Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Datenbanksysteme · Prof. Dr. Grust

EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming

SS 2017

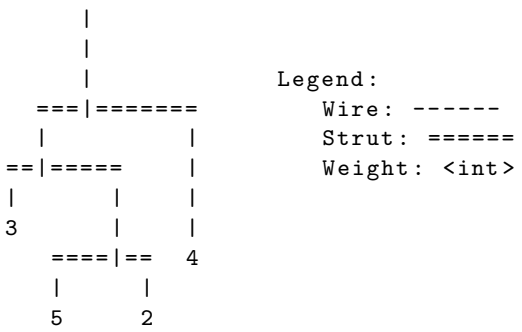
Benjamin Dietrich

Assignment #5

Submission Deadline: Thu, 1.6.2017

Exercise 1: Mobiles (No, Not Phones) (20 Points)

A mobile is a hanging ornament made of wires, struts and weights as illustrated in the following figure:



Mobiles can be modelled using the following types:

```
-- | A mobile is represented by the length of a wire and the object
-- which hangs on the wire.
data Mobile = Mobile Int Object

-- | Hanging on a wire we have either a weight or a strut which
-- supports two smaller mobiles on both ends.
data Object = Weight Int
            | Strut Branch Branch

-- | A branch is defined by its length and the mobile which hangs on
-- its end.
data Branch = Branch Int Mobile
```

These types along with some sample mobiles are provided in the file `Mobile.hs` which you can find in your repository.

Based on these types, we start with some warm-up exercises:

1. Write a function `weight :: Mobile -> Int` that computes the sum of all weights in a mobile.

2. Write a function `balance :: Mobile -> Mobile` which balances a given mobile, by moving the anchor points of its wires on their struts (the total length of struts is fixed and must not change). A mobile is balanced, if for all struts the moment of force on both sides is the same: $s_l * G_l = s_r * G_r$. Given a strut with a *left branch* of length s_l and a mobile of weight G_l hanging on it and a *right branch* with s_r and G_r, the new strut lengths s'_l and s'_r can be calculated as follows:

$$s'_l = round\left(\frac{s_l + s_r}{\frac{G_l}{G_r} + 1}\right), \quad s'_r = s_l + r_r - s'_l$$

Of course we don't want to draw mobiles—such as the one above shown above—by hand; so in the following we are going implement drawing of mobiles. Take the drawings presented above and in `Mobile.hs` as a specification these drawings. You may assume that a mobile can be rendered without collisions, *i.e.* without branches overlapping each other. Furthermore, we assume mobiles to be balanced; ignore gravitational effects and render each strut on a horizontal line, without considering the weights in the branches.

As this is the first substantial piece of Haskell code we write in an assignment, the following guideline should help you to structure the problem somewhat.

1. General strategy: Construct the drawing bottom-up by drawing the leaves (weights) and combine them into larger drawings while going up the tree.
2. Store drawings of (a part of) a mobile as a rectangular area of characters. You could represent this for example using a list of lines (strings). It might be helpful to explicitly annotate these *bounding boxes* with their widths left and right of the wire attachment. This is just a proposal, though:

```
-- | A bounding box for mobile drawings. Its horizontal origin is the
-- attachment point of the wire. Next to the drawing, its extents left
-- and right to the origin are provided.
data Box = Box Int       -- ^ The width left of the wire attachment
               Int       -- ^ The width right of the wire attachment
               [String]  -- ^ The drawing itself
```

3. Start with the simple base case: define a function to create a drawing of a `Weight` object. To keep things simple, start with the assumption that a weight is only a one-digit integer.
4. To deal with `Strut` objects, you need to combine the drawings of their sub-mobiles into one drawing. Start with a function that takes two drawings and places them next to each other, with the correct amount of whitespace in between. Once aligned in this way, the new drawing can be decorated with the strut (`===`) on top.
5. Drawing a mobile is easy: Take the drawing of the object contained in it and decorate the drawing with a wire of the given length.
6. Finally, draw the resulting `Box` by concatenating its lines with a separator string `"\n"` and print it using the function `putStrLn :: String -> IO ()`.
7. In general: Attack problems by dividing them into smaller subproblems and introduce helper functions to deal with those subproblems.