Eindtoets

INTRODUCTIE

De cursus Functioneel programmeren wordt afgesloten met een schriftelijk tentamen van twee uur. Het is toegestaan om tijdens het tentamen gebruik te maken van het cursusmateriaal. Het is echter niet toegestaan om tijdens het tentamen gebruik te maken van elektronische hulpmiddelen. Ook mogen apparaten waarop een Haskell-compiler of -interpretator aanwezig is niet worden gebruikt. Het schriftelijk tentamen bestaat uit een aantal open vragen; de opgaven zijn in aantal en moeilijkheidsgraad vergelijkbaar met de opgaven van deze eindtoets.
De terugkoppeling met de antwoorden op de opgaven staat direct na de toets zelf. We willen benadrukken dat u het meeste van deze toets leert als u eerst de opgaven allemaal maakt en pas daarna de antwoorden controleert. Maak de toets alsof het om een tentamen zou gaan. Dat wil zeggen: werk er in een aaneengesloten periode aan en gebruik hiervoor niet meer dan twee uur.

De eindtoets bestaat uit vier open vragen. Voor deze vragen samen kunt u maximaal 100 punten behalen. Voor een voldoende moet u een score hebben van minimaal 55 punten. De puntenwaardering staat per onderdeel bij de opgaven zelf aangegeven.

Studeeraanwijzingen
De studielast van deze eindtoets is, inclusief het bekijken van de terugkoppeling, circa 3 uur.

Opgaven

OPGAVE 1: (20 PUNTEN)

Run-length encoding is een eenvoudig algoritme om informatie met veel herhaling te comprimeren. De codering van het algoritme werkt als volgt: gegeven een lijst van symbolen wordt een lijst van tupels opgeleverd, waarvan de eerste component telkens aangeeft hoe vaak een symbool (de tweede component) moet worden herhaald. Het coderen werkt niet alleen voor strings, maar ook voor lijsten met andere soorten elementen.

```
Hugs> codeer "aaabbaaac"
[(3,'a'),(2,'b'),(3,'a'),(1,'c')]

Hugs> codeer [0,0,0,0,1,1,0,0]
[(4,0),(2,1),(2,0)]
```

5 punten a Declareer een zo algemeen mogelijk type voor de functie *codeer*.

5 punten b Geef een definitie van de functie *codeer*.

5 punten c De functie *decodeer* neemt een gecodeerde waarde, en reconstrueert de oorspronkelijke waarde. Definieer *decodeer* als een recursieve functie, en declareer een zo algemeen mogelijk type.
Voorbeeld: decodeer $[(3,\texttt{'a'}),(2,\texttt{'b'}),(3,\texttt{'a'}),(1,\texttt{'c'})]$ geeft `"aaabbaaac"`.

5 punten d De functie *decodeer* is met een *foldr* te schrijven. Maak de volgende definitie af door de gaten in te vullen.

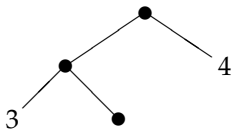
```
decodeer = foldr op leeg
where
  op ... = ...
  leeg   = ...
```

OPGAVE 2: (30 PUNTEN)

Met het volgende datatype kunnen binaire bomen worden gerepresenteerd die ook leeg kunnen zijn:

data *Boom a* = *Leeg* | *Blad a* | *Knoop* (*Boom a*) (*Boom a*)

Hieronder staat een voorbeeld van een binaire boom afgebeeld, waarin naast twee bladeren met de getallen 3 en 4 van het type *Int* ook een lege subboom is te vinden.

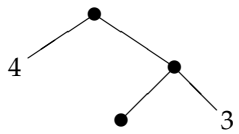


5 punten a Geef een definitie voor de afgebeelde binaire boom, en noem deze waarde *boom*. Declareer ook het type van *boom*.

5 punten b Definieer de functie *naarLijst* :: *Boom a* → [*a*], die de opgeslagen waarden in een boom verzamelt in een lijst. Bijvoorbeeld, *naarLijst boom* levert [3, 4] op.

5 punten c De gekozen representatie is weinig efficiënt omdat subbomen leeg kunnen zijn. Schrijf een functie *compact* :: *Boom a* → *Boom a* die het aantal lege subbomen minimaliseert, zonder dat er waarden verloren gaan.
Voorbeeld: De expressie *compact boom* zal evalueren naar *Knoop* (*Blad* 3) (*Blad* 4), dus zonder de constructor *Leeg*.

5 punten d Definieer de functie *spiegel* die een boom spiegelt door alle linker- en rechtertakken in de boom om te wisselen. Declareer ook het type van de functie. Het spiegelen van de voorbeeldboom moet het volgende resultaat opleveren:



10 punten e Het twee keer spiegelen van een boom moet resulteren in de originele boom. Meer precies is om te beweren dat de eigenschap *spiegel* ∘ *spiegel* = *id* geldt. Bewijs dat *spiegel* (*spiegel x*) gelijk is aan *x*, voor elke boom *x*. Gebruik het principe van inductie.

OPGAVE 3: (30 PUNTEN)

De Top 2000 is een lijst die jaarlijks wordt samengesteld met de meest populaire platen. De vijf hoogst genoteerde platen in de lijst van 2011 zijn:

```
type Plaat = (String, String, Int)

top5 :: [Plaat]
top5 = [("Queen","Bohemian rhapsody",1975)
       ,("Eagles","Hotel California",1977)
       ,("Deep Purple","Child in time",1972)
       ,("Boudewijn de Groot","Avond",1997)
       ,("Led Zeppelin","Stairway to heaven",1971)
       ]
```

Voor iedere plaat zijn de artiest, de titel en het jaartal waarin de plaat is uitgebracht gegeven. In deze opgave wordt gekeken hoe de informatie van de lijst op een overzichtelijke manier kan worden getoond.
Een blok is een lijst van strings die onder elkaar moeten worden getoond. We gaan ervan uit dat alle strings in een blok dezelfde lengte hebben. We geven verder nog een hulpfunctie om een blok netjes te tonen.

```
type Blok = [String]   -- alle strings zijn even lang
```

$toonBlok :: Blok \to IO\ ()$
$toonBlok = putStrLn \circ unlines$

De functie $unlines :: [String] \to String$ komt uit de Prelude en plaatst het 'newline'-karakter tussen strings.

3 punten

a Schrijf een functie $hoogte :: Blok \to Int$ die de hoogte van een blok, oftewel het aantal regels, oplevert.
Voorbeeld:

```
Hugs> hoogte ["Queen      ","Eagles     ","Deep Purple"]
3
```

5 punten

b Schrijf een functie $breedte :: [String] \to Int$ die de lengte van de langste string in de lijst oplevert.
Voorbeeld:

```
Hugs> breedte ["Queen","Eagles","Deep Purple"]
11
```

5 punten

c Schrijf een functie $maakBreed :: Int \to String \to String$ die een string van een gegeven lengte maakt door achteraan spaties toe te voegen.
Voorbeeld:

```
Hugs> maakBreed 11 "Queen"
"Queen      "
```

5 punten

d Schrijf een functie $maakBlok :: [String] \to Blok$ die van een lijst van strings een blok maakt. De strings in de lijst kunnen verschillende lengtes hebben. Alle strings in het blok moeten even lang zijn.
Voorbeeld:

```
Hugs> maakBlok ["Queen","Eagles","Deep Purple"]
["Queen      ","Eagles     ","Deep Purple"]
```

5 punten

e Definieer de operator <> die twee blokken naast elkaar zet en geef het type van deze operator. Ga ervan uit dat de blokken even hoog zijn. Tussen de blokken moet een verticale streep komen te staan: doe dit door in iedere rij de string `" | "` in te voegen.
Voorbeeld:

```
Hugs> maakBlok ["zeven", "tien"] <> maakBlok ["7", "10"]
["zeven | 7 ","tien  | 10"]
```

7 punten

f Definieer de functie $toonLijst :: [Plaat] \to IO\ ()$ die een genummerde lijst toont, precies zoals hieronder is weergegeven. De eerste kolom bevat posities.
Voorbeeld:

```
Hugs> toonLijst top5
1 | Queen              | Bohemian rhapsody  | 1975
2 | Eagles             | Hotel California   | 1977
3 | Deep Purple        | Child in time      | 1972
4 | Boudewijn de Groot | Avond              | 1997
5 | Led Zeppelin       | Stairway to heaven | 1971
```

Aanwijzing: De standaardfunctie $unzip3 :: [(a,b,c)] \to ([a],[b],[c])$ kan worden gebruikt om de 3-tupels uit elkaar te halen. Gebruik ook de gegeven functie *toonBlok*.

OPGAVE 4: (20 PUNTEN)

Een monoïde is een wiskundige structuur die bestaat uit een binaire operatie ($\oplus$), en een neutraal element voor die operatie (e). Voor een monoïde gelden de volgende eigenschappen:

$$\begin{aligned} e \oplus x &= x \\ x \oplus e &= x \\ x \oplus (y \oplus z) &= (x \oplus y) \oplus z \end{aligned}$$

voor willekeurige waarden x, y en z.

In Haskell wordt een monoïde op de volgende manier gemodelleerd:

```
class Monoid a where
  combi :: a → a → a   -- de binaire operatie (⊕)
  leeg  :: a           -- het neutrale element (e)
```

Het type *Int* kan tot een instantie worden gemaakt door optellen te gebruiken als de operatie, met 0 als neutraal element:

```
instance Monoid Int where
  combi = (+)
  leeg  = 0
```

6 punten a Lijsten kunnen worden gecombineerd door ze achter elkaar te plakken. Maak lijsten tot een instantie van de typeklasse *Monoid*.

6 punten b Schrijf een functie *combineren* die een lijst van waarden combineert tot een enkele waarde. Het type van de waarden moet behoren tot de *Monoid* typeklasse. Declareer ook het meest algemene type van de functie *combineren*.

4 punten c Wat is het resultaat van de expressie *combineren* $[[4,5,6],[1],[2,3]]$?

4 punten d Met welke functies uit de Prelude komt *combineren* overeen voor *Int*-waarden, en met welke functie voor lijsten?

Uitwerking van de eindtoets

ANTWOORD OPGAVE 1

a Het meest algemene type is:

$$codeer :: Eq\ a \Rightarrow [a] \rightarrow [(Int, a)]$$

Het type van *codeer* is polymorf vanwege de typevariabele a. Er is wel een beperking op het type a, namelijk dat het tot de typeklasse *Eq* moet behoren (vanwege het testen op gelijkheid).

b De definitie van de *codeer*-functie is:

$$\begin{aligned}
&codeer\ [\,] &&= [\,] \\
&codeer\ (x : xs) &&= (aantal, x) : codeer\ rest \\
&\textbf{where} \\
&\quad gelijk &&= takeWhile\ (==x)\ xs \\
&\quad rest &&= dropWhile\ (==x)\ xs \\
&\quad aantal &&= 1 + length\ gelijk
\end{aligned}$$

c Decoderen met recursie kan op de volgende manier worden bereikt:

$$\begin{aligned}
&decodeer :: Eq\ a \Rightarrow [(Int, a)] \rightarrow [a] \\
&decodeer\ [\,] &&= [\,] \\
&decodeer\ ((n, c) : xs) &&= replicate\ n\ c +\!\!+ decodeer\ xs
\end{aligned}$$

De functie $replicate :: Int \rightarrow a \rightarrow [a]$ uit de Prelude herhaalt een element een gegeven aantal keer.

d De definitie met een *foldr* is af te leiden uit de definitie bij onderdeel c.

$$\begin{aligned}
&decodeer :: Eq\ a \Rightarrow [(Int, a)] \rightarrow [a] \\
&decodeer = foldr\ op\ leeg \\
&\textbf{where} \\
&\quad op\ (n, c)\ s = replicate\ n\ c +\!\!+ s \\
&\quad leeg\ \ \ \ \ \ = [\,]
\end{aligned}$$

ANTWOORD OPGAVE 2

a De waarde *boom* is:

$$\begin{aligned}
&boom :: Boom\ Int \\
&boom = Knoop\ (Knoop\ (Blad\ 3)\ Leeg)\ (Blad\ 4)
\end{aligned}$$

b Voor elke constructor van het datatype *Boom* definiëren we de vertaling naar een lijst:

$$\begin{aligned}
&naarLijst :: Boom\ a \rightarrow [a] \\
&naarLijst\ Leeg &&= [\,] \\
&naarLijst\ (Blad\ x) &&= [x] \\
&naarLijst\ (Knoop\ l\ r) &&= naarLijst\ l +\!\!+ naarLijst\ r
\end{aligned}$$

c Het is handig om eerst een hulpfunctie te schrijven die twee compacte bomen combineert tot een compacte boom:

$$\begin{aligned}
&combineer :: Boom\ a \rightarrow Boom\ a \rightarrow Boom\ a \\
&combineer\ Leeg\ r &&= r \\
&combineer\ l\ \ Leeg &&= l \\
&combineer\ l\ \ r &&= Knoop\ l\ r
\end{aligned}$$

Het geval $combineer\ Leeg\ Leeg = Leeg$ hoeft niet te worden gegeven: dit wordt al afgedekt door het eerste geval. Met deze hulpfunctie kan *compact* worden gedefinieerd met recursie.

$$\begin{aligned}
&compact :: Boom\ a \rightarrow Boom\ a \\
&compact\ (Knoop\ l\ r) &&= combineer\ (compact\ l)\ (compact\ r) \\
&compact\ x &&= x
\end{aligned}$$

De gevallen *compact Leeg* en *compact* (*Blad x*) zijn in de uitwerking samengenomen, aangezien in beide gevallen dezelfde boom wordt opgeleverd.
Aanwijzing: Om te controleren of uw eigen uitwerking correct is, kunt u uitproberen of *compact* (*Knoop* (*Knoop Leeg Leeg*) *Leeg*) werkelijk gelijk is aan *Leeg*.

d Het spiegelen van een boom gaat als volgt:

```
spiegel :: Boom a → Boom a
spiegel (Knoop l r) = Knoop (spiegel l) (spiegel r)
spiegel (Blad x)    = Blad x
spiegel Leeg        = Leeg
```

De laatste twee gevallen mogen worden samengenomen.

e Om aan te tonen dat *spiegel* (*spiegel x*) = *x* stellen we een inductief bewijs op met gevalsonderscheid naar *x*. Er zijn twee basisgevallen.
Het basisgeval *Leeg*:

```
    spiegel (spiegel Leeg)
=       { toepassen binnenste spiegel }
    spiegel Leeg
=       { toepassen spiegel }
    Leeg
```

Het basisgeval (*Blad x*):

```
    spiegel (spiegel (Blad x))
=       { toepassen binnenste spiegel }
    spiegel (Blad x)
=       { toepassen spiegel }
    Blad x
```

Het inductieve geval (*Knoop l r*), waarvoor de inductiehypothese *spiegel* (*spiegel l*) = *l* en de inductiehypothese *spiegel* (*spiegel r*) = *r* mag worden gebruikt, is:

```
    spiegel (spiegel (Knoop l r))
=       { toepassen binnenste spiegel }
    spiegel (Knoop (spiegel r) (spiegel l))
=       { toepassen spiegel }
    Knoop (spiegel (spiegel l)) (spiegel (spiegel r))
=       { inductiehypothese voor l }
    Knoop l (spiegel (spiegel r))
=       { inductiehypothese voor r }
    Knoop l r
```

ANTWOORD OPGAVE 3

a Voor het bepalen van de hoogte kan de standaardfunctie *length* worden gebruikt.

```
hoogte :: Blok → Int
hoogte = length
```

b De breedte bepalen we door eerst de lengte te berekenen van iedere string, en hier vervolgens het maximum van te nemen.

```
breedte :: [String] → Int
breedte = maximum ∘ map length
```

De functie *breedte* is niet gedefinieerd voor de lege lijst.

c De onderstaande oplossing gebruikt de oneindige string met spaties, gegenereerd door de expressie *repeat* `' '`.

```
maakBreed :: Int → String → String
maakBreed n s = take n (s ++ repeat ' ')
```

Een alternatieve oplossing is om *replicate* te gebruiken:

```
maakBreed :: Int → String → String
maakBreed n s = s ++ replicate (n − length s) ' '
```

d De definitie voor *maakBlok* is:

```
maakBlok :: [String] → Blok
maakBlok xs = map (maakBreed n) xs
  where n = breedte xs
```

e Het naast elkaar plaatsen van twee blokken kan met een lijst-comprehensie worden opgeschreven:

```
(<>) :: Blok → Blok → Blok
xs <> ys = [x ++ "  |  " ++ y | (x, y) ← zip xs ys]
```

f De functie kan stapsgewijs worden gedefinieerd, waarbij lokale definities voor tussenresultaten worden gebruikt.

```
toonLijst :: [Plaat] → IO ()
toonLijst xs = toonBlok totaal
where
  (artiest, titel, jaar) = unzip3 xs

  posBlok  = maakBlok (map show [1 .. length xs])
  jaarBlok = maakBlok (map show jaar)
  totaal   = posBlok <> maakBlok artiest <> maakBlok titel <> jaarBlok
```

De tussenstappen zijn:
- Met *unzip3* wordt de lijst met platen gesplitst in een lijst met artiesten, een lijst met titels en een lijst met jaartallen.
- Voor de eerste kolom wordt *posBlok* opgebouwd: van de getallen worden strings gemaakt met de functie *show*.
- Voor de laatste kolom wordt *jaarBlok* gedefinieerd.
- De waarde *totaal* combineert blokken met de <>-operator.
- Het *totaal* wordt getoond met behulp van de functie *toonBlok*.

ANTWOORD OPGAVE 4

a De instance-declaratie voor lijsten is:

```
instance Monoid [a] where
  combi = (++)
  leeg  = []
```

b De functie *combineren* kan als een recursieve functie worden opgeschreven:

```
combineren :: Monoid a ⇒ [a] → a
combineren []       = leeg
combineren (x : xs) = combi x (combineren xs)
```

De functie *foldr* gebruiken geeft een kortere definitie:

```
combineren :: Monoid a ⇒ [a] → a
combineren = foldr combi leeg
```

c De expressie *combineren* $[[4,5,6],[1],[2,3]]$ zal de lijst $[4,5,6,1,2,3]$ als resultaat hebben.

d De functie *combineren* komt overeen met de functie:
- *sum* :: *Num a* ⇒ $[a] \to a$ voor *Int* waarden
- *concat* :: $[[a]] \to [a]$ voor lijsten