Final Study Guide

You should read all the notes we have discussed so far (up to but NOT including list comprehensions), 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).

1. Describe the syntax of list comprehensions, identifying what can go into each location, as well as what the different parts that go to the right side are. Provide at least two examples.
2. Use list comprehensions to:
   a. compute all possible results of applying any of the four basic arithmetic operations to numbers from a given list `xs` (possibly using the same number on both sides).
   b. implement the `map` function and the `filter` function.
   c. implement a function that given a list of lists `xss` returns the concatenation of all elements into a single list.
   d. compute all the divisors of a number `n`.
   e. find all strings in a list of strings `ss` whose length is at least 5.
3. Explain why we cannot really implement the folding functions `foldr`, `foldl` using list comprehensions.
4. Define the "function application" operator/function `($)`, specifying its type as well as its definition. Demonstrate with an example why one might want to use this operator instead of the normal function application (i.e. something like "`f $ x`" instead of "`f x`").
5. Define the "function composition" operator/function that Haskell has, `(.)`, specifying its type as well as its definition.
6. Using point-free notation, function composition and operator sections where needed, write functions that:
   a. implement the mathematical function $5x + 2$.
   b. test if a number is odd by essentially performing the check ``x `mod` 2 == 1``.
   c. given a list of numbers compute the sum of only the odd terms in the list (assume the existence of a function `isOdd` that given a number returns whether the number is odd, and a function `sum` that given a list of numbers returns their sum).
7. Define what *difference lists* are, what their type is and how they are to be understood. How does a normal list turn into a difference list? How do we turn a "difference list" back to a list? How do we append a difference list to another difference list? What problem do difference lists solve?

8. Why do we need a special type (the IO type) to do system input/output operations like printing? Why can't a normal function of type say `f :: Int -> Int` also do some IO operations?

9. Write and explain the types of the IO primitives `getChar` and `putChar`.

10. Describe what types the following IO actions or functions involving IO actions should have:

    a. An action that asks the user for their first and last name, then produces a pair of strings with those names.

    b. A function that given an integer `n` produces an action that asks the user to type in that many numbers and produces a list of those numbers.

    c. A function that takes in a list of strings then prints each string on its own row.

11. Using the primitives `getChar`, `putChar` and `return`, as needed, write the following functions (also specify their type):

    a. a function `putStr` that given a string of characters prints that string.

    b. a function `getLine` that reads characters until a newline is encountered, then returns the resulting string (excluding the newline).

    c. a function `confirm` that expects the user to type `y` or `n`. If the user types one of those then the action produces the booleans `True/False` respectively, otherwise it keeps reading more characters from the user.

    d. a function `(<$>) :: (a -> b) -> IO a -> IO b` that is given a function and an action that produces an `a` value, and returns a function that produces a `b` value by performing the given action then applying the function to the resulting value.

12. Write the recursive type for a binary tree that holds `a` values at its nodes.

    a. Write a function `treeSum` that given such a binary tree adds up the values at the nodes. Make sure to correctly write the type for `treeSum`, including the class constraint that the tree values can be added up.

    b. Write a function `insert` which assuming that the content value type `a` implements the `Ord` class, takes in a new value and a tree that is a *binary search tree* and updates the tree with this new value inserted at the appropriate place. Make sure to correctly write the type for such a tree.

    c. Write a function `maybeMin` that, assuming the tree is a *binary search tree*, locates the smallest value in the tree. It should return a `Maybe a` value, with `Nothing` if the starting tree was empty. Make sure to get the function type correctly, including the suitable type-class constraint.

    d. Write a function `contains` that, given a value and a *binary search tree* determines if the tree contains the value. Make sure to get the function type correctly, including the suitable type-class constraint.

13. Write down the type for foldr and explain what each of the inputs does and what the function does overall. Also write down an implementation for foldr.
14. Show how sum and map can be implemented via foldr.
15. Specify the types of the functions any and all, and implement them via foldr.
16. Explain the idea of *information hiding* in programming, and its importance.
17. Explain what *modules* are, and why they are important in programming.
18. Explain the difference between *opaque* and *transparent* data types, and what the advantages and disadvantages of using each of these is.
19. Describe what functions (including their types) need to be implemented for the Functor type class and the Applicative type class, and demonstrate the specific instance implementations of these for the type [a] and the type Maybe a.