CIS 194: Homework 7

Due Wednesday, 25 March

Figure 1: The Haskell logo is modelled after the bind function `(>>=)` in the `Monad` type class

Preliminaries

Just as Haskell `Strings` are not as efficient as `ByteStrings`, lists are not as efficient as `Vectors`. Before you begin this homework you should make sure that you have the `vector` package installed by typing `cabal install vector` in your terminal. The `Data.Vector` module is designed to be similar to `Data.List` so you should see all of your favorite functions defined there! In order to avoid naming collisions, we `import qualified Data.Vector as V`. This means that you need to prefix all function calls with `V`. For example, `V.fromList [1..10]` constructs a `Vector` from the list `[1..10]`. We have also imported the `Vector a` type as well as a few functions non-qualified. These functions are `cons`, `(!)`, `(!?)`, and `(//)`. You can look these up on Hoogle to figure out what they do.

Later in the assignment, we will also use the `Control.Monad.Random` module which comes from the `MonadRandom` package. You will probably need to install this package by typing `cabal install MonadRandom`.

Finger exercises

The next few exercises are an opportunity to show off what you've learned about Monads! They are focused on using `Vectors` and the `Maybe` monad.

Exercise 1 Write the `liftM` function. This function lifts a regular function into a monad and applies it to an argument in that monad.

```
liftM :: Monad m => (a -> b) -> m a -> m b
```

Example: `liftM (+1) (Just 5) == Just 6`

The `Control.Monad` module defines many more lifting operations (ie, `liftM2`, `liftM3`,...). Use one of these functions to implement the function

```
swapV :: Int -> Int -> Vector a -> Maybe (Vector a)
```

that takes in two indices and swaps the elements at those indices in some `Vector`. This function should use the *safe* indexing operation `(!?)` and not the *unsafe* one `(!)`. If either of the indices are out of bounds (ie `(!?)` returns `Nothing`), then the result should be `Nothing`. You will probably find the `(//)` function useful.

Example: `swapV 0 2 (V.fromList [1, 2, 3]) == Just (V.fromList [3, 2, 1])`

Example: `swapV 0 2 (V.fromList [1, 2]) == Nothing`

Exercise 2 Implement the function `mapM` that maps a monadic function across a list:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
```

Example: `mapM Just [1..10] == Just [1..10]`

Now, use `mapM` to define a function that takes in a list of indices and a `Vector` and returns a list of the elements at those indices in the `Maybe` monad. If any of the indices don't exist, the function should return `Nothing`. Again, use the safe indexing operator `(!?)`.

```
getElts :: [Int] -> Vector a -> Maybe [a]
```

Example: `getElts [1,3] (V.fromList [0..9]) == Just [1, 3]`

Randomized Algorithms

Randomization is an extremely powerful tool in the field of algorithms. Using randomization, computationally intractable problems can be accurately approximated and slow algorithms can be made faster. In this section, we will focus on *Las Vegas* algorithms. A Las Vegas algorithm is an algorithm that always returns the correct result and uses randomization to improve the *expected* runtime. In other words, it gambles with computing resources, not with the correctness of the output.

The problem with randomization is that it is impure; a randomized function may return a different result each time it is called. Does this mean we can't use randomization in Haskell? Of course not! To get around this impurity, we will use the randomness (`Rnd`) monad.

In the same way that `IO` actions are not actual computations, but rather *descriptions* of computations, functions in the `Rnd` monad are *descriptions* of randomized computations. To evaluate a randomized function, we can use:

```
evalRandIO :: Rnd a -> IO a
```

This makes a lot of sense; a value of type `Rnd a` is pure, but running it produces a value in the `IO` monad (which, as we know, is the only way to get an impure value in Haskell). To get a random value, use of the following functions:

```
getRandom  :: Random a => Rnd a
getRandomR :: Random a => (a, a) -> Rnd a
```

The only difference between these two functions is that `getRandomR` produces a random value between the two values supplied in the tuple and `getRandom` produces any value of type `a` where `a` is a member of the `Random` type class.

Exercise 3 Use the randomness monad to produce a random element of a `Vector`. This function should only return `Nothing` if the `Vector` has length 0.

```
randomElt :: Vector a -> Rnd (Maybe a)
```

Exercise 4 Now define the following two functions:

```
randomVec  :: Random a => Int -> Rnd (Vector a)
randomVecR :: Random a => Int -> (a, a) -> Rnd (Vector a)
```

These functions should produce vectors of the specified length that are populated with random elements. The elements of `randomVec n` can have any value of type `a` whereas the ones from `randomVecR n (lo, hi)` should all be within the specified range. Try to make use of functions from the `Control.Monad` module.

Exercise 5 In this exercise you will implement a function that shuffles a `Vector`. The Fisher-Yates algorithm shuffles the elements of an array such that each element is equally likely to end up in each position. The algorithm is defined as follows: for $i = n - 1$ down to 1, choose a random $j \in \{0, 1, \ldots, i\}$ and swap the elements at positions i and j. In order to save yourself some hassle, you can use the unsafe indexing operator `(!)`, but only use it if you are sure that it won't fail!

```
shuffle :: Vector a -> Rnd (Vector a)
```

Figure 2: Sorting is hard when you don't use the right algorithms!

Quicksort

The Quicksort algorithm is a very efficient sorting algorithm that was invented by Tony Hoare in 1960. The algorithm itself is very simple. You first choose a *pivot*, then partition the array into elements that are less than and greater than the pivot, and finally recurse on the two resulting sublists. The efficiency of the algorithm depends highly on the choice of pivot. More on this shortly!

Exercise 6 Before you implement Quicksort, you should implement a helper function that partitions a `Vector` around the element at a given index.

```
partitionAt :: Ord a => Vector a -> Int -> (Vector a, a, Vector a)
```

The first argument is the `Vector` to be partitioned and the second argument is the index of the pivot. The output is a 3-tuple containing

a `Vector` of the elements less than the *value* of the pivot, the value of the pivot itself, and a `Vector` of the elements greater than or equal to the pivot in that order.

Example:

```
partitionAt (V.fromList [5, 2, 8, 3, 6, 1]) 3 ==
  (V.fromList [2, 1], 3, V.fromList [5, 8, 6])
```

Example:

```
partitionAt (V.fromList [1, 6, 4, 7, 2, 4]) 2 ==
  (V.fromList [1, 2], 4, V.fromList [6, 7, 4])
```

Exercise 7 A naïve Quicksort implementation picks the first element in the array to be the pivot every time. In the worst case, this causes the array to be partitioned into a segment of size 0 and a segment of size $\Theta(n)$ causing the runtime to be $O(n^2)$.

An implementation of Quicksort that uses the first element as the pivot to sort *lists* is provided for you. Your job is to implement the same algorithm for `Vector`s instead of lists.

```
qsort :: Ord a => Vector a -> Vector a
```

Note that `Vector` is a monad! That means that you can use Monad Comprehensions to construct a `Vector` in the same way they are used for lists!

Exercise 8 We can improve the expected runtime of Quicksort from $O(n^2)$ to $O(n \log n)$ by making one simple modification. Instead of choosing the first element as the pivot each time, we will choose a *random* element to be the pivot. Intuitively, the reason why this is so effective is because there is a 50% chance that a random element has rank greater than $\frac{n}{4}$ and less than $\frac{3n}{4}$, therefore with good probability the array will be split into roughly even segments. Implement randomized Quicksort:

```
qsortR :: Ord a => Vector a -> Rnd (Vector a)
```

Remember that you implemented the `partitionAt` function! This function should be useful here.

We know that randomized Quicksort is *theoretically* faster than deterministic Quicksort, but is it really faster in practice? Try sorting `V.fromList (reverse [1..10000])`, the integers from 1 to 10,000 in reverse order, using both implementations:

```
> let v = V.fromList $ reverse [1..10000]
> qsort v
> evalRandIO $ qsortR v
```

Did you notice a difference? On my machine, the deterministic algorithm took over a minute and the randomized one was instantaneous!

Exercise 9 There is a randomized algorithm for *selection* that is very closely related to Quicksort. The purpose of this algorithm is to select the element with rank i in an *unsorted* array. For example, `select 0 v` selects the minimum element, `select (n - 1) v` selects the maximum element, and `select (n 'div' 2) v` selects the median.

The naïve implementation of this algorithm would simply sort the array and then return the element at position i. However, sorting is overkill here since all we care about is a single element. A better version uses the same divide and conquer approach that Quicksort uses and runs in expected linear time. First, you choose a *random* pivot p and partition the array around it. This gives you left and right sub-arrays L and R. If $i < |L|$, then you know that the element you are looking for is in L, so you should recurse on L. If $i = |L|$, then p must have rank i. Finally, if $i > |L|$, the element must be in R, so you should recurse on R searching for rank $i - |L| - 1$. Implement the function:

```
select :: Ord a => Int -> Vector a -> Rnd (Maybe a)
```

This function selects the element of rank i by the algorithm described above. Notice that this function returns an `Rnd (Maybe a)`. This is because the rank that is asked for may be outside the bounds of the `Vector`. In this case, you should return `Nothing`.

Playing Cards

The `Cards.hs` module defines several data types to represent playing cards. The code in this module is mostly boilerplate, but you should familiarize yourself with the `Deck` and `Card` types before you start the next few exercises.

Exercise 10 It would be useful to be able to get a deck that contains all of the cards grouped by suit (`Spade`, `Heart`, `Club`, `Diamond`), and arranged from `Two` to `Ace`. Implement:

```
allCards :: Deck
```

which is simply a `Vector` of `Cards` arranged in the order stated above (don't worry, this is *much* easier than `allCodes` from HW02). You should implement `allCards` as a Monad Conprehension. You will probably find `suits` and `labels` in the `Cards` module useful.

Now that you have `allCards`, write:

```
newDeck :: Rnd Deck
```

This should return a new `Deck` that contains all of the cards from `allCards`, but in a random order. Remember that you already implemented `shuffle`!

Exercise 11 We also need some sort of *uncons'ing* operation for `Decks`. That is, a function that takes in a `Deck` and gives you the head and tail (since we can't pattern match on `Vector` like we can with lists). Implement the function:

```
nextCard :: Deck -> Maybe (Card, Deck)
```

This function takes in a `Deck` and returns the head and tail of the `Deck` in the `Maybe` monad. If the `Deck` is empty, it should return `Nothing`.

Exercise 12 In many card games, we need to draw multiple cards at once from a deck. Implement the function:

```
getCards :: Int -> Deck -> Maybe ([Card], Deck)
```

This function should draw n cards from the given `Deck` where n is the first input to the function. Try to make use of `nextCard` and the `Maybe` monad in order to avoid pattern matching on `Maybe`. If the `Deck` has fewer than n `Cards` remaining then this function should return `Nothing`.

Exercise 13 Relax! This isn't a *real* exercise. Now that you have implemented some operations on the `Deck` type, you can have some fun. Compile this file with GHC by typing `GHC HW07.hs -main-is HW07` and then run the resulting executable.

In the spirit of the *Las Vegas* algorithms you implemented earlier, this is a text-based War game with betting. Each round, you place a bet and then you and the computer both draw a card from the deck. Whoever has a higher card wins. If there is a tie, then each player draws 3 more cards and the last card that each player draws is compared.

The game ends when you either run out of money, the deck is empty, or you choose to leave. Have fun, and gamble responsibly!

Figure 3: Don't put all your eggs in one basket!