# Machine Problem: Data Types and HOFs

## Overview

In this assignment you'll try your hand at defining and writing functions for your own data types, using a handful of higher order functions along the way.

## Starter code repository

Claim your repository via the invitation link on the class homepage and git clone your starter code.

Next, before continuing, remember to sign the honor pledge in the "package.yaml" file so we know who your submission belongs to.

## Part 1: Binary Tree

For the first part of this machine problem you'll start by implementing a number of functions that operate on a binary tree data type, defined for you as:

```
data BinTree a = Node a (BinTree a) (BinTree a)
```

Unlike the `Tree` type covered in class (found at line 338 of "src/Lect/Lect08Complete.lhs"), this tree type has only one value constructor -- `Node` -- which takes a value and two trees, which we'll refer to as the left and right subtrees. Because there is no terminal (i.e., non-self-referential) value constructor, all `BinTree`s are effectively infinite!

Here is a (partial) `BinTree`, where the nodes are populated with the natural numbers, starting at the "root" of the tree and then downwards and from left to right across each level of the tree:

```
Node 1
  (Node 2
    (Node 4 ...) (Node 5 ...))
  (Node 3
    (Node 6 ...) (Node 7 ...))
```

Here is an ASCII art representation of the above `BinTree`:

```
     1
   /   \
  2     3
 / \   / \
4   5 6   7
```

You are to implement the following functions that take/return `BinTree`s, found and documented in "src/MP3a.hs":

1. `treeRepeat :: a -> BinTree a`
2. `treeNats :: BinTree Integer`

3. `treeVal :: [Direction] -> BinTree a -> a`
4. `treeToList :: BinTree a -> [a]`
5. `treeFlip :: BinTree a -> BinTree a`
6. `treeFromList :: [a] -> BinTree a`
7. `treeIterate :: (a -> a) -> a -> BinTree a`

Finally, make the `BinTree` an instance of the `Functor` and `Applicative` classes. `fmap` should apply the given function to all values in a tree. `(<*>)` should apply functions found in the first tree to values found in corresponding nodes in the second tree. E.g.,

```
> take 20 $ treeToList $ (+) <$> treeNats <*> treeFlip treeNats

[2,5,5,11,11,11,11,23,23,23,23,23,23,23,23,47,47,47,47,47]
```

## Part 2: Poker Hand Analysis

In part 2 you'll come up with your own type definitions to use in representing playing cards, and write functions to determine what poker hand corresponds to a five-card list and to come up with aggregate hand statistics.

If you're not already familiar with the standard poker hands, see this page for a quick briefer.

Notice that `Hand` is already defined for you --- it contains values that represent all 10 hand types, from lowest to highest in strength.

After deciding on your card representation (at the very least you'll need supporting types to represent ranks and suits), you'll need to define `deck`, which represents all the cards in a standard playing card deck --- i.e., 52 `Card`s in any order.

When you've finished defining `deck`, you can try out the provided `genDeck`, `genHand`, and `genHands` functions, all of which are based on a random shuffle of the base `deck` of cards. Note that all three functions return a `Gen a` type, which requires that you run it through the `generate` function in order to produce a randomized result in an I/O action (which we'll talk about in class soon).

E.g., to get a randomized deck of cards, do:

```
> generate genDeck
[Card ...]
```

to get a hand of five cards, do:

```
> generate genHand
[Card ...]
```

and to get multiple five-card hands (up to 10 hands from a 53-card deck), do:

```
> generate $ genHands 5
[[Card ...], [Card ...], ...]
```

Next, you'll need to implement the following two functions (and any supporting functions you wish):

1. `hand`: takes an array of 5 cards and returns the strongest hand they can be used to make.

   Examples (your `Cards` representations will likely look different):

   ```
   > hand [Card 2 H, Card 3 D, Card Ace H, Card 5 D, Card 4 S]
   Straight

   > hand [Card 2 D, Card 3 C, Card 2 C, Card 3 D, Card 2 H]
   FullHouse
   ```

   Though you may not include additional libraries for use in your implementation, you should definitely check out the sort and group functions and others from `Data.List`. And don't forget your higher order functions (e.g., `map`, `filter`, etc.)!

2. `computeStats`: takes an array of 5-`Card` arrays, and returns a list of tuples of type `(Int, Hand)`, where each tuple indicates the number of times a certain `Hand` occurs in the input list. The tuples should be sorted in decreasing order of frequency.

   To test your function, you can run `computeStats` on hands returned by `genHands`, like so (see if you can figure out what the `<$>` operator is being used for here):

   ```
   > computeStats <$> (generate $ genHands 8)
   [(3,HighCard),(3,Pair),(1,TwoPair),(1,ThreeOfAKind)]
   ```

   We also provide you with `test`, which calls `computeStats` on N hands dealt from separate, freshly shuffled decks. This will give you a good sense of the statistics of various poker hands. Here are outputs we got on some sample runs (yours, of course, will differ):

   ```
   > test 1000
   [(498,HighCard),(419,Pair),(38,TwoPair),
    (32,ThreeOfAKind),(10,Straight),(3,Flush)]

   > test 100000
   [(50166,HighCard),(42220,Pair),(4779,TwoPair),
   (2092,ThreeOfAKind),(387,Straight),(192,Flush),
   (141,FullHouse),(22,FourOfAKind),(1,StraightFlush)]

   > test 1000000
   [(500430,HighCard),(423893,Pair),(47202,TwoPair),
   (20930,ThreeOfAKind),(3926,Straight),(1891,Flush),
   (1465,FullHouse),(250,FourOfAKind),(12,StraightFlush),
   (1,RoyalFlush)]
   ```

You can also test your implementation by using the provided `app/Main.hs` to test your implementation by running `stack exec mp3 N` from the command line, where `N` is the number of hands to generate.

```
module Main where

import System.Environment
import MP.MP2
```

```
main :: IO ()
main = do
    args <- getArgs
    stats <- test $ (read $ head args :: Int)
    print stats
```

And you can now do `stack build`, followed by `stack exec cs340-exe N` from the command line to gather poker statistics from the command line.

## Part 3: Writing Tests

For part 3, you are to add tests to `src/test/MP/MP2Spec.hs` to verify that your functions above (for parts 1 and 2) work correctly. We do not require that you write property-based tests, nor will we check that your tests are 100% thorough --- we hope that pride will motivate you to get there on your own! We do, however, ask that you achieve at least 80% code coverage in your tests.

Remember that to check code coverage, you can do stack test --coverage to generate a test coverage report, as described in Lect05.

## Grading

### Part 1:

- Each function implementation in part 1 is worth 3 points; there are 7 functions and 3 methods, worth a total of 30 points. undefined or non-compiling functions are worth no points

### Part 2:

- The `Card` data type and an accompanying, logical `deck` definition are worth 5 points
- The `hand` and `computeStats` functions are worth 10 points each

### Part 3:

- A test suite that achieves 80% code coverage is worth 10 points; points are allocated linearly starting at 20% code coverage (anything below 20% earns 0 points for this part)

---

Maximum points = 30 + 25 + 10 = 65

## Submission

First, make sure you correctly signed and committed the "package.yaml" file. We won't be able to easily map your repository to your name if you don't!

To submit your work, simply commit all changes and push to your GitHub repository.

---

Last updated: Mon Mar 21 14:47:35 2022