Uitwerking tentamen Functioneel Programmeren—9 november 2007

1. (10 punten)

```
foldr1 :: (a -> a -> a) -> [a] -> a

foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)
```

2. (20 punten)

a)

```
zip :: [a] -> [b] -> [(a,b)]
zip [] bs = []
zip as [] = []
zip (a:as) (b:bs) = (a,b):(zip as bs)
```

b)
basis:

```
  zip (fst (unzip [])) (snd (unzip []))
= { def. unzip}
  zip (fst ([],[])) (snd ([],[]))
= { def. fst,snd }
  zip [] []
= { def. zip }
  []
```

Ind.hypo: Stel het is bewezen voor xs, laat het dan nu zien voor (a,b):xs.

```
  zip (fst (unzip ((a,b):xs)) (snd (unzip ((a,b):xs))
= { def. unzip, (*) let (as,bs) = unzip xs }
  zip (fst (a:as,b:bs)) (snd (a:as,b:bs))
= { def. fst,snd, (*) }
  zip (a:as) (b:bs)
= { def. zip, (*) }
  (a,b):(zip as bs)
= { def. fst,snd, (*) }
  (a,b):(zip (fst (as,bs)) (snd (as,bs)))
= { (*) }
  (a,b):(zip (fst (unzip xs)) (snd (unzip xs)))
= { Ind.hypo }
  (a,b):xs
```

Q.E.D.

3. (15 punten)

a)

```
curry :: ((a,b) -> c) -> (a -> b -> c)
uncurry :: (d -> e -> f) -> ((d,e) -> f)
Als uncurry argument van curry wil zijn, moet gelden
(d -> e -> f) = ((a,b) -> c), we moeten dan kiezen
c = (e -> f), maar we kunnen geen a,b kiezen
zodanig dat (a,b) = d.
```

b)

```
uncurry :: (a -> b -> c) -> ((a,b) -> c)
zip :: [d] -> [e] -> [(d,e)]
Als zip argument is van uncurry, geldt
a = [d], b = [e], c = [(d,e)]
dus is
uncurry zip :: ([d],[e]) -> [(d,e)]
hetgeen het type is van de inverse functie van unzip!
```

4. (15 punten)

a)

```
inter (S []) _ = S []
inter _ (S []) = S []
inter (S (x:xs)) (S (y:ys))
  | x < y = inter (S xs) (S (y:ys))
  | x==y = S (x:zs)
  | otherwise = inter (S (x:xs)) (S ys)
    where S zs = inter (S xs) (S ys)
```

b)

```
subset (S []) _ = True
subset (S (x:xs)) b = member x b && subset (S xs) b
```

c)

```
makeSet = S . remDups . sort
   where remDups [] = []
         remDups [x] = x
         remDups (x:y:ys)
            | x < y = x:(remDups (y:ys))
            | otherwise = remDups (y:ys)
```

d)

```
mapSet f ( S xs) = makeSet (map f xs)
```

5. (15 punten)

a)

```
newtype Exp = E Exp Exp | C Char | I Int
```

b)

```
pExp :: Parse Char Exp
pExp = pEEExp ‘alt‘ pCExp ‘alt‘ pIExp

pEEExp = (pExp >*> token ’,’ >*> pExp) ‘build‘ mkEEExp
mkEEExp (e1,(_,e2)) = E e1 e2

pCExp = (token ’C’ >*> spot isLetter) ‘build‘ mkCExp
mkCExp (_,c) = C c

pIExp = (token ’I’ >*> spot Isdigit) ‘build‘ mkIExp
mkIexp (_,i) = I (ord i - ord ’0’)

isLetter c = ’a’<=c && c<=’z’
isDigit c  = ’0’<=c && c<=’9’
```