Lund University
Department of Computer Science

EDAN40: Functional Programming
12th January 2016, 14–19

Exam

1. Given the following typeclass definition:

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)         :: a -> a -> a
    negate, abs, signum :: a -> a
    fromInteger          :: Integer -> a
```

   and given the following definition of type `ListNatural`:

```
data ListNatural = Empty | () :-: ListNatural
    deriving (Eq, Show)
```

   so that e.g.:

```
twoL = () :-: () :-: Empty
threeL = () :-: () :-: () :-: Empty
-- (or: threeL = () :-: twoL)
```

   consider the following functions:

```
f1 Empty y = y
f1 (() :-: x) y = () :-: (f1 x y)

f2 Empty y = Empty
f2 (() :-: x) y = f1 y (f2 x y)

f3 x Empty = x
f3 Empty x = error "foo"
f3 (() :-: x) (() :-: y) = f3 x y
```

   and make the following definition complete:

```
instance Num ListNatural where
    ...
```

   Define appropriate auxiliary functions, if necessary.

2. Rewrite the following two definitions into a point-free form (i.e., `f = ...`, `g = ...`), using neither lambda-expressions nor list comprehensions nor enumeration nor `where` clause nor `let` clause:

```
f x y = (2 + x) * y
g x y = y x
```

3. - Write a definition of a function `tffos`, first using list comprehension, and then using `filter` and `map`.

     ```
     tffos :: [(Int,Int)] -> [Int]
     ```

     `tffos` is a function taking a list of integer pairs and returning a list of triplicated first elements of those pairs, in which the second elements are odd (`tffos` = triplicated firsts for even seconds:-)
   - Use a fold (which one?) to define

     ```
     reverse :: [a] -> [a]
     ```

     which returns a list with the elements in reverse order.
   - Write, using list comprehension syntax, a single function definition (try to avoid `if`, `case` and similar constructs) with signature `[[Int]] -> [[Int]]`, which, from a list of lists of `Int`, returns a list of the tails of those lists using, as filtering condition, that the head of each `[Int]` must be larger than 5. Also, your function must not trigger an error when it meets an empty `[Int]`, but rather silently skip such an entry.

4. What is the type and value of the following expression:

   ```
   do [1, 2, 3, 4]; "curry"
   ```

   What would be the answer in the following case:

   ```
   do [1, 2, 3, 4]; return "uncurry"
   ```

   Please explain both answers.

5. Explain the concept of *lazy evaluation* in Haskell. What are its consequences?

6. Give the types of these three expressions. Also explain what each of the expressions mean.

   ```
   zipWith map
   map zipWith
   map.zipWith
   ```

Good Luck!