Functional Programming – Final exam – Thursday 7/11/2019

Name:

Student number:

Q:	1	2	3	4	5	Total
P:	20	18	18	20	14	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.
- 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`, `curry`, `uncurry`, and all the members of the type classes `Show`, `Eq`, `Ord`, `Enum`, `Num`, `Monoid`, `Functor`, `Applicative`, and `Monad`.

1. Functors and Monoids

(a) (4 points) Consider the following data type `First a`, somewhat similar to the type `Maybe a`

```
data First a = TheFirst a | Never deriving (Show,Eq,Ord)
```

Make `First` an instance of `Functor`

(b) (4 points) Recall the `Monoid` typeclass

```
class Monoid m where
  mempty :: m
  (<>)   :: m -> m -> m
```

which models a type `m` with an associative binary operation `<>` which combines two elements into one, and a unit element `mempty`.

Make `First a` an instance of `Monoid` in which the associative operation returns its left argument as long as it is a `TheFirst a` and its right argument otherwise: such that for example

```
foldr (<>) Never [Never, TheFirst 6, TheFirst 1, TheFirst 10] == TheFirst 6
```

(c) (4 points) Consider the following data type `These`, which has two type parameters l and r:

```
data These l r = Neither
               | This l
               | That r
               | Both l r
```

By making `These l` an instance of functor we can apply functions to the `r` value in a `These l r`. However, we would also like to be able to map over the `l` value. That is, we would like a function

```
bimapThese :: (l -> l') -> (r -> r') -> These l r -> These l' r'
```

Implement this function.

(d) (4 points) There are also other types that have two type parameters, for example a pair `(l,r)`. So, we could similarly define a function `bimapPair` that takes two functions `f` and `g` and applies them on the left and right values in a pair, respectively. This is a good opportunity to introduce a new typeclass `Bifunctor` which expresses that for some type `t`, parameterized by two types `l` and `r`, we can map functions `f` and `g` over the `l` and `r` values using the function `bimap`. We can then make `These` and pairs (`(,)`) an instance of this typeclass:

```
instance Bifunctor These where
  bimap = bimapThese

instance Bifunctor (,) where
  bimap = bimapPair
```

Define the typeclass `Bifunctor`.

(e) (4 points) Write a function `foo :: Bifunctor t => t Int Float -> t String String` which, when specialized to `t` equal to `(,)` converts a pair of an `Int` and a `Float` into a pair of `Strings`. For example, `foo (42, 66.6) = ("42", "66.6")`.

2. **Testing**

Consider the function `(++)` which concatenates two lists. We will develop a function `concatSpec` that can be used to test if an implementation of `(++)` correctly concatenates the input lists.

(a) (4 points) Write a function `isPrefixOf :: Eq a => [a] -> [a] -> Bool` such that `isPrefixOf xs ys` tests if a list `xs` is a prefix of `ys`, that is, if `ys == xs ++ bs`, for some list `bs`.

(b) (6 points) Recall the data type `First a` as defined in the first exercise. Consider a function `findFirst :: [a] -> [a] -> First Int` that, given arguments `xs` and `ys` returns `TheFirst i` if and only if `xs` is a contiguous (in Dutch: "aaneengesloten") sublist of `ys` starting at index `i`. The function returns `Never` if and only if `xs` does not occur as a contiguous sublist of `ys`.

Using `findFirst`, write a function `before :: Eq a => [a] -> [a] -> [a] -> Bool` for which `before xs ys zs` returns `True` if and only if `xs` and `ys` are contiguous sublists of `zs` and (the first occurrence of) `xs` starts no later than (the first occurrence of) `ys` in the list `zs`.

(c) (4 points) Using the above functions, we can now give the following specification `concatSpec` that tests if an implementation `concatImpl` correctly concatenates two lists:

```
concatSpec :: Eq a => ([a] -> [a] -> [a]) -> [a] -> [a] -> Bool
concatSpec concatImpl xs ys = before xs ys (xs ‘concatImpl‘ ys)
```

This specification is incomplete, since there may be an implementation, say `myConcat`, that satisfies the above specification, but for which the result is different than for the real implementation of `(++)`. That is, we may have `concatSpec myConcat xs ys == True` while `xs ‘myConcat‘ ys /= xs ++ ys`. Please explain why this specification is incomplete.

(d) (4 points) Extend the above specification of `concatSpec` to correctly test if a given implementation correctly concatenates two lists.

Note that you are not allowed to refer to the ”real” function `(++)` in your answer.

```
concatSpec :: Eq a => ([a] -> [a] -> [a]) -> [a] -> [a] -> Bool
concatSpec concatImpl xs ys = let zs = xs ‘concatImpl‘ ys
                              in before xs ys zs && ....
```

3. **Monads**

(a) (4 points) Recall that

1. a `Map k v` is a data structure that associates keys of type `k` with values of type `v`, and allows us to efficiently retrieve the value associated with a key, if it exists, using the function
`lookup :: Ord k => k -> Map k v -> Maybe v`.
If the key does not occur in the `Map`, lookup returns a `Nothing`.
2. `Maybe` is an instance of `Monad`.

Consider a function `combineLookup :: Ord k => (v -> v -> Maybe b) -> k -> k -> Map k v -> Maybe b` that looks up two keys in a `Map k v` (using the `lookup` function), and combines their values using a user supplied function.

Here are some example uses of `combineLookup`, in which `m = Map.fromList [(1,"foo"), (2, "bar"), (6,""), (8,"baz")]` is a Map that maps the key `1` to ”foo”, 2 to ”bar” etc.

```
> combineLookup (\v1 v2 -> Just (v1 ++ v2)) 1 2 m
Just "foobar"
> combineLookup (\v1 v2 -> if null v1 then Just v2 else Nothing) 1 2 m
Nothing
> combineLookup (\v1 v2 -> if null v1 then Just v2 else Nothing) 6 2 m
Just "bar"
> combineLookup (\v1 v2 -> if null v1 then Just v2 else Nothing) 3 2 m
Nothing
```

Using do-notation, please implement `combineLookup`.

(b) (4 points) Translate the following piece of code using do-notation to using `return` and `>>=` directly.

```
main = do (fp:h:_) <- getArgs
          putStrLn h
          s        <- readFile fp
          return (length s)
```

(c) (3 points) Consider the following data type `Log a`, which annotates a value of type `a` with a list of log messages, and the function `withLogging` that prints these messages to the terminal and then returns the `a`:

```
data Log a = MkLog [String] a

withLogging             :: Log a -> IO a
withLogging (MkLog l a) = do mapM_ putStrLn l
                             return a
```

Write a function `log :: String -> Log ()` which logs a single message and returns a value of type `()`.

(d) (5 points) We can make `Log` an instance of `Monad` so that we can write nice logging code. For example:

```
readInput :: Log Int
readInput = do log "about to read some input"
               return 5

computeSomething :: Log String
computeSomething = do i <- readInput
                      log "read some input"
                      let out = i * i
                      log "computed something"
                      return (show out)

computeIO :: IO String
computeIO = withLogging computeSomething
```

So that evaluating `computeIO` prints

```
about to read some input
read some input
computed something
```

to standard output and returns the string `"25"`. Complete the `Monad` instance for `Log`, i.e. give the implementation of

1. `return :: a -> Log a`
   which does not log any messages
2. `(>>=) :: Log a -> (a -> Log b) -> Log b`
   which collects all messages

(e) (2 points) Write a function `withoutLogging :: Log a -> IO a` that returns the `a` in `Log a`, but does not actually print any log messages.

4. Equational reasoning

Given the definitions

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

```
a) foldr f e [] = e             b) foldr f e (x:xs)    = f x (foldr f e xs)

c) map f []     = []            d) map f (x:xs)        = f x : map f xs

e) size Leaf    = 0             f) size (Node l x r)   = size l + 1 + size r

g) toList Leaf  = []            h) toList (Node l x r) = toList l ++ [x] ++ toList r

i) (.) f g x    = f (g x)
```

and the lemma

`j)` for all lists `xs`, `ys`, and `zs`: `length xs + length ys + length zs = length (xs ++ ys ++ zs)`

prove that the following two equations hold:

(a) (12 points) for all lists `xs`: `map f (foldr (\x r -> g x : r) [] xs) = map (f . g) xs`

(b) (8 points) `size = length . toList`

5. **Lazy evaluation**

(a) (10 points) For each of these expressions, indicate if they are in WHNF or not. For the ones that are in WHNF, state in one sentence why. For the ones not in WHNF, evaluate them to WHNF or, in case they crash upon evaluation, indicate this.

A. `(1 + 5) : succ 4 : map (+1) [1,2]`

B. `isNothing (Just 4)`

C. `\a b c -> and b`

D. `foldr undefined e []`

E. `seq fmap`

(b) (4 points) Consider the following two statements:

1. the expression `(\x f -> f x) undefined` crashes when it is evaluated
2. the expression `seq (undefined,undefined) (const 5 undefined)` crashes when it is evaluated

Which of these statements are true? Choose *one* answer.

A. Both statements are false

B. Only statement 1 is true

C. Only statement 2 is true

D. Both statements are true