Uitwerking Tentamen Functioneel Programmeren, 21 november 2000

Deze uitwerking is als lhs-file op te halen: `www.cs.rug.nl/~wim.onderwijs.fp/fpuitw0011.lhs`.

Tijdsduur 3 uur.

Houd de programma's kort door zoveel mogelijk standaardfuncties te gebruiken (uit de prelude van Hugs en het boek van Bird). Je hoeft geen definities te geven van: id, head, tail, take, drop, map, filter, and, or, zip, curry, uncurry. Geef van andere standaardfuncties wel steeds definities (gelijk of equivalent met de standaarddefinities).

Geef van elke hulpfunctie die je gebruikt een informele specificatie.

Het begrip stijgend betekent steeds strikt stijgend.

Opgave 1 (2 punt). Maak een functie

```
ontbind :: [Int] -> [[Int]]
```

zodanig dat `ontbind xs` de zo kort mogelijke lijst `yss` is van stijgende deellijsten van xs waarvoor `concat yss` gelijk is aan `xs`. Voorbeeld:

```
ontbind [1,7,5,5,7,8,-3,1] = [[1,7],[5],[5,7,8],[-3,1]]
```

Uitwerking.

```
ontbind, ontbindA :: Ord a => [a] -> [[a]]
ontbind [] = []                     -- de lege lijst [] is korter dan [[]]
ontbind [x] = [[x]]
ontbind (x:y:zs) =                  -- us is de eerste stijgende deelrij van (y:zs)
  if x < y then (x:us): vss         -- vss is de ontbinding van de rest
  else [x]: (us: vss)               -- deze haakjes zijn overbodig
  where  us:vss = ontbind (y:zs)
ontbindA = foldr f [] where         -- een alternatief
  f x uss =
    if null uss || x >= head (head uss) then [x]: uss
    else (x:head uss): (tail uss)
```

Opgave 2 (2 punt). De functie

```
inv :: (Int -> Int) -> Int -> Int
```

is zodanig dat `(inv f x)` altijd een getal y oplevert dat voldoet aan `f y` $\leq$ `x` $<$ `f(y+1)`, tenminste als de functie `f` voldoet aan $\lim_{x\to\infty} \mathtt{f}(x) = +\infty$ en $\lim_{x\to-\infty} \mathtt{f}(x) = -\infty$.

Gevraagd wordt een implementatie van `inv` met logaritmische complexiteit, dwz als voor alle $n \geq 2^k$ geldt dat $\mathtt{f}(-n) \leq \mathtt{x} < \mathtt{f}(n)$, dan dient `(inv f x)` berekend te worden in $\mathcal{O}(k)$ stappen. Je mag hierbij aannemen dat `(f x)` berekend wordt in één stap.

Uitwerking. Een variant van binair zoeken. We nemen $p < q$ als invariant. Eerst wordt $q - p$ steeds verdubbeld todat `f p` $\leq$ `x` $<$ `f q`. Dit blijft dan invariant, terwijl `q - p` steeds gehalveerd wordt totdat q-p=1.

```
inv f x = binsearch 0 1 where
  binsearch p q
    | f p > x   = binsearch (2*p-q) q
    | f q <= x  = binsearch p (2*q-p)
    | p+1 == q  = p              -- deze test moet als derde: eerder of later is fout
    | f m <= x  = binsearch m q
    | f m > x   = binsearch p m
    where m = div (p+q) 2
-- probeer inv (\x->x*x*x) (-1000)
```

Ik had eigenlijk moeten vragen naar `inv` getypeerd volgens

```
inv :: (Integer -> Integer) -> Integer -> Integer
```

Sommigen hebben nu de boven- en ondergrens van het type `Int` gebruikt. Dit is minder elegant, maar niet fout gerekend.

Opgave 3 (2 punt). De functie

```
genereer :: (Int -> [Int]) -> [Int] -> [Int]
```

is zodanig dat, als `xs` een stijgende lijst van getallen is en als `(f x)` voor elke `x` een stijgende lijst van getallen $>$ `x` is, dan is `(genereer f xs)` een stijgende lijst die `xs` als deellijst bevat en die met elk element `x` ook de lijst `(f x)` als deelllijst bevat, en die geen verdere elementen bevat. Voorbeeld:

```
genereer (\x->[2*x]) [1,7] = [1,2,4,7,8,14,16,28,32,56, enz
```

Implementeer `genereer`. De lijsten `xs` en `(f x)` moeten leeg, eindig, of oneindig kunnen zijn.
Uitwerking.

```
genereer f [] = []
genereer f (x: xs) = x: genereer f (merge (f x) xs) where
  merge [] ys = ys
  merge xs [] = xs
  merge (x:xs) (y:ys)
    | x < y    = x: merge xs (y:ys)
    | x > y    = y: merge (x:xs) ys
    | x == y   = x: merge xs ys
```

Merk op, dat `genereer` tenminste één element oplevert vóór de recursieve aanroep plaats vindt. Sommige oplossingen voldoen hier niet aan en leveren het goede resultaat alleen als dat resultaat een eindige lijst is, bv voor

```
genereer (\x -> if x < 1000 then [2*x] else []) [1,7]
```

Opgave 4 (2 punt). Beschouw de definities

```
data Htree a = Nil | Node a (Htree a) (Htree a)
fl Nil = []
fl (Node x yh zh) = x: fl yh ++ fl zh
```

We beschouwen nu tevens een functie `flac` gespecificeerd door

```
flac us xh = fl xh ++ us
```

(a) Leid uit deze specificatie een definitie voor `flac` af die efficienter is dan de specificatie omdat het gebruik van de operator `++` eruit geëlimineerd is.
(b) We definieren `afl = flac []`. We schrijven $T(\texttt{fl})(h)$ en $T(\texttt{afl})(h)$ voor het aantal stappen nodig voor de berekening van `fl xh` of `afl xh` als `xh` een volledige (gebalanceerde) boom met hoogte h is. Leid recurrente betrekkingen voor deze twee functies af. Je hoeft deze recurrente betrekkingen niet op te lossen.

Uitwerking, zie ook Bird 7.3.1.

```
(a)   flac us Nil = fl Nil ++ us = [] ++ us = us
        flac us (Node x yh zh)
      =  fl (Node x yh zh) ++ us
      =  (x: fl yh ++ fl zh) ++ us
      =  x: fl yh ++ (fl zh ++ us)
      =  x: fl yh ++ (flac us zh)
      =  x: flac (flac us zh) yh
```

We definiëren dus

```
flac us Nil = []
flac us (Node x yh zh) = x: flac (flac us zh) yh
```

(b) De lijst `(fl yh)` heeft lengte 2^h als `yh` een volledige boom met hoogte h is. Berekening van de `++` in `fl` kost dus 2^h. Er geldt daarom

$$T(\texttt{fl})(0) = 1 \ ,$$
$$T(\texttt{fl})(h+1) = 1 + 2^h + 2 \cdot T(\texttt{fl})(h) \ .$$

We schrijven $T(\texttt{flac})(h)$ voor het aantal stappen ter berekening van `(flac us xh)` als `xh` een volledige boom is met hoogte h. Dit kan omdat `flac` niets met `us` doet. We vinden dan

$$T(\texttt{flac})(0) = 1 \ ,$$
$$T(\texttt{flac})(h+1) = 1 + 2 \cdot T(\texttt{flac})(h) \ .$$

Omdat `afl` in één stap uit `flac` berekend wordt, geeft dit

$$T(\texttt{afl})(0) = 2 \ ,$$
$$T(\texttt{afl})(h+1) = 2 \cdot T(\texttt{afl})(h) \ .$$

Opgave 5 (2 punt). We beschouwen eindige lijsten als een abstract datatype `(List a)` met de operaties

```
nil :: List a
null :: List a -> Bool
cons :: a -> List a -> List a
head :: List a -> a
tail :: List a -> List a
```

en de axioma's

```
null nil = ...
null (cons x xl) = ...
head (cons x xl) = ...
tail (cons x xl) = ...
```

(a) Vul de rechter leden van de axioma's goed in.
(b) We representeren (implementeren) het abstracte datatype met

```
type FList a = (Int -> a, Int)
```

waarbij (f, n) de lijst representeert met lengte n en met $f(k)$ als k–de element.

Geef geschikte operaties `fnil`, `fnull`, `fcons`, `fhead`, `ftail` die de abstracte operatoren `nil`, `null`, `cons`, `head`, `tail` implementeren met type `(List a)` vervangen door `(FList a)`. Geef tevens een abstractiefunctie

```
abstr :: FList a -> List a
```

Uitwerking. Een combinatie van de opgaven 8.1.1 en 8.1.2 uit Bird.

(a) Dit gaf weinig problemen:

```
null nil = True
null (cons x xl) = False
head (cons x xl) = x
tail (cons x xl) = xl
```

(b) De operaties kunnen als volgt geïmplementeerd worden.

```
fnil :: FList a
fnil = (error "empty",0)
fnull :: FList a -> Bool
fnull (f,n) = n == 0
fcons :: a -> FList a -> FList a
fcons x (f,n) = (g,n+1) where
  g 0 = x                      -- we tellen vanaf 0
  g (k+1) = f k
fhead :: FList a -> a
fhead (f,n) = f 0              -- we tellen vanaf 0
ftail :: FList a -> FList a
ftail (f,n) = (g,n-1) where g k = f (k+1)
```

Een goede abstractiefunctie is

```
abstr :: FList a -> List a
abstr ccr =
  if fnull ccr then nil
  else cons (fhead ccr) (abstr (ftail ccr))
```

Merk op, dat `(List a)` een ander type is dan `[a]`. Je mag hier dus niet `[]` en `:` gebruiken. Vergelijk met

```
concr :: FList a -> [a]
concr ccr =
  if fnull ccr then []
  else  (fhead ccr): (concr (ftail ccr))
-- probeer concr (\x -> x*x*x, 100)
```