FP 2008-2008, Tussentoets
2008, Maart 4 , 13.00-16.00

Het tentamen bestaat uit 6 multiple-choice vragen (1 punt elk) en 2 open vragen (2 punten elk). Lever het aparte antwoordblad in. **Vergeet niet je naam ook in te vullen.** maak je programma's eerst in klad, en schrijf ze dan goed leesbaar op het antwoordvel. Let op de haakjes in patronen!

1. Wat is het type van *map concat*

   (a) $[[[a]]] \to [a]$
   (b) $[[[a]]] \to [[a]]$
   (c) $[[a]] \to [a]$
   (d) geen van (a) t/m (c)

   Het goede antwoord is (b). De functie *concat* heeft type $[[a]] \to [a]$, en als we die op een lijst loslaten m.b.v. *map*, dan komt er dus voor het argument en het resultaat een extra []-paar bij.

2. Welke van de volgende uitspraken is waar m.b.t. de expressie *takeWhile True* (*repeat True*):

   (a) resultaat is gelijk aan *repeat True*
   (b) resultaat is gelijk aan []
   (c) evaluatie gaat in een loop, en er verschijnt geen resultaat
   (d) is niet goed getypeerd

   De functie *takeWhile* heeft type $(a \to Bool) \to [a] \to [a]$. Het eerste alternatief ziet er kennelijk aantrekkelijk uit, maar het goede antwoord is dus (d), want *True* is geen functie!.

3. Wat is het correcte resultaat van de expressie *foldr* $((-) \,.\, (+2))\ 2\ [2,2,2]$?

   (a) 0
   (b) 2
   (c) -2
   (d) -6

   We bekijken de rare operator $(-) \,.\, (+2)$ en laten die los op het volgende element van de lijst (een 2 dus) en het tot dusverre berekende resultaat, zeg r: $((-) \,.\, (+2)\ 2\ r$. Dit kunnen we herschrijven tot $(-)\ (2+2)\ r$, en dit weer tot $(2+2) - r$. Als r dus gelijk is aan 2 dan is het resultaat na het verwerken van de volgende 2 dus 2. Nu is het eenheidselement uiteindelijk 2, dus met inductie zien we dat het resultaat dus uiteindelijk een 2 is: (b).

4. Wat is de correcte uitspraak over het volgende programma?

```
main = do c ← return 'c'
          c ← return (do c ← getChar
                         return (putChar c))
          c
          c
          return ()
```

   (a) Leest twee karakters in (met echo) en doet verder niets.
   (b) Geeft als uitvoer `'c'cc`
   (c) Leest een karakter in en drukt dat eindeloos af
   (d) Leest twee karakters in (met echo) en drukt die weer af.

Dit moet (a) zijn. Deze opgave is geinspireerd door de vraag op het college. Het eerste statement geeft c de waarde `'c'`, maar direct daarna wordt c opnieuw gebonden. We kunnen dit statement dus net zo goed weglaten. Daarna krijgt c via de *return* de waarde **do** $\{c \leftarrow getChar; return\ (putChar\ c\}$. Dit levert als effect op dat er een karakter wordt ingelezen, en dat het statement dat dit karakter afdrukt wordt opgeleverd. Daarna wordt c twee keer uitgevoerd, met als gevolg dat er twee keer een karakter ingelezen wordt, en twee keer het *putChar c* statement wordt opgeleverd. Dit wordt echter nergens aan gebonden (eigenlijk staat er twee keer $\_ \leftarrow c$), en verdwijnt dus. Had er geen *return* om de *putChar c* gestaan dan zou het karakter ook telkens weer afgedrukt zijn.

5. Iemand vertaalt het volgende programma, zodat het geen lijstcomprehensies meer bevat. Wat is het correcte resultaat?

```
segs xs = [] : [t | i ← inits xs, t ← tails i, not (null t)]
```

(a)
```
[] : concat (map f . map g) (inits xs)
    where f i = tails i
          g t = if null t then [] else [t]
```

(b)
```
[] : filter (not . null) . concat (map f . map g) (inits xs)
    where f i = tails i
          g t = [t]
```

(c)
```
[] : concat (map f (inits xs))
    where f i = concat (map g (tails i))
                where g t = if not (null t) then [] else [t]
```

(d)
```
[] : concat (map f (inits xs))
    where f i = concat (map g (tails i))
                where g t = if not (null t) then [t] else []
```

(d)

6. Welk van de volgende types is equivalent aan een binaire zoekboom? Voor de volledigheid:

```
data GTree f a = GLeaf
               | GNode a (f (GTree f a))
data Paar a    = Paar a a
```

(a) **type** $ZB\ a = GTree\ Paar\ (a, a)$

(b) **type** $ZB\ a = GTree\ []\ a$

(c) **type** $ZB\ a = GTree\ Paar\ a$

(d) **type** $ZB\ a = GTree\ (a, a)\ a$

(c)

7. Given the type

```
data Tree = Node Tree Int Tree
          | Leaf
```

Write a function $smallPathsOnly :: Int \rightarrow Tree \rightarrow [[Int]]$ that efficiently produces all paths from the root to the leafs of which the sum of the elements on that path is less than the given *Int* parameter.

```
data Tree = Leaf | Node Tree Int Tree
smallPathsOnly :: Int → Tree → [[Int]]
smallPathsOnly k ns = spo restval id ns []
where
  spo _ p Leaf = (p []:)
  spo restval p (Node l n r)
```

```
  | restval < 0 = id
  | otherwise  = spo' l . spo' r
where
  spo' = spo (restval - n) (p . (n:))
```

Iets minder efficiente oplossingen zijn ook goed. Hierboven is het uiterste gedaan om de resultaatlijst zo efficient mogelijk op te bouwen.

8. Schrijf een functie *lines* :: *Int* → [*String*] → [*String*] die een lijst met woorden opsplitst in een aantal regels, waarbij de regels niet langer mogen zijn dan de meegegeven *Int* waarde. In de resulterende regels zijn de woorden gescheiden door een `' '`. Woorden mogen niet afgebroken worden, en de lengte van het langste woord is hoogstens de meegegeven *Int* waarde.

```
lines :: Int → [String] → [String]
lines _ [] = []
lines k (w : ws) = line (length w) (w ++) ws
where
  line _ p [] = [p ""]
  line acc p (w : ws)
       |   acc' ⩽ k  = line acc' (p . (' ':) . (w ++)) ws
       |   otherwise = p "" : line (length w) (w ++) ws
    where
          acc' = acc + length w + 1
```

De eerste parameter van *line* is de lengte van de tot dusverre opgebouwde regel, en de tweede parameter de tot dusverre opgebouwde regel zelf (weer in een effiicente representatie). Als we aan het eind van de lijst met woorden zijn gekomen, bouwen de efficiente representatie van de laatste regel om tot een echte *String* via *p* `""`, en leveren deze laatste regel in een lijst van één lang op. Anders kijken we wat we met het volgende woord moeten doen; als het samen met een voorafgaande spatie nog op de huidige regel past voegen we het toe, en zo niet dan sluiten we de huidige regel af, en beginnen aan een nieuwe die we initialiseren met dit woord en als eerste parameter weer de daarmee al geconsumeerde ruimte.