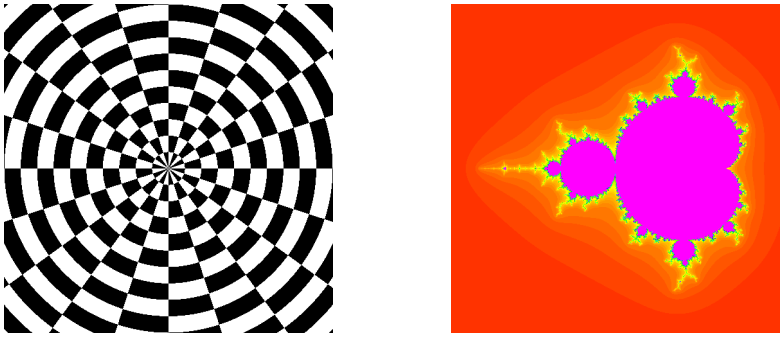
4 Bitmaps

This is a series of practicals on generating and rendering images. We start off with rendering simple bitmap images. Later, we will look at generating those images—in particular, in Practical 6 we will generate some 'Op art' images such as the polar chequerboard in Figure 1(a) below, and in Practical 7 some fractal images such as the *Mandelbrot Set* shown in Figure 1(b).



(a) (b)

Figure 1: (a) Some Op Art, and (b) a part of the Mandelbrot Set

4.1 Bitmaps as lists of lists

We'll start off by considering a very basic representation of images, as lists of lists of values.

type *Grid a* = [[*a*]]

For simplicity, we'll assume that these lists are *rectangular* (all the individual lists have the same length), and *non-empty* (so it's a non-empty list of non-empty lists). It is possible to enforce those invariants in the Haskell type, but a little awkward to do so; so instead, we will just have to take care to satisfy them in our definitions.

For example, here is a bitmap image of a cat:

```
catPic :: Grid Char
catPic =
  ["   *       *   ",
    " * *     * * ",
    " *   ***   * ",
    "*            *",
    "*   *    *   *",
    "*     *     *",
    " *          * ",
    "   *******   "]
```

1. Define a function *charRender*, to render a character grid as text output on the console. That is, *charRender catPic* should give you:

```
   *       *
  * *     * *
  *   ***   *
 *             *
 *   *     *   *
 *     *     *
  *           *
    *******
```

   (Hint: use the standard function *unlines*, which concatenates a list of *String*s into a single *String* with intervening newline characters, and *putStr* :: *String* → *IO* (), which 'interprets' a *String* with embedded newlines.)

   charRender :: *Grid Char* → *IO* ()

   Try it out by typing *charRender catPic* in GHCi.

   ```
   charRender = putStr ∘ unlines
   ```

2. Define functions to produce character bitmaps of a solid square, a hollow square, and a right triangle, all of a given side length:

```
solidSquare    :: Int → Grid Char
hollowSquare   :: Int → Grid Char
rightTriangle  :: Int → Grid Char
```

For example, with side length 5, you should get the following three pictures:

```
*****     *****     *
*****     *   *     **
*****     *   *     ***
*****     *   *     ****
*****     *****     *****
```

```
solidSquare s = replicate s (replicate s '*')

hollowSquare s | s < 2 = solidSquare s
hollowSquare s = [edge] ++ replicate (s - 2) mid ++ [edge]
  where
    edge = replicate s '*'
    mid = "*" ++ replicate (s - 2) ' ' ++ "*"

rightTriangle s = [replicate (s - i) ' ' ++ replicate i '*'
                   | i ← [1..s]]
```

3. The picture of a cat above is a bitmap: there is only one bit of information in each pixel (namely, whether the pixel is non-blank). So we might as well have started with a boolean grid. Define a function

   ```
   bwCharView :: Grid Bool → Grid Char
   ```

   to generate a character grid from a boolean grid. For example, it should satisfy

   ```
   catPic = bwCharView catBitmap
   ```

   where *catBitmap* is the boolean grid corresponding to the cat picture:

   ```
   catBitmap :: Grid Bool
   catBitmap = [
   ```

```
  [False, False, True, False, False, False, False, False, True, False, False],
  [False, True, False, True, False, False, False, True, False, True, False],
  [False, True, False, False, True, True, True, False, False, True, False],
  [True, False, False, False, False, False, False, False, False, False, True],
  [True, False, False, True, False, False, False, True, False, False, True],
  [True, False, False, False, False, True, False, False, False, False, True],
  [False, True, False, False, False, False, False, False, False, True, False],
  [False, False, True, True, True, True, True, True, True, False, False]
  ]
```

Here is another bitmap for you to experiment with:

```
fprBitmap :: Grid Bool
fprBitmap = [
  [f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f],
  [f, t, t, t, t, f, f, t, t, t, f, f, f, t, t, t, f, f],
  [f, t, f, f, f, f, f, f, t, f, f, t, f, f, t, f, f, t, f],
  [f, t, t, t, f, f, f, t, t, t, f, f, f, t, t, t, f, f],
  [f, t, f, f, f, f, f, f, t, f, f, f, f, f, t, f, f, t, f],
  [f, t, f, f, f, f, f, f, t, f, f, f, f, f, t, f, f, t, f],
  [f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f] ]
  where f = False; t = True
```

```
bwCharView = map (map (λb → if b then '*' else ' '))
```

4.2 Points

Rather than explicitly specifying the value of each and every pixel, a more convenient way to describe a bitmap might be simply to list the positions of the non-blank pixels—especially if the image is rather sparse. For example, the cat bitmap is mostly blank, and only the following positions are non-blank:

```
type Point = (Integer, Integer)
catPoints :: [Point]
catPoints =
  [(2, 0), (8, 0), (1, 1), (3, 1), (7, 1), (9, 1), (1, 2), (4, 2), (5, 2), (6, 2),
```

$$(9,2),(0,3),(10,3),(0,4),(3,4),(7,4),(10,4),(0,5),(5,5),$$
$$(10,5),(1,6),(9,6),(2,7),(3,7),(4,7),(5,7),(6,7),(7,7),(8,7)\,]$$

Here, each point is a pair of (x,y) coordinates; the x coordinate is horizontal, counting rightwards, and the y coordinate vertical, counting downwards, with the top left corner at the origin.

4. Define a function

   $$pointsBitmap :: [\,Point\,] \to Grid\ Bool$$

   to convert such a list of points to a boolean grid, so that

   $$catBitmap = pointsBitmap\ catPoints$$

   (Hint: it helps to draw an actual grid on a piece of paper. Which points on your grid correspond to the ones given in *catPoints*? What should happen to them? and the others?)

```
pointsBitmap ps
  = [ [ (x, y) ∈ ps | x ← [ minimum xs .. maximum xs ] ]
                    | y ← [ minimum ys .. maximum ys ] ]
    where (xs, ys) = (map fst ps, map snd ps)
```

5. Define a function

   $$gridPoints :: Grid\ Bool \to [\,Point\,]$$

   to perform the reverse conversion. You should find that

   $$pointsBitmap\ (gridPoints\ catBitmap) = catBitmap$$

   (Why is it not the case that $pointsBitmap \circ gridPoints = id$ more generally? And why is it generally not the case that $gridPoints \circ pointsBitmap = id$ either?)

$$gridPoints\ bss = [\,(x,y) \mid (bs,y) \leftarrow zip\ bss\ [0\,..\,],\ (b,x) \leftarrow zip\ bs\ [0\,..\,], b\,]$$

Given a list of points *ps*, we get a new list of points *qs* by

$$qs = gridPoints\ (pointsBitmap\ ps)$$

In general, *ps* and *qs* will be in different orders; moreover, *ps* may have duplicates where *qs* will not. But (at least with my solution) they will both contain the same set of points. In the opposite direction, if the bitmap has blank rows or columns at the edges, these won't show up in the list of points, so won't get reconstructed.

4.3 Greymaps

Because the *Grid* type is parametrized, we have the freedom to use other types than *Bool* to represent individual pixel values. For example, we can model greyscale images by using numeric pixel values (perhaps *Float* in the unit interval). For example, here is a greyscale grid (using only three shades of grey) depicting the Haskell logo as seen on `http://www.haskell.org/haskellwiki/Haskell_logos#Current_Haskell_logo`.

```
logoShades :: Grid Float
logoShades = [
  [ h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, h, h, h, h, 0, 0, h, 1, 1, 1, h, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, h, h, h, h, 0, 0, h, 1, 1, 1, h, 0, 0, h, h, h, h, h, h, h, h, h ],
  [ 0, 0, 0, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, h, h, h, h, h, h, h, h ],
  [ 0, 0, 0, 0, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, 0, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 1, h, 0, 0, h, h, h, h, h, h, h ],
  [ 0, 0, 0, h, h, h, h, 0, 0, h, 1, 1, 1, 1, 1, 1, 0, 0, 0, h, h, h, h, h, h ],
  [ 0, 0, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 0, h, 1, 1, 1, h, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, h, h, h, h, 0, 0, h, 1, 1, 1, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0 ],
  [ 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0 ],
  [ h, h, h, h, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, h, 1, 1, 1, h, 0, 0, 0, 0, 0 ]
  ] where h = 0.5
```

Of course, to benefit from this greater precision, we also need a wider choice of characters to display on the screen. Previously we only used '*' for black and a space for white. Now we define a 'palette' of characters, varying linearly from 'white' to 'black':

```
charPalette :: [Char]
charPalette = " .:oO8@"
```

This list is somewhat arbitrary, and how good it is will depend on what screen font you use; but the principle is that the characters are in order of darkness. You might have better results with Unicode *block elements*:

```
charPaletteBlocks = " \9617\9618\9619\9608"
```

(this only seems to work on Windows if you run GHCi within Cygwin). Using *charPalette*, we can create our own ASCII art.

6. Define a function *greyCharView* that converts a greyscale grid (where each cell is a *Float* between 0 and 1) into a character grid, parametrized on a palette such as *charPalette*. The function should work whatever the palette size.

```
greyCharView :: [Char] → Grid Float → Grid Char
```

(Hint: use *fromIntegral* to convert an *Int* to a *Float*, and *floor* to convert back, rounding down.) For example, you should be able to see the Haskell logo via

```
charRender (greyCharView charPalette logoShades)
```

Actually, there's no reason to fix the palette and the output type to characters; so we define *greyCharView* as an instance of a polymorphic *paletteView* (which we will use later at a different type).

```
greyCharView = paletteView
paletteView :: [a] → Grid Float → Grid a
paletteView p = map (map (get p))
  where
    get p r = p !! (floor (r * fromIntegral l) `min` (l - 1))
    l = length p
```

7. The Haskell logo is supposed to be dark on a light background; so the greyscale grid above assumes that your console window also shows dark characters on a light background. How would you properly display an image on a console that has white characters on a black background?

It suffices to reverse the palette:

```
charRender (greyCharView (reverse charPalette) logoShades)
```

4.4 Portable Anymaps

It's all very well displaying ASCII art on the console, but it's not very sophisticated. How can we make the more glamorous images shown at the start of this practical? We need to transform our grids into some standard image file format, write them out to a file, and view them using an image viewer.

Most image file formats are rather complicated—in particular, they use data compression to save space, rather than explicitly specifying the intensity of every pixel. However, one particularly simple format is the

Portable Anymap family by Jef Poskanzer, with provision for bitmaps (black and white), greyscale and colour images. Portable Anymap files are plain ASCII text, and very forgiving about whitespace, linebreaks and so on. They make very inefficient use of space, but they are easy to manipulate. A brief description of the Portable Anymap family of formats is provided at the end of this practical, in Section 4.5.

8. As you'll see from Section 4.5, the three variants of the Portable Anymap format that we consider are all very similar; they consist of a magic identifier, some dimensions, and a long list of pixel values, all separated by arbitrary whitespace. The dimensions include at least the width and the height of the image. For greyscale and colour images, they also include the colour depth (the maximum allowable value for grey, red, green, or blue); this is omitted for black and white images. So it makes sense to define one generic function to output in any of the formats:

   $$makePNM :: Show\ a \Rightarrow String \to String \to Grid\ a \to String$$

   The first parameter is the magic identifier (eg `"P1"`). The second parameter is a string representing the colour depth; for bitmaps, this should just be the empty string. Complete the definition of *makePNM*.

```
makePNM magic max g
  = unlines ([magic, show w, show h, max] ++ map row g)
  where
    w   = length (head g)
    h   = length g
    row = unwords ∘ map show
```

9. Hence define a function

   $$makePBM :: Grid\ Bool \to String$$

   to translate a boolean grid into PBM format. For example,

   putStr (*makePBM fprBitmap*)

   should produce something like the following output:

```
P1
18 7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 1 0 0
0 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1 0
0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Given *makePBM*, we can easily write a PBM file:

```
pbmRender :: String → Grid Bool → IO ()
pbmRender file = writeFile file ∘ makePBM
```

The first parameter is the filename; for example,

```
pbmRender "test.pnm" fprBitmap
```

writes the image to the file `test.pnm`.

```
makePBM = makePNM "P1" "" ∘ map (map fromEnum)
```

10. Similarly, define functions

```
makePGM  :: Int → Grid Float → String
pgmRender :: String → Int → Grid Float → IO ()
```

to translate a greyscale grid into PGM format. In both cases, the *Int* parameter is (one less than) the number of grey levels in the output. For example, here is a greyscale grid:

```
fprGreymap :: Grid Float
fprGreymap = [
  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, a, a, a, a, 0, 0, b, b, b, 0, 0, 0, 1, 1, 1, 0, 0],
  [0, a, 0, 0, 0, 0, 0, b, 0, 0, b, 0, 0, 1, 0, 0, 1, 0],
  [0, a, a, a, 0, 0, 0, b, b, b, 0, 0, 0, 1, 1, 1, 0, 0],
  [0, a, 0, 0, 0, 0, 0, b, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
  [0, a, 0, 0, 0, 0, 0, b, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0],
```

```
    [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]
    where a = 1 / 3; b = 2 / 3
```

and

```
putStr (makePGM 15 fprGreymap)
```

yields

```
P2
18 7
15
0  0  0  0  0  0  0   0   0   0   0  0  0   0   0   0   0  0
0  5  5  5  5  0  0  10  10  10   0  0  0  15  15  15   0  0
0  5  0  0  0  0  0  10   0   0  10  0  0  15   0   0  15  0
0  5  5  5  0  0  0  10  10  10   0  0  0  15  15  15   0  0
0  5  0  0  0  0  0  10   0   0   0  0  0  15   0   0  15  0
0  5  0  0  0  0  0  10   0   0   0  0  0  15   0   0  15  0
0  0  0  0  0  0  0   0   0   0   0  0  0   0   0   0   0  0
```

```
makePGM max = makePNM "P2" (show max) ∘
                  map (map (floor ∘ (∗fromIntegral max)))
pgmRender file max = writeFile file ∘ makePGM max
```

11. For colour pixmaps, we need a representations of colour pixels. The standard way to do this for video images is with triples specifying red, green, and blue intensities.

```
data RGB = RGB Int Int Int
instance Show RGB where
  show (RGB r g b) = show r ++ "  " ++ show g ++ "  " ++ show b
```

For example, here is a colour grid, with some red, orange, and yellow pixels on a black background.

```
fprPixmap :: Grid RGB
fprPixmap = [
  [b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b],
  [b, r, r, r, r, b, b, o, o, o, b, b, b, y, y, y, b, b],
  [b, r, b, b, b, b, b, o, b, b, o, b, b, y, b, b, y, b],
  [b, r, r, r, b, b, b, o, o, o, b, b, b, y, y, y, b, b],
  [b, r, b, b, b, b, b, o, b, b, b, b, b, y, b, b, y, b],
  [b, r, b, b, b, b, b, o, b, b, b, b, b, y, b, b, y, b],
  [b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b]]
  where r = RGB 7 0 0; o = RGB 7 3 0; y = RGB 7 7 0; b = RGB 0 0 0
```

Define functions to translate an RGB grid into PPM format:

```
makePPM   :: Int → Grid RGB → String
ppmRender :: String → Int → Grid RGB → IO ()
```

Again, the *Int* parameter is the maximum allowable colour value. Evaluating

```
putStr (makePPM 7 fprPixmap)
```

gives the output

```
P3
18 7
7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 0 0 7 0 0 7 0 0 7 0 0 0 0 0 0 0 0 0 0 7 3 0 7 3 0
7 3 0 0 0 0 0 0 0 0 0 0 0 7 7 0 7 7 0 7 7 0 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 3 0 0 0 0
0 0 0 7 3 0 0 0 0 0 0 0 0 7 7 0 0 0 0 0 0 0 0 7 7 0 0 0 0
0 0 0 7 0 0 7 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 7 3 0 7 3 0
7 3 0 0 0 0 0 0 0 0 0 0 0 7 7 0 7 7 0 7 7 0 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 3 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 0 0 7 7 0 0 0 0 0 0 0 7 7 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 7 3 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 7 7 0 0 0 0 0 0 0 7 7 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
makePPM max = makePNM "P3" (show max)
ppmRender file max = writeFile file ∘ makePPM max
```

12. Define a function

    $group :: Int \to [a] \to [[a]]$

    that divides a list into chunks of the given length; for example,

    $group\ 3\ \texttt{"ablewhackets"} = [\texttt{"abl"}, \texttt{"ewh"}, \texttt{"ack"}, \texttt{"ets"}]$

```
group n [ ] = [ ]
group n xs = ys : group n zs where (ys, zs) = splitAt n xs
```

13. Hence define functions to parse the list of words in a PGM file into a greyscale grid:

    $pgmParse :: [String] \to Grid\ Float$

    (Hint: *read* parses a string as an *Integer*, and *fromInteger* converts an *Integer* to a *Float*.) You can use this function to read in a PGM file as follows:

```
pgmRead :: String → IO (Grid Float)
pgmRead f = do {s ← readFile f; return (pgmParse (words s))}
```

    (where *words* is a standard function to split a string into words separated by whitespace). Try

```
do {x ← pgmRead "radcliffe64.pgm";
    charRender (greyCharView (reverse charPalette) x)}
```

(You can try PBM and PPM files too; but I don't have any pretty pictures for you to test them on.)

```
pgmParse ("P2" : sw : sh : sd : ns) = group (read sw) xs
  where
    xs = [fromInteger (read sn) / d | sn ← ns]
    d  = fromInteger (read sd)
```

4.5 Appendix: Portable Anymap file format

Jef Poskanzer's *Portable Anymap* file format is an extremely simple family of file formats for graphical images. All members of the family are plain text formats, and the specifications are very forgiving of layout in terms of whitespace, linebreaks and so on. The three members of the family are *Portable Bitmaps* (for black and white images), *Portable Greymaps* and *Portable Pixmaps* (for full-colour images). The usual file extension for all three is `.pnm`. A simplified summary of the format of each is presented below.

Portable Bitmap format

The contents of a portable bitmap file consists of the following items:

- the 'magic identifier' `P1`;
- whitespace (blanks, tabs, carriage returns, linefeeds);
- the image width, in pixels, formatted in ASCII in decimal;
- whitespace;
- the image height, formatted like the width;
- whitespace;
- width × height bits, each either 0 (white) or 1 (black), in rows from the top left of the image, separated by whitespace.

For example, here is a small portable bitmap file.

```
P1
18 7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 1 0 0
0 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 1 0
```

```
0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0
0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

It has been broken into lines of 18 pixels for presentation purposes, but that's not necessary: the pixels could be one per line, or differently broken up. Strictly speaking, there should not be more than 70 characters per line.

Portable Greymap format

The contents of a portable greymap file consists of the following items:

- the 'magic identifier' `P2`;
- whitespace;
- the image width, in pixels, formatted in ASCII in decimal;
- whitespace;
- the image height, formatted like the width;
- whitespace;
- the maximum grey value, formatted like the width;
- whitespace;
- width × height grey values, each between 0 (black) and the maximum (white) and formatted in ASCII in decimal, in rows from the top left of the image, separated by whitespace.

For example, here is a small portable greymap file.

```
P2
18 7
15
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 0  5  5  5  5  0  0 10 10 10  0  0  0 15 15 15  0  0
 0  5  0  0  0  0  0 10  0  0 10  0  0 15  0  0 15  0
 0  5  5  5  0  0  0 10 10 10  0  0  0 15 15 15  0  0
 0  5  0  0  0  0  0 10  0  0  0  0  0 15  0  0 15  0
 0  5  0  0  0  0  0 10  0  0  0  0  0 15  0  0 15  0
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

Portable Pixmap format

The contents of a portable pixmap file consists of the following items:

- the ‘magic identifier’ `P3`;
- whitespace;
- the image width, in pixels, formatted in ASCII in decimal;
- whitespace;
- the image height, formatted like the width;
- whitespace;
- the maximum colour-component value, formatted like the width;
- whitespace;
- width × height pixels, each consisting of three colour component values (red, green and blue), each between 0 (completely off) and the maximum (completely on) and formatted in ASCII in decimal, in rows from the top left of the image, separated by whitespace.

For example, here is a small portable pixmap file.

```
P3
18 7
7
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 0 0 7 0 0 7 0 0 7 0 0 0 0 0
0 0 0 7 3 0 7 3 0 7 3 0 0 0 0 0 0 0
0 0 0 7 7 0 7 7 0 7 7 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 3 0 0 0 0 0 0 0 7 3 0 0 0 0
0 0 0 7 7 0 0 0 0 0 0 0 7 7 0 0 0 0
0 0 0 7 0 0 7 0 0 7 0 0 0 0 0 0 0 0
0 0 0 7 3 0 7 3 0 7 3 0 0 0 0 0 0 0
0 0 0 7 7 0 7 7 0 7 7 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 3 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 7 0 0 0 0 0 0 0 7 7 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 3 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 7 0 0 0 0 0 0 0 7 7 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Viewers

Under MacOS X, the `Preview` application can display Portable Anymaps. There is also a separate tool called `ToyViewer` for the same purpose; this can be downloaded from `http://waltz.cs.scitec.kobe-u.ac.jp/OSX/toyv-eng.html`. (In fact, Emacs for Mac from

```
http://emacsformacosx.com/
```

also displays Portable Anymaps fine. I don't know about other Emacsen.)

Under Windows, the built-in Windows Picture and Fax Viewer does not support Portable Anymap files, but various other viewers and editors do. Try `PMView` (`www.pmview.com`) or *Paint Shop Pro* (`http://www.jasc.com/products/paintshoppro`); neither is free, but both provide free trials.

Under X windows on Unix systems, the standard image viewer `xv` can be used to view Portable Anymap files, and to convert them to other formats.

As an alternative to all of these, I will provide a little Java program `PNMViewer` to view a Portable Anymap file. This won't permit conversion to other formats, but at least it lets you see what you're doing. From the command line, you can type

```
java -jar PNMViewer.jar myfile.ppm
```

to open a window displaying your image in file `myfile.ppm`. And there is a neat online viewer at

```
http://paulcuth.me.uk/netpbm-viewer/
```