CIS 194: Homework 4

Due Friday, September 26, 2014

No template file is provided for this homework. Start your `HW04.hs` Haskell file with your name, any sources you consulted, and any other relevant comments (just like in previous assignments). Then, say

```
module HW04 where
```

and off you go. Do make sure to include this module header, as it makes grading much easier for us.

Parametricity

Haskell comes with a wonderful feature called *parametric polymorphism*, which allows programmers to write datatypes and functions that operate over a range of different types. Other languages support similar features, such as C++'s templates, Java's generics, and OCaml's parametric polymorphism (which is ever-so-slightly different than Haskell's!)

In these exercises, I will be providing a type, and you have to provide an implementation—*any* implementation—that typechecks. The only rule is that your implementation must be *total*. That is, given any input, your implementations must always run in a finite time and return a value. This means that any of the following are disallowed:

- Infinite recursion
- Pattern-matching that does not contain all cases
- `undefined`
- `error`
- Any function beginning with `unsafe`.
- The use of non-total functions, such as `head`, `tail`, `init`, etc.

This list is not complete, but you will generally write total functions unless you're trying to be devious. (Or accidentally write a loop.)

After writing your implementation for each exercise, write a comment describing the nature of *all* implementations of functions with that type. In particular, if the number of functions that can inhabit the type is finite, say what it is. If it's not finite, try to come up with properties of the functions that you can read right from the type.

For example, if you have to implement

```
example1 :: a -> a
```

the *only* possible answer is

```
example1 x = x
```

We know that this must be so because `example1` returns an `a`. We don't know what `a` is. And, `example1` takes in exactly one `a`. So, the only possible valid return value is the one and only `a` we were passed in! Now, it's possible to write various Haskell syntax that is different than precisely what we see above, but any variation is guaranteed to have exactly the same result (assuming totality). A student answer to a question like `example1` would be the implementation we see above plus a comment saying that this is the only possible implementation.

As another example, say you have to implement

```
example2 :: (a -> b) -> [a] -> [b]
```

(Do you recognize that type?) There are a variety of implementations of this type—in fact, an infinite number. But, one thing we know for sure is that if the second argument is an empty list, the result must also be empty. To see why, think about how we can possibly get a `b`. The only way is to use the provided function on an `a`. But, if the input list is empty, we have no `a`s! Thus, we can produce no `b`s, and the result list must also be empty. A suitable student answer for this would include an implementation and the comment "If the input list is empty, the output list must also be empty." Your answers do *not* need to include explanations.

For each exercise, you must include at least one interesting fact about functions implementing the type. More is better, though!

It is possible that some of the types below have *no* possible implementations. If that's the case, make the function's definition be `error "impossible"`. Your comment must say why the function is impossible to write.

Exercise 1

```
ex1 :: a -> b -> b
```

Exercise 2

```
ex2 :: a -> a -> a
```

Exercise 3

```
ex3 :: Int -> a -> a
```

Exercise 4

```
ex4 :: Bool -> a -> a -> a
```

Your answer must include information on how many distinct functions inhabit this type.

Exercise 5

```
ex5 :: Bool -> Bool
```

Your answer must include information on how many distinct functions inhabit this type.

Exercise 6

```
ex6 :: (a -> a) -> a
```

Exercise 7

```
ex7 :: (a -> a) -> a -> a
```

Exercise 8

```
ex8 :: [a] -> [a]
```

Exercise 9

```
ex9 :: (a -> b) -> [a] -> [b]
```

Exercise 10

```
ex10 :: Maybe a -> a
```

Exercise 11

```
ex11 :: a -> Maybe a
```

Exercise 12

```
ex12 :: Maybe a -> Maybe a
```

Binary search trees

A binary search tree is a recursive data stucture frequently used to store a *set*—that is, a chunk of data that is easily (and efficiently) searched and added to. Here is the datatype definition for a binary search tree in Haskell (included in `BST.hs`):

```
data BST a = Leaf | Node (BST a) a (BST a)
```

That is, a `BST a` is either a leaf (a placeholder that holds no data) or an interior node, containing left and right sub-trees and a chunk of data. The key property of a binary search tree is that the data in a left sub-tree must all be less than or equal to the data in a node, and that the data in a right sub-tree must all be greater than or equal to the data in a node. Search around online for more info if you're unfamiliar—there's plenty out there.

Some of you may have implemented a binary search tree in an imperative language (such as Java). It's *much* easier in Haskell!

Exercise 13 Write the insertion method for a binary search tree:

```
insertBST :: (a -> a -> Ordering) -> a -> BST a -> BST a
```

Recall the definition of `Ordering` (part of the `Prelude`):

```
data Ordering = LT | EQ | GT
```

The first parameter to `insertBST` is a comparison function that takes two `a`s and says what their relationship is. The next is the new element to insert. Last we have the current tree.

This function is actually quite simple—thinking through the answers to the following questions will essentially write the function for you:

1. What should you do when inserting into an empty tree (that is, a `Leaf`)?
2. What should you do when inserting `x` into a non-empty tree whose root node has a value greater than `x`?
3. What should you do when inserting `x` into a non-empty tree whose root node has a value less than `x`?

It is interesting to note that, because of parametric polymorphism, every call of `insertBST` must be accompanied by a comparison operation. Otherwise, there's no way to know how to compare elements. We'll see a mechanism—called *type classes*—that will make this less burdensome.

Visiting the library

An effective Haskell programmer must know how to use the standard libraries. The exercises below will require you to read through the documentation of `Data.List`, `Data.Maybe`, and `Data.Char` to write succinct solutions. Each of these functions has a simple, one-liner answer!

Many functions (especially in `Data.List`) have `Eq a =>` in the types. We'll see exactly what this means shortly. For now, we'll just say that those functions can only be called on concrete types, like `Int` or `String`, but not unknown types, like `a`.

In our quest to avoid partial functions, you may want to use these:

```
safeHead :: [a] -> Maybe a
safeHead []    = Nothing
safeHead (x:_) = Just x

safeTail :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (_:xs) = Just xs
```

Exercise 14 Check to see if a list of strings contains only capitalized words:

```
allCaps :: [String] -> Bool
```

Examples:

```
allCaps ["Hi","There"]   == True
allCaps []               == True
allCaps ["", "Blah"]     == False
allCaps ["Hi","there"]   == False
```

Exercise 15 Drop the trailing whitespace from a string:

```
dropTrailingWhitespace :: String -> String
```

Examples:

```
dropTrailingWhitespace "foo"   == "foo"
dropTrailingWhitespace ""      == ""
dropTrailingWhitespace "bar  " == "bar"
```

Exercise 16 Get the first letter (if it exists) of a list of strings:

```
firstLetters :: [String] -> [Char]
```

Examples:

```
firstLetters ["foo", "bar"] == ['f','b']
firstLetters ["alpha",""]   == ['a']
firstLetters []             == []
firstLetters ["",""]        == []
```

Exercise 17 Render a proper bracketed list given a list of strings:

```
asList :: [String] -> String
```

Examples:

```
asList ["alpha","beta","gamma"] == "[alpha,beta,gamma]"
asList []                       == "[]"
asList ["lonely"]               == "[lonely]"
```