Tentamen Functioneel Programmeren

1 november 2004, 14.00–17.00 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 bij een onderdeel van een opgave onverhoopt niet een gevraagde Haskell-definitie kunt geven van een functie met een gegeven specificatie, dan mag je verderop in de betreffende opgave toch gewoon naar die functie verwijzen (en daarbij dus aannemen dat aan de specificatie voldaan is).

Opgave 1.

(i) De volgende elegante Haskell-implementatie van de lijst van de Fibonacci-getallen is bekend uit de theorie:

```
fibs ::  [Int]
fibs = 1 :  1 :  zipWith (+) fibs (tail fibs)
```

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

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

(ii) De standaardfunctie `flip` wordt gedefiniëerd door:

```
flip f x y = f y x
```

(a) Geef het type van `flip` .

(b) Bepaal de waarden van de volgende expressie:
`sum(filter (flip(<)4) [2,4,1,5,3,6])`

Opgave 2. Bewijs inductief dat voor alle eindige lijsten `xs`, `ys` (van hetzelfde type):

```
foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs
```

(onder passende aannamen over de types van `f` en `z`). *Hint:* doe inductie over `xs` (*niet* over `ys`).

Opgave 3. Specificeer de functie `merge` van type `Ord a => [a] -> [a] -> [a]` als volgt.
Als `xs` en `ys` strikt stijgende lijsten (van hetzelfde type) zijn, dan is `merge xs ys` de strikt stijgende lijst met als elementen: de elementen van `xs` en `ys`.
(Voorbeeld: `merge[2,5,8][1,3,7,8] = [1,2,3,5,7,8]`.)

(i) Geef een recursieve Haskell-definitie van `merge`.

(ii) Beschouw de volgende recursieve sorteermethode voor eindige lijsten van *verschillende* elementen van een Ord-type. Als zo'n lijst hooguit lengte 1 heeft, dan wordt de lijst zelf opgeleverd. Als de lijst minstens lengte 2 heeft, dan wordt hij 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 de resultaten gecombineerd worden met behulp van `merge`. Geef een Haskell-implementatie van deze methode, en wel door een passende definitie te geven van de betreffende functie `mergeSort :: Ord a => [a] -> [a]`.

(iii) (Andere sorteerfunctie, `quickSort`, zonder `merge`.) Geef ook een Haskell-implementatie van de volgende alternatieve recursieve sorteermethode voor de in (ii) genoemde lijsten. Het basisgeval is het triviale geval van de lege lijst. In het geval van een lijst van de vorm `x:xs` worden twee lijsten uit de staart `xs` gemaakt: de ene bestaat uit de respectievelijke elementen `< x` en de andere bestaat uit de respectievelijke elementen `> x`. Die twee lijsten worden gesorteerd en de resultaten worden, met `x` ertussen, aan elkaar geplakt.

(iv) (Nu, in aansluiting op (i) en (ii), een andere toepassing van `merge`.) Definiëer de verzameling H van integers inductief door: H bevat 1 en voor elke integer x geldt: als x voorkomt in H, dan komen $2x$ en $3x$ ook voor in H. Geef door middel van één enkele vergelijking een Haskell-definitie van de strikt stijgende lijst `hs` van type `[Int]` die precies uit alle elementen van H bestaat. Maak hierbij handig gebruik van (o.a.) `merge`.

(*Opmerking en aanwijzing.* (1) H is eenvoudigweg te omschrijven als de verzameling van alle getallen van de vorm $2^i 3^j$, met $i, j \in \mathbb{N}$, maar de genoemde inductieve karakterisering is een geschikter uitgangspunt voor de bedoelde Haskell-implementatie. (2) De bedoelde definitie van `hs` is van de vorm `hs = .....`, met in het rechterlid een expressie die op handige wijze representeert: de strikt stijgende lijst bestaande uit 1 en, in passende onderlinge volgorde, de tweevouden en de drievouden van de elementen van `hs`.)

Opgave 4.

(i) Geef een Haskell-definitie van een functie

```
voegtoe ::  a -> [a] -> [[a]]
```

die voor een object `x` en een eindige lijst `xs` van objecten van hetzelfde type als `x` oplevert: een lijst van alle lijsten die uit `xs` te verkrijgen zijn door `x` ergens toe te voegen aan `xs`.
Voorbeeld: `voegtoe 3 [2,4]` is de lijst met als verschillende elementen (al dan niet in deze volgorde): `[3,2,4]`, `[2,3,4]` en `[2,4,3]`.

Zet de definitie van `voegtoe` recursief op door gelijkheden te geven voor `voegtoe x []` en voor `voegtoe x (y:ys)` , maar vermijd verdere expliciete recursie (en wel door handig gebruik van standaardfunctie(s)).

(ii) Geef met behulp van `voegtoe` een handige Haskell-definitie van een functie

```
voegtoeL ::  a -> [[a]] -> [[a]]
```

die voor een object `x` en een lijst `xss` van eindige lijsten van objecten van hetzelfde type als `x` oplevert: een lijst van alle lijsten die te verkrijgen zijn door een lijst uit `xss` te nemen en `x` ergens daaraan toe te voegen.
Voorbeeld: `voegtoeL 3 [[2,4],[5]]` is een lijst met als elementen (al dan niet in deze volgorde): `[3,2,4]`, `[2,3,4]`, `[2,4,3]`, `[3,5]`, `[5,3]`.

Opgave 5. Definiëer de boomtypes `Boom` en `LBoom` door:

```
data Boom = Blad | Knoop Boom Boom
data LBoom = LBlad Int | LKnoop Int LBoom LBoom
```

`Boom` is het type van "kale binaire bomen" en `LBoom` is het type van "Int-gelabelde binaire bomen".

(i) Geef een recursieve Haskell-implementatie van de functie `maxLabel ::  LBoom -> Int` die het maximale label bepaalt binnen een (eindige) Int-gelabelde binaire boom.

(ii) Geef een recursieve Haskell-implementatie van de functie `labelD ::  Int -> Boom -> LBoom` met de volgende eigenschap. Als `n` van type `Int` is, dan voorziet de functie `(labelD n) :: Boom -> LBoom` de knopen van een willekeurige kale binaire boom `b` als volgt van labels. De wortel van `b` krijgt als label de waarde van `n`, en voor elke willekeurige knoop van `b` geldt: als deze label x krijgt, dan krijgen de eventuele zonen ervan labels $2x$, resp. $2x + 1$.

Opgave 1

(i) (a)
```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
zipWith _ _ _ = []
```

(b)
```
(map zipWith) :: [a -> b -> c] -> [[a] -> [b] -> [c]]
```

(ii) (a)
```
flip :: (a -> b -> c) -> b -> a -> c
```

(b) 6

Opgave 2

Basis:
```
foldr f z ([] ++ ys)
=> [] ++ ys = ys
foldr f z ys
=> foldr _ init [] = init
foldr f (foldr f z ys) []
=> xs = []
foldr f (foldr f z ys) xs
```

Inductiehypothese:
```
foldr f z (xs ++ ys) = foldr f (foldr f z ys) xs
```

Bewijs:
```
foldr f (foldr f z ys) x:xs
=> definitie foldr
f x (foldr f (foldr f z ys) xs)
=> inductiehypothese
f x (foldr f z (xs ++ ys))
=> definitie foldr
foldr f z ((x:xs) ++ ys)
```

Opgave 3

(i)
```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys)
  | x > y = y:merge (x:xs) ys
  | y > x = x:merge xs (y:ys)
  | otherwise = x:merge x sys
```

(ii)
```
mergeSort :: Ord a => [a] => [a]
mergeSort [] = []
mergeSort (x:[]) = [x]
mergeSort xs = merge (mergeSort left) (mergeSort right)
  where left = take halflength xs
        right = drop halflength xs
        halflength = (length xs) `div` 2
```

(iii)
```
quickSort :: Ord a => [a] => [a]
quickSort [] = []
quickSort (x:[]) = [x]
quickSort (x:xs) = left ++ [x] ++ right
  where left = quickSort (filter (flip (<) x) xs)
        right = quickSort (filter (< x) xs)
```

(iv)
```
hs :: [Int]
hs = merge (map (*2) hs) (map (*3) hs)
```

Opgave 4

(i)
```
voegtoe :: a -> [a] -> [[a]]
voegtoe x [] = [[x]]
voegtoe x (y:ys) = [(x:y:ys)] ++ map (y:) (voegtoe x ys)
```

(ii)
```
voegtoeL :: a -> [[a]] -> [[a]]
voegtoeL = concat [voegtoe x xs | xs <- xxs]
```

Opgave 5

(i)
```
maxLabel :: LBoom -> Int
maxLabel (LBlad x) = x
maxLabel (LKnoop x li re) = max x (max (maxLabel x li) (maxLabel x ri))
```

(ii)
```
labelD :: Int -> Boom -> LBoom
labelD n (Blad) = (LBlad n)
labelD n (Knoop li re) = (LKnoop n (labelD (2*n) li) (labelD (2*n+1) re)
```