Answers Exam Functional Programming – Dec. 3rd 2019

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

```
[True]:[]
```

```
Type correct. The type is [[Bool]]
```

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

```
[]:[True]
```

```
NO. The expression is type incorrect.
```

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

```
f = (\x -> \y -> \z -> [x (y z), y z])
```

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

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

```
g = \x -> \y -> \z -> x.y.z
```

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

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

```
h = foldr (&&)
```

```
h :: Bool -> [Bool] -> Bool
```

2. **Programming in Haskell** (10 points)
   We call an `Integer` n a *trinumber* if n can be expressed as a sum of distinct powers of three (i.e. no duplicates of powers of three are allowed). For example, the numbers 1, 3, 9, 12, and 118 are all trinumbers because:

$$
\begin{aligned}
1 &= 3^0 \\
3 &= 3^1 \\
9 &= 3^2 \\
12 &= 3^1 + 3^2 \\
118 &= 3^0 + 3^2 + 3^3 + 3^4
\end{aligned}
$$

   Note that the number 20 can be expressed as a sum of powers of three as follows: $20 = 3^0 + 3^0 + 3^2 + 3^2$, however 20 is not a trinumber because the powers of three are not distinct.

   Give a implementation of `isTriNumber n` (including its type) which returns `True` if and only if `n` is a trinumber.

```
triNumber :: Integer -> Bool
triNumber n = tri 1 n
  where tri m n = (m==n) || ((m<n) && ((tri (3*m) n) || (tri (3*m) (n-m))))
```

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

- Give an implementation of the function `length` that makes use of `foldr`.

```
length xs = foldr (\_ -> (1+)) 0 xs
```

- The function `aligned` accepts two lists, and returns the number of aligned elements in the two lists. For example, `aligned "abca" "abdae"` should return 3. Give an implementation of the function `aligned` that does not make use of recursion or a list comprehension. What is the type of the function `aligned`?

```
aligned :: Eq a => [a] -> [a] -> Int
aligned xs ys = length (filter id (zipWith (==) xs ys))
```

- The function `concatMap` is defined as follows: `concatMap f xs = concat(map f xs)`
  Give an alternative implementation of `concatMap` using the function `foldr`. What is the type of `concatMap`?

```
concatMap :: (a -> [b]) -> [a] -> [b]
%concatMap f = foldr ((++) . f) []
```

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

- What is the output of the expression `take 6 [(x,y) | x <- [1..], y <- [x+1..]]`?

```
[(1,2),(1,3),(1,4),(1,5),(1,6),(1,7)]
```

- The function `evenLists` is defined as: `evenLists xss = map (filter even) xss`.
  Given an alternative implementation of this function using a list comprehension.

```
evenLists xxs = [ [ x | x <- xs, x `mod` 2 == 0 ] | xs <- xxs]
```

- The function `triples` takes three finite lists and combines them as follows. Let `xs=`$[x_0, x_1, x_2, .., x_l]$, `ys=`$[y_0, y_1, y_2, .., y_m]$, and `zs=`$[z_0, z_1, z_2, .., z_n]$, and q the minimum of l, m, and n.
  Then `triples xs ys zs=`$[(x_0, y_0, y_0), (x_1, y_1, z_1), (x_2, y_2, z_2,)..,(x_q, y_q, z_q)]$.
  For example, `triples [0..3] [2..10] [3..20]=[(0,2,3),(1,3,4),(2,4,5),(3,5,6)]`.
  Give the type of the function `triples` and an implementation using a list comprehension.

```
triples :: [a] -> [b] -> [c] -> [(a, b, c)]
triples xs = [(a,b,c) | (a,(b,c)) <- zip xs (zip ys zs)]
```

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

- Define the infinite list `fibs` of Fibonacci numbers using a list comprehension. So, `take 10 fibs` should return `[0,1,1,2,3,5,8,13,21,34]`. Note that `fibs=[fib n| n <- [0..]]` is not considered a valid answer.

```
fibs = 0 : 1 : [ x + y | (x,y) <- zip fibs (tail fibs)]
```

- Without using a list comprehension, give a definition of the infinite list `natlists=[[0],[0,1],[0,1,2],...]`.

```
natlists = [0]:map ((0:).(map (+1))) natlists
```

- Implement the function `multiples` that takes a finite list of `Integers` and outputs the increasing infinite list of positive integers that can be expressed as a multiple of one (or more) of the numbers in the input list.
  For example, `take 10 (multiples [5,2,8])` should return `[2,4,5,6,8,10,12,14,15,16]`.

```
multiples xs = foldr merge [] [[x,2*x..] | x <- xs]
  where
    merge (x:xs) (y:ys)
      | x < y = x:merge xs (y:ys)
      | y < x = y:merge (x:xs) ys
      | otherwise = x:merge xs ys
```

6. (15 points) The type `Complex` is an Abstract Data Type (ADT) for complex numbers.
Implement a `module Complex` such that the implementation of the type `Complex` is hidden to the user. Recall that the complex number $a + ib$ (where i is the imaginary number for which $i^2 = -1$) can be represented as a pair (a, b) where a and b are `Double`s. The following operations need to be implemented:

   - `add`: returns the complex addition of two complex numbers. Recall that $(a + ib) + (c + id) = (a + c) + i(b + d)$.
   - `sub`: returns the complex subtraction. Recall that $(a + ib) - (c + id) = (a - c) + i(b - d)$.
   - `mul`: returns the multiplication of two complex numbers. Recall that $(a + ib)(c + id) = (ac - bd) + i(ad + bc)$.

```
module Complex (Complex, make, add, sub, mul) where

data Complex = C Double Double

{- Note: the following function 'make' was not asked for, and hence is
   not taken into account in the grading. However, you need such a
   function to make use of the ADT.
 -}
make :: (Double,Double) -> Complex
make (a,b) = C a b

add :: Complex -> Complex -> Complex
add (C a b) (C c d) = C (a+b) (c+d)

sub :: Complex -> Complex -> Complex
sub (C a b) (C c d) = C (a-b) (c-d)

mul :: Complex -> Complex -> Complex
mul (C a b) (C c d) = C (a*c - b*d) (a*d + b*c)
```

7. **Proof (lists)** (10 points) Consider the following Haskell function `rvl`.

```
rvl [] ys = ys
rvl (x:xs) ys = rvl xs (x:ys)
```

Prove that `rvl (xs++ys) [] = rvl ys (rvl xs [])` for all finite lists `xs` and `ys`.

```
The property is too specific to prove directly. It is a lot easier to prove the
more general lemma:  rvl (xs++ys) zs = rvl ys (rvl xs zs)
If we can prove that, then the property is trivial, since we can
substitue zs=[] in the lemma.

The lemma is easily proved using structural induction on xs.
Base: rvl ys (rvl [] zs) = rvl ys zs = rvl ([] ++ ys) zs.

Inductive case: Assume that the lemma holds for xs.
   rvl ((x:xs)++ys) zs
 = {def. ++}
   rvl (x:(xs++ys)) zs
 = {def. rvl}
   rvl (xs++ys) (x:zs)
 = {induction hypothesis}
   rvl ys (rvl xs (x:zs))
 = {def. rvl}
   rvl ys (rvl (x:xs) zs)   QED.
```

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

```
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

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

Prove for all finite trees `t`: `inorder (mirror t) = reverse (inorder t)`
[Note: You may use without proof that the operator ++ is associative. If you need any other lemmas to complete the proof, then prove these lemmas separately.]

```
The prove is by structural induction on Trees.
However, in that proof we need the following lemma:
  reverse (xs++ys) = reverse ys ++ reverse xs
We start by proving the lemma using structural induction on the list xs.
Base: reverse ([]++ys) = reverse ys = reverse ys ++ [] = reverse ys ++ reverse []

Inductive step: assume that the lemma holds for xs.
  reverse ((x:xs)++ys)
 = {def. ++}
  reverse (x:(xs++ys))
 = {def. reverse}
  reverse (xs++ys) ++ [x]
 = {induction hypothesis}
  (reverse ys ++ reverse xs) ++ [x]
 = {associativity ++}
  reverse ys ++ (reverse xs ++ [x])
 = {def. reverse}
  reverse ys ++ reverse (x:xs)   QED.

Now, we prove the main property on Trees.
Base: inorder (mirror Empty) = inorder Empty = []
    = reverse [] = reverse (inorder Empty)

Inductive case: Assume that the property holds for tree l and r.
  inorder (mirror (Node x l r))
 ={def. mirror}
  inorder (Node x (mirror r) (mirror l))
 ={def. inorder}
  inorder(mirror r) ++ [x] ++ inoder(mirror l)
 ={induction hypothesis (twice)}
  reverse(inorder r) ++ [x] ++ reverse(inorder l)
 ={[x] = []++x = (reverse [])++[x]=reverse(x:[])=reverse[x]}
  reverse(inorder r) ++ reverse [x] ++ reverse(inorder l)
 ={associativity ++}
  (reverse(inorder r) ++ reverse [x]) ++ reverse(inorder l)
 ={lemma}
  reverse([x]++inorder r) ++ reverse(inorder l)
 ={lemma}
  reverse(inorder l ++ [x] ++ inorder r)
 ={def. inorder}
  reverse(inorder (Node x l r))    QED.
```