Homework 1: Functional Programming in Haskell

Due: problems 1–3, Tuesday, Jan 30; problems 4–12, Tuesday, Feb 6; problems 13–20, Tuesday, Feb 13.

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 a Haskell interpreter (such as `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. Some questions just require an answer in English.

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

Read Davie's book, chapter 1-3. Read Davie's chapter 4 as you find you need it to work the problems, or as we discuss it in class.

1. (6 points) Using only numerals, `:`, and `[]`, build the following lists.

   (a) `[1,2]`

   (b) `[[1,2],[3],[]]`

   You can either do this entirely on the command line of `hugs` or you can make a file with a definition such as

   ```
   answer1a = -- your code for part a goes here
   answer1b = -- your code for part b goes here
   ```

2. (5 points) Write a function

   ```
   butFst :: (a, b, c) -> (b, c)
   ```

   that takes a 3-tuple and returns a 2-tuple consisting of the last 2 elements of its argument. For example.

   ```
   butFst (1,2,3) = (2,3)
   butFst ('a','b','c') = ('b','c')
   ```

3. (8 points) Write a function

   ```
   tails :: [a] -> [[a]]
   ```

   that takes a list and returns a list of lists consisting of the list, and its tails. For example.

   ```
   tails [] = []
   tails [1,2,3] = [[1,2,3], [2,3], [3], []]
   tails [2,3] = [[2,3], [3], []]
   take 4 (tails [1 ..]) = [[1 ..], [2 ..], [3 ..], [4 ..]]
   ```

   Note: don't try testing the last example by just printing it out!

4. (10 points) 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 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]
delete_second 3 []::[Integer] = []
```

Hint, you may need a helping function.

5. (suggested practice) Do problem 3.1(c) in Davie's book, page 55. [expression value] Think about it first, then use the interpreter to find the answer.

6. (10 points) Do problem 3.2 in Davie's book, page 56. For part (a), your function should be

```
copy :: [a] -> [a]
```

7. (suggested practice) Do problems 3.3 and 3.4 in Davie's book, page 56. For problem 4, use the following names and types.

```
my_take :: Integer -> [a] -> [a]
my_drop :: Integer -> [a] -> [a]
```

and don't use the Haskell built-in primitives `take` and `drop`.

8. (10 points) Do problem 3.7 in Davie's book, page 56. [for]

9. (suggested practice) Do problems 3.8, 3.9, and 3.10 in Davie's book, page 56.

10. (15 points extra credit) Write answers to at least 3 of the above problems using list comprehension notation (ZF notation).

11. (15 points) Do problem 3.17 on Davie's p. 57. [association lists]

12. (10 points) Haskell does not have a built-in type for records. Do you think that is a serious problem in the language? Briefly explain your answer.

13. (20 points) Consider the following functions.

```
> member :: (Eq a) => a -> ([a] -> Bool)
> member x [] = False
> member x (y:ys) = x == y || member x ys

> apply_to_all :: (a -> b) -> ([a] -> [b])
> apply_to_all f [] = []
> apply_to_all f (y:ys) = (f y):(apply_to_all f ys)

> add_all :: (Num a) => [a] -> a
> add_all [] = 0
> add_all (y:ys) = y + (add_all ys)

> mult_all :: (Num a) => [a] -> a
> mult_all [] = 1
> mult_all (y:ys) = y * (mult_all ys)
```

Write a function,

```
flat_recur :: b -> (((a, b) -> b) -> ([a] -> b))
```

such that one can define the above functions as follows

```
member item =  flat_recur False (\ (y,ys_ans) -> item == y || ys_ans)
apply_to_all f = flat_recur []  (\ (y,ys_ans) -> (f y):ys_ans)
add_all :: (Num a) => [a] -> a
add_all = flat_recur 0 (\ (y,ys_ans) -> y + ys_ans)
mult_all :: (Num a) => [a] -> a
prod_all = flat_recur 1 (\ (y,ys_ans) -> y * ys_ans)
```

Hint: think about what parts of the above functions are similar, and what are different. Put the similar parts into `flat_recur`.

14. (10 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.

15. Consider the following type as a representation of binary relations.

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

(a) (10 points) 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) 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 ([(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 ([]::[(Integer, Integer)], [(2,'b'), (3, 'c')]) = []
brelCompose([]::[(Integer, Integer)], []::[(Integer, Integer)]) = []
```

16. (40 points; extra credit) Write Haskell definitions for all the operations of the relational algebra as in the theory of relational databases for binary relations.

17. (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 $\phi(\text{"}\texttt{coke}\text{"}) = \phi(\text{"}\texttt{pepsi}\text{"}) = 6$ 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).

    The elements of a bag must be of a type in the class `Eq`; that is, they must have a function `==` defined for them. 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 a pair of 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
```

18. (suggested practice) Do problems 4.1, 4.2, and 4.3 on page 76 of Davie's book.

19. (25 points; extra credit) Do problems 4.5 on page 76 of Davie's book.

20. (25 points) Consider the following data definitions.

```
> data Exp = BoolLit Bool | IntLit Integer
>          | Sub Exp Exp
>          | Equal Exp Exp
>          | If Exp Exp Exp
> data OType = OBoolean | OInt | 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 (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.