# Answers Exam Functional Programming – November 3rd 2014

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

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

   **Answer:** `zip :: [a] -> [b] -> [(a, b)]`

   **(b)** What is the type of the standard Haskell function `concat`?

   ```
   concat = foldr (++) []
   ```

   **Answer:** `concat :: [[a]] -> [a]`

   **(c)** What is the type of the following Haskell function `uncurry`?

   ```
   uncurry f = (\(a,b) -> f a b)
   ```

   **Answer:** `uncurry :: (a -> b -> c) -> (a, b) -> c`

   **(d)** What is the type of the following Haskell function `plus1`?

   ```
   plus1 = map (+ 1)
   ```

   **Answer:** `plus1 :: [Integer] -> [Integer]`

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

   ```
   f = sum.h.g
   g = (\x -> (head x, (head.reverse) x))
   h (x,y) = [x,y]
   ```

   **Answer:** `f ::  Num a => [a] -> a`

2. (10 points) A Dutch Citizen Service Number (DCSN) has always 9 digits and the first digit can be a 0. Many websites use the following rudimentary check to validate the correctness of the (9 digit) number $ABCDEFGHI$. First compute $X = 9\times A + 8\times B + 7\times C + 6\times D + 5\times E + 4\times F + 3\times G + 2\times H - 1\times I$. Note that the last digit has a negative weight. If $X$ is a multiple of 11, then the number $ABCDEFGHI$ passes the test, otherwise it is invalid.

   Write a Haskell functie `isDCSN` (including its type) that determines whether its argument passes the test described above.

   **Answer:**

   ```
   isDCSN :: Integer -> Bool
   isDCSN n = (mergesum (digits n) (-1:[2..9])) `mod` 11 == 0
     where
       digits 0              = []
       digits n              = n `mod` 10 : digits (n `div` 10)
       mergesum [] _         = 0
       mergesum (x:xs) (y:ys) = x*y + mergesum xs ys
   ```

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

- Write a function `relPrimePairs n` that returns the list of pairs `(i,j)` where `1<i<j<=n` and `i` and `j` have no common factor (you may use the function `gcd` that computes the greatest common divisor of its two arguments). The implementation of `relPrimePairs` must be a list comprehension.

  **Answer:** `relPrimePairs n = [(i,j) | i <- [2..n], j <- [i+1..n], gcd i j == 1]`

- Given are the Haskell definitions of `suits`, `cards` and `honours`:

```
suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
cards = map show [2..10]
honours = ["J","Q","K","A"]
```

  Write a list comprehension for `deck`, where `deck` is

```
[("Clubs","2"),("Clubs","3"),("Clubs","4"),("Clubs","5"),("Clubs","6"),("Clubs","7"),
("Clubs","8"),("Clubs","9"),("Clubs","10"),("Clubs","J"),("Clubs","Q"),("Clubs","K"),
("Clubs","A"),("Diamonds","2"),("Diamonds","3"),("Diamonds","4"),("Diamonds","5"),
("Diamonds","6"),("Diamonds","7"),("Diamonds","8"),("Diamonds","9"),("Diamonds","10"),
("Diamonds","J"),("Diamonds","Q"),("Diamonds","K"),("Diamonds","A"),("Hearts","2"),
("Hearts","3"),("Hearts","4"),("Hearts","5"),("Hearts","6"),("Hearts","7"),("Hearts","8"),
("Hearts","9"),("Hearts","10"),("Hearts","J"),("Hearts","Q"),("Hearts","K"),("Hearts","A"),
("Spades","2"),("Spades","3"),("Spades","4"),("Spades","5"),("Spades","6"),("Spades","7"),
("Spades","8"),("Spades","9"),("Spades","10"),("Spades","J"),("Spades","Q"),("Spades","K"),
("Spades","A")]
```

  **Answer:** `deck = [(suit,card) | suit <- suits, card <- (cards ++ honours)]`

- Use a list comprehension and the function `zip` to write a Haskell function `locations n xs` that returns the list of all indexes `i` such that the `i`th element of `xs` is `n` (i.e. `xs!!i == n`). Note that the first elelement of a list has index 0. You are not allowed to use the indexing operator `!!`.
  Example: `locations 0 [x `mod` 10 | x <- [1..50]]` should yield `[9,19,29,39,49]`.

  **Answer:** `locations x xs = [i-1 | (y,i) <- zip xs [1..length xs], y == x]`

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

- The function `iterate` creates an infinite list where the first item is calculated by applying the function its first argument on its second argument, the second item by applying the function on the previous result and so on. For example, `iterate (2*) 1` yields the infinite list `[2,4,8,16,32,64,128,256,512,...]`. Give a Haskell implementation (including its type) of the function `iterate`.
  **Answer:**

```
iterate :: (a -> a) -> a -> [a]
iterate f x = f x : iterate f (f x)
```

- Define the infinite list `ints`, which is the list of all integers. It should be ordered in such a way that you can find any given integer after searching a finite number of elements in `ints`. In other words, this is not going to work: `ints = [0..] ++ [-1, -2..]`
  **Answer:**

```
ints :: [Integer]
ints = 0 : intsfrom 1
  where intsfrom n = n : (-n) : intsfrom (n+1)
```

- Given is the infinite list `primes` of prime numbers. Use `primes` to define the infinite list `composites` of non-primes. So, `take 10 composites` should yield `[4,6,8,9,10,12,14,15,16,18]`. Note that we skip the value 1.
  **Answer:**

```
union (x:xs) (y:ys)
  | x < y     = x : union xs (y:ys)
  | y < x     = y : union (x:xs) ys
  | otherwise = x : union xs ys

composites = foldr fuse [] (map multiples primes)
  where
    multiples p    = [ n*p | n <- [2..]]
    fuse (x:xs) ys = x : (union xs ys)
```

5. (10 points) We are used to write expressions using *infix* notation. For instance, we write `10 - (4 + 3) * 2`. The downside of this notation is that we have to use parentheses to denote precedence. *Reverse Polish Notation* (RPN) is another way of writing down expressions, and does not need parentheses. In RPN, every operator follows its operands, therefore RPN is also called *postfix notation*. The above expression in RPN is: `10 4 3 + 2 * -`
Evaluating such an expression goes as follows. We keep pushing numbers onto a stack, until we encounter the first operator. So, when we encounter the +, the stack contains [3, 4, 10] (here, the head of the list is the top of the stack). We replace the two top numbers from the stack by their sum. The stack is now [7, 10]. Next, we push 2 on the stack (so, [2, 7, 10]). Now, we encounter an operator again, we pop 2 and 7 off the stack, apply the operator and push the result to the stack yielding [14, 10]. Finally, there is a −. We pop 10 and 14 from the stack, subtract 14 from 10 and push that back. The number on the stack is now -4, which is the final result.

   We use the following data type for representing RPN literals:

```
data RPN = Value Integer | Plus | Minus | Times |Div
```

   Write a Haskell funtion `rpn :: [RPN] -> Integer` that evaluates an RPN expression to an `Integer`.

   **Answer:**

```
rpn :: [RPN] -> Integer
rpn xss = eval xss []
  where
    eval [] stack                 = head stack
    eval ((Value n):expr) stack   = eval expr (n:stack)
    eval (Plus:expr) (x:y:stack)  = eval expr ((y+x):stack)
    eval (Minus:expr) (x:y:stack) = eval expr ((y-x):stack)
    eval (Times:expr) (x:y:stack) = eval expr ((y*x):stack)
    eval (Div:expr) (x:y:stack)   = eval expr ((y `div` x):stack)
```

6. (15 points) The abstract data type (ADT) `Fifo tp` implements a simple data type for the storage of elements of the type `tp`, from which elements are retrieved in the same order as in which they are inserted: FIFO stands for *First In First Out queue*.

   Implement a `module Fifo` such that the concrete implementation of the type Fifo is hidden from the user.

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

   - `empty` returns an empty queue.
   - `isEmpty` returns `True` for an empty queue, otherwise `False`.
   - `insert`: returns the queue that is the result of inserting an element.
   - `top`: returns the 'oldest' element of the queue.
   - `remove`: returns the queue that is obtained by removing the 'oldest' element.

   **Answer:**

```
module Fifo (Fifo, empty, isEmpty, insert, top, remove) where

data Fifo a = F [a]

empty            = F []
isEmpty (F xs)   = ([] == xs)
insert x (F xs) = F(xs ++ [x])
top (F xs)       = head xs
remove (F xs)    = F (tail xs)
```

7. (10 points) Given is the data type `Tree`:

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

Given are the functions `leaves` and `nodes`:

```
leaves (Leaf _)     = 1
leaves (Node a l r) = leaves l + leaves r
nodes  (Leaf _)     = 0
nodes  (Node a l r) = 1 + nodes l + nodes r
```

Prove for all finite trees `t`: `leaves t = nodes t + 1`

**Answer:** Using induction, this proof is straightforward:

```
Base: leaves (Leaf n) = nodes (Leaf n) + 1
  Proof: leaves (Leaf n) = 1 = 0 + 1 = nodes (Leaf n) + 1

Induction: prove leaves (Node a l r) = nodes (Node a l r) + 1 given
  IH1: leaves l = nodes l + 1    and   IH2: leaves r = nodes r + 1

  Proof:  leaves (Node a l r)
        = {def. leaves}
          leaves l + leaves r + 1
        = {ind. hyp. IH1 and IH 2}
          nodes l + 1 + nodes r + 1
        = {assoc. +}
          (1 + nodes l + nodes r) + 1
        = {def. nodes}
          nodes (Node a l r) + 1
   QED.
```

8. (15 points) Given are the definition of the functions `rev1`, `shunt`, and `rev2`:

```
rev1 [] = []
rev1 (x:xs) = (rev1 xs) ++ [x]

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

rev2 xs = shunt xs []
```

Prove that `rev1 xs = rev2 xs` for all finite lists `xs`.

**Answer:** If you try to prove the claim directly, you will notice that you get stuck quickly. We first need to prove a lemma: `shunt xs ys = (rev1 xs) ++ ys` for all finite lists `xs` and `ys`. The proof of this lemma is straightforward:

```
Base: prove shunt [] ys = (rev1 []) ++ ys
  Proof: shunt [] ys = ys = [] ++ ys = (rev1 []) ++ ys

Induction: prove shunt (x:xs) ys = (rev1 (x:xs)) ++ ys given lemma is true for xs

  Proof:  shunt (x:xs) ys                (rev1 (x:xs)) ++ ys
        = {def. shunt}                 = {def. rev1}
          shunt xs (x:ys)                ((rev1 xs) ++ [x]) ++ ys
        = {ind. hyp.}                  = {assoc. ++}
          (rev1 xs) ++ (x:ys)            (rev1 xs) ++ ([x] ++ ys)
                                       = {def. ++, note that [x]=x:[]}
                                         (rev1 xs) ++ (x:([] ++ ys))
                                       = {def. ++}
                                         (rev1 xs) ++ (x:ys)
   QED.
```

Given the lemma, the claim is easily proved: `rev2 xs = shunt xs [] = (rev1 xs) ++ [] = rev1 xs`