Exam Functional Programming – November 5th 2018

Name	
Student number	
I study CS/AI/Other	

- Write **neatly** and carefully. Use a pen (no pencil!) with black or blue ink.
- Write your answers in the answer boxes. If you need more space, use back side of the sheet and make a reference to it.
- You can score 90 points. You get 10 points for free, yielding a maximum of 100 points in total. Your exam grade is calculated as the number of obtained points divided by 10.

You may use the following standard Haskell functions throughout the entire exam:

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

concat []                = []
concat (xs:xss)          = xs ++ concat xss

map f []                 = []
map f (x:xs)             = f x : map f xs

filter p xs              = [x | x <- xs, p x]

foldr f z []             = z
foldr f z (x:xs)         = f x (foldr f z xs)

sum []                   = 0
sum (x:xs)               = x + sum xs

reverse []               = []
reverse (x:xs)           = reverse xs ++ [x]

head (x:xs)              = x
tail (x:xs)              = xs

length []                = 0
length (x:xs)            = 1 + length xs

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

(f . g) x                = f (g x)

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

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

1. **Types** ($5\times$ 2=10 points)

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

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

```
NO
```

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

```
(True:[]):[]
```

```
YES
[[Bool]]
```

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

```
(True:[]):[]++[False]
```

```
NO
```

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

```
(True:[]):[]++[[False]]
```

```
YES
[[Bool]]
```

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

```
f = map.filter
```

```
(a->Bool) -> [[a]] -> [[a]]
```

2. **Programming in Haskell** (10 points)
This problem is about pattern matching. A *pattern* is a `String` that specifies (describes) the strings that match the pattern. A pattern may only consist of lower case letters from the alphabet (i.e. `a..z`), asterisks (i.e. the `*` character), and question marks (i.e. the `?` character). A question mark may only follow a letter and indicates zero or one occurrence of the preceding character. For example, `colou?r` matches both `color` and `colour`. An asterisk may only follow a letter and indicates zero or more occurrences of the preceding character. For example, `ab*c` matches `ac`, `abc`, `abbc`, `abbbc`, and so on.

Write a Haskell function `isMatch :: String -> String -> Bool` such that `isMatch pat str` return `True` if and only the string `str` can be produced by the pattern `pat`. For example, `isMatch "h?i?el*o?" "hello"` should return `True`, while `isMatch "h?iel*" "ill"` should return `False`.

```
isMatch :: String -> String -> Bool
isMatch [] str                    = str == []
isMatch (a:'?':pattern) []        = isMatch pattern []
isMatch (a:'?':pattern) (b:str) = isMatch pattern (b:str)
                                    || (a==b && isMatch pattern str)
isMatch (a:'*':pattern) []        = isMatch pattern []
isMatch (a:'*':pattern) (b:str) = isMatch pattern (b:str)
                                    || (a==b && isMatch (a:'*':pattern) str)
isMatch (a:pattern) (b:str)      = (a==b) && isMatch pattern str
isMatch _ _                       = False
```

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

- Use the higher-order function `foldr` to implement the function `factorial` (including its type) which takes a non-negative integer `n` and return the factorial of `n` (i.e. `n*(n-1)*(n-2)*...*1`). So, `factorial 5` should return `120`.

```
factorial :: Integer -> Integer
factorial n = foldr (*) 1  [1..n]
```

- The higher order function `foldr` is used for reducing a list as in the following example:

  `foldr f 0 [1..5] = f 1 (f 2 (f 3 (f 4 (f 5 0))))))`

  Implement the 'mirror' operation `foldl` (including its type) such that

  `foldl f 0 [1..5] = f (f (f (f (f 0 1) 2) 3) 4) 5`

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f z []     = z
foldl f z (x:xs) = foldl f (f z x) xs
```

- Using function composition (i.e. `'.'`), `foldr` and the cons operator (i.e. `':'`) to implement the function `folmap` (including its type), which is your version of the standard function `map`. So, `folmap (*2) [1,2,3,4]` should yield `[2,4,6,8]`.

```
folmap :: (a -> b) -> [a] -> [b]
folmap f = foldr ((:) . f) []
```

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

- Implement the function `pairs` (including its type) using a list comprehension. The call `pairs [1..3] ['a','b']` should return `[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]`.

```
pairs    :: [a] -> [b] -> [(a,b)]
pairs xs ys = [(x,y) | x<-xs, y<-ys]
```

- Use a list comprehension to implement the function `locations` (including its type) that takes a value of some type and a list of that type, and returns a list with locations (indexes starting from zero) where the value occurs in the list. For example, `locations 1 [1,0,1,0,4,1]` should return `[0,2,5]`.

```
locations :: a -> [a] -> [Int]
locations x xs = [i | (a,i) <- zip xs [0..], a==x]
```

- The function `sumProdPairs = zipWith (\x y -> (x+y,x*y))` is defined using the function `zipWith`. Give an equivalent definition of `sumProdPairs` that uses a list comprehension instead.

```
sumProdPairs :: [Integer] -> [Integer] -> [(Integer,Integer)]
sumProdPairs xs ys = [(x+y,x*y) | (x,y) <- zip xs ys]
```

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

- Give a definition of the Haskell function `repeat` (including its type) that takes an argument and produces the list that indefinitely repeats that argument. So, `repeat 42=[42,42,42,42,42,42,....]`.

```
repeat :: a -> [a]
repeat a = a:repeat a
```

- Give a definition of the infnite list `binaries` which is the list of all non-empty lists containing zeros and ones. The order of the elements of the list should be as in the following example: `take 14 binaries` should return `[[0],[1],[0,0],[1,0],[0,1],[1,1],[0,0,0],[1,0,0],[0,1,0],[1,1,0],[0,0,1],[1,0,1], [0,1,1],[1,1,1]]`.

```
binaries = [0]:[1]:[bit:binary | binary <- binaries, bit <- [0,1]]
```

- Consider $(x+1)^n$, for integer $n \geq 0$. We can write this in coefficient normal for, i.e. in the form $a_n x^n + a_{n-1}x^{n-1} + .. + a_0$. For example, $(x+1)^4 = x^4 + 4x^3 + 6x^2 + 4x + 1$, yielding the list of coefficients `[1,4,6,4,1]`. Give a definition of the infinite list `coefficients` of lists of coefficients, such that the nth list corresponds with the coefficients of $(x+1)^n$.
  For example, `take 5 coefficients` should produce `[[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]`.

```
coefficients = [1] : map next coefficients
  where next xs = (1:zipWith (+) xs (tail xs))++[1]
```

6. (15 points) The type `Peano` is an Abstract Data Type (ADT) for implementing natural numbers as follows:

- `Zero` is a constructor that represents the natural number 0.
- `Succ n`, where `n` is of the type `Peano`, represents the number that is 1 greater than the number that `n` represents.

Implement a `module Peano` such that the concrete implementation of the type `Peano` is hidden to the user.

The following operations on Peano numbers need to be implemented:

- `peanoToInteger n` converts the Peano number `n` into its decimal `Integer` value.
- `isZero n` returns `True` if and only if the peano Number `n` represents 0.
- `isLessThan a b`: returns `True` if and only if the Peano number `a` is less than the Peano number `b`.
- `plus a b`: returns the Peano representation of adding the Peano numbers `a` and `b`.
- `mul a b`: returns the Peano representation of multiplying the Peano numbers `a` and `b`.

```
module Peano(Peano, peanoToInteger, plus, mul, isZero, isLessThan) where

data Peano = Zero | Succ Peano

peanoToInteger :: Peano -> Integer
peanoToInteger Zero = 0
peanoToInteger (Succ n) = 1 + peanoToInteger n

plus :: Peano -> Peano -> Peano
plus Zero b = b
plus (Succ a) b = Succ (plus a b)

mul :: Peano -> Peano -> Peano
mul Zero n = Zero
mul n Zero = Zero
mul (Succ n) m = plus m (mul n m)

isZero :: Peano -> Bool
isZero Zero  = True
isZero _     = False

isLessThan :: Peano -> Peano -> Bool
isLessThan Zero Zero = False
isLessThan Zero n = True
isLessThan n Zero = False
isLessThan (Succ m) (Succ n) = isLessThan m n
```

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

```
f 0 = 0
f 1 = 1
f n = f (n-1) + f (n-2)

g 0 a b = a
g n a b = g (n-1) b (a+b)
```

Prove that `f n = g n 0 1` for all non-negative integers `n`.

```
If you try to prove the property directly, then you'll find out that the claim
is too specific (due to the values 0 and 1). So, we need a more general lemma.
Lemma: for n>1 we have g n a b = g (n-1) a b + g (n-2) a b

Proof by induction on n:
* Base case n=2: lhs = g 2 a b = g 1 b (a+b) = g 0 (a+b) (a+2*b) = a+b
*                rhs = g 1 a b + g 0 a b = g 0 b (a+b) + g 0 a b = b+a = a+b
*                So, lhs=rhs.
* Inductive case: assume g n a b = g (n-1) a b + g (n-2) a b
*   g (n+1) a b = lhs                     g n a b + g (n-1) a b = rhs
* = {def. g}                            = {def. g}
*   g n b (a+b)                           g (n-1) b (a+b) + g (n-2) b (a+b)
* = {Ind. Hypothesis}
*   g (n-1) b (a+b) + g (n-2) b (a+b)
* So, lhs=rhs and we completed the proof of the lemma.

Next, we prove the property: f n = g n 0 1 using induction on n.
* Base case n = 0: f 0 = 0 = g 0 0 1
* Base case n = 1: f 1 = 1 = g 0 1 1 = g 1 0 1
* Inductive case: assume f n = g n 0 1
*   f (n+1) = lhs                     g (n+1) 0 1 = rhs
* = {def. f}                        = {Lemma}
*   f n + f (n-1)                     g n 0 1 + g (n-1) 0 1
* = {Ind. Hypothesis}
*   g n 0 1 + g (n-1) 0 1
* So, lhs=rhs and we completed the proof of the property.
```

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

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

foldT :: (a->a->a) -> a -> Tree a -> a
foldT f z Empty = z
foldT f z (Node x l r)  = f (f (foldT f z l) x) (foldT f z r)

mapT :: (a -> b) -> Tree a -> Tree b
mapT f Empty = Empty
mapT f (Node x t1 t2) = Node (f x) (mapT f t1) (mapT f t2)

inorder :: Tree a -> [a]
inorder Empty = []
inorder (Node x l r) = inorder l ++ [x] ++ inorder r
```

Let `f::a->a->a` be an associative function (i.e. `f a (f b c)=f (f a b) c`) with identity element `z` such that `f x z=f z x=x`.
Prove for all finite trees `t`: `foldT f z t = foldr f z (inorder t)`
[Note: You may use that the operator ++ is associative without giving a proof.]

```
We prove this by structural induction on t.
* Base case t=Empty: lhs = foldT f z Empty = z
*                    rhs = foldr f z (inorder Empty) = foldr f z [] = z
*                    So, lhs=rhs.
*
* Inductive case: IH1:  foldT f z l = foldr f z (inorder l)
*                 IH2:  foldT f z r = foldr f z (inorder r)
*    foldT f z (Node x l r) = lhs
*  = {def. foldT}
*    f (f (foldT f z l) x) (foldT f z r)
*
*    foldr f z (inorder (Node x l r)) = rhs
*  = {def. inorder}
*    foldr f z (inorder l ++ [x] ++ inorder r)
*  = {associativity ++}
*    foldr f z ((inorder l ++ [x]) ++ inorder r)
*  = {lemma: foldr f z (xs++ys) = f (foldr f z xs) (foldr f z ys)}
*    f (foldr f z (inorder l ++ [x])) (foldr f z (inorder r))
*  = {same lemma once more}
*    f (f (foldr f z (inorder l) (foldr f z [x]))) (foldr f z (inorder r))
*  = {IH1 and IH 2}
*    f (f (foldT f z l) (foldr f z [x]))) (foldT f z r)
*  = {foldr f z [x] = foldr f z (x:[]) = f x (foldr z []) = f x z = x}
*    f (f (foldT f z l) x) (foldT f z r)
* So, lhs=rhs.

What remains to be done is to prove the lemma (using induction on xs).
* Lemma: foldr f z (xs++ys) = f (foldr f z xs) (foldr f z ys)
* Base case xs=[]: lhs = foldr f z ([]++ys) = foldr f z ys
*                  rhs = f (foldr f z []) (foldr f z ys)
                       = f z (foldr f z ys) = foldr f z ys
* Inductive case: IH: foldr f z (xs++ys) = f (foldr f z xs) (foldr f z ys)
*    foldr f z ((x:xs)++ys) = lhs           f (foldr f z (x:xs) (foldr f z ys)) = rhs
*  = {def. ++}                            = {def. foldr}
*    foldr f z (x:(xs++ys))                 f (f x (foldr f z xs)) (foldr f z ys))
*  = {def. foldr}                         = {associativity f}
*    f x (foldr f z (xs++ys))               f x (f (foldr f z xs) (foldr f z ys))
*  = {IH}
*    f x (f (foldr f z xs) (folder f z ys))
* So, again lhs=rhs. This concludes the proof.
```