RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK II
RWTH Aachen · D-52056 Aachen · GERMANY
http://www-i2.informatik.rwth-aachen.de/lufgi2
Prof. Dr. Jürgen Giesl

Prof. Dr. Jürgen Giesl
Carsten Kern, Peter Schneider-Kamp, René Thiemann

Diplomvorprüfung / Zwischenprüfung Informatik I - Programmierung

1. 3. 2006

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte ankreuzen):

- ○ Informatik Diplom ○ Informatik Lehramt
- ○ Sonstige: _____

- Schreiben Sie bitte auf jedes Blatt **Vorname**, **Name** und **Matrikelnummer**.
- Geben Sie Ihre Antworten bitte in lesbarer und verständlicher Form an. Schreiben Sie bitte nicht mit roten Stiften oder Bleistiften.
- Bitte beantworten Sie die Aufgaben auf den **Aufgabenblättern**. Benutzen Sie ggf. auch die Rückseiten der **zur jeweiligen Aufgabe gehörenden** Aufgabenblätter.
- Antworten auf anderen Blättern können nur berücksichtigt werden, wenn **Name, Matrikelnummer und Aufgabennummer** deutlich darauf erkennbar sind.
- Was nicht bewertet werden soll, kennzeichnen Sie bitte durch **Durchstreichen**.
- Werden Täuschungsversuche beobachtet, so wird die Klausur mit **nicht bestanden** bewertet.
- Geben Sie bitte am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	14	
Aufgabe 2	12	
Aufgabe 3	14	
Aufgabe 4	26	
Aufgabe 5	17	
Aufgabe 6	17	
Summe	100	
Prozentzahl		

Vorname	Name	Matr.-Nr.

Aufgabe 1 (Programmanalyse, 8 + 6 Punkte)

a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Schreiben Sie hierzu jeweils die ausgegebenen Zeichen hinter den Kommentar "`OUT:`".

```
public class A {
    public static int x = 1;
    public A(int x) {
        A.x += x;
    }
    public int f(int x) {
        x++;
        return x;
    }
}

public class B extends A {
    public int y = 2;
    public B(int y) {
        super(y+2);
    }
    public int f(int x) {
        y += x;
        return y;
    }
}

public class M {
    public static void main(String[] args) {
        A a = new A(2);
        System.out.println(A.x);          // OUT: 3
        int z = a.f(3);
        System.out.println(z+" "+A.x);    // OUT: 4 3
        B b = new B(5);
        System.out.println(B.x+" "+b.y); // OUT: 10 2
        z = b.f(6);
        System.out.println(z+" "+b.y);    // OUT: 8 8
        a = b;
        z = a.f(7);
        System.out.println(z);            // OUT: 15
    }
}
```

Vorname	Name	Matr.-Nr.

3

b) Wir schreiben zusätzlich zu `A` und `B` eine neue Klasse `C`. Welche drei Fehler treten beim Compilieren auf? Begründen Sie Ihre Antwort kurz.

```
public class C extends B {
    final static int z = 9;
    public C() {
        super(z,z);
        z = 10;
    }
    public double f(int z) {
        return x;
    }
}
```

- *Die Anweisung* `super(z,z);` *im Konstruktor* `C()` *ist nicht erlaubt, da es in der direkten Oberklasse* `B` *keinen zweistelligen Konstruktor mit zwei Argumenten vom Typ* `int` *gibt.*
- *Die Anweisung* `z = 10;` *im Konstruktor* `C()` *ist nicht erlaubt, da* `z` *als* `final` *deklariert wurde und sein Wert somit nicht mehr verändert werden darf.*
- *Das Überschreiben der Methode* `f` *aus der Oberklasse* `B` *durch die Methode* `public double f(int z)` *in der Klasse* `C` *ist nicht erlaubt, da der Rückgabetyp von* `f` *in* `B` *der Typ* `int` *und nicht* `double` *ist.*

Vorname	Name	Matr.-Nr.

Aufgabe 2 (Verifikation, 10 + 2 Punkte**)**

Der Algorithmus P berechnet für zwei Zahlen $a \in \mathbb{R}$ und $n \in \mathbb{N}$ den Wert a^{2^n} (d.h. $a^{(2^n)}$).

a) Vervollständigen Sie die folgende Verifikation des Algorithmus P im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Zwei Zusicherungen dürfen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt. Sie dürfen auch weitere zusätzliche Zusicherungen einfügen.

Algorithmus: P
Eingabe: $n \in \mathbb{N}$, $a \in \mathbb{R}$
Ausgabe: res
Vorbedingung: $n \geq 0$
Nachbedingung: $res = a^{2^n}$

$\langle\ n \geq 0\ \rangle$

$\langle\ n \geq 0 \wedge a = a^{2^{n-n}}\ \rangle$

```
res = a;
```

$\langle\ n \geq 0 \wedge res = a^{2^{n-n}}\ \rangle$

```
i = n;
```

$\langle\ i \geq 0 \wedge res = a^{2^{n-i}}\ \rangle$

```
while (i > 0) {
```

$\langle\ i > 0 \wedge i \geq 0 \wedge res = a^{2^{n-i}}\ \rangle$

$\langle\ i - 1 \geq 0 \wedge res * res = a^{2^{n-(i-1)}}\ \rangle$

```
  res = res * res;
```

$\langle\ i - 1 \geq 0 \wedge res = a^{2^{n-(i-1)}}\ \rangle$

```
  i = i - 1;
```

$\langle\ i \geq 0 \wedge res = a^{2^{n-i}}\ \rangle$

```
}
```

$\langle\ i \not> 0 \wedge i \geq 0 \wedge res = a^{2^{n-i}}\ \rangle$

$\langle\ res = a^{2^n}\ \rangle$

Vorname	Name	Matr.-Nr.

5

b) Beweisen Sie die Terminierung des Algorithmus P.

Wir wählen die Variante i. Dann gilt $i > 0 \Rightarrow i \geq 0$ und

$$\langle\ i = m \wedge i > 0\ \rangle$$

$$\langle\ i - 1 < m\ \rangle$$

```
res = res * res;
```

$$\langle\ i - 1 < m\ \rangle$$

```
i = i - 1;
```

$$\langle\ i < m\ \rangle$$

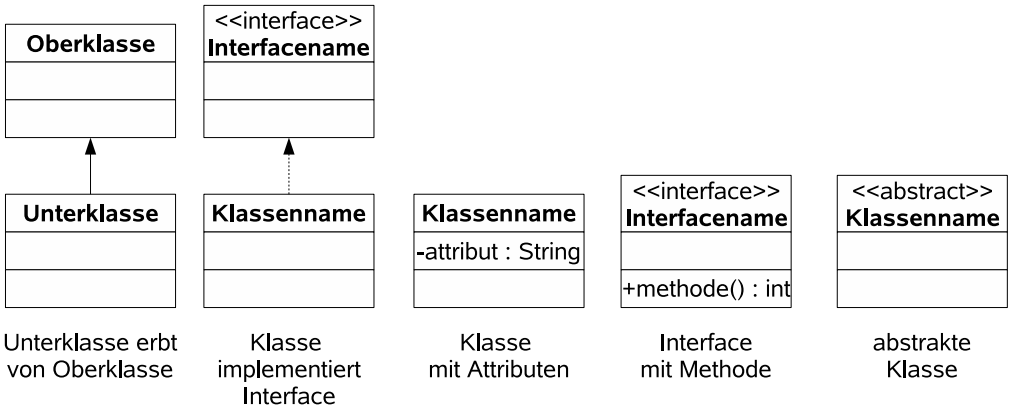
Vorname	Name	Matr.-Nr.

Aufgabe 3 (Datenstrukturen in Java, 6 + 8 Punkte**)**

Ihre Aufgabe ist es, eine objektorientierte Datenstruktur zur Verwaltung von Pflanzen zu entwerfen. Bei der vorangehenden Analyse wurden folgende Eigenschaften der verschiedenen Pflanzen ermittelt.

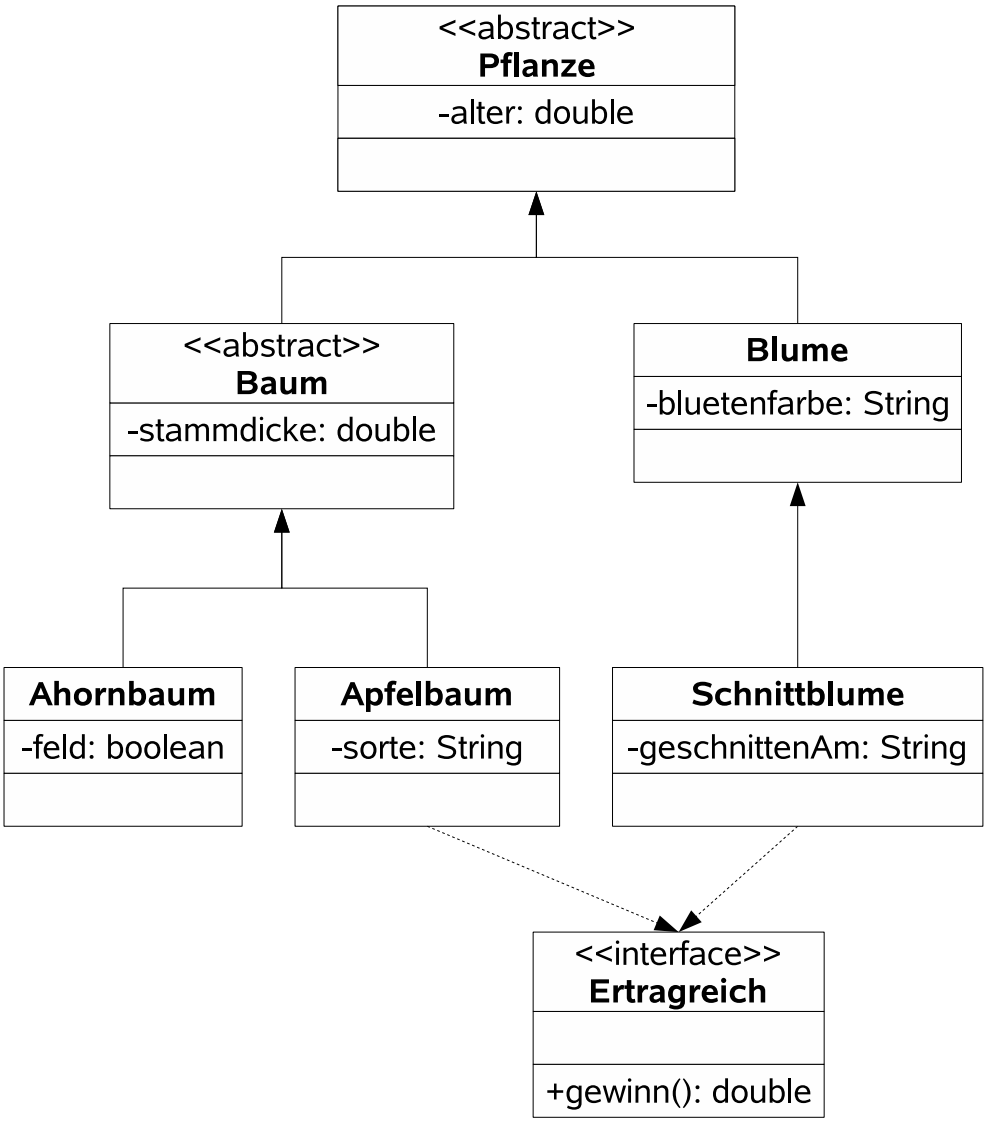
- Ein Apfelbaum ist eine Pflanze. Jeder Apfelbaum hat ein bestimmtes Alter und eine Stammdicke. Zudem gibt es zu jedem Apfelbaum eine Sorte von Äpfeln, die an ihm wachsen.
- Ahornbäume kennzeichnen sich durch ihre Stammdicke und ihr Alter. Man unterscheidet zwischen Feld- und Spitzahornbäumen. Natürlich ist ein Ahornbaum auch eine Pflanze.
- Eine Blume ist eine Pflanze und hat ein bestimmtes Alter. Bei Blumen ist vor allem die Blütenfarbe interessant.
- Schnittblumen sind Pflanzen mit einer bestimmten Blütenfarbe. Neben dem Alter der Schnittblume ist auch die Dauer seit dem Abschneiden interessant.
- Apfelbäume und Schnittblumen sind ertragreiche Pflanzen. Für diese kann man berechnen, wieviel Gewinn man erzielt (durch den Verkauf der jeweiligen Äpfel eines Apfelbaums oder durch den Verkauf der Schnittblume).

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Pflanzen. Achten Sie darauf, dass gemeinsame Merkmale in (evtl. abstrakten) Oberklassen zusammengefasst werden. Notieren Sie Ihren Entwurf graphisch und verwenden Sie dazu die folgende Notation:



Geben Sie für jede Klasse ausschließlich den jeweiligen Namen und die Namen ihrer Attribute an. Methoden von Klassen müssen nicht angegeben werden. Geben Sie für jedes Interface ausschließlich den jeweiligen Namen sowie die Namen seiner Methoden an.

Vorname	Name	Matr.-Nr.



Vorname	Name	Matr.-Nr.

b) Implementieren Sie in Java eine Methode `schlechterBaum`. Die Methode bekommt als Parameter ein Array von ertragreichen Pflanzen übergeben. Sie soll den ersten Apfelbaum liefern, der einen geringeren Gewinn hat als der durchschnittliche Gewinn der ertragreichen Pflanzen im Array. Ansonsten soll die Methode `null` zurückliefern.

Gehen Sie dabei davon aus, dass das übergebene Array nicht der `null`-Wert ist und dass es keine `null`-Werte enthält. Kennzeichnen Sie die Methode mit dem Schlüsselwort "`static`", falls angebracht.

```
public static Apfelbaum schlechterBaum(Ertragreich[] array) {
  double durchschnitt = 0.0;
  int anzahl = array.length;

  if (anzahl == 0) return null;

  for (int i=0; i<anzahl; i++) {
    durchschnitt += array[i].gewinn();
  }
  durchschnitt /= anzahl;

  for (int i=0; i<anzahl; i++) {
    if (array[i] instanceof Apfelbaum &&
        array[i].gewinn() < durchschnitt) {
      return (Apfelbaum) array[i];
    }
  }

  return null;
}
```

Vorname	Name	Matr.-Nr.

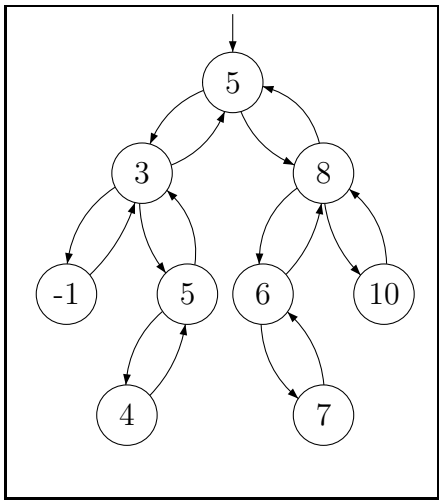
Aufgabe 4 (Programmierung in Java, 7 + 7 + 12 Punkte)

Gegeben ist das folgende Interface.

```
public interface Vergleichbar {
    public boolean groesser(Vergleichbar other);
}
```

Für zwei Objekte `s` und `t` vom Typ `Vergleichbar` liefert `s.groesser(t)` den Wert `true` zurück, falls `s` größer als `t` ist und ansonsten liefert `s.groesser(t)` den Wert `false`.

In dieser Aufgabe betrachten wir einen doppelt verketteten sortierten Binärbaum. Das folgende Bild zeigt einen solchen Baum schematisch:



Ein solcher Baum heißt *sortiert*, wenn die folgenden Bedingungen erfüllt sind:

- jeder Knoten enthält einen Wert `wert` vom Typ `Vergleichbar`
- falls ein Knoten einen linken Unterbaum hat, so sind alle Werte im linken Unterbaum kleiner oder gleich `wert`
- falls ein Knoten einen rechten Unterbaum hat, so sind alle Werte im rechten Unterbaum größer als `wert`

Die Datenstruktur heißt doppelt verkettet, da jeder Knoten sowohl Verweise auf seine Nachfolger (`links` und `rechts`) als auch auf seinen Vorgänger (`vater`) besitzt.

Die Datenstruktur sei folgendermaßen in Java implementiert:

```
public class Baum {
    private Knoten wurzel;

    ...
}
```

Vorname	Name	Matr.-Nr.

10

```
public class Knoten {
    Vergleichbar wert;
    Knoten links, rechts, vater;

    public Knoten (Vergleichbar wert) {
        this.wert = wert;
    }
}
```

Implementieren Sie die folgenden Methoden in der Klasse `Baum`. Dabei dürfen Sie direkt (ohne Verwendung von Selektoren) auf die Attribute aus der Klasse `Knoten` zugreifen. Selbstverständlich dürfen Sie auch weitere benötigte Hilfsmethoden implementieren. Verwenden Sie die Schlüsselworte `public` und `private` auf sinnvolle Weise und kennzeichnen Sie Methoden als `static`, falls angebracht.

a) Implementieren Sie eine Methode `public int hoehe()`, die die Höhe des Binärbaums berechnet. Die *Höhe* eines Binärbaums sei dabei wie folgt definiert:

- Der leere Binärbaum besitzt die Höhe 0.
- Der Binärbaum, der nur einen Knoten besitzt (an der Wurzel), hat die Höhe 1.
- Jeder andere Binärbaum besitzt eine Höhe von $1 + \max$(Höhe des linken Unterbaums, Höhe des rechten Unterbaums).

Der Binärbaum im Beispiel hat also die Höhe 4.

Beachten Sie, dass bei der Implementierung dieser Methode keine Schleifen verwendet werden dürfen. Sie dürfen aber Rekursion benutzen.

b) Implementieren Sie eine Methode `public boolean balanciert()`, die berechnet, ob der Binärbaum balanciert ist. Ein Binärbaum heißt *balanciert*, wenn für jeden seiner Knoten gilt, dass sich die Höhen der Unterbäume dieses Knotens maximal um 1 unterscheiden. Sie können hier die Funktion `public static int abs (int zahl)` der Klasse `Math` verwenden, die den absoluten Betrag einer `int`-Zahl `zahl` zurückliefert.

c) Implementieren Sie eine Methode `public void einfuegen(Vergleichbar w)`, die einen gegebenen Wert `w` in den Binärbaum einfügt. Gehen Sie hierbei davon aus, dass `w` nicht `null` ist und dass auch in jedem Knoten des Baums der jeweilige `wert` nicht `null` ist. Der Binärbaum muss nach diesem Einfügen wieder *sortiert* sein. Achten Sie darauf, dass auch die `vater`-Verweise auf die jeweiligen Vorgänger entsprechend aktualisiert werden.

Vorname	Name	Matr.-Nr.


```
private static int hoehe(Knoten current){
    if    (current == null)
          return 0; // der leere Baum hat die Hoehe 0
    else {int hoeheLinks = hoehe(current.links);
          int hoeheRechts = hoehe(current.rechts);
          return 1 + (hoeheLinks >= hoeheRechts ? hoeheLinks : hoeheRechts);
    }
}

public int hoehe(){
    return hoehe(wurzel);
}

private static boolean balanciert(Knoten current){
    if    (current == null)
          return true; //der leere Baum ist per Definition balanciert
    else  return balanciert(current.links) &&
                 balanciert(current.rechts) &&
                 Math.abs(hoehe(current.links) - hoehe(current.rechts)) <= 1;
}


public boolean balanciert(){
    return balanciert(wurzel);
}

private static void einfuegen(Knoten current, Vergleichbar w){
    boolean groesser = w.groesser(current.wert);
    Knoten kind = groesser ? current.rechts : current.links;
                 //Der Wert w muss in den Teilbaum eingefuegt werden,
                 //der mit kind beginnt.
    if   (kind != null)
         einfuegen(kind,w);
    else {kind = new Knoten(w);
          kind.vater = current;
          if  (groesser)
               current.rechts = kind;
          else current.links = kind;
    }
}

public void einfuegen(Vergleichbar w){
    if   (wurzel == null)
         wurzel = new Knoten(w);
    else einfuegen(wurzel, w);
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 5 (Funktionale Programmierung in Haskell, $2 + 2 + 3 + 2 + 3 + 5$ Punkte)

a) Geben Sie den allgemeinsten Typ der Funktionen `f` und `g` an. Gehen Sie dabei davon aus, dass die Funktion `+` den Typ `Int -> Int -> Int` hat.

```
f x y = (x+y):[x]
```

```
f :: Int -> Int -> [Int]
```

```
g x = \y -> x:y
```

```
g :: a -> [a] -> [a]
```

b) Bestimmen Sie das Ergebnis der Auswertung für die folgenden Ausdrücke:

```
(\x y -> (x+1):(y++[1])) 2 [3,4]
```

Der Ausdruck wertet zu `[3,3,4,1]` *aus.*

```
map (\x -> x + 3) (filter (\x -> x < 3) [3,1,4,2,5])
```

Der Ausdruck wertet zu `[4,5]` *aus.*

Vorname	Name	Matr.-Nr.

c) Eine Matrix kann als Liste von Einträgen dargestellt werden. Jeder Eintrag besteht aus einem Zeilenindex, einem Spaltenindex und dem jeweiligen Element. Als Beispiel betrachten wir die folgende Matrix:

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 4.0 & 5.0 & 6.0 \end{pmatrix}$$

Matrizen sollen als Objekte vom Typ `[(Int, Int, a)]` repräsentiert werden. Die obige Matrix kann z.B. durch das folgende Objekt `m` des Typs `[(Int, Int, Float)]` dargestellt werden:

```
[(1,1,1.0),(1,2,2.0),(1,3,3.0),(2,1,4.0),(2,2,5.0),(2,3,6.0)]
```

Hier bedeutet der Eintrag `(2,1,4.0)`, dass in der zweiten Zeile in der ersten Spalte das Element `4.0` steht.

Schreiben Sie eine Funktion `transpose`, die eine Matrix mit Elementen eines beliebigen Typs `a` transponiert. In unserem Beispiel ergibt sich die folgende transponierte Matrix:

$$\begin{pmatrix} 1.0 & 4.0 \\ 2.0 & 5.0 \\ 3.0 & 6.0 \end{pmatrix}$$

Geben Sie neben den Funktionsdeklarationen auch die Typdeklaration von `transpose` an.

```
transpose :: [(Int, Int, a)] -> [(Int, Int, a)]
transpose [] = []
transpose ((i,j,x):xs) = (j,i,x):transpose xs
```

Alternative Lösung:

```
transpose :: [(Int, Int, a)] -> [(Int, Int, a)]
transpose = map (\(i,j,x) -> (j,i,x))
```

d) Definieren Sie eine eigene Datenstruktur `Matrix`, um Matrizen ähnlich wie in Aufgabenteil (c) darzustellen. Dabei dürfen Sie keine vordefinierten Listen oder Tupel verwenden. Die Matrizen sollen Elemente beliebigen Typs enthalten können.

```
data Matrix a = End | Entry Int Int a (Matrix a)
```

Vorname	Name	Matr.-Nr.

e) Schreiben Sie eine Funktion `spur`, die die *Spur* einer Matrix berechnet. Die *Spur* einer Matrix ist definiert als die Summe der Elemente der Diagonalen. Für die Beispielmatrix aus (c) ist die Spur also $1.0 + 5.0 = 6.0$. Geben Sie neben den Funktionsdeklarationen auch die Typdeklaration von `spur` an. Verwenden Sie hierbei Ihre Datenstruktur aus Aufgabenteil (d) und gehen Sie davon aus, dass die Elemente der Matrix vom Typ `Float` sind.

```
spur :: Matrix Float -> Float
spur End = 0
spur (Entry i j x xs) | i == j = x + spur xs
                      | True   = spur xs
```

Alternative Lösung:

```
spur :: Matrix Float -> Float
spur End = 0
spur (Entry i j x xs) = (if i == j then (x +) else (\y -> y)) (spur xs)
```

f) Schreiben Sie eine Funktion `isDeterministic`, die überprüft, ob eine Matrix mehrere Einträge für die gleiche Position (mit gleicher Zeilen- und Spaltennummer) besitzt. Geben Sie neben den Funktionsdeklarationen auch die Typdeklaration von `isDeterministic` und von evtl. benötigten Hilfsfunktionen an. Die Matrizen sollten wieder Elemente beliebigen Typs enthalten. Zur Darstellung von Matrizen können Sie wahlweise die Datenstruktur aus Teil (c) oder Teil (d) verwenden.

Falls die Datenstruktur aus Teil (c) verwendet wird, so liefert `isDeterministic` für die Beispielmatrix `m` aus Teil (c) das Resultat `True`. Für die Matrix `[(1,1,42),(1,1,42)]` liefert `isDeterministic` hingegen `False`, da es zwei Einträge für die erste Spalte der ersten Zeile gibt.

```
isDeterministic :: [(Int,Int,a)] -> Bool
isDeterministic [] = True
isDeterministic ((i,j,_) : xs) = (check i j xs) && (isDeterministic xs)
  where check :: Int -> Int -> [(Int,Int,a)] -> Bool
        check i j [] = True
        check i j ((i',j',_):xs) = (i /= i' || j /= j') && check i j xs
```

Alternative Lösung:

```
isDeterministic ::  [(Int,Int,a)] -> Bool
isDeterministic [] = True
isDeterministic ((i,j,_) : xs) =
                     isEmpty (filter (\(i',j',_) -> i == i' && j == j') xs)
                     && isDeterministic xs
  where isEmpty :: [b] -> Bool
        isEmpty [] = True
        isEmpty _  = False
```

Vorname	Name	Matr.-Nr.

Aufgabe 6 (Logische Programmierung in Prolog, $6 + 5 + 6$ Punkte**)**

a) Sie dürfen in der gesamten Teilaufgabe keine vordefinierten Prädikate außer `>`, `<`, `>=` und `=<` verwenden.

- Definieren Sie in Prolog ein zweistelliges Prädikat `praefix`. Für zwei Listen `Xs` und `Ys` soll `praefix(Xs,Ys)` genau dann erfüllt sein, wenn `Xs` ein Präfix von `Ys` ist. Beispielsweise gilt: `praefix([1,2],[1,2,3,4]).`

```
praefix([], L).
praefix([X|Xs], [X|Ys]) :- praefix(Xs, Ys).
```

- Definieren Sie in Prolog ein zweistelliges Prädikat `max`, das zu einer nicht-leeren Liste natürlicher Zahlen das Maximum bestimmt. Beispielsweise gilt: `max([3,7,1],7).`

```
max([X], X).
max([X,Y|Xs], Z) :- X>=Y, max([X|Xs], Z).
max([X,Y|Xs], Z) :- X<Y, max([Y|Xs], Z).
```

Vorname	Name	Matr.-Nr.

b) Geben Sie den allgemeinsten Unifikator für folgende Termpaare an, oder begründen Sie, warum er nicht existiert.

- `f(Y, g(f(X,a), Z))` und `f(Z, g(Z, f(X,X)))`

σ_1=`MGU(Y, Z)`={`Y=Z`}
σ_2=`MGU(g(f(X,a), Z), g(Z, f(X,X)))`={`Z=f(a,a), X=a`}, *denn*
σ_{21}=`MGU(f(X,a), Z)`={`Z=f(X,a)`}
σ_{22}=`MGU(f(X,a), f(X,X))`={`X=a`}, *denn*
σ_{221}=`MGU(X,X)`={ }
σ_{222}=`MGU(a,X)`={`X=a`}

Die beiden Terme sind unifizierbar mit MGU $\sigma = \sigma_2 \circ \sigma_1$ = {`Z=f(a,a), X=a, Y=f(a,a)`}

- `f(g(X), h(Y))` und `f(Y, h(X))`

σ_1=`MGU(g(X), Y)`={`Y=g(X)`}
σ_2=`MGU(h(g(X)), h(X))`
σ_{21}=`MGU(g(X), X)` $\Longrightarrow$ *occur failure*

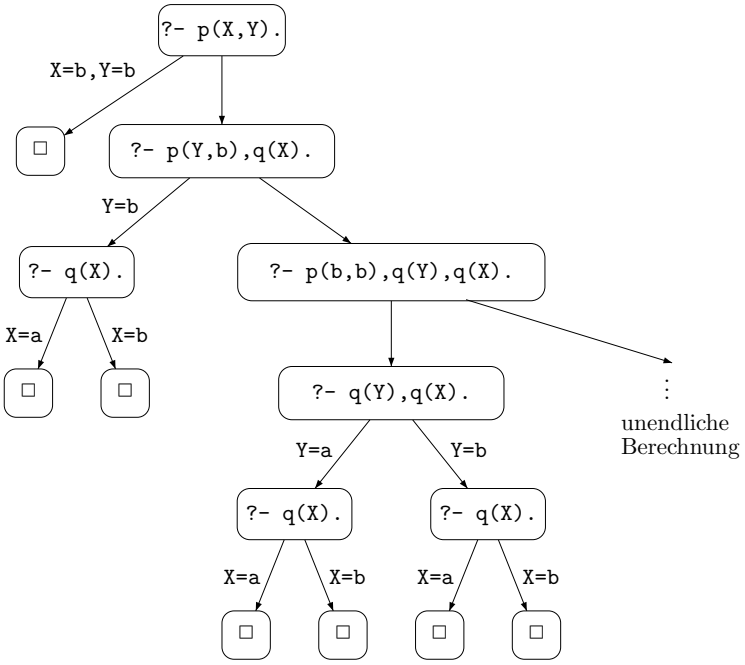
Die beiden Terme sind aufgrund eines occur failure *nicht unifizierbar.*

Vorname	Name	Matr.-Nr.

c) Gegeben sei das folgende Prolog-Programm:

```
p(b,b).
q(a).
q(b).
p(X,Y) :- p(Y,b), q(X).
```

- Stellen Sie den Beweisbaum für die Anfrage "`?- p(X,Y).`" graphisch dar. Brechen Sie Ihre Zeichnung ab, sobald Anfragen mit mehr als 3 Atomen zu beweisen sind.



- Wieviele (nicht unbedingt verschiedene) Lösungen findet Prolog (d.h. wie oft muss man bei der Anfrage "`?- p(X,Y).`" ein "`;`" eingeben, bis das Programm terminiert)?

Man kann unendlich oft "`;`" eingeben und Prolog findet immer wieder Lösungen. Das Programm terminiert also nicht.