Functional Programming

Exercise Sheet 3 – Datatypes, Typeclasses

2021-05-07

Exercise 1 (Vectors)

Define a data type for 2D vectors with `Double` components. Try to specify a `Num` typeclass instance for your data type.

Generalize your data type and its operations to support any kind of components. Specify the type signatures for the operations.

Note: To find the methods used in the `Num` typeclass, or to quickly search for documentation, you can use Hoogle (http://www.haskell.org/hoogle/).

Exercise 2 (Monoids)

A typeclass that is used quite often in Haskell programs is `Monoid`, modeled after the algebraic structure of the same name: the *monoid*[1]. A monoid is a set with an associative operation (`mappend` in Haskell) and a neutral element (`mzero`).

The most prominent `Monoid` instance are lists, with `mappend = (++)` and `mzero = []`.

If possible, try to define monoid instances for the data types in the previous exercises.

Exercise 3 (Vector graphics)

In a vector drawing program, images are not described by their pixels but by the arrangement of various geometric elements such as circles, Rectangles, lines, Bezier curves, ... For example, the following image is "a square with edge length 1 over a triangle with height 1 and base length 1 ". In a program, of course, this description needs to be specified.

1. Define a datatype `Picture` which can describe lines, rectangles, circles and can combine different pictures. Try to use your datatype to create various pictures. Define at least the figure shown above and the "House with roof window" shown below.

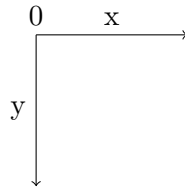   Does this datatypes forms a monoid? What should be the `mappend` operation?
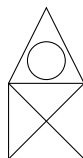
2. In order to show pictures, we will use the SVG format thanks to the svg-builder library[2]. The SVG format allows to define complex vector pictures and is supported by many viewer and web browsers. While SVG is very rich, we will only use the `path` operation. Here is a simple file that writes a line from the point $(0, 100)$ to the point $(30, 40)$.

```
{-# LANGUAGE OverloadedStrings #-}
import Graphics.Svg
myline = path_ [D_ <<- mA 0 100 <> lA 30 40]
svg = doctype <> with (svg11_ myline) [Width_ <<- "100", Height_ <<- "100"]
main = renderToFile "path/to/file" svg
```

---

[1] http://en.wikipedia.org/wiki/Monoid
[2] https://hackage.haskell.org/package/svg-builder

Beware, the origin in SVG is in the upper left corner, like so:

3. Thanks to the `Picture` datatype, we can represent pictures as values and manipulate them. For example, we can implement operations for moving and scaling pictures:

```
move: (Float, Float) -> Picture -> Picture
scale: Float -> Picture -> Picture
```

Use these operations to create a landscape by combining houses of various scales and position.
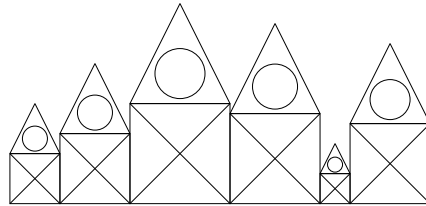
Bonus: You can also use a random number generator[3] to generate this landscape in a procedural way.

4. Add a rotation function that takes a rotation center and an angle.

```
rotate: (Float, Float) -> Float -> Picture -> Picture
```

Use it to write a recursive function `dragon: Integer -> Picture` that generates the dragon curve[4]. You should only need lines and rotations.

---

[3] https://hackage.haskell.org/package/random-1.1/docs/System-Random.html
[4] https://en.wikipedia.org/wiki/Dragon_curve