Answers Exam Functional Programming – Dec. 4th 2018

You may use the following standard Haskell functions throughout the entire exam:

```
fst (a,b)               = a
snd (a,b)               = b
id x                    = x

[]     ++ ys            =  ys
(x:xs) ++ ys            =  x : (xs++ys)

concat []               = []
concat (xs:xss)         = xs ++ concat xss

map f []                = []
map f (x:xs)            = f x : map f xs

filter p []             = []
filter p (x:xs)
  | p x                 = x:filter p xs
  | otherwise           = filter p xs

foldr f z []            = z
foldr f z (x:xs)        = f x (foldr f z xs)

sum []                  = 0
sum (x:xs)              = x + sum xs

reverse []              = []
reverse (x:xs)          = reverse xs ++ [x]

head (x:xs)             = x
tail (x:xs)             = xs

length []               = 0
length (x:xs)           = 1 + length xs

(f . g) x               = f (g x)

zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip  _ _                = []

zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _ _           = []
```

1. **Types** (5× 2=10 points)
(a) Is the following expression type correct? If your answer is `YES`, then give the most general type of the expression.

```
("x",'x',[True]):[]
```

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

(b) Is the following expression type correct? If your answer is `YES`, then give the most general type of the expression.

```
(+1) . (0<)
```

```
NO
```

(c) Is the following expression type correct? If your answer is `YES`, then give the most general type of the expression.

```
(+1) . (0+)
```

```
YES
Num a => a -> a
```

(d) Is the following definition of `f` type correct? If your answer is `YES`, then give the most general type of `f`.

```
f = []:[[xs] | xs <- f]
```

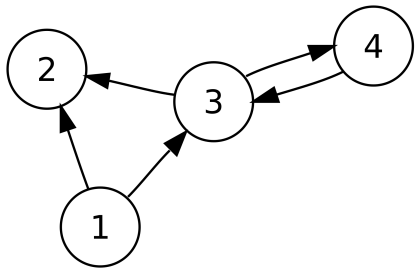
```
NO
```

(d) What is the type of the following function `g`?

```
g = (.).(.)
```

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

2. **Programming in Haskell** (10 points)
We can represent a directed graph by a list (with the type `[(Int,Int)]`) of arcs.



For example, the graph in the above figure can be represented by the list `[(1,2),(1,3),(3,2),(3,4),(4,3)]`. Write a function `paths :: Int -> Int -> [(Int,Int)] -> [[Int]]` such that `paths a b arcs` returns a list containing all the paths from node `a` to node `b` using steps taken from the list `arcs`. Note that these paths may not use an arc more than once. For example, `paths 1 4 [(1,2),(1,3),(3,2),(3,4),(4,3)]` should return `[[1,2],[1,3,2],[1,3,4,3,2]]`.

```
paths a b arcs
  | a == b = [[b]]
  | otherwise = [a:path | arc <- arcs, (fst arc) == a,
                          path <- (paths (snd arc) b [e | e <- arcs, e /= arc])
                ]
```

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

- Give a Haskell implementation (including its type) of the function `mapEach` such that `mapEach f xss` returns a list of lists, containing the result of applying `f` to each element of each list in `xss`.
  For example, `mapEach (+ 2) [[5, 4, 1], [7, 6], []]= [[7, 6, 3], [9, 8], []]`.

```
mapEach :: (a->b) -> [[a]] -> [[b]]
mapEach f xss = [map f xs | xs <- xss]
```

- Assuming the availability of the function `gcd` that returns the greatest common divisor of its two arguments (i.e. `gcd 36 42` returns `6`), implement the function `listgcd` that takes a list of integers, and returns the greatest common divisor of all elements in the list. Your implementation must make use of the function `foldr`. For example, `listgcd [25,15,125,555]` should return 5.

```
listgcd xs = foldr gcd 0 xs
```

- Consider the following haskell definition of the function `scanl`:

  ```
  scanl f z xs = [foldr f z (take len xs) | len <- [0..length xs]]
  ```

  For example, `scanl (+) 1 [1..10]` returns `[1,2,4,7,11,16,22,29,37,46,56]`. The above implementation of `scanl` is quite inefficient (it has quadratic time complexity). Give an equivalent implementation (including its type) that runs in linear time.

```
scanl :: (a -> b -> b) -> b -> [a] -> [b]
scanl f z [] = [z]
scanl f z (x:xs) = z:scanl f (f x z) xs
```

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

- Give an implementation of the standard Haskell function `filter` (including its type) using a list comprehension.

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

- Write a function `sumdiv3not5` (including its type) that takes a list of `Integers` and returns the sum of the squares of those numbers in the list that are divisible by 3 but not by 5. For example, `sumdiv3not5 [-6,15,2,3] = 45`. You must use a list comprehension, and are not allowed to use recursion.

```
sum3div5 :: [Integer] -> Integer
sum3div5 xs = sum [x^2 | x <- xs, mod x 3 == 0, mod x 5 /= 0]
```

- Give an implementation of the standard Haskell function `zipWith` (including its type) using a list comprehension.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipwith f xs ys = [f x y | (x,y) <- zip xs ys]
```

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

- Give a definition of the infinite list `inits` (including its type) of which the nth element is a list containing the numbers 0,1,2,..,n. So, `take 5 inits` should return `[[0],[0,1],[0,1,2],[0,1,2,3],[0,1,2,3,4]]`.

```
inits :: [[Integer]]
inits = [[0..n] | n <- [0..]]
```

- Give a definition of the function `powerfunc` (including its type) that accepts a function `f` on its input, and outputs the infinite list of repeated applications of `f`. The first element (index 0) of this list should be the function `f` raised to the power zero, i.e. the identity function. The second element (index 1) is the function `f` raised to the power one, i.e. `f` itself. The third element (index 2) is the function `f` raised to the power two, i.e. `f(f(x))` for all `x`, and so on. For example, `head ((drop 5) (powerfunc (+2))) 0` should return `10`.

```
powerfunc :: (a -> a) -> [a -> a]
powerfunc f = id:[f.g | g <- powerfunc f]
```

- Consider the following Haskell code:

```
f a b = a : b

g a [] = a : []
g a b  = a : b

list1 = foldr f [] [1..]
list2 = foldr g [] [1..]
```

What will happen if we try to compute `take 10 list1`? And what happens if we try to compute `take 10 list2`? Explain your answers.

```
We first have a look at list1:
list1 = foldr f [] [1..] = foldr f [] (1:[2..]) = f 1 (foldr f [] [2..])
      = 1:(foldr f [] [2..])=....=1:2:3:4:5:6:7:8:9:10:(foldr f [] [11..])
So, using lazy evaluation, take 10 list1=[1,2,3,4,5,6,7,8,9,10].

Next, we take a look at list2:
list2 = foldr g [] [1..] = foldr g [] (1:[2..]) = g 1 (foldr f [] [2..])
But now we have a problem. To apply the pattern matching that is used in the
function definition of g, the system has to compute (foldr f [] [2..]) to
see whether it is [] or not. But that calculation, of course, runs into the
same problem. Hence, nothing is returned, and the evaluation loops forever.
```

6. (15 points) The *unary numeral system* is the simplest numeral system to represent natural numbers. To represent the natural number N, an arbitrarily chosen symbol representing one is repeated N times. For example, the number 5 can be represented by the list $[1, 1, 1, 1, 1]$ (here, the arbitrary chosen symbol is the digit 1). Hence, in this notation, the length of the list is the actual value it represents. We represent the value zero by the empty list.

   The type `NatNum` is an Abstract Data Type (ADT) for implementing natural numbers. Its implementation uses the unary numeral system. Implement a `module NatNum` such that the concrete implementation of the type `NatNum` is hidden to the user.

   The following operations on natural numbers need to be implemented:

   - `integerToNat n` converts the `Integer n` into the `NatNum` that represents `n`.
   - `natToInteger n` converts the natural number `n` into its decimal `Integer` value.
   - `isZero n` returns `True` if and only if the natural number `n` represents 0.
   - `isLessThan a b`: returns `True` if and only if the natural number `a` is less than the natural number `b`.
   - `plus a b`: returns the natural number that is obtained by adding the natural numbers `a` and `b`.
   - `mul a b`: returns the natural number that is obtained by multiplying the natural numbers `a` and `b`.

```
module NatNum(NatNum,integerToNat,natToInteger,plus,mul,isZero,isLessThan) where

data NatNum = N [Int]  -- Or any other list, e.g. [Char]

integerToNat :: Integer -> NatNum
integerToNat n = N [1 | x <- [1..n]]

natToInteger :: NatNum -> Integer
natToInteger (N xs) = length xs

plus :: NatNum -> NatNum -> NatNum
plus (N m) (N n) = N (m++n)

mul :: NatNum -> NatNum -> NatNum
mul (N []) _    = N []
mul _ (N [])    = N []
mul (N (x:xs)) (N n) = plus (N n) (mul (N xs) (N n))

isZero :: NatNum -> Bool
isZero (N n) = ([] == n)

isLessThan :: NatNum -> NatNum -> Bool
isLessThan (N xs) (N ys) = (length xs) < (length ys)
```

7. **Proof on `foldr` and `foldl` (10 points)**
Consider the following Haskell definitions of the functions `foldr` and `foldl`:

```
foldr f z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

foldl f z [] = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Prove that `foldr (+) z xs = foldl (+) z xs` for all finite lists `xs`.

```
We use structural induction:
Base: xs = []
  foldr (+) z [] = z = foldl (+) z []

Ind. Step:
  foldr (+) z (x:xs)
 = -- def. foldr, using infix notation
  x + (foldr (+) z xs)
 = -- ind. hyp.
  x + (foldl (+) z xs)

  foldl (+) z (x:xs)
 = -- def. foldl, using infix notation
  foldl (+) (z + x) xs

So, we need the lemma x + (foldl (+) z xs) = foldl (+) (z + x) xs, which we
prove by induction.

Base: xs = []
  x + (foldl (+) z []) = x + z = z + x = foldl (+) (z + x) []

Ind. Step:
   x + (foldl (+) z (x':xs))
 = -- def. foldl, using infix notation
   x + (foldl (+) (z + x') xs)
 = -- ind. hyp.
   foldl (+) ((z + x')+x) xs
 = -- calculus
   foldl (+) ((x+z) + x') xs
 = -- def. foldl, using infix notation
   foldl (+) (x+z) (x':xs)    (Q.E.D.)
```

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

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

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

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

Prove for all finite trees `t`: `inorder(mirror t) = reverse(inorder t)`
You may use, without a proof, that the operator `++` is associative, i.e. `xs++ys++zs=(xs++ys)++zs=xs++(ys++zs)`.

```
We use structural induction:
Base: t = Empty
  inorder(mirror Empty)=inorder(Empty) = [] = reverse [] = reverse(inorder Empty)

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

  reverse(inorder (Node x l r))
 = -- def. inorder
  reverse(inorder l ++ [x] ++ inorder r)
 = -- assoc. ++
  reverse((inorder l ++ [x]) ++ inorder r)
 = -- lemma reverse
  reverse(inorder r) ++ reverse(inorder l ++ [x])
 = -- lemma reverse
  reverse(inorder r) ++ (reverse [x] ++ reverse (inorder l))
 = -- reverse [x] = [x], assoc. ++
  reverse(inorder r) ++ [x] ++ reverse(inorder l)

So, lhs=rhs. We used the lemma: reverse (xs++ys) = reverse ys ++ reverse xs
Base: xs = []
  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.)
```