Exam Functional Programming – November 7th 2016

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 the 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.
- If you need auxiliary lemmas in a proof, then prove the validity of these lemmas as well.

You may use throughout the entire exam the following functions and lemmas:

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

map f xs                = [f x | x <- 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  xs     ys          = []

zipwith f xs ys         = [f x y | (x,y) <- zip xs ys]

-- Lemma associativity of ++ (may be used without proof):
-- (xs ++ ys) ++ zs = xs ++ (ys ++ zs) = xs ++ ys ++ zs

-- Lemma concatenation with []  (may be used without proof):
-- xs ++ [] = xs
```

1. **Types** (5× 2=10 points)
   (a) What is the type of the following expression?

   ```
   (42,[42],[[42]])
   ```

   ```
   (Num a, Num b, Num c) => (a, [b], [[c]])

   Note that (Int,[Int],[[Int]]) is accepted as a valid answer.
   ```

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

   ```
   f = filter (== 'A')
   ```

   ```
   [Char] -> [Char]
   ```

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

   ```
   g = (\x -> (\y -> (y,x)))
   ```

   ```
   a -> b -> (b,a)
   ```

   (d) What is the type of the function `foldr`?

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

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

   ```
   h =(\f -> map f "Text" == [1,2,3,4])
   ```

   ```
   Num a => (Char -> a) -> Bool

   Note that (Char -> Int) -> Bool is accepted as a valid answer.
   ```

2. **Programming in Haskell** (10 points)
   The increasing list `[1,2,3,4,5]` has 9 non-empty increasing sublists that contain as many even numbers as odd numbers.

   Write a Haskell function `balancedSublists` (including its type) that takes an increasing list and returns the list of its non-empty increasing sublists that have as many even numbers as odd numbers. The order of the sublists is irrelevant.

   For example, `balancedSubLists [1,2,3,4,5]` may return the list
   `[[4,5],[3,4],[2,5],[2,3],[2,3,4,5],[1,4],[1,2],[1,2,4,5],[1,2,3,4]]`.

   ```
   balancedSublists :: [Int] -> [[Int]]
   balancedSublists xs = filter (/= []) (balsub xs 0 [])
     where  {- note that balance = #even - #odd -}
       balsub [] balance ys = if balance == 0 then [reverse ys] else []
       balsub (x:xs) balance ys
         | even x = balsub xs (balance+1) (x:ys) ++ balsub xs balance ys
         | otherwise = balsub xs (balance-1) (x:ys) ++ balsub xs balance ys
   ```

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

- Write a function `isEqual` (including its type) that accepts three arguments: the first two arguments are functions (both having the same type), which can be applied to each element of a list (the third argument). The function should return `True` if and only if applying both functions to each element of the third argument yields the same result.
  For example, `isEqual (+1) (1+) [1,2,3]` should yield `True`, while `isEqual (^2) (2^) [1,2,3]` should yield `False`. Your are not allowed to use recursion.

```
isEqual :: Eq b => (a -> b) -> (a -> b) -> [a] -> Bool
isEqual f g xs = (map f xs) == (map g xs)
```

- The function `concat` concatenates the elements of a list of lists. For example, `concat [[1,2],[3],[4,2,3]]` yields the list `[1,2,3,4,2,3]`. Give an implementation of the function `concat` (including its type) using `foldr`.

```
concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss
```

- Write a function `mulinceven` (including its type) that takes a list of `Integer`s, and returns the product of one plus every number in the input that is at least 4. For example, `mulinceven [7,3,2,4,5]` returns `240`, because `(7+1)*(4+1)*(5+1)=240`. Your implementation must make use of `map`, `filter`, and `foldr`.

```
mulinceven :: [Integer] -> Integer
mulinceven xs = foldr (*) 1 (map (+1) (filter (>=4) xs))
```

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

- Write a function `oddeven` (including its type) that takes a list of pairs and returns a list containing the first element from each of the pairs in even-numbered positions and the second element from each of the pairs in odd-numbered positions, where numbering of list elements begins from 0.
  For example, `oddeven [(1,2),(3,4),(5,6),(7,8)]` should return the list `[1,4,5,8]`. Another example is `oddeven [("hello","world"),("from","Venus")]` which should return `["hello", "Venus"]`. The implementation off `oddeven` must be a list comprehension.

```
oddeven :: [(a,a)] -> [a]
oddeven xs = [ if i `mod` 2 == 0 then a else b | (i,(a,b)) <- zip [0..] xs]
```

- Write a function `removeRepetition` (including its type) that removes all but one occurrence of consecutive repeated elements from its input list.
  For example, `removeRepetition [1,2,2,3,3,3,4,5,1,1]` should return `[1,2,3,4,5,1]`. Another example is `removeRepetition "Haaassskkkell"` which should return `"Haskel"`. The definition of the function `removeRepetition` must make use of a list comprehension.

```
removeRepetition :: Eq a => [a] -> [a]
removeRepetition [] = []
removeRepetition (c:cs) = c:[ b | (a,b) <- zip (c:cs) cs, a /= b]
```

- Write a function `sublists` (including its type) that takes a list and returns the list of all its possible sublists (the order of the sublists is irrelevant). Use a list comprehension in combination with recursion.
  For example, `sublists [1,2,3]` may return `[[],[1],[2],[3],[1,2],[1,3],[2, 3],[1,2,3]]`.

```
sublists :: [a] -> [[a]]
sublists [] = [[]]
sublists (x:xs) = sublists xs ++ [x:sublist | sublist <- sublists xs]
```

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

- Given is the infinite list of primes `primes :: [Integer]` that is produced by the following code:

  ```
  primes = sieve [2..] where sieve (p:xs) = p:sieve [x|x <- xs, x `mod` p > 0]
  ```

  Write a function `isPrime` such that `isPrime n` returns `True` if and only if `n` is in the list `primes`.

  ```
  isPrime :: Integer -> Bool
  isPrime n = n == head(dropWhile (<n) primes)
  ```

- The infinite list `ones` is defined as `ones = 1:ones`.
  Use only `ones`, arithmetic operators, and `zipWith` to create two mutually recursive definitions of the infinite lists `evens` and `odds`, where `evens=[0,2,4,6,8,..]` and `odds=[1,3,5,7,9,..]`. Mutual recursive means that `evens` (but not `odds`) can appear in the definition of `odds` and `odds` (but not `evens`) can appear in the definition of `evens`.

  ```
  evens = 0 :  zipWith (+) odds ones
  odds = zipWith (+) evens ones
  ```

- Define the function `multiples :: [Integer] -> [Integer]`, that takes a finite list of `Integer`s and produces the infinite sorted list (without repetitions) of all multiples of the numbers in the input list.
  For example, `take 10 (multiples [2,3,5])` should return `[0,2,3,4,5,6,8,9,10,12]`.

  ```
  multiples :: [Integer] -> [Integer]
  multiples xs = foldr merge [] [[0,x..] | x <- xs]
    where
      merge xs []   = xs
      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 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.

```haskell
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)
```

7. **Proof on lists** (10 points) Given is the recursive definition of the function `drop`:

```
drop :: Int -> [a] -> [a]
drop 0 xs     = xs
drop n []     = []
drop n (x:xs) = drop (n-1) xs
```

Prove that `drop m (drop n xs) = drop (m+n) xs` for all finite lists `xs` and $\mathtt{m}, \mathtt{n} \geq 0$.

```
Base case: xs=[]
    drop m (drop n [])            drop (m+n) []
  = { def. drop }               = { def. drop }
    drop m []                     []
  = { def. drop }
    []

Ind. case:  x:xs
  For the case n=0, the proof is trivial:
    drop m (drop n xs) = drop m xs = drop (m+0) xs
  Next, we consider the case n>0:
    drop m (drop n (x:xs))          drop (m+n) (x:xs)
  = { def. drop, n>0 }            = {def. drop, n>0 }
    drop m (drop (n-1) xs)          drop (m+n-1) xs
  = { induction }
    drop (m+n-1) xs               QED.
```

8. **Proof on trees** (15 points) Given is the data type `Tree` and the functions `lrorder`, `rlorder`, and `mirror`:

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

mirror :: BinTree a -> BinTree a
mirror Empty = Empty
mirror (Node x l r) = Node x (mirror r) (mirror l)

lrorder, rlorder :: BinTree a -> [a]
lrorder Empty = []
lrorder (Node x l r) = lrorder l ++ [x] ++ lrorder r

rlorder Empty = []
rlorder (Node x l r) = rlorder r ++ [x] ++ rlorder l
```

Prove for all finite trees `t`: `lrorder t = rlorder (mirror t)`

```
Base case: t=Empty
    lrorder Empty             rlorder(mirror Empty)
  = { def. lrorder}         = { def. mirror }
    []                        rlorder Empty
                            = { def. rlorder }
                              []

Ind. case: t=Node x l r
    lrorder (Node x l r)          rlorder (mirror (Node x l r))
  = { def. lrorder }            = { def. mirror }
    lrorder l++[x]++lrorder r     rlorder (Node x (mirror r) (mirror l))
                                = { def. rlorder }
                                  rlorder (mirror l) ++ [x] ++ rlorder (mirror r)
                                = { ind. hypothesis twice }
                                  lrorder l ++ [x] ++ lrorder r
```