

## Uitwerking deeltentamen Functioneel programmeren 18 december 2002

De antwoorden worden hieronder uitgebreid toegelicht. Dat is hier gedaan als extra uitleg; op het tentamen hoef je dat zelf niet te doen. Bij de MKV-vragen volstaat de letter; bij de open vragen de functiedefinitie.

1. Het goede antwoord is C. Herinner je de definitie van de functie `concat`, die een lijst van lijsten allemaal aan elkaar plakt, door tussen alle elementen de `++`-operator te zetten: `concat = foldr (++) []`. Deze functie werkt op een lijst van lijsten, en heeft een gewone lijst als resultaat. De expressie in deze opgave is hetzelfde, maar mist nog die lege-lijstparameter. Deze functie heeft dus nog een extra parameter:  $[a] \rightarrow [[a]] \rightarrow [a]$ .
2. Het goede antwoord is A. Dat is nu eenmaal de betekenis van de lijstcomprehensie-notatie.
3. Het goede antwoord is C. Je kunt de betekenis van `foldr` als volgt onthouden: begin aan de rechterkant (vandaar de ‘r’ in ‘foldr’) van de lijst met het neutrale element, en zet de operator dan steeds tussen de elementen. Het duidelijkst zie je dat als je de operator inderdaad als infix-operator schrijft, zoals in antwoord A. De expressie `3 'f' 0` is hetzelfde als `f 3 0`. Schrijf je de `foldr` op deze manier uit, dan krijg je het antwoord onder B. De betekenis van `(f 1 . f 2 . f 3) 0` is: doe eerst `f 3` op 0, en doe daarna `f 2` op het resultaat, en dan `f 1` op het resultaat daarvan. De expressie achter antwoord D is dus ook hetzelfde als achter B. Alleen antwoord C is anders: de functie `f` krijgt zijn parameters hier in omgekeerde volgorde, en dat is echt wat anders.
4. Het goede antwoord is B. Je kunt het puur formeel bekijken: het pijltje in types associeert naar rechts, en dat geeft per definitie dat  $(a \rightarrow b) \rightarrow c \rightarrow d$  gelijk is aan  $(a \rightarrow b) \rightarrow (c \rightarrow d)$ . Of je kunt de Currying proberen te begrijpen: het type in de vraagstelling heeft een functie  $(a \rightarrow b)$  als parameter, daarna nog een  $c$ , en uiteindelijk een  $d$  als resultaat. Als hij dus die functie  $(a \rightarrow b)$  heeft gehad, is er nog een functie  $(c \rightarrow d)$  overgebleven.
5. Het goede antwoord is B. Dankzij lazy evaluatie kun je de oneindige lijst getallen aan de functies meegeven, en krijg je toch het eindige deel te zien van het antwoord: de getallen 0 t/m 99. In het geval van `take` en `takeWhile` wordt de lijst daarmee ook afgesloten. De functie `filter` echter moet, alvorens hij de lijst kan afsluiten, eerst controleren of er verderop in de oneindige lijst misschien nog kleine getallen staan. Daar gaat hij oneindig lang mee door, en de lijst kan dus nooit echt worden afgesloten.
6. Het goede antwoord is D. Bedenk eerst wat de betekenis is van zo’n data-definitie. Er wordt een nieuw type `Tree a` gedefinieerd, en twee constructorfuncties `Tak` en `Blad`. Het type van de parameters staat erachter. Aldus geldt dus: `Tak :: a -> (Boom a) -> (Boom a)`. In de aanroep `Tak 1 Blad` krijgt `Tak` alvast twee parameters. De eerste parameter is een `Int`. Dat mag, want de functie kan elk willekeurig type `a` accepteren, maar hiermee hebben we dus wel `a` geconcretiseerd als `Int`. De tweede parameter, `Blad`, is een boom, en dat is dus precies wat we nodig hebben. De derde parameter, ook een boom, heeft-ie dan nog niet gehad. Het resultaat van de constructorfunctie is natuurlijk ook een boom. Het type is dus `Boom Int -> Boom Int`.

7. Dit is op twee practicumssessies geoefend, op het college besproken en als quiz aan de orde geweest, en het staat als voorbeeld letterlijk in het diktaat. In dit geval gaat het om lijsten van getallen, dus we kunnen de functie gewoon voor `Int` schrijven, met gebruikmaking van de operator `<=`. Gedoe met `Ordering` is hier niet nodig. (Eigenlijk had er ‘gehele getallen’ in de opgave moeten staan, want `Float` zijn ook getallen. Nou ja, als je de functie toch generiek had aangepakt is het ook goedgerekend). hier zijn de functiedefinities nog eens:

```
insert x [] = [x]
insert x (y:ys) | x<=y = x : y : ys
                  | x>y  = y : insert x ys
sorteer [] = []
sorteer (x:xs) = insert x (sorteer xs)
sorteer2 = foldr insert []
```

8. De zeer oplettende lezer van het diktaat heeft deze functie misschien zien staan in het hoofdstuk over de vergelijking met Prolog. De functie `splits` is immers in zekere zin het omgekeerde van `++`. In Prolog krijg je dat kado, in Haskell moet er de nodige moeite worden gedaan. Je kunt de functie bedenken op dezelfde manier als al die andere functies in hoofdstuk 4.1. De hint die je nodig hebt voor de oplossing krijg je door in het voorbeeld het resultaat van de recursieve aanroep `splits [2,3,4]` uit te werken. Je ziet dan dat je in al die tupeltjes de linkerhelft moet uitbreiden met het getal 1, en dat er bovendien nog een geheel nieuw tupel voorgezet moet worden. Wat betreft de recursiebasis: er is één manier om een lege lijst te splitsen, namelijk in twee lege lijsten. De lijst met oplossingen bestaat in dit geval dus uit één tupel.

```
splits []      = [ ([],[]) ]
splits (x:xs) = nieuw : map f (splits xs)
                where f (links,rechts) = (x:links, rechts)
```

Sommige slimmeriken hebben gezien dat je het gebruik van recursie kunt vermijden door gebruik te maken van de (op college niet behandelde, maar wel in de prelude aanwezige) functie `splitAt`, in combinatie met een lijstcomprehensie of met `map`:

```
splits' xs = [ splitAt n xs | n <- [0..length xs] ]
splits'' xs = map (\n -> splitAt n xs) [0..length xs]
```

9. Om een functie te schrijven op data-structuren gebruik je meestal patroonherkenning voor alle constructoren. Daarbij moet je namen verzinnen voor de parameters. In dit geval is er ook nog een extra functie-parameter `f`. De invuloefening wordt dus:

```
overall f Leaf      =
overall f (Fork e ts) =
```

Hierin is `e` het element dat in de fork-punten is opgeslagen, en `ts` is de lijst van uitsplitsende takken. De functie `f` moet op alle elementen worden toegepast, dus we hebben in ieder geval `f x` nodig. De lijst `ts` is een lijst van trees. Op elk van deze trees

kan de functie recursief worden toegepast; `map` zorgt ervoor dat dit op alle trees apart gebeurt: `map (overall f) ts`. De respectievelijke resultaten worden weer samengepakt in een nieuwe `Fork`. Bij het `Leaf`-geval verandert er niets, omdat hier geen elementen zijn opgeslagen.

```
overall f Leaf          = Leaf
overall f (Fork e ts) = Fork (f e) (map (overall f) ts)
```

Het type van `overall` is als volgt te beredeneren: er gaat een functie `a->b` in, en een `Boom a`, en door het toepassen van de functie op al die `a`'s, wordt het een `Boom b`. Dus `overall :: (a->b) -> Tree a -> Tree b`.

Eventueel had je de opgave kunnen interpreteren als zou er een lijst (in plaats van een boom) met resultaten uit moeten komen. Daarom is ook het volgende goedgekeurd:

```
overall' :: (a->b) -> Tree a -> [b]
overall' f Leaf          = []
overall' f (Tree e ts) = f e : concat (map (overall' f) ts)
```

10. Drie mogelijke definities van een functie die de (oneindige) tafel van een bepaald getal geeft:

```
tafel  n = map ((*n) [1..])
tafel' n = [n, n+n, .. ]
tafel'' n = iterate ((+)n) n
```

Als je die functie eenmaal hebt, hoef je hem alleen nog maar op alle getallen toe te passen om het gevraagde oneindige blok te krijgen:

```
tafels = map tafel [1..]
```

In beide gevallen kun je desgewenst in plaats van `[1..]` ook schrijven `iterate ((+)1) 1`.

De functie `diagonaal` is een rechstreekse toepassing van recursie:

```
diagonaal (xs:xss) = head xs : diagonaal (map tail xss)
```

Een recursiebasis is niet nodig, omdat hij alleen maar op een oneindige lijst wordt gebruikt. Maar als je wilt kun je nog toevoegen: `diagonaal [] = []`.

Ook goed (maar minder leuk) is gebruik te maken van de operator `!!`, in combinatie met een lijst-comprehensie of anders met een `map`. Ook had je `zipWith` slim kunnen inzetten:

```
diagonaal'  xss = [ (xss!!n)!!n | n <- [0..] ]
diagonaal'' xss = map (\n->(xss!!n)!!n) [0..]
diagonaal''' xss = zipWith (!!)' xss [0..]
```