Solutions Exam Functional Programming – Nov. 2nd 2015

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

   ```
   (1,'2',"3")
   ```

   Answer: `(Integer, Char, [Char])`

   (b) What is the most general type of the Haskell function `f = map not` ?

   Answer: `[Bool] -> [Bool]`

   (c) What is the most general type of the Haskell function `g = (\(a,b) -> a + b)` ?

   Answer: `Num a => (a, a) -> a`

   (d) What is the type of the standard Haskell operator `(.)` that is used for function composition?

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

   (e) What is the type of the following Haskell function `h = head.(:['a'])` ?

   Answer: `Char -> Char`

2. **Programming in Haskell** (10 points) Write a Haskell function `middleElement` (including its type) that computes the middle element of a list. The function should return `Nothing` in case there is no middle element, and `Just m` in case `m` is the middle element, so

   - `middleElement [1,2,3]` should return `Just 2`.
   - `middleElement [1,2]` should return `Nothing`.

   You are not allowed to use the to use the function `length` (nor are you allowed to implement it yourself).
   [Hint: A possible solution uses a recursive helper function `mid xs ys`, of which the first argument 'shrinks' faster than the second argument during the recursion. Define `middleElement xs = mid xs xs where mid = ....`]

   Answer:

   ```
   middleElement :: [a] -> Maybe a
   middleElement xs = mid xs xs
     where mid [] _ = Nothing
           mid [_] (y:_) = Just y
           mid (_:_:xs) (_:ys) = mid xs ys
   ```

3. **Programming using list comprehensions and higher order functions** (10 points)
   Counting sort is a well-known algorithm for sorting a list of small integers. In this problem, you may assume that the input is a list of digits (i.e. [0..9]). The algorithm consists of two passes. In the first part, a histogram of the input is computed: for each possible value, the number of occurrences of this value is computed. In the second pass, using the histogram from the first pass, the sorted output is produced.

   Give an implementation of counting sort that uses solely list comprehensions and higher order functions. The use of recursion is not allowed.

   Answer:

   ```
   histogram :: [Int] -> [(Int, Int)]
   histogram xs = [(i,length(filter (== i) xs))| i <- [0..9]]

   csort :: [Int] -> [Int]
   csort xs = foldr (++) [] [replicate n v | (v,n) <- histogram xs]
   ```

4. **List comprehensions** (3+3+4=10 points)

   - Write a function `palindromes` (including its type) that accepts as its input a list of strings, and returns a list of all strings which are palindromes that can be constructed by concatenating two strings from the input list.
     For example, `palindromes ["a", "ab", "abb", "ac", "ca"]` should return a list containing the strings `"aa"`,`"aca"`,`"aba"`,`"abba"`,`"aca"`,`"acca"`, and `"caac"` (in any order). The implementation must be a list comprehension.
     Answer:
     ```
     palindromes :: [String] -> [String]
     palindromes xss = [s1 ++ s2 | s1 <- xss, s2 <- xss, s1++s2 == reverse(s1++s2)]
     ```
   - Give a definition of the list `fun42` which contains 42 elements. These elements are functions of the type `Integer -> Integer`. The first element is the function that adds 0 to its argument, the second adds 1 to its argument, and so on: the ith element add $i-1$ to its argument. The definition of `fun42` must be a list comprehension.
     For example, `(fun42!!5) 10` should return `15`.
     Answer: `fun42 = [(\x -> x + n) | n <- [0..41]]`
   - Write a function `triples` (including its type) that accepts as its input an integer `n`, and returns the lexicographically sorted list of all triples `(a,b,c)` such that $\mathtt{a}+\mathtt{b}+\mathtt{c}=\mathtt{n}$ and $0 \leq \mathtt{a} < \mathtt{b} < \mathtt{c}$. The implementation must be a list comprehension.
     For example `triples 8` should return the list `[(0,1,7),(0,2,6),(0,3,5),(1,2,5),(1,3,4)]`.
     [Note: you can earn 3 points for a correct implementation, or 4 points for an efficient correct implementation]
     Answer:
     ```
     triples :: Integer -> [(Integer,Integer,Integer)]
     triples n = [(a,b,n-a-b) | a <- [0..n `div` 3], b <- [a+1..(n-a) `div` 2], n-a > 2*b]
     ```

5. **infinite lists** (3+3+4=10 points)

   - Give a definition of the infinite list `[[1], [1,2], [1,2,3], [1,2,3,4], ....]`
     Answer: `[[1..n] | n <- [1..]]`
   - Define the infinite list `fibs`, which is the infinite list of fibonacci numbers. Recall that $fib(0) = 0$, $fib(1) = 1$, and $fib(n) = fib(n-1)+fib(n-2)$ for $n > 1$. So, `take 10 fibs` should return `[0,1,1,2,3,5,8,13,21,34]`.
     Answer: `fibs = 0 : 1 : zipWith (+) fibs (tail fibs)`
   - Define the infinite list `abc`, which is the infinite list of all non-empty strings over the alphabet $\{'\mathtt{a}', '\mathtt{b}', '\mathtt{c}'\}$. This list needs to be sorted based on the length of the strings. Moreover, strings of equal length should be sorted lexicographically (dictionary order). For example, `take 15 abc` should return:
     `["a","b","c","aa","ab","ac","ba","bb","bc","ca","cb","cc","aaa","aab", "aac"]`
     Answer: `abc = [x++[a] | x <- "":abc, a <- ['a'..'c']]`

6. (15 points) The abstract data type (ADT) `PQ tp` implements a simple data type for the storage of elements of the type `tp`. The name PQ stands for *Priority Queue*. Elements can be inserted in such a queue in arbitrary order. However, retrieving an element from a non-empty priority queue always yields the smallest element.

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

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

   - `empty`: returns an empty priority queue.
   - `isEmpty`: returns `True` for an empty priority queue, otherwise `False`.
   - `insert`: returns the queue that is the result of inserting an element.
   - `getmin`: returns the 'smallest' element of the queue.
   - `remove`: returns the queue that is obtained by removing the smallest element.

Answer:

```
module PQ (PQ, empty, isEmpty, insert, getmin, remove) where

data PQ a = P [a]

empty = P []

isEmpty (P xs) = null xs

insert x (P xs) = P (ins x xs)

getmin (P xs) = head xs

remove (P xs) = P (tail xs)

-- Note: ins is not exported. ins keeps the array sorted
ins x [] = [x]
ins x (y:ys)
  | x < y     = x:y:ys
  | x == y    = y:ys
  | otherwise = y:(ins x ys)
```

7. **Proof on lists** (10 points) Given are the definitions of the functions `take`, and `drop`:

```
take :: Int -> [a] -> [a]
take 0 xs     = []
take n []     = []
take n (x:xs) = x:take (n-1) xs

drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop n []     = []
drop n (x:xs) = drop (n-1) xs
```

Prove that `take n xs ++ drop n xs = xs` for all finite lists `xs` and $\mathtt{n} \geq 0$.

Answer: First, we prove the case $\mathtt{n} = 0$: `take 0 xs ++ drop 0 xs = [] ++ xs = xs`. So, in the remainder we may assume that $\mathtt{n} > 1$. Of course we use structural induction on the list:

```
Base case: xs = []
  take n [] ++ drop n []
 = -- def. take, drop
  [] ++ []
 = -- def. ++
  []

Ind. Step:
  take n (x:xs) ++ drop n (x:xs)
 = -- def. take, drop
  (x:take (n-1) xs) ++ (drop (n-1) xs)
 = -- def. ++
  x:((take (n-1) xs) ++ (drop (n-1) xs))
 = -- Ind. hypothesis
  x:xs   (Q.E.D.)
```

8. **Proof on trees** (15 points) Given is the data type `Tree`:

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

Given are the functions `inorder` and `mirror`:

```
mirror :: BinTree a -> BinTree a
mirror Empty = Empty
mirror (Node x l r) = Node x (mirror r) (mirror l)

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

Prove for all finite trees `t`: `reverse(inorder(mirror t)) = inorder t`

Answer: We use structural induction:

```
Base: t = Empty
  reverse(inorder(mirror Empty))
 = -- def. mirror
  reverse(inorder(Empty))
 = -- def. inorder
  reverse []
 = -- def. reverse
  []
 = -- def. inorder
  inorder Empty

Ind. Step:
  reverse(inorder(mirror (Node x l r)))
 = -- def. mirror
  reverse(inorder(Node x (mirror r) (mirror l)))
 = -- def. inorder
  reverse(inorder (mirror r) ++ [x] ++ inorder (mirror l))
 = -- assoc. ++ and lemma reverse
  reverse ([x] ++ inorder (mirror l)) ++ reverse(inorder (mirror r))
 = -- again, lemma reverse
  reverse(inorder (mirror l)) ++ reverse [x] ++ reverse(inorder (mirror r))
 = -- ind. hyp. twice
  inorder l ++ [x] ++ inorder r
 = -- def. inorder
  inorder (Node x l r)

We used the lemma: reverse (xs++ys) = reverse ys ++ reverse xs
Base: reverse ([] ++ ys) = reverse ys = reverse ys ++ [] = reverse ys ++ reverse []

Ind. Step:
  reverse ((x:xs)++ys)
 = -- def. ++
  reverse (x:(xs ++ ys))
 = -- def reverse
  reverse (xs ++ ys) ++ [x]
 = -- ind. hyp.
  reverse ys ++ reverse xs ++ [x]
 = -- assoc ++
  reverse ys ++ (reverse xs ++ [x])
 = -- def. reverse
  reverse ys ++ reverse (x:xs)    (Q.E.D.)
```