Tentamen Functioneel Programmeren

21 maart 2002, 9.00–12.00 uur

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 door handig gebruik te maken van standaardfuncties uit Bird's boek.
De opgaven van dit tentamen zijn verdeeld over drie vellen.

Opgave 1. Geef een Haskell-definitie van een functie `gaten` die voor een gegeven niet-lege lijst `xs` oplevert: een lijst bestaande uit alle lijsten die uit `xs` te verkrijgen zijn door één element weg te laten.
Voorbeeld: `gaten`[1,2,3] = [[2,3],[1,3],[1,2]].

Opgave 2. In deze opgave zijn "lijsten"steeds eindige lijsten. Een *segment* van een lijst is een aaneengesloten deellijst.(Formeel: een segment van een lijst `xs` is een lijst us met de eigenschap dat `xs` te schrijven is als: `xs = ys ++ us ++ zs`.)
Geef hieronder steeds de types van de te definiëren functies. Let op de volledige beantwoording van (iii).

(i) Geef een Haskell-definitie van de functie `inits` die elke lijst `xs` overvoert in een lijst bestaande uit alle beginsegmenten van `xs`.
Voorbeeld: `inits [1,2,3] = [[],[1],[1,2],[1,2,3]]`.

(ii) Geef een Haskell-definitie van een functie `segs` die elke lijst `xs` overvoert in een lijst bestaande uit alle segmenten van `xs`. Maak hierbij gebruik van `inits`.

(iii) Geef een recursieve Haskell-definitie van een functie `ings` die aan de volgende specificatie voldoet: `ings xs = (inits xs,segs xs)`.
In deze definitie mag de functie wel zichzelf aanroepen, maar *niet* `inits` en `segs`. Verder moet de efficiëntie zo gunstig mogelijk zijn. Beredeneer hoe via deze definitie een efficiëntere Haskell-implementatie van `segs` te verkrijgen is.

Opgave 3.

(i) Uit de theorie is bekend dat de functie `concat :: [[a]] -> [a]` als volgt met behulp van de functie `foldr` te karakteriseren is:

`concat = foldr(++)[]`.

Geef de Haskell-standaarddefinities van alle drie functies die in deze gelijkheid voorkomen. Geef daarbij ook hun types.

(ii) Definiëer de functies `append` en `rev` door:

```
append x xs = xs ++ [x]
rev = foldr append []
```

Bewijs nu inductief dat voor alle eindige lijsten `xs` en `ys` van hetzelfde type:
`rev(xs ++ ys) = (rev ys) ++ (rev xs)`.
Je mag hierbij gebruik maken van het feit dat de operatie `++` associatief is en de lege lijst als rechtseenheid heeft (d.w.z.: `zs++[]=zs`).

Opgave 4. We beschouwen prefixexpressies ("prefexps") volgens de syntax
prefexp = *operator prefexp prefexp* | *variabele*
variabele = `'a'` | ... | `'z'`
operator = `'+'` | `'-'` | `'*'`
Er mogen geen spaties in de invoer staan.

Definiëer het corresponderende Haskell-datatype `Boom` door:

```
data Boom = Blad Char | Op Char Boom Boom | Fout
```

(i) Geef een Haskell-definitie van een prefixexpressie-parser
`prefparser :: [Char] -> (Boom,[Char])`,
die van een string een prefexp afhaalt en omzet in een boom en het restant van de string als tweede component teruggeeft. Als de invoer niet met een prefexp begint, dan moet een paar `(b,s)` opgeleverd worden zodanig dat `Fout` in de boom `b` voorkomt.
Aanwijzing: maak eerst hulpfuncties van de vorm
`isVariabele, isOperator :: Char -> Bool`.

(ii) Een *environment* is een functie die bij elke (kleine) letter een geheel getal geeft. Construeeer een recursieve valuatiefunctie
`val :: Boom -> (Char -> Integer) -> Integer`
met de volgende eigenschap: als `b` een prefexp representeert en `e` een environment, dan is `(val b e)` de corresponderende waarde van die prefexp (ten aanzien van de standaardinterpretatie van `'+'`, `'-'` en `'*'` als de vertrouwde rekenkundige bewerkingen).

Opgave 5.

(i) Geef een Haskell-definitie van een functie die, gegeven een oneindige lijst van type `[Integer]`, de waarde oplevert van de eerste twee gelijke opeenvolgende elementen van de lijst. (Voorbeeld: deze functie levert 5 op voor de lijst `[6,3,7,5,5,8,4,3,3,3,3,3,....]`.)

(ii) Geef een Haskell-definitie van een functie die, gegeven een functie `f` van type `(a -> a)` en een startwaarde `x` van type `a`, de oneindige lijst `[x, f x, f(f x), f(f(f x)), ....]` van iteraties van `f` op `x` bepaalt.

(iii) Als `f` een functie is van type `(Integer -> Integer)` is, dan heet een `x` van type `Integer` een *dekpunt* van `f` als `(f x) = x` .

Uitgaande van een functie `f` van type `(Integer -> Integer)` en een `x` van type `Integer` zou men kunnen proberen een dekpunt van `f` te vinden door `f` herhaald toe te passen op `x`. Als `x` zelf al een dekpunt is, dan levert dit zoekproces direkt al `x` zelf op. Als `x` en `(f x)` geen dekpunten zijn van f, maar `f(f x)` wel, dan levert het zoekproces die waarde `f(f x)` op.

Geef een Haskell-definitie van de functie

```
dek ::  (Integer -> Integer) -> Integer -> Integer
```

die het resultaat van dit zoekproces oplevert (als dit bestaat). In de twee voorbeelden moeten we dus krijgen: `dek f x = x`, respectievelijk `dek f x = f(f x)` (maar de waarde van, bijvoorbeeld, `dek (+1) 1` is ongedefiniëerd).

Maak in de definitie van `dek` gebruik van de functies uit (ii) en (i).

(iv) Geef ook een definitie van `dek` met behulp van de functie `until`.