

Algorithmen und Programmierung

10. Aufgabenblatt

Aufgabe 10.1 (9.2)

a) Instanz von Ord für Tupel

Für eine Instanz von *Ord* wird immer eine Instanz von *Eq* benötigt.

```
instance (Ord x, Ord y) => Eq (x,y) where
  -- Für Eq reicht es (==) zu definieren
  (==) (a,b) (c,d) = a==c
```

Nun definieren wir *Ord* auf Tupeln, indem wir den Vergleich von 2 Tupeln auf den Vergleich der beiden Tupel Elemente zurückführen. Ein Tupel (a,b) soll \leq einem Tupel (c,d) sein, wenn das erste Element jedes Tupels \leq ist.

```
instance (Ord x, Ord y) => Ord (x,y) where
  -- Für Ord reicht es (<=) zu definieren
  (<=) (a,b) (c,d) = a<=c
```

Instanz von Ord für Listen

Analog zur Instanz für Tupel benötigen wir zuerst eine Instanz von *Eq*.

```
instance (Ord a) => Eq [a] where
  -- Für Eq reicht es (==) zu definieren
  (==) [] [] = True
  (==) (x:xs) (y:ys) = x==y && (==) xs ys
```

Wir wollen für die Listen eine lexikographische Ordnung realisieren. Eine Liste ist lexikographisch kleiner als eine andere Liste, falls das erste Zeichen kleiner ist oder das erste Zeichen übereinstimmt, jedoch der Rest der Liste kleiner ist:
 $abc \leq abd$ liefert True

```
instance (Ord a) => Ord [a] where
  -- Für Ord reicht es (<=) zu definieren
  (<=) [] [] = True
  (<=) (x:xs) (y:ys) = x<y || x==y && (<=) xs ys
```

b) Unifikation

(1) (Int, a,a) - (b,b,[Bool])

Es gilt: a == [Bool]
 b == Int

```

a == b
=> Int == a == [Bool]
=> Widerspruch

```

Die beiden Tupel können nicht unifiziert werden, da der polymorphe Typ a im linken Tupel zu [Bool] und im rechten Tupel zu Int unifiziert werden müsste. Damit müsste a == Int und == [Bool] sein => Widerspruch.

(2) (Int -> b) - (a -> Bool)

```

Es gilt:   b == Bool
           a == Int
=> Unifikation nach (Int, Bool) möglich

```

(3) (a, [a]) - (b,c)

```

Es gilt:   a == b
           [a] == c
=> Unifikation nach (a, [a]) möglich

```

Zusatzfragen:

Unifikation möglich nach

```

(Bool, [Bool])?   Möglich, ersetze dafür a in (a, [a]) durch Bool
([Int], [[Bool]])? Nicht möglich, a in (a, [a]) müsste [Int] und [Bool] sein
([Int], [[Int]])? Möglich, ersetze dafür a in (a, [a]) durch [Int]

```

Aufgabe 10.1 (9.2)

```

data Winkel =      Grad Int Int Float |
                  Neugrad Float |
                  Rad Float

-- eigene Definition für pi,
-- damit weniger Rundungsfehler auftreten
mypi = 3.141592

-- Hilfsfunktionen --

-- Normalisierung eines Winkels
norm :: Winkel -> Winkel
norm (Rad a)
  | a >= 2*mypi      = norm (Rad (a-2*mypi))
  | a < 0             = norm (Rad (a+2*mypi))
  | otherwise        = Rad a
norm (Neugrad a) = Neugrad (a `mod` 400)

-- Uebertrag beachten: 0° 61' == 1° 1'
norm (Grad a b c)
  = Grad d e (toEnum f)

```

```

        where
            d = a `mod` 360 + b `div` 60
            e = b `mod` 60 + fromEnum c `div` 60
            f = fromEnum c `mod` 60

-- Umwandlung in Rad
toRad :: Winkel -> Winkel
toRad (Rad a) = Rad a
toRad (Grad a b c)
    = Rad (( toEnum a + (toEnum b/60) + (c/360))/180 * mypi)
toRad (Neugrad a) = Rad (a/(400*mypi))

-- Umwandlung in Grad
toGrad :: Winkel -> Winkel
toGrad (Grad a b c) = Grad a b c
toGrad (Neugrad a) = toGrad (toRad (Neugrad a))
toGrad (Rad a)
    = Grad grad minuten sekunden
    where
        b = a*180 / mypi
        -- Nachkommastellen durch Abrunden entfernen
        grad = floor b
        -- Nachkommastellen extrahieren
        c = (b - toEnum grad) * 60
        minuten = floor c
        -- Nachkommastellen extrahieren
        sekunden = (c - toEnum minuten) * 60

instance Eq Winkel where
    -- Für Eq genügt die Definition von (==)
    (==) (Rad a) (Rad b) = c==d
        where
            (Rad c) = norm (Rad a)
            (Rad d) = norm (Rad b)
    (==) a b = (toRad (norm a)) == (toRad (norm b))

instance Ord Winkel where
    -- Für Ord genügt die Definition von (<=)
    (<=) (Rad a) (Rad b) = c<=d
        where
            (Rad c) = norm (Rad a)
            (Rad d) = norm (Rad b)
    (<=) a b = (toRad (norm a)) <= (toRad (norm b))

instance Num Winkel where
    (+) (Rad a) (Rad b) = norm (Rad (a+b))
    (+) a b = (toRad (norm a)) + (toRad (norm b))
    (-) (Rad a) (Rad b) = norm (Rad (a-b))
    (-) a b = (toRad (norm a)) - (toRad (norm b))

    (*) = error ""
    signum = error ""
    abs = error ""
    fromInteger = error ""

```

```

-- Ausgabe von Winkeln erfolgt in Grad
instance Show Winkel where
  show (Grad a b c)
    = show d ++ "° " ++ show e ++ "' " ++ show f ++ "\""
      where (Grad d e f) = norm (Grad a b c)
  show a = show (toGrad (norm a))

```

Testläufe

```

*Winkel> (Neugrad 30) == (Neugrad 430)
True
*Winkel> (Rad (2*mypi)) == (Rad (4*mypi))
True
*Winkel> (Grad 360 0 0)
0° 0' 0.0"
*Winkel> (Rad 0)
0° 0' 0.0"
*Winkel> (Rad 0) <= (Grad 0 0 0)
True
*Winkel> (Grad 360 0 0) <= (Grad 0 0 0)
True
*Winkel> (Rad (2*mypi)) <= (Rad 0)
True
*Winkel> (Rad (2*mypi)) <= (Grad 360 0 0)
True

```

Aufgabe 10.3

Folgende Funktionen sind aus der Vorlesung bekannt:

```

-- Wandelt String in Int um
getInt' :: String -> Int
getInt' xs = read xs

-- Interaktionsfunktion für IO-Operationen
interaction :: (String->String) -> IO()
interaction f
  = do x <- getLine
      if (x == []) then return ()
      else do putStrLn ((f x))
              interaction f

```

a) Die Funktion *echoInt* liest so lange Zahlen ein, bis die Eingabe leer ist.

```

echoInt :: IO ()
echoInt = interaction (show.getInt')

```

Testlauf:

```
*Programme> echoInt
11111
11111
2222222
2222222
```

```
*Programme>
```

b) Die Funktion *palindrom* liest einen String ein und prüft, ob es sich um ein Palindrom handelt.

```
palindrom :: IO ()
palindrom = interaction(\x -> show ((reverse x)==x))
```

Testlauf:

```
*Programme> palindrom
dieliebegehthegebeileid
True
otto
True
```

Patrick Schäfer et al.