Solution sheet 3

Introduction

Please note that there can be other solutions than those listed in this document.

This is a literate Haskell file which is available as PDF, as well as literate Haskell source code (.lhs extension). If you are looking at the PDF version, you should download the .lhs file, which can be loaded into ghci or compiled with ghc. This document uses ***Markdown*** syntax, compiled to PDF using *Pandoc*.

Everything that is not enclosed in a code block like below is treated as text.

```
-- This is code
import Data.List -- Used for sorting later
main :: IO ()
main = undefined
```

1 - Replicate

To avoid nameclash with the original `replicate` function, we choose to call it `replicate'`. Finding a valid implementation of `replicate` should be quite straightforward after completing exercise sheet 2.

```
replicate' :: Int -> a -> [a]
replicate' 0 _ = []
replicate' n x = x : replicate' (n-1) x
```

2 - zipIdx

In zipIdx, we can make use of helper functions to keep track of the current index. There are many other implementations.

```
zipIdx :: [a] -> [(a, Int)]
zipIdx []   = []             -- catches empty lists
zipIdx list = combine 0 list -- list has some content
  where
    combine _ []       = []
    combine n (y : ys) = (y, n) : combine (n+1) ys
```

Or one could simply:

```
zipIdx' :: [a] -> [(a, Int)]
zipIdx' l = zip l [0..]
```

3 - setIdx

It is a design decision whether `setIdx` should add an element to an empty list when given index 0. Let's assume that this is a desired property. Doing this, we will also be able to append to the end of a list, but the code is not quite as nice, as errors could occur.

```
setIdx :: [a] -> Int -> a -> [a]
setIdx [] 0 x = [x] -- Described case
setIdx [] _ _ = error "List too short, or negative index given"
setIdx (_:ys) 0 x = x : ys -- At desired idx
setIdx (y:ys) n x = y : setIdx ys (n-1) x -- Keep looking for idx
```

4 - modIdx

In this function, inserting new elements is not an option.

```
modIdx :: [a] -> Int -> (a -> a) -> [a]
modIdx [] _ _     = []
modIdx (y:ys) 0 f = f y : ys -- At desired idx
modIdx (y:ys) n f = y : modIdx ys (n-1) f -- Keep looking for idx
```

5 - duplicate elimination

a - nubEq

One way of doing this, would be by removing all duplicates of an element from the rest of a list, as it is added to the result. This could be implemented using recursion and list comprehensions.

```
nubEq :: Eq a => [a] -> [a]
nubEq []       = []
nubEq (x : xs) = x : nubEq [y | y <- xs, y /= x]
```

Complexity $\frac{n(n-1)}{2} \equiv (n - 1 + n - 2 + \ldots + 1)$

b - nub (no typeclass)

It does not really make sense to remove duplicates when duplicates are something equal to something else already in the list, when we cannot check for equality.

c - nubOrd

Including `Ord` is a good indication that we should consider sorting. As known, sorting can be done in a worst case complexity of `n log(n)`. While we could implement a sort ourselves, which eliminated duplicates on-the-fly, this implementation will simply use the build in sort, and do a single run-through afterwards.

```
nubOrd :: Ord a => [a] -> [a]
nubOrd []   = []
nubOrd list = nubOrd' (sort list)
  where
    nubOrd' []  = []
    nubOrd' [x] = [x]
    nubOrd' (x:y:xs)
              | x == y    = nubOrd' (y:xs)
              | otherwise = x : nubOrd' (y:xs)
```

6 - elemCounts

a - elemCountsEq

The following is an option, without proof of optimality.

```
elemCountsEq :: Eq a => [a] -> [(a, Int)]
elemCountsEq []      = []
elemCountsEq l@(x:_) = (x, length getEq) : elemCountsEq getNEq
  where
    getEq  = [ y | y <- l, y == x]
    getNEq = [ y | y <- l, y /= x]
```

Computing the two lists (`getEq` & `getNEq`) take $|l|$ time each. `length` will eventually go through every element of the original list, no more and no less. Therefore, the worst case is a list where all elements are distinct. In that case, we remove a single element between each recursive call. We then have that the complexity will look something like $(n + 2\frac{n(n+1)}{2})$, where $\frac{n(n+1)}{2} \equiv (n + n - 1 + n - 2 + \ldots + 1)$.

b - elemCountsOrd

Again, using the Ord typeclass indicates that one should consider sorting.

```
elemCountsOrd :: Ord a => [a] -> [(a, Int)]
elemCountsOrd [] = []
elemCountsOrd l  = elemCountsOrd' (first, 1) rest
  where
    (first:rest) = sort l -- lets us access first element immediately
    elemCountsOrd' (cur, occ) [] = [(cur, occ)]
    elemCountsOrd' (cur, occ) (x:xs)
      | cur == x  = elemCountsOrd' (x, occ+1) xs
      | otherwise = (cur, occ) : elemCountsOrd' (x, 1) xs
```

The complexity of this version with ordering, is somthing like $n + n\ log(n)$. Additionally, this version will have the result sorted.

7 - fromElemCounts

We can use the function `replicate` defined earlier, to shorten this function a bit. Additionally, the `++` operator combines two lists some examples:

`[1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]`

`[1,2,3] ++ [1,2,3] = [1,2,3,1,2,3]`

We can define `fromElemCounts`:

```
fromElemCounts :: [(a, Int)] -> [a]
fromElemCounts [] = []
fromElemCounts ((x, n):xs) = replicate n x ++ fromElemCounts xs
```

8 - Probability Distributions

We define the type:

```
type Dist a = [(a, Double)]
```

a - uniformly

```
uniformly :: [a] -> Dist a
uniformly l = zip l (repeat prob)
  where
    prob = 1 / fromIntegral (length l)
```

The probability of each element of a list l, in a uniform distribution, can be described as $\frac{1}{|l|}$

b - uniformlyEq & uniformlyOrd

We can use the previously defined `elemCounts` functions to simplify these tasks.

```
uniformlyEq :: Eq a => [a] -> Dist a
uniformlyEq l = zip (map fst ecounts) probs
  where
    ecounts = elemCountsEq l
    prob a = fromIntegral a / fromIntegral (length l)
    probs = [prob p | (_, p) <- ecounts]
```

```
uniformlyOrd :: Ord a => [a] -> Dist a
uniformlyOrd l = zip (map fst ecounts) probs
  where
    ecounts = elemCountsOrd l
    prob a = fromIntegral a / fromIntegral (length l)
    probs = [prob p | (_, p) <- ecounts]
```

Again, the `Ord` version produces sorted output.

c - join & flatten Dists

Calling joinDists with the input

[(True, 0.5), (False, 0.5)] [(0, 0.5), (1, 0.5)]

Should return

[((True,0),0.25),((True,1),0.25),((False,0),0.25),((False,1),0.25)]

We can implement this using a single *list comprehension*:

```
joinDists :: Dist a -> Dist b -> Dist (a, b)
joinDists a b = [((x, y), p) | (x, xp) <- a
                              , (y, yp) <- b
                              , let p = xp * yp]
```

`a` and `b` are distributions, `x` takes the value of each element in `a`, `xp` takes the probability, `y` and `yp` do the same for `b`. The probability of getting a specific combination from the two distributions, can be calculated by multiplying the probability of `x` in `a` with the probability of `y` in `b`.

Calling `flattenDist` on

[([(0,0.5),(1,0.5)],0.5),([(2,0.5),(3,0.5)],0.5)]

Should return

[(0,0.25),(1,0.25),(2,0.25),(3,0.25)]

Again, a single list comprehension is sufficient.

```
flattenDist :: Dist (Dist a) -> Dist a
flattenDist dd = [(x, xp) | (d, dp) <- dd
                           , (x, p)  <- d
                           , let xp = dp * p]
```

`dd` is the distribution of distributions, `d` takes one distribution from `dd` at a time, `dp` takes the probability of probability `d`, `x` takes the value of one element of `d` at a time, while `p` takes the probability of `x` within probability `d`. The probability of `x` occuring in the flattened distribution, can then be calculated by multiplying it by `dp`, the probability of `d` within `dd`.