Answers Exam Functional Programming – December 1, 2020

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.

   `[not,(&& True)]`

   ```
   [Bool->Bool]
   ```

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

   `[[(*)],(+)]`

   ```
   NO
   ```

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

   `(42 - ) . (+ (42::Int))`

   ```
   Int -> Int
   ```

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

   `g f = (:).f`

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

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

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

   ```
   f :: (a->a,a) -> a -> (a,a)
   ```

2. **Programming in Haskell** (10 points)
   Implement a function `longestPalsub :: Eq a => [a] -> [a]` such that the call `longestPalSub xs` returns the longest subsequence of `xs` which is a palindrome. Here, a subsequence consists of a consecutive run of elements from `xs`. The time complexity of your solution should not exceed $O(n^3)$, where n is the length of `xs`. You are not allowed to use the indexing operator `(!!)`. For example, `longestPalsub "Be careful to step on no pets he said."` should return `" step on no pets "`, while `longestPalsub [3,1,4,1,5,9,2,6,5]` should return `[1,4,1]`.

   ```
   revsubs []     = []
   revsubs (x:xs) = prefixes xs [x] ++ revsubs xs
     where
       prefixes [] acc     = [acc]
       prefixes (x:xs) acc = acc:prefixes xs (x:acc)

   isPalindrome xs = xs == reverse xs

   longest xss = lng xss [] 0
     where
       lng [] ys _         = ys
       lng (xs:xss) ys ln
         | length xs > ln = lng xss xs (length xs)
         | otherwise      = lng xss ys ln

   longestPalsub xs = longest (filter isPalindrome (revsubs xs))
   ```

3. **Higher order functions** ($5 \times 2 = 10$ points)

   - Write a function `splitWhen` (including its type) which takes a predicate `p` and a list `xs` and returns a tuple `(x,ys,zs)` such that `p x` is `True`, `xs=ys++[x]++zs`, and `p y` is `False` for all `y` in `ys`. You may assume that `p x` is `True` for at least one element of `xs`. For example, `splitWhen even [1,3,4,5,2,1]` should return `(4,[1,3],[5,2,1])`.

```
splitWhen :: (a -> Bool) -> [a] -> (a,[a],[a])
splitWhen p xs = split xs []
  where
    split (x:xs) ys
      | p x       = (x,reverse ys,xs)
      | otherwise = split xs (x:ys)
```

   - Give an implementation (and its type) of the standard Haskell function `curry`.

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f = (\a b -> f (a,b))
```

   - Implement a funtion `map2` (including its type) which takes a function `f` and a list of lists `xss` and outputs the list of lists that is obtained by applying `f` to the elements of the lists in `xss`. For example `map2 (*2) [[],[1,2],[5,6]]` should return `[[],[2,4],[10,12]]`.

```
map2 :: (a -> b) -> [[a]] -> [[b]]
map2 f xss = map (map f) xss
```

   - The function `count` is recursively defined as:

```
count _ [] = 0
count p (x:xs)
  |p x       = 1 + count p xs
  |otherwise = count p xs
```

   Given an implementation of count (including its type) that does not use recursion nor a list comprehension.

```
count :: (a -> Bool) -> [a] -> Int
count p = length.filter p
```

   - Implement the function `reverse` using `foldr`.

```
reverse = foldr (\x ys -> ys ++ [x]) []
```

4. **List comprehensions** (2+2+3+3=10 points)

   - Implement the function `isSorted :: [Int] -> Bool` such that `isSorted xs` is `True` if and only if the list `xs` is ascending (i.e. each element is less or equal to its successor). Make use of a list comprehension together with the function `and`. For example, `isSorted [1,2,3,3,4]` should return `True` while `isSorted [2,1]` should return `False`.

```
isSorted  xs = and [x<=y | (x,y) <- zip  xs (tail  xs)]
```

   - Use a list comprehension to implement the function `locations :: Eq a => a -> [a] -> [Int]` which takes an item `x` and a list `xs` ands returns a list of indexes at which `x` is found in `xs`. Note that the first element of a list has index 0. For example, `locations 1 [3,1,4,1,5,9,2,6,5,1]` should return `[1,3,9]`. You are not allowed to use the indexing operator (`!!`).

```
locations a xs = [idx | (x,idx) <- zip idx [0..], x==a]
```

   - Given is the following function `fun`.

```
fun p n = concat (map f (filter p [1..n]))
  where f x = map (\y -> (x,y)) [1..x]
```

   Give an equivalent implementation using a list comprehension. You are not allowed to use `concat`, `filter` or `map`. Also, give the type of the function `fun`.

```
fun ::  (Int -> Bool) -> Int -> [(Int,Int)]
fun p n = [(x,y) | x <- [1..n], p x, y<-[1..x]]
```

- Matrices can be represented in Haskell as lists of lists. For example, `[[1,2,3],[4,5,6]]` represents the 2×3 matrix of which the first row is `[1,2,3]` and the second row is `[4,5,6]`. Write a function `transpose` that takes a matrix (i.e. a lists of lists) and returns the transposed matrix. So, `transpose[[1,2,3],[4,5,6]]` should return `[[1,4],[2,5],[3,6]]`. Your solution must make use of list comprehensions combined with recursion. You may assume that the input matrix is rectangular (i.e. each row has the same length). You are not allowed to use the indexing operator (`!!`).

```
transpose ([]:xss) = []
transpose xss = [ x | (x:xs) <- xss]:transpose [ xs | (x:xs) <- xss]
```

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

- Assuming the availablility of the infinite list `primes::[Integer]` of prime numbers, use it to define the infinite list `composites::[Integer]` which is the list of all positive integers which are not prime.

```
composites = sieve 1 primes
  where sieve n (p:ps)
          | p == n    = sieve (n+1) ps
          | otherwise = n:sieve (n+1) (p:ps)
```

- Using `zip` or `zipWith`, give a definition of the infinite list `fs` which is the list of numbers which are defined as:

$$F(0) = 0, \quad F(1) = 1, \quad F(n) = 2F(n-1) + F(n-2) \text{ for } n \geq 2$$

So, the expression `take 10 fs` equals `[0,1,2,5,12,29,70,169,408,985]`. Your implementation should be such that `take n fs` has an $O(n)$ time complexity.

```
fs = zipWith (\x y -> 2*x + y) (0:fs) (0:1:fs)
```

- Implement the ordered infinite list `ds23` of all positive integers that can be expressed as $2^i \cdot 3^j$ (where i and j are non-negative integers). For example, `take 15 ds23` equals `[1,2,3,4,6,8,9,12,16,18,24,27,32,36,48]`.

```
ds23 = 1:merge (map (*2) ds23) (map (*3) ds23)
  where
    merge (x:xs) (y:ys)
      | x<y       = x:merge xs (y:ys)
      | x>y       = y:merge (x:xs) ys
      | otherwise = x:merge xs ys
```

6. (15 points) The type `Fifo a` is an Abstract Data Type (ADT) for FIFO queues containing elements of type `a`. Recall that a Fifo queue is a container that works according the First-In-First-Out principle. Implement a module `Fifo` which exports the abstract data type but hides the concrete implementation. You may choose yourself a suitable data representation for Fifo queues.The following operations on queues need to be implemented:

- `empty`: returns an empty queue.
- `isEmpty`: returns `True` for an empty queue, otherwise `False`.
- `insert`: inserts an element in a Fifo queue.
- `retrieve`: returns the 'oldest' element from a non-empty Fifo queue.
- `delete`: returns the fifo that is obtained by removing the 'oldest' element from the queue.
- `size`: returns the number of elements of the Fifo queue.

```
module Fifo(Fifo,empty,isEmpty,insert,retrieve,delete,size) where

data Fifo a = F [a]

empty = F []
```

```
isEmpty (F qs) = qs == []

insert x (F qs) = F (qs ++ [x])

retrieve (F (x:qs)) = x

delete (F (x:qs)) = F qs

size (F qs) = length qs
```

7. **Proof of equality** (10 points) Consider the following Haskell functions.

```
f xs ys zs = g xs (ys ++ zs)

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

Prove that `length (f xs ys zs) = length xs*length ys + length xs*length zs` for all finite lists `xs`, `ys`, and `zs`.

```
The property is easily proved using structural induction on the list xs.
Base case (xs=[]):
  LHS = length (f [] ys zs) = length (g [] (ys++zs)) = length [] = 0
  RHS = length []*length ys + length []*length zs = 0*length ys + 0*length zs = 0
  so LHS=RHS.
Induction step: Assume that the property holds for xs, prove it for x:xs.
    LHS
  = length (f (x:xs) ys zs)
  = {definition f}
    length (g (x:xs) (ys++zs))
  = {definition g}
    length((ys++zs) ++ g xs (ys++zs))
  = {lemma below}
    length (ys++zs) + length (g xs (ys++zs))
  = {definition f}
    length (ys++zs) + length (f xs ys zs)
  = {induction hypothesis}
    length (ys++zs) + length xs*length ys + length xs*length zs

    RHS
  = length (x:xs)*length ys + length (x:xs)*length zs
  = {definition length}
    (1 + length xs)*length ys + (1 + length xs)*length zs
  = {arithmetic}
    length ys + length zs + length xs*length ys + length xs*length zs
  = {lemma below}
    length (ys++zs) + length xs*length ys + length xs*length zs
  so LHS=RHS.

We made use of the following lemma: length (xs++ys) = length xs + length ys
The proof is again by structural induction on xs.
Base: length([]++ys)=length ys = 0 + length ys = length [] + length ys
Inductive step: Assume length (xs++ys) = length xs + length ys
      length ((x:xs)++ys)
    = { definition ++ }
      length (x:(xs++ys))
    = {definition length}
```

```
      1 + length(xs++ys)
    = {induction hypothesis}
      1 + lenght xs + length ys
    = {definition length}
      length(x:xs) + length ys
 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`: `reverse(inorder(mirror t)) = inorder t`
[Note: If you need one or more lemmas to complete the proof, then prove these lemmas separately. You may use without proof that ++ is an associative operator, and that `xs++[]=xs`.]

```
We prove this property using structural induction on Trees.
Base case (t=Empty):
   reverse(inorder(mirror Empty))
 = {definition mirror}
   reverse(inorder Empty)
 = {definition inorder}
   reverse []
 = {definition reverse}
   []
 = {definition inorder}
   inorder Empty

Inductive step (t=Node x l r): assume that the property holds for l and r.
   reverse(inorder(mirror (Node x l r)))
 = {definition mirror}
   reverse(inorder(Node x (mirror r) (mirror l)))
 = {definition inorder}
   reverse(inorder(mirror r) ++ [x] ++ inorder(mirror l))
 = {lemma reverse (see below); ++ is associative}
   reverse(inorder(mirror l)) ++ reverse(inorder(mirror r) ++ [x])
 = {lemma reverse again)}
   reverse(inorder(mirror l)) ++ reverse [x] ++ reverse(inorder(mirror r))
 = {induction hypothesis twice}
   inorder l ++ reverse [x] ++ inorder r
 = {reverse (x:[])=reverse [] ++ [x] = [] ++ [x] = [x]}
   inorder l ++ [x] ++ inorder r
 = {definition inorder}
   inorder (Node x l r)

We used the following lemma: reverse(xs++ys) = reverse ys ++ reverse xs
We prove the lemma using structural induction on \verb|xs|.
Base case (xs=[]):
   reverse ([] ++ ys)
 = {definition ++}
   reverse ys
 = { xs = xs ++ []}
   reverse ys ++ []
 = {definition reverse}
   reverse ys ++ reverse []
```

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