CS 463 - Homework 8 - Monads

Due Friday, April 29th, 5pm

ChangeLog

- added notes on using `cabal` instructions to enable the `mtl` (monads) module.

Assignment Details

- **Turn It In on BlackBoard**
- **Starter Template with blank definitions**: Template8H.hs (be sure to rename it to `Homework8.hs`).
- **Tester File**: Tester8H.hs
- **You may work with one partner**. You *must* list them in your top comment-block (or "none" if you work alone), and you still must turn in your own work.

Table of Contents

You will create a file named `Homework8.hs`, which begins with:

```
{-
Your Name: ____________________
Partner:   ____________________

-}

module Homework8 where

import Control.Monad
import Control.Monad.State      -- State, runState, get, put, guard

data SnocList a = Lin | Snoc (SnocList a) a deriving (Show, Ord)
data Tree a = L | V a | Br (Tree a) a (Tree a) deriving (Show)

type Name = String
type FamilyTree = [(Name,Name)]
```

Some definitions are very short and simple, others are more significant computations. You might be able to work on any of the four sections, instead of needing to go straight through via the file's ordering.

Getting access to the `mtl` module (for monads)

More recent installations of Haskell hide some modules by default. You can explicitly install them for yourself with the two following commands to the `cabal` package management system:

```
cabal update
cabal install mtl --lib
```

The first one ensures you have up-to-date definitions; the second one both requests installation of the `mtl` library as well as asks that it be used for the current user as a library and not a stand-along project.

You may also need to run this variant; I'm not yet sure which exact installation path/platform may need this but it seems geared towards the newest version of cabal.

```
cabal new-install --lib mtl
```

These instructions and more are included in our class notes on Haskell. If you are unable to run the `cabal` instructions, you can also download the source code directly (from hackage) and carefully put the `Control/` folder adjacent to your code. That would cause `ghci` to use that local copy in lieu of the installed approach, and wouldn't need any changes on the grading side.

General Notes on monadic code

- In general, a line in do-notation such as this:

  ```
  v <- funcM args
  ```

  implies that funcM has some type `funcM :: argsTypes -> m a`, and that `v::a` accordingly. You might think of the binding action as *calculating* `v`, or as an alternative notion, *running* the monadic expression (`funcM args`) to get to the answer, `v`. In the rest of the code, we can use `v` anywhere we need a value of type `a`, because that is its type. But we can't use `funcM args` as if its type were `a`, because it's not - it *represents a monadic computation that will yield something of type a*.

- The notation for `let` bindings in do-notation is a bit different: we simply leave off the `in` portion, because all following lines are "after" the let line. Lets can be useful for pattern-matching something to pick it apart, naming an expression that will be used multiple times, or generally just giving things names. But note! the expression given a name by the let-binding doesn't undergo any unboxing in the sense that an `v <- funcM args` line does for `v`. If you also need to perform the monadic action, then you can pattern-match on those as well:

  ```
  (Just an_example) <- performM something
  ```

- Assume the following types of things for our next discussion, where `m` is some monad (such as `Maybe`, `State`, etc):

  ```
  funcM :: Int -> Int -> m Int
  func2M :: Int -> Int -> Int -> m Int
  calculateTripletM :: [Int] -> m (Int,Int,Int)
  max :: Int -> Int -> Int    -- from Prelude

  w, x, y, z :: Int
  args :: [Int]
  ```

  Consider this meaningless code (it's one big expression):

  ```
  do
     a <- funcM w x
     b <- funcM y z
     (v1, v2, v3) <- calculateTripletM args
     let biggest = max (max v1 v2) v3
     c <- func2M a b biggest
     return (c + v2*v3)
  ```

  We see that `biggest` is defined with a let-binding; but that line itself doesn't do anything monadic. `max(max v1 v2) v3)` has no monadic types in use anywhere in there, and so `biggest::Int` also does nothing monadic on its own, as its type guarantees.

Datatypes, Typeclasses

1. `SnocList a`

A "snoclist" is like a cons-style list, except instead of storing the first and the rest, we store the init and the last item. (init is everything except the last item). Although we should be able to do everything with a snoclist as a conslist, it is more convenient to access things near the end of the list than from the beginning of the list.

```
data SnocList a = Lin | Snoc (SnocList a) a    deriving (Show,Ord)
```

These two list values are logically the same list, though of course different Haskell values:

```
(1 : (2 : (3 : [] )))
(Snoc (Snoc (Snoc Lin 1) 2) 3)
```

Specifically, they're in the same order. `1` is the first value in both list representations.

1. **`instance (Eq a) => Eq (SnocList a)`**. Implement the `Eq` typeclass so that a snoclist of equatable things is equivalent to another if all the items at the same positions are equivalent (==), and there are the same number of items in each list.
2. **`instance Functor SnocList`**. (Note there's no type-variable after `SnocList`). Implement the `Functor` typeclass; to `fmap` a function across a snoclist will preserve the shape and apply the function at each spot in the snoclist to generate the new snoclist.
3. **`snocLast :: SnocList a -> Maybe a`**. Given a snoclist, since it might have items in it or not, find the last item and return `Just` that if present; return `Nothing` otherwise. (This is a really short function.)
4. **`snocProduct :: (Num a) => SnocList a -> a`**. Multiply all the ints in the snoclist together. An empty snoclist would result in `1`.
5. **`snocMax :: (Ord a) => SnocList a -> Maybe a`**. Given a snoclist of orderable things, return the largest when present. Hint: this does not particularly benefit from using Maybe as a monad; just use it like a plain datatype for this function.
6. **`longestSnocSuffix :: (Eq a) => SnocList a -> SnocList a -> SnocList a`**. Given two snoclists, return a snoclist of the longest-matching *suffix*.
7. **`snocZip :: SnocList a -> SnocList b -> SnocList (a,b)`**. Given two snoclists, zip them together *from the end towards the front*, with the answer no longer than the shortest list of the two.
8. **`snocify :: [a] -> SnocList a`**. Given a regular list, convert it into a snoclist. You must preserve the original ordering.
9. **`unSnocify :: SnocList a -> [a]`**. Given a snoclist, convert it into a regular list, preserving the values and ordering.
10. **`uniques :: SnocList Int -> SnocList Int`**. Given a snoclist of ints, return the list of items that are unique in the original snoclist; preserve ordering of the last occurrence of each unique value. (like removing earlier duplicates until the list has all unique values in it, though this isn't a great algorithm).
11. **`snocReverse :: SnocList a -> SnocList a`**. Reverse the contents of a snoclist. You could call the built-in `reverse`, but it only works on regular lists, no `SnocList` values.

2. `Tree a`

We use this particular kind of tree, which has an empty constructor, a one-value constructor, and a binary-with-value constructor.

```
data Tree a = L | V a | Br (Tree a) a (Tree a) deriving (Show)
```

Implement the following things:

1. **`instance (Eq a) => Eq (Tree a)`**. The two trees being compared must be the same shape with the equal values at each spot in order to be considered equal.
2. **`instance (Ord a) => Ord (Tree a)`**. Given two trees, return an `Ordering` value (`LT`, `EQ`, or `GT`) that relates them.
   - The first structural difference or value difference in an *in-order* traversal that differs answers when one is less than or greater than the other.
   - We define that an empty leaf is less than any non-empty leaf.
   - We define that `V` node is always less than a `Br` node.
3. **`insertTree :: (Ord a) => a -> Tree a -> Tree a`**. Given a value and a tree that is ordered (exhibits an ordered in-order traversal), insert the value in the first in-order position available that preserves in-order ordering. (Really, we're building an entirely new tree with the new item present in this new value, we're not actually modifying the old value).
   - to add a value to the left or right of a `V`, convert it into a `Br` and put the new value to the left or right accordingly.
   - insert a duplicate value to the *left* of existing matching values.
4. **`inOrder :: Tree a -> [a]`**. Walk the tree in-order, and generate the list of values as they are visited.
5. **`treeSort :: (Ord a) => [a] -> [a]`**. Insert all the items of a list into a tree, then walk it `inOrder` to get the sorted list.
6. **`treeMin :: (Ord a) => Tree a -> Maybe a`**. Given a tree of values, give back the maybe minimum value. This tree is not guaranteed to have any internal ordering.

Monads

3. The Maybe Monad

Every class in Java should eventually reach `Object` when tracing up through parent classes. We will write some functions that deal with finding ancestors, most-specific shared ancestors, and so on.

You must use `Maybe` as a monad (e.g. with do-notation) on `ancestors` and `leastUpperBound` for full credit. Otherwise a five point deduction may be assessed.

Required type synonyms: `Name` and `FamilyTree` (mentioned at top of this file).

1. **`parent::Name -> FamilyTree -> Maybe Name`**. (not a monadic definition; will be used in one later). Given a name of a Java class, a `FamilyTree` to look for this class's single parent's name, dig through and either find it (returning `Just` the parent's name) or fail to find it (returning `Nothing`). (You don't need to use the maybe monad just yet - we're creating a function that returns a maybe-value that we can use later on).
   - first, let's use this definition in our samples:

```
familyTree = [
  ("Animal", "Object"),
  ("Cat","Animal"),
  ("Dog","Animal"),
  ("Siamese","Cat"),
  ("Calico","Cat"),
  ("Labrador","Dog"),
  ("Pug","Dog"),
  ("Book","Object"),
  ("Garbage","Can")
  ]
```

   - and here are some sample runs:

```
*Code> parent "Animal" familyTree
Just "Object"
*Code> parent "Object" familyTree
Nothing
*Code> parent "Siamese" familyTree
Just "Cat"
*Code> parent "Pug" familyTree
Just "Dog"
```

2. **`ancestors::Name -> FamilyTree -> Maybe [Name]`**. You must use do-notation and the `Maybe` monad for this recursive definition that calls upon `parent` to construct the list of ancestors from the given family tree. When we reach `Object`, we are done searching; if any step of ancestry can't find a parent, we are done (with `Nothing` to return).
   - one special corner case - `ancestors` of "`Object`" should be `Just []`.

```
*Code> ancestors "Dog" familyTree
Just ["Animal","Object"]
*Code> ancestors "Book" familyTree
Just ["Object"]
*Code> ancestors "Garbage" familyTree
Nothing
*Code> ancestors "Calico" familyTree
Just ["Cat","Animal","Object"]
*Code> ancestors "NotEvenPresent" familyTree
Nothing
```

3. **`headMaybe::[a] -> Maybe a`**. (not a monadic function; will be used in one later). Given a list, return `Just` the first item if it exists, otherwise `Nothing`.

```
*Code> headMaybe []
Nothing
*Code> headMaybe [5]
Just 5
*Code> headMaybe [5,10,15]
Just 5
```

4. **`leastUpperBound::Name -> Name -> FamilyTree -> Maybe Name`**. You must use do-notation and the `Maybe` monad for this definition. Given two names, what is the most specific type that they share in their chains of inheritance? Either one might not have successfully been mapped all the way to `Object`, yielding `Nothing`.
   - Watch out for when one is a subtype of the other!
   - *hint*: though not required, you might find use for some of your `SnocList` definitions here…

```
*Code> leastUpperBound "Pug" "Calico" familyTree
Just "Animal"
*Code> leastUpperBound "Pug" "Animal" familyTree
Just "Animal"
*Code> leastUpperBound "Pug" "Book" familyTree
Just "Object"
*Code> leastUpperBound "Pug" "Garbage" familyTree
Nothing
*Code> leastUpperBound "NOTFOUND" "Book" familyTree
Nothing
```

4. The `State` Monad

1. First, a warmup.
   - **`tribM::Int -> State (Int,Int,Int) Int`** . (Note the return type is `State _ _`; that informs you that you will need to write this with do-notation, assuming you don't want to do the deep dive into `>>=` usage). This is not quite the same as our in-class presentation of fibonacci with the `State` monad, but it's similar. We accept one int argument `n`, and we will use the `State` monad to store the three previous tribonacci numbers as the state. We eventually return the *nth* tribonacci item when we're done recursing on smaller and smaller values for `n`.
   - **`trib::Int -> Int`** . Use `runState` or `evalState` to successfully run your `tribM` definition and return that nth tribonacci number.
2. **Partitioning Lists**. Given a predicate function `(a->Bool)` and a list of values `[a]`, separate the list into two lists of items that passed and didn't pass the predicate function's test. You will create two definitions:
   - **`parti::(a->Bool) -> [a] -> ([a],[a])`**. This first version can be written without any monadic behavior; it just has to work as defined.

     ```
     *Code> parti even [1,2,3,4,5,6]
     ([2,4,6],[1,3,5])
     *Code> parti (<5) [8,7,6,5,4,3,2]
     ([4,3,2],[8,7,6,5])
     *Code> parti even [1,3,5]
     ([],[1,3,5])
     *Code> parti odd [1,3,5]
     ([1,3,5],[])
     ```

   - **`partitionM::(a->Bool) -> [a] -> State [a] [a]`**. This is a state-monadic version that stores the *passing* items as its state, and collects the *failing* items as its answer-value. You might find that you will use similar tactics as in your non-monadic code: multiple pattern matches, guards, recursive calls, etc. But you must also write with the monadic do-notation. Remember, `get` and `put` are some of your most basic tools when writing state-monadic code. They access the stored state.
     - when something passes the predicate test, you need to update the list that is the state so that it's included.
     - when something fails the predicate test, you still need to recurse through the rest of the list, but it won't necessarily be tail recursion (as in, you might not have the recursive call be the last bit of effort in that equation). After sequencing the current spot with the rest you can build up the entire answer-list of all that failed, and return it.
     - note that you can't really test this on its own without calling `runState`, which conveniently yields "the resulting answer" as well as "the last state". Piece these together when you implement **`partition :: (a->Bool) -> [a] -> ([a],[a])`**. It must use `runState` and your `partitionM` definition (see below for examples).

       ```
       *Main> runState (partitionM even [1,2,3,4,5,6]) []
       ([1,3,5],[2,4,6])
       *Main> runState (partitionM even [1,2,3,4,5,6]) [1000]   -- injecting more stuff into the initial state.
       ([1,3,5],[1000,2,4,6])
       *Main> runState (partitionM (<5) [7,6,5,4,3]) []
       ([7,6,5],[4,3])
       *Main> runState (partitionM even [1,3,5]) []
       ([1,3,5],[])
       *Main> runState (partitionM odd [1,3,5]) []
       ([],[1,3,5])
       *Main> let heavy (a,b) = a>b
       *Main> runState (partitionM heavy [(5,3), (6,7), (10,1),(2,2)]) []
       ([(6,7),(2,2)],[(5,3),(10,1)])
       *Main> :t runState (partitionM heavy [(5,3), (6,7), (10,1),(2,2)]) []
       runState (partitionM heavy [(5,3), (6,7), (10,1),(2,2)]) [] :: (Ord a, Num a) => ([(a, a)], [(a, a)])
       ```

   - **`partition::(a->Bool) -> [a] -> ([a],[a])`**. This must use your `partitionM` by feeding it to `runState`. `runState` gives back both its notion of "the answer" as well as "the last state". Carefully piece this together into the answer for your function as needed. Remember, `runState` returns `(a,s)`, but our answer `a` is the failure values, and our state `s` is the passed-the-test values, which is not the order we expect in the answer of `partition`.

1. **Balanced Things** We will analyze a string to check if all parentheses, braces, angle brackets, and square brackets are well nested. We allow strings of any characters, and ensure that all parentheses `()`, braces `{}`, angled brackets `<>`, and square brackets `[]` are legally balanced. This means we ignore all other characters other than those eight, and then also enforce that each opening of any kind is closed by the matching kind of close, and in the same nested order as they were opened.

   `simpleBalanced::String -> Bool`
   non-monadic version. You learn the calculations involved in a supposedly more familiar environment. You may want to use helper functions here.

   ```
    *Code> simpleBalanced "(){}<>[]"
   True
   *Code> simpleBalanced "(([{}])[<>]<>{})"
   True
   *Code> simpleBalanced "(others [are] allowed)"
   True
   *Code> simpleBalanced "(()"      -- needed another close-)
   False
   *Code> simpleBalanced "([)]"     -- these are tangled, not nested.
   False
   *Code> simpleBalanced ""
   True
   ```

   `balancedM::String -> State [Char] Bool`
   monadic version that stores a *stack* of unclosed pairings that have been opened earlier in the string; the `Bool` result answers if the entire string is balanced or not.
   - treat the stored state, `[Char]`, like a stack. Push things onto it sometimes, and require things successfully pop off when needed.
   - you might enjoy making helpers that `push` and `pop`, but ultimately your "stack" manipulation will happen via uses of the standard `get` and `put` methods from the `MonadState` typeclass, like in our class examples.
   - while it might not ultimately look much prettier, the goal is to see a stateful style of computation, where the stack of unmatched things is maintained and modified as we go through the list.

   `balanced::String -> Bool`
   Use `balancedM` definition here to implement the same functionality as `simpleBalanced`.

   ```
   *Code> balanced "(){}[]"
   True
   *Code> balanced "[[]"
   False
   *Code> balanced "[(])"
   False
   *Code> balanced "[others {are} okay]"
   True
   ```

5. The List Monad

Use do-notation and the list monad to calculate each of the following. Some of them may be quite short definitions. It's okay to make helper functions and call them inside, as long as you're still doing list-monadic things in the main definitions. Although list comprehensions are quite similar, you will not get credit if you turn in list comprehension solutions.

1. `divisors :: Int -> [Int]` :: finds all the divisors of the first argument.

   ```
   *Code> divisors 12
   [1,2,3,4,6,12]
   *Code> divisors 1
   [1]
   *Code> divisors 100
   [1,2,4,5,10,20,25,50,100]
   *Code> divisors 1117
   [1,1117]
   *Code> divisors 1119
   [1,3,373,1119]
   ```

2. `geometric :: Int -> Int -> [Int]` :: Given a starting value and a factor, generates the infinite sequence of numbers produced by multiplying previous values by the factor.

```
*Code> take 6 $ geometric 1 5
[1,5,25,125,625,3125]
*Code> take 6 $ geometric 2 5
[2,10,50,250,1250,6250]
*Code> take 6 $ geometric 5 1
[5,5,5,5,5,5]
*Code> take 6 $ geometric 5 2
[5,10,20,40,80,160]
*Code> take 6 $ geometric 0 4
[0,0,0,0,0,0]
*Code> take 6 $ geometric 4 (-1)
[4,-4,4,-4,4,-4]
```

3. `mersennes :: [Int]` :: The (infinite) list of Mersenne numbers `M_1`, `M_2`, `M_3`, … . They are not all prime, though the largest prime number known happens to be a Mersenne number. Each Mersenne number is of the form `M_n = 2^n - 1`.

```
mersennes == [1,3,7,15,31,63,127,...]
```

4. `unitTriangles :: Int -> [(Int,Int,Int)]`. All sides of a triangle must be less than the sum of the other two legs (or else they couldn't reach each other). Given an upper limit `n`, create all the triplets of triangle side lengths that are whole numbers, from shortest to longest.