Functional Programming – Mid-term exam – Tuesday 1/10/2019

Name:	
Student number:	

Q:	1	2	3	4	5	Total
P:	22	15	19	16	18	90
S:						

Before you begin:

- Do not forget to write down your name and student number above.
- If necessary, explain your answers *in English or Dutch*.
- Use *only* the empty boxes under the questions to write your answer and explanations in.
- The exam consists of *five* (5) questions in *seven* (7) pages.
- At the end of the exam, only hand in the filled-in exam paper. Use the blank paper provided with this exam only as scratch paper (kladpapier).
- Answers will not only be judged for correctness, but also for clarity and conciseness.

In any of the answers below you may (but do not have to) use the following well-known Haskell functions and operators, unless stated otherwise: `id`, `(.)`, `const`, `flip`, `head`, `tail`, `(++)`, `concat`, `foldr` (and its variants), `map`, `filter`, `sum`, `all`, `max`, `min`, `any`, `elem`, `not`, `(&&)`, `(||)`, `zip`, `reverse`, `take`, `drop`, and all the members of the type classes `Show`, `Eq`, `Ord`, `Enum` and `Num`.

1. In this question, we implement a simple sorting algorithm.

   (a) (4 points) The function `merge :: Ord a => [a] -> [a] -> [a]` merges two sorted lists to give a single sorted list. For example:

   ```
   > merge [2, 5, 6] [1, 3, 4, 5]
   [1, 2, 3, 4, 5, 5, 6]
   ```

   Give a definition of `merge` using direct recursion.

   (b) (3 points) Recall that `take n xs` returns the first `n` elements of `xs`. For example, `take 2 [3, 4, 5] == [3, 4]`. We can similarly define `drop n xs` such that it returns `xs` without its first `n` elements. For example, by defining

   ```
   drop n xs = let m = length xs in (reverse . take (m - n) . reverse) xs
   ```

   The function `halve :: [a] -> ([a], [a])` has the specification that `halve xs = (ls,rs)` implies that `length ls <= length rs && length rs <= (length xs + 1 )‘div‘ 2 && ls ++ rs == xs`. Use `take` and `drop` to implement this function.

(c) (6 points) Using `merge` and `halve` and direct recursion, please give an implementation of a sorting algorithm `mergeSort :: Ord a => [a] -> [a]`.

(d) (2 points) Given the data type

```
data Pair a b = MkPair a b
```

please define functions `getA` and `getB` that, respectively, get the values of type `a` and `b` stored in a `Pair a b`.

(e) (2 points) Define an `Eq`-instance for `Pair a b` in which two pairs are considered equal if and only if they have the same `b` values. Furthermore, define an `Ord`-instance for `Pair a b` in which the order of two pairs is defined as the order of their respective `b` values.

(f) (5 points) Define a function `sortOn :: Ord b => (a -> b) -> [a] -> [a]` such that `sortOn f xs` sorts the `x` in `xs` according to the value of `f x` in increasing order of `b`. In case two elements have equal `f x` value, you may sort these two elements in any order. For example,

```
> sortOn length ["Never", "gonna", "give", "you", "up"]
["up","you","give","Never","gonna"]
```

Please use `mergeSort`, higher-order functions and the `Pair a b` data type defined above.

2. This question will test your command of higher-order functions and, in particular, folds.

   (a) (4 points) `any p xs` computes whether any element of `xs` satisfies the predicate `p`. Please complete the following definition:

   `any :: (a -> Bool) -> [a] -> Bool` using a fold:

   ```
   any p = foldr f x0 where
     f  = ...
     x0 = ...
   ```

   You may add extra arguments to `f` and `x0` if you wish.

   (b) (2 points) Please give a definition, using direct recursion, of the function `partition :: (a -> Bool) -> [a] -> ([a], [a])` which has the definition that `partition p xs = (trues, falses)` where `trues` contains exactly the elements of `xs` which satisfy the predicate `p` and `falses` contains the remaining elements of `xs` (both in their original order).

(c) (5 points) Now, please implement `partition` using a fold, instead (no direct recursion allowed).

(d) (4 points) The function `maximum' :: Ord a => a -> [a] -> a` has the specification that `maximum' x0 xs` calculates the maximum value of value of `x0` and the elements in `xs`. Please complete the following definition of `maximum'` by using its first argument as an accumulator:

```
maximum' x0 [] = x0
```

3. In this question, we will be working with a polymorphic data type

```
data WTree a = Leaf Int
             | Node (WTree a) a (WTree a)
```

of binary trees, holding values of a type `a` in each (internal) node, and a value of type `Int` in each leaf. We call the values stored in the leaves *weights*.

(a) (3 points) Please define, using direct recursion, a function `height :: WTree a -> Int` which calculates the *height* of a tree (where the `height` of a `Leaf` is defined to be zero).

(b) (4 points) We define the *total weight* of a weighted tree as the sum of the weights of all of its leaves. Using direct recursion, please write a function `totalWeight :: WTree a -> Int` which calculates the total weight of a tree.

(c) (4 points) We call a weighted tree *balanced* if it is a leaf or if the total weight of the left subtree is at most three times that of the right subtree and vice versa and both the left and right subtree are balanced themselves. Using the `totalWeight` function, please write a function `isBalanced :: WTree a -> Bool` which checks whether a weighted tree is balanced.

(d) (5 points) For efficiency reasons, we want to define a function `weightAndBalanced :: WTree a -> (Int, Bool)` which simultaneously computes the total weight of a weighted tree and whether it is balanced. Please define this function.

(e) (3 points) Please define a variation on the data type `WTree` which is, additionally, polymorphic in the type of weight stored in the leaves, where the type of weights stored in the leaves can be different from the type of values stored in the nodes. Use `Leaf` and `Node` as the names of the constructors.

4. In this question, we test your understanding of type inference.

(a) (10 points) Determine the type of the following expressions or demonstrate that they are not well-typed.

- `map elem ['a', 'b'] ['a', 'b', 'c', 'd']`

- `foldr (.)`

(b) (6 points) Which of the following statements are true? Circle *all* correct answers.

A. `map (:)` has type `[e] -> [[e] -> [f]]`

B. `(<= 4)` has type `(Ord v, Num v) => v -> Bool`

5. We end on some multiple choice questions.

(a) (6 points) Please circle *all* of the expressions that have the same value as `[ (f x, x) | x <- xs, p x, q x ]`

A. `(filter p . filter q . map (\x -> (f x, x))) xs`

B. `filter (\y -> p y && q y) (map f xs)`

C. `map (\x f -> (x, f x)) (filter p (filter q xs))`

D. `(map (\x -> (f x, x)) . filter p . filter q) xs`

(b) (6 points) Let us consider a data type describing a zoo:

```
data Zoo = MkZoo [(Enclosure, [Animal])] [Staff]
data Enclosure = PlotOfSand
               | Jungle
               | Aquarium deriving Eq
data Animal = Gorilla
            | Hippo
            | Crab
            | GoldFish
            | Other String
data Staff = MkStaff String Int
```

Which of the following expressions are well-typed at type `Zoo`? Please circle *all* correct answers.

A.
```
MkZoo [(Aquarium, GoldFish), (Jungle, Gorilla)]
      [MkStaff "Frank" 31, MkStaff "Matthijs" 29]
```

B.
```
MkZoo [(PlotOfSand, []), (Aquarium, [Hippo, Crab, Gorilla, Crab, Other "Rat"])]
      [MkStaff "Henk" 52, MkStaff "Ingrid" 42]
```

C.
```
MkZoo [(Aquarium, [Other "Shark", GoldFish "Bertus"])] []
```

(c) (6 points) Clearly, some enclosures are not suitable for some animals. For example, animals that cannot swim cannot be held in an aquarium. Which of the following implementations of `drowns :: (Animal -> Bool) -> Zoo -> [Animal]` check correctly which animals are at risk, assuming that its first argument is a function which checks whether an `Animal` can swim? Please circle *all* correct answers.

A.
```
drowns canSwim (MkZoo eas _) =
    [ a | as <- aquariumAnimals, a <- as, not (canSwim a) ]
   where
    aquariumAnimals =
      map (\(_, as) -> as) (filter (\(e, _) -> e == Aquarium) eas)
```

B.
```
drowns canSwim (MkZoo eas _) =
    concat [ filter (not . canSwim) as | (Aquarium, as) <- eas ]
```

C.
```
drowns canSwim (MkZoo (ea : eas) s) = getDrowningAnimals ea
    ++ drowns canSwim (MkZoo eas s) where
    getDrowningAnimals (Aquarium, a : as) = filter (not . canSwim) as
```