Lund University
Department of Computer Science

EDAN40: Functional Programming
3rd June 2019, 14–19

Exam

1. **Proving program properties (1p)**

   The `Functor` class is defined as follows:

   ```
   class Functor f where
     fmap :: (a -> b) -> f a -> f b
   ```

   It is mandatory that all instances of `Functor` should obey:

   ```
   fmap id      = id
   fmap (p . q) = (fmap p) . (fmap q)
   ```

   Assume the following definition of lists as a functor instance:

   ```
   instance Functor [] where
     fmap g []     = []
     fmap g (x:xs) = [g x] ++ (fmap g xs)
   ```

   Is this a correct definition of a functor instance? Why or why not? **Prove your claim**.

2. **Types and type classes (2p)**

   - (0.3p) Define a tree data structure so that the trees are ternary (i.e., each node has either exactly three children or is a leaf) and store strings in each node.
   - (0.3p) Generalize your definition so that your ternary trees can contain objects of an arbitrary predetermined type in a node.
   - (0,7p) Assuming your polymorphic trees type is denoted by `Tree3 a`, write all necessary code so that the following function is correct:

     ```
     myLength :: Tree3 String -> Tree3 Integer
     myLength = fmap length
     ```

     and yelds an (obviously ternary) tree with nodes containing lengths of the strings placed in the respective nodes of the argument tree. Your solution must contain the word `instance` to get full credit.
   - (0,7p) write all necessary code so that you can compare two ternary trees for equality using the `==` operator.

3. **Point-free notation (1p)**

   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 = (3 - y) * x
g x y = map x $ filter (<3) y
```

Hint: you may find `flip` useful.

4. **Sparks (1p)**

   Consider the following code parallelizing the quicksort algorithm.

```
-- file: ch24/Sorting.hs
import Control.Parallel (par, pseq)
parSort (x:xs)   = greater `par` (lesser `pseq`
                                  (lesser ++ x:greater))
  where lesser   = parSort [y | y <- xs, y <  x]
        greater  = parSort [y | y <- xs, y >= x]
parSort _        = []
```

   It will be almost as (in)efficient as the sequential version:

```
-- file: ch24/Sorting.hs
sort :: (Ord a) => [a] -> [a]
sort (x:xs) = lesser ++ x:greater
    where lesser  = sort [y | y <- xs, y <  x]
          greater = sort [y | y <- xs, y >= x]
sort _ = []
```

   Why?

5. **Parsing (1p)**

   This is an excerpt from the assignment N3 code:

```
assignment = word #- accept ":=" # Expr.parse
                  #- require ";" >-> buildAss
buildAss (v, e) = Assignment v e
```

   - (0,5p) Provide types for all functions and operators named here.
   - (0,5p) Rewrite `assignment` using `do`-notation.

Good Luck!