Homework 3: Haskell and Functional Programming

Due:

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 1.3 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, 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.

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.

1. (a) (8 points) Write a function

   ```
   inits :: [a] -> [[a]]
   ```

   that takes a list and returns a list of lists consisting of the list, and its initial segments. The following are examples. (The first example gives an explicit type to one expression, so that the empty list will print correctly.)

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

   (b) Does your program work when passed an infinite list as an argument? Briefly explain why it does or does not.

2. (10 points) Write a function

   ```
   delete_all :: (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 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]
   ```

3. (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 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]
```

Hint: you may need a helping function.

4. (suggested practice) Write your own versions of the Haskell built-in primitives `take` and `drop` (without using them, of course). Use the following names and types.

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

5. (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.)

6. (15 points extra credit) Write answers to at least 3 of the above problems using list comprehension notation (ZF notation).

7. (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`.

8. (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.

9. (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')]
   ```

10. (30 points) A set can be described by a "characteristic predicate" (whose range is the booleans) that determines whether an element occurs in the set. For example, the predicate ϕ such that $\phi(3) = \phi(4) = True$ and for all other arguments x, $\phi(x) = False$ is the characteristic predicate for the set $\{3, 4\}$. Allowing the user to construct a set from a characteristic predicate gives one the power to construct sets that may "contain" an infinite number of elements (such as the set of all prime numbers).

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

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

Hint: think about using a function type.

The operations on sets are described informally as follows.

(a) The function

`singletonSet :: (Eq a) => a -> Set a`

is defined for elements of a type in the class `Eq`; that is, for a type that has elements with a predicate `==` defined. This function takes a single element x and returns the set $\{x\}$.

(b) The constant

`emptySet :: Set a`

represents the empty set, {}.

(c) The function

`setSuchThat :: (a -> Bool) -> (Set a)`

takes a characteristic predicate, f and returns the set $\{x \mid f(x) = True\}$.

(d) The function

`unionSet :: (Set a, Set a) -> Set a`

takes a pair of sets, with characteristic predicates f and g, and returns their union, $\{x \mid f(x) = True$ or $g(x) = True\}$.

(e) The function

`memberSet :: (Set a, a) -> Bool`

returns True just when the item (of type `a`) is in the set.

As examples, consider the following.

```
memberSet (singletonSet 3, 3) = True
memberSet (singletonSet 3, 4) = False
memberSet (emptySet, 3) = False
memberSet(setSuchThat(\ x -> x == "coke"), "coke") = True
memberSet(setSuchThat(\ x -> x == "coke"), "pepsi") = False
memberSet(setSuchThat(\ x -> True), 3.14159) = True
memberSet(unionSet(singletonSet 3, singletonSet 4), 3) = True
memberSet(unionSet(singletonSet 3, singletonSet 4), 4) = True
memberSet(unionSet(singletonSet 3, singletonSet 4), 5) = False
```

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

```
memberSet(unionSet(setSuchThat f, setSuchThat g), x) = (f x) || (g x)
memberSet(setSuchThat f, x)) = f x
```

11. (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 is left up to you; explain any decisions you feel the need to make.)

12. (40 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:

    - (5 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),
    - (10 points) the main points made in the article and what you learned from it,
    - (10 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.