Programming I: Functional Programming in Haskell

Unassessed Exercises 5: **User-Defined Types and Typeclasses**

These exercises are unassessed so you do not need to submit them. They are designed to help you master the language, so you should do as many as you can at your own speed.

There are probably more questions on these sheets than you may need in order to get the hang of a particular concept, so feel free to skip over some of the questions. You can always go back to them later if you need to.

Model answers to each set will be handed out throughout the course.

---

1. Define a data type `Shape` suitable for representing arbitrary triangles, whose dimensions are specified by the length of the triangle's three sides (all `Float`s), squares, whose dimensions are given by a single `Float`, and circles, each specified by its radius (again a `Float`). Define a function `area :: Shape -> Float` which returns the area of a given Shape. Recall: The area of a triangle with dimensions a, b, c is given by:

   $$A = \sqrt{s(s-a)(s-b)(s-c)}$$

   where $s = \frac{a+b+c}{2}$.

2. Now augment `Shape` with (convex) polygons, specified by a list of the polygon vertices, i.e. (x, y) coordinates in the Cartesian plane. Augment the area function accordingly. Note: the area of a convex polygon is the area of the triangle formed by its first three vertices plus the area of the polygon remaining when the triangle has been removed from the polygon. The length of the line joining vertex (x, y) with vertex (x', y') is given by:

   $$L = \sqrt{(x-x')^2 + (y-y')^2}$$

3. Define a single-constructor datatype `Date` suitable for representing calendar dates. Write a function `age` which given the birth date of a person and the current date returns the person's age in years as an `Int`.

4. For the following binary tree data type:

   ```haskell
   data Tree = Leaf | Node Tree Tree deriving (Eq, Show)
   ```

   define a function `makeTrees :: Int -> [Tree]` that, given an integer n, will return the list of all binary trees with n nodes (i.e. `Node` constructors). `makeTrees 0` should thus return `[Leaf]`. Note: the number of such trees is given by the n^{th} *Catalan number*:

   $$\frac{1}{n+1}\binom{2n}{n}$$

   You can use this to test your solution. Hint: use a list comprehension.

5. Define a polymorphic data type for describing binary trees in which all values are stored at the leaves; there should be no concept of an empty tree. Using this data type:

   (a) Write a function `build` that will construct a *balanced* binary tree from a non-empty list of values by using the built-in function `splitAt`. The resulting tree should have the property that the elements in the leaves, when read left to right, are in the same order as the original list. You could issue an error message if build is applied to an empty list, or assume a precondition that all input lists are non-empty.

       Note: A balanced binary tree has the property that at each internal node the sizes of the left and right subtrees differ by at most 1.

   (b) Write a function `ends` which will convert a binary tree to a list, preserving a left-to-right ordering of the elements at the fringe of the tree. Your functions should obey the rule:

       ```
       ends (build xs) == xs
       ```

   (c) Write a function `swap` which will interchange the subtrees of a binary tree at every level. For a list `xs` how does the value of `ends (swap (build xs))` relate to `xs`?

6. Design a data type `Time` which is capable of representing time in two ways: either in 24-hour format (for example 1356) or in conventional "wall clock" hours and minutes, with an additional flag to indicate whether the time is am or pm, e.g. 1:56pm. Use an additional enumerated data type `AmPm` to represent the two possibilities. Use `deriving` to enable times to be both displayed and compared for structural equality. Notice that a comparison of the two times above will yield `False` because they are structurally different, even though they depict the same time.

   (a) Write a function `to24` which will convert a time in either format to 24-hour format. A convention is that 12:00pm is midday (1200) and 12:00am is midnight (0000).

   (b) Using `to24` write a function `equalTime` which given two times in either format returns `True` if the times are the same; `False` otherwise. For example, given the representations of 1514 and 3:14pm, `equalTime` should return `True`.

   (c) Now delete the `deriving` clause from your data type and make `Time` an instance of the class `Eq`. At the same time define `(==)` on `Time`s which delivers `True` iff two times are equal, regardless of their format. Test your implementation by comparing various `Time`s for both equality and inequality (recall that `(/=)` is defined by default in terms of `(==)` in the definition of class `Eq`).

   (d) Now make `Time` an instance of the class `Show` and provide a definition of the function `show` for displaying `Time`s. A time in 24-hour format should be displayed in the form `HH:MM`, the four-digit 24-hour time. For a time in the wall-clock format, hours (`HH`) and minutes (`MM`) should be displayed in the form `HH-MMam` for times before midday, and `HH:MMpm` for times after midday, with the special cases that 12-00pm and 12-00am should be displayed as `"Midday"` and `"Midnight"` respectively. Test your implementation by typing various expressions of type `Time` on the Haskell command line.

7. The following questions relate to a generalisation of the binary trees that we've covered in lectures.

   (a) Define a polymorphic data type `Tree a b` which stores values both at the internal nodes and the leaves, and for which there is an `Empty` data constructor for representing empty trees. Arrange it so that values at the nodes may have a different type (type `a`) to those at the leaves (type `b`).

(b) Define a function `mapTree` which maps two functions over a given `Tree`. The first function should be applied to values at the leaves and the second to values at the nodes. The functions can each change one of the two parameter types for the tree.

(c) Define the function `foldTree`, a version of `fold` for objects of type `Tree` that takes two functions, one to transform a given leaf value and one to reduce an internal node. You can picture this as a transformation on a tree which replaces the empty tree by a given base case, and the leaf and node constructors with the supplied functions. Define functions in terms of `foldTree` to do the following:

   i. Count the number of leaves in a given `Tree a b`. An empty tree should not be counted as a leaf.

   ii. Sum the values in a given `Tree Int Int`.

   iii. Perform a left-to-right *in-order* flattening of a `Tree a a` to yield a `[a]`.

   iv. Perform a right-to-left in-order flattening of a tree which delivers the same result as above, but in the reverse order. Do this by modifying your folding functions; do not use a reverse function on the result from above!

   v. Evaluate a `Tree (Int -> Int -> Int) Int` where the internal nodes represent binary functions over integers and the leaves represent integer constants. In order to test this, define a tree with `(+)` at the root, `(*)` at the root of the left subtree and `(-)` at the root of the right subtree. Define the leaves to be 2, 4, 15 and 9 respectively, reading from left to right. The answer should be 14.

   vi. Implement the `mapTree` function from before.