

TENTAMEN I CD5100 FUNKTIONELL PROGRAMMERING

Torsdagen den 17 augusti 2006, kl 8:30 – 13:30

LÖSNINGSFÖRSLAG

UPPGIFT 1 (6 POÄNG)

a) Vi gör en lösning med ackumulerande argument (hjälpfunktionen `bs2int1`), där det ackumulerande argumentet successivt samlar på sig heltalet som ska returneras:

```
bs2int 1 = bs2int1 0 1
```

```
bs2int1 n [] = n
```

```
bs2int1 n (b:bs) = bs2int1 (2*n + b) bs
```

b) Vi plockar successivt ut bitarna i den binära representationen genom att dela med två och testa om det blir någon rest. En komplikation är att vi då får ut bitarna i omvänd ordning mot den önskade (minst signifikant bit först), så vi använder standardfunktionen `reverse` för att vända på resultatet på slutet:

```
int2bs 1 = reverse (int2bs1 1)
```

```
int2bs1 0 = [0]
```

```
int2bs1 1 = [1]
```

```
int2bs1 n = (n `mod` 2) : int2bs1 (n `div` 2)
```

UPPGIFT 2 (3 POÄNG)

Trots att anropet `f (-3)` inte terminerar, kommer `fst (17, f (-3))` att evalueras till 17 i Haskell. Detta beror på den lata evalueringen, som inte räknar ut mer än nödvändigt av argumentet till `fst` för att kunna returnera svaret. I definitionen av `fst` kommer `x` direkt att matchas mot 17 som då direkt returneras eftersom det redan är färdiguträknat.

UPPGIFT 3 (4 POÄNG)

Man kan använda standardfunktionerna `zipWith` och `map` för att enkelt definiera `vadd` och `scale`:

```
vadd l1 l2 = zipWith (+) l1 l2
```

```
scale x l = map (x *) l
```

`zipWith` kommer då att bestämma hur fallet med olika långa vektorer hanteras. Vad som händer är att eventuella "överskjutande" element i en lista ignoreras, eftersom `zipWith` terminerar när den kortaste listan tar slut. Detta kan anses vara ett vettigt beteende, om vi betraktar de första dimensionerna i en vektor som dimensioner av "lägre" grad: semantiken hos `vadd` blir då att man bara betraktar de "relevanta" elementen av den längre vektorn. Andra sätt att hantera fallet med olika långa vektorer är också minst lika vettiga, t.ex. kan man kräva att vektorerna är lika långa för att få addera dem.

UPPGIFT 4 (5 POÄNG)

```
class Vector a where
  vadd :: a -> a -> a

instance (Num a) => Vector [a] where
  vadd l1 l2 = zipWith (+) l1 l2
```

UPPGIFT 5 (2 POÄNG)

[getChar, getChar, getChar, getChar, getChar] är bara en lista av actions. Den kan skickas runt i Haskell-programmet som argument, eller returneras som resultat, men ingen action kommer att utföras innan man kombinerar alla actions i listan till en "stor" action, med do-notation, och placerar den i en omgivning där den exekveras, t.ex. genom att skriva den på kommandoraden i hugs eller att lägga den i main-funktionen i ett kompilerat Haskellprogram.

UPPGIFT 6 (7 POÄNG)

a)

```
data TTree a = Leaf a | Node a (TTree a) (TTree a) (TTree a) deriving Show
```

("deriving Show" har vi med bara för att enkelt kunna skriva ut resultat av typen TTree a.)

b)

```
sumtree (Leaf x) = x
sumtree (Node x t1 t2 t3) = x + sumtree t1 + sumtree t2 + sumtree t3
```

c)

```
foldtree f (Leaf x) = x
foldtree f (Node x t1 t2 t3) =
  f (f (f x (foldtree f t1)) (foldtree f t2)) (foldtree f t3)
```

UPPGIFT 7 (3 POÄNG)

Betrakta deklARATIONEN av head:

```
head (x:xs) = x
head [] = error "Cannot take head of empty list"
```

(:) har typ a -> [a] -> [a], för godtycklig typ a. Antag nu att x har typ a och xs typ [a] (det mest generella antagandet om deras respektive typer). Då erhåller vi x:xs :: [a], och ur första deklARATIONEN att head :: [a] -> a. Hur stämmer det med den andra deklARATIONEN? [] har typ [a], för godtycklig typ a. Om b = a så stämmer argumenttypen för head. error, slutligen, har typ String -> c för godtyckligt c. Väljer vi c = a får högerledet i deklARATION två typen a vilket stämmer med typen [a] -> a för head.

Resultatet är alltså att head :: [a] -> a. Eftersom vi aldrig gjort starkare antaganden än absolut nödvändigt om typen på någon storhet följer att detta är den mest generella typen för head.