Solutions Functional Programming – December 2nd 2015

1. **Types** (5× 2=10 points)
   (a) What is the type of the following Haskell expression?

   ```
   [('4'=='4', 2)]
   ```

   ```
   [(Bool, Int)]
   ```

   (b) What is the most general type of the Haskell function `f`?

   ```
   f = map length
   ```

   ```
   [[a]] -> [Int]
   ```

   (c) What is the most general type of the Haskell function `g`?

   ```
   g = foldr (+) 0
   ```

   ```
   Num a => [a] -> a
   ```

   (d) What is the type of the the following Haskell expression?

   ```
   (\f -> (\g -> (\x -> f (g x))))
   ```

   ```
   (a -> c) -> (b -> a) -> b -> c
   ```

   (e) What is the type of the following Haskell function `h`?

   ```
   h = head.tail.fst
   ```

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

2. **Programming in Haskell** (10 points) Write a Haskell function `perms` (including its type) that accepts as its argument a list, and outputs the list of lists that are the permutations of the input list. For example, `perms "abc"` should return the list `["abc","acb","bac","bca","cab","cba"]`. Note that the order of the elements in the output list is not important.

   ```
   perms :: [a] -> [[a]]
   perms [] = [[]]
   perms xs = [(x:ys) | x <- xs, ys <- perms (filter (/= x) xs)]
   ```

3. **List comprehensions** ($3 \times 5 = 15$ points)

- Implement the function `filter` (including its type, see front page) as a list comprehension.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

- A *Pythagorean triad* is a triple of positive integers (a, b, c) such that $a^2 + b^2 = c^2$. Write a Haskell function `pytriads n` that returns a list of all Pythagorean triplets such that $0 < a \leq b \leq c \leq n$. This list must be ordered based on the value of `c`.
For example, `pytriads 18` should return `[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17)]`.
The implementation of `pytriads` must be a list comprehension.

```
pytriads n = [(a,b,c) | c <- [1..n], b <- [1..c], a <- [1..b] , a^2 + b^2 == c^2]
```

- An integer n is called a *perfect number* if it is equal to the sum of *all* its proper divisors (excluding n itself). For example, 6 is a perfect number, since 6=1+2+3. An integer n is called a *semi-perfect number* if it is equal to the sum of a *subset* of its proper divisors (excluding n itself). An example of a semi-perfect number that is not a perfect number is 12, since the proper divisors of 12 are 1, 2, 3, 4, and 6 which add up to 16, but 2+4+6=12.

  Write a Haskell function `spf n` that computes the list of all semi-perfect numbers in the range $[1..\mathtt{n}]$. The implementation must be a list comprehension (which may use helper function if needed). For example, `spf 50` should yield `[6,12,18,20,24,28,30,36,40,42,48]`.

```
divisors x = [y | y <- [1..x `div` 2], x `mod` y == 0]

powerset [] = [[]]
powerset (head:tail) = acc ++ map (head:) acc where acc = powerset tail

spf n = [x | x <- [2..n], any (== x) [sum xs | xs <- powerset (divisors x)]]
```

4. **infinite lists** ($3 \times 5 = 15$ points)

- Give a *recursive* definition of the infinite list `tf` of non-empty lists of alternating boolean values that end with the value `True`. So, `take 4 tf` yields
`[[True], [False, True], [True,False,True], [False,True,False,True]]`

```
tf = [True] : [ (not(head bs)):bs | bs <- tf]
```

- Give a *recursive* definition of the list `bits` of all binary (character) strings that end with a zero, and do not contain two consecutive ones. The order of the strings may be chosen arbitrarily.
For example, `take 8 bits` may yield `["0","10","00","100","010","1010","000","1000"]`.

```
bits = "0":"1":[prefix ++ xs | xs <- bits, prefix <- ["0", "10"]]
```

- Give a definition of the function `multiples` that takes as its argument a finite list of positive integers, and outputs the infinite list of all multiples of the input numbers. Note that the output list must be generated in increasing order, and should not contain any duplicates.
For example, `take 14 (multiples [3,5,7])` yields `[0,3,5,6,7,9,10,12,14,15,18,20,21,24]`.

```
multiples xs = foldr merge [] [[0,x..] | x <- xs]
  where merge xs [] = xs
        merge (x:xs) (y:ys)
          | x < y     = x:(merge xs (y:ys))
          | x > y     = y:(merge (x:xs) ys)
          | otherwise = x:(merge xs ys)
```

5. (15 points) The abstract data type `Bag tp` implements a simple data type for the storage of elements of the type `tp`. A bag is also known as a *multiset*, since elements may occur multiple times. For example, if we insert some element twice in a bag, and remove one of them afterwards, then one element still remains in the bag (in contrast with a standard set).

   Implement a `module Bag` such that the concrete implementation of the type `Bag` is hidden from the user.

   The following operations on the data type `Bag` must be implemented:

   - `empty`: returns an empty bag.
   - `isEmpty`: returns `True` for an empty bag, otherwise `False`.
   - `insert`: returns the bag that is the result of inserting an element.
   - `remove`: returns the bag that is obtained by removing an element.
   - `cardinality`: returns the number of occurrences of an element in the bag.
   - `union`: returns the union of two bags.
   - `intersection`: returns the intersection of two bags.

```
module Bag(Bag,empty,isEmpty,insert,remove,cardinality,union,intersection) where

data Bag a = B [a]

empty :: Bag a
empty = B []

isEmpty :: Eq a => Bag a -> Bool
isEmpty (B xs) = (xs == [])

insert :: Eq a => a -> Bag a -> Bag a
insert x (B bag) = B (x:bag)

remove :: Eq a => a -> Bag a -> Bag a
remove x (B xs) = B (rm x xs)
  where rm x [] = []
        rm x (y:ys)
          | x == y    = ys
          | otherwise = y:(rm x ys)

cardinality :: Eq a => a -> Bag a -> Int
cardinality x (B bag) = length (filter (==x) bag)

union :: Bag a -> Bag a -> Bag a
union (B xs) (B ys) = B (xs ++ ys)

intersection :: Eq a => Bag a -> Bag a -> Bag a
intersection (B l) (B r) = B (isect l r [])
  where isect [] _ _ = []
        isect (x:xs) [] acc = isect xs acc []
        isect (x:xs) (y:ys) acc
          | x == y    = x:isect xs (acc++ys) []
          | otherwise = isect (x:xs) ys (y:acc)
```

6. **Proof on lists** (10 points) The definitions of the functions `filter`, and `(++)` are given on the front page of this exam. Prove that `filter p (xs ++ ys) = filter p xs ++ filter p ys` for all finite lists `xs` and `ys`.

```
base:   filter p ([] ++ ys)
      = -- def. ++
        filter p ys
      = -- def. ++
        [] ++ (filter p ys)
      = -- def. filter
        filter p [] ++ filter p ys

ind:    filter p ((x:xs) ++ ys)
      = -- def. ++
        filter p (x : (xs ++ ys))

      case 1) assume p x
      = -- def filter, p x is true
        x : filter (xs ++ ys)
      = -- ind. hypothesis
        x : (filter p xs ++ filter p ys)
      = -- def. ++
        (x : filter p xs) ++ filter p ys
      = -- def. filter, p x is true
        filter p (x:xs) ++ filter p ys

      case 2) assume p s is false
      = -- def filter, p x is false
        filter p (xs ++ ys)
      = -- ind. hypothesis
        filter p xs ++ filter p ys
      = -- def. filter, p x is false
        filter p (x:xs) ++ filter p ys
 QED.
```

7. **Proof on trees** (15 points) Given is the data type `BinTree`, and the functions `inorder` and `maptree`:

```
data BinTree a = Empty | Node a (BinTree a) (BinTree a)

inorder :: BinTree a -> [a]
inorder Empty = []
inorder (Node x l r) = inorder l ++ [x] ++ inorder r

maptree :: (a -> b) -> BinTree a -> BinTree b
maptree f Empty = Empty
maptree f (Node x l r) = Node (f x) (maptree f l) (maptree f r)
```

Prove for all finite trees `t`: `inorder(maptree f t) = map f (inorder t)`

```
base:   inorder(maptree f Empty)
      = -- def. maptree
        inorder Empty
      = - def. inorder
        []
      = -- def map
        map f []
      = -- def inorder
        map f (inorder Empty)

ind:    inorder(maptree f (Node x l r))
      = -- def. maptree
        inorder(Node (f x) (maptree f l) (maptree f r))
      = -- def. inorder
        inorder (maptree f l) ++ [f x] ++ inorder (maptree f r)
      = -- ind. hypothesis (twice)
        map f (inorder l) ++ [f x] ++ map f (inorder r)

        map f (inorder (Node x l r))
      = -- def. inorder
        map f (inorder l ++ [x] ++ inorder r)
      = -- use associativity ++; lemma map f (xs ++ ys) = map f xs ++ map f ys
        map f (inorder l) ++ map f [x] ++ map f (inorder r)
      = -- def. map.  [x] = x:[]
        map f (inorder l) ++ (f x):[] ++ map f (inorder r)
      = -- def :
        map f (inorder l) ++ [f x] ++ map f (inorder r)
   QED.

Proof lemma map f (xs ++ ys) = map f xs ++ map f ys:
Base:   map f ([] ++ ys) = map f ys = [] ++ map f ys = map f [] ++ map f ys

ind:    map f ((x:xs) ++ ys)
      = -- def. ++
        map f (x:(xs ++ ys))
      = -- def. map
        (f x):map f (xs ++ ys)
      = -- ind. hypothesis
        (f x):(map f xs ++ map f ys)
      = -- def ++
        ((f x):map f xs) ++ map f ys
      = -- def. map
        map f (x:xs) ++ map f ys
   QED.
```