RHEINISCH-
WESTFÄLISCHE
TECHNISCHE
HOCHSCHULE
AACHEN

LEHR- UND FORSCHUNGSGEBIET INFORMATIK 2
RWTH Aachen · D-52056 Aachen · GERMANY
http://programmierung.informatik.rwth-aachen.de/
Prof. Dr. Jürgen Giesl

Prof. Dr. Jürgen Giesl
Fabian Emmes, Carsten Fuhs, Carsten Otto, Stephan Swiderski

Prüfungsklausur Programmierung

25. 2. 2009

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte ankreuzen):

- ○ Informatik Bachelor ○ Informatik Diplom ○ Informatik Lehramt
- ○ Mathematik Bachelor
- ○ 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 mit 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 **0 Punkten** 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	12	
Aufgabe 4	28	
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 in die Kästchen hinter den Kommentar „`OUT:`“.

```
public class A {
    public int x = 3;
    public static int y = 1;
    public A() {
        int x = 1;
        y *= 2;
    }
    public A f(int y) {
        x += 1;
        y *= 2;
        return this;
    }
}

public class B extends A {
    public int x = 8;
    public B() {
        x++;
    }
    public A f(int y) {
        super.x += y;
        return (A) this;
    }
}

public class M {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x + " " + a.y);  // OUT: [   3] [   2]
        a = a.f(4);
        System.out.println(a.x + " " + a.y);  // OUT: [   4] [   2]
        B b = new B();
        System.out.println(b.x + " " +
                    ((A) b).x + " " + a.y);  // OUT: [   9] [   3] [   4]
        a = b.f(16);
        System.out.println(a.x);              // OUT: [  19]
    }
}
```

Vorname	Name	Matr.-Nr.

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 A {
    final int x = 0;

    private C(int x) {
        this();
    }

    A f(int x) {
        return new A();
    }

    public A f(double x) {
        String y = "Viel Erfolg!";
        y += x;
        this.x = (int) x;
        return this;
    }
}
```

- *Der Konstruktoraufruf* `this()` *ist falsch, da in der Klasse* `C` *kein Konstruktor ohne Argumente existiert.*
- *Die Methode* `f(int x)` *ändert die Sichtbarkeit von* `public` *zu der restriktiveren Package-Sichtbarkeit (ohne Schlüsselwort).*
- *Der Zugriff* `this.x = (int) x` *schreibt auf das Attribut* `x`*, was als* `final` *deklariert ist und deshalb nicht mehr verändert werden darf.*

Vorname	Name	Matr.-Nr.

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

Der Algorithmus P berechnet für eine natürliche Zahl $n \in \mathbb{N}$ den Wert $3n$.

a) Vervollständigen Sie die folgende Verifikation des Algorithmus P im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen 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.

Algorithmus: P
Eingabe: $n \in \mathbb{N}$
Ausgabe: res
Vorbedingung: $n \geq 0$
Nachbedingung: $res = 3n$

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

$\langle\ n \geq 0 \wedge n = n\ \rangle$

$k = n;$

$\langle\ k \geq 0 \wedge k = n\ \rangle$

$\langle\ k \geq 0 \wedge k = n \wedge n = n\ \rangle$

$res = n;$

$\langle\ k \geq 0 \wedge k = n \wedge res = n\ \rangle$

$\langle\ k \geq 0 \wedge res = n + 2(n - k)\ \rangle$

`while` $(k > 0)$ {

$\langle\ k \geq 0 \wedge res = n + 2(n - k) \wedge k > 0\ \rangle$

$\langle\ k > 0 \wedge res + 2 = n + 2(n - k) + 2\ \rangle$

$res = res + 2;$

$\langle\ k > 0 \wedge res = n + 2(n - k) + 2\ \rangle$

$\langle\ k - 1 \geq 0 \wedge res = n + 2(n - (k - 1))\ \rangle$

$k = k - 1;$

$\langle\ k \geq 0 \wedge res = n + 2(n - k)\ \rangle$

}

$\langle\ k \geq 0 \wedge res = n + 2(n - k) \wedge k \not> 0\ \rangle$

$\langle\ res = 3n\ \rangle$

Vorname	Name	Matr.-Nr.

5

b) Beweisen Sie die Terminierung des Algorithmus P.
Wir wählen die Variante k. Dann gilt $k > 0 \Rightarrow k \geq 0$ und

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

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

$res = res + 2;$

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

$k = k - 1;$

$$\langle\ k < m\ \rangle$$

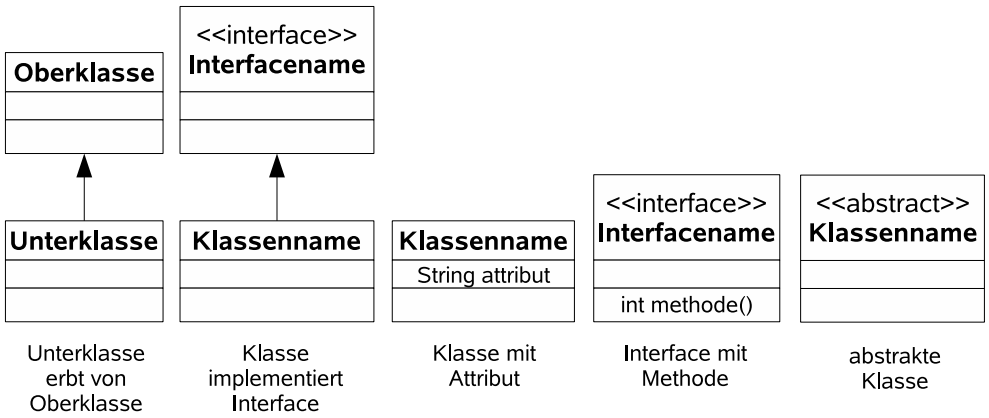
Vorname	Name	Matr.-Nr.

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

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

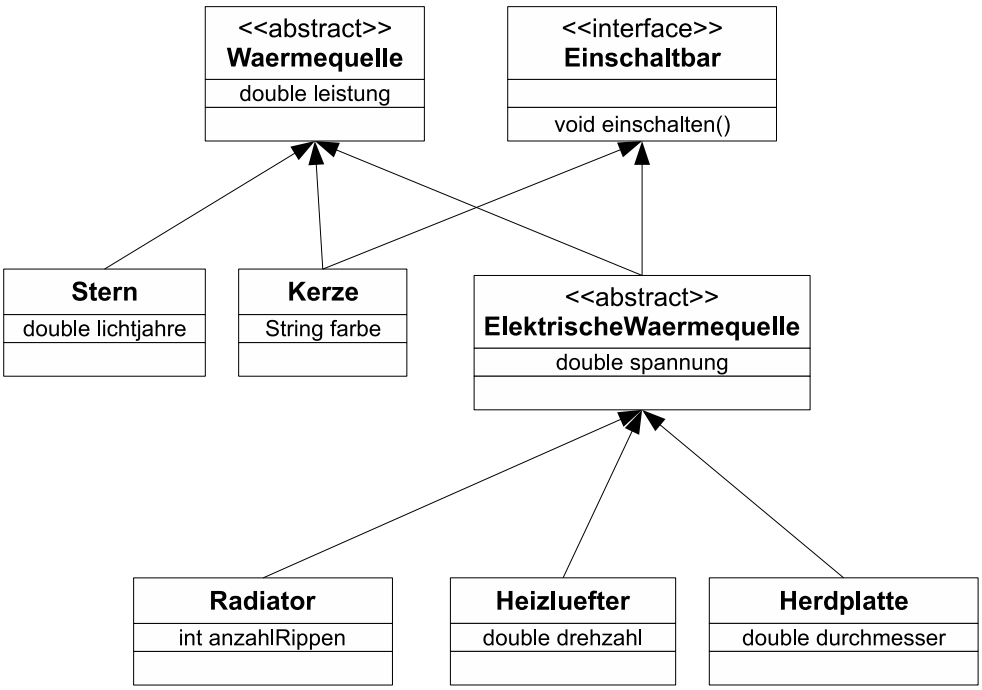
- Jede Wärmequelle wird durch ihre thermische Leistung in Watt gekennzeichnet.
- Eine Kerze ist eine Wärmequelle, die sich durch ihre Farbe auszeichnet.
- Ein Stern ist eine Wärmequelle, für die der Abstand von der Erde in Lichtjahren eine wichtige Kenngröße ist.
- Eine elektrische Wärmequelle ist eine Wärmequelle, die die Wärme durch Umwandlung von elektrischer Energie in Wärmeenergie erzeugt. Für elektrische Wärmequellen ist die Betriebsspannung relevant.
- Ein Radiator ist eine elektrische Wärmequelle mit einer bestimmten Anzahl von Heizrippen.
- Heizlüfter sind ebenfalls elektrische Wärmequellen. Ein Heizlüfter zeichnet sich durch einen eingebauten Ventilator mit einer bestimmten Drehzahl aus.
- Eine Herdplatte ist eine elektrische Wärmequelle, für die der Durchmesser wichtig ist.
- Sowohl elektrische Wärmequellen als auch Kerzen können vom Menschen in Betrieb gesetzt werden. Deshalb stellen sie eine Methode zur Verfügung, mit der sie eingeschaltet werden können.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Wärmequellen. 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 und Datentypen ihrer Attribute an. Methoden von Klassen müssen nicht angegeben werden. Geben Sie für jedes Interface ausschließlich den jeweiligen Namen sowie die Namen und Ein- und Ausgabetypen seiner Methoden an.

Vorname	Name	Matr.-Nr.



Vorname	Name	Matr.-Nr.

b) Implementieren Sie in Java eine Methode `machWarm`. Die Methode bekommt als Eingabeparameter ein Array von Wärmequellen. Sie soll alle Wärmequellen einschalten, bei denen dies für den Menschen möglich ist. Als Ergebnis soll die Methode die Anzahl der in Betrieb gesetzten Wärmequellen zurückgeben.

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 int machWarm(Waermequelle[] array) {
    int ergebnis = 0;

    for (int i = 0; i < array.length; i++) {
        if (array[i] instanceof Einschaltbar) {
            Einschaltbar e = (Einschaltbar) array[i];
            e.einschalten();
            ergebnis++;
        }
    }

    return ergebnis;
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 4 (Programmierung in Java, 4 + 4 + 10 + 10 Punkte**)**

Die Klasse `ListeVonZahlen` repräsentiert eine Liste von Zahlen. Jedes Listenelement wird als Objekt der Klasse `WertElement` dargestellt. Ein Listenelement enthält eine Zahl und einen Verweis auf den Nachfolger. Die Zahl wird in dem Attribut `wert` gespeichert und das Attribut `naechstes` zeigt auf das nächste Element der Liste. Das letzte Element einer Liste hat keinen Nachfolger, so dass dessen Attribut `naechstes` auf `null` zeigt. Objekte der Klasse `ListeVonZahlen` haben ein Attribut `kopf`, das auf das erste Element der Liste zeigt. Eine leere Liste hat kein erstes Element, so dass hier das Attribut `kopf` auf `null` zeigt.

```
public class ListeVonZahlen {
    public WertElement kopf;
    public ListeVonZahlen(WertElement kopf) {
        this.kopf = kopf;
    }
}

public class WertElement {
    public double wert;
    public WertElement naechstes;

    public WertElement(double wert, WertElement naechstes) {
        this.wert = wert;
        this.naechstes = naechstes;
    }
}
```

Die Klasse `ListeVonListen` repräsentiert eine Liste von Listen von Zahlen. Jedes Listenelement wird als Objekt der Klasse `ListenElement` dargestellt. Die Attribute in diesen Klassen sind analog zu den Attributen in den Klassen `ListeVonZahlen` und `WertElement`.

```
public class ListeVonListen {
    public ListenElement kopf;
}

public class ListenElement {
    public ListeVonZahlen liste;
    public ListenElement naechstes;
}
```

Verwenden Sie bei der Implementierung der folgenden Methoden die Schlüsselworte `public` und `private` auf sinnvolle Weise und kennzeichnen Sie Methoden als `static`, falls angebracht.

Vorname	Name	Matr.-Nr.

10

a) Implementieren Sie die Methode

`public int laenge()`

der Klasse `ListeVonZahlen`, die die Länge der aktuellen Liste berechnet. Sie dürfen beliebige Hilfsmethoden einführen. Verwenden Sie dabei aber keine Schleifen, sondern ausschließlich Rekursion.

```
public int laenge() {
    return laenge(this.kopf);
}

private static int laenge(WertElement aktuell) {
    if (aktuell == null) {
        return 0;
    }
    return 1+laenge(aktuell.naechstes);
}
```

Vorname	Name	Matr.-Nr.

b) Implementieren Sie die Methode

```
public double summe()
```

der Klasse `ListeVonZahlen`, die die Summe aller Zahlen in der aktuellen Liste berechnet. Falls die aktuelle Liste leer ist, soll die Methode `0.0` zurückliefern. Sie dürfen beliebige Hilfsmethoden einführen. Verwenden Sie dabei aber keine Schleifen, sondern ausschließlich Rekursion.

```
public double summe() {
    return summe(this.kopf);
}

private static double summe(WertElement aktuell) {
    if (aktuell == null) {
        return 0.0;
    }
    return aktuell.wert+summe(aktuell.naechstes);
}
```

Vorname	Name	Matr.-Nr.

c) Die Methode `anwenden` der Klasse `ListeVonListen` wendet einen Kombinator auf die Liste von Listen an. Ein Kombinator ist ein Objekt einer Klasse, welche das Interface `Kombinator` implementiert. Ein Beispiel hierfür ist die Klasse `LaengeKombinator`.

```
public interface Kombinator {
    public double kombinieren(ListeVonZahlen liste);
}

public class LaengeKombinator implements Kombinator {
    public double kombinieren(ListeVonZahlen liste) {
        return liste.laenge();
    }
}
```

Jeder Kombinator besitzt also eine Methode `kombinieren`, die eine Liste von Zahlen auf eine Zahl abbildet. In der Klasse `LaengeKombinator` bildet die Methode `kombinieren` beispielsweise Listen auf ihre Länge ab.

Die Methode `kombinieren` des Kombinators soll durch die Methode `anwenden` auf jede Element-Liste der `ListeVonListen` angewendet werden. Für eine `ListeVonListen` `ll` der Form `[`l_1`,`l_2`,`l_3`]` soll der Aufruf `ll.anwenden(k)` für einen Kombinator `k` dann folgende `ListeVonZahlen` ergeben: `[k.kombinieren(`l_1`), k.kombinieren(`l_2`), k.kombinieren(`l_3`)]`. Falls `k` beispielsweise ein Objekt der Klasse `LaengeKombinator` ist und l_1 die Länge 5, l_2 die Länge 9 und l_3 die Länge 7 hat, so ergibt `ll.anwenden(k)` die Ergebnisliste `[5,9,7]`.

Implementieren Sie die Methode

```
public ListeVonZahlen anwenden(Kombinator k)
```

der Klasse `ListeVonListen`. Sie dürfen beliebige Hilfsmethoden einführen. Verwenden Sie dabei aber keine Schleifen, sondern ausschließlich Rekursion.

```
public ListeVonZahlen anwenden(Kombinator k){
    return new ListeVonZahlen(anwenden(this.kopf,k));
}

private static WertElement anwenden(ListenElement akt,Kombinator k) {
    if (akt == null){
        return null;
    } else {
        double d = k.kombinieren(akt.liste);
        return new WertElement(d,anwenden(akt.naechstes,k));
    }
}
```

Vorname	Name	Matr.-Nr.

Vorname	Name	Matr.-Nr.

d) Die Methode `durchschnitt` der Klasse `ListeVonListen` berechnet erst den Durchschnitt jeder einzelnen Element-Liste der aktuellen Liste von Listen und daraus dann den Gesamtdurchschnitt. Der Durchschnitt einer Liste von Zahlen ist die Summe der Zahlen in der Liste geteilt durch die Anzahl der Elemente in der Liste, wobei eine leere Liste (und auch die Liste `null`) den Durchschnitt 0 hat. Der Gesamtdurchschnitt einer Liste von Listen ist die Summe der Durchschnitte der einzelnen Element-Listen geteilt durch die Anzahl der Element-Listen. Wenn die Liste von Listen leer ist, ist ihr Gesamtdurchschnitt ebenfalls 0.

Implementieren Sie zunächst die Klasse `DurchschnittKombinator`, welche das Interface `Kombinator` implementiert. Die Methode `kombinieren` der Klasse `DurchschnittKombinator` soll jede Liste von Zahlen auf ihren Durchschnitt abbilden. Verwenden Sie dann die Methode `anwenden` aus Aufgabenteil (c), um die Methode

```
public double durchschnitt()
```

der Klasse `ListeVonListen` zu implementieren.

```
public class DurchschnittKombinator implements Kombinator {
    public double kombinieren(ListeVonZahlen liste) {
        if (liste == null || liste.kopf == null) return 0;
        else return liste.summe()/liste.laenge();
    }
}

public double durchschnitt() {
    DurchschnittKombinator d = new DurchschnittKombinator();
    return d.kombinieren(this.anwenden(d));
}
```

Vorname	Name	Matr.-Nr.

Aufgabe 5 (Funktionale Programmierung in Haskell, $4 + 5 + 4 + 4$ Punkte**)**

a) Geben Sie den allgemeinsten Typ der Funktionen `f` und `g` an, die wie folgt definiert sind. Gehen Sie davon aus, dass `1` den Typ `Int` hat.

```
f x y z = 1 : (f (x-1) z y)

f :: Int -> a -> a -> [Int]

g x = \y -> x

g :: a -> b -> a
```

b) Gegeben seien folgende Datendeklarationen:

```
data Richtung = Norden | Osten | Sueden | Westen

data Anweisung = StiftHeben | StiftSenken | Bewegung Richtung Int
```

Im Folgenden sollen Sie Funktionen zur Steuerung eines Plotters in Haskell programmieren. Der Plotter kann einen Stift heben und senken und sich in jede Himmelsrichtung um einen beliebigen positiven `Int`-Wert bewegen. Eine Anweisungsfolge wäre beispielsweise:

> Stift senken,
> 6 Schritte Richtung Norden bewegen,
> Stift heben,
> 2 Schritte Richtung Westen bewegen,
> 3 Schritte Richtung Süden bewegen,
> Stift senken,
> 4 Schritte Richtung Osten bewegen.

Solche Anweisungsfolgen lassen sich als Listen vom Typ `[Anweisung]` schreiben. Die obige Anweisungsfolge würde also durch folgende Liste repräsentiert:

```
[StiftSenken, (Bewegung Norden 6), StiftHeben, (Bewegung Westen 2),
(Bewegung Sueden 3), StiftSenken, (Bewegung Osten 4)]
```

Vorname	Name	Matr.-Nr.

Implementieren Sie die Funktion `strichlaenge` vom Typ `[Anweisung] -> Int`, die für eine gegebene Anweisungsfolge berechnet, welche Strecke mit **gesenktem** Stift **abgefahren** wurde. Gehen Sie hierbei davon aus, dass der Stift zu Beginn nicht abgesenkt ist. Erneutes Heben eines gehobenen Stiftes bzw. erneutes Senken eines abgesenkten Stiftes haben keinen Einfluss. Für die oben angegebene Anweisungsfolge errechnet sich beispielsweise eine Strichlänge von 10. Sie dürfen dabei beliebige Hilfsfunktionen schreiben.

```
strichlaenge :: [Anweisung] -> Int
strichlaenge = ohneStift
    where ohneStift :: [Anweisung] -> Int
          ohneStift (StiftSenken:r) = mitStift r
          ohneStift (_:r) = ohneStift r
          ohneStift [] = 0
          mitStift :: [Anweisung] -> Int
          mitStift (StiftHeben:r) = ohneStift r
          mitStift (StiftSenken:r) = mitStift r
          mitStift ((Bewegung _ i):r) = i + (mitStift r)
          mitStift [] = 0
```

Vorname	Name	Matr.-Nr.

c) Implementieren Sie die Funktion `bewegen` vom Typ `(Int, Int) -> Anweisung -> (Int, Int)`. Diese Funktion bekommt eine Position und eine Anweisung als Parameter und liefert die Position zurück, die nach der Ausführung der Anweisung vorliegt. Hierbei ist die erste Zahl des Positions-Tupels die West-Ost Ausrichtung (nach Osten steigend) und die zweite Zahl die Nord-Süd Ausrichtung (nach Norden steigend). Falls beispielsweise `(5, 3)` die aktuelle Position ist, dann liefert `bewegen` für eine Bewegung um `5` nach Süden die Position `(5, -2)`.

```
bewegen :: (Int, Int) -> Anweisung -> (Int, Int)
bewegen (x, y) (Bewegung Norden i) = (x, y + i)
bewegen (x, y) (Bewegung Osten  i) = (x + i, y)
bewegen (x, y) (Bewegung Sueden i) = (x, y - i)
bewegen (x, y) (Bewegung Westen i) = (x - i, y)
bewegen p _ = p
```

d) Implementieren Sie die Funktion `position` vom Typ `(Int, Int) -> [Anweisung] -> (Int, Int)`. Diese Funktion bekommt eine Anfangsposition und eine Liste von Anweisungen als Parameter. Sie liefert die Position, an der sich der Stift befindet, nachdem alle Anweisungen der Liste bearbeitet wurden.

```
position :: (Int, Int) -> [Anweisung] -> (Int, Int)
position p [] = p
position p (x : xs) = position (bewegen p x) xs
```

Vorname	Name	Matr.-Nr.

Aufgabe 6 (Logische Programmierung in Prolog, (2 + 2) + 5 + (4 + 4) Punkte**)**

a) Geben Sie für die folgenden Paare von Termen den allgemeinsten Unifikator an, oder begründen Sie kurz, warum dieser nicht existiert.

- `h(g(X),p(Y),p(g(A)),p(A))` und `h(Y,Z,Z,p(X))`

  ```
  X = A
  Y = g(A)
  Z = p(g(A))
  ```

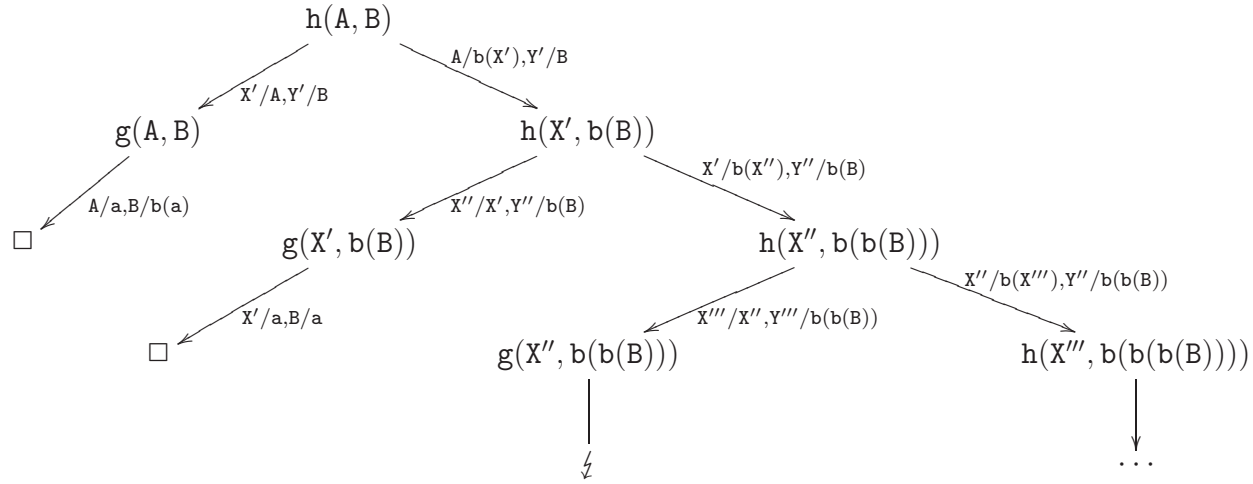
- `f(X,g(X),Y)` und `f(g(Z),Y,X)`

  ```
  X = g(Z)
  Y = g(g(Z))
  ```
  für letztes Paar:
  `g(Z)` $\stackrel{!}{=}$ `g(g(Z))` $\Rightarrow$ `occur failure`

Vorname	Name	Matr.-Nr.

b) Erstellen Sie für das folgende Logikprogramm den Beweisbaum zur Anfrage „`?- h(A,B).`“ und geben Sie alle Antwortsubstitutionen an. Sie dürfen Pfade abbrechen, sobald diese eine Anfrage enthalten, in der das Funktionssymbol `b` dreimal auftritt. Kennzeichnen Sie unendliche Pfade durch „. . .“.

```
g(a,b(a)).
h(X,Y) :- g(X,Y).
h(b(X),Y) :- h(X,b(Y)).
```



Die Lösungen sind die Substitutionen `A=a, B=b(a)` *und* `A=b(a), B=a`.

Vorname	Name	Matr.-Nr.

20

c)
- Programmieren Sie das dreistellige Prädikat `insert`. Dieses Prädikat soll dafür verwendet werden, eine natürliche Zahl an der richtigen Stelle in eine aufsteigend sortierte Liste einzufügen. Beispielsweise liefert die Anfrage „`?- insert([1,4,6],3,YS).`“ die Antwortsubstitution `YS = [1,3,4,6]`. Falls das erste Argument keiner sortierten Liste entspricht, kann sich Ihr Programm beliebig verhalten. Verwenden Sie keine vordefinierten Prädikate bis auf `<` und `>=`.

  ```
  insert([], E, [E]).
  insert([X|XS], E, [E,X|XS]) :- E < X.
  insert([X|XS], E, [X|YS]) :- E >= X, insert(XS, E, YS).
  ```

- Programmieren Sie das zweistellige Prädikat `insSort`, wobei „`?- insSort`(l_1, l_2).“ für zwei Listen l_1 und l_2 wahr ist, falls l_2 aufsteigend sortiert ist und die gleichen Elemente wie l_1 enthält. Beispielsweise ergibt „`?- insSort([2,1],[1,2]).`“ die Antwort `true` und die Anfrage „`?- insSort([7,2,5,3],YS).`“ liefert die Antwortsubstitution `YS = [2,3,5,7]`. Verwenden Sie in Ihrer Implementierung das Prädikat `insert`.

  ```
  insSort([], []).
  insSort([X|XS], ZS) :- insSort(XS, YS), insert(YS,X,ZS).
  ```