Answers Exam Functional Programming – October 31, 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,[]]
   ```

   ```
   NO
   ```

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

   ```
   [[not],[]]
   ```

   ```
   Yes, [[not],[]] :: [[Bool->Bool]]
   ```

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

   ```
   (&&).(&&)
   ```

   ```
   NO
   ```

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

   ```
   g = not.not
   ```

   ```
   g :: Bool -> Bool
   ```

   (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)
   Consider the following two lists: `[[1,2,3],[4,2],[]]` and `[[2,4],[],[2,3,1]]`. If we ignore the order of elements in lists, then these lists are equal.
   Implement a Haskell function `compareListOfLists :: Eq a => [[a]] -> [[a]] -> Bool` such that the call `compareListOfLists xss yss` returns `True` if and only if `xss` and `yss` are equal if we ignore the order of elements in lists.

```
listCompare :: Eq a => (a -> a -> Bool) -> [a] -> [a] -> Bool
listCompare cmp xs ys = compare xs ys
  where
    compare [] ys     = ys==[]
    compare xs []     = False
    compare (x:xs) ys = compare xs (rm x ys)
    rm _ [] = []
    rm x (y:ys)
      | cmp x y   = ys
      | otherwise = y:rm x ys

compareLists :: Eq a => [a] -> [a] -> Bool
compareLists = listCompare (==)

compareListOfLists :: Eq a => [[a]] -> [[a]] -> Bool
compareListOfLists = listCompare compareLists
```

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

- Write a function `flip` such that `(flip f) a b` returns `f b a`. The implementation must make use of a lambda expression. Also, give the type of the function `flip`.

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

- Without using recursion or a list comprehension, write a function `evenLists` which takes a list of lists of `Integers` as its argument and removes from it every list not containing an even numer. Also, give the type of this function.
  For example, `evenLists [[1,2],[7,5,11],[1,3],[21,2,42]]` should return `[[1,2],[21,2,42]]`.

```
evenLists :: [[Integer] -> [[Integer]]
evenLists = filter (any even)
```

- Implement the funtion `append` such that `append xs ys` returns `xs++ys`. You must make use of the standard function `foldr`, and are not allowed to use the ++ operator itself.

```
append xs ys = foldr (:) ys xs
```

- Without using recursion or a list comprehension, implement the funtion `pals` that takes a list of lists as its argument and removes from it every list that is not a palindrome. Also, give the most general type of this function.
  For example, `pals ["madam","pop","your","stack"]` should return `["madam","pop"]`.

```
pals :: Eq a => [[a]] -> [[a]]
pals = filter (\xs -> xs == reverse xs)
```

- Implement the function `zip` using `zipWith`.

```
zip = zipWith (\x -> \y -> (x,y))
```

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

- Implement the function `replicate` using a list comprehension.

```
 replicate n x = [x | i <- [1..n]]
```

- Use a list comprehension to implement the function `pairs` which takes a list `xs` ands returns a list of all pairs that can be constructed from `xs`. For example, `pairs [1,2,3]` should return the following list (in this order): `[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]`

```
pairs xs = [(x,y) | x <- xs, y <- xs]
```

- The function `pairs2` also takes a list `xs` and outputs a list of pairs. A recursive implementation is given below.
  For example, `pairs2 [1,2,3,4]` returns `[(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]`.

```
pairs2 [] = []
pairs2 (x:xs) = p x xs ++ pairs2 xs
  where p x [] = []
        p x (y:ys) = (x,y):p x ys
```

  Give an equivalent implementation that makes use of (a) list comprehension(s) that replaces the recursions.

```
pairs2 xs = [(x,y) | (x,i) <- zip xs [1..], y <- drop i xs]
```

- The function `perms` takes a list of `Ints` and returns a list of all possible permutations of this list. For example, `perms [1..3]` should return (in this order) `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`. Implement `perms` using a list comprehension.

```
perms [] = [[]]
perms xs = [ x:ys| x <- xs, ys <- perms (delete x xs)]
  where
    delete x [] = []
    delete x (y:ys)
      | x == y    = ys
      | otherwise = y:delete x ys
```

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

   - Assuming the availablility of the infinite list `primes::Integer` of prime numbers, write a function `isPrime n` that returns `True` only if `n` is a prime number.

```
isPrime n = n == head(reverse(takeWhile (<=n) primes))
```

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

$$F(n) = n \text{ for } n < 3, \quad F(n) = F(n-1) + F(n-3) \text{ for } n \geq 3$$

   So, the expression `take 10 delayedFib` equals `[0,1,2,2,3,5,7,10,15,22]`.

```
delayedFib = 0:1:2:zipWith (+) delayedFib (drop 2 delayedFib)
```

   - Implement the infinite list `abc` which consist of all strings that can be produced with the letters ’`a`’, ’`b`’, and ’`c`’. For example, `take 25 abc` should return:
   `["a","b","c","aa","ba","ca","ab","bb","cb","ac","bc","cc","aaa","baa","caa",`
   `"aba","bba","cba","aca","bca","cca","aab","bab","cab","abb"]`

```
abc = [ c:s | s <- "":abc, c <- "abc"]
```

6. (15 points) The type `RLElist` is an Abstract Data Type (ADT) that is used to store lists that typically contain chunks of repeated values. A typical example of that would be a list like `[1,1,1,4,5,2,2,2,2,2,2,1,1,1]`. This list can be stored more compactly as a list of pairs, where the first element represents a data element and the second contains the length of the chunk. This type of data storage is called RLE (Run Length Encoding). For the given example, this representation would be `[(1,3),(4,1),(5,1),(2,6),(1,3)]`. abc = [c:s — s ¡- ””:abc, c ¡- ”abc”] Implement a module `RLElist` that exports the ADT `RLElist` but hides the implementation. The following operations need to be implemented:

   - `fromList xs` returns the `RLElist` representation of the standard list `xs`.
   - `toList xs` converts the `RLElist xs` into a standard list.
   - `hd xs` returns the head of the non empty `RLElist xs`.
   - `tl xs` return the tail of the non empty `RLElist xs`.
   - `cons x xs` returns the `RLElist` that is obtained by placing the element `x` ahead of the `RLElist xs`.
   - `cat xs ys` returns the `RLElist` that is obtained by concatenating the `RLElists` `xs` and `ys`.
   - `len xs` returns the length (the number of data items) in the `RLElist xs`.
   - `rev xs` returns the `RLElist` that is obtained by reversing the data lements in the `RLElist xs`.

```
module RLElist(RLElist,fromList,toList, hd, tl, cons, cat, len, rev) where

data RLElist a = RLE [(a,Int)]

fromList xs = foldr cons (RLE []) xs

toList (RLE xs) = concat [replicate n x | (x,n) <- xs]

hd (RLE ((x,_):xs)) = x

tl (RLE ((x,1):xs)) = RLE xs
tl (RLE ((x,n):xs)) = RLE ((x,n-1):xs)

cons x (RLE []) = RLE [(x,1)]
cons x (RLE ((y,n):ys))
  | x == y    = RLE ((x,n+1):ys)
  | otherwise = RLE ((x,1):(y,n):ys)

cat (RLE xs) (RLE ys) = RLE (ct xs ys)
```

```
  where
    ct [] ys          = ys
    ct [(x,m)] ((y,n):ys)
     | x == y         = (x,m+n):ys
     | otherwise      = (x,m):(y,n):ys
    ct ((x,m):xs) ys = (x,m):ct xs ys

len (RLE xs) = sum [n | (x,n) <- xs]

rev (RLE xs) = RLE (reverse xs)
```

7. **Proof of equality** (10 points) Given is the following Haskell function.

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

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

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

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 `size`:

```
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)

size :: Tree a -> Integer
size Empty = 0
size (Node x l r) = size l + 1 + size r
```

Prove for all finite trees `t`: `size(t) = length(inorder t)`
[Note: If you need one or more lemmas to complete the proof, then prove these lemmas separately.]

```
We prove this property using structural induction on Trees.
Base case (t=Empty):
   size(Empty)
 = {definition size}
   0
 = {def. length}
   length []
 = {definition inorder}
   length(inorder Empty)

Inductive step (t=Node x l r): assume that the property holds for l and r.
   size (Node x l r)
 = {definition size}
   size l + 1 + size r
 = {induction hypothesis twice}
   length(inorder l) + 1 + length(inorder r)
 = {definition length}
   length(inorder l) ++ length(x:inorder r)
 = {lemma used in problem 7}
   length(inorder l ++ (x:inorder r))
 = {definition inorder}
   length(inorder (Node x l r))
```