Midterm 2 Study Guide

You should read all the notes we have discussed starting from random numbers, and the corresponding textbook sections. These questions are here to help guide your studies, but are not meant to be exhaustive of everything you should know (though they do try to touch all the areas).

In all "coding" problems below, you must always include a type for each value/function/action you write.

1. Be able to write random-number-generating functions of various forms, both functions that take as input a generator and return a value/generator pair as well as functions that use and update the standard generator in order to build IO value results (getManyIO in the notes is such an example). Some examples:

   - generate the sum of the rolls of two 6-sided dice.
   - generate a random capital letter.
   - generate a random string of capital letters of a provided length.
   - generate one of the values True or False, with a 4/5th chance of generating True.
   - select at random one from a list of values.

2. Adapt the work of the shuffle function to write a function that given a list of values and a number n returns a random selection of n of the values from the list, without being allowed to select the same value twice.

3. For the following typeclass/type pairs, implement the corresponding instance (i.e. how the type is an instance of that class):

   - Eq and Maybe Int, Eq and [Int]
   - Num and MyInt where data MyInt = Neg Integer | Zero | Pos Integer represents the integers where the sign is specified via the prefixes. For example -5 in this type is actually Neg 5. The integers contained in Neg Integer and Pos Integer are required to be positive numbers. To define Num for such numbers you need to define (+), (−), (∗), negate, abs, signum which returns (in the type) -1, 0 or 1 depending on whether the provided numbers if negative/zero/positive, and also fromInteger.
   - Eq and Num for the following MyInt type: data MyInt = Actual Integer | PosInf | NegInf | Unknown (the extra values standing for positive infinity, negative infinity and unknown indicating that it is the result of an operation that has unpredictable results, like doing infinity minus infinity) with the following rules:
     - Actual integers are equal when the numbers are the same. Positive infinity is equal to itself, and negative infinity is equal to itself. Unknown is not equal to any number, including itself. Implement both (==) and (/=) to achieve this.

- Operations on actual integers result in the appropriate actual integers. Any operation involving Unknown results in Unknown.
- Infinity plus or minus an actual number is the same kind of infinity.
- Positive infinity plus or times another positive infinity is positive infinity.
- Negative infinity plus another negative infinity is negative infinity.
- Negative infinity times another negative infinity is positive infinity.
- Positive infinty times negative infinity (or other way around) is negative infinity.
- Negative infinity minus positive infinity is negative infinity. Positive infinity minus negative infinity is positive infinity.
- Negative infinity minus negative infinity, or positive infinity plus positive infinity, are both unknown.
- Multiplying any infinity by 0 is unknown.
- Multiplying any infinity by a non-zero number is an appropriately signed infinity (e.g. positive infinity times a negative number is negative infinity).
- abs of both infinities would be positive infinity. signum of negative infinity would be -1, and of positive infinity would be 1.

4. Be able to implement parts of a class like `Fraction` as described in the *modules and hiding information* notes.

5. (Big problem, could ask you for parts) We want to define a class `ModInt` for integers modulo a number. For example the numbers "5 modulo 6" and "11 modulo 6" are actually the same. We decide to store this information via a data type `data ModInt = MI Int Int` where the first integer is the number and the second is the modulo. E.g. `MI 5 6` would be the number described above. Write a module `Modulo` that provides this type with the following conditions:

   - All created values of this type must satisfy the condition that the modulo is greater than 1 and the number is from 0 to one less than the modulo. You must maintain this invariant. Users should not be able to create values that don't satisfy it.
   - `ModInt` should be an instance of `Eq`. Numbers with different modulo are not equal, and numbers with the same modulo are equal if they are equal modulo that number (your invariant makes that an easy check though).
   - `ModInt` should be an instance of `Num`. Numbers with different modulo should error, while those with the same modulo should behave appropriately. Calling abs or signum should error.
   - There should be a `sameModulo` function which returns true if the two numbers have the same modulo.
   - There should be a `modulo` function which returns the modulo of the number.
   - There should be a `shiftDown` function which is given a `ModInt` number and a new modulo, `m`, and if `m` divides the number's modulo "downgrades" the number so that it is now modulo `m`. For example the number "5 modulo 8"

when shifted down to the modulo 4 would become “1 modulo 4” , as would the number “1 modulo 8”.

6. Implement a module for a “D&D dice”. A D&D die is specified informally with an expression like 2d8+3. This means “we roll two 8-sided dice, add the results, then add 3 to the final”. Implement a module for such dice, where the data type used is `data Die = D Int Int Int`, with the three integers representing in order the number of dice (2 in our example), the number of sides (8 in our example) and the extra added value at the end.

   - There should be a way to build such a die from the 3 components as well as functions `d6`, `d10`, `d20` which create the standard dice 1d6, 1d10 and 1d20 respectively.
   - There should be a `roll :: RandomGen g => g -> Die -> (Int, g)` function that rolls the die using the provided random number generator, which it updates and returns as the random number generation process describes.
   - There should be `minValue :: Die -> Int` and `maxValue :: Die -> Int` functions that return the smallest and largest possible value for such a roll.
   - The type should be an instance of `Eq` and `Show`, with 2 dice being equal if all their parameters match, and with `show die` producing the kind of 2d8+3 output described above.

7. Be able to implement various functions related to the `Maybe` type.

8. Be able to implement the various functions for trees described in the “recursive types” notes.

9. Have a basic understanding of the State type and how to use it to remember and update state data.

10. Be able to define and use the `fmap` function that `Functor` provides, and to demonstrate its behavior and definition for `Maybe`, lists, and IO.

11. Be able to define the basic functions that are part of `Applicative`, namely `pure` and `<*>`, and implement them for `Maybe`, lists and `IO`.

12. Define the main `Monad` operators (`return`, `>>=`) and explain the difference between Monad and Applicative in terms of what they allow us to do. Implement the monad operators for `Maybe` and lists.

13. Implement the following functions, which work with a monad `m`:

```haskell
sequence  :: Monad m => [m a] -> m [a]
sequence_  :: Monad m => [m a] -> m ()
join  :: Monad m => m (m a) -> m a
```

14. Write a program that uses two threads, each printing its own character on the screen (say A and B) then using the `Random` module to pause for a random amount of time up to 1 second, before printing the character again, going on forever (so each thread is a loop).

15. Write a program that behaves like 14 above, except that the two threads use an MVar to communicate and keep track of how many characters have been printed so that they terminate after a total of 10 characters combined.