HASKELL PROGRAMMING PROBLEM SET 4

LENNART JANSSON AND BRANDON AZAD

Read Chapter 7 and Chapter 8 of *Learn You a Haskell*, but you can ignore the bits about typeclasses and functors. We'll discuss them in depth next week.

Using Maps

Here are a couple problems to practice using the `Data.Map` type. It's a good idea to import the module qualified to avoid namespace collisions.

Problem 1. Histograms

Write a function `makeHistogram :: [a] -> Map a Int` that takes a list of elements and creates a `Map` from the distinct elements of the list to the number of times each element appears in the list. (Hint: higher-order functions make things pretty!)

```
> makeHistogram "hello"
fromList [('e',1),('h',1),('l',2),('o',1)]
```

Problem 2. Changing passwords

In this problem we'll be dealing with the following types and type synonyms, that could be used in the backend of a web application with user logins.

```
type Username = String
type Password = String
type PassMap = Map Username Password
type PassMapModifier = PassMap -> PassMap
data PasswordChangeViewmodel = PCVM {
  cpvUsername :: Username,
  cpvOldPassword :: Password,
  cpvNewPassword1 :: Password,
  cpvNewPassword2 :: Password
  } deriving (Show)
```

Write a function `changePasswordModifier :: PasswordChangeViewmodel -> PassMapModifier`, that will give a `PassMapModifier` that will change a user's password, which is a function that can be applied to the actual `Map` being used by the application. For security, the password should only be changed if the old password matches the user's current password, and if the two new passwords match.

In real-world Haskell programming, it's considered good practice to use type synonyms like this, even though `Username`s and `Password`s aren't represented by different types, since it makes it even clearer what a data structure like `Map Username Password` should represent.

Practice with Maybe and Either

`Maybe` and `Either` are two very important types in Haskell that are seen almost everywhere elegant error handling is required. Here are a few easy problems to get a sense of how they behave.

Problem 3. Bind

In this function we will continue to be coy about what monads actually are and continue to give vague hints. Here you will implement bind for the Maybe monad.

Write a function `bindMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b`. If the first parameter is `Nothing` then the function can't be applied and the result should be `Nothing` as well.

```
> bindMaybe (Just "lennart") (λ name -> Map.lookup name passwordMap)
Just "mypassword"
```

```
> bindMaybe (Just "notauser") (λ name -> Map.lookup name passwordMap)
Nothing
```

Problem 4. Data.Maybe functions

The Data.Maybe module contains many useful functions for working with `Maybe`s. Here are two simple ones to practice implementing:

(1) Write the function `catMaybes :: [Maybe a] -> [a]`, which takes a list of `Maybe`s and returns a list of only the `Just` values.
```
> catMaybes [Just 1, Just 2, Nothing, Just 3]
[1, 2, 3]
```
(2) Write the function `mapMaybe :: (a -> Maybe b) -> [a] -> [b]`, which works the same as a standard `map`, but only keeps values in the return list if the function gives a `Just`, so when the function gives `Nothing` there is no corresponding value of type `b` in the return list.
```
> mapMaybe (λ n -> if (even n) then (Just (n + 1)) else Nothing) [1..4]
[3, 5]
```

Problem 5. ArrowChoice

This function comes from the module Control.Arrow, which is a typeclass related to generalized abstractions of functions. We probably won't have a chance to discuss it this quarter, but arrows are pretty cool.

Write the function `(+++) :: (b -> c) -> (b' -> c') -> (Either b b' -> Either c c')`.

Problem 6. More bind

Write a function `bindEither :: Either a b -> (b -> Either a c) -> Either a c`. Hm, that type signature looks slightly similar to the `Maybe` version. Probably the implementation should be similar as well! But what in the world is this good for...

Skew heaps

In the next few problems you will implement a skew heap data type, a simple, unstructured, breed of heaps well-adapted to Haskell that can be used to implement priority queues with every operation in at most $O(\log n)$ time. Cool!

A skew heap is built like a binary tree on the inside, where the heap property is respected. This means that every element is smaller than (or equal to) both of its children, and every element is larger than (or equal to) it's parent. The top of the heap is therefore the smallest element in the whole heap. Almost every operation on the skew heap is defined in terms of the union function, which combines two heaps into one and has type `union :: Ord a => SkewHeap a -> SkewHeap a -> SkewHeap a`. Therefore, we'll implement that first.

First, let's get used to the data type.

```
data SkewHeap a = Empty | SkewNode a (SkewHeap a) (SkewHeap a)
```

The way to parse this type: a `SkewHeap` that contains elements of type `a` is either `Empty`, or is a `SkewNode`, a node of a skew heap, that contains an element of `a` at the root, then a left subheap `SkewHeap a`, then a right subheap `SkewHeap a`. That's it! It's really identical in structure to the binary tree discussed in *Learn You a Haskell* Chapter 8, but with different names.

Problem 7. Simple things

Implement `empty :: SkewHeap a`. This should give a skew heap with no elements in it.

Implement `singleton :: a -> SkewHeap a`. This should take a single element and put it in a skew heap with just that one element.

Implement `null :: SkewHeap a -> Bool`, which should return `True` if the skew heap is completely empty, and `False` if it does contain some elements.

Problem 8. Union

To take the union of two heaps, we find the heap with smaller root element, call it t_a and the other t_b. Then, we take the union of t_b and the right child of t_a, and that becomes the new left child of t_a. Then the original left child of t_a becomes the new right child of t_a. Now t_a contains all the elements from both t_a and t_b, and the heap property is still respected, so the new t_a is the union of the two heaps and is returned.

Implement `union :: Ord a => SkewHeap a -> SkewHeap a -> SkewHeap a`.

Also, define a new operator, for example `(+*+)`, that is an infix synonym for `union`. This will make your code prettier down the line since union will be very useful for the following functions.

Problem 9. Insert and Extract

Implement `insert :: Ord a => a -> SkewHeap a -> SkewHeap a`. This takes an element and a skew heap and returns the skew heap with that element added.

Implement `extractMin :: Ord a => SkewHeap a -> (a, SkewHeap a)`, which takes a skew heap, and returns a tuple of the minimum element in the skew heap as well as the modified skew heap without that element.

Problem 10. List interface

Implement `fromList :: Ord a => [a] -> SkewHeap a`, which takes a list (not necessarily in sorted order) and puts the elements in a skew heap (where the heap property is respected).

Implement `toAscList :: Ord a => SkewHeap a -> [a]`, which takes a skew heap and gives a list of all the elements in the heap in ascending order.

Congratulations! You've implemented a new container type with a similar interface to `Data.Set` and `Data.Map`, and a not-too-shabby amount of code. Hopefully this gives you some idea about what goes on under the surface when you use container type modules: it's not mysterious stuff, just plain old Haskell you could understand or implement yourself.

As a reward…

Problem 11. Heap sort

Use skew heaps to implement `heapsort :: Ord a => [a] -> [a]` :)