Oplossingen Week 1

Dit oplossingenbestand is geschreven in Bird Style Literate Haskell met markdown syntax, en met behulp van Pandoc omgezet naar PDF-formaat. Dit houdt in dat je het `.lhs` bestand gewoon kan compileren met ghc of interactief inladen met `ghci -optL -q`.

Net zoals in een gewoon Haskell bronbestand plaatsen we de imports bovenaan.

```
import Data.Maybe (fromJust)
```

Het voorlaatste element

Om het voorlaatste element uit een lijst te halen, zijn verschillende methoden mogelijk. Een eerste methode zal eerst de lengte van de lijst berekenen, en vervolgens het voorlaatste element via index opvragen.

```
voorlaatste :: [Int] -> Int
voorlaatste l = l !! (length l - 2)
```

Anders kan je ook een recursieve oplossing schrijven.

```
voorlaatste' :: [Int] -> Int
voorlaatste' [x, y] = x
voorlaatste' (x:xs)   = voorlaatste' xs
```

Hier geven we verschillende definities voor `voorlaatste'`: als we voorlaatste toepassen op een lijst met 2 elementen, geven we gewoon het eerste terug. Tenslotte is dat dan het voorlaatste element. Als die eerste definitie niet past, zullen we de tweede proberen toepassen. Hier matchen we het gegeven argument met `(x:xs)`. Dit zal onze lijst deconstrueren in een eerste element `x` en de rest van de lijst, `xs`. Omdat onze oplossing niet moet werken voor te kleine lijsten, gaan we er vanuit dat onze lijst minstens 3 elementen lang is. Het voorlaatste element hiervan is gewoon het voorlaatste element van de rest.

Tenslotte kunnen we ingebouwde functies gebruiken om het voorlaatste element te bepalen, met name:

```
init :: [a] -> [a]
last :: [a] -> a
```

die respectivelijk alles behalve het laatste element en het laatste element van de gegeven lijst zullen teruggeven.

```
voorlaatste'' :: [Int] -> Int
voorlaatste'' l = last (init l)
```

Dit kunnen we nu nog korter schrijven met behulp van de functie-chaining operator `.` (uitgesproken als “na”):

```haskell
(.) :: (b -> c) -> (a -> b) -> (a -> c)

voorlaatste''' :: [Int] -> Int
voorlaatste''' = last . init
```

Het voorlaatste element is dus “last na init”: we nemen het laatste element nadat we alles behalve het laatste element namen.

Nde Element van een lijst

In vorige opgave zagen we al kort `!!` gebruikt worden.

```haskell
geefNde :: Int -> [Int] -> Int
geefNde n l = l !! n
```

We gebruiken hier de infix-notatie, wat standaard is voor functies wiens naam enkel uit niet-alfanumerieke tekens bestaat. We kunnen `!!` ook in prefix-notatie gebruiken door er haakjes rond te plaatsen, of `geefNde` in infix-notatie door er backticks rond te zetten. Volgende definities zijn allemaal equivalent.

```haskell
geefNde n l = (!!) l n
n `geefNde` l = l !! n
n `geefNde` l = (!!) l n
```

We zien hoe `geefNde` eigenlijk gewoon `!!` is, maar dan met zijn argumenten van volgorde gewisseld. We kunnen dus ook gebruik maken van de ingebouwde `flip` functie, die een functie als argument neemt en diezelfde functie met gewisselde argumenten teruggeeft.

```haskell
flip :: (a -> b -> c) -> (b -> a -> c)

geefNde' :: Int -> [Int] -> Int
geefNde' = flip (!!)
```

Berekenen of een lijst een palindroom is

Merk eerst kort op dat in Haskell een string gewoon een lijst van karakters is.

```haskell
"asdf" :: [Char]
```

Volgende oplossing zal dus werken op strings.

```haskell
palindroom :: [Char] -> Bool
palindroom l = reverse l == l
```

Verdubbelen van lijsten

Het functies met lijsten kunnen we op vaak op drie standaard manieren oplossen: met recursie, met *list comprehensions* en met de ingebouwde `map` functie. We

verkiezen het gebruik van `map` over LCs over recursie, maar leesbaarheid is de belangrijkste factor.

Om deze opgave bijvoorbeeld recursief op te lossen, schrijven we volgende definitie.

```
verdubbel :: [a] -> [a]
verdubbel []     = []
verdubbel (x:xs) = x : x : verdubbel xs
```

We gebruiken hier een algemeen type `a` omdat het ons niet echt kan schelen wat er in de lijst zit. Omdat dit een eenvoudige opgave is, is ook deze recursieve definitie goed leesbaar. Ze is echter al een stuk langer dan de volgende oplossing met behulp van een LC.

```
verdubbel' :: [a] -> [a]
verdubbel' l = concat [ [x, x] | x <- l ]
```

Wie vertrouwt is met de concat-functie, ziet in één oogopslag wat deze functie doet. Zo ook voor de oplossing met behulp van `map`.

```
verdubbel'' :: [a] -> [a]
verdubbel'' l = concat (map (\x -> [x, x]) l)
```

Dit is ook mooier te schrijven:

```
verdubbel'' l = concat (map (\x -> [x, x]) l)
    -- `a $ b` is synoniem voor (a) (b)
verdubbel'' l = concat $ map (\x -> [x, x]) l
    -- `f $ g x` is steeds te schrijven als `f . g $ x`
verdubbel'' l = concat . map (\x -> [x, x]) $ l
    -- `g x = f x` is hetzelfde als `g = f`
verdubbel'' = concat . map (\x -> [x, x])
```

Bovenstaand combinatie van `concat` en `map` komt vaak voor, dus bestaat de combinatie ook.

```
verdubbel''' :: [a] -> [a]
verdubbel''' = concatMap (\x -> [x, x])
```

Ritsen van lijsten

Natuurlijk vonden jullie op hoogle allemaal dat `zip` reeds bestaat, maar hieronder vinden jullie nog een recursieve oplossing.

```
rits :: [a] -> [b] -> [(a, b)]
rits []     _      = []
rits _      []     = []
rits (x:xs) (y:ys) = (x, y) : rits xs ys
```

Omdat we hier te maken krijgen met twee lijsten die we tegelijkertijd doorlopen, zal recusie een stuk makkelijker zijn dan LCs of ingebouwde functies.

Lijsten splitsen

```
splits :: Int -> [a] -> ([a], [a])
splits 0 l      = ([], l)
splits n (l:ls) = let (first, second) = splits (n - 1) ls
                  in (l:first, second)
```

De eerste oplossing maakt gebruik van recursie. We reduceren op het aantal elementen dat we afsplitsen, met triviaal basisgeval: geen elementen afsplitsen. Het recursief geval zal een oproep doen waarin we 1 element minder afsplitsen van een ingekorte lijst. Merk op hoe we *let-in* kunnen gebruiken om ons tuple te deconstrueren.

```
splits' :: Int -> [a] -> ([a], [a])
splits' n l = ( [ x | (i, x) <- zip [0..] l, i < n ]
              , [ x | (i, x) <- zip [0..] l, i >= n ]
              )
```

Hier gebruiken we *list comprehensions*. We ritsen onze lijst met een oneindige lijst startende met 0, waardoor we indexering krijgen. Door dan te kijken of die index voor of na `n` ligt, splitsen we onze lijst. Let wel op: deze oplossing zal niet zo goed werken op oneindige lijsten. Als `l` een oneindige lijst is, zal bij de andere oplossingen het eerste deel wel een eindige lijst zijn, waar dit hier niet het geval is. Onze LC kan namelijk niet weten dat de conditie nooit terug "waar" zal worden verderop in de lijst.

```
splits'' :: Int -> [a] -> ([a], [a])
splits'' n l = (take n l, drop n l)
```

Als je dit gevonden had, kon je eigenlijk ook gewoon `splits = splitAt` schrijven.

Elementen uit een lijst verwijderen

```
verwijderK :: Int -> [a] -> [a]
verwijderK 0 (l:ls) = ls
verwijderK n (l:ls) = l : verwijderK (n - 1) ls
```

```
verwijderK' :: Int -> [a] -> [a]
verwijderK' n l = let (fsts, snd:snds) = splitAt n l
                  in fsts ++ snds
```

```
verwijderK'' :: Int -> [a] -> [a]
verwijderK'' n l = let (fsts, snds) = splitAt n l
```

```
                    in fsts ++ tail snds
verwijderK''' :: Int -> [a] -> [a]
verwijderK''' n l = [ x | (i, x) <- zip [0..] l, i /= n ]
```

Roteren van lijsten

```
roteer :: Int -> [a] -> [a]
roteer 0 l = l
roteer n (l:ls) = roteer (n - 1) (ls ++ [l])
```

En een mooie oplossing met cycle (zie hoogle) waar ik zelf niet aan dacht, geïnspireerd door QDV.

```
roteer' :: Int -> [a] -> [a]
roteer' n l = take (length l) $ drop n $ cycle l
```

Valentijnsdag

Deze tabellen waren gegeven in de opgave.

```
maandgetal, jaargetal :: [Int]
eeuwgetal :: [(Int, Int)]
weekdagen :: [String] -- ofwel [[Char]], want type String = [Char]

maandgetal = [ 0, 3, 3, 6, 1, 4, 6, 2, 5, 0, 3, 5 ]
jaargetal  = [ 0, 1, 2, 3, 5, 6, 0, 1, 3, 4, 5, 6, 1, 2
             , 3, 4, 6, 0, 1, 2, 4, 5, 6, 0, 2, 3, 4, 5 ]
eeuwgetal  = [ (15, 0), (19, 0), (23, 0), (16, 6)
             , (20, 6), (24, 6), (17, 4), (21, 4)
             , (25, 4), (18, 2), (22, 2), (26, 2) ]
weekdagen  = [ "zondag", "maandag", "dinsdag", "woensdag"
             , "donderdag", "vrijdag", "zaterdag" ]
```

Geef het maandgetal, jaargetal, eeuwgetal en dag terug uit de tabel.

```
zoekMaandgetal, zoekJaargetal, zoekEeuwgetal :: Int -> Int
zoekWeekdag :: Int -> [String] -> String

zoekMaandgetal m = maandgetal !! m
zoekJaargetal  n = jaargetal !! (n `mod` length jaargetal)
zoekEeuwgetal  n = fromJust $ lookup n eeuwgetal
zoekWeekdag x l = l !! mod (mod x 7 + 7) 7
```

Geef terug of een jaar een schrikkeljaar is of niet. We herinneren ons: een jaar is een schrikkeljaar als het deelbaar is door 4, tenzij het deelbaar is door 100, tenzij het deelbaar is door 400.

```haskell
schrikkeljaar :: Int -> Bool
schrikkeljaar j | mod j 400 == 0 = True
                | mod j 100 == 0 = False
                | otherwise      = mod j 4 == 0
```

We gebruiken hier een `guard` clause, geschreven als `|`. Zo kunnen we extra voorwaarden koppelen aan onze gematchte argumenten. Weetje: De otherwise is geen speciale syntax: `otherwise = True`.

```haskell
weekdag :: Int -> Int -> Int -> Int -> Int
weekdag d m e j = (i + if voorSchrikkeldag then -1 else 0) `mod` 7
  where
    i = sum [d, zoekMaandgetal m, zoekJaargetal j, zoekEeuwgetal e]
    -- in een schrikkeljaar voor 29 februari
    voorSchrikkeldag = schrikkeljaar (100 * e + j) && m <= 2
```

Gegeven de eeuw en het jaar geef de weekdag waarop valentijn valt dat jaar.

```haskell
valentijn :: Int -> Int -> String
valentijn e j = zoekWeekdag (weekdag 14 2 e j) weekdagen
```