# Functional Programming: Assignment 7

Group 60

Lucas van der Laan
s1047485

## 1 Warm-up

### 1

The best function is null, as it does a foldr that simply checks if the list has any elements, if it does, return False, if it doesn’t, return true. The other good function is the case argument, because we do not have an EQ constraint like we do with list == []. The one that we generally want to avoid is the length function, because it needs to go through the entire list to calculate the length, this leads to bad performance, while the rest is constant time.

### 2

justs :: 1 → 2
justs xs = [ x | Just x ← xs, x /= ’ ’ ]
xs :: 1 = ?
ys :: 2 = ?
left :: 3 = ?
right :: [3] = ?
1 = [3]
justs :: [3] → 2
Just x, thus x is a Maybe, so that makes 3 = Maybe a, 1 = [Maybe a]
x /= ’ ’ → x = Char, x :: a, so a = Maybe Char, so 1 = [Maybe Char]
The result of the list interpolation is x, thus 2 = [Char]
justs :: [Maybe Char] → [Char]

orderPairs :: 1 → 2
orderPairs xs = map (\(x,y)→(min x y, max x y)) xs
orderPairs :: [3] → 2, where 1 = [3]
orderPairs :: [(4,4)] → 2, where 3 = (4,4)
orderPairs :: Ord 4 => [(4,4)] → 2, where 3 = (4,4)
orderPairs :: Ord 4 => [(4,4)] → [(4,4)], where 2 = (4,4)
orderPairs :: Ord a => [(a,a)] → [(a,a)], where a = 4

unmaybe :: 1 → 2
unmaybe :: 3 → 2, where 3 = Maybe 1
unmaybe :: 3 → 4, where 4 = Maybe 2
unmaybe :: 5 → 4, where 5 = Maybe 3

unmaybe :: Maybe (Maybe a) → Maybe a

accumulate :: 1 → 2 → 3
accumulate :: (2 → (4,2)) → 2 → 3, 1 is a function applied to 2 and results in (4,2)
accumulate :: (2 → (4,2)) → 2 → [4], we recursively add 4 : accumulate 1 2
accumulate :: (b → (a,b)) → b → [a]

### 3

```
mapFilter :: (a -> Maybe b) -> [a] -> [b]
mapFilter f = map $ (\(Just x) -> x) . f

lift :: (a -> b -> Maybe c) -> (Maybe a -> Maybe b -> Maybe c)
lift f (Just x1) (Just y1) = f x1 y1

compute :: (Monoid n) => (a -> n) -> [a] -> n
compute f = mconcat . map f

fuse :: (a -> b -> c) -> (a -> b) -> a -> c
fuse fa fb x = fa x (fb x)
```

## 2 Mandatory

### 4

```
frequencies :: (Ord a) => [a] -> [(a,Int)]
frequencies = map (\x -> (head x, length x)) . group . sort
```

### 5

```
-- Left it like this as this was my thougt process
huffman :: [(a, Int)] -> Btree a
huffman = head . map fst . step3
  where
    step1 = map (\(x,i) -> (Tip x, i))
    step2 = sortOn snd . step1
    step3 = step3Helper . sortOn snd . step2
    step3Helper :: [(Btree a, Int)] -> [(Btree a, Int)]
    step3Helper [] = []
    step3Helper [x] = [x]
    step3Helper (x:y:ys) =
      step3Helper $
      sortOn snd ((Bin (fst x) (fst y), snd x + snd y) : ys)
```

## 6

See Huffman.hs lines 33-50

## 7

See Huffman.hs lines 54-66
I don't understand how to get explicit types working for findItem and decodeHelper, when I make the functions global functions they work, but as helper functions they don't.