Functional Programming – Mid-term exam – Tuesday 2/10/2018

	Name:		Q:	1	2	3	4	5	Total
			P:	23	20	25	17	15	100
	Student number:		S:						

Before you begin:

- Do not forget to write down your name and student number above.
- If necessary, explain your answers *in English*.
- 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, any, elem, not, (&&), (||), zip, reverse, and all the members of the type classes Show, Eq, Ord, Enum and Num.

1. The function `oneAfterEach` takes a predicate p, a value v, and a list xs. The resulting value is a new list in which each value of xs which satisfies p is followed by v. For example:

```
> oneAfterEach even 0 [1,2,3,4]
[1,2,0,3,4,0]
> oneAfterEach even 0 [1,3,5]  -- No value satisfies the predicate
[1,3,5]
> oneAfterEach even 0 []       -- No values in the list
[]
```

(a) (7 points) Write the implementation of `oneAfterEach` *using direct recursion*.

(b) (3 points) Using the function `oneAfterEach`, define a new function:
    `duplicateACharacter :: Char -> String -> String`
    which duplicates each appearance of a character in a string. For example:

```
> duplicateACharacter 'o' "Hello world"
"Helloo woorld"
```

(c) (6 points) Consider the following type signature for `oneAfterEach`:

```
oneAfterEach :: (a -> Bool) -> b -> [a] -> [b]
```

This type is *rejected* by the compiler. Answer the following two questions.

1. Why is this type incorrect? Give a concise explanation.
2. What is the type that would be *inferred* by the compiler? You do not need to give a formal account of the type inference process for this exercise.

(d) (7 points) Complete the definition of the converse function `oneBeforeEach` which adds a value v *before* each element in xs which satisfies a predicate p.

```
oneBeforeEach p v xs = foldr combine initial xs
  where combine = ...
        initial = ...
```

2. We want to define a set of data types to represent cooking recipes. Each recipe is composed by three pieces of information:

1. A category, that is, whether it is a starter, a main course, a dessert, or a snack.
2. A list of ingredients. The name of the ingredient is represented as a string, and it is given along with the amount required, also as a string (for example "1 cup").
3. A list of tasks. Each task is defined by three elements: *where* should the step be executed – a frying pan, the oven, ... —, *what* should you do — fry an egg, mix the dough, ... —, and for *how long* you have to perform the task, expressed as a number which represents minutes.

For example, here is a value of the `Recipe` type which tells you how to boil rice.

```
Recipe MainCourse
        [Ingredient "rice" "1 cup", Ingredient "water" "500 ml.",
         Ingredient "salt" "1 teaspoon", Ingredient "oil" "1 spoon"]
        [Task "pan" "warm the oil" 2, Task "pan" "add the rice and mix" 2,
         Task "pan" "add water and wait until it boils" 1,
         Task "pan" "wait until water is almost gone" 10]
```

(a) (10 points) Define the data types `Recipe`, `Category`, `Ingredient`, and `Task`.

(b) (4 points) Write a function `cookingTime` which computes the total number of minutes required to perform the recipe (assuming that the steps and performed sequentially and without interruptions). For example, the cooking time of the recipe shown above is $2 + 2 + 1 + 10 = 15$ minutes.

(c) (6 points) Write the following function *without using direct recursion*:

`foodsWithout :: [String] -> [Recipe] -> [Recipe]`

Given a call of the form `foodsWithout is rs`, the result should be a sublist of recipes from `rs`, such that no recipe contains any ingredient from the list `is`. For example:

`> foodsWithout ["apple", "flour"] listOfRecipes`

would not contain a recipe for appeltaart!

Hint: use the function `elem :: Eq a => a -> [a] -> Bool` to look for an element in a list.

3. A `PSTree` extends the idea of a binary search tree with the ability to keep pre-computed integral values:

```
data PSTree a = Leaf Integer | Node Integer a (PSTree a) (PSTree a)
```

Examples of things we can pre-compute are the height or the size of the tree. Here is an example of the former case, a binary search tree of characters in which each subtree remembers its height:

```
tr = Node 2 'd' (Node 1 'z' (Leaf 0) (Leaf 0))
                (Leaf 0)
```

(a) (2 points) Is the value `tr` defined above a binary search tree? Explain why or why not. In the negative case, show a correct search tree with the same elements.

(b) (4 points) Implement *and* give the type of the function `pre` which obtains the get the `Integer` value at the root of the tree. In the previous example, the result should be 2.

(c) (7 points) Write the `Eq` instance for `PSTree a`. This instance should *ignore* the pre-computed integral values, and only check equality of the structure and the contained elements of type a.

(d) (5 points) Define a function with the type:

```
mapPS :: (a -> b) -> PSTree a -> PSTree b
```

which applies the function over each value contained in the tree, and keeps the pre-computed values as they are.

(e) (7 points) Define the function which inserts a new value in the tree:

```
insertPS :: Ord a => Integer -> (Integer -> Integer -> Integer)
         -> a -> PSTree a -> PSTree a
```

This function should perform two tasks:

1. The value should be inserted respecting the invariants of a binary search tree. If the value is already present in the tree, do *not* insert a duplicate.
2. The pre-computed values should be updated. For that reason, we need to additional arguments: what is the pre-computed value for a Leaf – this is the first argument – and how to combine the pre-computed value of two subtrees in a Node — that is the function with type Integer -> Integer -> Integer.

For example, if a tree t contains the pre-computed heights of each subtree, the way to insert a new value v is by calling:

```
> insertPS 0 (\l r -> 1 + max l r) v t
```

4. (a) (5 points) Rewrite the following function `f` to its η-expanded version:

```
f p = filter p . map p
```

In other words, rewrite `f` to the following form. Do so *without* using function composition.

```
f p xs = ...
```

(b) (12 points) Determine the type of the following expression:

```
map (foldr id)
```

5. *Multiple choice questions*. Choose *one* answer.

(a) (5 points) We want to define map in terms of `foldr`. Which of these is the *correct* definition?

    A. `map f = foldr f []`

    B. `map f = foldr (\x r -> f x : r) []`

    C. `map f = foldr (:) []`

    D. It is not possible to define map that way.

(b) (5 points) Which of these statements is *false*?

    A. It is possible to re-define the `(+)` operator for a custom data type.

    B. The expression `[1,2,3]` is equivalent to `[1 .. 3]`.

    C. The type class `Ord` has an instance for `Bool -> Int`.

    D. The type class `Eq` has an instance for `[(Bool, Int)]`.

(c) (5 points) Given the following two expressions:

    I. `[id, length]`

    II. `[sum, length]`

    A. Only (I) is well-typed.

    B. Only (II) is well-typed.

    C. Both (I) and (II) are well-typed.

    D. None of the two expressions are well-typed.