Functional Programming – Mid-term exam – Tuesday 3/10/2017

Name: MODEL SOLUTION	Q:	1	2	3	4	5	Total
Student number:	P:	15	20	30	15	20	100
	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.
- 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`, `not`, `(&&)`, `(||)`, `zip`, `reverse`, and all the members of the type classes `Show`, `Eq`, `Ord`, `Enum` and `Num`.

1. The function `intercalate` takes a separator `s` and a list `xs`, and produces a new list where the elements all the elements in the list `xs` appear separated by `s`. For example:

```
> intercalate " / " ["a", "bc", "d"]
"a / bc / d"
> intercalate [1,2] [[3,4],[5,6,7]]
[3,4,1,2,5,6,7]
```

(a) (7 points) Write the function `intercalate :: [a] -> [[a]] -> [a]` *using direct recursion*.

```
intercalate _ [] = []
intercalate _ [x] = x
intercalate s (x:xs) = x ++ s ++ intercalate s xs
```

Another possibility for the last branch

```
                  = concat (x : s : intercalate s xs)
```

(b) (6 points) Complete the definition of `f` in this implementation of `intercalate`:

```
intercalate _ []      = []
intercalate s (x:xs) = x ++ concat (map f xs)
  where f = ...
```

```
... f = (s++)
```

(c) (2 points) Define the function `map` using *only one* list comprehension:

```
map f xs = ...
```

```
... = [f x | x <- xs]
```

2. We want to define a data type `EncItem` to represent items in a encyclopedia. Each of the items could be:

- An entry, which has a string for the title of the entry and some text for the definition.
- A list which contains a header – defined as a string – and a list of subitems, which can be either entries or more lists.

One example of a value of that type is:

```
List "Animals" [ List "Mammals" [ Entry "Dog" "A dog is ..."
                                , Entry "Cat" "A cat is ..."
                                ]
               , Entry "Domestic animals" "These animals ..."
               ]
```

(a) (7 points) Define the data type `EncItem`.

```
data EncItem
    = Entry String String
    | List  String [EncItem]
```

(b) (6 points) Define a function `noEntries` using recursion to count the number of entries *at any level* in an `EncItem` value. For example, `noEntries` applied to the previous example should return 3. Give also the type signature for that function.

```
noEntries :: EncItem -> Int    (Integer is also ok)
noEntries (Entry _ _) = 1
noEntries (List _ is) = sum (map noEntries is)
```

(c) (7 points) Write an `Eq` instance for `EncItem`. The equality check should compare all values for equality, except for the titles of the entries and lists, which should be compared ignoring differences in casing (so `"bad"` equals `"BAD"` equals `"BaD"`).
Hint: use `map toUpper s` to turn the string `s` to capital letters.

```
instance Eq EncItem where
  Entry t1 d1 == Entry t2 d2
     = map toUpper t1 == map toUpper t2 && d1 == d2
  List t1 i1 == List t2 i2
     = map toUpper t1 == map toUpper t2 && i1 == i2
  _ == _ = False
```

3. Take the following data type representing binary trees:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

We define a function `root` to get the top-most element in the tree:

```
root :: Tree a -> a
root (Leaf x)     = x
root (Node x _ _) = x
```

(a) (9 points) Define a function `allSums` which takes a `Tree Int` and returns a new tree where the internal nodes have been replaced by the sum of each subtree and the internal node itself. For example, given the tree:
`t = Node 3 (Node 2 (Leaf 1) (Leaf 0)) (Leaf 4)`
the result of `allSums t` should be:
`Node 10 (Node 3 (Leaf 1) (Leaf 0)) (Leaf 4)`

```
allSums l@(Leaf _) = l
allSums (Node x l r) =
   Node (x + root l' + root r') l' r'
   where l' = allSums l
         r' = allSums r
```

(b) (9 points) Define a function `foldTree` by recursion over the tree with the type

```
foldTree :: (a -> b -> b -> b)
         -> (a -> b)
         -> Tree a -> b
```

such that `root (allSums t) = foldTree (\x y z -> x + y + z) (\x -> x) t.`

```
foldTree _ g (Leaf x) = g x
foldTree f g (Node x l r) =
    f x (foldTree f g l) (foldTree f g r)
```

(c) (4 points) Complete the following definition of a function which obtains the height of the tree. The height of a `Leaf` should be zero.

```
height :: Tree a -> Int
height t = foldTree f g t
  where f = ...
        g = ...
```

```
... f = \_ l r -> 1 + max l r
        The solution 1 + maximum [l,r] is also Ok
    g = \_ -> 0
```

(d) (8 points) Complete the following definition of a function `findTree` which obtains the first element in the tree which satisfies a given predicate.

```
findTree :: (a -> Bool) -> Tree a -> Maybe a
findTree p t = foldTree f g t
  where f = ...
        g = ...
```

Hint: the following function from the base library is useful to define this function.

```
(<|>) :: Maybe a -> Maybe a -> Maybe a
Just x  <|> _ = Just x
Nothing <|> y = y
```

```
... g x | p x = Just x
        | otherwise = Nothing

    f x l r = g x <|> l <|> r
    -- Or also
    f x l r | p x       = Just x
            | otherwise = l <|> r
```

4. *Types and type inference.*

   (a) (10 points) Determine the type of `map foldr`.
   Hint: the type of `foldr` is `(a -> b -> b) -> b -> [a] -> b`.

The types of the functions are:

```
 map   :: (c → d) → [c] → [d]
 foldr :: (a → b → b) → b → [a] → b
```

foldr is the <u>first argument</u> to map, thus:

$$c \to d = (a \to b \to b) \to (b \to [a] \to b)$$

Solving the equation gives us:

$$c = a \to b \to b \qquad d = b \to [a] \to b$$

The result type is $[c] \to [d]$. We substitute:

```
   map foldr :: [a→b→b] → [b→[a]→b]
```

(b) (5 points) When we say that `flip` may take 1, 2 or 3 arguments, what do we mean?

- flip may be partially applied
- Haskell uses curried functions, which means they take one argument at a time

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

(a) (5 points) Which is the type of `\f g x -> (f x, g x)`?

A. This expression is not well-typed.
[✓ B.] `(a -> b) -> (a -> c) -> a -> (b, c)`
C. `(a -> b) -> (b -> c) -> a -> (b, c)`
D. `(a -> b, b -> c) -> a -> (b, c)`

(b) (5 points) Using only the definitions in the base library:

1. `[Int -> Bool]` is an instance of `Eq`, (functions are not in Eq)
2. `([Int], [Bool])` is an instance of `Eq`.

A. None of them is true.
B. (1) is true and (2) is false.
[✓ C.] (1) is false and (2) is true.
D. Both of them are true.

(c) (5 points) What expression has the same value as `[f x | x <- [1 .. 4], even x]`?

A. `filter even (map f [1 .. 4])`
B. `f (map even [1 .. 4])`
[✓ C.] `map f (filter even [1 .. 4])`
D. `filter f (map even [1 .. 4])`

(d) (5 points) What is the result of the following expression?

```
let lst = [1.0, 2.0, 3.0]
 in (foldr (/) 1.0 lst, foldl (/) 1.0 lst)
```

[✓ A.] (1.5, 0.1666666)
B. (0.1666666, 1.5)
C. (1.5, 1.5)
D. The expression is not well-typed.