Homework 3: Haskell and Functional Programming

Due: 1(a), 1(b), 3, 7(a) on Friday, November 6, 1998, 7(b), 15, 16 on Tuesday, November 10, 1998, all other problems on Tuesday, November 17, 1998.

In this homework you will recall (or learn about) basic techniques of recursive programming over various types of data, and learn about abstracting from patterns, higher-order functions, currying, and infinite data. Many of the problems below exhibit polymorphism. Finally, the problems as a whole illustrate how functional languages work without commands or variables.

Since the purpose of this homework is to ensure skills in functional programming, this is an individual homework. That is, for this homework, you are to do the work on your own, not in groups.

For all Haskell programs, you must run your code with Haskell 1.4 (for example, using `hugs`). You must also provide evidence that your program is correct (for example, test cases). Hand in a printout of your code and your testing, for questions that require code.

Be sure to clearly label what problem each function solves with a comment.

Read Thompson's book, chapters 1–2, 4, 6–7, and 10. Read Thompson's chapter 8–9 as you find you need it to work the problems, or as we discuss it in class.

Some of the problems build on each other. Don't hesitate to contact the staff if you are stuck at some point.

Acknowledgement: many of these problems are due to John Hughes.

0. (suggested practice) Read a tutorial on the concepts of functional programming. For example, chapter 13 in Watt's *Programming Language Concepts and Paradigms*. Or look at some of the books recommended in the "Introduction to the Literature" handout.

   Also take a look at the functions programmed on lists and available to you in the Haskell prelude, which is in the file `/usr/unsup/hugs/lib/Prelude.hs`. You can't write good Haskell programs without some familiarity with this prelude, although the notation may be confusing at first.

1. (30 points) Write a function

```
> delete_all :: (Eq a) => a -> ([a] -> [a])
```

   that takes an item (of a type that is an instance of the `Eq` class) and a list, and returns a list just like the argument list, but with the each occurrence of the item (if any) removed. For example.

```
delete_all 3 ([]::[Int]) = [] :: [Int]
delete_all 1 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [2, 3, 2, 2, 3, 2]
delete_all 4 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [1, 2, 3, 2, 1, 2, 3, 2, 1]
delete_all 3 [1, 2, 3] = [1, 2]
```

   Do this (a) by writing out the recursion yourself, (b) using a list comprehension, and (c) by just using functions in the Haskell Prelude.

2. (suggested practice) Write a function

```
delete_second :: (Eq a) => a -> ([a] -> [a])
```

that takes an item (of a type that has an == function defined for it) and a list, and returns a list just like the argument list, but with the second occurrence of the item (if any) removed. For example.

```
delete_second 3 ([]::[Int]) = [] :: [Int]
delete_second 1 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [1, 2, 3, 2, 2, 3, 2, 1]
delete_second 4 [1, 2, 3, 2, 1, 2, 3, 2, 1] = [1, 2, 3, 2, 1, 2, 3, 2, 1]
delete_second 3 [1, 2, 3] = [1, 2, 3]
```

Do this both (a) by writing out the recursion yourself, and (b) by just using functions in the Haskell Prelude.

Hint: for part (a), you may need a helping function.

3. (10 points) Write a function

```
merge :: (Ord a) => [[a]] -> [a]
```

that takes a finite list of sorted finite lists and merges them into a single sorted list. A "sorted list" means a list sorted in increasing order (using <); you may assume that the sorted lists are finite. For example

```
merge ([[]]::[[Int]]) = [] :: [Int]
merge [[1, 2, 3]] = [1, 2, 3]
merge [[1, 3, 5, 7],  [2, 4, 6]] = [1, 2, 3, 4, 5, 6, 7]
merge [[1,3,5,7], [2,4,6], [3,5,9,10,11,12]] = [1,2,3,3,4,5,5,6,7,9,10,11,12]
take 8 (merge [[1, 3, 5, 7], [1,2,3,4,5,6,7,8]]) = [1, 1, 2, 3, 3, 4, 5, 5]
```

(For 30 points extra credit, make your solution work when the sorted lists are not necessarily finite; you can still assume that there are a finite number of sorted lists.)

4. (5 points) Redefine the ++ operator on lists using `foldr`, by completing the following module by adding arguments to `foldr`. You'll find the code for `foldr` in the Haskell prelude, which is in `/usr/unsup/hugs/lib/Prelude.hs`. (Hint: you can pass the ":" constructor as a function by writing `(:)`.)

```
module MyAppend where
import Prelude hiding ((++))
(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr ____________________________
```

5. (10 points) Again using `foldr`, define the function

```
> doubleAll :: [Integer] -> [Integer]
```

that takes a list of Integers, and returns a list with each of its elements doubled. You must use `foldr` in a way similar to the previous problem. (Hint: use a `where` to define the function you need to pass to `foldr`. You might want to use function composition, written with an infix dot `(.)` in Haskell.) The following are examples.

```
doubleAll [] = []
doubleAll [8,10] = [16,20]
doubleAll [1, 2 .. 500] = [2, 4 .. 1000]
```

6. (15 points) (a) Define the `map` functional using `foldr`. As part of your testing, use `map` to (a) define `doubleAll`, and (b) to add 1 to all the elements of a list of Integers. (Hint: import the Prelude hiding `map` in the module for this answer; see problem 4 above.)

7. Consider the following data type for trees, which represents a Tree of type `a` as a Node, which contains an item of type `a` and a list of Trees of type `a`.

```
> data Tree a = Node a [Tree a]
```

   (a) (10 points) Define a function `sumTree`

```
> sumTree :: Tree Integer -> Integer
```

   which adds together all the Integers in a Tree of Integers. For example,

```
sumTree (Node 4 []) = 4
sumTree (Node 3 [Node 4 [], Node 7 []]) = 14
sumTree (Node 10 [Node 3 [Node 4 [], Node 7 []],
                  Node 10 [Node 20 [], Node 30 [], Node 40 []]]) = 124
```

   (b) (10 points) Define a function `preorderTree`

```
> preorderTree :: Tree a -> [a]
```

   which takes a Tree and returns a list of the elements in its node in a preorder traversal. For example,

```
preorderTree (Node True []) = [True]
preorderTree (Node 5 [Node 6 [], Node 7 []]) = [5, 6, 7]
preorderTree (Node 10 [Node 3 [Node 4 [], Node 7 []],
                       Node 10 [Node 20 [], Node 30 [], Node 40 []]])
     = [10, 3, 4, 7, 10, 20, 30, 40]
```

   (c) (15 points) Give a Haskell `instance` declaration that makes `Tree` an instance of the type class `Functor`. (See the Prelude for the definition of the `Functor` class.) Your code should start as follows.

```
instance Functor Tree where
    map f ...
```

   (d) (30 points) By generalizing your answers to the above problems, define a function `foldTree`

```
> foldTree :: (a -> b -> c) -> (c -> b -> b) -> b -> Tree a -> c
```

   that is analogous to `foldr` for lists. This should take a function to replace the Node constructor, one to replace the `(:)` constructor for lists, and a value to replace the empty list. You should, for example, be able to define `sumTree`, `preorderTree`, and the Functor instance for `map` on Trees as follows.

```
> sumTree = foldTree (+) (+) 0
> preorderTree = foldTree (:) (++) []
> instance Functor Tree where
>    map f = foldTree (Node . f) (:) []
```

8. The following relate to modularization of numeric code using functional techniques and lazy evaluation. In particular, we will explore the Newton-Raphson algorithm. This algorithm computes better and better approximations to the square root of a number `n` from a previous approximation `x` by using the following function.

```
> next :: (Real a, Fractional a) => a -> a -> a
> next n x = (x + n / x) / 2
```

(a) (10 points) Using the `iterate` function in the Haskell Prelude, write a function

```
> approximations :: (Real a, Fractional a) => a -> a -> [a]
```

such that `approximations n a0` returns the infinite list of approximations to the square root of `n`, starting with `a0`. For example,

```
approximations 1.0 1.0 = [1.0, 1.0 ..]
take 5 (approximations 2.0 1.0) = [1.0, 1.5, 1.41667, 1.41422, 1.41421]
take 5 (approximations 64.0 1.0) = [1.0, 32.5, 17.2346, 10.474, 8.29219]
```

(b) (20 points) Define a function `within`

```
> within :: (Ord a, Num a) => a -> [a] -> a
```

that takes a tolerance, that is, a number `epsilon`, and an infinite list of numbers, and looks down the list to find two consecutive numbers in the list that differ by no more than `epsilon`; it returns the second of these. (It might never return if there is no such pair of consecutive elements.) For example,

```
within 1.0 [1.0 ..] = 2.0
within 0.5 ([1.0, 32.5, 17.2346, 10.474, 8.29219, 8.00515] ++ [8.0, 8.0 ..])
      = 8.00515
```

(c) (10 points) Using the two pieces above, make a function `squareRoot`

```
> squareRoot :: (Real a, Fractional a) => a -> a -> a -> a
```

that takes an initial guess, a tolerance `epsilon`, and a number, `n` and returns an approximation to the square root of `n` that is within `epsilon`. For example,

```
squareRoot 1.0 0.0000001 2.0 = 1.41421
squareRoot 1.0 0.0000001 64.0 = 8.0
```

(d) (15 points) Write a function `relativeSquareRoot`

```
> relativeSquareRoot :: (Real a, Fractional a) => a -> a -> a -> a
```

which keeps iterating until the ratio of the difference between the last and the previous approximation to the last approximation approaches 0, instead of waiting for the differences between the approximations themselves to approach zero. (This is equivalent to iterating until the ratio of the last two approximations approaches 1.) This is better for square roots of very large numbers, and for square roots of very small numbers. The function `relativeSquareRoot` takes an initial approximation, a tolerance `epsilon` (for

how closely the ratio between the last two approximations must approach 1), and the number `n`. (Hint, define a function `relative` that plays the role of `within`; use absolute values.) For example:

```
relativeSquareRoot 1.0 0.1e-5 9.0e+10 = 3.0e+5
relativeSquareRoot 1.0 0.1e-5 9.0e-40 = 3.0e-20
```

9. (15 points) You may recall that the derivative of a function `f` at a point `x` can be approximated by the following function.

```
> easydiff :: (Real a, Fractional a) => (a -> a) -> a -> a -> a
> easydiff f x delta = (f(x+delta) - f(x)) / delta
```

Good approximations are given by small values of `delta`, but if `delta` is too small, then rounding errors may swamp the result. One way to choose `delta` is to compute a sequence of approximations, starting with a reasonably large one. If `(within epsilon)` is used to select the first approximation that is accurate enough, this can reduce the risk of a rounding error affecting the result. Write function

```
> diffApproxims :: (Real a, Fractional a) => a -> (a -> a) -> a -> [a]
```

that takes an initial value for `delta`, and the function `f`, and a point `x`, and returns an infinite list of approximations to the derivative of `f` at `x`, where at each step, the current `delta` is halved. For example

```
take 9 (diffApproxims 500.0 (\x -> x*x) 20)
    = [540.0, 290.0, 165.0, 102.5, 71.25, 55.625, 47.8125, 43.9062, 41.9531]
take 8 (diffApproxims 100.0 (\x -> x*x*x) 10)
    = [13300.0, 4300.0, 1675.0, 831.25, 526.562, 403.516, 349.316, 324.048]
```

10. (15 points) Write a function

```
> differentiate :: (Real a, Fractional a) => a -> a -> (a -> a) -> a -> a
```

that takes a tolerance, `epsilon`, an initial value for `delta`, and the function `f`, and a point `x`, and returns an approximation to the derivative of `f` at `x`. For example.

```
differentiate 0.1e-6 500.0 (\x -> x*x) 20 = 40.0
differentiate 0.1e-6 100.0 (\x -> x*x*x) 10 = 300
```

11. (30 points; extra credit) Write a function in Haskell to do numerical integration, using the ideas above.

12. (15 points) Write a function

```
> compose :: [(a -> a)] -> (a -> a)
```

that takes a list of functions, and returns a function which is their composition. For example.

```
compose [] [1, 2, 3] = [1, 2, 3]
compose [(\ x -> x + 1), (\ x -> x + 2)] 4 = 7
compose [tail, tail, tail] [1, 2, 3, 4, 5] = [4, 5]
compose [(\ x -> 3 : x), (\ y -> 4 : y)] [] = 3 : (4 : [])
```

Hint: note that `compose []` is the identity function.

13. (extra credit) Consider the following type as a representation of binary relations.

```
> type BinaryRel a b = [(a, b)]
```

(a) (10 points, extra credit) Write a function

```
> isFunction :: (Eq a, Eq b) => (BinaryRel a b) -> Bool
```

that returns True just when its argument satisfies the standard definition of a function; that is, `isFunction r` is True just when for each pair (x, y) in the list `r`, there is no pair (x, z) in `r` such that $y \neq z$.

(b) (10 points, extra credit) Write a function

```
> brelCompose :: (Eq a, Eq b, Eq c) =>
>                (BinaryRel a b) -> (BinaryRel b c) -> (BinaryRel a c)
```

that returns the relational composition of its arguments. That is, a pair (x, y) is in the result if and only if there is a pair (x, z) in the first relation argument of the pair of arguments, and a pair (z, y) in the second argument. For example,

```
brelCompose ([]::[(Int, Int)]) [(2,'b'), (3, 'c')] = []
brelCompose []::[(Int, Int)] ([]::[(Int, Int)]) = []
brelCompose [(1,2),(2,3)]  [(2,'b'), (3, 'c')] = [(1, 'b'), (2, 'c')]
brelCompose [(1,2),(1,3)]  [(2,'b'), (3, 'c')] = [(1, 'b'), (1, 'c')]
brelCompose [(1,3), (2,3)] [(3,'b'), (3,'c')]
            = [(1,'b'), (1,'c'), (2,'b'), (2,'c')]
```

14. (30 points) A bag (or multi-set) can be described by a "characteristic function" (whose range is the integers) that determines how many times an element occurs in the bag. For example, the function ϕ such that $\phi($"`coke`"$) = \phi($"`pepsi`"$) = 6$ and for all other arguments x, $\phi(x) = 0$ is the characteristic function for a set containing a 6 `"coke"`, 6 `"pepsi"` and nothing else. Allowing the user to construct a bag from a characteristic function gives one the power to construct bags that may "contain" an infinite number of elements (such as the bag of all prime numbers).

Let the polymorphic type constructor `Bag` be some polymorphic type that you decide on (you can declare this with something like the following).

```
type Bag a = ...
-- or perhaps something like
data Bag a = ...
```

Hint: think about using a function type.

The operations on bags are described informally as follows.

(a) The function

```
bagSuchThat :: (a -> Integer) -> (Bag a)
```

takes a characteristic function, f and returns a bag such that each value x (of appropriate type) occurs exactly fx times.

(b) The function

```
addBag :: Bag a -> Bag a -> Bag a
```

takes two bags, with characteristic functions f and g, and returns a bag such that each value x (of appropriate type) occurs exactly $(fx) + (gx)$ times.

(c) The function

```
countBag :: Bag a -> a -> Integer
```

returns the number of times the item (of type `a`) occurs in the bag.

As examples, consider the following.

```
countBag (bagSuchThat (\ x -> if x == "coke" then 6 else 0)) "coke" = 6
countBag (bagSuchThat (\ x -> if x == "coke" then 6 else 0)) "pepsi" = 0
countBag (addBag (bagSuchThat (\ x -> if x == "coke" then 6 else 0))
                 (bagSuchThat (\ x -> if x == "pepsi" then 6 else 0)))
         "pepsi" = 6
countBag (addBag (bagSuchThat(\ x -> if x == "pepsi" then 3 else 0))
                 (bagSuchThat(\ x -> if x == "pepsi" then 6 else 0)))
         "pepsi" = 9
```

Note (hint, hint) that the following equations must hold, for all `f`, `g`, and `x` of appropriate types.

```
countBag(addBag(bagSuchThat f, bagSuchThat g), x) = (f x) + (g x)
countBag(bagSuchThat f, x)) = f x
```

15. (25 points) Consider the following data definitions.

```
> data Exp = BoolLit Bool | IntLit Integer | CharLit Char
>          | Sub Exp Exp
>          | Equal Exp Exp
>          | If Exp Exp Exp
> data OType = OBoolean | OInt | OChar | OWrong
>            deriving Eq
```

Write a function

```
typeOf :: Exp -> OType
```

that takes an `Exp` and returns its `OType`. For example.

```
typeOf (Equal (IntLit 3) (IntLit 4)) = OBoolean
typeOf (Sub (IntLit 3) (IntLit 4)) = OInt
typeOf (Sub (CharLit 'a') (IntLit 4)) = OWrong
typeOf (If (BoolLit True) (IntLit 4) (IntLit 5)) = OInt
typeOf (If (BoolLit True) (IntLit 4) (BoolLit True)) = OWrong
typeOf (If (IntLit 3) (IntLit 4) (IntLit 5)) = OWrong
```

Your program should incorporate a reasonable notion of what the exact type rules are. (Exactly what "reasonable" is left up to you; explain any decisions you feel the need to make.)

16. (50 points total) Read a published research article in a journal or conference proceedings that is related to functional programming languages (such as Haskell, SML, etc.). By a published research article, I mean an article that is not in a trade journal (e.g., it has references at the end) and that is not on the web, but in the library. (It's okay to get a copy of a published article from the web, however.) Articles about functional programming techniques are acceptable, although I'd prefer an article about functional programming *languages*. If the article seems too technical, try to find another one. I am happy to suggest articles for you, or evaluate your choices for suitability if you are in doubt.

    Write a short (1 or 2 page maximum) review of the article, stating:

    - (10 points) what the problem was that the article was claiming to solve, (or if the article is a survey article, then the scope of the survey),
    - (20 points) the main points made in the article and what you learned from it,
    - (20 points) what contribution it make vs. any related work mentioned in the article (or if the article was a survey article, your evaluation of its completeness and clarity of thought and expression, and a statement of what you learned from it).

    Please be nice to the class by not checking out a proceedings or journal volume from the library, but only making a copy of a relevant article, or reading it in the library.