Functional Programming Practicals

0 Getting started

We will be using GHCi for the practicals. To run GHCi, simply open a terminal window and type '`ghci`'. One typically uses a text editor to write or edit a Haskell script, saves that to disk, and loads it into GHCi. To load a script, it is helpful if you run GHCi from the directory containing the script. You can simply give the name of the script file as a parameter to the command `ghci`. Or, within GHCi, you can type '`:l`' followed by the name of the script to load, and '`:r`' with no parameter to reload the file previously loaded.

There are model answers for most of the exercises, and in some cases skeletons of a solution to save you from having to type in what is provided to start with. For instructions on how to find those, see the practical assistant.

Jeremy Gibbons
(with thanks to Colin Runciman, Jon Hill, Ralf Hinze, Nicolas Wu)
September 2014

1 Basic definitions

1. Define the following numeric functions:

   - a function *square* that squares its argument, then a function *quad* that raises its argument to the fourth power using *square*;
   - a function *larger* that returns the larger of its two arguments;
   - a function for computing the area of a circle with a given radius (use the type *Double*). (Hint: the formula for calculating the area A of a circle with a radius r is $A = \pi r^2$, where π is called *pi* in Haskell.)

2. Here is a script of function definitions:

$$
\begin{array}{l}
\mathit{add} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
\mathit{add}\; x\; y \quad = x + y \\
\mathit{double} :: \mathit{Integer} \to \mathit{Integer} \\
\mathit{double}\; x \quad = x + x \\
\mathit{first} :: \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
\mathit{first}\; x\; y \quad = x \\
\mathit{cond} :: \mathit{Bool} \to \mathit{Integer} \to \mathit{Integer} \to \mathit{Integer} \\
\mathit{cond}\; x\; y\; z = \mathbf{if}\; x\; \mathbf{then}\; y\; \mathbf{else}\; z \\
\mathit{twice} :: (\mathit{Integer} \to \mathit{Integer}) \to \mathit{Integer} \to \mathit{Integer} \\
\mathit{twice}\; f\; x \quad = f\,(f\,x) \\
\mathit{infinity} :: \mathit{Integer} \\
\mathit{infinity} \quad = \mathit{infinity} + 1
\end{array}
$$

   (The function *twice* is an example of a higher-order function, which takes another function as one of its arguments. Although we haven't studied higher-order functions yet, just follow the reduction rules.) Give the applicative- and normal-order reduction sequences for the following expressions:

   - $\mathit{first}\; 42\; (\mathit{double}\; (\mathit{add}\; 1\; 2))$
   - $\mathit{first}\; 42\; (\mathit{double}\; (\mathit{add}\; 1\; \mathit{infinity}))$
   - $\mathit{first}\; \mathit{infinity}\; (\mathit{double}\; (\mathit{add}\; 1\; 2))$
   - $\mathit{add}\; (\mathit{cond}\; \mathit{True}\; 42\; (1 + \mathit{infinity}))\; 4$
   - $\mathit{twice}\; \mathit{double}\; (\mathit{add}\; 1\; 2)$
   - $\mathit{twice}\; (\mathit{add}\; 1)\; 0$

Note: There is not a mistake in the last expression; all you need to know for the time being is that a function application that doesn't have enough arguments is already in normal form. Just follow the rules when reducing the expression.

3. Give a reduction sequence for *fact* 3, where the factorial function *fact* is as defined in the lectures:

$$
\begin{aligned}
&\mathit{fact} :: \mathit{Integer} \to \mathit{Integer} \\
&\mathit{fact}\ 0 = 1 \\
&\mathit{fact}\ n = n * \mathit{fact}\ (n - 1)
\end{aligned}
$$

2 Basic types

1. Define $(||)$, using pattern-matching; and similarly, $(\&\&)$.

2. Define $(\&\&)$ and $(||)$ once *conditional expressions* (that is, **if**...**then**... **else**...). Note that there is more than one plausible way to do it, but only one correct way because of strictness. Now do the same using *guarded equations*.

3. Define a function *charToNum* that converts a digit character to its numeric equivalent; for example,

   charToNum `'3'` = 3

   (Hint: You will need the predefined function $ord :: Char \rightarrow Int$. To use it, add "**import** *Data.Char*" at the top of your Haskell file.)

4. Define a function *showDate* that takes three integers representing the day, month and year, and returns them formatted as a string (Hint: The $(+\!\!+)$ operator to appends two strings, and the *show* function converts a number to a string). For example:

   showDate 2 8 2004 = `"2 August 2004"`

   If that was too easy, make the day number an ordinal:

   showDate 2 8 2004 = `"2nd August 2004"`

3 Recursive definitions on lists

Using pattern-matching on lists, give recursive definitions of:

1. a function $prod :: [Int] \to Int$ that calculates the product of a list of integers;

2. a function $allTrue :: [Bool] \to Bool$ that determines whether every element of a list of booleans is true;

3. a function *allFalse* that similarly determines whether every element of a list of booleans is false;

4. a function $decAll :: [Int] \to [Int]$ that decrements each integer element of a list by one;

5. a function $convertIntBool :: [Int] \to [Bool]$ that, given a list of integers, converts any zeros to *False*, and any other number to *True*;

6. a function $pairUp :: [Int] \to [Char] \to [(Int, Char)]$ that pairs up corresponding elements of the two lists, stopping when either list runs out. For example:

$$
\begin{array}{lll}
pairUp\,[1,2,3]\,[\texttt{'a'},\texttt{'b'},\texttt{'c'}] & = & [(1,\texttt{'a'}),(2,\texttt{'b'}),(3,\texttt{'c'})] \\
pairUp\,[1,2]\,[\texttt{'a'},\texttt{'b'},\texttt{'c'}] & = & [(1,\texttt{'a'}),(2,\texttt{'b'})] \\
pairUp\,[1,2,3]\,[\texttt{'a'},\texttt{'b'}] & = & [(1,\texttt{'a'}),(2,\texttt{'b'})]
\end{array}
$$

7. a function $takePrefix :: Int \to [a] \to [a]$ that returns the prefix of the specified length of the given list (or the whole list, if it is too short);

8. a function $dropPrefix :: Int \to [a] \to [a]$ that similarly drops such a prefix (or the whole list, if it is too short);

9. a function $member :: Eq\ a \Rightarrow [a] \to a \to Bool$ that determines whether a given list contains a specified element.

10. a function $equals :: Eq\ a \Rightarrow [a] \to [a] \to Bool$ that determines whether two lists contain the same elements in the same order.

The definitions of the following functions deviate slightly from the usual pattern. You may find useful the function $error :: String \to a$, which takes a *String* as an error message and has whatever type you want. Give recursive definitions for:

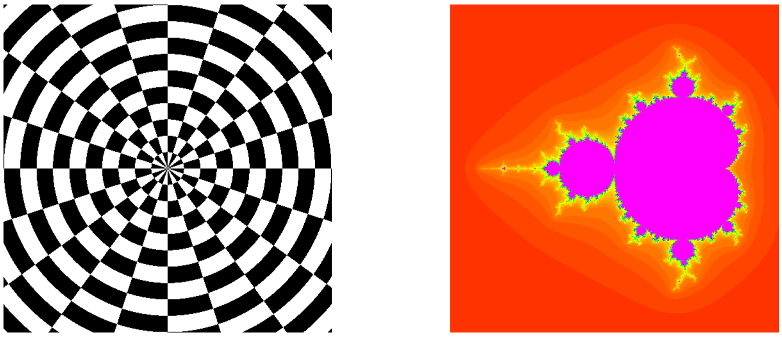
11. a function $select :: [a] \to Int \to a$ that selects the element of the list at the given position;
12. a function $largest :: [Int] \to Int$ that calculates the largest value in a list of integers;
13. a function *smallest* that similarly calculates the smallest value in a list of integers;

Some of your definitions probably have more general types than required; can you say which? (Hint: you will find model answers to all parts in the Haskell standard libraries; but they have been given different names here, so that you don't have name clashes.)

When we have covered the corresponding material in the lectures, you may want to return to consider which of these functions can be written more simply using *list comprehensions* or standard *higher-order operators* like *map* and *foldr*.

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 **??** we will generate some 'Op art' images such as the polar chequerboard in Figure **??**(a) below, and in Practical **??** some fractal images such as the *Mandelbrot Set* shown in Figure **??**(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.

$$\textbf{type}\ \mathit{Grid}\ a = [\,[\,a\,]\,]$$

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. (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.

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:

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

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

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

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

```haskell
catPic = bwCharView catBitmap
```

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

```haskell
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, False, False, False]
  ]
```

Here is another bitmap for you to experiment with:

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

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:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Point} = (\mathit{Integer}, \mathit{Integer}) \\
\mathit{catPoints} :: [\mathit{Point}] \\
\mathit{catPoints} = \\
\quad [(2,0),(8,0),(1,1),(3,1),(7,1),(9,1),(1,2),(4,2),(5,2),(6,2), \\
\qquad (9,2),(0,3),(10,3),(0,4),(3,4),(7,4),(10,4),(0,5),(5,5), \\
\qquad (10,5),(1,6),(9,6),(2,7),(3,7),(4,7),(5,7),(6,7),(7,7)]
\end{array}
$$

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

   $$\mathit{pointsBitmap} :: [\mathit{Point}] \to \mathit{Grid}\ \mathit{Bool}$$

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

   $$\mathit{catBitmap} = \mathit{pointsBitmap}\ \mathit{catPoints}$$

5. Define a function

   $$\mathit{gridPoints} :: \mathit{Grid}\ \mathit{Bool} \to [\mathit{Point}]$$

   to perform the reverse conversion, so that

   $$\mathit{pointsBitmap} \circ \mathit{gridPoints} = \mathit{id}$$

   (Why is it not the case that $\mathit{gridPoints} \circ \mathit{pointsBitmap} = \mathit{id}$ too?)

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],
  [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, h, 1, 1, 1, h, 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, 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, h, 1, 1, 1, h, 0, 0, h, 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, 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, 0, h, h, h, h, 0, 0, 1, 1, 1, 1, 1, h, 0, 0, h, 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, h],
  [0, 0, 0, h, h, h, h, 0, 0, 1, 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],
  [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],
  [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, 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, 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; so we define a ‘palette’ of shades:

```
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, varying linearly from ‘white’ to ‘black’.

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)
   ```

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?

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 **??**.

8. As you'll see from Section **??**, 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:

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

   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*.

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:

$$\begin{array}{l} pbmRender :: String \to Grid\ Bool \to IO\ () \\ pbmRender\ file = writeFile\ file \circ makePBM \end{array}$$

The first parameter is the filename; for example,

$$pbmRender\ \texttt{"test.pnm"}\ fprBitmap$$

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

10. Similarly, define functions

$$\begin{array}{ll} makePGM & :: Int \to Grid\ Float \to String \\ pgmRender & :: String \to Int \to Grid\ Float \to IO\ () \end{array}$$

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:

$$\begin{array}{l} fprGreymap :: Grid\ Float \\ fprGreymap = [\\ \quad [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0], \\ \quad [0,a,a,a,a,0,0,b,b,b,0,0,0,1,1,1,0,0], \\ \quad [0,a,0,0,0,0,0,b,0,0,b,0,0,1,0,0,1,0], \end{array}$$

[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]]
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
```

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*],

$[b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b, b]]$
$\textbf{where } r = RGB\ 7\ 0\ 0; o = RGB\ 7\ 3\ 0; y = RGB\ 7\ 7\ 0; b = RGB\ 0\ 0\ 0$

Define functions to translate an RGB grid into PPM format:

$$\begin{aligned} &makePPM \quad :: Int \to Grid\ RGB \to String \\ &ppmRender :: String \to Int \to Grid\ RGB \to IO\ () \end{aligned}$$

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 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
```

12. Define a function

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

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

$group\ 3$ `"ablewhackets"` $=$ `["abl","ewh","ack","ets"]`

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:

$$\begin{aligned}&pgmRead :: String \to IO\ (Grid\ Float)\\&pgmRead\ f = \mathbf{do}\ \{\, s \leftarrow readFile\ f; return\ (pgmParse\ (words\ s))\,\}\end{aligned}$$

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

$$\begin{aligned}&\mathbf{do}\ \{\, x \leftarrow pgmRead\ \texttt{"radcliffe64.pgm"};\\&\quad charRender\ (greyCharView\ (reverse\ charPalette)\ x)\,\}\end{aligned}$$

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

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, there is a 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/`

5 Recursive definitions on trees

The datatype declaration

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)\ \textbf{deriving}\ \mathit{Show}$$

defines binary trees. Use pattern-matching to give recursive definitions of:

1. a function $\mathit{size} :: \mathit{Tree}\ a \rightarrow \mathit{Integer}$ that calculates the number of elements in a tree;

2. a function $\mathit{tree} :: [a] \rightarrow \mathit{Tree}\ a$ that converts a list into a tree;

3. a function $\mathit{memberT} :: \mathit{Eq}\ a \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Bool}$ that determines whether a given tree contains a specified element;

4. a function $\mathit{searchTree} :: \mathit{Ord}\ a \Rightarrow [a] \rightarrow \mathit{Tree}\ a$ that converts a list into a search tree (a search tree is a tree in which, for a given node, all the values in the left subtree are smaller and all the values in the right subtree are larger);

5. a function $\mathit{memberS} :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Bool}$ that determines whether a given search tree contains a specified element (this should be more efficient than your definition of *memberT*);

6. a function $\mathit{inOrder} :: \mathit{Tree}\ a \rightarrow [a]$ that produces the list of elements from an in-order tree traversal (an in-order tree traversal is one in which the left subtree is traversed, then the root node, and then the right subtree).

5.1 More exercises on algebraic datatypes

There are some more exercises on various kinds of tree datatype in the optional practicals on Huffman Coding and the MILan programming language, which you can find towards the end of this handout. I consider them to be optional exercises; I expect you'll have time for them only if you're flying through the other exercises. I may work through some parts as 'live coding' during the course.

6 Functional images

This practical expores one way of generating bitmap images of the form that we were rendering in Practical **??**. But rather than cartoon cats, or lettering, or photographs, the images will be more geometric in nature.

The crucial observation is that a bitmap image assigns a shade—a bit, a grey value, or a colour value—to each pixel in the image; that is, it is a function from pixels to shades. Traditional image formats like GIF, JPG, and PNM, and the list-of-lists representation we used in Practical **??**, represent this function indirectly, by explicitly specifying its value for every possible element of the (finite!) domain of the function. But functional programming makes available another approach: we can represent the function from pixels to shades *directly*, by specifying instead the method for computing the shade of each pixel. That is, an image is a *function*, and operations that manipulate images are *higher-order functions*, functions that work on other functions.

This approach to image representation is based on the Pan language designed by Conal Elliott. Elliott wrote a chapter *Functional Images* for a book *The Fun of Programming* that I co-edited; see `http://conal.net/papers/functional-images/`, and especially browse through the gallery of images for inspiration. Pan arose out of earlier work of Elliott with Paul Hudak on *Functional Reactive Animation*, in which animations too are represented as functions—for example, from time to image. Elliott and Hudak's paper of that title from ICFP 1997 won the 'most influential paper' award ten years later for its impact on the field. That line of work has subsequently evolved into *Functional Reactive Programming*, which generalises from animated images to other time-dependent behaviours, such as robotics and graphical user interfaces; some of that work is covered in Hudak's textbook *The Haskell School of Expression*.

The essential idea is that a two-dimensional image is a function from points in the 2D plane to shades. We represent the 2D plane as complex numbers:

$$\textbf{type}\ \mathit{CF} = \mathit{Complex\ Float}$$
$$\mathit{point} :: \mathit{CF}$$
$$\mathit{point} = (-1.0) :+ 0.5$$

Complex numbers are of course an instance of Haskell's *Num* type class, so they come with an assortment of useful numeric functions such as addition and multiplication; we could just have used pairs, but then we would have had to define addition and multiplication ourselves. The constructor :+ constructs a complex number $x :+ y = x + i\,y$ from its 'real' and 'imaginary' parts x and y, where $i = \sqrt{-1}$. But you don't need to worry about imaginary numbers if you haven't encountered them before; just think of (x, y) as a point in the plane. There are two functions to extract the Cartesian coordinates of a point:

$$\mathit{realPart}, \mathit{imagPart} :: \mathit{CF} \rightarrow \mathit{Float}$$

and two more to extract the polar coordinates:

$$\mathit{magnitude}, \mathit{phase} :: \mathit{CF} \rightarrow \mathit{Float}$$

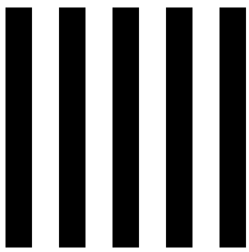
An image with shades of type c is simply a function from CF to c:

$$\textbf{type}\ \mathit{Image}\ c = \mathit{CF} \rightarrow c$$

For example, the following image has unit-width columns, alternating between black and white: a pixel is black (*True*) if the integer part of its x-coordinate is even, and white (*False*) if it is odd.

$$\begin{array}{l} \mathit{cols} :: \mathit{Image}\ \mathit{Bool} \\ \mathit{cols} = \mathit{even} \circ \mathit{floor} \circ \mathit{realPart} \end{array}$$

Note that an image is infinite in width and height, and also infinite in precision; when rendering it, we have to specify a finite window into the infinite whole, and specify at what resolution to sample the image within that window. So if we sample the image *cols* over 90×90 points evenly spaced between $0.05 :+ 0.05$ and $8.95 :+ 8.95$, we get the following bitmap:



(We've carefully chosen the coordinates to avoid whole numbers, to prevent possible rounding issues with *Float*s.)

1. Define a function

$$\mathit{grid} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{CF} \rightarrow \mathit{CF} \rightarrow \mathit{Grid}\ \mathit{CF}$$

that takes two integers m, n and two complex numbers p, q and constructs an $m \times n$ grid of complex numbers, evenly spaced between p and q. For example,

$$
\begin{aligned}
& \mathit{grid}\ 4\ 3\ (0 :+ 0)\ (3 :+ 2) \\
& \quad = [\,[\,0.0 :+ 2.0, 1.0 :+ 2.0, 2.0 :+ 2.0, 3.0 :+ 2.0\,], \\
& \qquad\ [\,0.0 :+ 1.0, 1.0 :+ 1.0, 2.0 :+ 1.0, 3.0 :+ 1.0\,], \\
& \qquad\ [\,0.0 :+ 0.0, 1.0 :+ 0.0, 2.0 :+ 0.0, 3.0 :+ 0.0\,]\,]
\end{aligned}
$$

Note that the y-coordinates are in reverse order, so that if the given points are the bottom-left and top-right corners then the first row is the top row.

2. Define a function

   $$\mathit{sample} :: \mathit{Grid}\ \mathit{CF} \to \mathit{Image}\ c \to \mathit{Grid}\ c$$

   that takes a grid of sample points (like that returned by *grid* above) and a continuous image, and samples the image at each of the given grid points. For example,

   $$
   \begin{aligned}
   & \mathit{charRender}\ (\mathit{bwCharView} \\
   & \quad (\mathit{sample}\ (\mathit{grid}\ 7\ 7\ (0.5 :+ 0.5)\ (6.5 :+ 6.5))\ \mathit{cols}))
   \end{aligned}
   $$

   should yield the following output:

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

   and

   pbmRender `"test.pbm"`
   (*sample* (*grid* 90 90 (0.05 :+ 0.05) (8.95 :+ 8.95)) *cols*)

   the image above.

3. Define an image

   $$\mathit{rows} :: \mathit{Image}\ \mathit{Bool}$$

   consisting of alternating black and white horizontal unit-height rows.

4. Define an image

   $$chequer :: Image\ Bool$$

   consisting of a chequerboard of unit-size squares. Can you reuse the definitions of *cols* and *rows*?

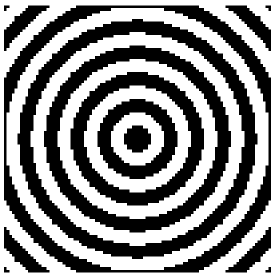
5. Define an image

   $$rings :: Image\ Bool$$

   of alternating black and white unit-width rings about the origin. For example,

   pbmRender `"test.pbm"`
   $(sample\ (grid\ 100\ 100\ (-(10 :+ 10))\ (10 :+ 10))\ rings)$

   should yield

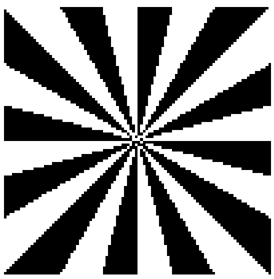


   (Now we need not worry so much about rounding errors, so we just sample at integral grid points.)

6. Define an image

   $$wedges :: Int \to Image\ Bool$$

   so that *wedges* n consists of $2n$ alternating black and white wedges, radiating from the origin; for example, *wedges* 12 yields



7. Can you provide definitions of *rings* and *wedges* in terms of *cols* and *rows*?

8. Define a polar chequerboard image with an even number of 'wedges'

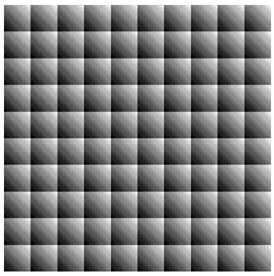
   $$polarChequer :: Int \rightarrow Image\ Bool$$

   like that at the start of Practical **??**.

9. All the images so far have been bitmaps; but the *Image* type is polymorphic in the shade type, so that's not required. Define a greyscale image

   $$shadedChequer :: Image\ Float$$

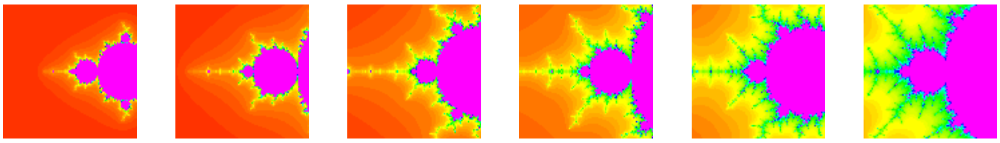
   consisting of a chequerboard in which each square is gradually shaded from black (shade 0.0) in the bottom left corner to white (1.0) in the top right:

10. Try to replicate some of the images in Conal Elliott's gallery at `http://conal.net/Pan/Gallery/`.

7 Fractals

We continue the series of practicals on bitmap images, now turning to fractal images such as the Mandelbrot set shown at the start of Practical **??**. The 'fractal' nature comes from self-similarity: the whole image of the Mandelbrot set is closely replicated at finer and finer scales. The following series of images starts at the same scale as shown in Practical **??**, only shifted a little to the left, and zooms in by a factor of two at each step on the bulb on the left; that bulb as its own little bulb on the left, which in turn has an even smaller bulb on the left, and so ad infinitum.



The Mandelbrot set is defined in terms of the following equation:

$$z_{n+1} = z_n^2 + c$$

For a given complex number c, this defines an infinite sequence $z_0, z_1, z_2 \ldots$ starting with $z_0 = 0$. We say that the given number c is in the Mandelbrot set if the magnitude of the z_i values in this sequence is bounded by some constant, and outside the set if the magnitude grows unboundedly. For example, with $c = 1$ the sequence starts $0, 1, 2, 5, 26 \ldots$ and grows unboundedly, so $c = 1$ is not in the Mandelbrot set; but with $c = -1$, the sequence oscillates $0, -1, 0, -1 \ldots$ with magnitude at most 1, and so $c = -1$ is in the set.

We can represent the equation by the function *next*:

```
next :: CF → CF → CF
next c z = z * z + c
```

1. Define a function to compute the trajectory of values for given c as an infinite list:

   ```
   mandelbrot :: CF → [CF]
   ```

   For example, *mandelbrot* 1 = [0, 1, 2, 5, 26...]. We will call a function like *mandelbrot* a 'trajectory function'.

It is in general impossible to tell whether a given point is in the Mandelbrot set: some cases are clearcut, but in general the trajectory might wander around indefinitely without either obviously converging or obviously diverging, in which case it is impossible to decide. But for the purposes of making pretty pictures, it suffices to approximate. We can take the first 'few' (say, 100) elements of the trajectory, and look to see whether they are all 'fairly close' to the origin (say, with magnitude at most 100). If these first few elements are all fairly close, we consider the point to be in the set; if not, we consider it to be outside the set.

2. Define a function

   $$fairlyClose :: CF \to Bool$$

   to approximate whether a point has not yet diverged.

3. Define a function

   $$firstFew :: [\,CF] \to [\,CF]$$

   to take the first few elements of an infinite sequence

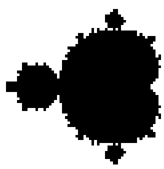
4. Hence define a function

   $$approximate :: (CF \to [\,CF]) \to Image\ Bool$$

   that takes a trajectory function like *mandelbrot* and yields a boolean image, black for the points c whose trajectory is (approximately) bounded, and white for the points whose trajectory is (approximately) unbounded. For example,

   ```
   pbmRender "test.pbm"
     (sample (grid 100 100 ((−2.25) :+ (−1.5)) (0.75 :+ 1.5))
             (approximate mandelbrot))
   ```

   samples the approximate Mandelbrot set to produce the following bitmap image:



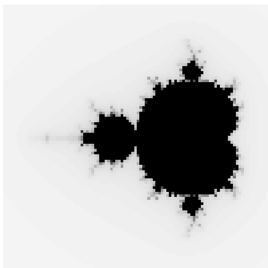
5. We can produce slightly more sophisticated pictures by asking a more sophisticated question: not just 'does the trajectory diverge?', but 'how quickly does the trajectory diverge?'. Define a function

   $$fuzzy :: (CF \rightarrow [CF]) \rightarrow Image\ Float$$

   that takes a trajectory function like *mandelbrot* and produces a greyscale image: a point c for which the first 'few' elements of the trajectory remain 'fairly close' should be black, as before, and a point c that is already beyond 'fairly close' should be white; but in between, points whose trajectories take some number of steps greater than zero but less than a 'few' to diverge should be some corresponding shade of grey. For example,

   ```
   pgmRender "test.pgm" 255
     (sample (grid 100 100 ((−2.25) :+ (−1.5)) (0.75 :+ 1.5))
             (fuzzy mandelbrot))
   ```

   yields this greyscale image:

6. We can produce prettier pictures still by colouring the images. There is still only one dimension of information in the pixels, so the three-dimensional colouring is not adding any information (although it can make details easier to see); this is called *pseudo-colouring*, and you may have seen it also in physical images such as heatmaps and relief maps. We define a palette of colours:

   ```
   rgbPalette :: [RGB]
   rgbPalette =
     [RGB i 0 15 | i ← [15, 14 .. 0]] ++    -- purple to blue
     [RGB 0 i 15 | i ← [0 .. 15]]      ++    -- blue to cyan
     [RGB 0 15 i | i ← [15, 14 .. 0]] ++    -- cyan to green
   ```

$$[\,RGB\ i\ 15\ 0 \mid i \leftarrow [\,0\,..\,15\,]\,] \quad +\!\!+ \quad \text{-- green to yellow}$$
$$[\,RGB\ 15\ i\ 0 \mid i \leftarrow [\,15, 14\,..\,0\,]\,] \quad \text{-- yellow to red}$$

Now define a function that will take a greyscale image such as that of the fuzzy Mandelbrot set above, and make a pseudo-colour image from it.

$$\mathit{ppmView} :: [\,\mathit{RGB}\,] \to \mathit{Grid\ Float} \to \mathit{Grid\ RGB}$$

(Hint: you already did most of the work in an earlier exercise.) For example,

$$\begin{array}{l}\mathit{ppmRender}\ \texttt{"test.ppm"}\ 15\ (\mathit{ppmView\ rgbPalette} \\ \quad (\mathit{sample}\ (\mathit{grid}\ 100\ 100\ ((-2.25) :\!+ (-1.5))\ (0.75 :\!+ 1.5)) \\ \qquad\qquad (\mathit{fuzzy\ mandelbrot})))\end{array}$$

produces the pseudo-coloured Mandelbrot image at the start of Practical **??**.

7. Another kind of fractal image is defined using the same equation

$$z_{n+1} = z_n^2 + c$$

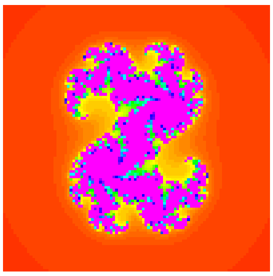
as before, but this time using a fixed constant for c and starting with z_0 being the candidate point rather than the origin. Define a function

$$\mathit{julia} :: \mathit{CF} \to \mathit{CF} \to [\,\mathit{CF}\,]$$

so that *julia c* is the trajectory function for a given constant c. For example, for $c = 0.32 :\!+ 0.043$ we can use

$$\begin{array}{l}\mathit{ppmRender}\ \texttt{"test.ppm"}\ 15\ (\mathit{ppmView\ rgbPalette} \\ \quad (\mathit{sample}\ (\mathit{grid}\ 100\ 100\ ((-1.5) :\!+ (-1.5))\ (1.5 :\!+ 1.5)) \\ \qquad\qquad (\mathit{fuzzy}\ (\mathit{julia}\ (0.32 :\!+ 0.043)))))\end{array}$$

to yield this so-called Julia set:



8 Huffman Coding (optional)

This practical provides another application of tree data structures. It consists of two parts:

- the development of a balanced dictionary abstract data type;
- the use of this dictionary in the implementation of a Huffman coding program, which provides a mechanism for compressing ASCII text.

For example, the text

```
As software becomes more and more complex, it
is  more  and  more important to structure it
well.  Well-structured  software  is  easy to
write,   easy   to   debug,  and  provides  a
collection  of modules that can be re-used to
reduce future programming costs. Conventional
languages place a conceptual limit on the way
problems   can   be  modularised.  Functional
languages  push  those  limits  back. In this
paper we show that two features of functional
languages    in    particular,   higher-order
functions and lazy evaluation, can contribute
greatly  to  modularity.  Since modularity is
the key to successful programming, functional
languages  are  vitally important to the real
world.
```

might be Huffman coded to the following data:


```
%%Begin HaskellEncode%%
5
:W#G@2A9,?FQXGPB\F%OPB\F&/$O#^$(:E=X"+R8"WX"+
R8V);RBV@1QR21B$DQJ5H/,: 0\\QJY)(Q"2?@.>!*%DP
-X UZD(]:)U#A  UZD (X \?Y"H0"WX N7^[X^ 6K'S#8
FO8#P"+^)A\$;5 Q;'YB3:B<?@CUD_$88@(A),%RY4K&.
V4&/R<T#7>_IVFO:Y7%LHBE'P+Q6&%#'MAEH1<&;'4#V$
;,4+34N7_PX\ &+8 _,"+^)BSO'YH!#"6Q->URN+91%*/
@"R<V C;\8&;'4X#]6?+00VPC>]4M2SD4&.;>@(VJ!*'$
!JA)/@\ @);$U[7*XME$4H^  VP "UE-83%D !O2FSEJ>
?'-8"6Q->^"WX8OQI!_5PBIKW"#%L,>TG?R(=2I-4RD!'
 B_B8LZE+0"';;#$7\3%G4I&]5&S'R=(1QQ&,/N@)@+ER
I6,=LJ$("6Q->URN+91%*/@%DP/NHN92-B6\HMH$<(V8D
```

```
URM#SGF@ 5
%%End   HaskellEncode%%
```

Here is a textual representation of the dictionary used to encode the document.

```
%%Begin HuffDict %%
      10101          000   ,   1000010   -  01101010
.   0110100  A 011010110  C 011010111  F 100001100
I 100001101  S 100001110  W 100001111  a      0101
b    111111  c     01100  d     11110  e      0011
f    100000  g    001010  h    011011  i      1101
k  11111001  l      1100  m     10001  n      1011
o      0111  p    001011  r      1001  s      1110
t      0100  u     00100  v   1111101  w    101000
x 111110000  y    101001  z 111110001
%%End   HuffDict %%
```

Given a dictionary and hhencoded document, the original text can be decoded. The files for this practical contain some data declarations and type signatures for you to fill in. *Ask the practical assistant if you are unsure where to obtain these files.*

8.1 Dictionaries

A dictionary can be constructed using an instance of the tree implemented in the previous practical. The trees we are interested in contain two pieces of data: the key, which will guide the search, and the value associated with that key (for example, a dictionary might contain words as keys and definitions as values).

The type for such a tree is given by the `DictTree` datatype, which is simply a synonym for a tree containing keys and values in its nodes.

```
> type DictTree a b = Tree (a, b)
```

which can be found in `DictTree.lhs`

The keys in a search tree do not speed up searches if the tree is severely unbalanced. It is all too easy to construct such trees. For example, the simplest way of defining `addDict` is to recursively add new data into the left subtree if it is smaller than the value at the root, or into the right subtree if it is larger. Thus, if a relatively large amount of new data is added in sorted order, a degenerate tree will always be the result. Of course, if it is known that the new data will be added in random order, then a more balanced tree will result. When all or most of the data that will go into the tree is available all at once, the best strategy is to fully sort it and then carefully construct a balanced tree.

The following exercises are concerned with the implementation of a dictionary using a `DictTree`. The type alias `Dict` and the associated function names and type signatures define a dictionary type and an interface to it:

```
> module Dict
>   (Dict, addDict, lookupDict, dict2List, list2Dict, showDict)
> where
>
> addDict    :: Ord a => Dict a b -> a -> b -> Dict a b
> lookupDict :: Ord a => Dict a b -> a -> Maybe b
> dict2List  :: (Ord a, Ord b) => Dict a b -> [(a,b)]
> list2Dict  :: [(a,b)]  -> Dict a b
> showDict   :: (a -> [Char]) -> (b -> [Char]) ->
>               Dict a b -> [Char]
```

The file `Dict.lhs` contains these signatures and dummy definitions for them. Replace them with your own definitions according to the instructions below.

1. Define a function

   ```
   dict2List :: Dict a b -> [(a,b)]
   ```

   that converts a dictionary of key-value pairs into a list containing all the key-value pair in the dictionary as tuples.

2. Define a function

   ```
   list2Dict :: (Ord a, Ord b) => [(a,b)] -> Dict a b
   ```

   that converts a list of key-value tuples into a *balanced* dictionary. For example,

   ```
   ? list2Dict [(2,'b'),(1,'a'),(4,'d')]
   ```

   should create a tree that has the following shape:

```
     ,---|
     |
 ,--{ 4 -> 'd'}
 |   |
 |   `---|
 |
-{ 2 -> 'b'}
 |
 |   ,---|
 |   |
 `--{ 1 -> 'a'}
     |
     `---|
```

Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

3. Define a function

```
addDict :: Ord a => Dict a b -> a -> b -> Dict a b
```

that adds a key-value pair to a dictionary and produces a dictionary as a result. Test your function on appropriate test data, cutting and pasting the example evaluations into your file. *(Hint: the function need not preserve balancing. If there is already an entry for that key, that it should be replaced by the new data.)*

4. Define a function

```
  lookupDict :: Ord a => Dict a b -> a -> Maybe b
```

that returns the corresponding value for the key of a dictionary using the `Maybe` type and returning `Nothing` to signify a failed lookup. For example,

```
  lookupDict (list2Dict [(2,'b'),(1,'a'),(4,'d')]) 2
    = Just 'b'
  lookupDict (list2Dict [(2,'b'),(1,'a'),(4,'d')]) 5
    = Nothing
```

Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

8.2 Constructing a frequency table

In this section, we will construct a dictionary containing the number of occurrences of each character within a piece of text. For example:

```
dict2List (frequencies "abbccccbbcacacb") = [ (a,4), (b,5), (c,7) ]
```

Two possible algorithms for constructing the frequency dictionary are:

- Sorting the text, which will produce a list of characters in which similar characters are grouped together. For example,

  ```
  sort "functional" = "acfilnnotu"
  ```

  The list can then be sub-divided into a list of lists such that the inner lists contains "runs" of the same character. This can be trivially converted into a dictionary containing the number of occurrences of each character by using the `list2Dict` function. *(Hint: The `Data.List` module has a function group that may be useful.)*

- Start off with a balanced dictionary, mapping every possible letter in a document to zero. For each character x of the input, replace the associated value $freq$ of x by $freq + 1$.

The file `Huff.lhs` contains the following type synonym

```
> type FreqDict = Dict Char Int
```

Use this type alias in defining functions over frequency dictionaries.

5. Define a function

   ```
   frequencies :: String -> FreqDict
   ```

   that creates a dictionary containing the number of occurrences of each character within a string. For example,

   ```
   ? frequencies "abbccccbbcacacb"
       ,---|
       |
    ,--{ 'c' -> 7}
    |  |
    |  `---|
    |
   ```

```
-{ 'b' -> 5}
 |
 |   ,---|
 |   |
 `--{ 'a' -> 3}
     |
     `---|
```

Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

8.3 Constructing a Huffman tree

The conventional representation of a textual document on a computer uses the ASCII character encoding scheme. The scheme uses eight bits to represent a character; a document containing 1024 characters will therefore occupy one kilobyte. The idea behind Huffman coding is to use the fact that some characters appear more frequently in a document. Therefore, Huffman encoding moves away from the fixed length encoding of ASCII to a variable length encoding in which the frequently used characters have a smaller bit encoding than rarer ones. For example the table shown below gives the ASCII and a possible Huffman encoding for five letters:

Character	ASCII encoding	Huffman encoding
‘a’	001100001	1
‘e’	001100101	01
‘t’	001110100	000
‘z’	001111010	0010
‘@’	001000000	0011

It is important that the codes are chosen in such a way that an encoded document gives a unique decoding that is the same as the original document. The reason why the codes shown in the document above are decipherable is that *no code is an initial segment of any other code*.

To create such a table, we first take a list of character-frequency pairs (which can be constructed using the functions implemented so far) and construct a Huffman tree. A Huffman tree is an instance of a *labelled leaf tree*. A *leaf tree* is a tree in which all data is held in the leaves. Such a tree can be defined by the datatype declaration

```
> data LeafTree a = Leaf a | Branch (LeafTree a) (LeafTree a)
```

Now, suppose there is some function `f :: LeafTree a -> b` that combines the elements of a tree by transforming the elements in the leaves of tree into values of type `b`, and then combines the two subtrees of a branch into a single value `b`. (Such a transformation rather common. As an example, consider a tree of strings for which you want to find the total length of all strings. At the leaf, the length of an individual string would calculated, and the lengths of the subtrees would be added together at each branch.) This is called a *reduction*.

If such a reduction is to be performed frequently, it might be preferable to calculate it once and store the result. This can be accomplished by caching the result of the reduction in the branches of the tree. This means that the branches are *labelled* with some extra information. Such a tree can be defined by the datatype declaration

```
> data LabTree a b
>   = LabLeaf a | LabBranch b (LabTree a b) (LabTree a b)
```

where `b` is the type of the cached value. These datatype declarations are found in `LabTree.lhs`.

A Huffman tree is a labelled leaf tree in which the leaves are characters/frequency pairs, and the reduction is the "weight" of a tree. The weight of a character is its frequency, and the weight of a branch is the sum of the weights of its subtrees. It is defined by the type alias

```
> type WeightedHuffTree = LabTree (Char,Int) Int
```

in `Huff.lhs`.

Given a list of character-frequency pairs, the algorithm that constructs a Huffman tree can be expressed as:

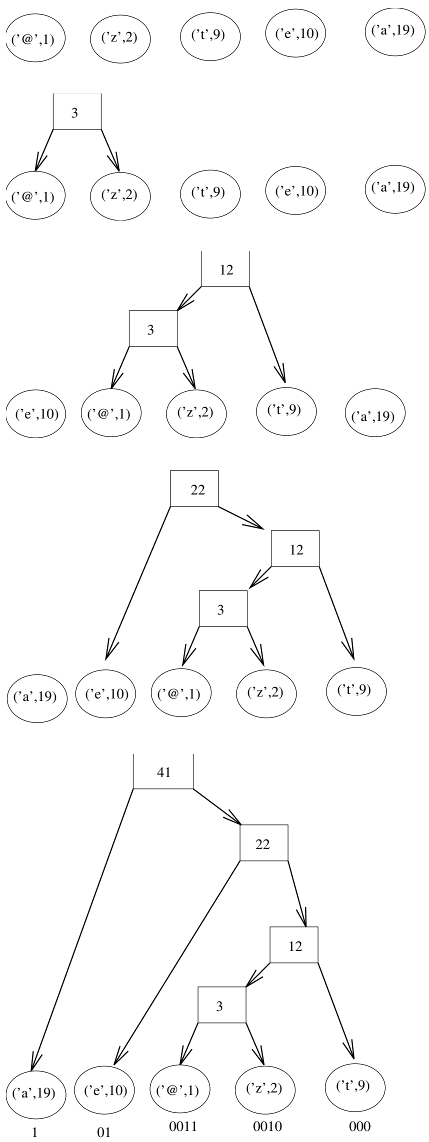
- sort the list of frequencies on the *frequency* part of the tuple—i.e less frequent characters will be at the front of the sorted list;

- convert the list of character-frequency pairs into a list of the leaves containing the character-frequency pairs—each leaf is of course a tree;

- take the first two trees off the list, and form a branch; insert this branch into the remaining list of trees in such a way that the resulting list is sorted on the "weight" of the trees.

- repeat the previous step until a singleton list containing the Huffman tree for the character-frequency pairs is created.

For the list of frequencies `[('e',10),('a',19),('t',9),('z',2),('@',1)]` the following diagrams show the algorithm pictorially.



Here is the resulting tree:

```
?  (mkHuffTree
     (list2Dict [('e',10),('a',19), ('t',9),('z',2), ('@',1)]))

       ,---t -> 9
       |
    ,--{ 12}
    |  |
```

```
     |   |   ,---z -> 2
     |   |   |
     |   `--{ 3}
     |       |
     |       `---@ -> 1
     |
 ,--{ 22}
 |   |
 |   `---e -> 10
 |
-{ 41}
 |
 `---a -> 19
```

To ensure that you produce exactly the same codings from the same frequencies every time, use the following constraints when building the tree:

- a frequency dictionary is converted to a list of tuples
- `sort` the list of tuples computed from the frequency dictionary. As the first part of each tuple will be a unique character, the list will be sorted by ascending ASCII values;
- apply a stable sort (one for which the order of equivalent elements within the list is not changed) to the list, that sorts by the second part of the tuple (the frequency). Together with the previous step, this enforces an ordering on any list of tuples, irrespective of the input order;
- when constructing the Huffman tree, insert a combined tree into the list of trees *before* any tree with equal weight.

6. Write a stable sorting function

   ```
   sortOnFreq :: Ord b => [(a,b)] -> [(a,b)]
   ```

   that sorts the argument list of tuples by using the second part of each tuple as the sort key. Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

7. Write a function

   ```
   weight :: WeightedHuffTree -> Int
   ```

that returns the weight of a weighted tree. Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

8. Write a function

```
insert :: WeightedHuffTree -> [WeightedHuffTree]
            -> [WeightedHuffTree]
```

   that inserts a weighted tree into a list of weighted trees sorted by weight, returning a similarly sorted result. (Remember, the tree should be inserted into the list *before* any tree of equal weight!)

9. Write a function

```
combine :: [WeightedHuffTree] -> [WeightedHuffTree]
```

   that combines the first two elements of the list of weighted trees into a single tree, then inserts the combined element into the remaining part of the list.

10. Write a function

```
mkHuffTree :: FreqDict -> WeightedHuffTree
```

   that creates a weighted Huffman tree from a frequency dictionary. Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

11. Write a function

```
huffCodeFor :: WeightedHuffTree -> Char -> [[Bool]]
```

   which generates the code of a character according to a particular Huffman tree. To generate a code for a given character, encode the path taken from the root to the leaf containing the target character such that taking the left subtree is represented by a 1, and the right by a 0. In our case, we are using a list of `Bool`s to represent our binary number, where a 0 corresponds to `False` and `True` corresponds to 1.

   Use the "list of successes" technique, such that if a character is encoded by the tree, then a *singleton* list is returned containing the binary encoding (represented by a list of booleans) of that character. For example, if

```
t = mkHuffTree (frequencies "Hellooooooooooooooo")
```

then

```
huffCodeFor t 'o' = [[False]]
huffCodeFor t 'l' = [[True,False]]
huffCodeFor t 'e' = [[True,True,False]]
huffCodeFor t 'x' = []
```

12. Write a function

    ```
    mkHuffDict :: WeightedHuffTree -> HuffDict
    ```

    that creates a dictionary containing characters as its keys, and a list of booleans representing the binary encoding for that character as values. The `HuffDict` type alias can be found in `Huff.lhs`. Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

8.4 Encoding ASCII text

It is now possible to Huffman encode a document represented as Haskell `String`. This can be accomplished by

- constructing a frequency table for the source document;
- using the frequency table to create a Huffman dictionary;
- and using the Huffman dictionary to map each character to its corresponding Huffman code

13. Write a function

    ```
    huffiseDocument :: HuffDict -> [Char] -> [Bool]
    ```

    such that, given a Huffman dictionary, converts the string of ASCII text into a sequence booleans representing the Huffman coding of the document. Test your function on appropriate test data, cutting and pasting the example evaluations into your file.

8.5 Decoding a Huffman binary

Decoding a list of booleans is best done all at once, that is, by converting the entire list into a list of characters. Intuitively, this means using the list of booleans as commands that guide you from the root of the tree towards a leaf. However, when you reach a leaf, you haven't necessarily finished—if there are more booleans, these must also be decoded and the resulting list of characters becomes part of the answer. Whenever the function restarts at the root of the Huffman coding tree in this way, it needs access to that tree again. This is achieved by giving the function two copies of the tree, one to traverse, and one to use when starting again.

The full specification of this function is as follows: `decodeHuffBin t1 t2 bs` is a list of characters such that the code for the first is a prefix of `bs`, according to the tree `t2`, while the remaining characters are the decoded version of the suffix of `bs`, decoded according to the tree `t1`.

The base case isn't obvious, but everything works out if a character is generated whenever a leaf is found, and the rest of the input is decoded by starting again at the root of the whole tree:

```
> decodeHuffBin :: DecodeHuffTree -> DecodeHuffTree -> [Bool]
>                    -> [Char]
> decodeHuffBin t (Leaf c) [] = [c]
> decodeHuffBin t (Leaf c) bs = c:decodeHuffBin t t bs
```

Note that strange results are obtained if the encoded document contains just one character, repeated many times—for which the Huffman tree consists only of a `Leaf`, and the path describing it is empty.

The recursive cases are simple in appearance, but based on a tricky idea. The first character coded by a list of booleans `True:bs` against a Huffman tree `Branch left right`, is the same as the first character coded by the list `bs` against the Huffman tree `left`. Similarly, the first character coded by a list of booleans `False:bs` against a Huffman tree `Branch left right`, is the same as the first character coded by the list `bs` against the Huffman tree `right`. However, once all of the bits representing the first character have been removed from the list, the characters represented by what remains is just the decoding described by the whole of the original tree.

In the recursive case, it is therefore necessary to decode using the correct child, and to pass the whole tree down so that the remaining parts of the list can be decoded:

```
> decodeHuffBin t (Branch left right) []
```

```
>   = error "ran out of bits"
> decodeHuffBin t (Branch left right) (True:bs)
>   = decodeHuffBin t left  bs
> decodeHuffBin t (Branch left right) (False:bs)
>   = decodeHuffBin t right bs
```

The file `Huff.lhs` contains a definition of the `decodeHuffBin` function.

8.6 Printing out a Huffman dictionary

In addition to outputting Huffman-coded text, it is necessary to be able to "bundle" the dictionary used to encode it, so that it can be decoded later. In order to accomplish this, we must be able to represent Huffman dictionaries as strings, and then reconstruct a dictionary from its string representation.

14. Define a function

    ```
    binToZeroOnes :: [Bool] -> [Char]
    ```

    that converts a list of booleans into an ASCII representation such that the character 1 represents `True`, and 0 represents `False`. For example,

    ```
    binToZeroOnes [True,False,True,True] = "1011"
    ```

15. An ASCII representation of a Huffman dictionary consists of a pair of start/end delimiters, and an encoding for each key-value pair in the dictionary. Given the pair `(c,binary)`, the ASCII encoding should print out `c`, followed by the tab character (`\t`), followed by a series of ones and zeros, and finished by another tab. The tabs are used as delimiters for each entry in the table. For example,

    ```
    ? pprHuffDict (mkHuffDict (mkHuffTree (frequencies "hello")))
    %%Begin HuffDict %%
    e       001     h       000     l       1       o       01
    %%End HuffDict %%
    ```

    Write a function

    ```
    pprHuffDict :: HuffDict -> [Char]
    ```

    that prints out a textual representation of a Huffman dictionary.

8.7 Reading an ASCII Huffman dictionary

16. Define a function

```
zeroOnesToBin :: [Char] -> [Bool]
```

that converts a list of zero-one characters into a binary representation of the string. For example,

```
zeroOnesToBin "0010101"
  = [False,False,True,False,True,False,True]
```

17. Write a function

```
readHuffDict :: [Char] -> [(Char,[Bool])]
```

that converts the ASCII representation of a Huffman dictionary produced by the `pprHuffDict` into a list of tuples containing a character and a list of booleans. *Hint: You can convert the original input list of characters into a list of lines using* `lines`*.*

18. Write a function

```
mkDecodeHuffTree :: [(Char,[Bool])] -> DecodeHuffTree
```

that creates a leaf tree of characters (recall that a leaf tree is called a `Tree` in `Tree.lhs`) such that the path down to a given leaf encodes the binary Huffman code for that character. For example,

```
? showDecodeHuffTree (mkDecodeHuffTree
    [('a',[True]),('b',[False,True]),('c',[False,False])])
     ,---'c'
     |
  ,--{ ' '}
  |  |
  |  `---'b'
  |
-{ ' '}
  |
  `---'a'
```

Test your function on suitable test input, cutting and pasting the example evaluations into your file.

8.8 ASCII Encoded Binaries

In the previous section we developed a Huffman encoding that was used to provide a mechanism for compressing ASCII text. In this section we focus on the implementation of a function similar to the Unix `uuencode` and `uudecode` commands which converts a *binary* file into an ASCII-encoded representation that can be sent using mail.

The first step is to implement a `uuencode`-like function. That is, a function that will convert a binary file (in this case represented by `[Bool]`) into an ASCII string (which would be useful for e.g. sending a file via email). Naturally, a corresponding decode function is necessary to recover the file. The type signatures and dummy definitions can be found in `Encode.lhs`.

19. Write a function

    ```
    binaryToNum :: [Bool] -> Int
    ```

    such that, given a binary number encoded as a list of booleans, the corresponding integer is returned by the function. For example,

    ```
    binaryToNum [True,False,True,True] = 11
    binaryToNum [False,False,True,True] = 3
    ```

20. Write a function

    ```
    binaryToPrintChar :: [Bool] -> Char
    ```

    that converts a binary number encoded as a list of booleans into a printable character ASCII character. Because ASCII characters below 32 are not printable, it is necessary to offset all numbers by adding 32 before converting to a character. For example,

    ```
    binaryToPrintChar [True,False,True,True] = '+'
    binaryToPrintChar [True,False,True,True,False] = '6'
    ```

21. Write a function

    ```
    groups :: Int -> [a] -> [[a]]
    ```

    such that `groups n xs` chunks the list `xs` into sub-lists each of length `n`—the last list in the resulting list may be smaller. For example,

```
groups 2 [1..7] = [[1,2],[3,4],[5,6],[7]]
groups 3 [1..7] = [[1,2,3],[4,5,6],[7]]
```

22. Write a function

    ```
    hhencode :: [Bool] -> [Char]
    ```

    that takes a sequence of binary numbers, represented as booleans, and converts it to a printable string.

    A printable ASCII character is within the range ASCII-32 to ASCII-126—i.e 94 different characters. Six binary bits have a range which is within the range of the printable ASCII characters. hhencode should use the following file format:

    - the start of the encoding document is identified by the string %%Begin HaskellEncode%%;
    - the binary list given to the function is split into sub-lists each of size six;
    - the last sub-list of this list can cause problems as it doesn't have *six* characters in it — we want to be able to distinguish between the six binary bits 000101 and the three binary bits 101 — so directly following the start of document delimiter should be a number that represents the size of the last chunk in the hhencoded document;
    - each chunk of six bits is converted into a printable ASCII character, these characters forming the body of the hhencoded document ;
    - each line of the document must contain at *most 70* characters;
    - the end of the hhencoded document is identified by the string %%End HaskellEncode%% — unfortunately this string could also occur in the body of the document, but the chances of this happening are 2901062411314618233730627546741369470976 to 1.

    For example,

```
? hhencode (concat (copy [True,True,False,True,False] 500))
%%Begin HaskellEncode%%
4
UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:
```

```
UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:
UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:
UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:
UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:
UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:UK6M:U*
%%End HaskellEncode%%
```

8.9 Haskell to Haskell decode

23. Write a function

```
numToBinary :: Int -> [Bool]
```

that converts a number into a binary representation. For example,

```
numToBinary 5 = [True,False,True]
numToBinary 67 = [True,False,False,False,False,True,True]
```

24. Write a function

```
printableCharToPadBinary :: Int -> Char -> [Bool]
```

such that `printableCharToPadBinary x c` converts the printable character `c` to a number based at zero instead of 32, but ensures that the binary representation contains at least x bits. For example,

```
printableCharToPadBinary 6 ' '
  = [False,False,False,False,False,False]
printableCharToPadBinary 6 '~'
  = [True,False,True,True,True,True,False]
```

25. Write a function

```
hhdecode :: [Char] -> [Bool]
```

that decodes the encoded binary produced by the hhencode function.

8.10 Gluing it all together

It is now possible to tie these two features together into a program that compresses an ASCII document using Huffman coding, and then converts the binary out into an ASCII document. The type signatures and dummy definitions for these functions can be found in `Huff.lhs`.

26. Write a function

    ```
    huffAndHaskellEncode :: [Char] -> [Char]
    ```

    such that the document is first traversed to build a Huffman table specific to that document, and then the document is Huffman encoded using that dictionary. The file format for the function should print out the Huffman dictionary of the document followed by a hhencoded version of the binary encoded document. You can use the function `processFile` from `Huff.lhs` to read a string from a file, process it, and write it back to a different file. For example,

    ```
    ? processFile "joke.txt" huffAndHaskellEncode "joke.hh"
    ```

27. Write a function

    ```
    huffAndHaskellDecode :: [Char] -> [Char]
    ```

    such that a document produced by the `huffAndHaskellEncode` is decoded. For example,

    ```
    ? processFile "joke.hh" huffAndHaskellDecode "joke2.txt"
    ```

28. For a given test document, compare the size of the original document with a Huffman/hhencoded version of the document. Encode the encoded document a second time—why isn't a doubly encoded document smaller than a singly encoded document?

29. Decode the hhencoded/Huffman coded file `message.hh`. Cut and paste the decoded document into your file.

9 MILan, Part 1 (optional): Execution

This practical is the first in a series about *MILan*, a 'miniscule imperative language'. The practicals will use a parser, compiler, executor and interpreter for a simple programming language to illustrate various concepts taught in the course. The whole suite (written by Colin Runciman) is made up from eleven modules, whose dependencies are illustrated in Figure **??**.

You'll find the source files (for all practicals in this series) in the directory `milan`. The initial version of the language can only handle a series of assignment statements, and the expressions can only be constants or variables. In the course of the practicals, we will extend the language to cover recursive expressions and `if`- and `while`-statements.

Part 1 concerns an *executor* for a simple assembly language — a virtual machine for MILan, if you like. It will provide you with some more examples of programming with lists, and with non-recursive user-defined datatypes.

9.1 Values

Look in the file `Value.lhs`, and you will find a module defining the values in MILan. This isn't very exciting so far; the type `Value` contains integers and an extra 'error' value.

```
> data Value
>   = Numeric Int
>   | Wrong
>   deriving (Eq, Show, Read)
```

9.2 Behaviours

Executing or interpreting a MILan program will yield a *trace*. This will contain all the values output by the program (MILan programs cannot read input!), but also extra information about its behaviour, such as whether it has crashed. In `Behaviour.lhs` you will find the following definitions:

```
> type Trace a = [Event a]
> data Event a
>   = Output a
>   | End
>   | Crash
```

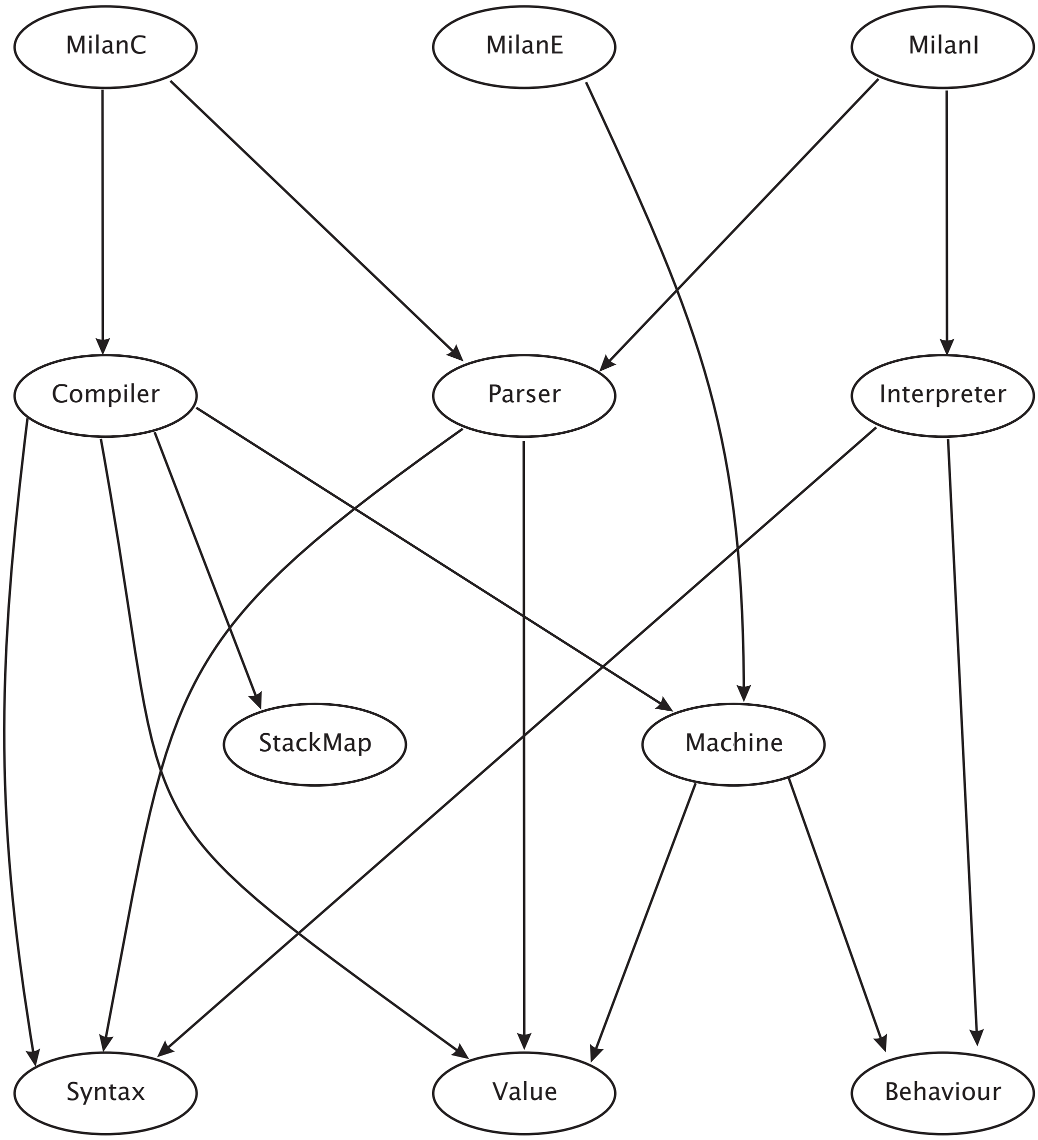


Figure 2: Dependencies between MILan modules

```
>   deriving (Eq, Show)
```

Thus, a `Trace` is a list of `Events`, each of which consists of an output value, or an indication that the program has terminated normally or abnormally. The intention is that every trace should contain either `End` or `Crash` as its last element, and neither anywhere else in the trace; but we cannot enforce that yet.

We also define an operation for concatenating `Traces`. The initial definition just uses list concatenation:

```
> (+++) :: Trace a -> Trace a -> Trace a
> s +++ t = s ++ t
```

1. This definition of trace concatenation breaks the invariant of `Traces` stated above; how? Give a better definition that does not. Bear in mind that a normally-terminating trace ought to run on into the following trace, whereas an abnormally-terminating trace ought to cancel a following trace. (We say that `End` is a *left-unit* of concatenation, whereas `Crash` is a *left-zero*. Can you see why we use these terms?)

9.3 The virtual machine

The virtual machine is described in `Machine.lhs`. It maintains a stack of values. There are machine instructions for manipulating the stack (pushing, popping, and modifying the element at a particular position), for output, and for termination.

```
> data Instruction
>   = Push Value
>   | Pop
>   | Fetch Int
>   | Store Int
>   | Display
>   | Halt
>   deriving (Eq, Show, Read)
```

The function `exec` executes a list of instructions, yielding the trace of the program's behaviour:

```
> exec :: [Instruction] -> Trace Value
> exec instrs = snd (run instrs 0 [])
```

This starts the program with ‘program counter’ (an index into the list of instructions) zero and an empty stack. (There is also a function execDebug, which returns the final stack as well as the behaviour; you will probably find this useful for forensic purposes.)

The function run takes the program, the program counter (initially zero), and the stack (initially empty), and yields a final stack and a behaviour. Here we make use of a let .. in .. expression, which introduces named variables with given values that are used in the body of the expression.

```
> run :: [Instruction] -> Int -> [Value] -> ([Value],Trace Value)
> run pg pc st
>   | pc < 0 || length pg <= pc = (st, [Crash])
>   | pg !! pc == Halt          = (st, [End])
>   | otherwise                 = let (pc',st',tr') = step pg pc st
>                                     (st'',tr'')   = run pg pc' st'
>                                 in  (st'', tr' +++ tr'')
```

If the program counter is out of range, the program crashes; if the current instruction is Halt, the program terminates normally; otherwise a single instruction is executed, yielding an intermediate state from which the remainder of the program is run. Note that the single instruction may produce a little bit of behaviour, which is concatenated with the behaviour of the rest of the excecution.

The function step deals with a single step of the execution.

```
> step :: [Instruction] -> Int -> [Value] -> (Int, [Value], Trace Value)
> step pg pc st =
>   case (pg !! pc, st) of
>     (Push x       , stack)     -> (pc', x : stack, [End])
>     (Pop          , _ : stack) -> error "Pop not implemented"
>     (Fetch n      , stack)     -> error "Fetch not implemented"
>     (Store n      , x : stack) -> error "Store not implemented"
>     (Display      , i : stack) -> (pc', stack, [Output i, End])
>     (_            , stack)     -> (pc', stack, [Crash])
>   where
>     pc' = pc + 1
```

A case analysis is performed on the current instruction and the stack. A Push operation involves pushing an element on the stack, and no behaviour. A Display operation pops a value from the stack, and generates some output. Any unrecognized operation causes the program to crash.

2. Execute the two programs

```
[Push (Numeric 3), Push (Numeric 4), Display, Display, Halt]
[Push (Numeric 3), Display, Push (Numeric 4), Display, Halt]
```

If the results are not what you expected, think carefully about what is going on!

9.4 Completing the initial version

3. Complete the definition of the Pop clause, in the obvious way.

4. Complete the definition of the `Fetch` clause. You will probably find list indexing (`!!`, as in the `case` statement) useful. The operand indicates the number of positions to count from the top of the stack; for example, `Fetch 0` duplicates the top of the stack, and `Fetch 1` takes a copy of the item one below the top.

5. Complete the definition of the `Store` clause. You may find it useful to define a function with the signature

```
>     replace :: Int -> a -> [a] -> [a]
```

The intention here is that, for example,

```
replace 1 'm' "apple" = "ample"
```

6. Execute the program swap1, defined in `MilanT1.lhs` to save you from having to type it out.

```
> swap1 = [Push Wrong, Push Wrong, Push Wrong, Push (Numeric 3),
>          Store 0, Push (Numeric 4), Store 1, Fetch 0, Store 2,
>          Fetch 1, Store 0, Fetch 2, Store 1, Fetch 0, Display,
>          Fetch 1, Display, Halt]
```

This is the output of the MILan compiler you'll define later, given the MILan source program

```
x := 3; y := 4;
z:=x; x:=y; y:=z;
print x; print y
```

As you might expect, it should output 4 then 3.

9.5 A command-line executor

In `MilanE.lhs` you will find a little Haskell program that uses monadic I/O to read a filename as a command-line argument and read the contents of that file as a list of instructions; it then runs the program and prints the resulting trace.

You can compile it with ghc, by typing

```
ghc --make -o milane MilanE.lhs
```

Then place a list of instructions (in Haskell format, ie with square brackets and commas, but spacing is ignored) in a file `foo.out` and type

```
./milane foo
```

9.6 Binary operators (optional)

7. In `Value.lhs`, add a datatype `Op2` of binary operators:

   ```
   > data Op2
   >   = Add
   >   | Sub
   >   deriving (Eq, Show, Read)
   ```

   and a function duo for applying these operators:

   ```
   > duo :: Op2 -> Value -> Value -> Value
   > duo Add (Numeric m) (Numeric n) = Numeric (m + n)
   > duo Sub (Numeric m) (Numeric n) = Numeric (m - n)
   > duo _   _           _           = Wrong
   ```

   Export both from the module:

   ```
   > module Value(Value(..), Op2(..), duo) where ...
   ```

   Add to the type of instructions a clause for binary operators:

   ```
   > data Instruction
   >   = ...
   >   | Apply2 Op2
   >   deriving (Eq, Show, Read)
   ```

How does the definition of `step` need to be extended to complete the execution of binary operators? (Be careful with the order of arguments.)

8. Test your binary operators on the program swap2, also defined in `MilanT1.lhs`:

```
> swap2 = [Push Wrong,Push Wrong,Push (Numeric 3),Store 0,
>          Push (Numeric 4),Store 1,Fetch 0,Fetch 2,Apply2 Add,
>          Store 1,Fetch 1,Fetch 1,Apply2 Sub,Store 0,Fetch 1,Fetch 1,
>          Apply2 Sub,Store 1,Fetch 0,Display,Fetch 1,Display,Halt]
```

This program has the same effect, but uses only two variables. Do you see how it works?

9. What is involved in adding multiplication, and `div` and `mod`?

9.7 Booleans (optional)

10. Add to the datatype `Value` a variant for booleans:

```
> data Value
>   = ...
>   | Logical Bool
>   deriving (Eq, Show, Read)
```

Add to `Op2` binary operators `Less` and `Eq`, and extend duo so that `Less` takes a pair of numbers and returning a boolean, and `Eq` takes either a pair of numbers or a pair of booleans and returning a boolean.

11. Add other boolean operators: `And`, `Or`, `LessEq`, etc.

9.8 Jumps (optional)

12. Add an `Instruction` called `Jump`, which takes an integer parameter:

```
> data Instruction
>   = ...
>   | Jump Int
>   deriving (Eq, Show, Read)
```

and extend `step` accordingly. The intention here is that `Jump n` should have the effect of advancing the program counter by an extra `n` (in addition to the usual 1); so `Jump 0` is a no-op, and `Jump (-1)` puts the program in an infinite loop.

13. In order to observe infinite loops in the behaviour of the program, it is convenient if the trace is productive even when the program isn't. In `Behaviour.lhs`, extend the type `Event` with a value `Tick`:

```
> data Event a
>   = ...
>   | Tick
>   deriving (Eq, Show)
```

    Modify `step` so that every time it makes a backwards jump, it generates another `Tick` in the behaviour. Test it by executing a program that gets stuck in an infinite loop.

14. In the same way, provide an instruction `JumpUnless`, also taking an `Int` parameter, which consumes a boolean from the top of the stack and only jumps if it is `False`.

9.9 Unary operators (optional)

15. Following the pattern for binary operators, provide also unary operators `Not` (on booleans) and `Minus` (on numbers).

16. Test your jumps and negation by executing the following program, for computing the greatest common divisor of two positive integers:

```
> gcdp = [Push Wrong,Push Wrong,Push (Numeric 148),Store 0,
>         Push (Numeric 58),Store 1,Fetch 0,Fetch 2,Apply2 Eq,
>         Apply1 Not,JumpUnless 14,Fetch 0,Fetch 2,Apply2 Less,
>         JumpUnless 5,Fetch 1,Fetch 1,Apply2 Sub,Store 1,Jump 4,
>         Fetch 0,Fetch 2,Apply2 Sub,Store 0,Jump (-19),Fetch 0,
>         Display,Halt]
```

    It is the result of compiling the following MILan source:

```
x := 148; y := 58;
while ~(x=y) do
```

```
  if x < y then y := y - x
  else x := x - y
  fi
od;
print x
```

10 MILan, Part 2 (optional): Compilation

In Part 2 of the series, we will develop a compiler for an *abstract syntax tree* representation of MILan source programs, as an application of recursive datatypes. (Later we'll develop a parser.) In `MilanC.lhs` you'll find a command-line version of the compiler, just as with the executor.

10.1 Syntax

In `Syntax.lhs`, you'll find a definition of the syntax of the language. In the simplest version, expressions are either constants or variable references:

```
> type Name = String
> data Expr
>   = Val Value
>   | Var Name
>   deriving (Eq, Show)
```

Programs are represented by the recursive datatype `Command`:

```
> data Command
>   = Skip
>   | Name := Expr
>   | Print Expr
>   | Command :-> Command
>   deriving (Eq, Show)
```

The `:->` constructor represents sequential composition.

The language has no types; or rather, variables are untyped, and expression evaluation is dynamically typed.

10.2 Memory model

Note that there are no declarations; all variables are 'implicitly declared' simply by being mentioned. So the first thing the compiler will have to do is to work out which variables are mentioned in a program. In `StackMap.lhs` you will find the skeletons of two functions, `expVars` and `comVars`, for determining the variables mentioned in an expression and a command respectively.

17. Complete the definition of `comVars`. (You may find the standard prelude function `List.union` useful.)

The run-time memory model keeps all values, both those in variables and the temporary ones generated during expression evaluation, on a stack. This is both simple (rather than having to look in separate places for the two different kinds of value) and realistic (block-structured languages allocate a *stack frame* for each block, facilitating recursive procedure calls).

This means that the compiler needs to keep track at all times how many temporary values are on the top of the stack, as well as which variable is stored in which location at the bottom of the stack, in order to resolve variable references. The type `StackMap` stores the requisite information:

```
> type StackMap = (Int,[Name])
```

For example, the `StackMap`

```
  (3, ["x", "y"])
```

indicates that there are five values on the stack: from top to bottom, three temporary values, the contents of the variable `x`, and the contents of the variable `y`.

18. Complete the definition of `initialStack`, representing the initial state of the stack map for a particular `Command`.

19. Complete the definitions of `push` and `pop`, which update a `StackMap` to reflect the consequences of pushing or popping the stack.

20. Define the function `depth`, which determines the depth of the stack given the stack map.

21. Define the function `location`, which determines how far down the stack a particular variable is stored. (Hint: you may find the standard prelude function `takeWhile` helpful.)

10.3 Compiling expressions

In `Compiler.lhs`, you'll find the compiler itself. In outline, the compiled code for a `Command` consists of some initialization (putting the value `Wrong` in every variable), the code per se (defined in terms of the stack map), and some finalization (appending a `Halt` instruction).

```
> compile :: Command -> [Instruction]
> compile c =
>   replicate (depth sm) (Push Wrong) ++
>   compObey sm c ++
>   [Halt]
>     where
>       sm = initialStack c
```

The function `compEval` generates the code for evaluating an expression.

```
> compEval :: StackMap -> Expr -> [Instruction]
> compEval sm (Val v) = [Push v]
...
```

This has the eventual effect of pushing onto the stack the value of that expression. For example, evaluating a constant is simply a matter of pushing that constant onto the stack.

22. Define the evaluation of a variable reference.

10.4 Compiling commands

The compilation of commands is similarly defined by induction over the structure of the command. For example, a `Skip` command compiles to no code:

```
> compObey :: StackMap -> Command -> [Instruction]
> compObey sm Skip = []
```

whereas the compilation of an assignment consists of evaluating the expression and storing it in the right place:

```
> compObey sm (v := e) = compEval sm e ++ [Store loc]
>   where loc = ...
```

23. What is the right location to store the assigned value?

24. Define the compilation of a `Print` command.

25. What code should be generated for the concatenation of two `Commands`?

26. Compile the following command:

```
> swap1 :: Command
> swap1 = ("x" := Val (Numeric 3)) :->
>         ("y" := Val (Numeric 4)) :->
>         ("z" := Var "x") :->
>         ("x" := Var "y") :->
>         ("y" := Var "z") :->
>         Print (Var "x") :->
>         Print (Var "y")
```

It should give you the code in Practical **??**.

10.5 Operators (optional)

This section depends on `Instructions` for binary and unary operators, as defined in Practical **??**. If you didn't get that far, skip this (or go back and redo that).

27. Extend compilation to generate instructions for binary and unary operators. Add appropriate `Uno` and `Duo` clauses to `Expr` in `Syntax.lhs`, extend `expVars` in `StackMap.lhs`, and extend `compEval` in `Compiler.lhs`. (Be careful with the stack map!)

28. Compile the following `Command`:

```
> swap2 :: Command
> swap2 = ("x" := Val (Numeric 3)) :->
>         ("y" := Val (Numeric 4)) :->
>         ("y" := Duo Add (Var "x") (Var "y")) :->
>         ("x" := Duo Sub (Var "y") (Var "x")) :->
>         ("y" := Duo Sub (Var "y") (Var "x")) :->
>         Print (Var "x") :->
>         Print (Var "y")
```

10.6 Flow of control (optional)

This section depends on the two jump instructions introduced in Practical **??**.

29. Add to `Command` a clause for if-then-else commands. Extend `comVars` accordingly. Provide the appropriate definition in `compObey`. (Hint: calculate the code for each of the branches in a `where` clause.)

30. Add to Command a clause for while commands. Extend comVars accordingly. Provide the appropriate definition in compObey. (Hint: again, calculate the code for the loop guard and body in a where clause.)

31. Compile the following gcd program:

```
> gcdc :: Command
> gcdc = ("x" := (Val (Numeric 148))) :->
>        ("y" := (Val (Numeric 58))) :->
>        (While
>          (Uno Not (Duo Eq (Var "x") (Var "y")))
>          (If (Duo Less (Var "x") (Var "y"))
>              ("y" := (Duo Sub (Var "y") (Var "x")))
>              ("x" := (Duo Sub (Var "x") (Var "y"))))) :->
>        (Print (Var "x"))
```

Again, it should yield the code given earlier.

11 MILan, Part 3 (optional): Interpreting

Another way of running a MILan program, represented as an element of the recursive datatype `Command`, is to interpret it directly rather than first compiling it to machine code. In `Interpreter.lhs`, you'll find the outline of such an interpreter, and in `MilanI.lhs` you'll find a command-line version of it, just as with the executor and compiler.

11.1 Environments

In interpreting a program, we will need to keep track of the values of its variables. The standard way to do this is with an *environment*, a mapping from names to values. We represent this simply as a list of pairs:

```
> type Env = [(Name,Value)]
```

32. Define a function `look`, which looks up the value of a particular variable in the environment.

    ```
    > look :: Env -> Name -> Value
    ```

    You should return `Wrong` as the value if the variable is not bound in the environment.

33. Define a function `update`, which takes a variable name and a value and updates an environment to record the new value of that variable.

    ```
    > update :: Name -> Value -> Env -> Env
    ```

11.2 Evaluating expressions

34. Define a function `eval`, to evaluate an expression in an environment. (The expression may have variables in; the environment specifies values for those variables.)

    ```
    > eval :: Expr -> Env -> Value
    ```

35. Extend `eval` to handle unary and binary operators, as defined in earlier practicals.

11.3 Obeying commands

We'll define a function run, which runs a Command in a given environment, yielding a behaviour and a new environment.

```
> run :: Command -> Env -> (Trace Value, Env)
```

We need to yield the new environment for when running one Command after another: the second one should run in the environment resulting from the first.

Then obeying a Command is straightforward: we start with the empty environment, and discard the final envionment.

```
> obey :: Command -> Trace Value
> obey p = fst (run p [])
```

You should find that run is defined for the Skip and assignment commands.

36. Define run on the Print command.

37. Define run on sequential composition; take care to chain together environments and behaviours correctly.

38. If you extended the Command datatype with conditional and looping constructs in Section **??**, extent your run function accordingly.

11.4 Enforcing invariants on traces

As another application of recursive datatypes, we can (finally) enforce the invariants we proposed for the Trace datatype, namely that every Trace should end with End or Crash, and that these two constructors should not appear elsewhere in a trace. We essentially define our own type of lists, but with these two terminators instead of the empty list, and Output and eventually Tick as two ways of adding an element to a list.

```
> data Trace a
>   = Tick (Trace a)
>   | Output a (Trace a)
>   | End
>   | Crash
>   deriving (Eq, Show)
```

39. Redefine +++ for this new type of Traces.

40. Redefine the functions that generate behaviours, to work with this new type.

12 MILan, Part 4 (optional): Parsing

This fourth and last practical on MILan is about constructing a parser for the source language. Programming a parser from scratch is nearly always the wrong thing to do these days. One reasonable technique for constructing parsers is to use a *domain-specific language*, like that implemented by `lex` and `yacc`. These are in effect compilers or interpreters themselves, for a special-purpose programming language (and so they each need their own parser!)

This kind of domain-specific language is implemented in a(nother) language; you don't get to 'see' the host language from within the hosted language. An alternative is to use a domain-specific *embedded language*, which is essentially a just library for use with the host language. The advantage (and sometimes also the disadvantage) of this approach is that the host language is not hidden: as well as the features of the hosted language, you have all the power of the host language also at your disposal. DSELs are an attractive approach in functional languages, where they can be implemented as *combinator libraries*: parametric polymorphism and higher-order functions make this surprisingly powerful. We will use a parser combinator library to write the MILan parser.

12.1 Parser combinators

You might expect that a parser recognizing values of type `a` will take a string and return an `a`; for example, and integer parser would have type `String -> Int`. But in order to chain together parsers, it is more convenient that a parser also returns the unparsed remainder of the input, so it yields a pair. Moreover, some parsers may succeed in multiple ways, and others may fail altogether on some input, so it is also convenient to return a list of results. We therefore define

```
> type Parser a = String -> [(a,String)]
```

In order to get the type we really want, we define another function to post-process this result, choosing the first result that consumes the whole input:

```
> the :: [(a,String)] -> a
> the ((x,""):_) = x
> the (_:rest)   = the rest
```

One simple parser always succeeds, consuming none of the input and returning a given fixed value:

```
> succeed :: a -> Parser a
> succeed v inp = [(v,inp)]
```

Another takes a predicate on characters; if the input is non-empty and the first character satisfies the predicate, that character is returned, and otherwise the parser fails.

```
> satisfy :: (Char -> Bool) -> Parser Char
> satisfy p []     = []
> satisfy p (x:xs) = if p x then [(x,xs)] else []
```

41. Define a function `lit`, which takes a character and returns the parser that recognizes only that character.

    ```
    > lit :: Char -> Parser Char
    ```

Those are the only two primitive parsers we will need; the other combinators put parsers together to make bigger parsers. One such combinator takes two parsers, and returns all the possible results of the first and all those of the second:

```
> (|||) :: Parser a -> Parser a -> Parser a
> (p1 ||| p2) inp = p1 inp ++ p2 inp
```

You might think of this as a choice between two parsers. A second combinator chains two parsers — for each result of the first, it runs the second parser on the *remaining* input, and returns pairs of results, one from each parser:

```
> (...) :: Parser a -> Parser b -> Parser (a,b)
> (...) p1 p2 inp = [ ((v1,v2),inp2) | (v1,inp1) <- p1 inp,
>                                      (v2,inp2) <- p2 inp1 ]
```

A third combinator makes a parser optional: it always succeeds with exactly one result, being the first result of the argument parser, if that was successful, and a given value otherwise.

```
> opt :: Parser a -> a -> Parser a
> opt p v inp = [head ((p ||| succeed v) inp)]
```

A fourth combinator takes a function and a parser, and applies the function to every result returned by the parser:

```
> using :: Parser a -> (a->b) -> Parser b
> using p f inp = [ (f v, out) | (v,out) <- p inp ]
```

42. Using `using`, define two variants of `...` that each discard one half of the result pairs.

```
> (..*) :: Parser a -> Parser b -> Parser a
> (*..) :: Parser a -> Parser b -> Parser b
```

(These are convenient when one of the two parsers is constant, perhaps recognizing punctuation; you won't be interested in the punctuation it returns, because it will always be the same thing.)

43. Using these two combinators, write the parser that recognizes simple character constants, of the form `'a'`.

Given the ingredients we have so far, we can define two combinators that repeat a given parser. The parser `many p` recognizes zero or more instances of the values recognized by `p`:

```
> many :: Parser a -> Parser [a]
> many p = ((p ... many p) `using` cons) `opt` []
>   where cons (x,xs) = x:xs
```

44. Define a combinator `some`, of the same type as `many` but recognizing *one or more* instances.

45. Define the parser `integer :: Parser Int` that parses an integer constant.

12.2 Parsing MILan

The beautiful thing about embedding a domain-specific language within a host language is that you have all the power of the host language with which to make new abstractions. For example, here is a combinator that recognizes a sequence of one or more as, separated by bs:

```
> sepseq :: Parser a -> Parser b -> ((a,[(b,a)])->c) -> Parser c
> sepseq p1 p2 f = (p1 ... many (p2 ... p1)) `using` f
```

46. Define the parser that recognizes a comma-separated sequence of integers and returns the sum of the sequence.

A MILan command consists basically of a sequence of non-sequential commands separated by semicolons:

```
> parse :: String -> Command
> parse s = the (command s)

> command :: Parser Command
> command = sepseq nonSeqCommand
>                  (lit ';' ... white)
>                  (\ (c,wcs) -> foldr1 (:->) (c : map snd wcs))
```

47. Define the parser white, which recognizes a string of whitespace characters.

    ```
    > white :: Parser String
    ```

    (Hint: there is a predicate isSpace in the standard prelude.)

48. Define the parser key, which recognizes a fixed given string followed by arbitrary whitespace (and throws both away, returning the unit value ()):

    ```
    > key :: String -> Parser ()
    ```

A non-sequential command is one of the other constructs:

```
> nonSeqCommand :: Parser Command
> nonSeqCommand =
>   key "skip" `using` const Skip |||
>   name ..* key ":=" ... expr `using` uncurry (:=) |||
>   key "print" *.. expr `using` Print |||
```

49. Extend nonSeqCommand to recognize if and while statements, of the form found in the GCD program from Practical **??**.

The parser expr recognizes constants and variable references as expressions:

```
> expr :: Parser Expr
> expr = nonBinExpr

> nonBinExpr :: Parser Expr
> nonBinExpr =
>   name `using` Var |||
>   value `using` Val
```

50. Define the parser `name`, which recognizes a program identifier (a sequence of one or more lowercase characters) followed by whitespace.

```
> name :: Parser Name
```

51. Define the parser `value`, which recognizes a number followed by whitespace.

```
> value :: Parser Value
```

52. Extend `value` to recognize boolean constants too.

53. Given the following parser for binary operators:

```
> op2 :: Parser Op2
> op2 = key "+"  *.. succeed Add |||
>       key "-"  *.. succeed Sub
```

extend `expr` so that it recognizes sequences of non-binary expressions separated by binary operators, and chains them together to yield the corresponding `Expr`.

54. What is the right way to recognize parenthesized expressions?

55. Define a parser for unary operators, and extend `expr` to allow them in expressions.

A Modules

Haskell has a relatively simple module system which allows programmers to create and import modules, where a *module* is simply a collection of related types and functions.

A.1 Declaring modules

Most projects begin with something like the following as the first line of code:

> **module** *Main* **where**

This declares that the current file defines functions to be held in the *Main* module. Apart from the *Main* module, it is recommended that you name your file to match the module name. So, for example, suppose you were defining a number of protocols to handle various mailing protocols, such as POP3 or IMAP. It would be sensible to hold these in separate modules, perhaps named *Network.Mail.POP3* and *Network.Mail.IMAP*, which would be held in separate files. Thus, the POP3 module would have the following line near the top of its source file.

> **module** *Network.Mail.POP3* **where**

This module would normally be held in a file named

`/src/Network/Mail/POP3.hs`.

Note that while modules may form a hierarchy, this is a relatively loose notion, and imposes nothing on the structure of your code.

By default, all of the types and functions defined in a module are exported. However, you might want certain types or functions to remain private to the module itself, and remain inaccessible to the outside world. To achieve this, the module system allows you to explicitly declare which functions are to be exported: everything else remains private. So, for example, if you had defined the type *POP3* and functions $send :: POP3 \to IO\ ()$ and $receive :: IO\ POP3$ within your module, then these could be exported explicitly by listing them in the module declaration:

> **module** *Network.Mail.POP3* (*POP3* (..), *send*, *receive*)

Note that for the type *POP3* we have written *POP3* (..). This declares that not only do we want to export the *type* called *POP3*, but we also want to export all of its constructors too.

A.2 Importing modules

The *Prelude* is a module which is always implicitly imported, since it contains the definitions of all kinds of useful functions such as $map :: (a \to b) \to f\ a \to f\ b$. Thus, all of its functions are in scope by default. To use the types and functions found in other modules, they must be imported explicitly. One useful module is the *Data.Maybe* module, which contains useful utility functions:

$$\begin{aligned}
&maybe &&:: b \to (a \to b) \to Maybe\ a \to b \\
&catMaybes &&:: [\,Maybe\ a\,] \to [\,a\,]
\end{aligned}$$

Importing all of the functions from *Data.Maybe* into a particular module is done by adding the following line below the module declaration, which imports every module exported by *Data.Maybe*

import *Data.Maybe*

It is generally accepted as good style to restrict the imports to only those you intend to use: this makes it easier for others to understand where some of the unusual definitions you might be importing come from. To do this, simply list the imports explicitly, and only those types and functions will be imported:

import *Data.Maybe* (*maybe*, *catMaybes*)

This imports *maybe* and *catMaybes* in addition to any other imports expressed in other lines.

A.3 Qualifying and hiding imports

Sometimes, importing modules might result in conflicts with functions that have already been defined. For example, one useful module is *Data.Map*. The base datatype that is provided is *Map* which efficiently stores values indexed by some key. There are a number of other useful functions defined in this module:

$$\begin{aligned}
&empty &&:: Map\ k\ v \\
&insert &&:: (Ord\ k) \Rightarrow k \to v \to Map\ k\ v \to Map\ k\ v \\
&update &&:: (Ord\ k) \Rightarrow k \to Map\ k\ v \to Maybe\ v
\end{aligned}$$

It might be tempting to import *Map* and these auxiliary functions as follows:

import *Data.Map* (*Map* (..), *empty*, *insert*, *lookup*)

However, there is a catch here! The *lookup* function is initially always implicitly in scope, since the *Prelude* defines its own version. There are a number of ways to resolve this. Perhaps the most common solution is to qualify the import, which means that the use of imports from *Data.Map* must be prefixed by the module name. Thus, we would write the following instead as the import statement:

import *qualified Data.Map*

To actually use the functions and types from *Data.Map*, this prefix would have to be written explicitly. For example, to use *lookup*, we would actually have to write *Data.Map.lookup* instead.

These long names can become somewhat tedious to use, and so the qualified import is usually given as something different:

import *qualified Data.Map as M*

This brings all of the functionality of *Data.Map* to be used by prefixing with *M* rather than *Data.Map*, thus allowing you to use *M.lookup* instead.

Another solution to module clashes is to hide the functions that are already in scope within the module by using the *hiding* keyword:

import *Prelude hiding* (*lookup*)

This will override the *Prelude* import so that the definition of *lookup* is excluded.