Hertentamen Functioneel Programmeren—04 februari 2009

De nagekeken tentamens zijn in te zien bij de docent, J.H. Jongejan, Bernoulliborg kamer 366.

Opmerkingen:

- Schrijf **netjes** en duidelijk, met zwarte of blauwe pen.
- Zet op het eerste blad alle gegevens als naam, etc., en het totaal aantal ingeleverde bladen, en nummer de ingeleverde bladen.
- Lees de opgaven eerst goed door.
- Houd je programma's kort en helder, mede door verstandig gebruik te maken van standaardfuncties uit het boek (in het bijzonder uit het gedeelte over lijsten) en/of door listcomprehension.
- Motiveer je antwoorden.

1. (20 punten)

   a) Geef het type én de implementatie van `foldr`.
   b) Geef het type én de implementatie van `curry`.
   c) Geef het type én de implementatie van `id`.
   d) Leidt het type af van de expressie `curry id`.

2. (15 punten)

   Gegeven is de functie

```
shunt :: [a] -> [a] -> [a]
shunt [] ys = ys                    -- (sh.1)
shunt (x:xs) ys = shunt xs (x:ys)   -- (sh.2)
```

   De functie `shunt` kan gebruikt worden om een heel efficiënte versie van `reverse` te maken. In deze opgave gaan we daar echter niet op in.

   Bewijs met volledige inductie over alle eindige lijsten `xs` dat voor willekeurige eindige lijsten `zs`

```
shunt (shunt xs zs) [] = shunt zs xs  -- (sh.3)
```

3. (25 punten)

We kunnen een abstract data type maken voor een **set** middels een `module Set`:

```
module Set
  (Set,              -- constructor
   empty,            -- Set a
   isEmpty,          -- Set a -> Bool
   addElem,          -- Ord a => a -> Set a -> Set a
   makeSet,          -- Ord a => [a] -> Set a
   union,            -- Ord a => Set a -> Set a -> Set a
   intersect         -- Ord a => Set a -> Set a -> Set a
   difference,       -- Ord a => Set a -> Set a -> Set a
  ) where
    data Set a = S [a]
    ...
```

Aan de `data` definitie zien we dat een lijst wordt gebruikt om de **set** te implementeren. Die lijst mag dus geen duplicaten bevatten! Voor een efficiënte implementatie is het vereist dat we de lijst geordend houden.
a) Geef de implementatie van `empty`.
b) Geef de implementatie van `addElem`.
c) Geef de implementatie van `makeSet`.
d) Geef de implementatie van `union`.
Het is mogelijk dat je nog één of meer extra hulpfuncties moet definiëren. Pas op: we willen met echte sets werken, niet met bags (ook wel: multi-sets).

4. (20 punten)

De Hamming getallen zijn alsvolgt gedefinieerd:

$$\{2^i 3^j 5^k \mid i, j, k \geq 0, i + j + k > 0\}$$

De Hamming getallen beginnen met: $2, 3, 4, 5, 6, 8, 9, 10, 12, 15$.

Gevraagd: definieer de functie `hamming`, die de (oneindige) lijst van Hamming getallen oplevert. Zorg ervoor dat de Hamming getallen in oplimmende volgorde worden berekend en maar één keer getoond worden (bijv. 2*6 en 3*4 leveren beide 12 op).

5. (20 punten)

Gegeven is dat de invoer voldoet aan de grammaticaregels:

```
E = Num | E ';' E
Num = D | Num D
D = '0'|'1'|..|'9'
```

a) Geef een data definitie Exp om E's te representeren.
b) Bouw een parser `pExp :: Parse Char Exp`, die de input (`[Char]`) omzet naar een `Exp`.
Je mag hierbij gebruik maken van:

```
type Parse a b = [a] -> (b,[a])

succeed :: b -> Parse a b
spot    :: (a -> Bool) -> Parse a a
token t = spot (==t)
alt     :: Parse a b -> Parse a b -> Parse a b
(>*>)   :: Parse a b -> Parse a c -> Parse a (b,c)
build   :: Parse a b -> (b -> c) -> Parse a c
neList  :: Parse a b -> Parse a [b]
```

Hierbij is `neList` een parser voor een niet-lege lijst van objecten. Alle andere hulpfuncties moet je definiëren.