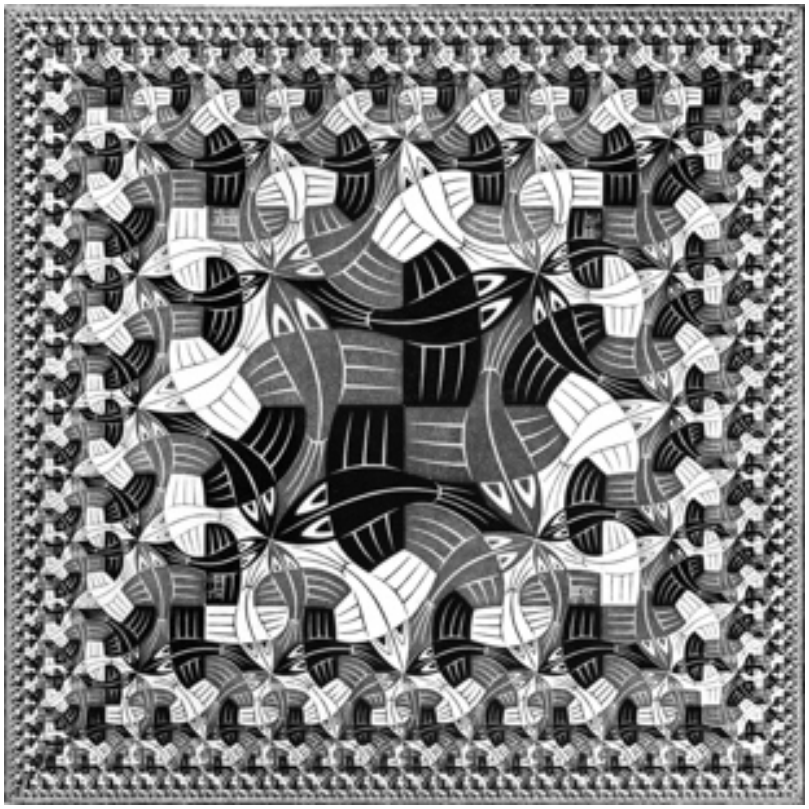


# Functional Geometry

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.

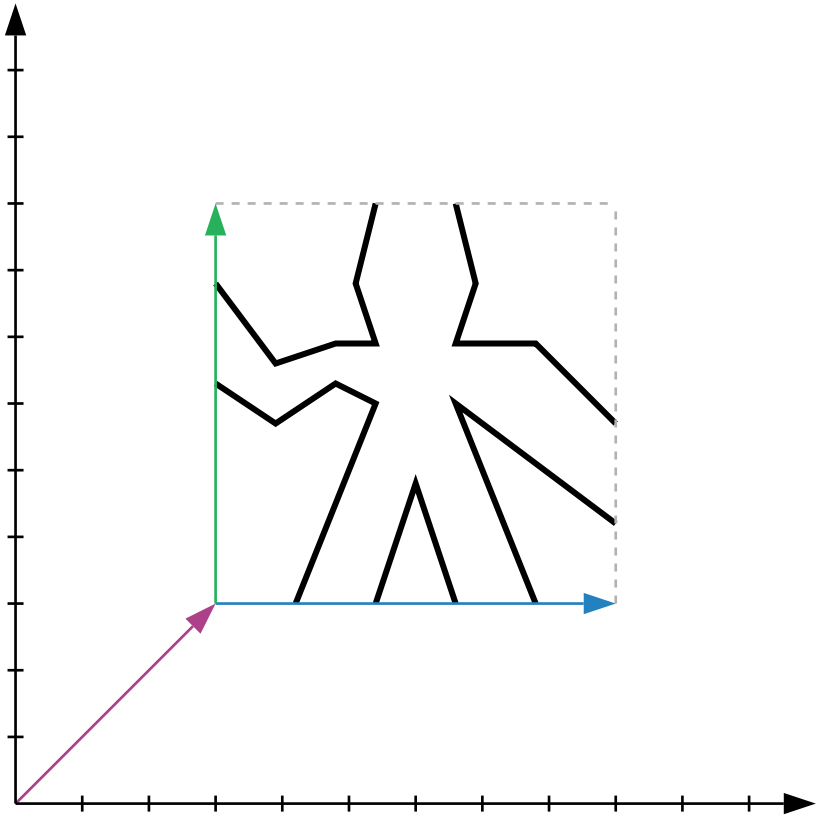
**Keywords:** Functional programming, graphics, geometry, algebraic style, architecture, specification.

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

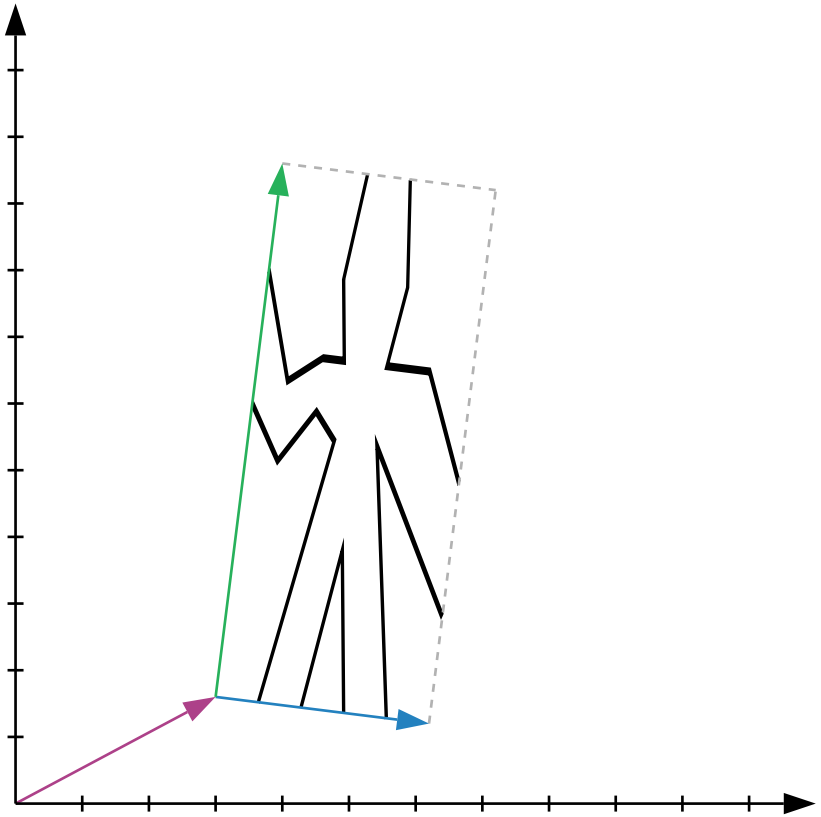
Let us define a picture as a function which takes three arguments, each being two-space vectors and returns a set of graphical objects to be rendered on the output device.

```
Picture : Box -> Rendering
```

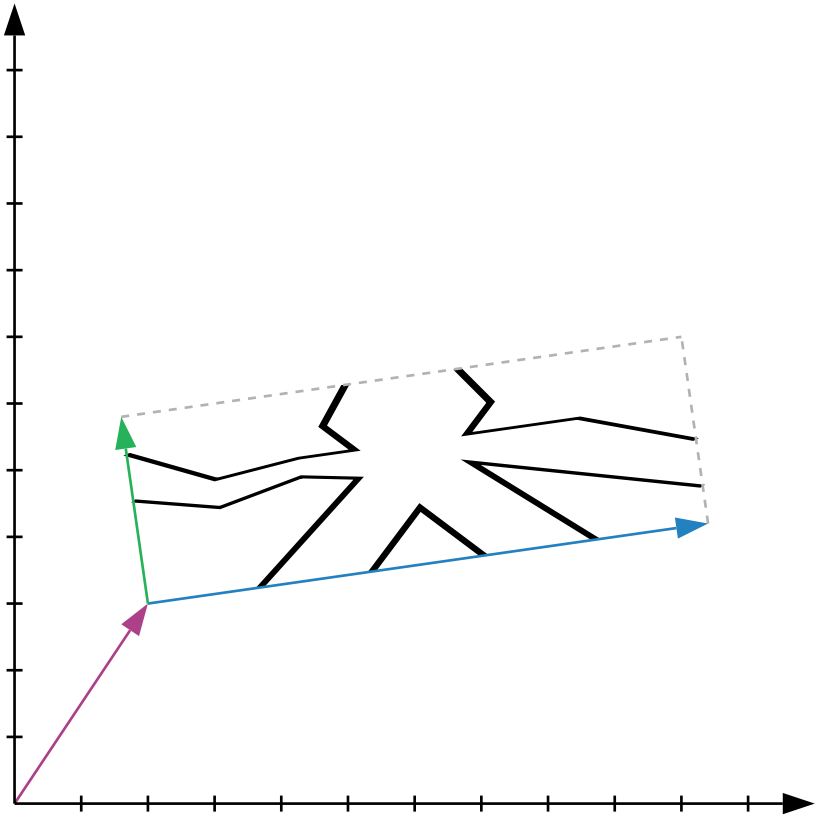
# george



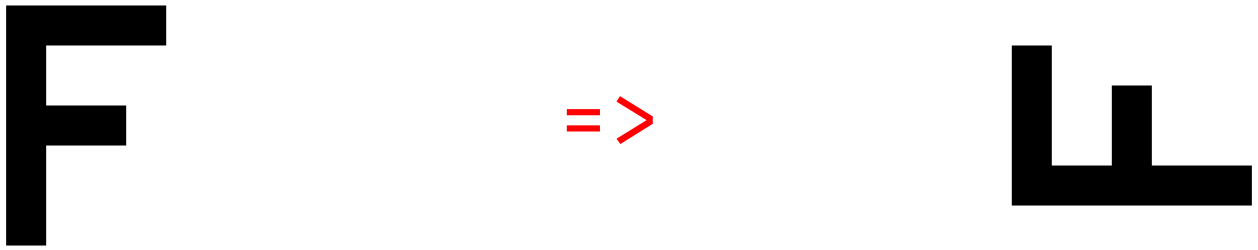
# also george



# still george



turn



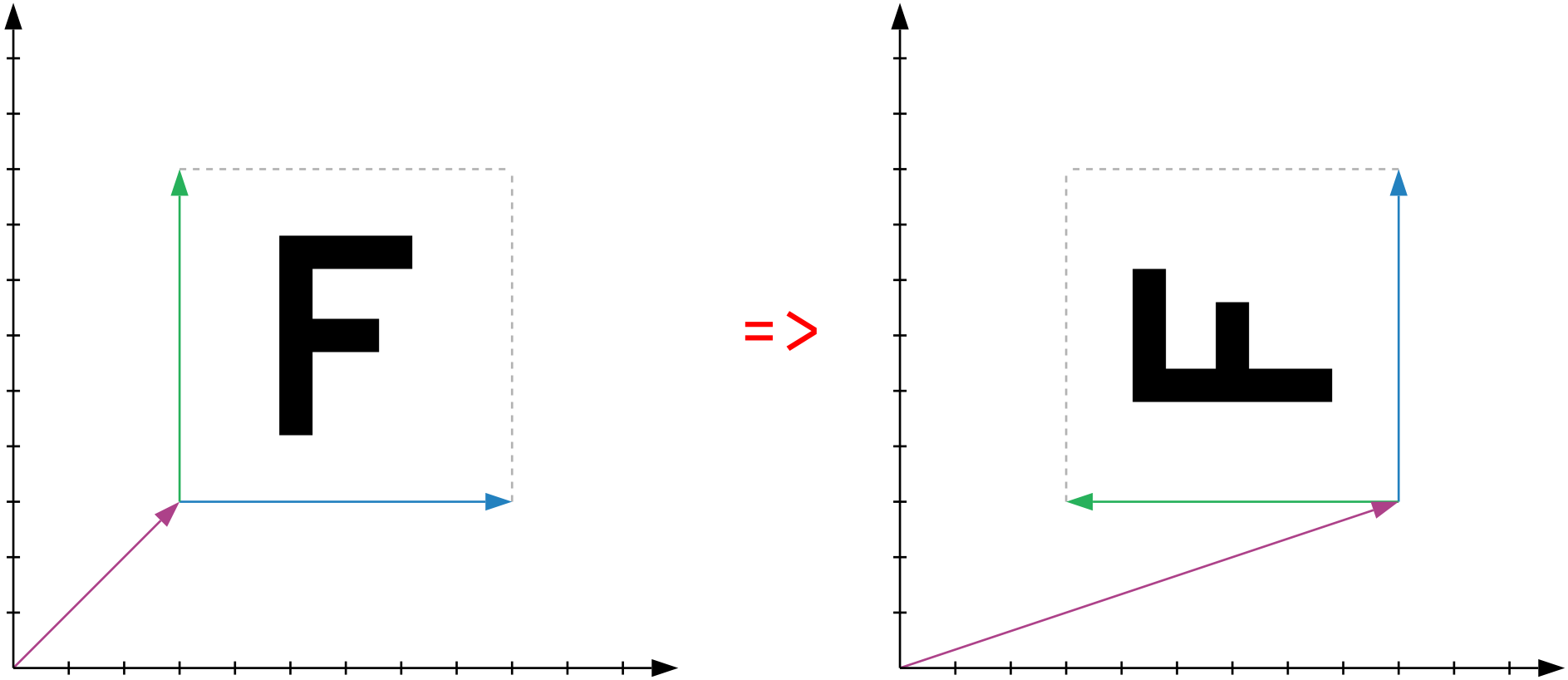
```
turn p (a,b,c) = p (a+b,c,-b)
```

```
turn p (a,b,c) = p (a+b,c,-b)
```

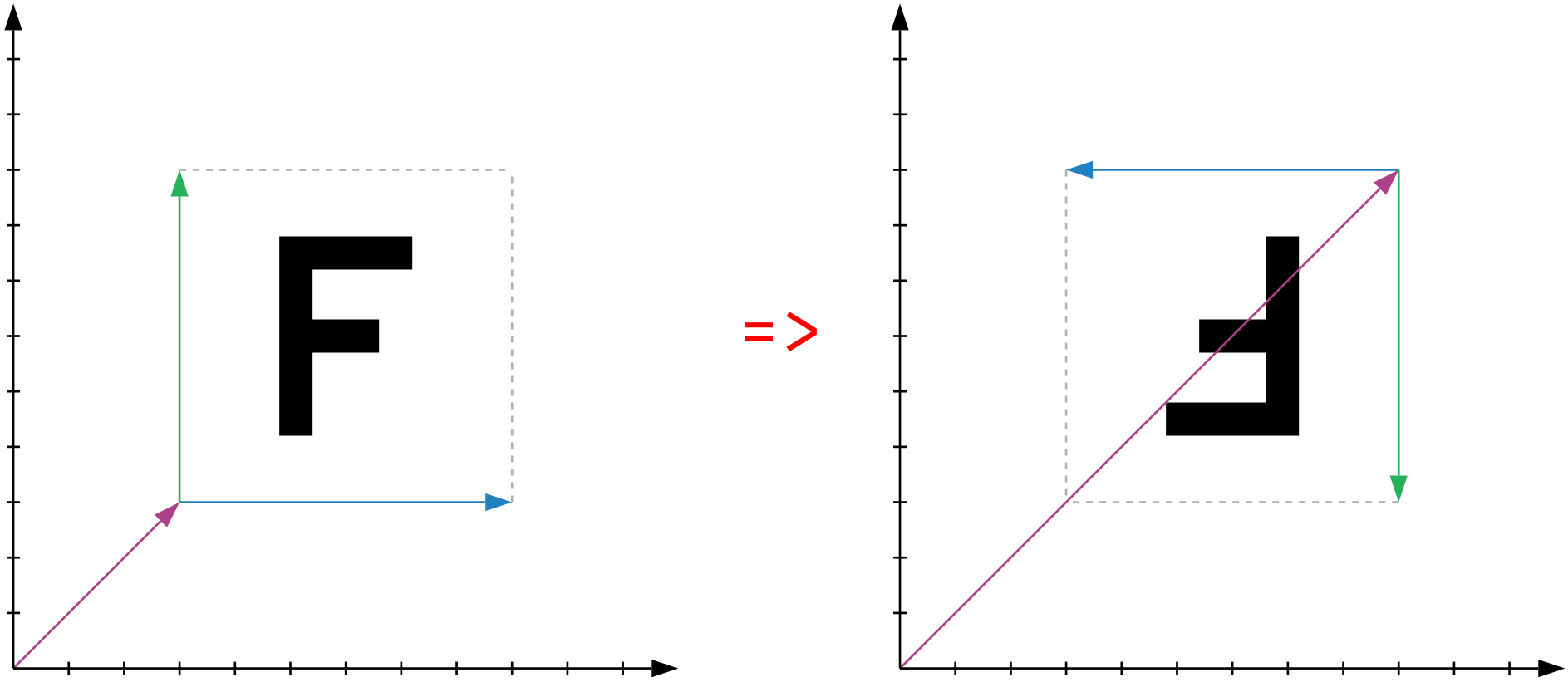
PSEUDOCODE

```
turnbox (a,b,c) = (a+b,c,-b)

turn p = p . turnbox
```

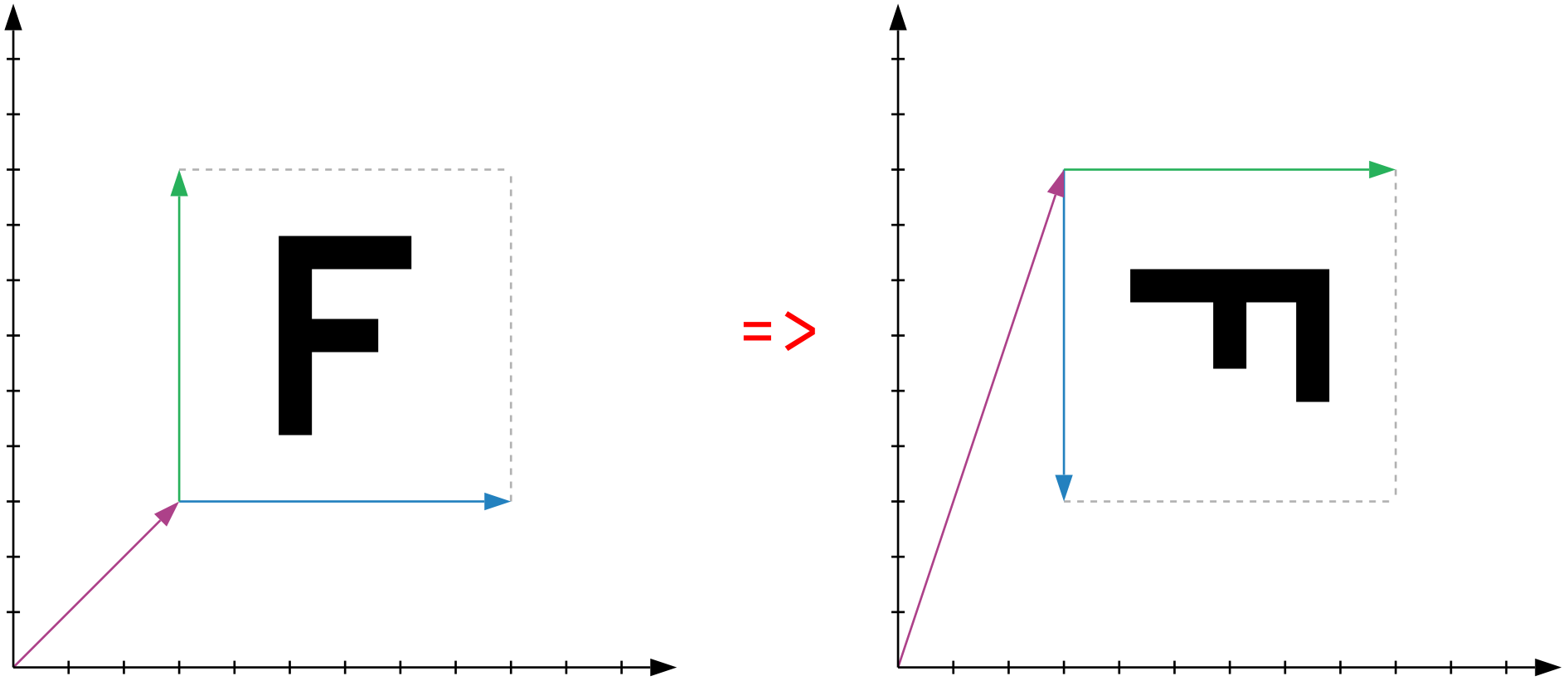
# turn



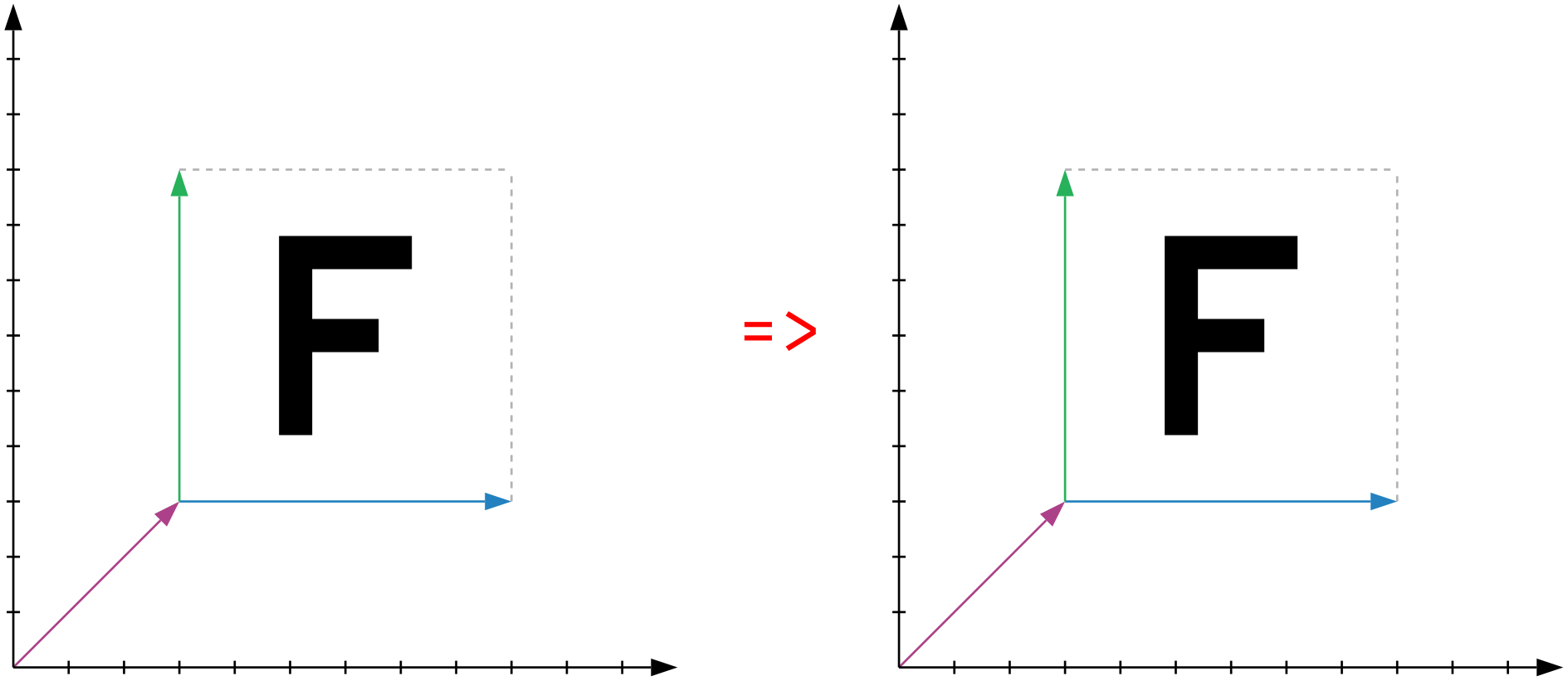
# turn . turn



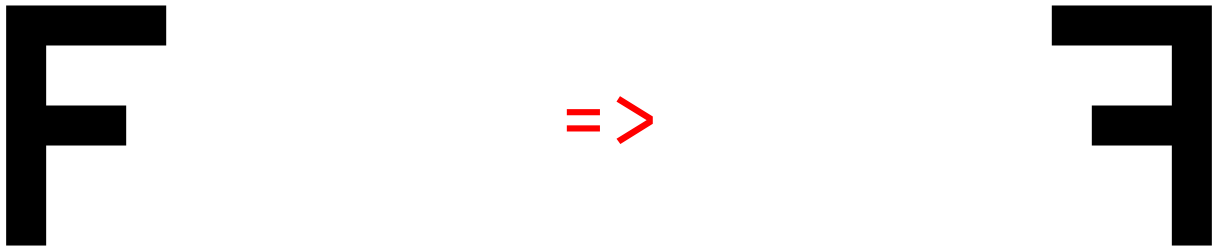
# turn . turn . turn



```
turn . turn . turn . turn
```

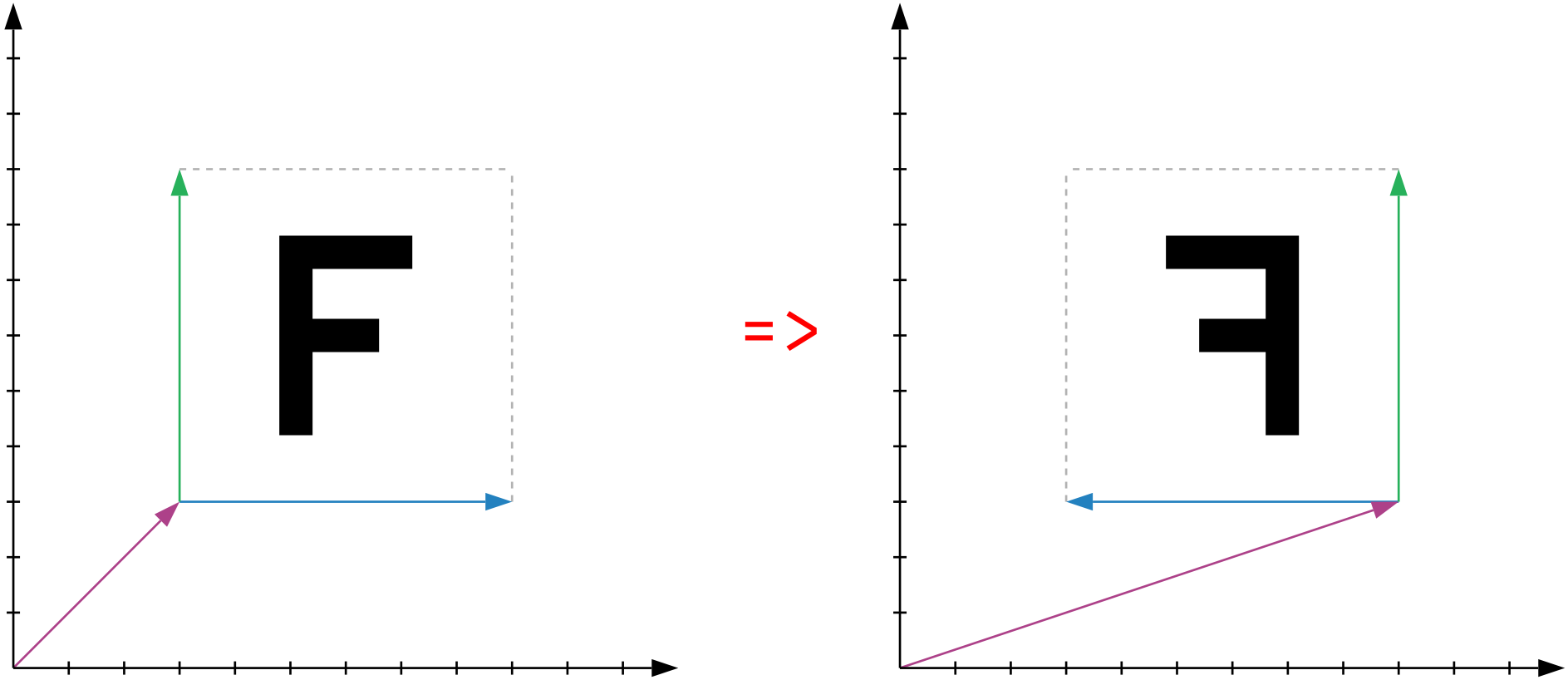


# flip

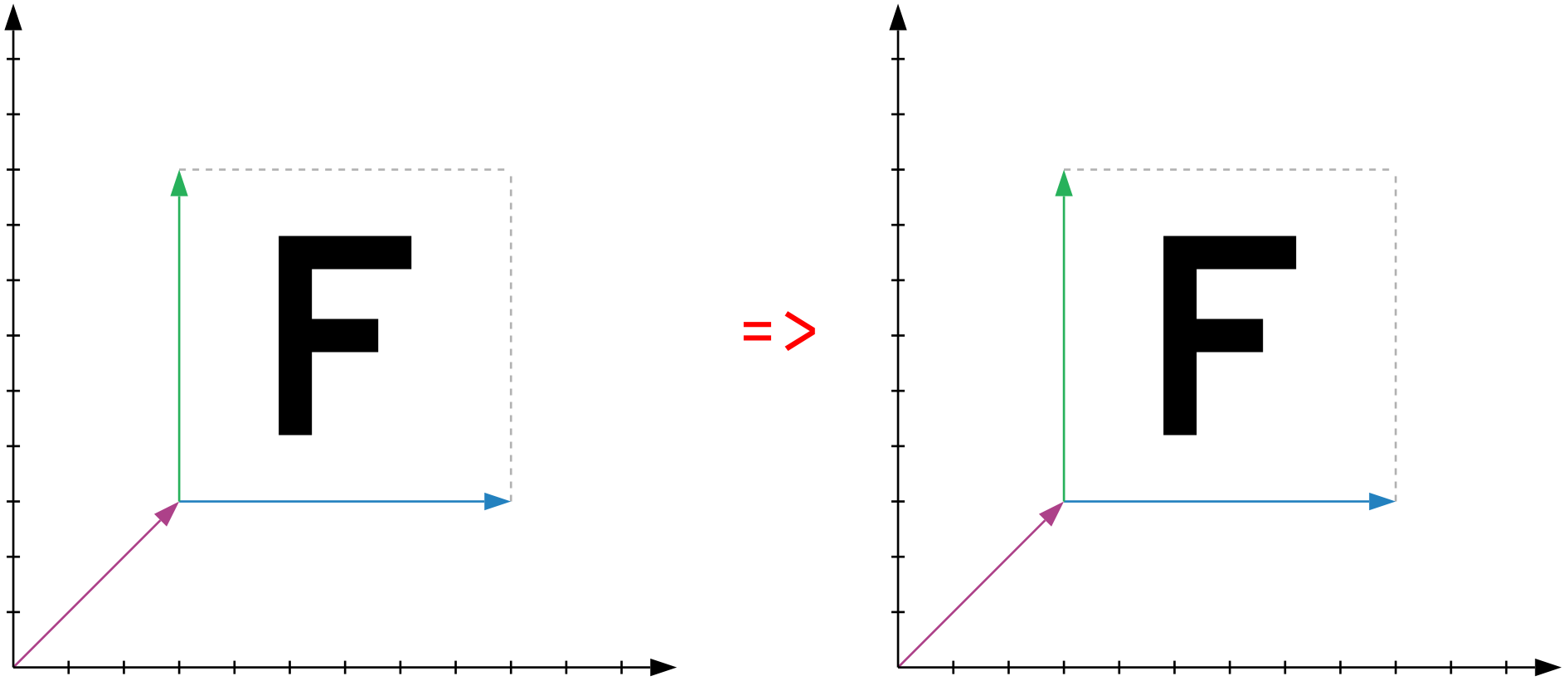


```
flipbox (a,b,c) = (a+b,-b,c)

flip p = p . flipbox
```

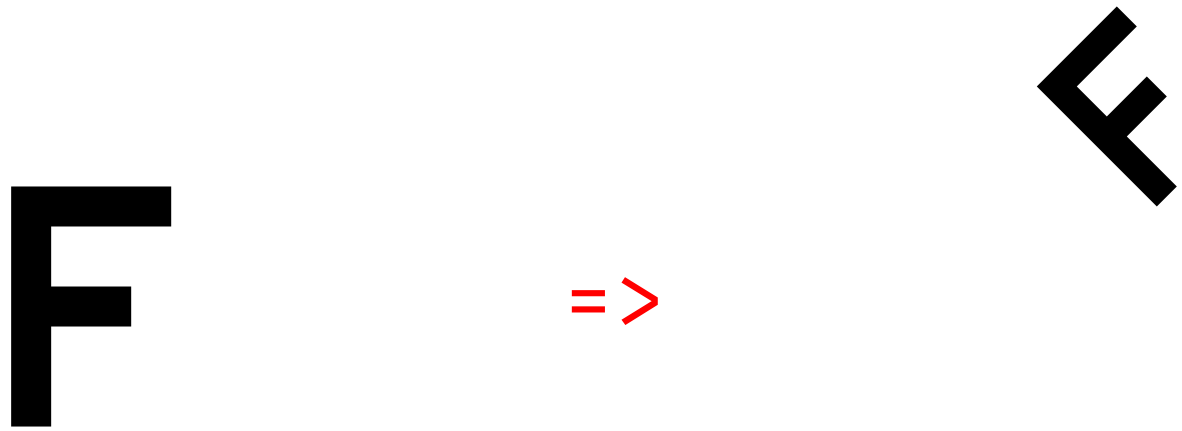
# flip



# flip . flip

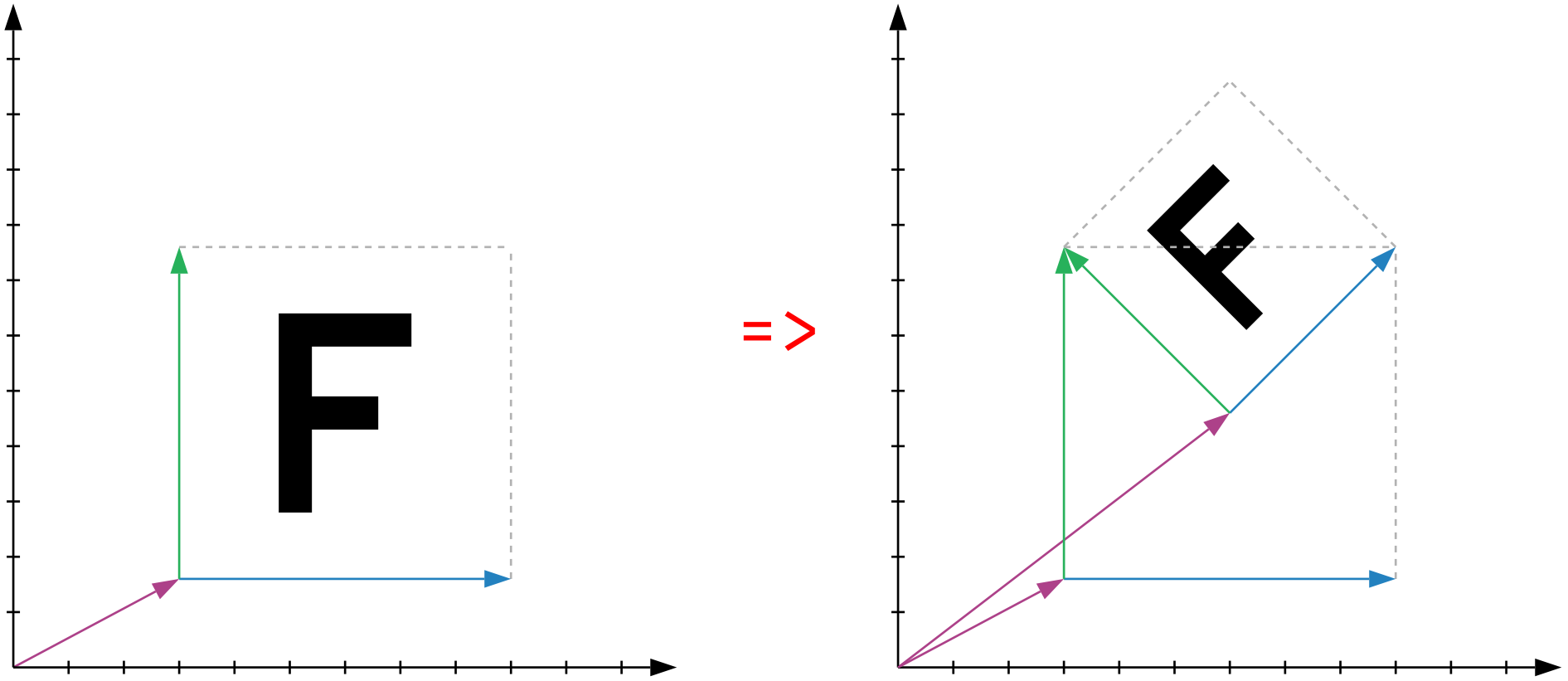


# toss

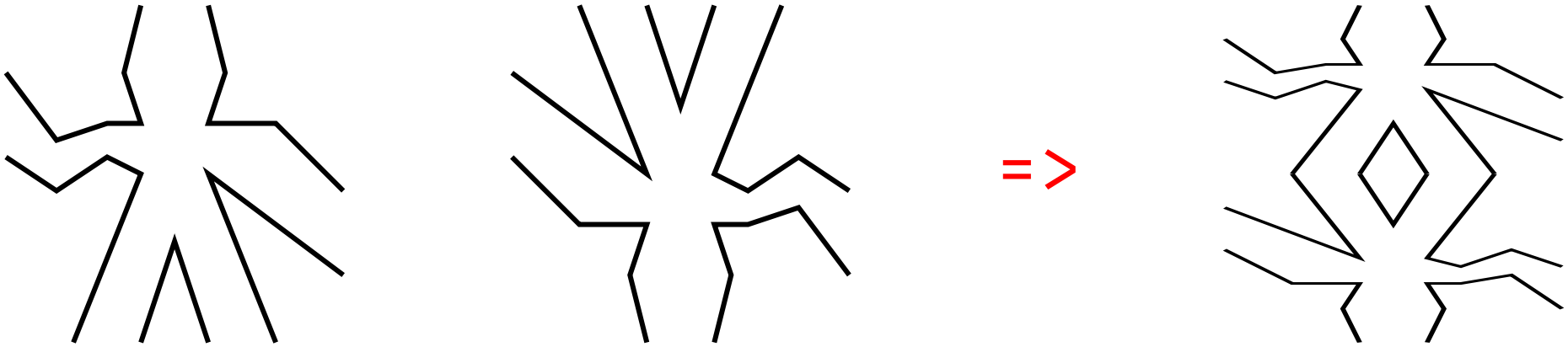


```
tossbox (a,b,c) =
  (a+(b+c)/2,(b+c)/2,(c-b)/2)

toss p = p . tossbox
```

# toss

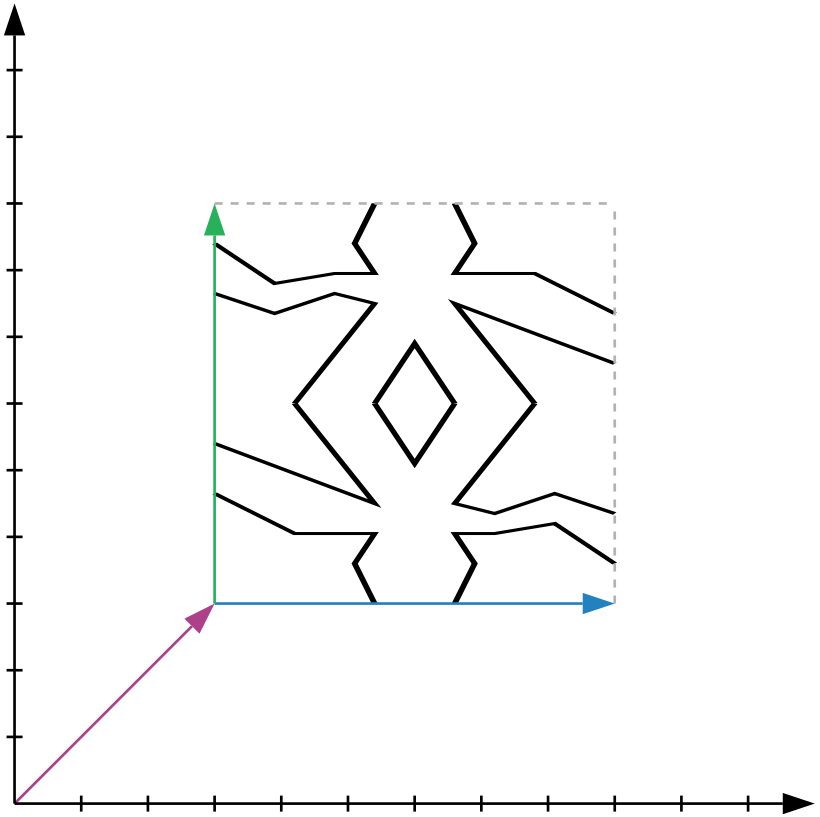


```
above george ((turn . turn) george)
```

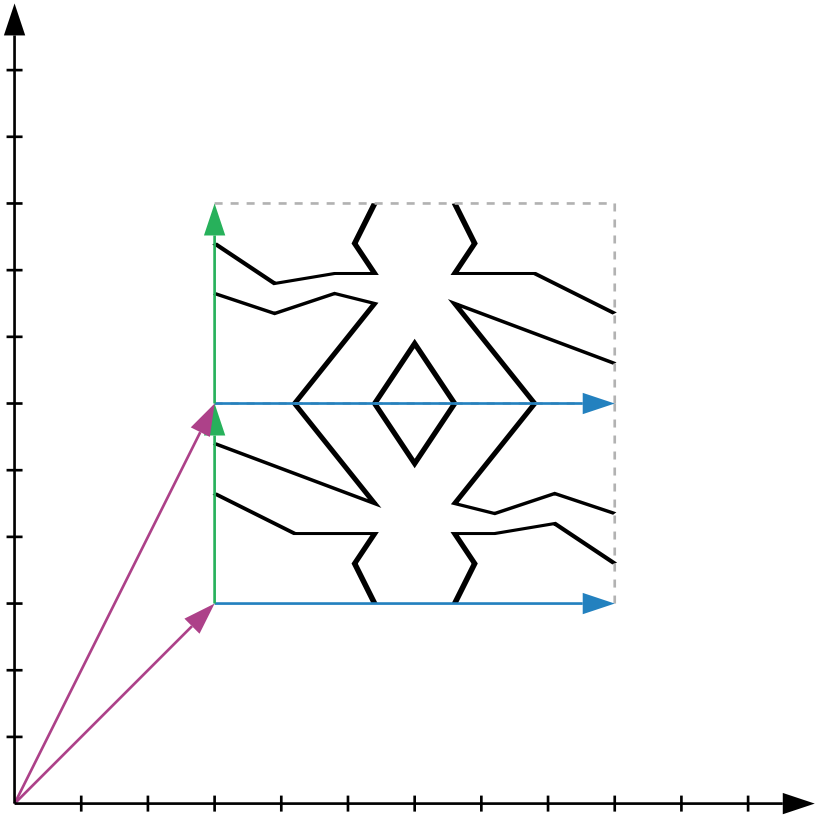


```
aboveratio m n p q (a,b,c) =
    p (a+c*n/(m+n),b,c*m/(m+n)) @
    q (a,b,c*n/(m+n))

above = aboveratio 1 1
```

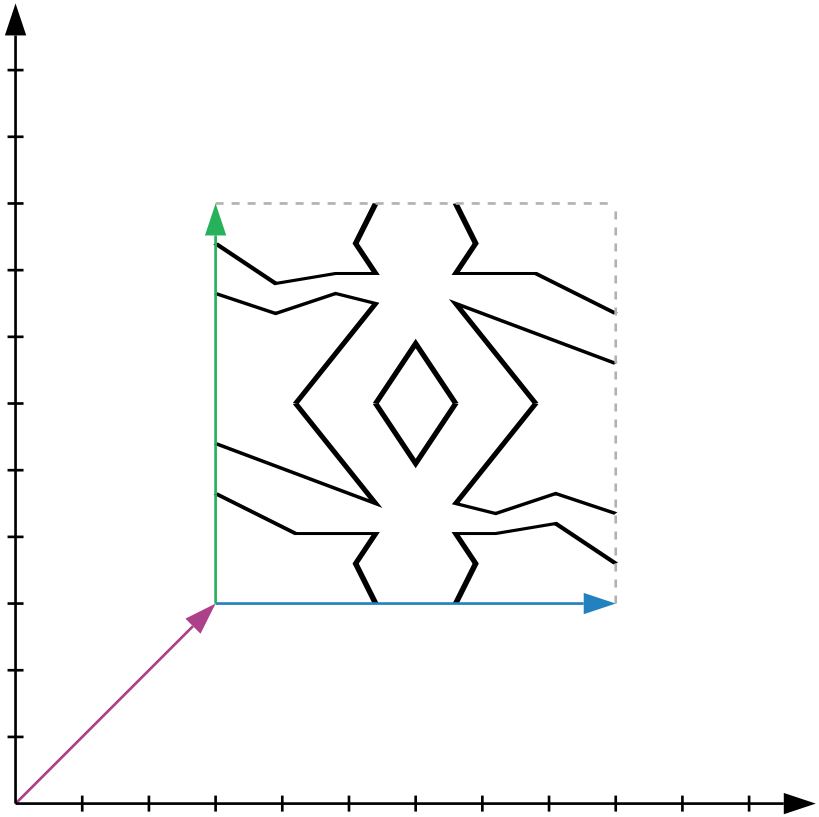
```
above george ((turn . turn) george)
```



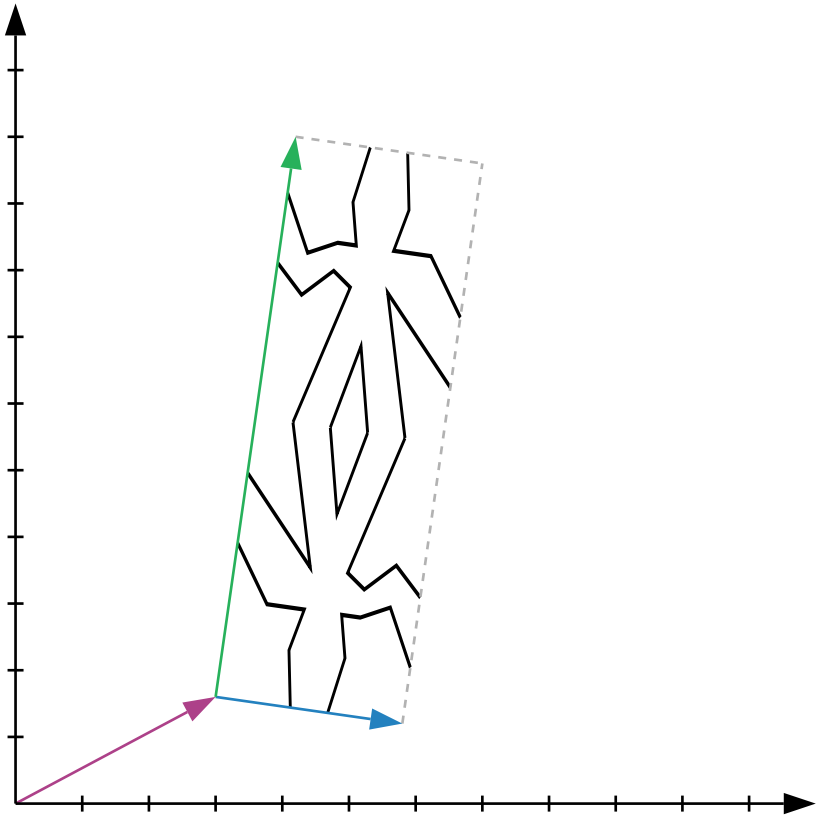
```
above george ((turn . turn) george)
```



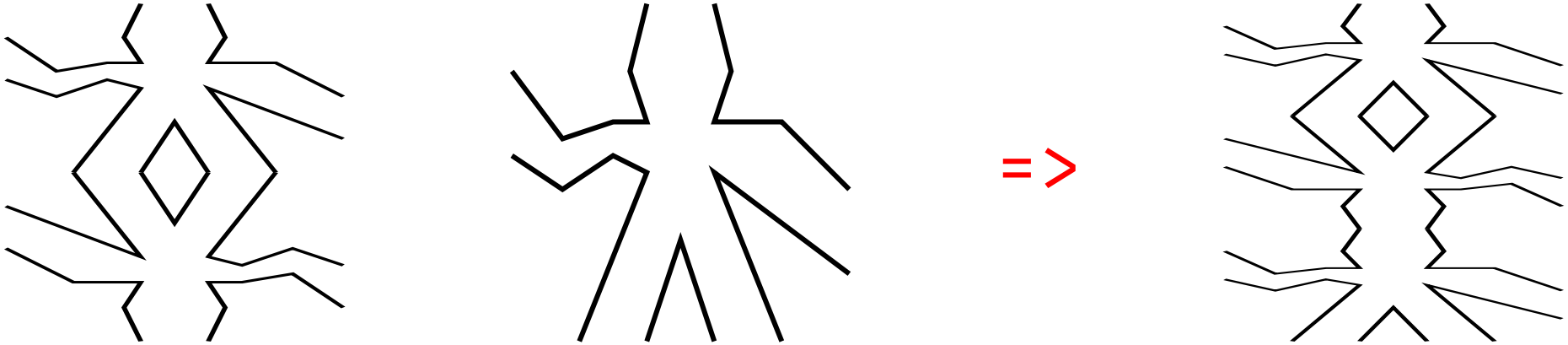
# mirrorgeorge



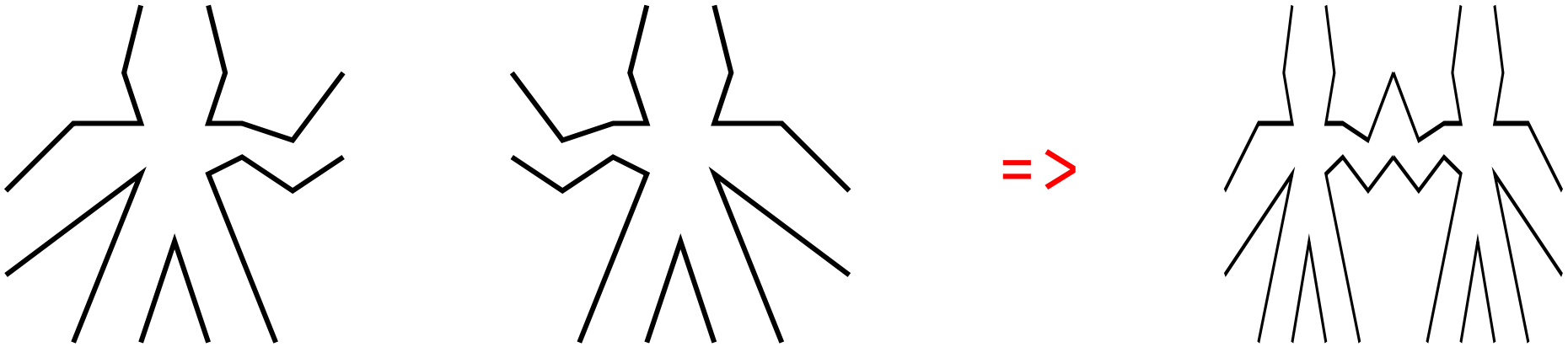
# mirrorgeorge



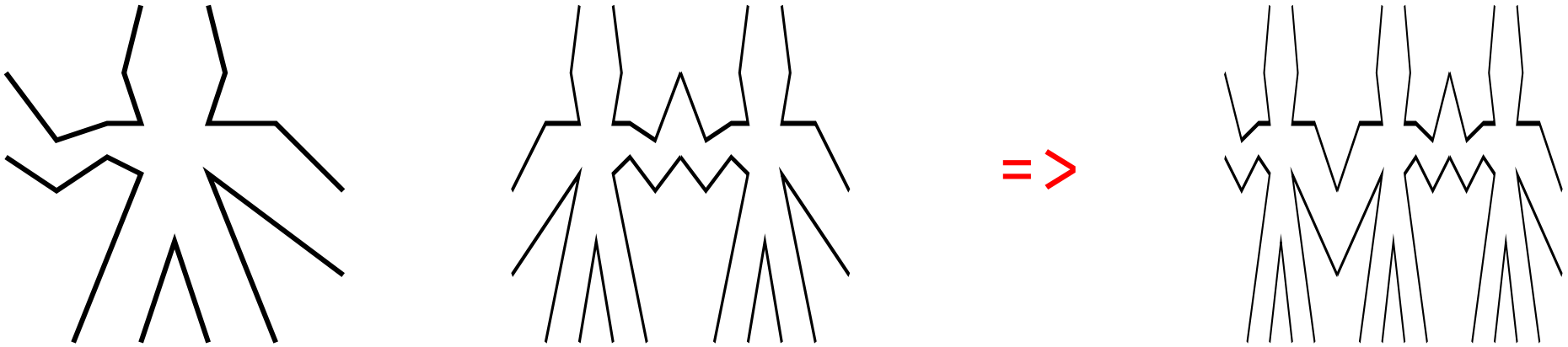
# aboveratio 2 1 mirrorgeorge george



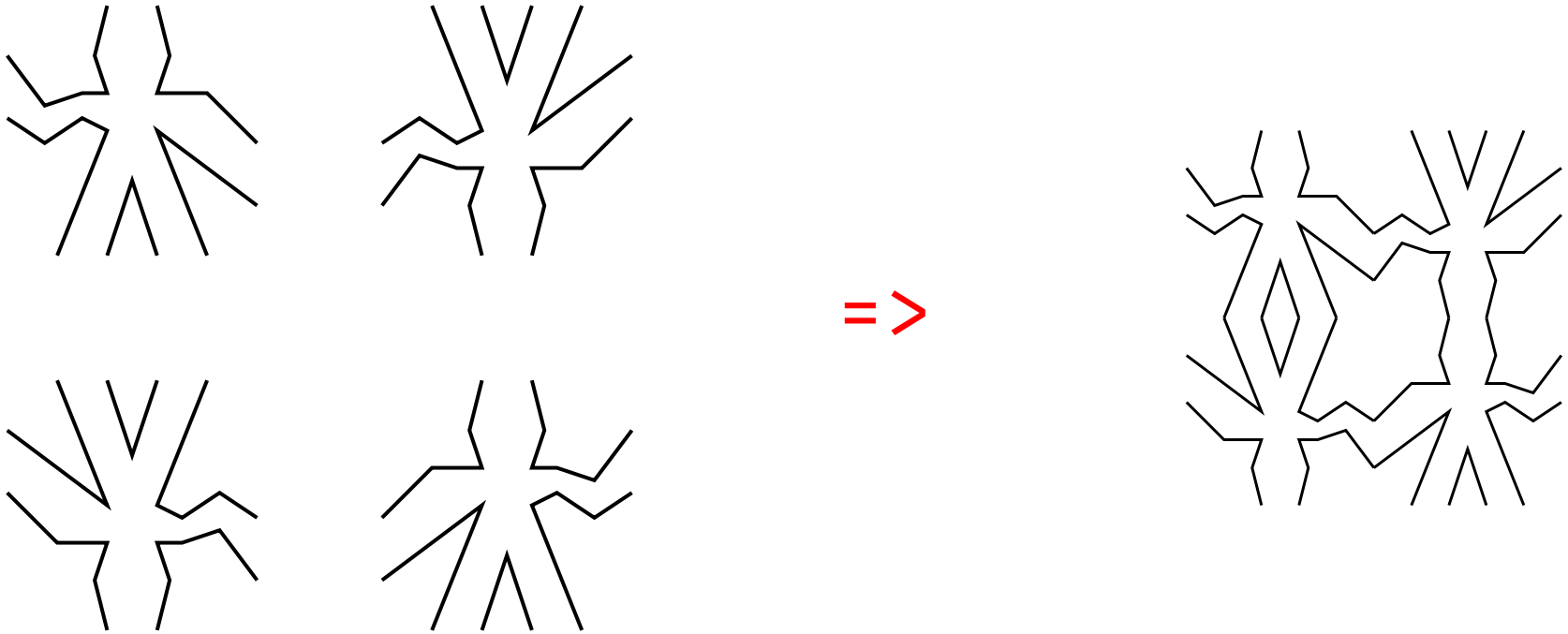
# beside (flip george) george



besideratio 1 2 george twingeorge

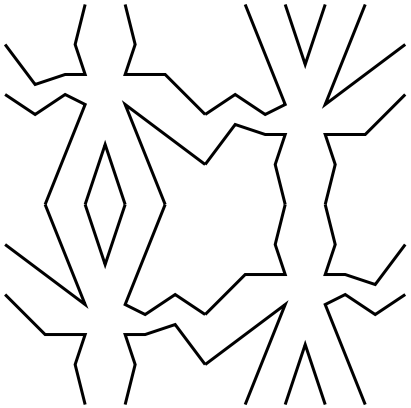


```
quartet g1 g2 g3 g4
```

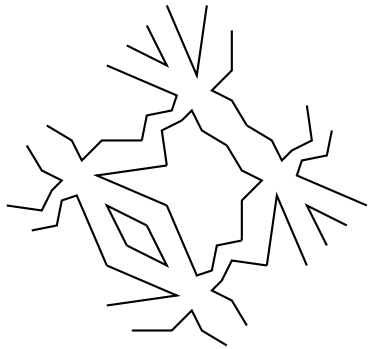


```
quartet nw ne sw se =
    above (beside nw ne)
          (beside sw se)
```

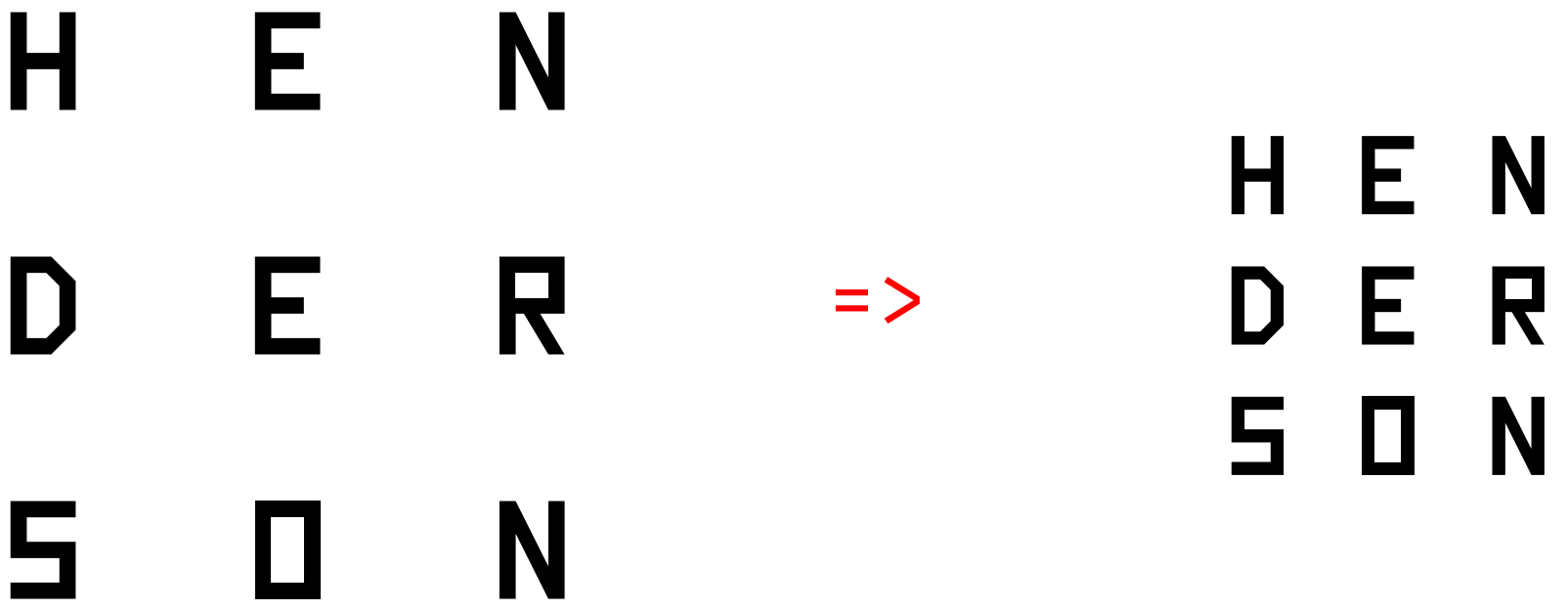
# toss



=>



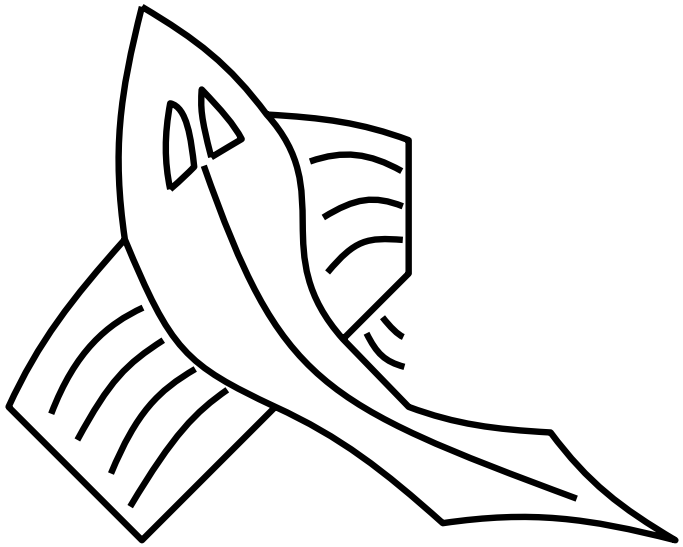
nonet h e n d e r s o n



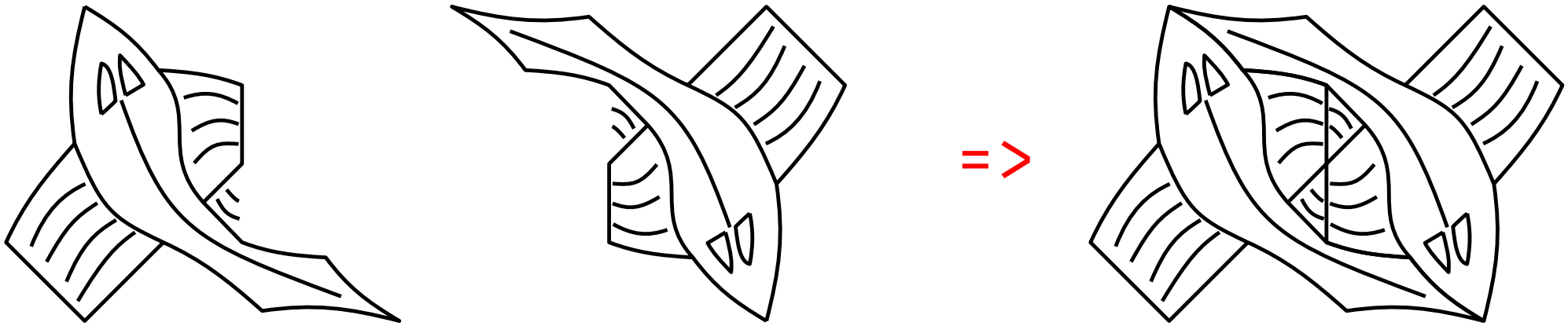
```
row w m e =
    besideratio 1 2 w (beside m e)

col n m s =
    aboveratio 1 2 n (above m s)

nonet nw nm ne mw mm me sw sm se =
    col (row nw nm ne)
        (row nw nm ne)
        (row nw nm ne)
```

# nonets are just pictures

# a fish picture

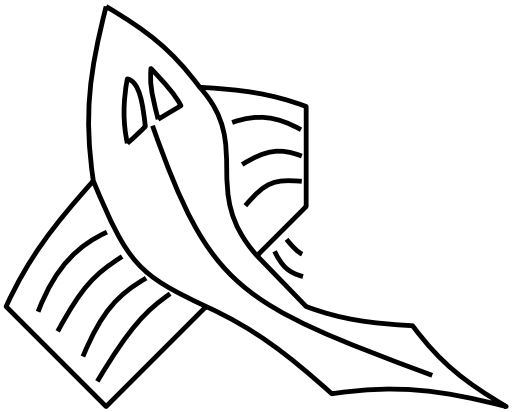


```
over fish ((turn . turn) fish)
```

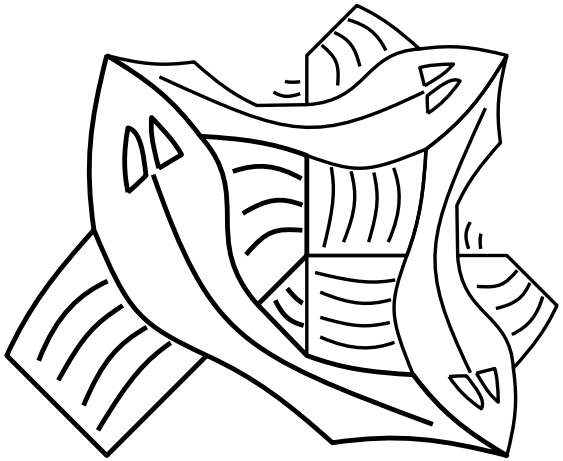


```
over p q = p (a,b,c) @ q (a,b,c)
```

# ttile

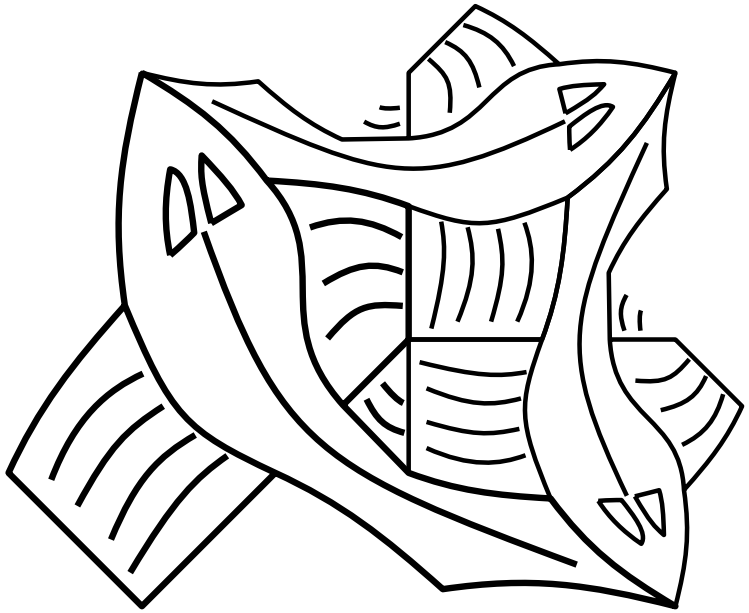


=>

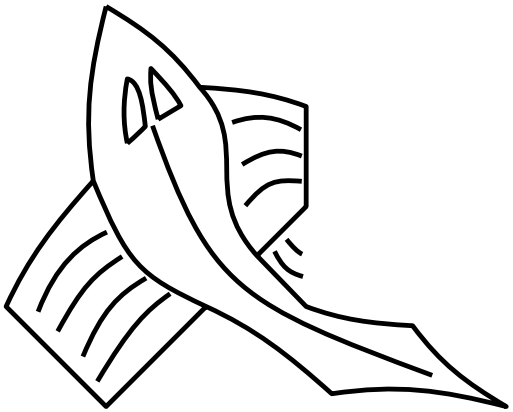


```
times 0 _ = id
times n fn = fn . (times (n-1) fn)

unturn = times 3 turn

ttile p =
    over p
         (over ((flip . toss) p)
               ((unturn . flip . toss) p))
```

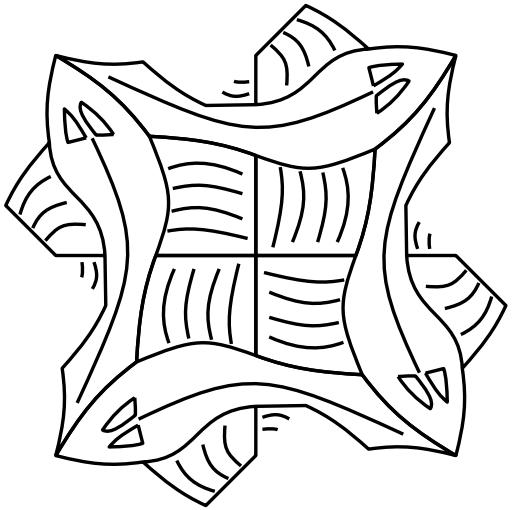
# ttile



# utile

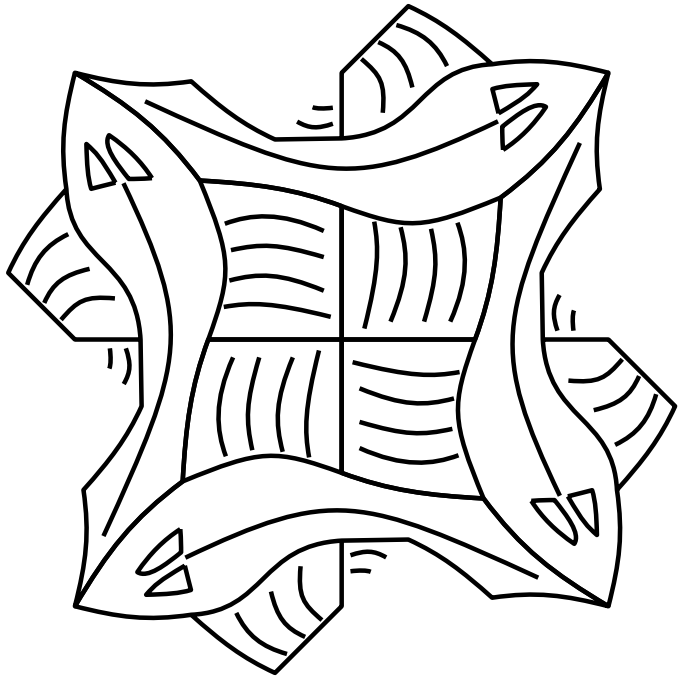


=>

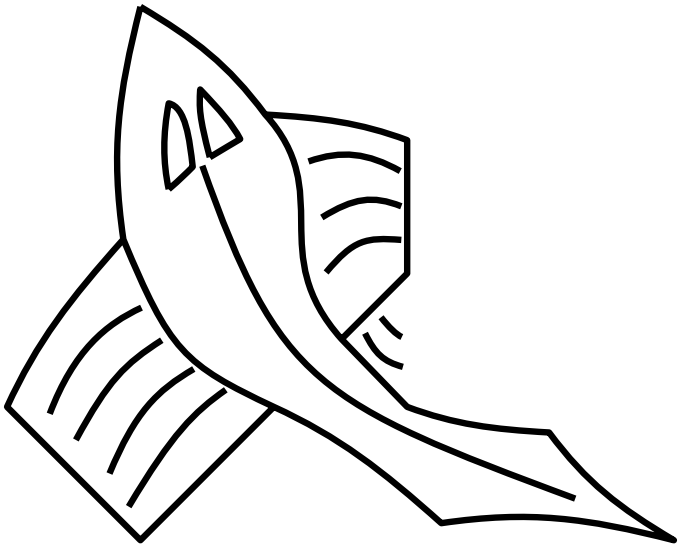


```
utile p =
    over ((flip . toss) p)
         (over ((turn . flip . toss) p)
               (over ((turn . turn . flip . toss) p)
                     ((unturn . flip . toss) p)))
```

# utile

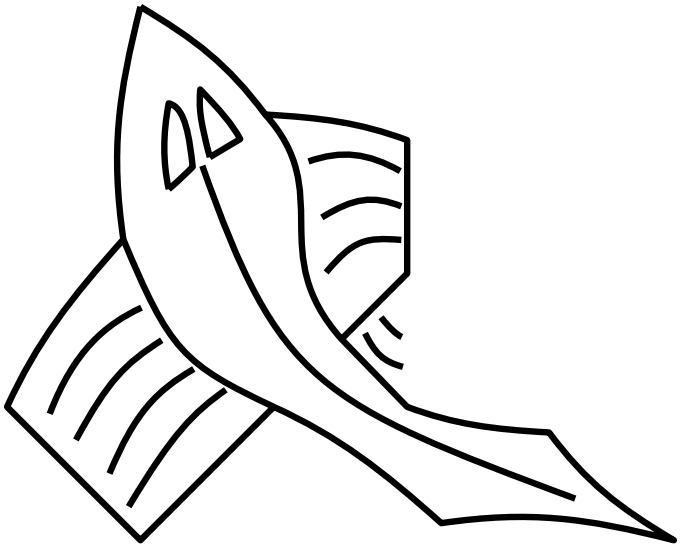


# side 0

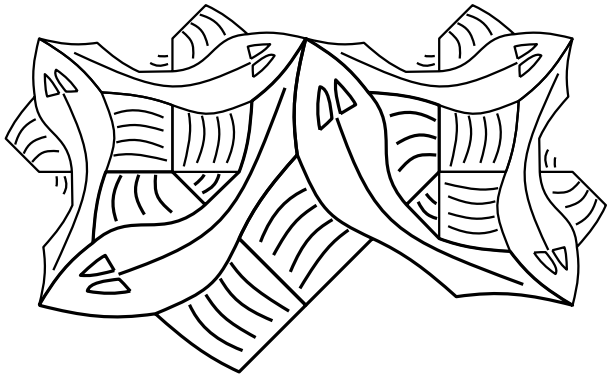


=>

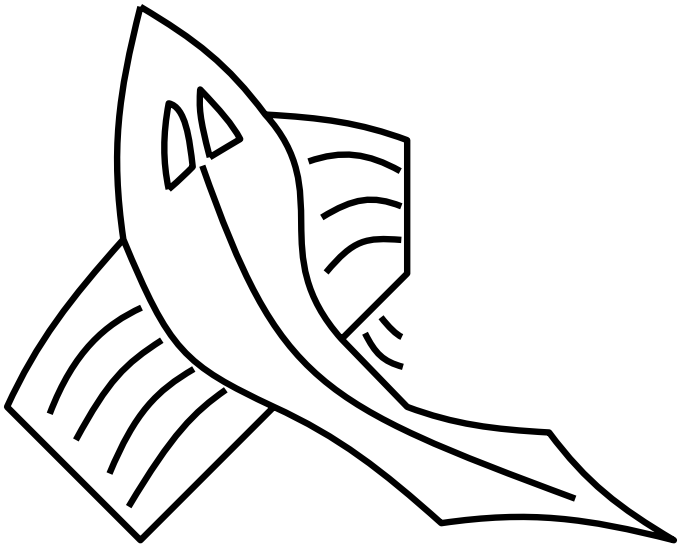
# side 1



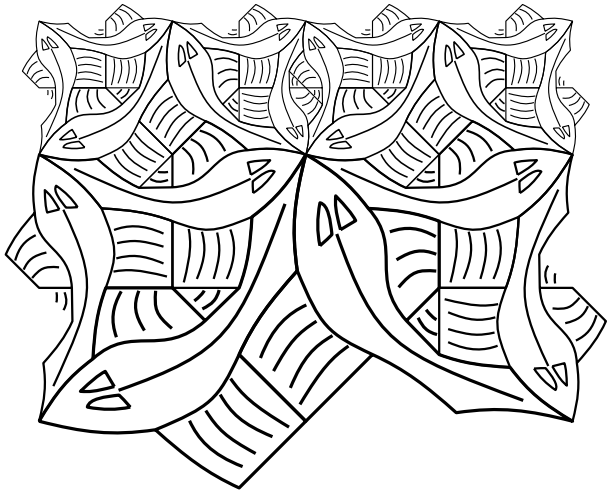
=>



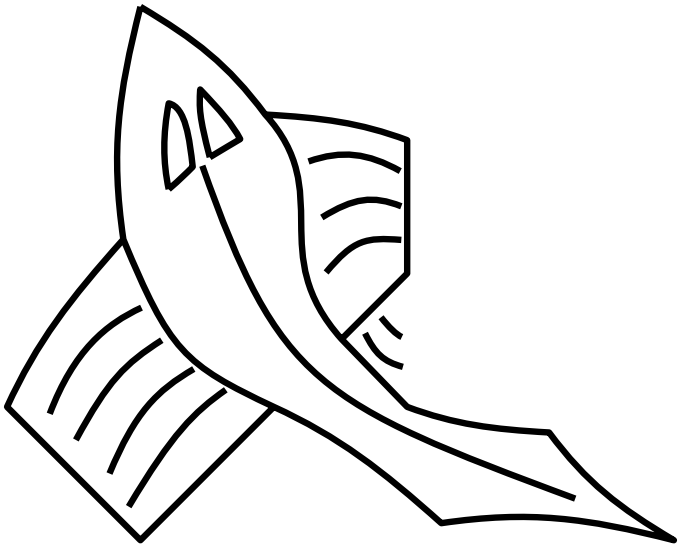
# side 2



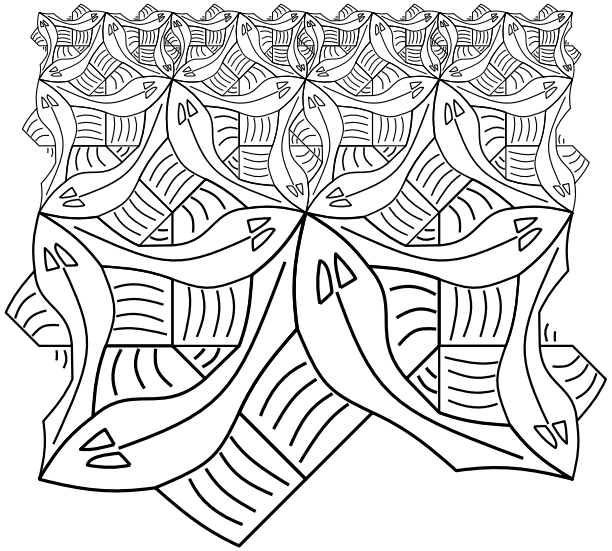
=>



# side 3

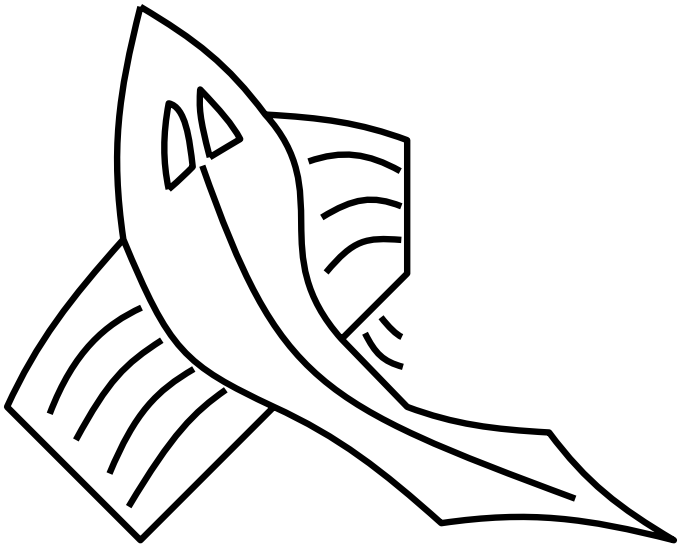


=>



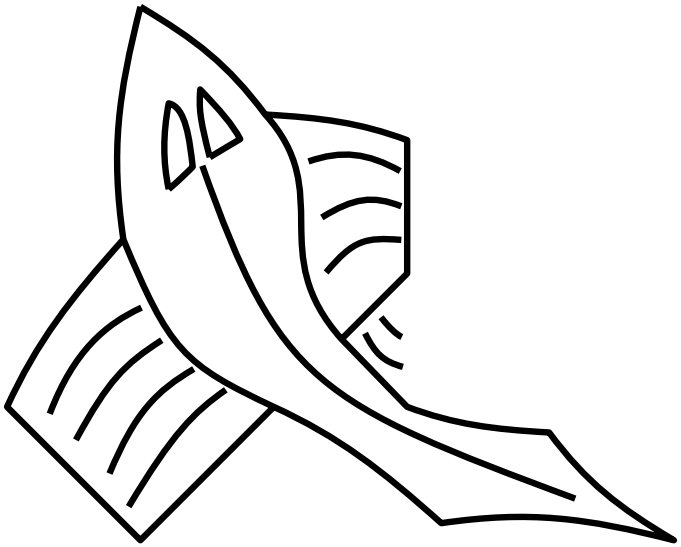
```
side 0 _ = blank
side n p =
    quartet (side (n-1) p)
            (side (n-1) p)
            (turn (ttile p))
            (ttile p)
```

corner 0

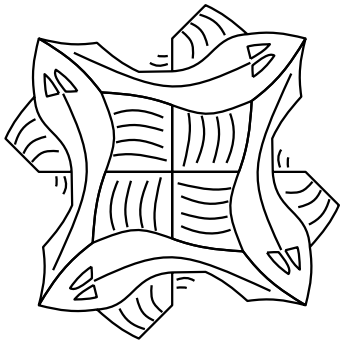


=>

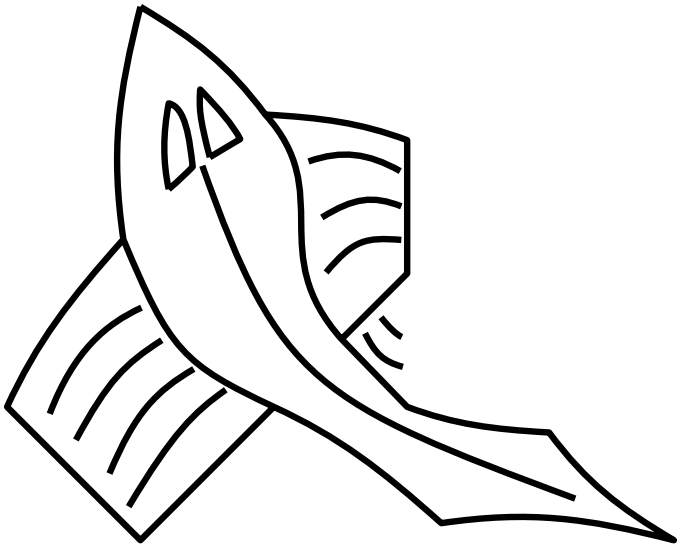
# corner 1



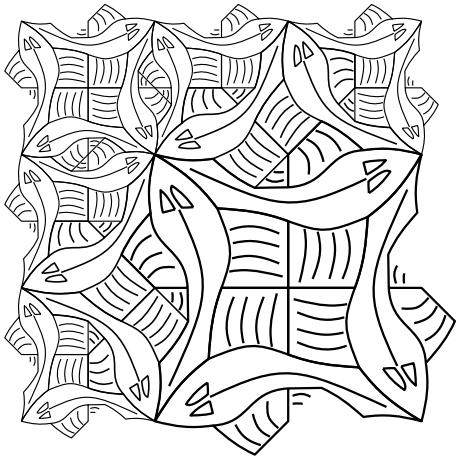
=>



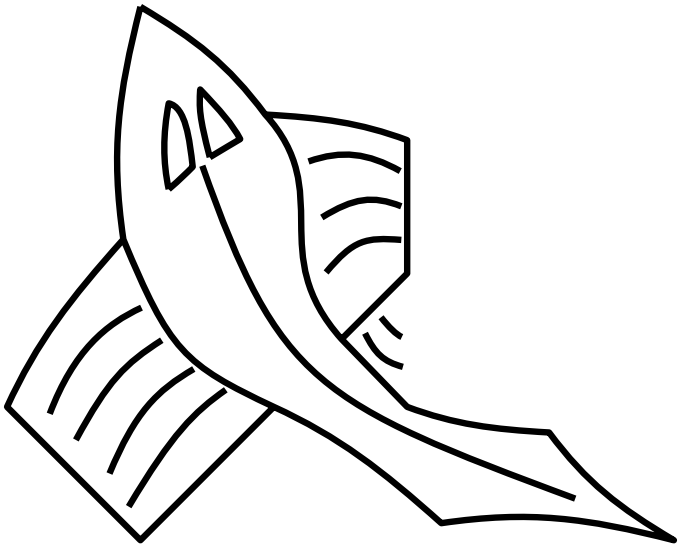
# corner 2



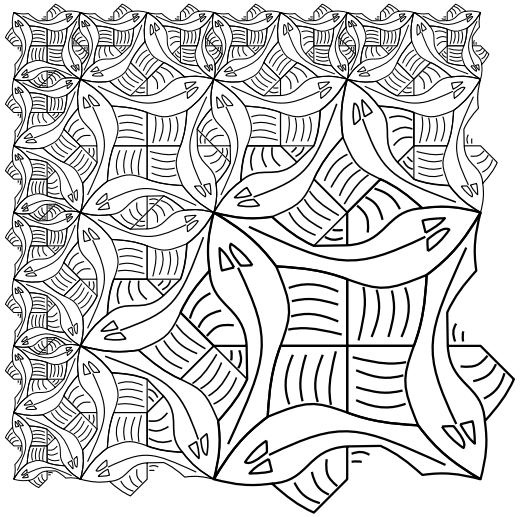
=>



# corner 3

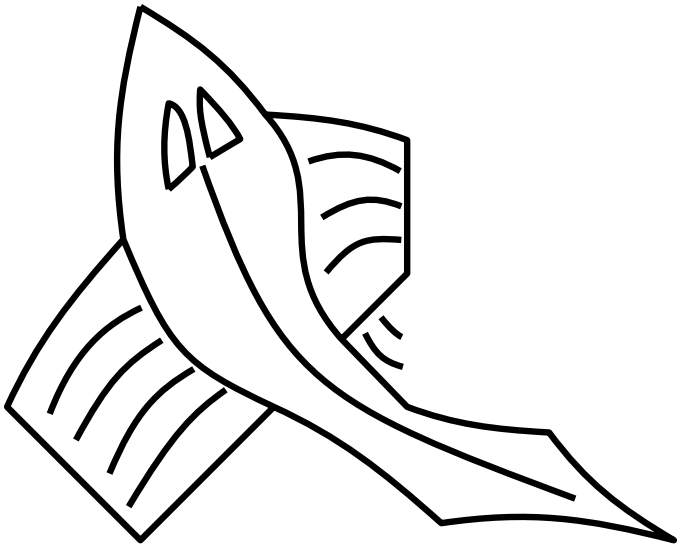


=>

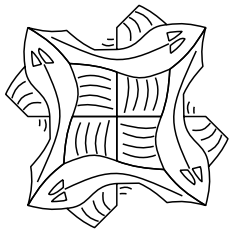


```
corner 0 _ = blank
corner n p =
    quartet (corner (n-1) p)
            (side (n-1) p)
            (turn (side (n-1) p))
            (utile p)
```

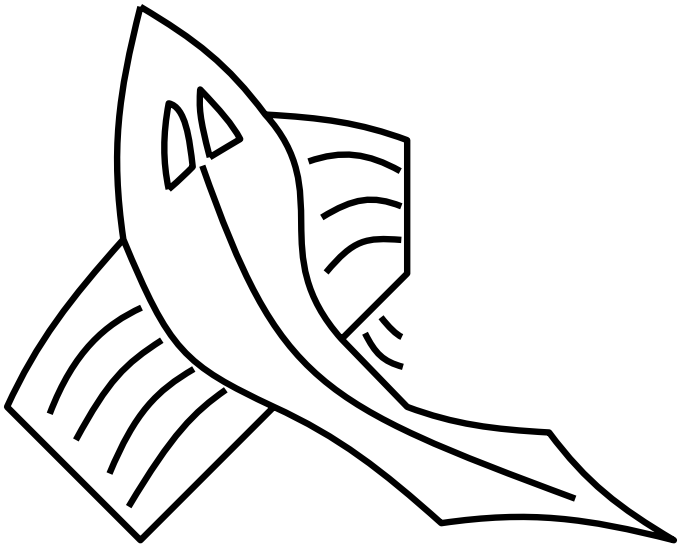
# square-limit 0



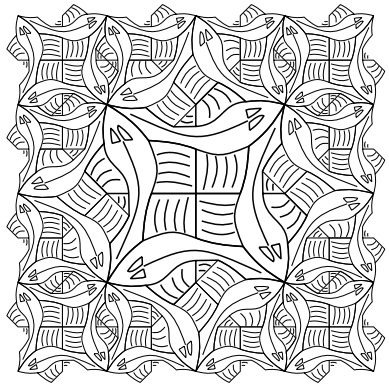
=>



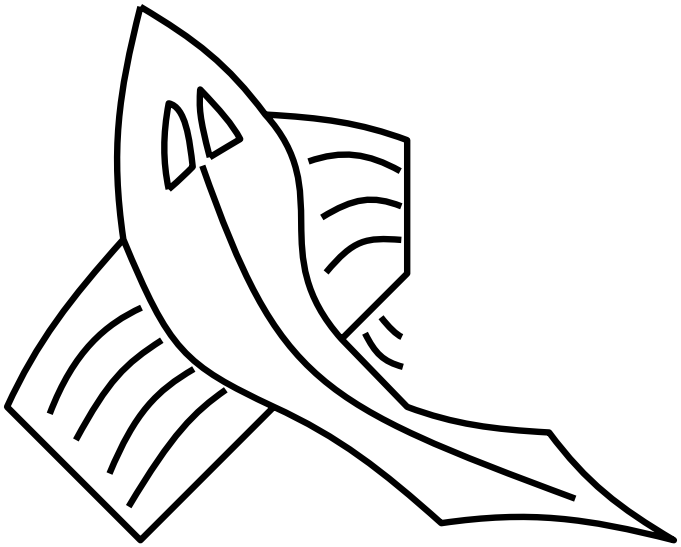
# square-limit 1



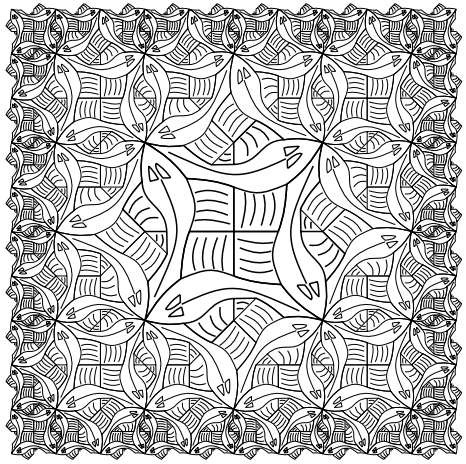
=>



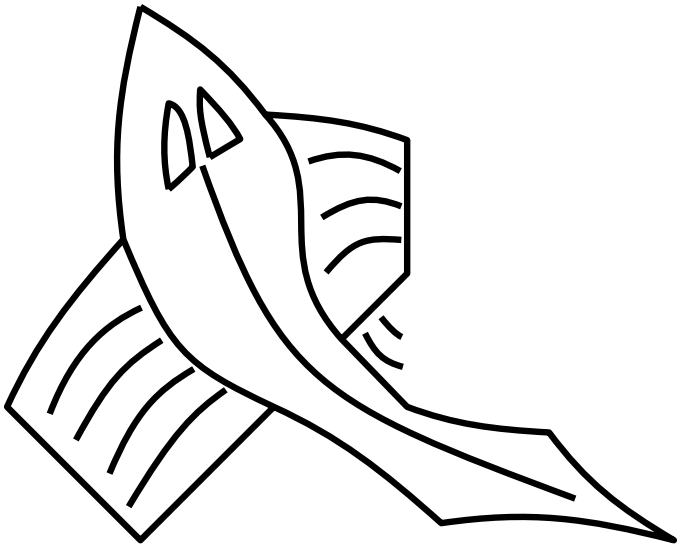
# square-limit 2



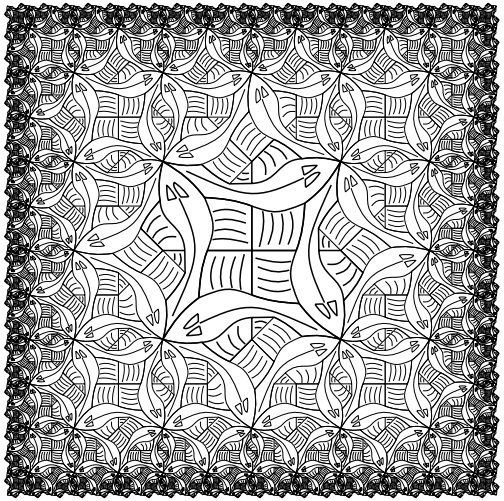
=>



# square-limit 3

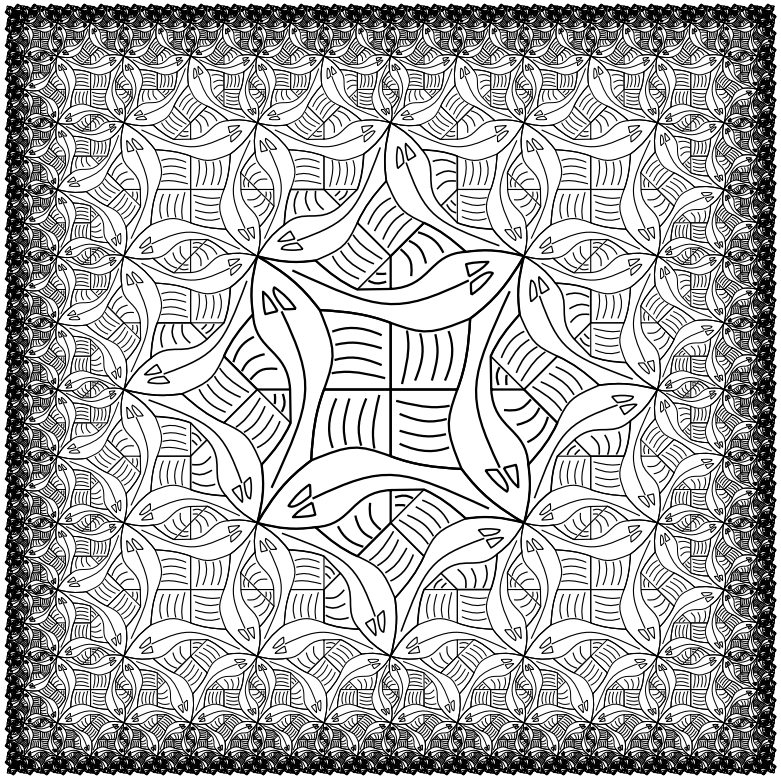


=>



```
squarelimit n p =
    nonet (corner n p)
          (side n p)
          (times 3 turn (corner n p))
          (turn (side n p))
          (utile p)
          (times 3 turn (side n p))
          (turn (corner n p))
          (times 2 turn (side n p))
          (times 2 turn (corner n p))
```

# 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.