CIS 194: Homework 8

Due Monday, March 18

- Files you should submit: `Party.hs`, containing a module of the same name (**make sure your file actually has `module Party where` at the top!**).

Planning the office party

As the most junior employee at Calculators R Us, Inc., you are tasked with organizing the office Spring Break party. As with all party organizers, your goal is, of course, to maximize the *amount of fun*[1] which is had at the party. Since some people enjoy parties more than others, you have estimated the amount of fun which will be had by each employee. So simply summing together all these values should indicate the amount of fun which will be had at the party in total, right?

[1] As measured, of course, in Standard Transnational Fun Units, or STFUs.

. . . well, there's one small problem. It is a well-known fact that anyone whose immediate boss is *also* at the party will not have any fun at all. So if *all* the company employees are at the party, only the CEO will have fun, and everyone else will stand around laughing nervously and trying to look natural while looking for their boss out of the corner of their eyes.

Your job, then, is to figure out *who to invite* to the party in order to maximize the total amount of fun.

Preliminaries

We have provided you with the file `Employee.hs`, which contains the following definitions:

```
-- Employee names are represented by Strings.
type Name = String

-- The amount of fun an employee would have at the party,
-- represented by an Integer number of STFUs
type Fun  = Integer

-- An Employee consists of a name and a fun score.
data Employee = Emp { empName :: Name, empFun :: Fun }
  deriving (Show, Read, Eq)
```

Note that the definition of `Employee` uses *record syntax*, which you can read more about in the Week 8 lecture notes (it was not covered in lecture but is provided in the lecture notes as an additional resource).

It also defines `testCompany :: Tree Employee`, a small company hierarchy which you can use for testing your code (although your actual company hierarchy is much larger).

Finally, `Employee.hs` defines a type to represent guest lists. The obvious possibility to represent a guest list would be `[Employee]`. However, we will frequently want to know the total amount of fun had by a particular guest list, and it would be inefficient to recompute it every time by adding up the fun scores for all the employees in the list. Instead, a `GuestList` contains both a list of `Employees` and a `Fun` score. Values of type `GuestList` should always satisfy the invariant that the sum of all the `Fun` scores in the list of `Employees` should be equal to the one, "cached" `Fun` score.

Exercise 1

Now define the following tools for working with `GuestLists`:

1. A function

   ```
   glCons :: Employee -> GuestList -> GuestList
   ```

   which adds an `Employee` to the `GuestList` (updating the cached `Fun` score appropriately). Of course, in general this is impossible: the updated fun score should depend on whether the `Employee` being added is already in the list, or if any of their direct subordinates are in the list, and so on. For our purposes, though, you may assume that none of these special cases will hold: that is, `glCons` should simply add the new `Employee` and add their fun score without doing any kind of checks.

2. A `Monoid` instance for `GuestList`.[2] (How is the `Monoid` instance supposed to work, you ask? You figure it out!)
3. A function `moreFun :: GuestList -> GuestList -> GuestList` which takes two `GuestLists` and returns whichever one of them is more fun, *i.e.* has the higher fun score. (If the scores are equal it does not matter which is returned.)

[2] Note that this requires creating an "orphan instance" (a type class instance `instance C T` which is defined in a module which is distinct from both the modules where `C` and `T` are defined), which GHC will warn you about. You can ignore the warning, or add `{-# OPTIONS_GHC -fno-warn-orphans #-}` to the top of your file.

Exercise 2

The `Data.Tree` module from the standard Haskell libraries defines the type of "rose trees", where each node stores a data element and has any number of children (*i.e.* a *list* of subtrees):

```
data Tree a = Node {
        rootLabel :: a,         -- label value
        subForest :: [Tree a]   -- zero or more child trees
    }
```

Strangely, `Data.Tree` does *not* define a fold for this type! Rectify the situation by implementing

```
treeFold :: ... -> Tree a -> b
```

(See if you can figure out what type(s) should replace the dots in the type of `treeFold`. If you are stuck, look back at the lecture notes from Week 7, or infer the proper type(s) from the remainder of this assignment.)

The algorithm

Now let's actually derive an algorithm to solve this problem. Clearly there must be some sort of recursion involved—in fact, it seems that we should be able to do it with a fold. This makes sense though—starting from the bottom of the tree and working our way up, we compute the best guest list for each subtree and somehow combine these to decide on the guest list for the next level up, and so on. So we need to write a combining function

I mean, why else would we have had you do Exercise 2?

```
combineGLs :: Employee -> [GuestList] -> GuestList
```

which takes an employee (the boss of some division) and the optimal guest list for each subdivision under him, and somehow combines this information to compute the best guest list for the entire division.

However, this obvious first attempt fails! The problem is that we don't get enough information from the recursive calls. If the best guest list for some subtree involves inviting that subtree's boss, then we are stuck, since we might want to consider inviting the boss of the entire tree—in which case we don't want to invite any of the subtree bosses (since they wouldn't have any fun anyway). But we might be able to do better than just taking the best possible guest list for each subtree and then excluding their bosses.

The solution is to generalize the recursion to compute *more* information, in such a way that we can actually make the recursive step. In particular, instead of just computing the best guest list for a given tree, we will compute *two* guest lists:

1. the best possible guest list we can create *if we invite the boss* (that is, the `Employee` at the root of the tree); and
2. the best possible guest list we can create if we *don't* invite the boss.

It turns out that this gives us enough information at each step to compute the optimal two guest lists for the next level up.

Exercise 3

Write a function

```
nextLevel :: Employee -> [(GuestList, GuestList)]
             -> (GuestList, GuestList)
```

which takes two arguments. The first is the "boss" of the current subtree (let's call him Bob). The second argument is a list of the results for each subtree under Bob. Each result is a pair of `GuestLists`: the first `GuestList` in the pair is the best possible guest list *with* the boss of that subtree; the second is the best possible guest list *without* the boss of that subtree. `nextLevel` should then compute the overall best guest list that includes Bob, and the overall best guest list that doesn't include Bob.

Exercise 4

Finally, put all of this together to define

```
maxFun :: Tree Employee -> GuestList
```

which takes a company hierarchy as input and outputs a fun-maximizing guest list. You can test your function on `testCompany`, provided in `Employee.hs`.

The whole company

Of course, the *actual* tree of employees in your company is much larger! We have provided you with a file, `company.txt`, containing the entire hierarchy for your company. The contents of this file were created by calling the `show` function on a `Tree Employee`,[3] so you can convert it back into a `Tree Employee` using the `read` function.

[3] We don't recommend actually looking at the contents of `company.txt`, assuming that you value your sanity.

Exercise 5

Implement `main :: IO ()` so that it reads your company's hierarchy from the file `company.txt`, and then prints out a formatted guest list, sorted by first name, which looks like

```
Total fun: 23924
Adam Debergues
Adeline Anselme
...
```

(Note: the above is just an example of the *format*; it is *not* the correct output!) You will probably find the `readFile` and `putStrLn` functions useful.

As much as possible, try to separate out the "pure" computation from the `IO` computation. In other words, your `main` function should actually be fairly short, calling out to helper functions (whose types do not involve `IO`) to do most of the work. If you find `IO` "infecting" all your function types, you are Doing It Wrong.