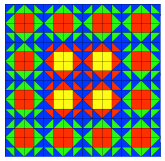
Divide, Conquer, and Glue with Pictures

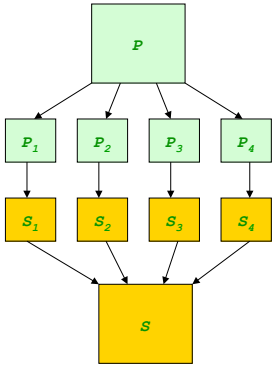


CS111 Computer Programming

Department of Computer Science
Wellesley College

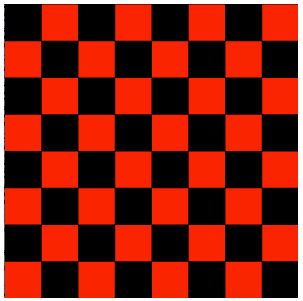
Recall big idea #3: Divide, conquer & glue (DCG)

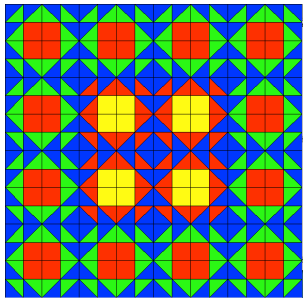
Divide
problem P into subproblems.
Conquer
each of the subproblems, &
Glue (combine)
the solutions to the subproblems into a solution S for P.



DCG to make cool pictures

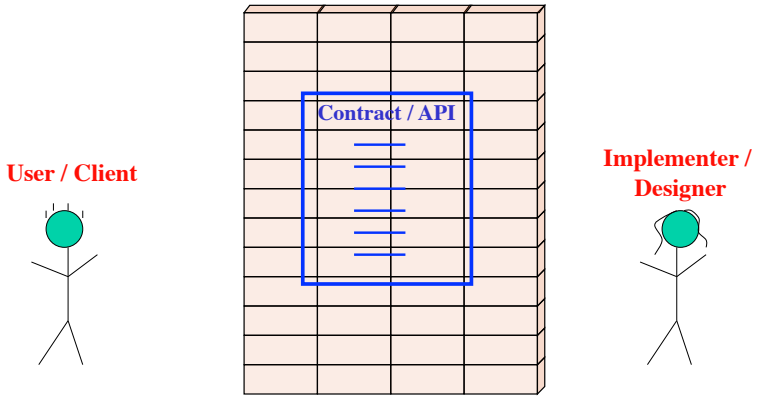
Today we'll see how to use DCG to make complex and interesting pictures in a simple way.



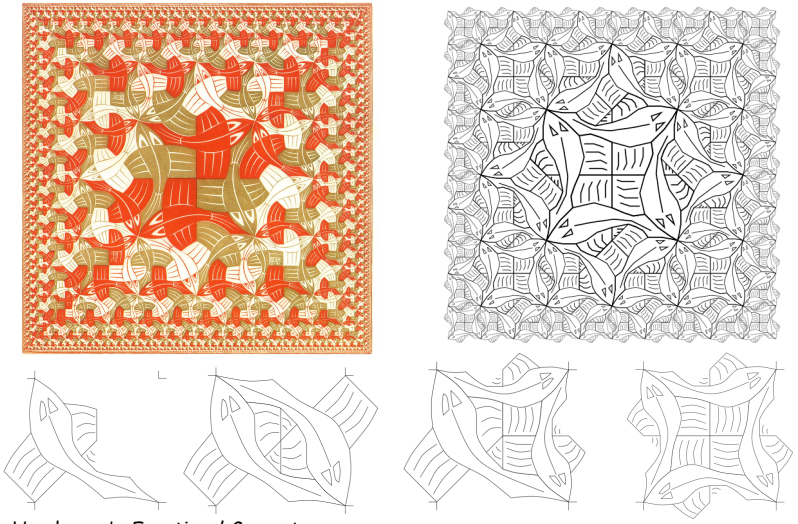


But first it will help to define a more abstract notion of "picture" starting with cs1graphics.

Recall big idea number 1: Abstraction



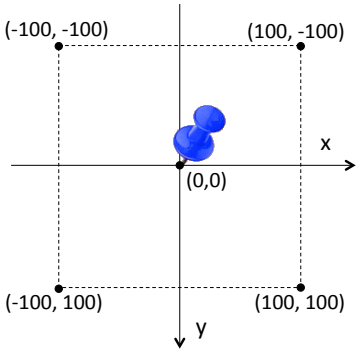
Peter Henderson's Picture Language



Henderson's *Functional Geometry* paper:
http://eprints.soton.ac.uk/257577/1/funcgeo2.pdf

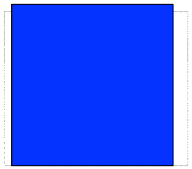
Picture abstraction on top of cs1graphics

- Inspired by Peter Henderson's picture language.
- We define a "picture" as a cs1graphics Drawable object that's 200x200 centered at reference point (0,0).
- By fixing the size of a picture, we never have to worry about pictures of different sizes.
- Rotations and flips of pictures by angles that are multiples of 90 degrees results in another picture!
- We will define many functions that take picture(s) as inputs and return a picture as an output.

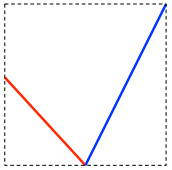


Some Primitive Pictures

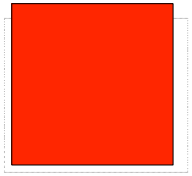
bp
(blue patch)



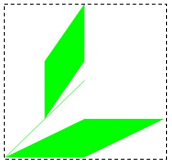
mark



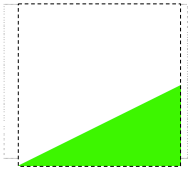
rp
(red patch)



gl
(green leaves)

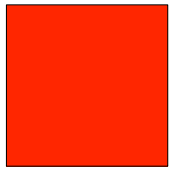


gw
(green
wedge)



empty

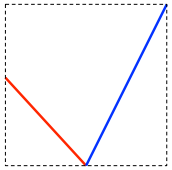
Sample primitives defined in Python [1]



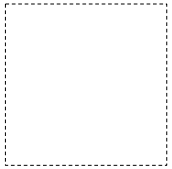
```
def patch(color):
  pic = Square(200)
  pic.setFillColor(color)
  pic.setBorderColor('black')
  return pic

rp = patch('red')
```

```
def wedge(color):
    pic = Polygon(Point(100, 0),
                  Point(100,100),
                  Point(-100, 100))
    # Shift reference point from (100,0) to (0,0)
    pic.adjustReference(-100,0)
    pic.setFillColor(color)
    pic.setBorderColor(color) # no border!
    return pic

gw = wedge('green')
```

Sample primitives defined in Python [2]

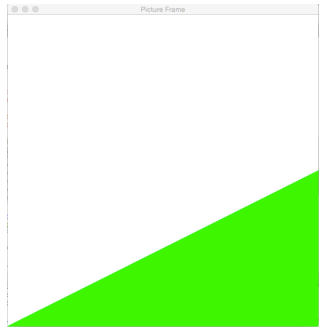


```
def checkmark(downColor, upColor):
    pic = Layer()
    downstroke = Path(Point(-100,0), Point(0,100))
    downstroke.setBorderColor(downColor)
    pic.add(downstroke)
    upstroke = Path(Point(0,100), Point(100,-100))
    upstroke.setBorderColor(upColor)
    pic.add(upstroke)
    return pic

mark = checkmark('red', 'blue')
```



```
empty = Layer()
```


Displaying Pictures



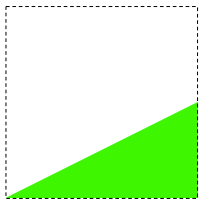
```
displayPic(gw)
```

```
def displayPic (pic):
    '''Display picture in 600x600 canvas'''
    frame = Canvas(600, 600,'white', 'Picture Frame')
    # Clone pic before changing it; otherwise
    # it wouldn't be a picture anymore!
    framedPic = pic.clone()
    framedPic.scale(3) # scale by 3 to fill 600x600 canvas
    framedPic.moveTo(300, 300) # move to center of canvas
    frame.add(framedPic)
```

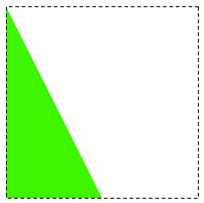
* The definition of displayPic is a tad more complex than shown here to allow for a closeAllPics() function that closes all canvases created by displayPic.

Clockwise rotations of pictures

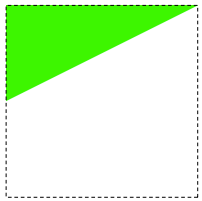
```
clockwise90(p);  # Returns a new picture that's p rotated  90° clockwise
clockwise180(p); # Returns a new picture that's p rotated 180° clockwise
clockwise270(p); # Returns a new picture that's p rotated 270° clockwise
```



gw



clockwise90(gw)



clockwise180(gw)

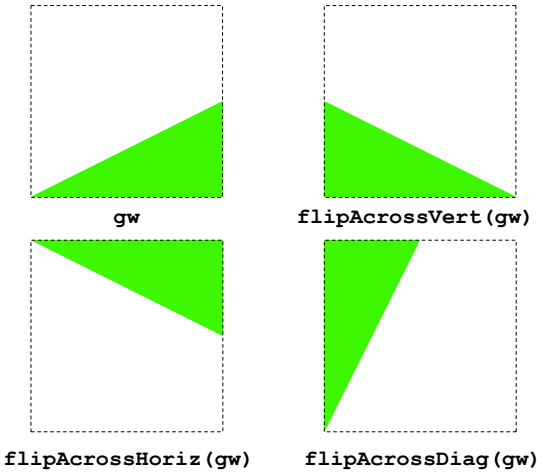
clockwise270(gw)

Defining picture rotations in Python

```
def clockwisePic(pic, angle):
    newPic = pic.clone() # create new picture by cloning it.
    newPic.rotate(angle) # if angle is a multiple of 90,
                         # result still satisfies definition
                         # of picture.
    return newPic

def clockwise90(pic):
    return clockwisePic (pic, 90)

def clockwise180(pic):
    return clockwisePic (pic, 180)

def clockwise270(pic):
    return clockwisePic (pic, 270)
```

Flipping pictures

```
flipAcrossVert(p); # Returns new picture that's p flipped across vertical axis
flipAcrossHoriz(p); # Returns new picture that's p flipped across horizontal axis
flipAcrossDiag(p); # Returns new picture that's p flipped across 45-degree axis
```

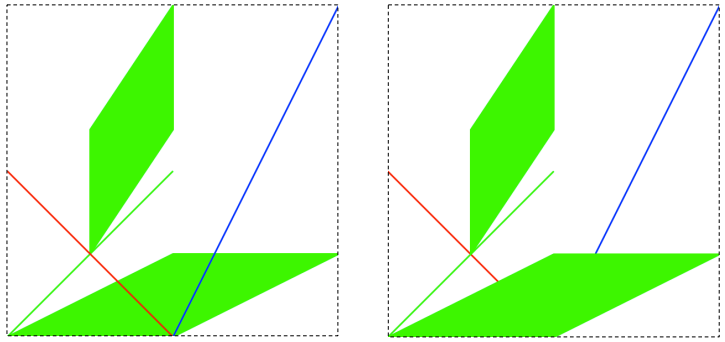


Defining picture flipping in Python

```
def flipPic (pic, angle):
    newPic = pic.clone() # create new picture by cloning it.
    newPic.flip(angle)   # if angle is a multiple of 45,
                         # result still satisfies definition
                         # of picture.
    return newPic

def flipAcrossVert(pic):
    return flipPic(pic, 0)

def flipAcrossHoriz(pic):
    return flipPic(pic, 90)

def flipAcrossDiag(pic):
    return flipPic(pic, 45)
```

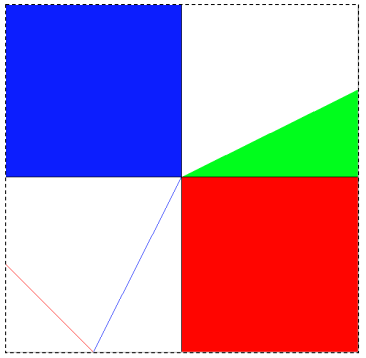

Overlaying pictures

```
def overlay(pic1, pic2):
    '''Returns a new pic in which pic1 appears on top of pic2'''
    newPic = Layer()
    newPic.add(pic2) # bottom pic goes first
    newPic.add(pic1) # top pic goes last
    return newPic
```



fourPics(): Combining four pictures

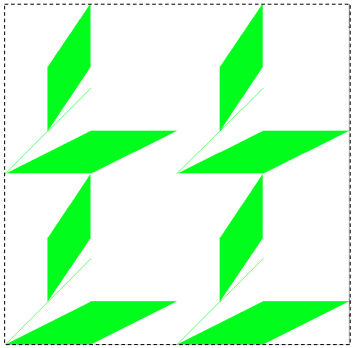
```
''' Returns a new picture with the four
    given pictures in its quadrants
          +-+-+
          |a|b|
          +-+-+
          |c|d|
          +-+-+                    '''
```




```
def fourPics(a, b, c, d):
    newPic = Layer()
    aHalf = a.clone()
    aHalf.scale(0.5)
    bHalf = b.clone()
    bHalf.scale(0.5)
    cHalf = c.clone()
    cHalf.scale(0.5)
    dHalf = d.clone()
    dHalf.scale(0.5)
    aHalf.move(-50,-50)
    bHalf.move(50,-50)
    cHalf.move(-50,50)
    dHalf.move(50,50)
    newPic.add(aHalf)
    newPic.add(bHalf)
    newPic.add(cHalf)
    newPic.add(dHalf)
    return newPic
```

fourSame(): Combining four copies of one picture

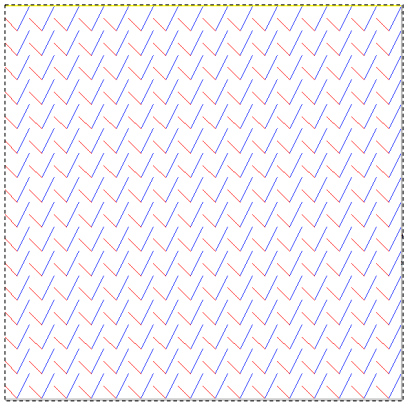
```
def fourSame(pic):
    return fourPics(pic, pic, pic, pic)
```



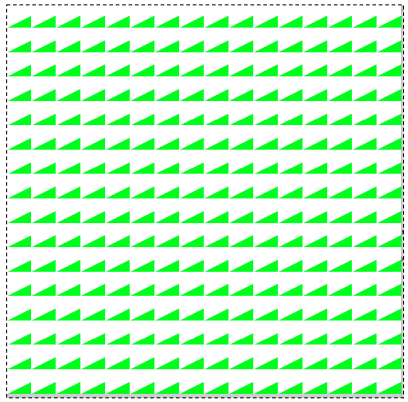
fourSame(gl)

Repeated tiling

```
def tiling(pic):
    return fourSame(fourSame(fourSame(fourSame(pic))))
```

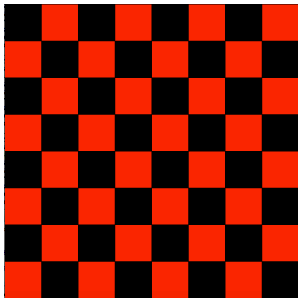


tiling(mark)

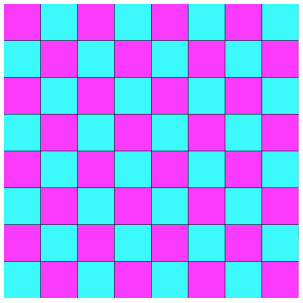


tiling(gw)

How to make a checkerboard?

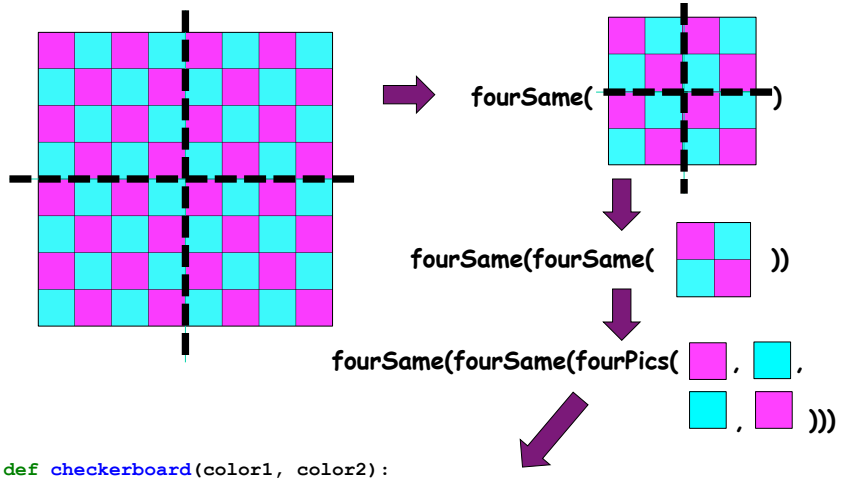


checkerboard('black', 'red')



checkerboard('magenta', 'cyan')

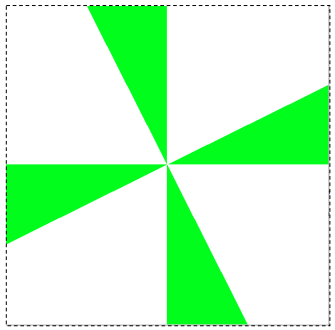
Divide/conquer/glue on checkerboard



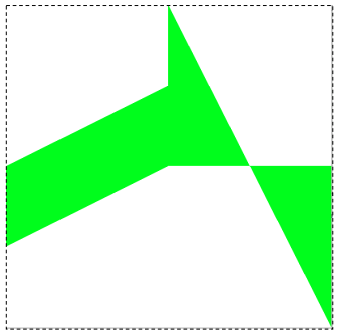

```
def checkerboard(color1, color2):
    return fourSame(fourSame(fourPics(patch(color1), patch(color2),
                                      patch(color2), patch(color1))))
```

Combining four rotations of a picture

```
def rotations(pic):
    return fourPics(clockwise270(pic), pic,
                    clockwise180(pic), clockwise90(pic))

def rotations2(pic):
    return fourPics(pic,clockwise90(pic),
                    clockwise180(pic), clockwise270(pic))
```



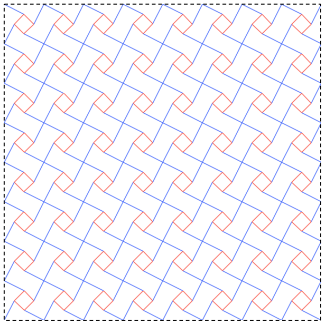
rotations(gw)



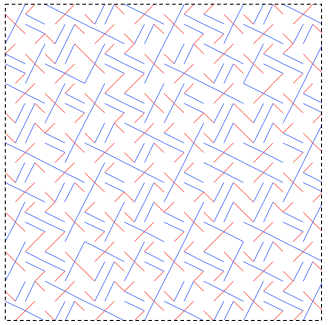
rotations2(gw)

A simple recipe for complexity

```
def wallpaper(pic):
    return rotations(rotations(rotations(rotations(pic))))

def design(pic):
    return rotations2(rotations2(rotations2(rotations2(pic))))
```



wallpaper(mark)

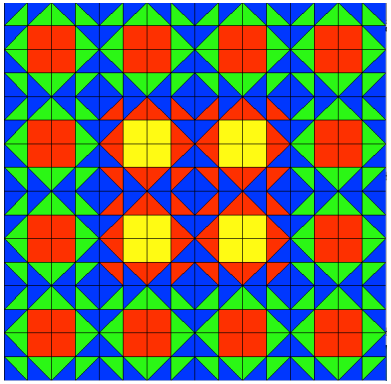


design(mark)

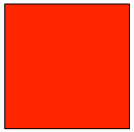
A quilt problem

How do we build this complex quilt …

… from simple primitive pictures like this?

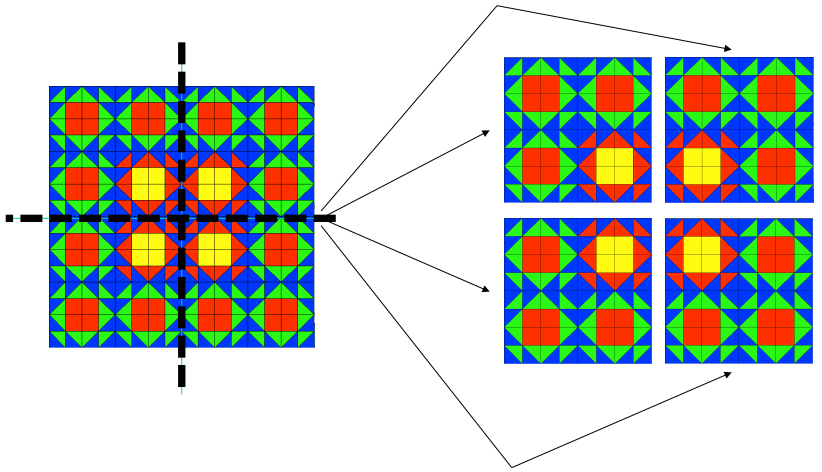


triangles('green', 'blue')

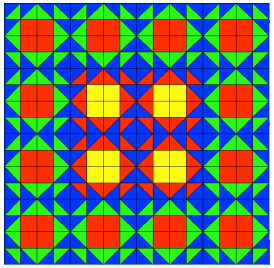


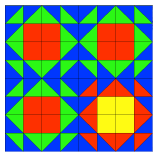
patch('red')

Divide the quilt into subproblems

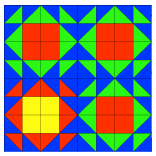


Conquer the subproblems using "wishful thinking"

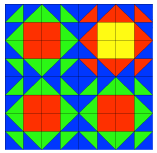




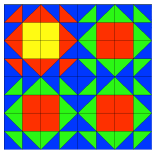
clockwise270(quadrant())



quadrant()

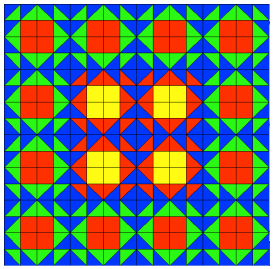


clockwise180(quadrant())

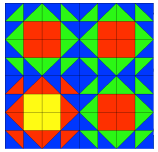


clockwise90(quadrant())

Glue the subsolutions to solve the problem



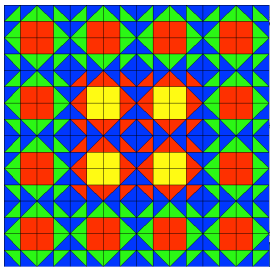
quilt()



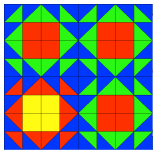
quadrant()

```
def quilt():
    return fourPics(clockwise270(quadrant()),
                    quadrant(),
                    clockwise180(quadrant()),
                    clockwise90(quadrant()))
```


Abstracting over the glue



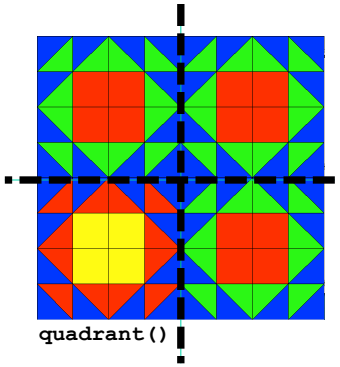
quilt()



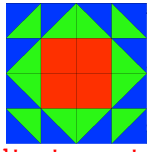
quadrant()

```
def quilt():
    return rotations(quadrant())

def rotations(pic): # picture function from before
    return fourPics(clockwise270(pic), pic,
                    clockwise180(pic), clockwise90(pic))
```


Now figure out quadrant()

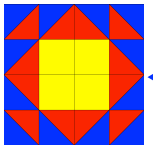


quadrant()



star('red', 'green', 'blue')

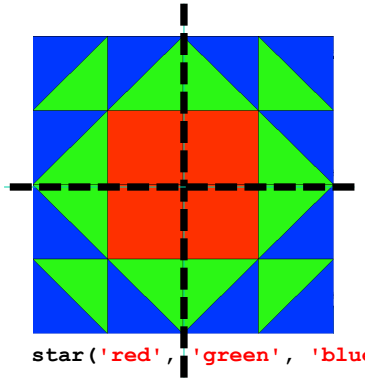
wishful thinking



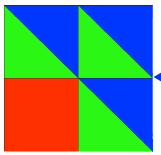
star('yellow', 'red', 'blue'')

```
def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))
def corner(llPic, outerPic):
    return fourPics(outerPic, outerPic,
                    llPic,    outerPic)
```

Continue the descent …



star('red', 'green', 'blue')

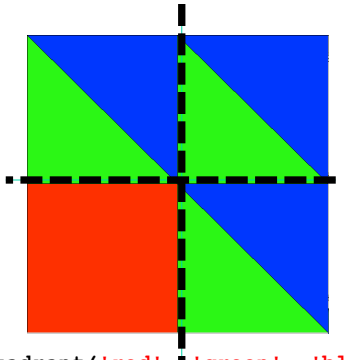


wishful thinking

starQuadrant('red', 'green', 'blue')

```
def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor,
                                  middleColor,
                                  outerColor))
```

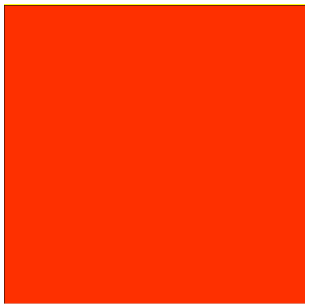

And descend some more …



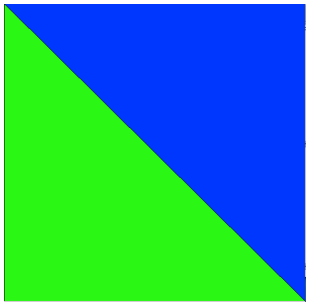
starQuadrant('red', 'green', 'blue')

```
def starQuadrant(squareColor, llTriColor, urTriColor):
    return corner(patch(squareColor),
                  triangles(llTriColor, urTriColor))
```


… until we reach primitives

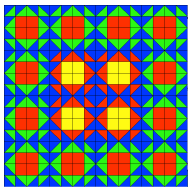


patch('red')



triangles('green', 'blue')

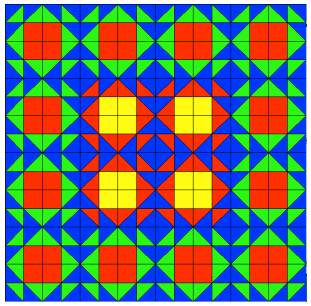
All together now



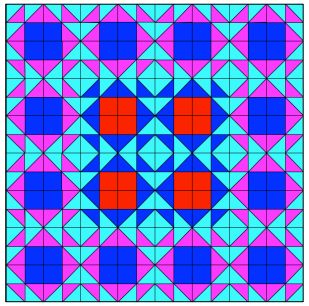
```
def quilt():
    return rotations(quadrant())

def quadrant():
    return corner(star('yellow', 'red', 'blue'),
                  star('red', 'green', 'blue'))

def star(innerColor, middleColor, outerColor):
    return rotations(starQuadrant(innerColor,
                                  middleColor,
                                  outerColor))

def corner(llPic, outerPic):
    return fourPics(outerPic, outerPic,
                    llPic,    outerPic)

def starQuadrant(squareColor, llTriColor, urTriColor):
    return corner(patch(squareColor),
                  triangles(llTriColor, urTriColor))
```

Abstracting over quilt colors

How to generalize **quilt** to define **quiltColors**?



```
quiltColors('yellow', 'red',
            'green', 'blue')
```



```
quiltColors('red', 'blue',
            'magenta', 'cyan'))
```