# FUNCTIONAL PROGRAMMING MT2018

# SHEET 6

GABRIEL MOISE

## 11.1

Recall that

```
data Bool = False | True
```

Therefore, the natural fold for this type is

```
> foldBool :: a -> a -> Bool -> a

> foldBool false true False = false

> foldBool false true True  = true
```

As expected, we have foldBool False True = id, where id :: Bool -> Bool.

Now, we introduce the data type:

```
> data Day = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday
```

Therefore, the natural fold in this case is:

```
> foldDay :: a -> a -> a -> a -> a -> a -> a -> Day -> a

> foldDay sunday monday tuesday wednesday thursday friday saturday Sunday = sunday

> foldDay sunday monday tuesday wednesday thursday friday saturday Monday = monday

> foldDay sunday monday tuesday wednesday thursday friday saturday Tuesday = tuesday

> foldDay sunday monday tuesday wednesday thursday friday saturday Wednesday = wednesday

> foldDay sunday monday tuesday wednesday thursday friday saturday Thursday = thursday

> foldDay sunday monday tuesday wednesday thursday friday saturday Friday = friday

> foldDay sunday monday tuesday wednesday thursday friday saturday Saturday = saturday
```

And also here we have:

foldDay Sunday Monday Tuesday Wednesday Thursday Friday Saturday = id, where id :: Day -> Day

## 11.2

We have

False <= False

False <= True

True  <= True

and True <= False is wrong.

Now, we can think of a<=b as (not a) || b, which is formed of two logical functions, however we are asked to express it with a single logical expression.

We recall that False "implies" both True and False, but True only "implies" True, so we have a<=b is equivalent to a -> b, where "->" is the logical function “implies”.

## 11.3

```
> data Set a = Empty | Singleton a | Union (Set a) (Set a)

> foldSet :: b -> (a -> b) -> (b -> b -> b) -> Set a -> b

> foldSet empty singleton union = f

>           where f Empty = empty

>                 f (Singleton x) = singleton x

>                 f (Union a b) = union (f a) (f b)

> isIn :: Eq a => a -> Set a -> Bool

> isIn x  = foldSet False (== x) (||)
```

The first argument of foldSet is False because if we have an empty set, then obviously x is not in the set. Next, in the case we want to know if the x value is in a singleton or not, we simply compare the element from it with x, therefore we have the function (== x). Finally, in the case of a union, x appears in the union if x appears either in the left subset or if it appears in the right subset, therefore we have the function (||).

First of all, we will create a function list which, given a set, it returns the normal list which contains all the elements of the set:

```
> list :: Set a -> [a]

> list = foldSet [] f (++)

>          where f x = [x]
```

In the case when we have an Empty Set, we return [], if we have a Singleton x, we need a function that takes the x and returns [x], and the union function is simply (++).

For example,

```
*Main> list (Union (Union (Singleton 3) (Singleton 5)) Empty)

[3,5]
```

Now, we can create the subset function:

```
> subset :: Eq a => Set a -> Set a -> Bool

> subset set1 set2 = subsetList (list set1) set2


> subsetList :: Eq a => [a] -> Set a -> Bool

> subsetList [] _ = True

> subsetList (x:xs) set2 = (isIn x set2) && (subsetList xs set2)
```

We need an auxiliary function which, given the transformed set1, (with list), and the set2 argument, returns True only if all the elements from the list are in set2 (here we use the isIn function).

Therefore, we can create the instance of equality on the data type Set a, which is:

```
> instance Eq a => Eq (Set a) where

>     xs == ys = (xs `subset` ys) && (ys `subset` xs)
```

## 11.4

```
> data BTree a = Leaf a | Fork (BTree a) (BTree a)

> data Direction = L | R deriving (Show)

> type Path = [Direction]
```

We first write the natural fold of the BTree data type:

```
> foldBTree :: (a -> b) -> (b -> b -> b) -> BTree a -> b

> foldBTree leaf fork = f

>   where f (Leaf a) = leaf a

>             f (Fork l r) = fork (f l) (f r)
```

The function check verifies if we have a pathing for an element a when we get to the Leaf that contains x, so if they are equal, we have Just [] and we will form the pathing by adding the L or R direction to the list, or Nothing if a does not equal x (so the path up to that leaf is not correct and does not get us to the desired value we want to "find", which is a).

```
> check :: Eq a => a -> a -> Maybe Path

> check a x = if a == x then Just [] else Nothing
```

The function pathing checks if we have a path in the left subtree so then we add L to the list of directions (path) from Just. Then, we check for the right subtree (we need to do it in this order to find the leftmost occurrence). At the end, if we don't have any pathing up to that point, we just return Nothing.

```
> pathing :: Maybe Path -> Maybe Path -> Maybe Path

> pathing (Just xs) _ = Just (L : xs)

> pathing _ (Just xs) = Just (R : xs)

> pathing _ _ = Nothing
```

Now, the find function is simply the foldBTree with the functions (check a) and pathing, which will give us the desired solution:

```
> find :: Eq a => a -> BTree a -> Maybe Path

> find a = foldBTree (check a) pathing
```

## 12.1

First, we begin by representing the Queue1 type, which is a list of elements in the order in which they joined the queue.

```
> type Queue1 a = [a]

> empty1 :: Queue1 a

> empty1 = []

> isEmpty1 :: Queue1 a -> Bool

> isEmpty1 = null
```

The time complexity is O(1).

```
> add1 :: a -> Queue1 a -> Queue1 a

> add1 x xs = xs ++ [x]
```

The time complexity is O(n), where n is the length of the queue xs, as the (++) function needs a number of operations equal to the length of the left argument.

```
> get1 :: Queue1 a -> (a, Queue1 a)

> get1 xs

>   | isEmpty1 xs = error "Empty queue"

>   | otherwise = (head xs, tail xs)
```

The time complexity is O(1), as that's the complexity needed for the head and tail functions.

So, overall, if we want to have m operations of either isEmpty1, add1 or get1, in the worst case scenario, if the input contains m additions (add1 operations), the time complexity will be O(m*n).

Now, we represent the Queue2 type, which is a list of elements in the reversed order in which they joined the queue.

```
> type Queue2 a = [a]

> empty2 :: Queue2 a

> empty2 = []

> isEmpty2 :: Queue2 a -> Bool

> isEmpty2 = null
```

The time complexity is O(1).

```
> add2 :: a -> Queue2 a -> Queue2 a

> add2 = (:)
```

The time complexity is O(1), as that's the complexity for (:). Notice that now we add an element at the beginning of the list, compared to the previous type of queue where we added elements at the end (a new element is the last that joined the queue, so it's going to be the first in the list).

```
> get2 :: Queue2 a -> (a, Queue2 a)

> get2 xs

>   | isEmpty2 xs = error "Empty queue"

>   | otherwise = (last xs, init xs)
```

The time complexity here is O(n) as that's the complexity needed for the last and init functions.

So, overall, if we want to have m operations of either isEmpty2, add2 or get2, in the worst case scenario, if the input contains m "gets" (get2 operations), the time complexity will be O(m*n). Generally, we can’t have m “gets” without having m “additions”, but if we say that we have m/2 additions and m/2 “gets”, then the number of operations will be (m*n)/2+m/2, which has the complexity O(m*n).

Therefore, representing the queue type the other way around didn't change much.

Now, we implement the most efficient type Queue, which makes use of two lists, front and back, so that we can split the elements into them to get the time complexity for each operation to be O(1).

```
> data Queue a = Queue [a] [a] deriving (Show)

> empty :: Queue a

> empty = Queue [] []

> isEmpty :: Queue a -> Bool

> isEmpty (Queue xs ys) = (null xs) && (null ys)
```

Here, the time complexity is O(1).

```
> add :: a -> Queue a -> Queue a

> add x (Queue xs ys) = Queue xs (x:ys)
```

We put add an element in the "back" list using (:), so the complexity here is O(1).

```
> get :: Queue a -> (a, Queue a)

> get (Queue xs ys)

>   | isEmpty (Queue xs ys) = error "Empty queue"

>   | null xs = get (Queue (reverse ys) [])

>   | otherwise = ((head xs), Queue (tail xs) ys)
```

Here, first we check that the queue is non-empty. Then, if we want to get the oldest element of the queue in the case we have nothing in the front list, we transfer the back list into the front list and we apply get to it, as at the next recursive call we will get to the otherwise guard. There, as we have the front list non-empty, we simply get the head of the list and we continue with the queue that has the front list as the tail of the previous front list and with the same back list. So, although the reverse function has an O(n) complexity, where n is the length of the list to be reversed, we notice that every element we introduce gets in the back list, then it gets in the front list and eventually it is removed with the "get" function when it becomes the head of the list, so for each element added we make 2 operations for it to be removed.

To make things clearer, if we have k elements added, then for them to be removed we need to get them in the back list first, which is one operation for each, and then when we reverse the back, we simply need k operations(in total), so again, one operation for each element.

Therefore, if we have m operations, the complexity of all the process is linear in m so the complexity we have is going to be O(m).

## 12.2

```
> fib :: Int -> Integer

> fib 0 = 0

> fib 1 = 1

> fib n = fib (n-1) + fib (n-2)
```

We get that:

```
*Main> fib 10

55

(0.00 secs, 126,784 bytes)

*Main> fib 20

6765

(0.04 secs, 7,557,616 bytes)

*Main> fib 30

832040

(2.36 secs, 921,587,864 bytes)
```

The number of operations needed to find fib(n) is the number of operations needed to find fib(n-1) + the number of operations needed to find fib(n-2)+1 (the addition). So, basically, the time also grows with a factor of approximately 1.6 as in the case of the Fibonacci sequence (they have similar recurrence relations), so the complexity is exponential. That's why, for big values of n, we need a lot of time to get the desired result.

Now, we define the function two:

```
> two :: Int -> (Integer,Integer)

> two 0 = (0,1)

> two n = (y,x+y)

>           where (x,y) = two (n-1)
```

Notice that now, instead of having an exponential complexity, we have a linear complexity. For two n we need to make an addition of the elements of two (n-1) to determine the second element of the resulting pair, as the first one is the second elemtent from two(n-1). So, we need approximately n steps to calculate two n, and fib n is simply fst (two n).

```
> fib' :: Int -> Integer

> fib' = fst.two
```

Now, let's make some tests to see if the time has improved:

```
*Main> fib' 10

55

(0.00 secs, 70,680 bytes)

*Main> fib' 20

6765

(0.00 secs, 79,024 bytes)

*Main> fib' 30

832040

(0.00 secs, 87,440 bytes)
```

Recall that in the previous version we needed 2.36 secs to solve fib 30 and here we do this in almost no time. Additionally, trying to calculate fib' 100000 takes just 1.39 secs.

Now, how big is the 10000th Fibonacci number?

We'll use the function:

```
> roughly :: Integer -> String

> roughly n = [head xs] ++ "e" ++ show (length xs - 1)

>                    where xs = show n

*Main> roughly (fib' 10000)

"3e2089"

(0.03 secs, 10,135,216 bytes)
```

So, we can deduct from this that the 10 000th Fibonacci number has 2090 digits, the first one being 3.

The induction for $F^n$ is at the end because I found it easier to write it by hand, rather than type it.

Now, we use the "power" function from the lecture notes to calculate $F^n$:

First, we'll create the 2x2 identity matrix and the F matrix we begin with:

```
> i2 :: [[Integer]]

> i2 = [[1,0],[0,1]]

> f :: [[Integer]]

> f = [[0,1],[1,1]]
> power :: ([[Integer]] -> [[Integer]] -> [[Integer]]) -> [[Integer]] -> [[Integer]] -> Int -> [[Integer]]

> power (*) y x n

>     | n == 0 = y

>     | even n = power (*) y (x `mul` x) (n `div` 2)

>     | odd  n = power (*) (x `mul` y) x (n-1)
```

We use the multiplication operation for 2x2 matrices here:

```
> mul :: [[Integer]] -> [[Integer]] -> [[Integer]]

> mul [[x11,x12],[x21,x22]] [[y11,y12],[y21,y22]] =

>          [[x11*y11+x12*y21,x11*y12+x12*y22],[x21*y11+x22*y21,x21*y12+x22*y22]]
```

By using the power function to calculate $F^n$, we need 2*log n steps in the worst case scenario as after each step we either halve n, or we subtract one from n, which will then follow by a halving (as n will become even). Therefore, in the worst case scenario we need 2*log n matrix multiplications (which require 8 multiplications and 4 additions each).

To get the nth Fibonacci term we simply need to write (last.head) (power mul i2 f n) as we already proved in the induction that fib n is on the secondary diagonal of the $F^n$ matrix.

```
> fib'' :: Int -> Integer

> fib'' n = (last.head) (power mul i2 f n)
```

Example:

```
*Main> fib'' 30

832040

(0.00 secs, 83,696 bytes)
```

Now, let's see "roughly" how big the 1 000 000th Fibonacci number is:

```
*Main> roughly (fib'' 1000000)

"1e208987"

(0.03 secs, 14,208,384 bytes)
```

So, we found out that the 1 000 000th Fibonacci number has 208988 digits, the first one being 1.