Answers Exam Functional Programming – December 3rd 2014

1. ($5\times$ 2=10 points)
   (a) What is the type of the following Haskell function `wtel`?

```
wtel [] = []
wtel (x:xs) = if x == [] then wxs else x:wxs
  where wxs = wtel xs
```

   Answer: `wtel :: Eq a => [[a]] -> [[a]]`

   (b) What is the type of the following Haskell function `cl`?

```
cl ps = ps ++ [(p,s) | (p,q) <- ps, (r,s) <- ps, q==r]
```

   Answer: `cl :: Eq a => [(a, a)] -> [(a, a)]`

   (c) What is the type of the standard Haskell indexing operator `!!` (as an example `[0..10]!!3 = 3`)?

   Answer: `(!!) :: [a] -> Int -> a`

   (d) What is the type of the following Haskell function `map2`?

```
map2 f [] [] = []
map2 f (x:xs) (y:ys) = (f x y) : map2 f xs ys
```

   Answer: `map2 :: (a -> b -> c) -> [a] -> [b] -> [c]`

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

```
tw = (\f -> (\x -> (f.f) x))
```

   Answer: `tw :: (a -> a) -> a -> a`

2. (15 points) Consider a positive integer N. We denote its decimal digits by X_0, X_1, ..., X_k. The number N is called a *funny number* if you can select at most three (but at least one) of its digits such that N is a divisor of the number $(X_0 + X_1 + ... + X_k - S)^S$, where S is the sum of the selected digits. As an example, 1458 is a funny number since $((1+4+5+8)-(1+5))^{1+5} = 12^6 = 2985984$ is divisible by 1458. Note that we selected the two digits 1 and 5.

   Write a Haskell function `isFunny` (including its type) that takes an integer number as its argument, and returns `True` if and only if this argument is a funny number.

   Solution:

```
choose :: Integer -> [Integer] -> [[Integer]]
choose 0 _      = [[]]
choose _ []     = []
choose n (x:xs) = [x:cs | cs <- choose (n-1) xs] ++ choose n xs

combinations :: [Integer] -> [[Integer]]
combinations xs = choose 1 xs ++ choose 2 xs ++ choose 3 xs

isFunny :: Integer -> Bool
isFunny n = or [(digsum - s)^s `mod` n == 0 | s <- combs]
  where digs   = digits n
        digsum = sum(digs)
        combs  = map sum (combinations digs)
```

3. (3+3+4=10 points)

- Use a list comprehension to define a function `inverse::[(a, b)]->[(b, a)]` such that `elem (x,y) ps` if and only if `elem (y,x) (inverse ps)`.

  Answer: `inverse ps = [(b,a) | (a,b) <- ps]`

- Use a list comprehension to make your own implementation of the standard Haskell function `replicate`. The call `replicate n x` yields a list of length `n` with `x` being the value of every element. So, `replicate 5 'a'` returns `"aaaaa"`.

  Answer: `replicate n x = [x | k <- [1..n]]`

- Define a function `doubleReverse` which takes a list of strings as its argument and reverses each element of the list and then reverses the resulting list. The implementation of `doubleReverse` must use a list comprehension. As an example, `doubleReverse ["hello", "world"] = ["dlrow", "olleh"]`.

  Answer: `doubleReverse xss = reverse [reverse xs | xs <- xss]`

4. (3+3+4=10 points)

- The function `powers n` returns the infinite list $[\mathtt{n}^0, \mathtt{n}^1, \mathtt{n}^2, \mathtt{n}^3, ...]$. Give a *recursive* Haskell implementation (including its type) of the function `powers`.

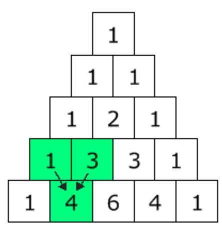
  Answer:

```
powers :: Integer -> [Integer]
powers n = 1 : map (n *) (powers n)
```

- The sequence a_k is defined as follows: $a_0 = 1$, $a_1 = 2$, $a_k = 3a_{k-1} + 2a_{k-2}$ for integer $k > 1$. Define the infinite list `seqa`, which is the list $[a_0, a_1, a_2, a_3, a_4, ...]$, so `seqa!!k` should yield a_k.

  Answer: `seqa = 1 : 2 : zipWith (+) (map (*2) seqa) ((map (* 3)) (tail seqa))`

- In the following figure you see the first 5 rows of Pascal's triangle:



  To build the triangle, start with the row `[1]` at the top (we call this row 0), then continue placing numbers below it in a triangular pattern. Each row consists of elements that are the sum of the two numbers above it (except for the boundaries, which are all `1`). In the figure, it is highlighted that the 4 in row 4 is obtained by adding the numbers 1 and 3 from row 3.
  Give a definition of the infinite list `pascalTriangle ::[[Integer]]`, such that `pascalTriangle!!n` yields the `n`th row of Pascal's triangle (i.e. `pascalTriangle!!4 = [1,4,6,4,1]`).

  Answer:

```
pascalTriangle = iterate nextRow [1]
  where nextRow row = zipWith (+) ([0] ++ row) (row ++ [0])
```

5. (15 points) The abstract data type (ADT) `Set tp` implements a data type for the storage of *sets* of the type `tp`, where `tp` is of the class `Ord` (i.e. the elements are ordered).

   Implement a `module Set` that exports the ADT `Set`. You can choose a concrete implementation yourself, however this implementation must be hidden from the user of this module.

   The following operations on the data type `Set` must be implemented:

   - `empty` returns an empty set.
   - `isEmpty` returns `True` for an empty set, otherwise `False`.
   - `insert`: returns the set after insertion of an element.
   - `delete`: returns the set after removal of an element.
   - `union`: returns the union of two sets.
   - `intersection`: returns the intersection of two sets.

   Answer:

```
module Set (Set, empty, isEmpty, insert, delete, union, intersection) where

data Set a = S [a]

empty = S []

isEmpty (S xs) = null xs

insert x (S xs) = S (ins x xs)

delete x (S xs) = S (del x xs)
  where
    del x [] = []
    del x (y:ys)
      | x < y     = y:(del x ys)
      | x == y    = ys
      | otherwise = y:ys

union (S xs) (S [])     = (S xs)
union (S xs) (S (y:ys)) = union (S (ins y xs)) (S ys)

intersection (S xs) (S [])         = S []
intersection (S []) (S ys)         = S []
intersection (S (x:xs)) (S (y:ys))
  | x < y                          = intersection (S xs) (S (y:ys))
  | x > y                          = intersection (S (x:xs)) (S ys)
  | otherwise                      = insert x (intersection (S xs) (S ys))

-- Note: ins is not exported
ins x [] = [x]
ins x (y:ys)
  | x < y     = x:y:ys
  | x == y    = y:ys
  | otherwise = y:(ins x ys)
```

6. (15 points) Given are the following Haskell definitions of the functions `f` and `g`:

```
f :: Integer -> Integer
f 0 = 0
f 1 = 1
f n = 5*(f (n-1)) - 6*(f (n-2))

g :: Integer -> Integer -> Integer
g n 0 = 1
g n e = n*(g n (e - 1))
```

Prove for all natural numbers `n`: `f n = g 3 n - g 2 n`

Answer: It is easy to see that $\mathtt{g\ n\ e} = \mathtt{n}^{\mathtt{e}}$. We start by proving this first:

Base case (e=0): `g n 0 = 1 =` $\mathbf{n}^0$
Inductive step: `g n (e+1) = n*(g n e)`$= \mathbf{n} * \mathbf{n}^{\mathbf{e}} = \mathbf{n}^{\mathbf{e}+1}$ QED.

So, we need to prove: $\mathtt{f\ n} = 3^{\mathtt{n}} - 2^{\mathtt{n}}$.

Base case (n=0): `f 0 = 0 = 1 - 1 =` $3^0 - 2^0$
Base case (n=1): `f 1 = 1 = 3 - 2 =` $3^1 - 2^1$
Inductive step: `f (n+1) = 5*(f n)-6*(f (n-1)) =` $5(3^n - 2^n) - 6(3^{n-1} - 2^{n-1})$
$= 5(3^n - 2^n) - 2 \cdot 3^n + 3 \cdot 2^n = 3 \cdot 3^n - 2 \cdot 2^n = 3^{\mathtt{n}+1} - 2^{\mathtt{n}+1}$

QED.

7. (15 points) Given are the definitions of the Haskell functions `sum`, and `reverse`:

```
sum :: [Integer] -> Integer
sum [] = 0
sum (x:xs) = (sum xs) + x

reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

Prove that `sum (reverse xs) = sum xs` for all finite lists `xs`.

Answer:

```
sum (reverse [])     = sum []                     -- def reverse
sum (reverse (x:xs)) = sum (reverse xs ++ [x])    -- def reverse
                     = sum (reverse xs) + sum [x] -- Lemma below
                     = sum (reverse xs) + x       -- def sum
                     = x + sum xs                 -- inductive hypothesis
                     = sum (x:xs)                 -- definition of sum

Lemma:  sum (xs ++ ys) == sum xs + sum ys

sum ([] ++ ys)      = sum ys            -- def (++)
                    = 0 + sum ys        -- identity of addition
                    = sum [] ++ sum ys -- def sum

sum ((x:xs) ++ ys) = sum (x : (xs ++ ys))  -- def (++)
                   = x + sum (xs ++ ys)    -- def sum
                   = x +  sum xs + sum ys  -- inductive hypothesis
                   = sum (x:xs) + sum ys   -- def sum
```

QED.