# Programming Paradigms 2023
# Session 11: Monads

## Problems for solving and discussing

Hans Hüttel

21 November 2023

## Problems that we will definitely talk about

1. ***(Work in pairs – 15 minutes)***

   An influencer on Instagram has become famous for his frequent updates about the List monad. Yesterday the influencer presented a new function called fourfirst :

   ```
   fourfirst xs = do
                       x <- xs
                       return (4,x)
   ```

   "This function takes a list and gives us a pair $(4, x)$ where $x$ is the first element of the list", the influencer concluded. Soon after a heated discussion had begun among 4 million teenage followers: Was the influencer right? It is your task to find out. Explain, using the definition of the List monad *but without executing this piece of code*, what the code actually does and how it does it.

2. ***(Work in pairs – 30 minutes)***

   Here is a piece of Haskell code.

   ```
   data W x = Bingo x deriving Show

   instance Functor W where
     fmap f (Bingo x) = Bingo (f x)

   instance Monad W where
     return x = Bingo x
     Bingo x >>= f = f x
   ```

   - For this to make sense, a definition of W as an applicative functor is missing. Write such a definition.
   - Use do-notation to define a function wrapadd :: Num b =>b -> W b -> W b which satisfies that

     wrapadd x (Bingo y) = Bingo (x+y)

     ***Warning!*** Do not use pattern matching inside do-blocks. Only use monadic notation.
   - Use do-notation to define a function h which satisfies that

     h (Bingo x) (Bingo y) = Bingo (x∗y)

     And find its type without asking Haskell. ***Warning!*** Do not use pattern matching inside do-blocks. Only use monadic notation.

3. ***(Everyone at the table – 30 minutes)***

   Consider trees whose elements are values of some type in the type class Ord. The type Tree a is defined by

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

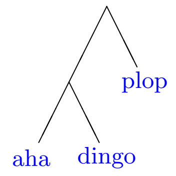


Figure 1: The ordered mytree1

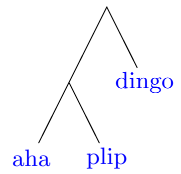


Figure 2: The unordered mytree2

*Use a monad* to write a function minorder that takes such a tree and checks if the numbers in the structure are in non-decreasing order when read from left to right. If it is, the function should return the smallest number in the tree, otherwise it should return Nothing. The tree mytree shown in Figure 1 is ordered, so minorder mytree1 should return Just "aha'', but the tree in Figure 2 is not ordered, so minorder mytree2 should return Nothing.

First define another function minmax that finds the minimal and the maximal element in a tree under the assumption that the tree is ordered. Then use minmax to define minorder.

*Hint:* Which monad should you use?

***Warning!*** Do not use pattern matching inside do-blocks. Only use monadic notation.

## More problems to solve at your own pace

a) Define a foldM function whose type should be

```
foldM :: Monad m => (t1 -> t2 -> m t2) -> [t1] -> t2 -> m t2
```

The idea is that the function works like foldl but folds over a monad.

Here is an example that shows what will happen if we fold over the IO monad. If we let

```
dingo x = do
            putStrLn (show x)
            return x
```

then we should see the following behaviour.

```
*Main> foldM (\x y -> (dingo (x+y))) [1,2,3,4] 0
1
3
6
10
```