Tentamen Functioneel Programmeren

31 januari 2006, 14.00 uur – 17.00 uur, Examenhal

Schrijf met blauwe of zwarte pen; *niet* met potlood en *niet* met rode pen. Voorzie alle bladen van je naam. Nummer de bladen en vermeld op het eerste blad het totale aantal.
Houd je programma's kort en helder, mede door handig gebruik te maken van standaardfuncties uit het cursusboek (in het bijzonder uit het gedeelte over lijsten). Als je ergens een programma opschrijft dat toch wat complexer van structuur is en daardoor minder helder op zich is, dan moet er een extra toelichting bij.

Opgave 1.

(i) In de algebra wordt het *inprodukt* van twee even lange vectoren $(x_1, \ldots, x_n)$ en $(y_1, \ldots, y_n)$ van reële getallen gedefiniëerd als de volgende som van produkten: $x_1y_1 + \cdots + x_ny_n$. Een elegante Haskell-implementatie van deze inproduktfunctie wordt gegeven door:

```
inprodukt ::  [Float] -> [Float] -> Float
inprodukt xs ys = sum (zipWith (*) xs ys)
```

(ia) Geef de definitie, inclusief typering, van de hier gebruikte standaardfunctie `zipWith`.

(iib) Bepaal het type van de expressie `(map zipWith)` .

(iia) Geef de definitie, inclusief typering, van de Haskell-functie `foldr` .

(iib) Beschouw de functie `sum` als boven. Geef een Haskell-definitie van deze functie met behulp van `foldr`.

(iic) Beargumenteer dat de expressie `(foldr sum)` *niet* typeerbaar is.

Opgave 2. beschouw de volgende gelijkheid:

```
concat (xss++yss) = concat xss ++ concat yss
```

(i) Maak op basis van de specificatie van de standaardfunctie `concat` aannemelijk dat deze gelijkheid geldt voor alle eindige lijsten `xss` en `yss` van eindige lijsten (van hetzelfde type).

(ii) Geef een exact bewijs van de geldigheid van de gelijkheid voor alle eindige lijsten `xss` en `yss` van eindige lijsten van hetzelfde type. Doe dit met behulp van inductie, en wel naar `xss`. Begin met het opschrijven van de recursieve Haskell-definities van de functies `(++)` en `concat`. In de verdere uitwerking van je bewijs mag je vrijelijk gebruik maken van de associativiteit van `(++)` in de zin dat

```
(us++vs)++ws = us++(vs++ws)
```

voor alle eindige lijsten `us`, `vs` en `ws` van hetzelfde type.

(iii) Bewijs de associativitiet van functiecompositie `(.)`; d.w.z.: bewijs dat

```
(f.g).h = f.(g.h)
```

voor alle functies `f`, `g`, `h` met passend aansluitende types. (Preciseer ook wat hier "met passend aansluitende types" inhoudt.)

Opgave 3.

(i) Geef een heldere Haskell-definitie van een functie

```
mergeSort :: Ord a => [a] -> [a]
```

die eindige lijsten sorteert volgens het volgende idee. (*Afspraak tussendoor.* In dit onderdeel zijn "lijsten" steeds: lijsten met verschillende elementen.) Als een lijst `xs` hooguit lengte 1 heeft, dan wordt de lijst `xs` zelf opgelevert. Als `xs` een eindige lijst is met minstens lengte 2, dan wordt `xs` opgesplitst in twee lijsten die even lang of bijna even lang zijn (in de zin dat hun lengten hooguit 1 verschillen) en dan worden die lijsten gesorteerd, waarna — met een passend te definiëren hulpfunctie `merge` — de resultaatlijsten gecombineerd worden tot een strikt stijgende lijst met als verzamelig van elementen: de vereniging van de verzamelingen van de elementen van die resultaatlijsten.

(ii) Geef door middel van één enkele vergelijking een Haskell-definitie van de oneindige lijst `fibList` van alle Fibonacci-getallen 1, 1, 2, 3, 5, 8, ...

(iii) Geef ook door middel van één enkele vergelijking een Haskell-definitie van de oneindige lijst `pow2 :: [Int]` van alle machten van 2 (in strikt stijgende volgorde). (*Aanwijzing.* Merk op dat de verzameling P van deze machten als volgt inductief te karakteriseren is. P bevat 1 en voor elke $x \in P$ geldt dat ook $2x \in P$.)

(iv) Geef een Haskell-definitie van de functie `elemNum :: Int -> [Int] -> Int` die voor een gegeven getal en voor een gegeven lijst van getallen oplevert: het aantal keren dat dat getal in die lijst voorkomt. Geef die defintie door middel van één enkele vergelijking voor `elemNum x xs`. Doe dit met behulp van lijstcomprehensie en/of standaardfuncties.

Opgave 4. Onder een *taal* verstaan we in deze opgave: een verzameling van eindige strings. In het onderhavige verband zijn *sparsers* en *parsers* voor talen bepaalde soorten Haskell-functies, zoals die aan de orde geweest zijn in de FP-zelfstudiestof en in de FP-werkcollegestof.

(i) Zij L een taal.

(ia) Wat is het Haskell-type van een sparser voor L?

(ib) Wat is het Haskell-type van een parser voor L, wanneer gewerkt wordt met een type T als type van representaties van de elementen van L. (*Aanwijzing.* Het gevraagde type is `Parse Char` T. Schrijf dit expliciet uit.)

(ii) Beschouw nu in het bijzonder: de taal L bestaande uit alle eindige strings van cijfers.

(iia) Construeer in Haskell een sparser voor L.

(iib) Construeer in Haskell een parser voor L, daarbij als representatie nemend van elke willekeurige string `xs` uit L: het paar (n, l) met n de waarde van `xs` (gezien als decimale representatie van een getal, eventueel na weglating van overbodige 0-en aan het begin, waarbij het zo is dat de lege string per definitie waarde 0 heeft) en met l de lengte van `xs` (gewoon gezien als string).

(iic) Construeer direkt — of zo direkt mogelijk — in Haskell: een efficiënte functie

```
numvalue :: String -> Int
```

met de volgende eigenschap: voor elke willekeurige eindige string xs is `(numvalue xs)` gelijk aan de waarde van het langste beginstuk van `xs` dat tot L behoort.