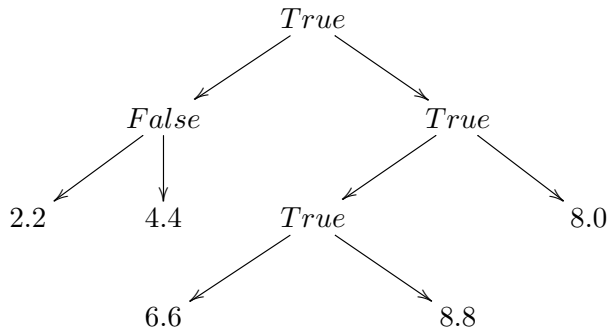
Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
13. Mai 2014

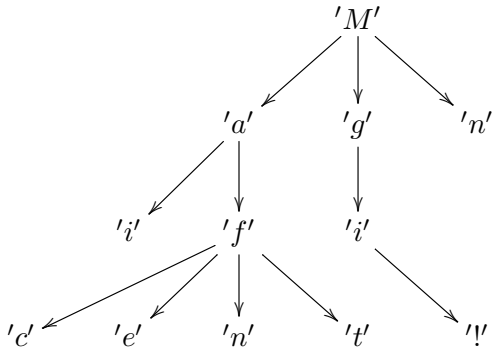
4. Musterlösung zur Vorlesung
# Programmierung und Modellierung

**A4-1** ***Bäume***

a) Schreiben Sie einen Datentyp, mit dem man folgenden Baum repräsentieren kann:




b) Schreiben Sie erneut einen Datentyp, mit dem man folgenden Baum repräsentieren kann (es darf auch die wieder gleiche Antwort sein, falls passend):




*Hinweis:* Um Ihre Lösung zu überprüfen ist es hilfreich, anschließend Werte des kreierten Datentypen zu definieren, welche die gezeigten Bäume implementieren.

**LÖSUNGSVORSCHLAG:**
Die Lösung dieser Aufgabe ist nicht ganz eindeutig. So kann man die ersten beiden Aufgaben jeweils mit speziellen Datentypen, oder auch mit einem parametrisierten Datentypen lösen.

```
data BoolDoubleBinaerBaum -- löst a)
  = CBBlatt Double
  | CBKnoten BoolDoubleBinaerBaum Bool BoolDoubleBinaerBaum

data CharBaum             -- löst b)
  = CBlatt Char
  | C1Knoten Char CharBaum
  | C2Knoten Char CharBaum CharBaum
  | C3Knoten Char CharBaum CharBaum CharBaum
  | C4Knoten Char CharBaum CharBaum CharBaum CharBaum

data Tree a b           -- löst a) und b)
  = Leaf a
  | Node b [Tree a b]
deriving (Show) -- um den Baum anzeigen zu können

baum1 :: Tree Double Bool
baum1 = Node True
          [ Node False [Leaf 2.2, Leaf 4.4]
          , Node True [Node True [Leaf 6.6, Leaf 8.8], Leaf 8.0]
          ]

baum2 :: Tree Char Char
baum2 = Node 'M'
          [ Node 'a' [Leaf 'i', Node 'f' [Leaf 'c', Leaf 'i', Leaf 'e', Leaf 'n']]
          , Node 'g' [Node 'i' [Leaf 't']]
          , Leaf 'n'
          ]
```

Beide Varianten haben Vor- und Nachteile: so erlaubt `Tree` beliebig viele Kind-Knoten, was man nicht immer erlauben möchte. Wenn man auf den Konstruktor `Leaf` verzichten würde, wäre `Tree` übrigens immer noch eine Lösung zu b), da man Blätter auch mit leeren Listen kodieren könnte. Allerdings erzwingt dies dann den gleichen Typ in Blatt und Knoten, was für Aufgabe a) nicht ausreicht.

**A4-2** ***Ordnung in Wald*** Gegeben ist der folgenden Datentyp:

```
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
```

Wir möchten Bäume nach Ihrer Größe ordnen, fügen Sie dazu den Typ `Tree a b` in die Typklasse `Ord` ein, unter der Bedingung, dass die Typen `a` und `b` eine Ordnung in Form einer Instanz von `Ord` vorliegt.

a) Ohne etwas zu implementieren: Was genau müssten wir denn dazu implementieren? Welche Funktion-, Typ-, Klassen- oder Instanzendeklarationen benötigen wir?

b) Implementieren Sie nun das notwendige dazu. Vergleichen Sie zwei Bäume rekursiv: Die Ordnung zwischen zwei Blättern entspricht der Ordnung der Werte die sie tragen. Weiterhin ist jedes Blatt kleiner als jeder Knoten. Die Ordnung zwischen zwei Knoten entspricht zunächst der Ordnung Ihrer enthaltenen Werte. Falls die Werte gleich sind, dann vergleichen Sie zunächst den linken Teilbaum. Sind auch diese gleich, entspricht die Ordnung der beiden Knoten der Ordnung der beiden rechten Teilbäume.

Beispiele:

```
b1,b2,b3,b4,b5 :: Tree Int Char
b1 = Leaf 42
b2 = Node (Leaf 0) 'b' b1
b3 = Node b1 'b' (Leaf 7)
b4 = Node b2 'a' b3
b5 = Node b2 'a' (Node b1 'c' (Leaf 5))
```

```
> b1 > b2
False
> b3 > b2
True
> b4 >= b3
False
> b5 >= b4
True
```

**LÖSUNGSVORSCHLAG:**

Wir müssen den Typ `Tree a b` zu einer Instanz der Klasse `Ord` machen. Dabei dürfen wir `Ord a` und `Ord b` annehmen. In der Instanzdeklaration für `Ord` müssen wir lediglich die Funktion `compare :: (Ord a, Ord b) => (Tree a b) -> (Tree a b) -> Ordering` implementieren. Dazu setzen wir 1:1 den Text der zweiten Teilaufgabe in Code um:

```
instance (Ord a, Ord b) => Ord (Tree a b) where
  compare (Leaf x) (Leaf y)         = compare x y   -- für Ord a
  compare (Leaf _) (Node _ _ _)     = LT
  compare (Node _ _ _) (Leaf _)     = GT
  compare (Node l1 x1 r1) (Node l2 x2 r2)
    | EQ == cmpMiddle, EQ == cmpLeft = cmpRight
    | EQ == cmpMiddle                = cmpLeft
    | otherwise                      = cmpMiddle
    where cmpMiddle = compare x1 x2                -- für Ord b
          cmpLeft   = compare l1 l2                -- für Tree a b, rekursiv
          cmpRight  = compare r1 r2                -- für Tree a b, rekursiv

instance (Ord a, Ord b) => Eq (Tree a b) where
  t1 == t2 = EQ == compare t1 t2
```

Da die Typklasse `Ord` eine Unterklasse der Typklasse `Eq` ist, müssen wir ebenfalls eine Instanzdeklaration für `Eq` angeben - hierbei können wir jedoch das zuvor definierte `compare` benutzen. Die Reihenfolge dieser beiden Instanzdeklarationen innerhalb des Programms ist tatsächlich egal.

**A4-3** ***Abstiegsfunktion III*** Beweisen Sie, dass die Funktion `baz` für nicht-leere Strings, welche nur das Zeichen `'a'` enthalten, wohldefiniert ist; d.h. das die Funktion ohne Fehlermeldung mit einem Ergebnis terminiert.

```
baz :: String -> Double
baz ('a':'a':'a':xs) = 3 * (baz ('a':xs))
baz ('a':c:xs) = (baz ('b':'b':xs)) / 2
baz ('b':d:xs) = 1 + (baz (d:xs))
baz ('c':xs) = undefined
baz [x] | x /= 'd' = 2
        | x == 'e' = 2 * baz ('a':x:'c':[])
```

**LÖSUNGSVORSCHLAG:**

**FEHLERTEUFEL:** Ursprünglich wurde in der zweiten definierenden Gleichung nur ein `b`-Zeichen hinzugefügt anstatt zwei, was die Aufgabe etwas einfacher macht. Die Lösung hier funktioniert für beide Versionen. Die reine Präsenzaufgabenlösung behandelt die einfacherer Version mit einer einfacherer Abstiegsfunktion.

**Auf)** Für die Teilmenge $A_0 = \{s \mid s$ besteht aus ein oder mehreren `'a'`-Zeichen$\}$ ist Auf) verletzt, denn der rekursive Aufruf der zweiten Gleichung bringt `'b'` Zeichen in das Argument hinein.

Daher müssen wir die Aussage verallgemeinern: Wir beweisen, dass die Funktion `baz` für nicht-leere Strings wohldefiniert ist, falls der Eingabestring nur aus Wiederholungen der Zeichen `'a'` und `'b'` besteht. Es sei also $A'$ die Menge aller nicht-leeren Strings über den Zeichen `'a'` und `'b'`. Wenn wir dies bewiesen haben, dann folgt auch die ursprüngliche Forderung wegen $A_0 \subset A'$.

Wir prüfen erneut: der rekursive Aufruf der ersten Gleichung führt nicht aus $A'$ heraus, da für den Aufruf nur das Zeichen `'a'` hinzugefügt wird; der zweite rekursive Aufruf ist auch ok, denn es werden nur `'b'` hinzugefügt; der dritte rekursive Aufruf macht das Argument nur kleiner, stellt aber durch `(:)` sicher, dass der String nicht leer ist, d.h. wenn dieser in $A'$ war, so bleibt er auch in $A'$. Den rekursiven Aufruf der letzten Zeile brauchen wir nicht zu beachten, da dieser Aufruf nur passiert, falls die Eingabe aus einem `'e'`-Zeichen besteht, aber `"e"` $\notin A'$ gilt.

**Def)** Probleme drohen nur für die vierte Gleichung, wegen der Verwendung von `undefined`; und falls die Eingabe leer ist, da dann jegliches Pattern-Matching fehlschlägt. Beide Fälle können jedoch für Eingaben aus $A'$ nicht eintreten, denn $A'$ enthält keine Strings, welche mit dem Zeichen `'c'` beginnen und auch nicht den leeren String.

Damit ist `baz` für alle Argumente in $A'$ definiert.

**Abst)** Da `'a'`-Zeichen in der Eingabe rekursive Aufrufe mit zwei `'b'`-Zeichen hervorrufen können, müssen `'a'`-Zeichen in der Eingabe "teurer" sein. Als Abstiegsfunktion $m : A' \to \mathbb{N}$ wählen wir daher die Anzahl der `'b'` Zeichen plus drei Mal die Anzahl der `'a'` Zeichen in der Eingabe. Die Summe von Zeichen-Anzahlen ist offensichtlich eine natürliche Zahl. Wir betrachten alle drei rekursive Aufrufe, welche für Argumente aus $A'$ eintreten können:

a) $m(\texttt{'a':'a':'a':}xs) = 9 + m(xs) > 3 + m(xs) = m(\texttt{'a':}xs)$

b) $m(\texttt{'a':c:}xs) = 3 + m(\texttt{c:}xs) > 3 + m(xs) > 2 + m(xs) = m(\texttt{'b':'b':}xs)$ Die erste Ungleichung folgt, da nach Annahme `c` ja entweder ein `'a'` oder ein `'b'` Zeichen sein muss. Wenn wir dieses Zeichen entfernen, dann wird die Abstiegsfunktion entweder um 3 oder um 1 kleiner - in beiden Fällen also echt kleiner.

*Hinweis:* Ob `xs` die leere Liste ist oder nicht ist hier unerheblich, das ist nur für Def) relevant, d.h. der rekursive Aufruf findet nur für nicht leere Listen statt. Unsere Abstiegsfunktion kann aber mit leeren Listen umgehen.

c) $m(\texttt{'b':d:}xs) = 1 + m(\texttt{d:}xs) > m(\texttt{d:}xs) = m(\texttt{d:}xs)$

**H4-1** ***Typsignaturen*** **(0 Punkte)** (Abgabeformat: Text oder PDF)
Lösen Sie diese Aufgabe mit Papier und Bleistift! Geben Sie zu jeder der folgenden Deklaration eine möglichst allgemeine Signatur an. Begründen Sie Ihre Antwort informell!

Überlegen Sie sich dazu, welche Argumente jeweils auftreten und wie diese verwendet werden, z.B. welche Funktion auf welches Argument angewendet wird. Falls Sie eine der verwendeten Funktion nicht kennen, schlagen Sie diese in den Vorlesungsfolien oder der Dokumentation der Standardbibliothek nach. Verwenden Sie GHCI höchstens im Nachhinein, um Ihre Antwort zu kontrollieren. Numerische Typklassen müssen Sie zur Vereinfachung nicht angeben, verwenden Sie einfach einen konkreten Typ wie `Int` oder `Double`.

a) `gaa xy z vw = if (read z == xy) then minBound else vw`

*Lösungsbeispiel:* Wir sehen als erstes, dass es sich um eine Funktion mit drei Argumenten handelt, deren Typ wir bestimmen müssen, sowie den Ergebnistyp der Funktion. Nennen wir diesen Typ vorläufig einmal $a \to b \to c \to d$, also `xy` $:: a$, `z` $:: b$ und `vw` $:: c$.

Das Argument `xy` wird in einem Vergleich verwendet, also muss `Eq` $a$ sein.

`z` ist ein Argument für die Funktion `read` und damit ist $b =$ `String`. Da das Ergebnis aber mit `xy` verglichen wird, muss es den gleichen Typ haben, und wir wissen also, dass auch `Read` $a$ gelten muss, denn es wurde ja ein Wert dieses Typs geparsed.

`vw` wird nur als Ergebnis verwendet, daraus können wir $c = d$ schließen. Da die beiden Zweige des Konditionals den gleichen Typ haben müssen und im `then`-Zweig der Wert `minBound` zurückgegeben wird, muss dieser Typ auch noch in der Klasses `Bounded` sein.

Wir haben also `gaa :: (Eq a, Read a, Bounded c) => a -> String -> c -> c`

*Hinweis:* Namen von Typvariablen und Reihenfolge der Class Constraints ist unbedeutend. `gaa :: (Read y2, Bounded zz, Eq y2) => y2 -> [Char] -> zz -> zz` wäre z.B. eine äquivalente Typsignatur.

b)
```
axx u (v,w) = if minBound || w then succ v else u
```

**LÖSUNGSVORSCHLAG:**

```
axx :: Enum a => a -> (a, Bool) -> a
```

`minBound` ist hier uninteressant, da es lediglich ein boolsches Argument für den Oder-Operator `(||)` liefert. `u` und `v` müssen den gleichen Typ wie das Ergebnis haben, da diese zurückgegeben werden. Die Funktionsanwendung von `succ` auf `v` verändert dessen Typ nicht, erzwingt aber dessen Einschränkung auf die Typklasse `Enum`.

c)
```
guu _ (h1:h2:t) = [h2]
guu x _         = show x
```

**LÖSUNGSVORSCHLAG:**

```
guu :: Show a => a -> String -> String
```

Die zweite Zeile legt wegen `show :: Show a => a -> String` Rückgabewert auf `String`fest und schränkt den Typ des ersten Argumentes auf die Typklasse `Show` ein. Damit `[h2]` vom Typ `String` ist, muss `h2` vom Typ `Char` sein. `h2` ist aber das zweite Element der Liste, welche wir als zweites Argument erhalten haben, also ist diese eine Liste von Zeichen.

d)
```
foo a b []    e f = show a
foo a b (c:d) e f
  | a==c, read b  = e ++ e
  | a/=c          = show d
  | otherwise     = show e
```

**LÖSUNGSVORSCHLAG:**

```
foo :: (Eq x, Show x) => x -> String -> [x] -> String -> y -> String
```

Da auf das erste Argument `a` die Funktion `show` angewendet wird, muss es irgendein Typ sein, der in der Typklasse `Show` liegt. Da wir `show a` zurückgegeben, muss der Ergebnistyp der Funktion `foo` schon mal `String` sein.

Das zweite Argument muss wegen der Anwendung von `read` ein String sein. (Das Ergebnis dieses `read`-Aufrufs ist für uns hie unerheblich – wir wissen aber, dass es von dem Typ `Bool` sein muss.)

Das dritte Argument wird mit Listen-Pattern gemacht, es ist also irgendein Listentyp, sagen wir einfach mal `[x]`. Damit ist die Variable `c` vom Typ `x` und die Variable `d` vom Typ `[x]`. Aufgrund des Vergleiches des ersten Argumentes mit der Variable `c` in Guards wissen wir, das beide den gleichen Typ besitzen, welche zusätzlich noch in der Typklasse `Eq` liegen muss. (Zusätzlich zu der Klasse `Show` wie wir bereits für das erste Argument festgestellt hatten.)

Auf das vierte Argument wird die Infix-Funktion `(++)` angewendet. Diese hat den Typ `[a]->[a]->[a]`. Da wir bereits wissen, dass der Ergebnistyp `String`, also `[Char]` ist, so muss auch das vierte Argument vom Typ `String` sein.

Das fünfte Argument wird nirgendwo verwendet. Es kann daher irgendeinen beliebigen Typ haben, der nicht auf eine Typklasse eingeschränkt werden muss.

e)
```
bar a b c d
  | b >= c    = bar b a d c
  | otherwise = a==d
```

**LÖSUNGSVORSCHLAG:**

```
bar :: (Ord a, Ord b) => a -> b -> b -> a -> Bool
```

Im ersten Zweig wird die Ordnung des zweiten und dritten Argumentes geprüft. Damit müssen diese beide Argumente dem gleichen Typ angehören, welcher der Typklasse `Ord` angehören muss.

Der rekursive Aufruf verrät uns leider gar nichts über den Rückgabetyp der Funktion. Allerdings sehen wir für den rekursiven Aufruf das jeweils das erste und zweite, und das dritte und vierte Argument vertauscht werden. GHC schließt daraus schon, dass alle Typen gleich sein müssen, doch dass stimmt nicht: Der Aufruf kann ja auch polymorph sein - es reicht wenn das erste und vierte Argument einem anderen Typ angehören, der den gleichen Einschränkung unterliegt, in diesem Falle also auch der Typklasse `Ord` angehört.

Der zweite Zweig verrät uns den Rückgabetyp: `Bool`, da dies der Ergebnistyp der verwendeten Funktion `(==)` ist. Die explizite Einschränkung der Argumenttypen auf die Typklasse `Eq` können wir uns hier sparen, da `Ord` eine Unterklasse der Typklasse `Eq` ist - was wir ordnen können, können wir auch vergleichen.

*Hinweis:* Der rekursive Aufruf in `bar` benötigt eine andere Instantiierung des polymorphen Typs (die Argumente tauschen Ihre Positionen, d.h. Typen a und b vertauschen sich bei jedem rekursiven Aufruf). Die nennt man *polymorphe Rekursion.*

Es ist bewiesen, dass Typinferenz für polymorph rekursive Funktionen unentscheidbar ist. GHC kann den oben angegebenen Typ daher nicht inferieren. Allerdings kann GHC eine Typsignatur für eine polymorph rekursive Funktion sehr wohl überprüfen! D.h. wenn wir die Typsignatur explizit im Quellcode angeben, dann prüft und akzeptiert GHC diese auch.

**H4-2 *Fehlerhafte Induktion* (2 Punkte)** (Abgabeformat: Text oder PDF)
Durch vollständige Induktion wollen wir zeigen:

*Alle Smartphones benutzen das gleiche Betriebssystem.*

Um Induktion einsetzen zu können, verallgemeinern die Aussage zu *"In einer Menge von $n$ Smartphones benutzen alle das gleiche Betriebssystem."* Da die Anzahl aller Smartphones in der Welt ist eine natürliche Zahl ist, reicht dies für die ursprüngliche Aussage.

Der Induktionsanfang ist klar: Ein Smartphone benutzt das gleiche Betriebssystem wie es selbst.

Angenommen es gäbe $n+1$ Smartphones in der Welt. Wählen wir irgendein Smartphone davon aus und bezeichnen es mit $s$. Die übrigen Smartphones bilden eine Menge von $n$ Smartphones. Nach Induktionsvoraussetzung benutzen diese $n$ Smartphones alle das gleiche Betriebssystem. Nun entfernt man von diesen $n$ Smartphones mit gleichem Betriebssystem eines und fügt $s$ wieder dazu. Die vorliegenden $n$ Smartphones benutzen nach Induktionsvoraussetzung wieder alle das gleiche Betriebssystem. Damit benutzen aber offenbar alle $n+1$ Smartphones das gleiche Betriebssystem und die Behauptung ist bewiesen.

Wo liegt der Fehler? Begründen Sie Ihre Antwort!

**LÖSUNGSVORSCHLAG:** Die Verallgemeinerung vor dem eigentlichen Beweis ist unproblematisch. In der Argumentation wird die Induktionsvoraussetzung korrekt verwendet. Jedoch wird beim zweiten Aussondern implizit davon ausgegangen, dass die Menge der verbleibenden $n-1$ Smartphones nicht leer ist. (Dies ist die Schnittmenge der beiden Mengen, für welche die Induktionsvoraussetzung verwendet wird.) Doch ein beliebiges Smartphone aus dieser Menge wird benötigt, um zu zeigen, dass $n+1$ Smartphones das gleiche Betriebssystem benutzen. Das Problem liegt also beim Fall $n+1=2$, also $n=1$.

**H4-3 *Polynome* (6 Punkte)** (`.hs`-Datei als Lösung abgeben)
In einer anderen wichtigen Informatik Vorlesung müssen wir bald mit Polynomen rechnen. Da wir bedauerlicherweise schon vieles in der Schule dazu Gelerntes wieder bereits vergessen haben, wollen wir einen passenden Datentyp für Polynome in Haskell schreiben, mit dem wir später unsere Polynomrechnungen mit Haskell durchführen können – nur zur Kontrolle natürlich!

Wir haben uns entschieden, unsere Polynome als eine Liste von Fließkommazahlen zu implementieren. Beispiele:

$$[] = 0.0 \qquad [1.0] = 1.0 \qquad [3,2,1,0,0] = x^2 + 2x + 3$$
$$[0.0] = 0.0 \qquad [-5,22] = 22x - 5 \qquad [5,0,3,0,1] = x^4 + 3x^2 + 5$$

Allgemein ordnen wird der Liste $[a_1, \ldots, a_n]$ also das Polynom $\sum_{i=0}^{n} a_i \cdot x^i$ zu.

Diese Repräsentation hat den Vorteil, dass Sie leicht nach dem so genannten Horner-Schema berechnet werden kann:

$$x \quad \mapsto \quad (a_0 + x \cdot (a_1 + x \cdot (a_2 + x \cdot \ldots (a_{n-1} + x \cdot (a_n + x \cdot 0)) \ldots)))$$

*Hinweis:* Sie dürfen zur Lösung alle Funktionen des Moduls `Prelude` der Standardbibliothek benutzen, wenn Sie möchten, aber keine anderen Haskell Module verwenden.

a) Deklarieren Sie den angegebenen Datentyp `Polynom`

**LÖSUNGSVORSCHLAG:**

```
data Polynom = Polynom [Double]
```

b) Implementieren Sie `berechnePolynom :: Double -> Polynom -> Double`, welche den Wert eines gegebenes Polynoms an einer angegebenen Stelle nach dem oben beschriebenen Horner-Schema berechnet.

**LÖSUNGSVORSCHLAG:**

```
evalPolynom :: Double -> Polynom -> Double
evalPolynom _ (Polynom []) = 0
evalPolynom x (Polynom cs) = evalAux cs
 where
  evalAux :: [Double] -> Double
  evalAux   []  = 0
  evalAux (h:t) = h + x * evalAux t
```

c) Machen Sie den Typ `Polynom` zu einer Instanz der Typklasse `Ord`. Wir sortieren Polynome zuerst nach dem höchsten Grad. Bei gleichem Grad vergleichen wir den Koeffizienten des höchstens Gerades, sind auch diese gleich, so prüfen wir kontinuierlich den nächstniederen Koeffizienten.
*Beispiele:*

$$2x^2 > x^2 + 7x + 3 \qquad x^3 + x^2 > x^3 + 9x + 9 \qquad x^2 + x + 1 > x^2 + x - 2$$

*Hinweis:* Achten Sie dabei auf alle möglichen Randfälle! Ihre Funktion muss alle Typkorrekten Eingaben richtig verarbeiten.

**LÖSUNGSVORSCHLAG:**

Im wesentlichen ein Rehash von A4-2:

```
instance Ord Polynom where
  compare (Polynom [])     (Polynom [])     = EQ
  compare (Polynom [])     (Polynom (y:ys))
    | y /= 0    = LT
    | otherwise = compare (Polynom []) (Polynom ys)
  compare (Polynom (x:xs)) (Polynom [])
    | x /= 0    = GT
    | otherwise = compare (Polynom xs) (Polynom [])
  compare (Polynom (x:xs)) (Polynom (y:ys))
    | compTail == EQ = compare x y
    | otherwise      = compTail
    where
      compTail = compare (Polynom xs) (Polynom ys)

instance Eq Polynom where
  p1 == p2 = compare p1 p2 == EQ
```

Wahlweise könnte man natürlich auch `(<=)` implementieren. Dabei muss man aber aufpassen, da die Antwort `True` sowohl Gleichheit als auch echt kleiner bedeuten kann. Auch `(<)`,`(>=)`,`(>)` und `compare` sollte man bei der Definition von `(<=)` nicht verwenden, da diese ja auf `(<=)` zurückgeführt werden und es so schnell zu subtilen Fehlern kommen kann! Z.B. wenn man in der folgenden Lösung am Ende der ersten Zeile `(>=)` anstatt `not` und `(<=)` verwenden würde, so funktioniert vieles, aber fälschlicherweise `> Polynom [2,2] <= Polynom [1,1] == True`

```
instance Ord Polynom where
  (Polynom (x:xs)) <= (Polynom (y:ys)) =
  (Polynom xs) <= (Polynom ys) && ( x<=y || (not $ (Polynom ys) <= (Polynom xs))) -- >=
  (Polynom    [] ) <= (Polynom (y:ys)) = y /= 0 || (Polynom []) <= (Polynom ys)
  (Polynom (x:xs)) <= (Polynom    [] ) = x == 0 && (Polynom xs) <= (Polynom [])
  (Polynom    [] ) <= (Polynom    [] ) = True
```

d) *Empfehlenswerte unbenotete Zusatzaufgabe:* Machen Sie den Typ `Polynom` zu einer Instanz der Typklasse `Show`. Achten Sie dabei auf eine hübsche, lesbare Ausgabe, d.h. die höchste Potenz wird zuerst ausgegeben, Potenzen mit Koeffizient 0 werden ausgelassen, bei negativen Koeffizienten wird anstatt `"1x^4 + -3x^3"` ordentlich `"x^4 - 3x^3"` ausgegeben, usw.

**LÖSUNGSVORSCHLAG:**

```
instance Show Polynom where
  show (Polynom fs) = sp_aux fs 0 ""
    where
      sp_aux :: [Double] -> Int -> String -> String
      sp_aux    []  _       []  = "0"
      sp_aux    []  d (' ':acc) = sp_aux []  d    acc
      sp_aux    []  d ('+':acc) = sp_aux []  d    acc
      sp_aux    []  _      acc  =                 acc
      sp_aux (x:xs) d      acc  = sp_aux xs (d+1) (coeff x d ++ acc)

      coeff :: Double -> Int -> String
      coeff x d
        | x ==   1 , d /= 0 = " +" ++            x2d
        | x == (-1), d /= 0 = " -" ++            x2d
        | x >  0            = " +" ++ (show x) ++ x2d
        | x <  0            = " "  ++ (show x) ++ x2d
        | otherwise         = ""
        where
          x2d | d == 0    = ""
              | d == 1    = "x"
              | otherwise = "x^" ++ (show d)
```

**Abgabe:** Lösungen zu den Hausaufgaben können bis Dienstag, den 20.05.2014, 11:00 Uhr mit UniworX abgegeben werden. Die Hausaufgaben müssen von Ihnen alleine gelöst werden; Abschreiben wird als Betrug gewertet und ans Prüfungsamt gemeldet.