Functional Programming

Month dd, 20yy
13:45 - 16:45

- You may use any published book on Haskell, the module guide *and* lecture slides. This material should not contain any notes.
- The last page of the exam contains a reference with some definitions.
- Calculators, laptops, mobile phones, etc. are not allowed.
  Please put those in your bag now (switched off)!
- You may use predefined Haskell functions and operators from the packages `Prelude`, `Data.List`, `Data.Char`, `Data.Maybe`, `Data.Either`, `System.IO`, `Control.Applicative`, `Data.Monoid`.
- Style and elegancy also play a role in the grading, e.g., do not use *unnecessary* helper functions, counters, etc.
- Write your answers on this paper, in the provided boxes
- **Hand in this complete exam with your student number and name (also when no questions are answered).**
- Total points: 90, grade = $\frac{\text{points}+10}{10}$

Your name:

(please underline your family name (i.e., the name on your student card), so that we know how to sort)

Your student number:

Question 1

25 points

1. **(5 points)** The function `lookup` receives a key and a list of (key, value)-pairs. For the given key it provides the value that belongs to the given key - when the key cannot be found it results in `Nothing`. Define the function `lookup` using *recursion* and give its type.

```
lookup :: Eq k => k -> [(k, v)] -> Maybe v
lookup k []                          = Nothing
lookup k ((k',v):xs) | k == k'   = Just v
                     | otherwise = lookup k xs
```

2. **(5 points)** The function `lefts :: [Either a b] -> [a]` gives all the `Left` values within a given list. Define the function `lefts` using *list comprehension*.

```
lefts :: [Either a b] -> [a]
lefts xs = [ l | (Left l) <- xs  ]
```

3. **(5 points)** The function `justs :: [Maybe a] -> Maybe [a]` receives a list of `Maybe` values, and results in `Nothing` when one or more of these values is `Nothing`, otherwise it produces the list of values. Define the function `justs` by combining *recursion* and the *applicative style*.

```
justs :: [Maybe a] -> Maybe [a]
justs []     = pure []
justs (x:xs) = (:) <$> x <*> justs xs
```

4. **(5 points)** The function `first :: [Maybe a] -> Maybe a` receives a list of `Maybe` values, and gives the *first* `Just` value and it otherwise results in `Nothing`. Define `first` using `foldl`.

```
first :: [Maybe a] -> Maybe a
first xs
 = foldl (\x y -> if (isJust x)
                    then x
                    else y)
         Nothing xs
```

5. **(5 points)** The `IO` action `range :: IO [Int]` results in a sequence of the following `IO` actions:

   (a) It prints "First value?" to the standard output (i.e., console).

   (b) It reads an `Int` from the standard input (i.e., the console).

   (c) It prints "Second value?" to the standard output (i.e., console).

   (d) It reads a second `Int` from the standard input (i.e., the console).

   It results in a list of `Int`s starting with the first given value, up to (and including) the second value. For example:

```
*Main> range
First value?
2
Second value?
4
[2,3,4]
```

   Use the *applicative style* to define `range :: IO [Int]`.

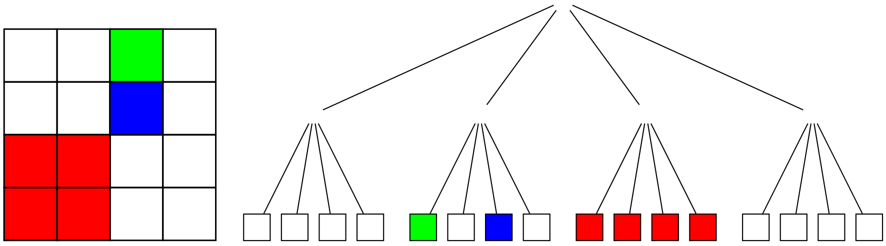
```
range :: IO [Int]
range = (\x y -> [read x .. read y])
    <$> (putStrLn "First value?" *> getLine)
    <*> (putStrLn "Second value?" *> getLine)
```

Question 2

30 points

A QuadTree is a tree wherein leafs contain the data, and internal nodes have *exactly* four children. It it ofted used to store image data by (recursively) splitting a matrix of values into four block matrices of equal size (we only consider the simple variant) and storing each of the four block matrices within a QuadTree node. Single values are stored within leafs. See Fig. 1 for an example.

For simplicity, we assume that all matrices are square matrices with dimensions that are a power of 2 (i.e., 1×1, 2×2, 4×4, 16×16, etc.)



(a) Figure (b) Corresponding QuadTree

Figure 1: Example of a QuadTree

1. **(5 points)** The function

   `splitMatrix :: [[a]] -> ([[a]], [[a]], [[a]], [[a]])`

   receives a matrix (list of lists) and splits the matrix into four new matrices of equal size. The order of the resulting matrices in the tuple are: north west, north east, south west, south east.

   Define the function `splitMatrix` using higher order functions and function application; do not use recursion or list comprehension.

```
splitMatrix :: [[a]] -> ([[a]], [[a]], [[a]], [[a]])
splitMatrix xss = (nw, ne, sw, se)
  where
    (v1, v2) = splitAt n xss
    (nw, ne) = unzip $ map (splitAt n) v1
    (sw, se) = unzip $ map (splitAt n) v2
    n        = length xss `div` 2
```

2. **(5 points)** The following data type describes a range:

   `data Range a = Range a a deriving (Eq, Show)`

   When two `Range`s `r1` and `r2` are merged, the result is the smallest range that includes both `r1` and `r2`. Give a `Monoid` instance for `Range`.

   Hint: have a look at the class `Bounded`. In your instance use the class constraint `(Ord a, Bounded a) =>`

```
instance (Ord a, Bounded a) => Monoid (Range a) where
  mempty = Range maxBound minBound
  (Range a b) `mappend` (Range x y)
         = Range (min a x) (max b y)
```

3. **(10 points)** The following data type describes a `QuadTree`:

```
data QuadTree a = QNode Int (Range a) (QuadTree a)
                                      (QuadTree a)
                                      (QuadTree a)
                                      (QuadTree a)
                | QLeaf Int a
```

Here, `a` is the type of the values stored inside the tree, `Int` is size of the original matrix, and `Range a` contains the minimum and maximum value stored in its children.

The function

```
makeQuadTree :: (Bounded a, Ord a) => [[a]] -> QuadTree a
```

receives a matrix and uses it to produce a `QuadTree a` wherein all values are filled in as described above.

Give a *recursive* definition of `makeQuadTree`.

```
makeQuadTree :: (Bounded a,Ord a) => [[a]] -> QuadTree a
makeQuadTree [[x]] = QL 1 x
makeQuadTree xss   = QN n r ne' nw' se' sw'
  where (ne, nw, se, sw) = splitMatrix xss
        ne' = makeQuadTree ne
        nw' = makeQuadTree nw
        se' = makeQuadTree se
        sw' = makeQuadTree sw
        r   = getRange ne' <>
              getRange nw' <>
              getRange se' <>
              getRange sw'
        n   = (length xss) `div` 2
```

4. **(10 points)** The function `compress :: QuadTree Int -> QuadTree Int` compresses a QuadTree by replacing a node wherein the `Range` is *exactly* a single value (e.g., `Range 1 1`) by a `Leaf` in a way to minimize the size of the QuadTree.

Give a recursive definition of `compress`.

```
compress :: QuadTree Int -> QuadTree Int
compress (QL n x) = QL n x
compress (QN n r@(Range l h) nw ne sw se)
   | l == h    = QL (2*n) l
   | otherwise = QN n r nw' ne' sw' se'
  where
    (nw', ne', sw',se') = (compress nw, compress ne,
                           compress sw, compress se)
```

Question 3

5 points The following code is *incorrect*. Explain briefly what the problem is (use *one* argument). Give the smallest change that fixes the problem.

```
data A z x = B { y :: x }

instance Functor (A z x) where
  fmap f = B . f . y
```

A `Functor` instance requires a type constructor with one argument, but the provided type constructor has zero arguments.

Correct code:

```
instance Functor (A z) where
  fmap f = B . f . y
```

Question 4

5 points Consider the following Haskell code:

```
r :: [Int]
r = [1..]
```

`r` an infinite list starting as $1, 2, 3, \ldots$.

Provide *recursive* definition for `r :: [Int]` that uses `zipWith`.

```
r :: [Int]
r = 1 : zipWith (+) r (repeat 1)
```

Question 5

25 points In this question you work with tuples with three values inside (triples), of the type `(a,b,c)`

1. **(5 points)** Define a `Functor` instance for `(a,b,c)`.

```
instance Functor ((,,) a b) where
  fmap f (a,b,c) = (a,b,f c)
```

2. **(5 points)** Prove the following `Functor` law for your `Functor` of `(a,b,c)`:

   `fmap (f . g)  ==  fmap f . fmap g`

Note, this is the same as proving:

```
fmap (f . g) (a,b,c) ==  (fmap f . fmap g) (a,b,c)
```

To finish the proof:

```
  fmap (f . g) (a,b,c)   = (a, b, f (f.g) c)
= (a, b, f (g c))        = fmap f (a,b,g c)
= fmap f (fmap g (a,b,c) = (fmap f . fmap g) (a,b,c)
```

3. **(5 points)** Define an `Applicative` instance for `(a,b,c)`, where you assume `a` and `b` are `Monoid` instances. Use the `Monoid` to combine the values in your definition of `Applicative` where possible.

```
instance (Monoid a, Monoid b) => Applicative ((,,) a b) where
  pure c = (mempty, mempty, c)
  (a, b, f) <*> (x, y, z) = (a <> x, b <> y, f z)
```

4. **(5 points)** Prove the following `Applicative` law:

   `pure f <*> pure x = pure (f x)`

> Using *your* `Applicative`:
>
> ```
>   pure f <*> pure x = pure (f x)
> = (mempty, mempty, f) <*> (mempty, mempty, x)
> = (mempty <> mempty, mempty <> mempty, f x)
> = (mempty, mempty, f x) -- mempty is the identity under <>
> = pure (f x)
> ```

5. **(5 points)** Provide a short but nontrivial example expression that uses the `Applicative` style with your applicative functor (use both `<$>` and `<*>`). Also provide the evaluation result of your expression (only the end result).

> ```
> x = (*) <$> ("a", [1], 10) <*> ("b", [2,3], 20)
> ```
>
> This evaluates to: `("ab",[1,2,3],200)`

Some important types, type classes and functions

```
data Maybe a = Nothing | Just a

data Either a b = Left a | Right b

class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a

class Bounded a where
  minBound :: a -- lowest value a can assume
  maxBound :: a -- highest value a can assume

class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a

class Functor where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

putStrLn :: String -> IO ()

getLine :: IO String
```