CS_221 Functional Programming I

Coursework 2

Due date: Thursday, 6 December 2007
Assessment: Only the questions 1, 2, 3, 5, 6, 9, 10 are assessed
Submission: Please follow the instructions given on page 6

In this coursework we develop a program for the graphical display of 2-dimensional regions.

We represent a region as a *bitmap*, that is, a function $f : \mathbb{R} \times \mathbb{R} \to \mathbb{B}$ where $\mathbb{R}$ is the set of real numbers and $\mathbb{B} = \{T, F\}$ is the set of Booleans: $f(x, y) = T$ means the point $(x, y)$ is in the region represented by $f$, $f(x, y) = F$ means it is outside. Therefore, $f$ represents the region $\{(x, y) \mid f(x, y) = T\}$.

```haskell
type Pt = (Float, Float)
type Bitmap = Pt -> Bool

For example, the function $f(x, y) := x^2 + y^2 \leq 1$, that is, $f(x, y) = T$ if $x^2 + y^2 \leq 1$ and $f(x, y) = F$ otherwise, represents the region $\{(x, y) \mid x^2 + y^2 \leq 1\}$, that is, the set of points that lie on a disc with radius 1 around the origin, $(0, 0)$.

discBm :: Bitmap
discBm (x,y) = x^2 + y^2 <= 1

A *picture* is a list of lists of Booleans representing a region restricted to a rectangular segment of the plane.

newtype Picture = Pic [[Bool]]

Example:

discPic :: Picture
discPic =
    Pic [[False,False,False,False,False,True, False,False,False,False,False],
         [False,False,True, True, True, True, True, True, True, True, False],
         [False,True, True, True, True, True, True, True, True, True, True],
         [False,True, True, True, True, True, True, True, True, True, False],
         [True, True, True, True, True, True, True, True, True, True, True],
         [False,True, True, True, True, True, True, True, True, True, False],
         [False,True, True, True, True, True, True, True, True, True, False],
         [False,False,False,False,False,True, False,False,False,False,False]]
```
The picture \texttt{discPic} has been obtained by sampling the bitmap \texttt{discBm} at $11 \times 9$ equidistant points in the square \{$(x, y) \mid -1 \leq x, y \leq 1$\} $\subseteq \mathbb{R} \times \mathbb{R}$. More precisely, the sampling points are of the form $(-1 + 0.2j, 1 - 0.25i)$ where $j \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$.

In general, the following additional data are required to compute a picture from a bitmap:

- The \textit{frame}: A rectangle, given by its bottom left and top right corner (in our example $(-1, -1)$ and $(1, 1)$).
- The \textit{resolution}: Two positive integers, $n, m$, determining the number of sampling points inside the rectangle (in our example 10 and 8).

We call these data (the frame together with the resolution) a \textit{setting}:

\[
\text{type Setting} = (\text{Pt}, \text{Pt}, \text{Int}, \text{Int})
\]

\textbf{Question 1.} Define a function \texttt{takePicture :: Setting \to Bitmap \to Picture} that computes a picture from a given setting and a bitmap. \hfill \textbf{[20 marks]}

A simple way of displaying pictures is to transform them into strings as follows: A picture

\[
\text{Pic } \left[ \text{bs1, bs2, ..., bsm} \right]
\]

is transformed into the string

\[
(\backslash\text{n}\text{:s1}) \,++\, (\backslash\text{n}\text{:s2}) \,...\,++\,(\backslash\text{n}\text{:sm})
\]

where \texttt{\backslash\text{n}} is the \textit{newline character} and each string \texttt{s_i} is obtained from the Boolean list \texttt{bs_i} by replacing \texttt{True} with '@' and \texttt{False} with '.'. For example, the picture

\[
\text{Pic } \left[ \left[ \text{True, True, True} \right], \left[ \text{True, True, False} \right], \left[ \text{True, False, False} \right] \right]
\]

is transformed into the string "\text{n@@\text{n@@.n@"}.

\textbf{Question 2.} Define a function \texttt{showPicture :: Picture \to String} that computes a string from a picture as described above. \hfill \textbf{[10 marks]}

Using the function \texttt{showPicture} you can put the type \texttt{Picture} into the type class \texttt{Show}:

\[
\text{instance Show Picture where}
\text{ show } = \text{showPicture}
\]

This has the effect that when evaluating an expression of type \texttt{Picture}, Haskell will display the result in the expected format.

If Questions 1., 2. and have been solved correctly, evaluation of the expressions \texttt{discPic} and \texttt{takePicture (((-1,-1),(1,1),10,8) discBm)} should yield the same result, which should be displayed by Haskell as follows:
Question 3. Using the setting

\[ \text{set0 :: Setting} \]
\[ \text{set0} = ((-1.5, -1.5), (1.5, 1.5), 80, 40) \]

take pictures of the following regions:

(a) \( \{(x, y) \mid y \leq x\} \).
(b) \( \{(x, y) \mid x \cdot y > 0\} \).
(c) \( \{(x, y) \mid x \cdot y > 0.25\} \).
(d) A square with vertices \((-1, -1), (1, -1), (1, 1), (-1, 1)\).

Remark: Use a small font for displaying the results. Depending on the monitor you are working with you might wish to change the numbers 80 and 40 to a different resolution.

[10 marks]

Question 4. Using the same setting as in question 3, take pictures of the following regions:

(a) \( \{(x, y) \mid y \leq 0.5 \cdot \sin(2 \pi x)\} \).
(b) \( \{(x, y) \mid y \leq x^{k+2} - x^k\} \) for varying integers \( k \); try out \( k = 0, 1, 2 \).
(c) \( \{(x, y) \mid |x|^k + |y|^k \leq 1\} \) for varying integers \( k \); try out \( k = 1, 2, \ldots, 20 \).

How are the regions in questions 3 (d) and 4 (g) related? Can you think of other interesting regions to take pictures off?

Question 5. Define the following operations on pictures:

(a) \text{appvPic}: appending two pictures vertically (second below first).
(b) \text{apphPic}: appending two pictures horizontally (side by side).
(c) \text{udPic}: flipping a picture upside down.
(d) \text{lrPic}: flipping left and right of a picture.
(e) \text{invPic}: inverting a picture (complement, that is, pointwise negation).

(f) \text{unionPic}: Combining two pictures disjunctively (union, pointwise (\mid\mid)).

(g) \text{interPic}: Combining two pictures conjunctively (intersection, pointwise (&&)).

(h) \text{exclPic}: Combining two pictures exclusively (pointwise (\neq)).

In questions (a), (b), (f), (g) and (h) it may be assumed that the given pictures are of the same size (but this is not relevant for the implementation).

Hint: All these operations can be defined in one line, possibly using map and zipWith.

Test the operations with the examples from question 3 (using a different setting if necessary).

\textbf{Question 6.} Recall the following definitions:

\[ (++): \text{[a]} \rightarrow \text{[a]} \rightarrow \text{[a]} \]
\[ [] ++ ys = ys \]
\[ (x:xs) ++ ys = x : (xs ++ ys) \]

\[ \text{map: (a -> b)} \rightarrow \text{[a]} \rightarrow \text{[b]} \]
\[ \text{map f} [\text{[]} = []; \]
\[ \text{map f} (x:xs) = f x : \text{map f} xs \]

\[ \text{reverse: [a]} \rightarrow \text{[a]} \]
\[ \text{reverse} [\text{[]} = []; \]
\[ \text{reverse} (x : xs) = \text{reverse xs} ++ [x] \]

(a) Prove by induction on \(xs :: \text{[a]}\)

\[ \text{map f} (xs ++ ys) = \text{map f} xs ++ \text{map f} ys \]

(b) Using part (a), prove by induction on \(xss :: \text{[[a]]}\)

\[ \text{map reverse} (\text{reverse xss}) = \text{reverse} (\text{map reverse} xss) \]

(that is, \(\text{map reverse} \cdot \text{reverse} = \text{reverse} \cdot \text{map reverse}\)).

(c) Use part (b) to show that the operations \text{udPic} and \text{lrPic} from Question 4 (c),(d) commute, that is, \(\text{lrPic} \cdot \text{udPic} = \text{udPic} \cdot \text{lrPic}\).
**Question 7.** The logistic map with parameter \( r \in [0, 4] \)

\[
\varphi_r : [0, 1] \to [0, 1], \quad \varphi_r(x) = r \cdot x \cdot (1 - x)
\]

([\(a, b]\) = \{\(x \in \mathbb{R} \mid a \leq x \leq b\}\}) is a simple example of a dynamical system with chaotic behaviour. It is used as a demographic model and as a testing function for implementations of real number operations. For more information on the logistic map see, for example, \[\text{http://en.wikipedia.org/wiki/Logistic\_map}\]

The chaotic behaviour of the logistic map is due to the fact that for many parameter values \( r \) close to 4 and many starting points \( x \in [0, 1] \) the orbit of the logistic map, \[
\{\varphi^n_r(x) \mid n = 1, 2, 3, \ldots \} = \{\varphi_r(x), \varphi_r(\varphi_r(x)), \varphi_r(\varphi_r(\varphi_r(x))), \ldots \}
\]
is a highly irregular set the elements of which (for large \( n \)) are difficult to compute because of the accumulation of rounding errors.

In the following we only consider the parameter \( r = 4 \) and set \( \varphi := \varphi_4 \).

Take pictures of the iterations, \( \varphi^n : [0, 1] \to [0, 1] \), of the logistic map for \( n = 1, \ldots, 10 \), that is, take pictures of the regions \[
\{(x, y) \mid x, y \in [0, 1], y < \varphi^n(x)\} \quad (n = 1, \ldots, 10)
\]

**Question 8.** The Mandelbrot set is the set of all complex numbers \( c \in \mathbb{C} \) with the property that the mapping \( f_c : \mathbb{C} \to \mathbb{C}, \quad f_c(z) = z^2 + c \)
has a bounded orbit at 0, that is, the set \[
\{|f^n_c(0)| \mid n = 1, 2, 3, \ldots \} = \{|f_c(0)|, |f_c(f_c(0))|, |f_c(f_c(f_c(0)))|, \ldots \} = \{|c|, |c^2+c|, |(c^2+c)^2+c|, \ldots \}
\]
is bounded. The Mandelbrot set is a fractal, that is, a set with an infinitely complex microstructure. For more information, see \[\text{http://en.wikipedia.org/wiki/Mandelbrot\_set}\]

By identifying a complex number \( c = x + i \cdot y \in \mathbb{C} \) with the point \((x, y)\) we may view the Mandelbrot set as a region in the 2-dimensional plane.

Using the setting \((-2, -1.3), (1, 1.3), 160, 80\) (maybe with a different resolution) take a picture of the following region that approximates the Mandelbrot set:

\[
\{c \in \mathbb{C} \mid |f^n_c(0)| < 2 \text{ for all } n \in \{0, 1, 2, \ldots, 20\}\}
\]

**Reminder on how to calculate with complex numbers:**

\[
(x + y \cdot i) + (x' + y' \cdot i) = (x + x') + (y + y') \cdot i
\]
\[
(x + y \cdot i) \cdot (x' + y' \cdot i) = (x \cdot x' - y \cdot y') + (x \cdot y' + y \cdot x') \cdot i
\]
\[
|x + y \cdot i| = \sqrt{x^2 + y^2}
\]

Hence \(|x + y \cdot i| < 2\) if and only if \(x^2 + y^2 < 4\).
Recall the definitions of the data types Shape and Region:

```haskell
type Radius = Float
type Angle = Float

data Shape = Ellipse Pt Radius Radius
            | Polygon [Pt]
            deriving Show

data Region = Sh Shape
              | Rotate Angle Region
              | Inter Region Region
              | Union Region Region
              | Compl Region
              | Empty
              deriving Show
```

**Question 9.** Define a function `inRegion :: Region -> Bitmap` that transforms a region into the corresponding bitmap.

Hint: use the function `inShape :: Shape -> Bitmap`, that transforms a shape into the corresponding bitmap. The code for this function is included in the template file `fp1-cw2-07-template.hs` which is available from the course web page. You may find it also useful to first define operations on bitmaps that correspond to the constructors `Rotate`, `Union`, `Inter`, and `Compl`. [10 marks]

**Question 10.** Define a function `sunFlower :: Int -> Radius -> Region` such that the region `sunFlower l r` looks like a sun flower with `l` petals and radius `r`.

Take a picture of a sunflower. [15 marks]

**Question 11.** The Sierpinski triangle is a fractal consisting of smaller and smaller rectangular triangles. It is defined recursively as follows: Let a size `d` (a floating point number) and a point `p = (x, y)` be given. The **Sierpinski triangle of size `d` at point `p`** consists of the rectangular triangle with vertices `p_1`, `p_2`, `p_3` together with the three Sierpinski triangles of size `d'` at the points `p`, `p_1`, `p_2`, where `d' = d/2`, `p_1 = (x + d', y)`, `p_2 = (x, y + d')` and `p_3 = (x + d', y + d')`.

Slightly different (and more general) forms of the Sierpinski triangle are described at [http://en.wikipedia.org/wiki/Sierpinski_triangle](http://en.wikipedia.org/wiki/Sierpinski_triangle)

Define a function `sierpinski :: Float -> Float -> Pt -> Region` such that the region `sierpinski s d p` is an approximation of the Sierpinski triangle of size `d` at point `p` where the recursion stops when `d < s`. Take a picture of a Sierpinski triangle.

**Instructions for submission:**

1. Write your solutions in the template file `fp1-cw2-07-template.hs`, **without including test results**.
2. Submit a print out of your coursework in the box on the second floor.
3. Demonstrate that your programs work in one of the lab sessions 25/26 November, 3/4/10/11 December.