solutions Exam Functional Programming – Nov. 6th 2017

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

```
[(('A', "BC"),[True])]
```

```
[((Char, [Char]), [Bool])]    or   [((Char, String), [Bool])]
```

(b) What is the most general type of the following expression?

```
[id,abs]
```

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

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

```
f g (x,y)= g x y
```

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

(d) What is the type of the function `map`?

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

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

```
h f g x y = f (g x y) x
```

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

2. **Programming in Haskell** (10 points)
   The *Luhn algorithm* is a simple checksum formula used to validate a credit card number. It works as follows.

Let n be the number to be checked, e.g. $n = 49927398716$. The first step is to double every second digit, starting from the right (so, in the example we only consider the digits 4.9.7.9.1.). If doubling a digit results in a number greater than 9 then subtract 9 from the number. For this example, the result of this first step is the number $x = 49947697726$. In the second step of the algorithm, all digits of x are summed up. For the example, we get $4+9+9+4+7+6+9+7+7+2+6 = 7$. If this sum is evenly divisible by 10, then the number is a valid credit card number, otherwise it is invalid. The conclusion is that the example number is a valid credit card number.

Write a Haskell function `isValidNumber :: Integer -> Bool` that returns `True` if and only if its first argument is a valid credit card number. So, `isValidNumber 49927398716` should return `True`.

```
isValidNumber :: Integer -> Bool
isValidNumber n = sum (double (digits n)) `mod` 10 == 0
  where
    digits n  = if n < 10 then [n] else (n `mod` 10):digits (n `div` 10)
    double (x:y:xs) = x:dbl y:double xs
    double xs = xs
    dbl x     = if x < 5 then 2*x else 2*x - 9
```

3. **Higher order functions** (3+3+4=10 points)

- Using the higher-order functions `foldr` and `map`, define a function `powersum` (including its type) which takes two non-negative integers `n` and `e`. It returns the sum of the first `n` positive integers raised to the power `e`.
  So, `powersum 4 3 = 1^3 + 2^3 + 3^3 + 4^3 = 1 + 8 + 27 + 64 = 100`.

```
powersum :: Integer -> Integer -> Integer
powersum n e = foldr (+) 0 (map (^e) [1..n])
```

- Define the function `filter` (including its type) using the function `foldr`.

```
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\x xs -> if p x then x:xs else xs) []
```

- Using function composition (i.e. '`.`'), `foldr`, `map` and the identity function `id`, write a function `pipeline` (including its type) which given a list of functions, each of type `a -> a`, will form a pipeline function of type `[a] -> [a]`. In such a pipeline, each function in the original function list is applied in turn to each element of the input (assume the functions are applied from right to left in this case). For example `pipeline [(+1),(*2)] [1,2,3]=[3,5,7]`.

```
pipeline :: [a->a] -> [a] -> [a]
pipeline fs = map (foldr (.) id fs)
```

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

- Give an implementation of the standard Haskell function `concat` (including its type) as a list comprehension. Recall that `concat [[1,2],[],[3]]=[1,2,3]`, and `concat ["hello", "world"]="helloworld"`.

```
concat     :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

- Show how the single comprehension `[(x,y) | x <- [0..m], y <- [0..n]]` with two generators can be re-expressed using two comprehensions with single generators. [Hint: make use of the library function `concat`.]

```
concat [[(x,y) | y<-[0..n]] | x <- [0..m]]
```

- The dot product of two vectors $a = [a_0, a_1, ..., a_n]$ and $b = [b_0, b1, ..., b_n]$ is defined as:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{n} a_i b_i = a_0 b_0 + a_1 b_1 + \cdots + a_n b_n$$

  Complete the following haskell function `dotProduct` using a list comprehension. You may assume that the input lists have the same length.

```
dotProduct :: Num a => [a] -> [a] -> a
dotProduct xs ys = sum comprehension
  where comprehension = [ x*y | (x,y) <- zip xs ys]
```

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

- Give a recursive expression for the list `pairs` which is the infinite list of pairs `(n,n+1)`, where `n` ranges over the natural numbers. So, `take 3 pairs=[(0,1),(1,2),(2,3)]`.

```
pairs = (0,1):map (\(x,y) -> (x+1,y+1)) pairs
```

- Give a definition of the function `gendups` which takes a (possibly infinite) list of positive `Integers` and generates the (posssibly infinite) list where each element x of the input list has been replaced by x copies of itself. For example, `gendups [1,2,1,3] [1,2,2,1,3,3,3]`, and `gendups [1..]=[1,2,2,3,3,3,4,4,4,4,....]`.

```
gendups xs = concat (map (\x -> replicate x x) xs)
```

- Define the function `sums::[Integer] -> [Integer]`, that takes an infinite list of `Integers` and produces the corresponding infinite list of prefix sums.
  For example, `sums [0,2..]` should produce the infinite list `[0, 0+2, 0+2+4, .....]=[0,2,6,....]`.

```
sums xs = map (\(a,b)->a+b) (zip (0:(sums xs)) xs)
```

6. (15 points) The type `Stack a` is an Abstract Data Type (ADT) for *stacks* containing elements of the type `a`. Recall that a stack is a container that works according the LIFO (Last In First Out) principle. In other words, the element that was most recently inserted by a `push` operation is returned by a `top` operation.

   Implement a `module Stack` such that the concrete implementatiion of the type Stack is hidden to the user. You may choose yourself a suitable data representation for stacks.

   The following operations on stacks need to be implemented:

   - `empty` returns an empty stack.
   - `isEmpty` returns `True` for an empty stack, otherwise `False`.
   - `push`: returns the stack that is obtained by adding an element to the stack.
   - `pop`: returns the stack that is obtained bij removing the top element from the stack.
   - `top`: returns the element that was most recently added to the stack.

```
module Stack(Stack, empty, isEmpty, push, pop, top) where

data Stack a = S [a]

empty :: Stack a
empty = S []

isEmpty :: Stack a -> Bool
isEmpty (S []) = True
isEmpty _ = False

push :: a -> Stack a -> Stack a
push x (S xs) = S (x:xs)

pop :: Stack a -> Stack a
pop (S (x:xs)) = S xs

top :: Stack a -> a
top (S (x:xs)) = x
```

7. **Proof on lists** (10 points) Prove that `map (f.g) xs = (map f . map g) xs` for all finite lists `xs`.

```
Base case for xs=[]:
     (map f.map g) []
  = -- definition: f.g x = f(g x)
    map f (map g [])
  = -- definition: map f [] = []
    map f []
  = -- definition: map f [] = []
    []
  = definition: map f [] = []
    map (f.g) []

Induction step: Assume map (f.g) xs = (map f . map g) xs
  We start with the left hand side.
    map (f.g) (x:xs)
  = -- definition: map f (x:xs) = f x : map f xs
    (f.g) x : map (f.g) xs
  = -- definition: (f.g) x = f(g x)
    f(g x) : map (f.g) xs
  = -- induction hypothesis
    f(g x) : ((map f . map g) xs)

  Next we consider the right hand side.
    (map f.map g) (x:xs)
  = -- definition: (f.g) x = f(g x)
    map f (map g (x:xs))
  = -- definition: map f (x:xs) = f x : map f xs
    map f ((g x): map g xs))
  = -- definition: map f (x:xs) = f x : map f xs
    f (g x):(map f (map g xs))
  = -- definition: (f.g) x = f(g x)
    f (g x):((map f.map g) xs)

  So lhs=rhs. QED.
```

8. **Proof on trees** (15 points) Given is the data type `Tree` and the functions `inorder`, and `mapTree`:

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

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

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Empty = Empty
mapTree f (Node x t1 t2) = Node (f x) (mapTree f t1) (mapTree f t2)
```

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

```
Base case for t=Empty:
    inorder (mapTree f Empty)
  = -- definition: mapTree f Empty = []
    inorder []
  = -- definition: inorder [] = []
    []
  = -- definition map: map f [] = []
    map f []
  = -- deinition: inorder Empty = []
    map f (inorder Empty)

Induction step:
Assumption IH1:  inorder (mapTree f t1) = map f (inorder t1)
Assumption IH2:  inorder (mapTree f t2) = map f (inorder t2)
To show: inorder (mapTree f (Node x t1 t2)) = map f (inorder (Node x t1 t2))

    inorder (mapTree f (Node x t1 t2))
  = -- def. of mapTree
    inorder (Node (f x) (mapTree f t1) (mapTree f t2))
  = -- def. of inorder
    inorder (mapTree f t1) ++ [f x] ++ inorder (mapTree f t2)
  = -- IH1 and IH2
    map f (inorder t1) ++ [f x] ++ map f (inorder t2)

    map f (inorder (Node x t1 t2))
  = -- def. inorder
    map f (inorder t1 ++ [x] ++ inorder t2)
  = -- lemma: map f (xs++ys) = map f xs ++ map f ys
    map f (inorder t1) ++ map f ([x] ++ inorder t2)
  = -- again lemma
    map f (inorder t1) ++ map f [x] ++ map f (inorder t2)
  = -- def. of map: map f [x] = map f (x:[]) = f x : map f [] = f x : [] = [f x]
  map f (inorder t1) ++ [f x] ++ map f (inorder t2)

  We need to prove the lemma: map f (xs++ys) = map f xs ++ map f ys
  Base case: map f ([] ++ ys) = map f ys = [] ++ map f ys = map f [] ++ map f ys
  Induction step:
    lhs = map f ((x:xs) ++ ys) = map f (x:(xs++ys)) = (f x):map f (xs++ys)
        = (f x):(map f xs ++ map f ys)
    rhs = map f (x:xs) ++ map f ys = ((f x):map f xs) ++ map f ys)
        =(f x):(map f xs ++ map f ys)

  So, lhs=rhs. QED.
```