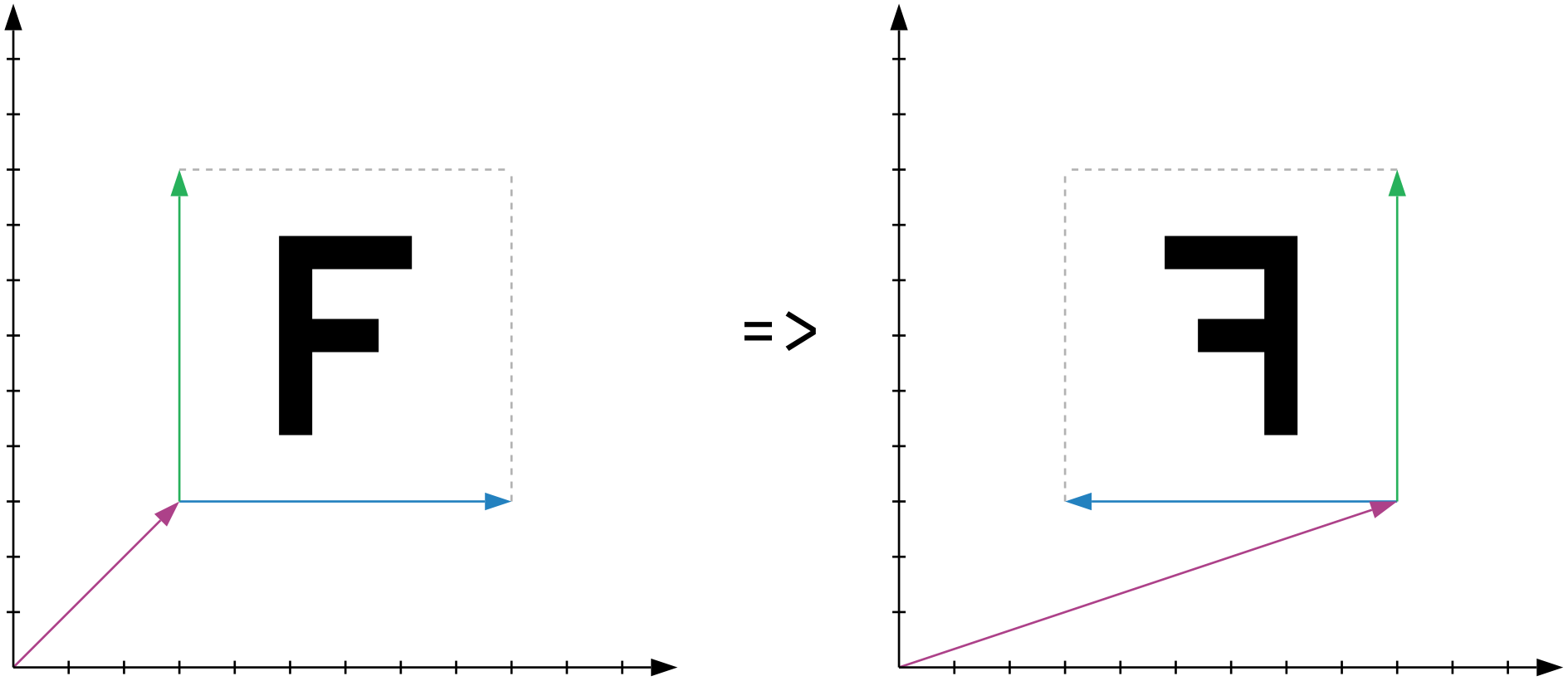
flip



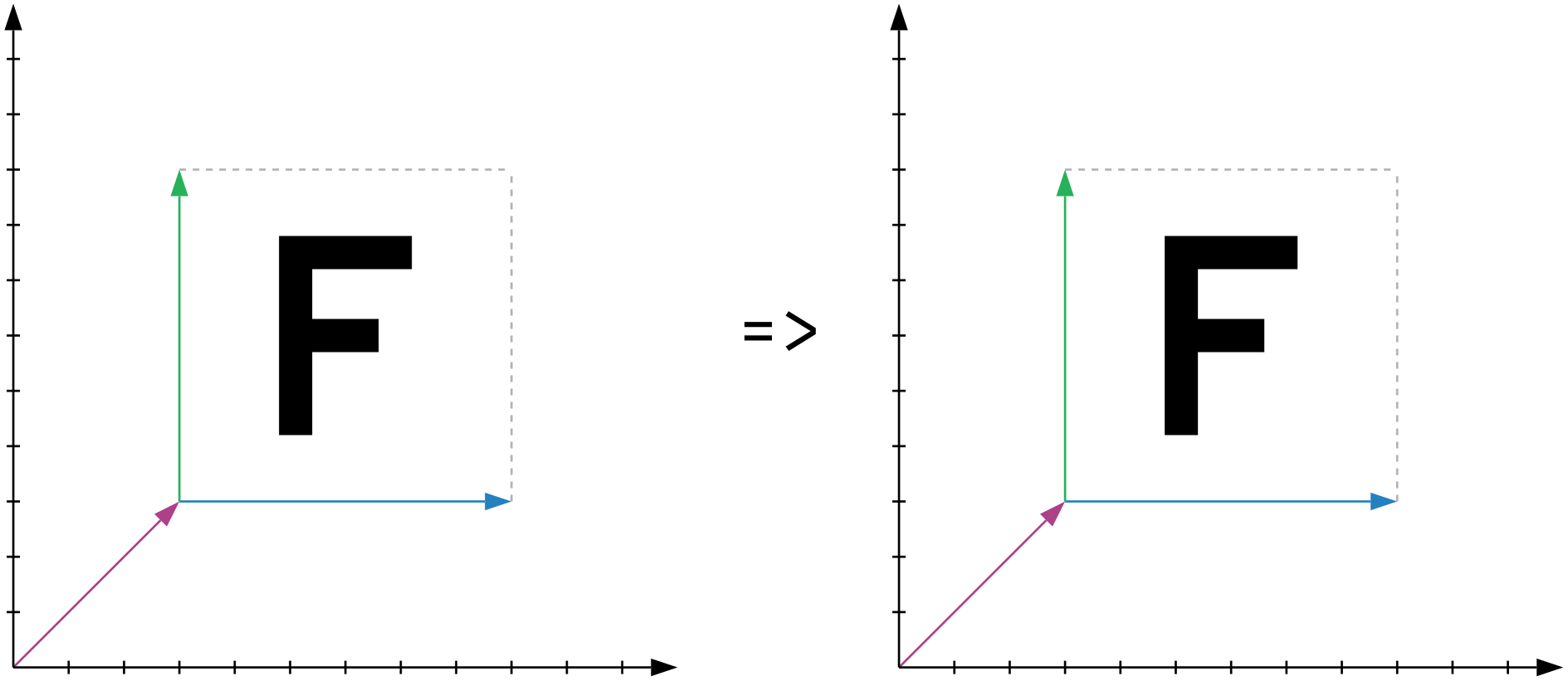
```
flipBox : Box -> Box
flipBox { a, b, c } = { a = add a b
                      , b = neg b
                      , c = neg c }

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

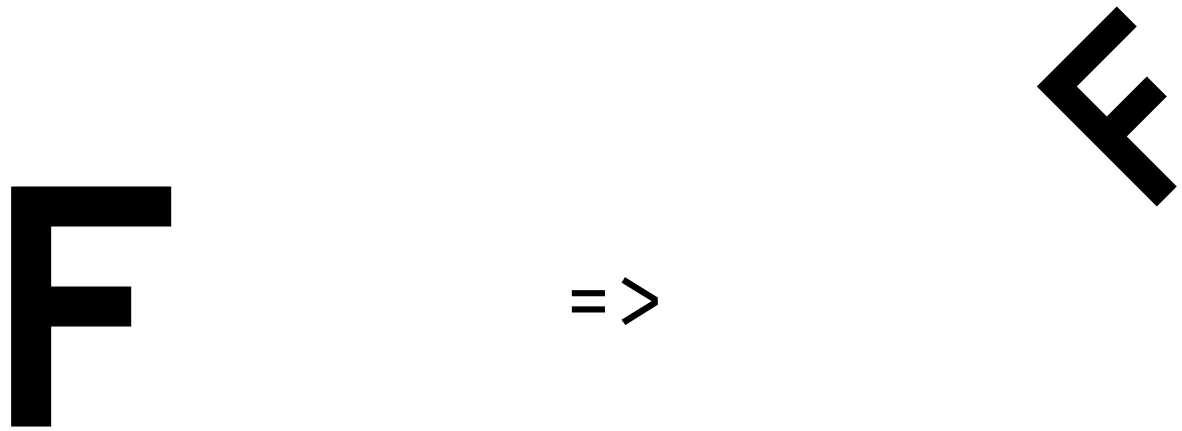
flip



flip >> flip



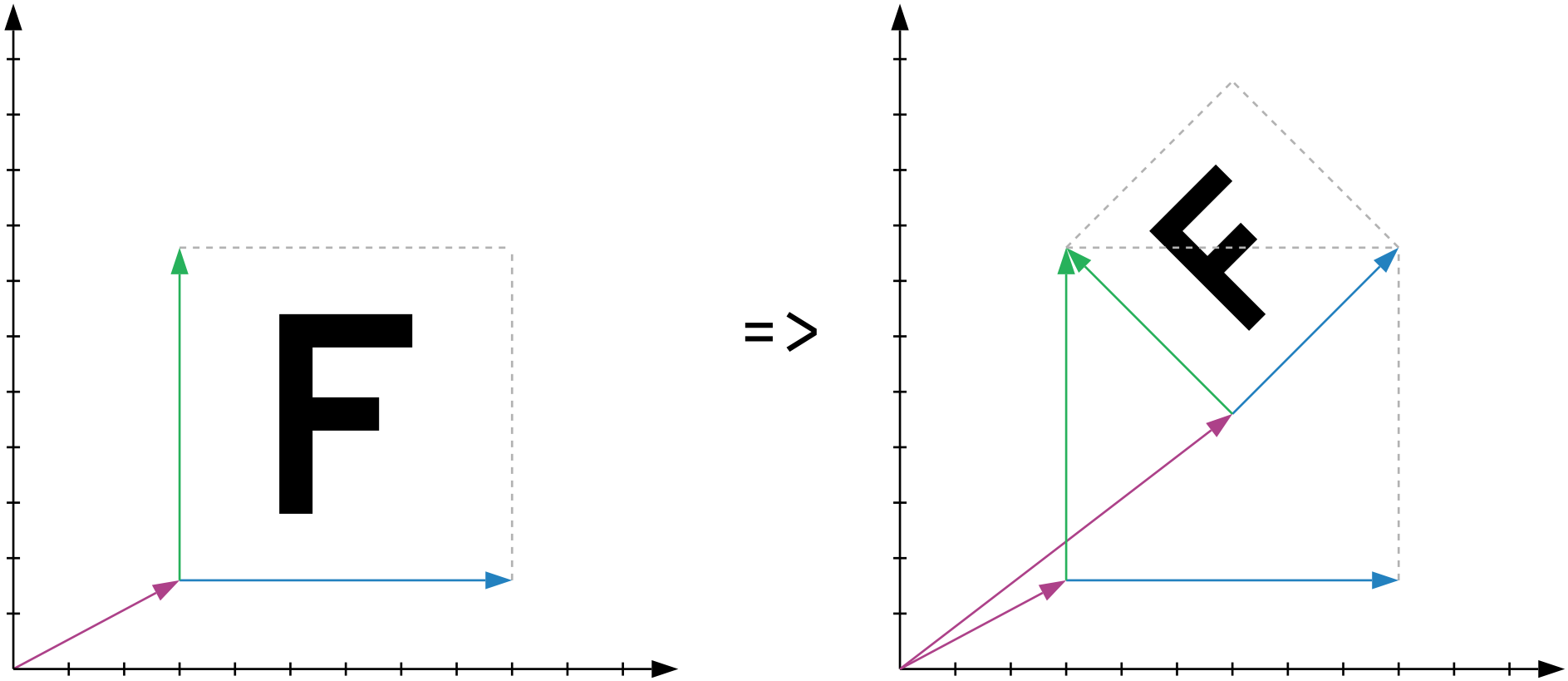
toss



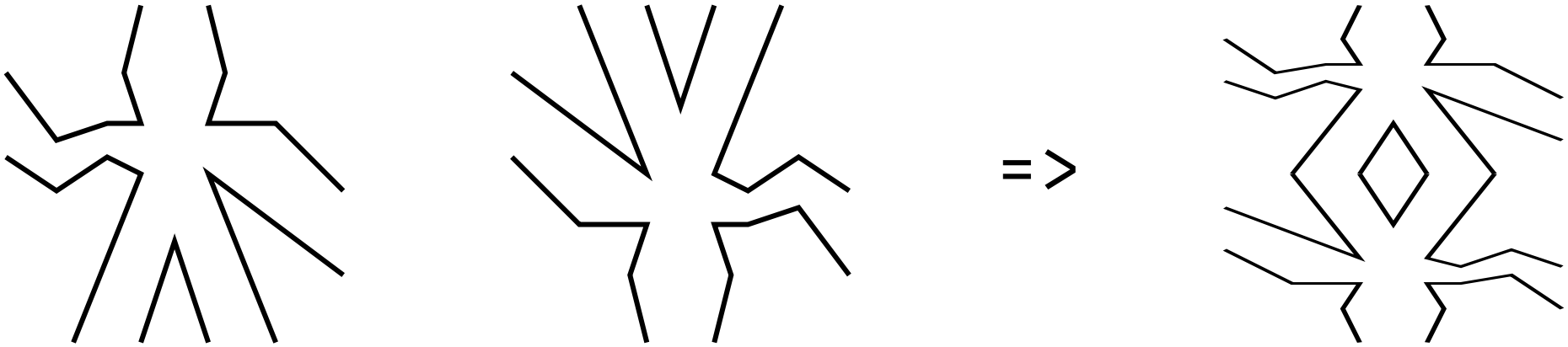
```
tossBox : Box -> Box
tossBox { a, b, c } =
    { a = add a (scale 0.5 (add b c))
    , b = scale 0.5 (add b c)
    , c = scale 0.5 (sub c b) }

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

toss



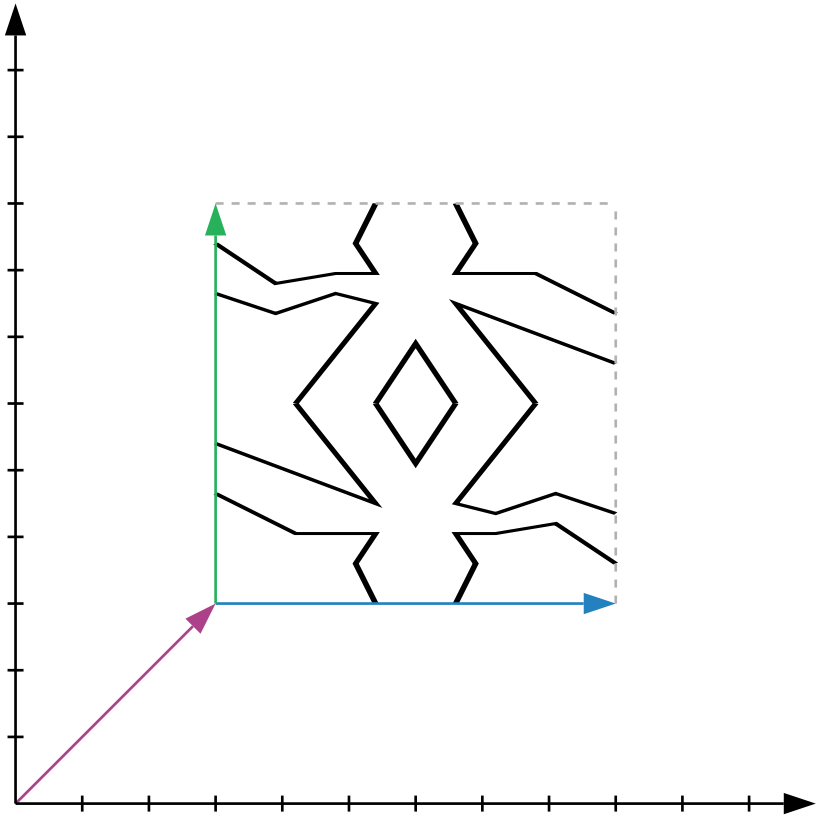
```
above george ((turn >> turn) george)
```



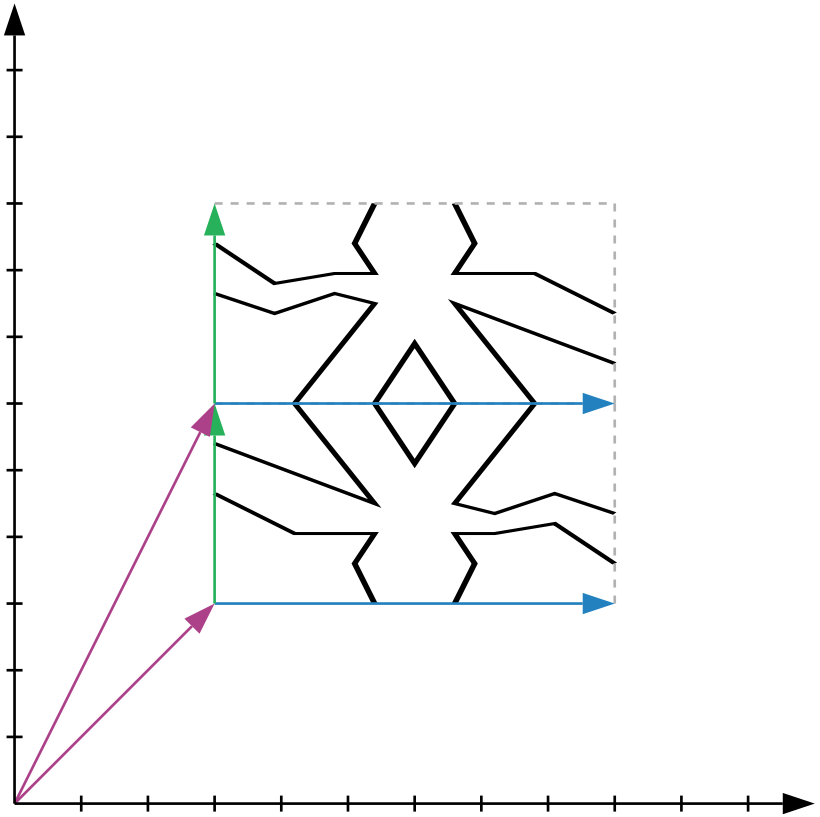
```
aboveRatio : Int -> Int -> Pic -> Pic -> Pic
aboveRatio m n p1 p2 =
    \box ->
        let
            f = m / (m + n)
            (b1, b2) = splitVertically f box
        in
            (p1 b1) ++ (p2 b2)

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

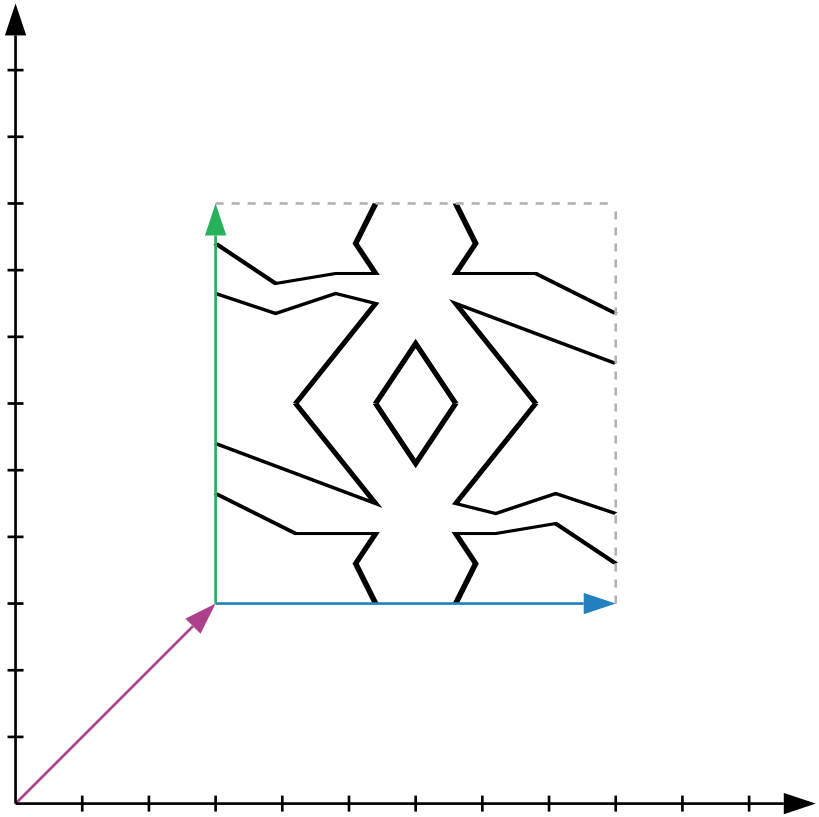
```
above george ((turn >> turn) george)
```



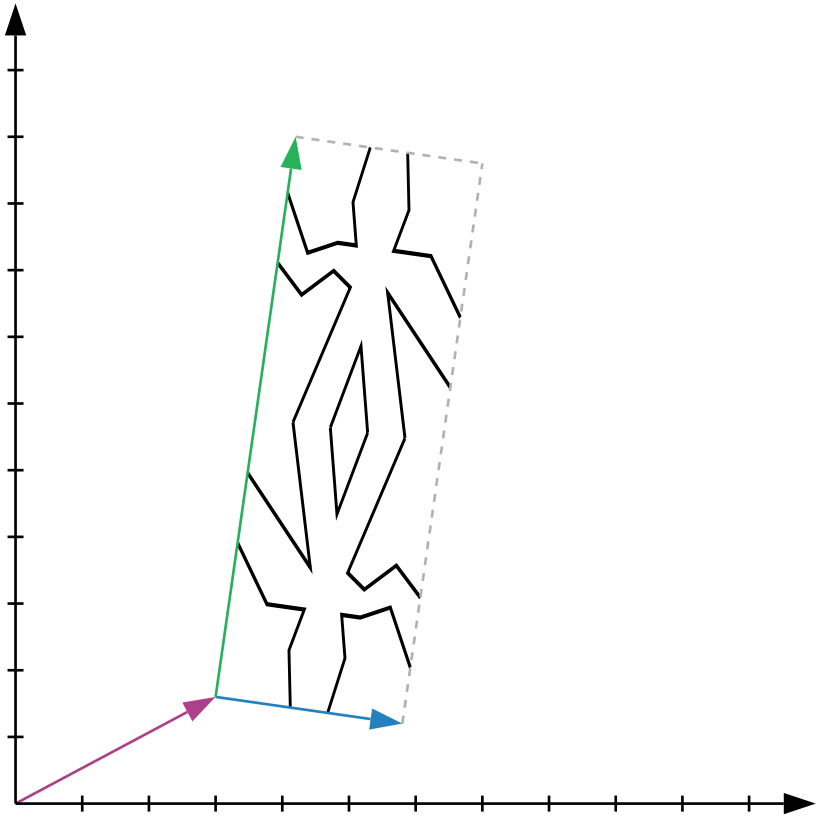
```
above george ((turn >> turn) george)
```



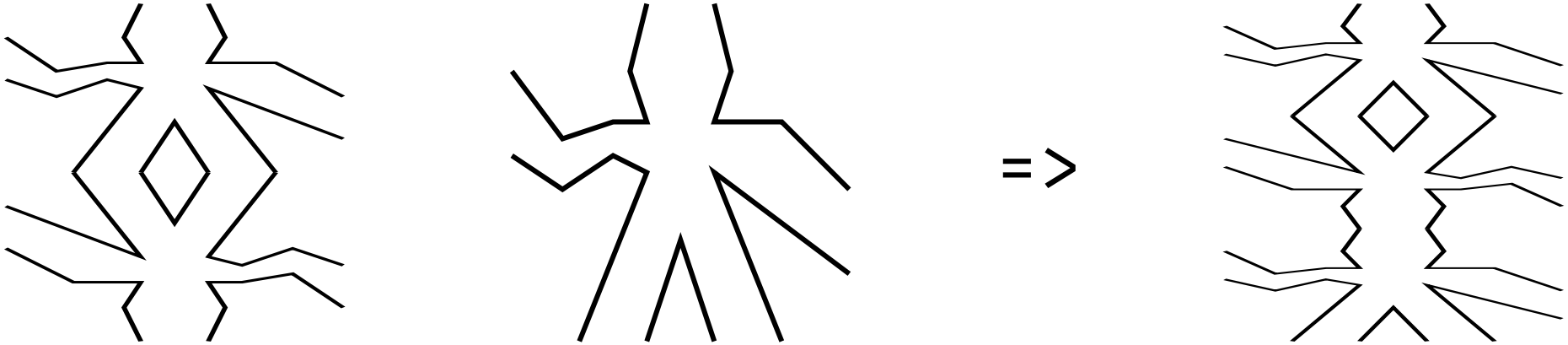
mirrorgeorge



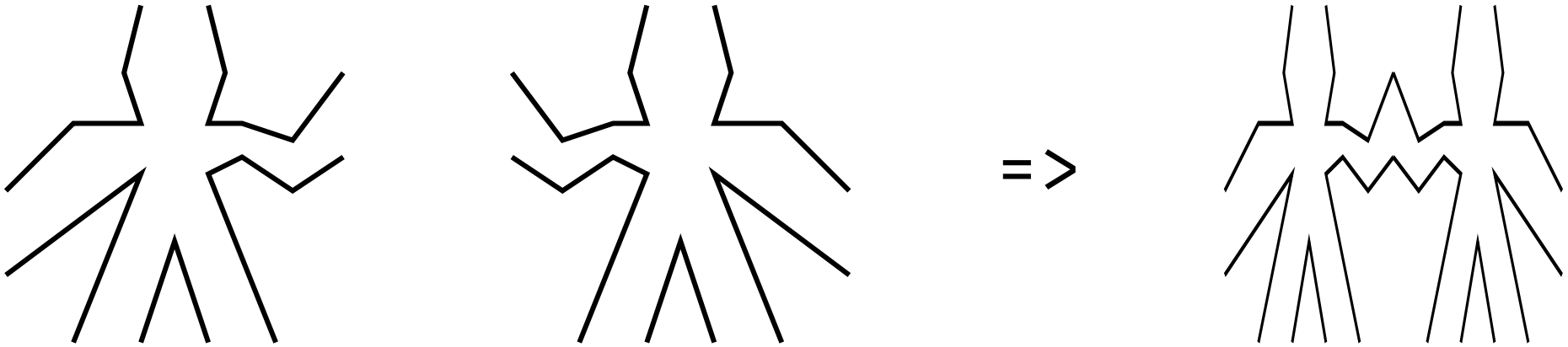
mirrorgeorge



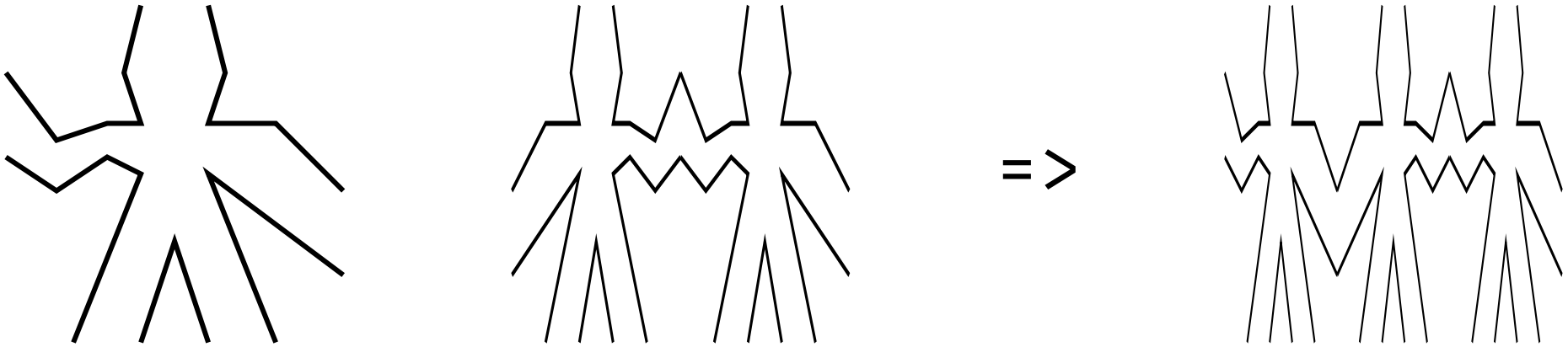
aboveRatio 2 1 mirrorgeorge george



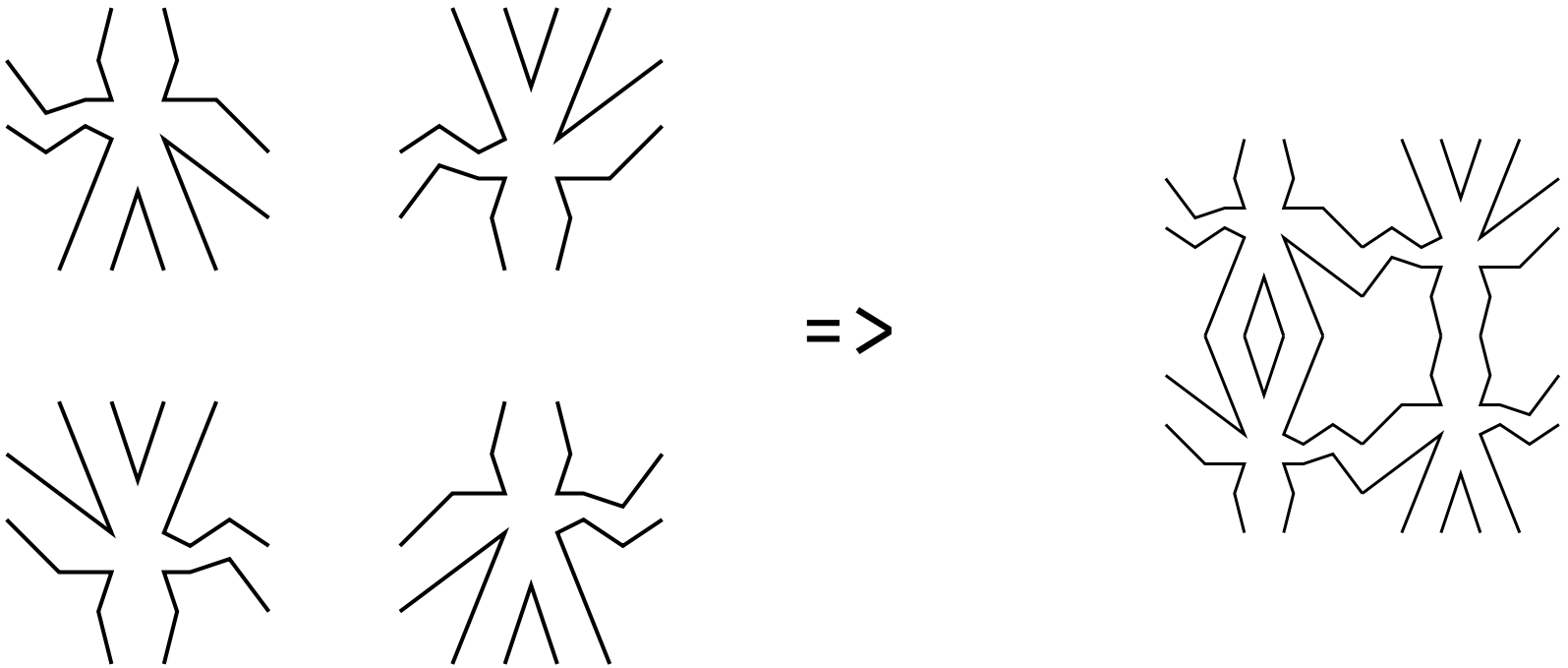
beside (flip george) george



besideRatio 1 2 george twingeorge

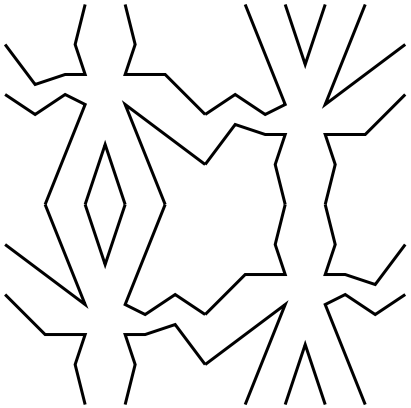


quartet g1 g2 g3 g4

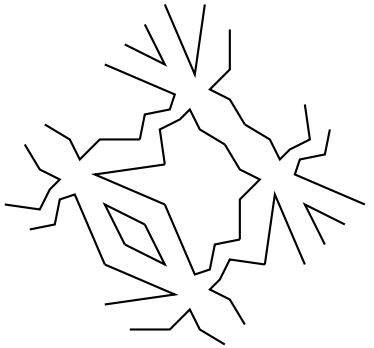


```
quartet : P -> P -> P -> P -> P
quartet nw ne sw se =
    above (beside nw ne)
          (beside sw se)
```

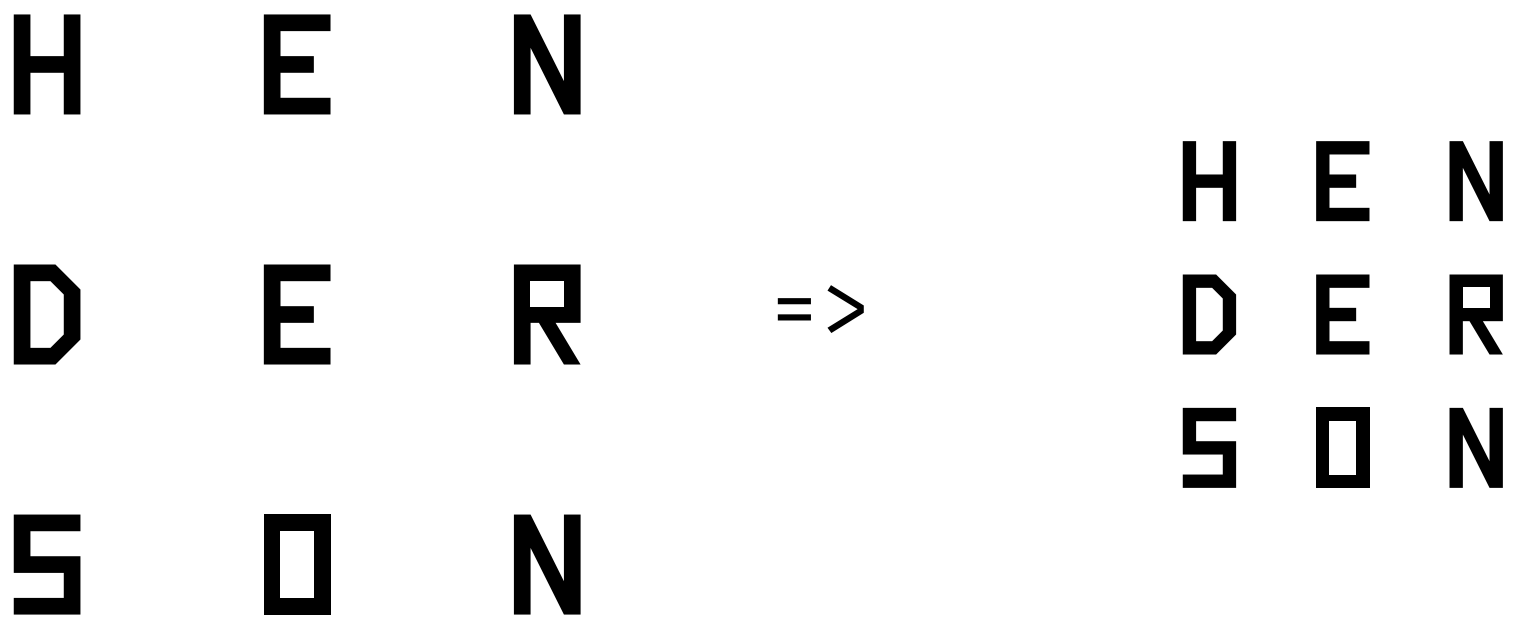
toss



=>



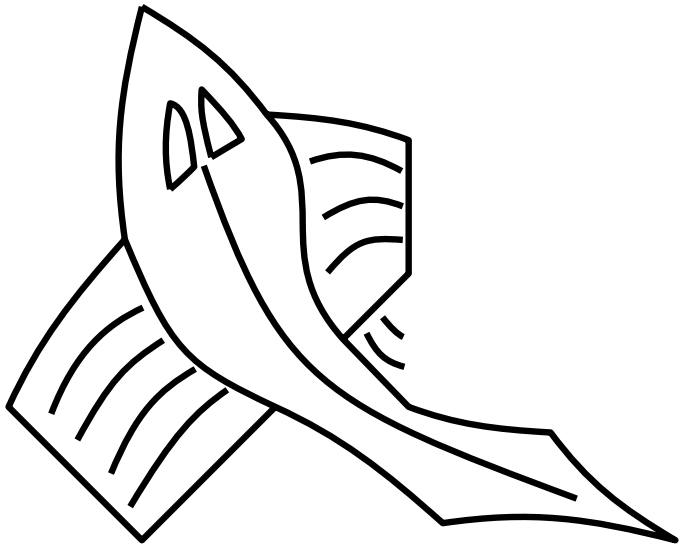
nonet h e n d e r s o n



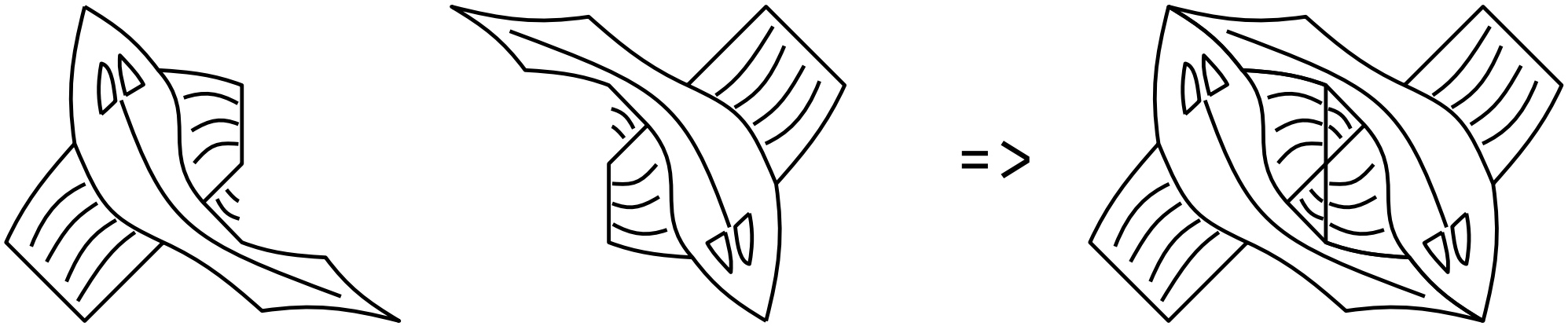
```
nonet : P -> P -> P -> P -> P -> P -> P -> P -> P -> P
    let
        row w m e = besideRatio 1 2 w (beside m e)
        col n m s = aboveRatio 1 2 n (above m s)
    in
        col (row nw nm ne)
            (row nw nm ne)
            (row nw nm ne)
```

nonets are just pictures

a fish picture

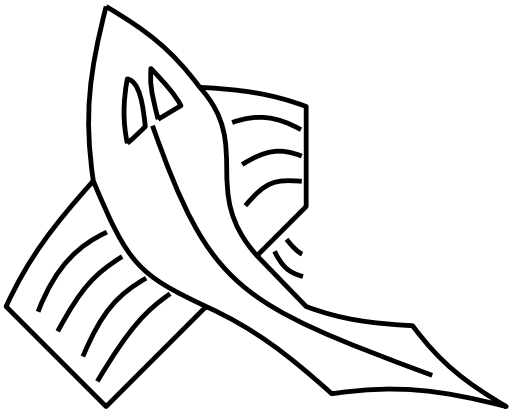


```
over fish ((turn >> turn) fish)
```

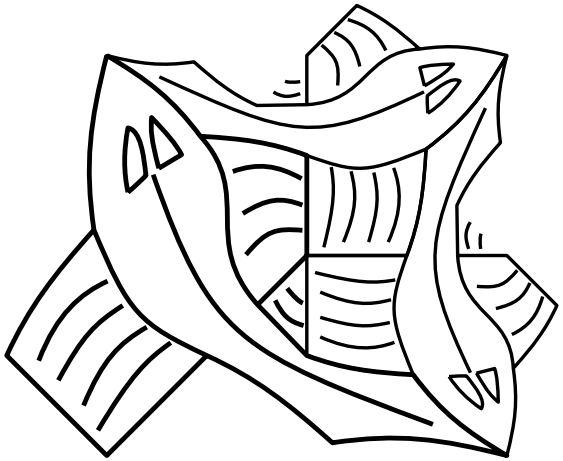


```
over : Pic -> Pic -> Pic
over p1 p2
     \box -> p1 box ++ p2 box
```

ttile

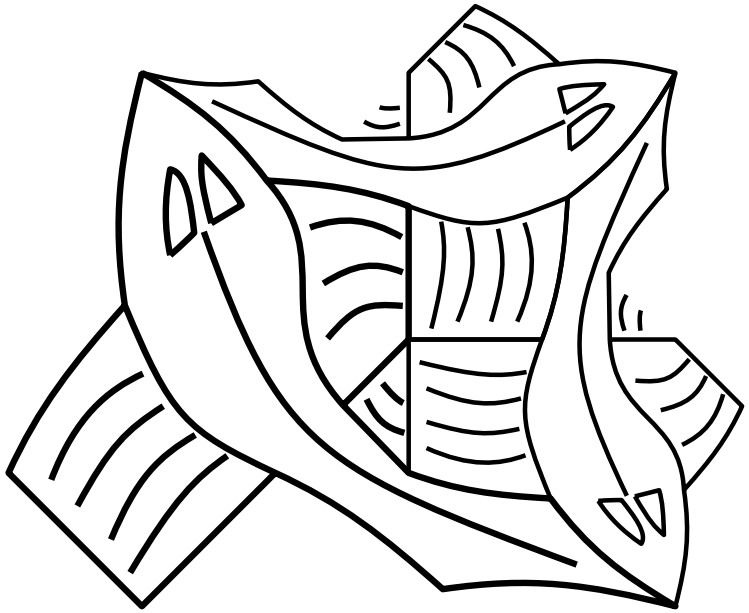


=>

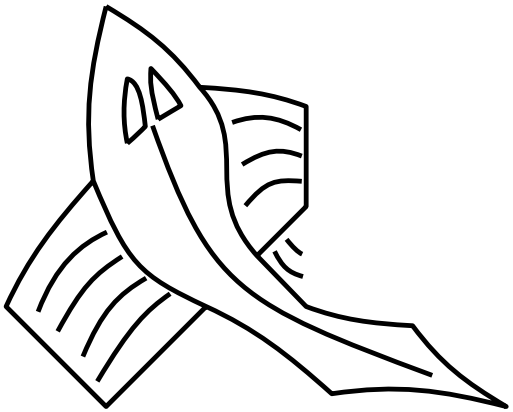


```
ttile : Picture -> Picture
ttile p =
    let
        pn = (toss >> flip) p
        pe = (turn >> turn >> turn) p
    in
        over p (over pn pe)
```

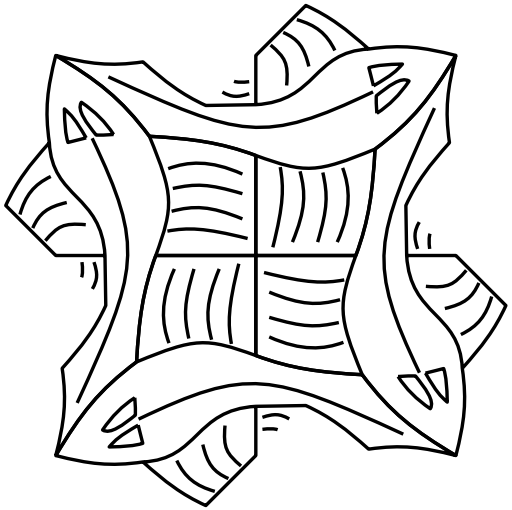
ttile



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