Functional Programming – Final exam – Thursday 8/11/2018

Name: SOLUTIONS
Student number:

Q:	1	2	3	4	5	6	7	Total
P:	14	16	16	27	12	15	5	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.
- 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`, `Num`, `Functor`, `Applicative`, and `Monad`.

1. We have often spoken of a tuple `(a, t)` as having two components of types `a` and `t`. Another way to look at them is as a value of type `t` *annotated* with some information of type `a`.

   (a) (4 points) Write the following two functions:

   ```
   annotateMaybe :: (t -> a) -> Maybe t -> Maybe (a, t)
   annotateList  :: (t -> a) -> [t]     -> [(a, t)]
   ```

   which annotate an optional value, or every element of a list of values, respectively, with the result of applying the function given as first argument.

```
annotateMaybe _ Nothing = Nothing
annotateMaybe f (Just x) = Just (f x, x)

annotateList f lst
    = [(f x, x) | x <- lst]
    = map (\x -> (f x, x)) lst
```

several options are possible...

   (b) (5 points) This function can be generalized to work over any `Functor`. Write the implementation of the following function:

   ```
   annotate :: Functor f => (t -> a) -> f t -> f (a, t)
   ```

```
annotate f = fmap (\x -> (f x, x))
```

(c) (5 points) Given an annotated data structure, we may split it in two parts: the first one contains only the annotations, and the second one the actual values.

```
split :: Functor f => f (a, t) -> (f a, f t)
split x = (fmap (\(a,_) -> a) x, fmap (\(_,t) -> t) x)
```

Now consider the following function, which splits a list `[(a, t)]` of annotated elements and puts back together another list of the same type using the corresponding `Monad` instance.

```
weird lst = do let (anns, vals) = split lst
               ann <- anns
               val <- vals
               return (ann, val)
```

Would this function, in general, give back the same list that was given as argument? Explain your answer, or give a counterexample if this is not the case.

Consider [(1,'a'),(2,'b')]. Applying 'weird' to this list gives you [(1,'a'),(1,'b'),(2,'a'),(2,'b')] because the 'do' in lists give you all possible combinations

2. The `Monoid` type class represents types along with a binary operation `(<>)` which is associative and has a neutral element `mempty`:

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

If we have a container whose elements come from a monoidal type, we can "crush" all of them into a single value. This leads to the definition of the `Crushable` type class, for which an instance for `[]` can be defined:

```
class Functor f => Crushable f where
  crush :: Monoid m => f m -> m

instance Crushable [] where
  crush []     = mempty
  crush (m:ms) = m <> crush ms
```

For example, `crush ["haskell", " is ", "fun!"]` returns `"haskell is fun!"`.

(a) (8 points) Write the `Crushable` instance for the `Maybe` type constructor, and for the following version of binary trees:

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

```
instance Crushable Maybe where
  crush Nothing = mempty
  crush (Just x) = x

instance Crushable Tree where
  crush (Leaf a) = a
  crush (Node l x r) = crush l <> x <> crush r
```

(b) (3 points) In order to distinguish between the different monoidal structures that we can give to numbers, we wrap them into a new data type. Here is the definition of `Sum`, whose `Monoid` instance uses the addition operation from the underlying numeric type:

```
data Sum a = Sum { unSum :: a }

instance Num a => Monoid (Sum a) where
  mempty = Sum 0
  Sum x <> Sum y = Sum (x + y)
```

Using `crush` and `Sum`, define a generalized version of the `length` function for any crushable functor. Remember that any `Crushable` is also a `Functor`.

```
glength :: Crushable f => f a -> Integer
```

```
glength = unSum . crush . fmap (\_ -> Sum 1)
```

(c) (5 points) A *tropical monoid* results from taking a type which can be ordered, and then defining the monoidal operation as taking the minimum between two values. In order to define a neutral element, we need to attach an additional value representing ∞, that is, $\min(x, \infty) = \min(\infty, x) = x$.

```
data Tropical a = Value a | Infinite
```

Complete the following definition of the `Monoid` instance for `Tropical a`:

```
instance Ord a => Monoid (Tropical a) where ...
```

```
mempty = Infinite
Infinite <> y = y
x <> Infinite = x
Value x <> Value y = Value (min x y)
```

3. EQUATIONAL REASONING AND INDUCTION

(a) (6 points) Given the following definitions for some well-known functions:

```
(a) sum []     = 0
(b) sum (x:xs) = x + sum x

(c) concat []     = []
(d) concat (x:xs) = x ++ concat xs

(e) map _ []     = []
(f) map f (x:xs) = f x : map f xs

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

Prove by induction that the following holds:

`length . concat = sum . map length`

In the proof you may assume that the following lemma is true:

`length (x ++ y) = length x + length y`

By extensionality we need to prove

length (concat xs) = sum (map length xs)

- Case []

length (concat []) = length [] = 0

sum (map length []) = sum [] = 0

- Case xs = z:zs

length (concat (z:zs))
= length (z ++ concat zs)
(lemma) = length z + length (concat zs)

sum (map length (z:zs))
= sum (length z : map length zs)
= length z + sum (map length zs)
(IH) = length z + length (concat zs)

(b) (10 points) Given the following definitions:

```
(a)  filter _ [] = []
(b)  filter p (x:xs)
(b1)    | p x       = x : filter p xs
(b2)    | otherwise =     filter p xs
```

```
(c)  mapMaybe _ [] = []
(d)  mapMaybe f (x:xs) = case f x of
(d1)                       Just y  -> y : mapMaybe f xs
(d2)                       Nothing ->     mapMaybe f xs
```

Prove that the following holds for any predicate `p :: a -> Bool` and list `xs :: [a]`,
`filter p xs = mapMaybe (\x -> if p x then Just x else Nothing) xs`
Use induction. State and prove here the `[]` case.

filter p [] = [] ← they are equal

mapMaybe (\x → ...) [] = []

State the induction hypothesis and prove here the `(z:zs)` case. You need to distinguish two cases, depending on whether the predicate p holds for the element z or not.

To prove: filter p (z:zs) = mapMaybe (\x→...) (z:zs)

IH: filter p zs = mapMaybe (\x→...) zs

· Case p z is True

filter p (z:zs)
= z : filter p zs
(IH)= z : mapMaybe ... zs

mapMaybe (\x→...) (z:zs)
= case (\x→...) z of
  Just y → y : mapMaybe ...
  Nothing → mapMaybe ...
=((\x→...) z = Just z)
  z : mapMaybe ... zs

· Case p z is False

filter p (z:zs)
= filter p zs
(IH)= mapMaybe ... zs

mapMaybe (\x→...) (z:zs)
= case (\x→...) z of ...
= (since (\x→...) z = Nothing)
  mapMaybe ... zs

4. Consider the following data type of arithmetic expressions in which the type of the operations op variables v may be chosen by the programmers:

```
data ArithExpr v = Variable v
                 | Literal Integer
                 | Add   (ArithExpr v) (ArithExpr v)
                 | Times (ArithExpr v) (ArithExpr v)
```

An *evaluator* for this data type describes how to obtain a final `Integer` value given the value of each variable. Here we assume that *every* variable has a defined value, and thus we can represent the mapping as a function:

```
eval :: (v -> Integer) -> ArithExpr v -> Integer
eval m (Variable v)  = m v
eval _ (Literal  n)  = n
eval m (Add   e1 e2) = eval m e1 + eval m e2
eval m (Times e1 e2) = eval m e1 * eval m e2
```

(a) (8 points) Write a monadic evaluator for `ArithExpr`, that is, complete the definition of the following `evalM` function. The difference with `eval` above is that the variable handling works in a monadic context.

```
evalM :: Monad m => (v -> m Integer) -> ArithExpr v -> m Integer
```

```
evalM m (Variable v) = m v
evalM _ (Literal n) = return n
evalM m (Add e1 e2) = do n1 ← evalM m e1
                         n2 ← evalM m e2
                         return (n1 + n2)
                    } using "do"

                    = (+) <$> evalM m e1
                          <*> evalM m e2
                    } using Applic.

(the same for Times e1 e2)
```

(b) (5 points) During the lectures we have worked with partial mappings, that is, mappings which are not defined for all possible inputs. We describes such mappings via a list of tuples `[(v, Integer)]`. Using the functions `evalM` defined above, and the function `lookup :: a -> [(a, b)] -> Maybe b`, write the following evaluator which fails if one of the variables is not present:

```
eval' :: Eq v => [(v, Integer)] -> ArithExpr v -> Maybe Integer
```

```
eval' m = evalM (\x → lookup x m)
   also = evalM (flip lookup m)
```

(c) (10 points) In addition to the possibility of failing, we want to keep track of which variables are used during the evaluation of an arithmetic expression. In order to do so, we define the following data type:

```
data TrackAndFail v a = TF (Maybe a, Set v)
```

A value of type `Set v` represents a *set* of values of type `v`. Sets form a monoid: they can be combined using the `(<>)` operation, and the empty set is represented by `mempty`.

Write the `Monad` instance for `TrackAndFail v`. First give the definition for `return`:

```
-- in general, return :: a -> m a
return x = ...
```

```
return x = TF (Just x, mempty)
```

Now write the definition of the bind operation, `(>>=)`. You may introduce additional pattern matching in the definition:

```
-- in general, (>>=) :: m a -> (a -> m b) -> m b
x >>= f = ...
```

```
TF (Nothing, s1) >>= _ = TF (Nothing, s1)
TF (Just x, s1) >>= f
    = let TF (r, s2) = f x
      in  TF (r, s1 <> s2)
```

(d) (4 points) Write a QuickCheck property that states that evaluating a literal with `verb'` always returns the value in that literal.

```
prop n = verb' (Literal n) == n
(if you take verb' = eval')
prop n = eval' (\_ -> 0) (Literal n) == n
```

5. (12 points) Consider the following piece of code, where `g :: Int -> Maybe Int` and `h :: a -> Maybe a`.

```
f w = do x <- g w
         let xs = do z <- [1, 2]
                     v <- ['a', 'b']
                     return (z, v)
         y <- h (snd (head xs))
         return y
```

Complete the following sentence by filling in the gaps:

In the `Maybe` monad, (1) _____ signals failure and (2) _____ a successful computation. In the above program, the type of `w` is (3) _____, the type of `x` is (4) _____ and the type of `xs` is (5) _____. If we run `f` and print the value of `xs` to the screen we would see (6) _____.

(1) Nothing
(2) Just x / Just
(3) Int
(4) Int
(5) [(Int, Char)]
(6) [(1,'a'), (1,'b'), (2,'a'), (2,'b')]

6. LAZINESS AND EVALUATION

(a) (8 points) For each of the following expressions, indicate whether they are in *weak head normal form* (WHNF), and explain why. If they are not, write the corresponding value in WHNF.

- `Just (4 + 4)`
- `map (+1) []`
- `(\x -> 0) 0`
- `[1 ..]`

- Yes, has a top-level constructor
- No, the function can't be applied
  WHNF : [1
- No, the function can be applied
  WHNF : 0
- Yes, it has a top-level constructor (:)

(b) (4 points) Consider the following two variations of the take function:

```
take1 _ []     = []
take1 0 _      = []
take1 n (x:xs) = x : take1 (n-1) xs

take2 0 []     = []
take2 0 (x:xs) = []
take2 n []     = []
take2 n (x:xs) = x : take2 (n-1) xs
```

Give an example of call to `take2` which results in `undefined`, which would return an actual value using `take1`. What does this tells about the strictness of each argument position?

take2 undefined [] = undefined
but take1 undefined [] = []

This tells us that take2 is more strict than take1 on its first argument

(c) (3 points) Write the function `take2` using `take1` and `seq`.

```
take2 n xs = n `seq` take1 n xs
```

we use seq to force the evaluation of the first argument

7. In the lecture about *Formal Verification in Agda*, the following data type was introduced: (BONUS)

```
data Vec (A : Set) : ℕ → Set where
  []   :                          Vec A Z
  _::_ : ∀{n} → A → Vec A n → Vec A (S n)
```

(a) (3 points) What does this data type represent?

Lists which keep track of how many elements they contain

(b) (2 points) How would you use this data type to define a version of the `head` function which never fails?

Only allow lists whose length is greater than zero
In code
head : Vec a (S n) → a