# Parel 6
# Functioneel Programmeren
# Toets

26 september 2013

*Alle opgaven tellen even zwaar.*

*U mag bijgevoegde lijst van Haskell operaties en functies gebruiken.*

**Opgave 1.** Iemand wil een recursieve functie `remdups` (voor "remove duplicates") schrijven die alle dubbele voorkomens van een element uit een lijst verwijdert. Bijvoorbeeld:

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

Het basisgeval van de recursieve definitie is:

```
remdups [] = []
```

Het recursieve geval is

```
remdups (x:xs) = ...
```

Welke van onderstaande expressies moet hij kiezen voor het recursieve geval?

*a.* `x : remdups [ y | y <- xs , y /= x ]`

*b.* `remdups [ y | y <- xs , y /= x ]`

*c.* `[ y | y <- xs , y /= x ]`

*d.* `x : remdups xs`

**Opgave 2.** Schrijf een functie die een gegeven lijst splitst in lijsten van gelijke elementen uit de gegeven lijst.

Bijvoorbeeld:

```
splitsGelijken [1,3,2,4,3,2,1,2,1] = [[1,1,1],[3,3],[2,2,2],[4]]
```

Hierbij is de volgorde waarin de lijsten van gelijke elementen worden gegeven niet van belang, dus bijvoorbeeld `[[4],[3,3],[2,2,2],[1,1,1]]` is ook goed.

**Opgave 3.** Een gereedschapswinkel heeft een voorraadadministratie van de volgende vorm:

```
voorraad = [("hamer",8,40), ("beitel",4,25), ("zaag",6,30), ..]
```

In ieder 3-tupel staat (van linksnaar rechts) het artikel, het aantal dat op voorraad is, en de prijs per stuk. Schrijf een functie `totaleWaarde` die de waarde van de hele voorraad uitrekent.

Als in het voorbeeld de voorraad niet groter zou zijn dan de drie genoemde artikelen, zou de totale waarde van de hele voorraad dus 600 euro zijn.

Z.O.Z.

**Opgave 4.** Gegeven is een lijst van $n$ lijsten die allemaal $n$ elementen bevatten — dus als je de lijsten onder elkaar zou zetten, zou je een "vierkant" van $n \times n$ elementen krijgen. Schrijf een functie `nevendiagonaal` die bij zo'n gegeven vierkant de lijst van elementen oplevert die op de nevendiagonaal (van rechtsboven naar linksonder) van dat vierkant staan. Bijvoorbeeld:

```
nevendiagonaal [[1,2,3],[4,5,6],[7,8,9]] = [3,5,7]
```

## Some standard Haskell operators and functions

**Remark.** The list below presents the standard Haskell types, *without* the type `Number` as used in this course.

```
negate, abs,
signum          :: Num a => a -> a
+, -, *         :: Num a => a -> a -> a
div, mod        :: Integral a => a -> a -> a
/               :: Fractional a => a -> a -> a
^               :: (Num a, Integral b) => a -> b -> a
abs, exp, log,
sqrt, sin,
cos             :: Floating a => a -> a
```
various arithmetical operations and functions

```
min, max        :: Ord a => a -> a -> a
```
gives the minimum, maximum of two arguments

```
not             :: Bool -> Bool
&&, ||          :: Bool -> Bool -> Bool
```
boolean operations negation, conjunction, disjunction

```
isLower,
isUpper         :: Char -> Bool
```
says whether a letter is lower-case or upper-case

```
isAlpha         :: Char -> Bool
```
says whether a character is a letter

```
isDigit         :: Char -> Bool
```
says whether a character is a digit

```
isAlphaNum      :: Char -> Bool
```
says whether a character is a letter or a digit

```
ord             :: Char -> Int
```
converts a character to its Unicode number

```
chr             :: Int  -> Char
```
converts a Unicode number to the corresponding character

```
show            :: Show a => a -> String
```

Converts a showable datatype (e.g. Int, Float) to a String

`read` :: `Read a => String -> a`
Converts a String to readable datatype (e.g. Int, Float)

`toLower`, `toUpper` :: `Char -> Char`
converts a letter to lower-case, upper-case

`==`, `/=` :: `Eq a => a -> a -> Bool`

`>`, `>=`, `<`, `<=` :: `Ord a => a -> a -> Bool`
various comparison operations

`even`, `odd` :: `Integral a => a -> Bool`
says whether a (integral) number is even or odd

`:` :: `a -> [a] -> [a]`
adds element to the front end of a list (*cons*)

`length` :: `[a] -> Int`
length of a list

`!!` :: `[a] -> Int -> a`
list indexing

`++`, `\\` :: `[a] -> [a] -> [a]`
list concatenation, list subtraction

␣ :: `(a->b) -> a -> b`
function application

`$` :: `(a->b) -> a -> b`
function application operator, that has a low right-associative binding precedence

`.` :: `(b->c) -> (a->b) -> (a->c)`
function composition

`head`, `last` :: `[a] -> a`

`tail`, `init`, `reverse` :: `[a] -> [a]`

`elem` :: `Eq a => a -> [a] -> Bool`

tests whether a list contains a given element

```
concat      :: [[a]] -> [a]
```
concats a list of lists into one list

```
sort        :: Ord a => [a] -> [a]
merge       :: Ord a => [a] -> [a] -> [a]
```
merges two sorted list into a single, sorted whole

```
sum         :: Num a => [a] -> a
minimum,
maximum     :: Ord a => [a] -> a
take, drop  :: Int -> [a] -> [a]
takeWhile,
dropWhile   :: (a->Bool) -> [a] -> [a]
```
various functions on lists

```
splitAt     :: Int -> [a] -> ([a],[a])
```
splits a list in the first $n$ elements and the rest

```
span        :: (a -> Bool) -> [a] -> ([a],[a])
```
splits a list at the first positon where property $p$ is not satisfied

```
insert      :: Ord a => a -> [a] -> [a]
```
inserts an element into an ordered list

```
and, or     :: [Bool] -> Bool
```
yields the conjunction, disjunction of a list of booleans

```
lines       :: String -> [String]
```
breaks a string at newlines (`'\n'`) into a list of strings

```
unlines     :: [String] -> String
```
glues a list of strings with `'\n'`

```
fst         :: (a,b) -> a
```
yields the first element of a pair

```
snd         :: (a,b) -> b
```
yields the second element of a pair

```
zip         :: [a] -> [b] -> [(a,b)]
```
turns two lists into a list of pairs

`unzip` :: `[(a,b)] -> ([a],[b])`
turns a list of pairs into a pair of lists

`zipWith` :: `(a->b->c) -> [a] -> [b] -> [c]`
zips two lists and applies a function to the corresponding elements

`map` :: `(a->b) -> [a] -> [b]`
applies a function to all elements in a list

`filter` :: `(a->Bool) -> [a] -> [a]`
selects those elements from a list which satisfy a property

`foldl` :: `(a->b->a) -> a -> [b] -> a`
"folds" a list with a function, starting with a given value. Works from left to right through the list

`foldr` :: `(a->b->b) -> b -> [a] -> b`
like `foldl`, but works from right to left

`foldl1`, `foldr1` :: `(a->a->a) -> [a] -> a`
like `foldl`, `foldr`, with first, last element of the list as starting value. Error for empty list

`scanl` :: `(a->b->a) -> a -> [b] -> [a]`
like `foldl`, but yielding intermediate results too

`scanr` :: `(a->b->b) -> b -> [a] -> [b]`
like `foldr`, but yielding intermediate results too

`seq` :: `a -> b -> b`
partially evaluates first argument, and delivers the second

`error` :: `String -> a`
causes error with given string as error message