Prof. Dr. J. Giesl
M. Brockschmidt, F. Emmes, Dr. C. Fuhs, C. Otto, T. Ströder

Klausur Programmierung WS 2011/2012

Vorname: _____

Nachname: _____

Matrikelnummer: _____

Studiengang (bitte **genau** einen markieren):

- ○ Informatik Bachelor
- ○ Informatik Lehramt (Bachelor)
- ○ Mathematik Bachelor
- ○ Informatik Lehramt (Staatsexamen)
- ○ Sonstiges: _____

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	**12**	
Aufgabe 2	**10**	
Aufgabe 3	**14**	
Aufgabe 4	**30**	
Aufgabe 5	**17**	
Aufgabe 6	**17**	
Summe	**100**	

Allgemeine Hinweise:

- **Auf alle Blätter** (inklusive zusätzliche Blätter) müssen Sie **Ihren Vornamen, Ihren Nachnamen und Ihre Matrikelnummer** schreiben.
- Geben Sie Ihre Antworten in lesbarer und verständlicher Form an.
- Schreiben Sie mit **dokumentenechten** Stiften, nicht mit roten oder grünen Stiften und nicht mit Bleistiften.
- Bitte beantworten Sie die Aufgaben auf den Aufgabenblättern (benutzen Sie auch die Rückseiten).
- Geben Sie für jede Aufgabe **maximal eine** Lösung an. Streichen Sie alles andere durch. Andernfalls werden alle Lösungen der Aufgabe mit **0 Punkten** bewertet.
- Werden **Täuschungsversuche** beobachtet, so wird die Klausur mit **0 Punkten** bewertet.
- Geben Sie am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.

Name: Matrikelnummer:

Aufgabe 1 (Programmanalyse): (12 Punkte)

Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Tragen Sie hierzu jeweils die ausgegebenen Zeichen in die markierten Stellen hinter „`OUT:`“ ein. Achten Sie darauf, dass Gleitkommazahlen in der Form “5.0” und nicht als ”5” ausgegeben werden.

```
public class A {
  public static double x = 2;
  public int y;

  public A() {
    this.x++;
    this.y = 1;
  }

  public A(double x) {
    this.y = (int) x;
    this.x += this.y;
  }

  public int f(long x) {
    return 2;
  }

  public int f(float x) {
    return 3;
  }
}
```

```
public class B extends A {
  public int x = 3;

  public B(int x) {
    super(x);
    this.x = x;
  }

  public int f(int x) {
    return 4;
  }

  public int f(long x) {
    return 5;
  }
}
```

```
public class M {
  public static void main(String[] args) {
    A aa = new A(1.5);
    System.out.println(aa.x + " " + aa.y);// OUT: [   ] [   ]

    int retAA = aa.f(1);
    System.out.println(retAA);             // OUT: [   ]

    B bb = new B(6);
    A ab = bb;
    System.out.println(bb.x);              // OUT: [   ]
    System.out.println(ab.x + " " + ab.y);// OUT: [   ] [   ]

    int retBB = bb.f(1f);
    System.out.println(retBB);             // OUT: [   ]

    int retAB = ab.f(1);
    System.out.println(retAB);             // OUT: [   ]
  }
}
```

Aufgabe 2 (Hoare-Kalkül): (8 + 2 = 10 Punkte)

Gegeben sei untenstehendes *Java*-Programm P, das zu einer Eingabe $\texttt{n} \geq 0$ den Wert $2^{\texttt{n}}$ berechnet.

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.

Hinweise:

- Sie dürfen beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x+1 = y+1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen.

$\langle 0 \leq \texttt{n} \rangle$

$\langle$ _____ $\rangle$

```
res = 1;
```

$\langle$ _____ $\rangle$

```
i = 0;
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
while (i < n) {
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
    res = 2 * res;
```

$\langle$ _____ $\rangle$

```
    i = i + 1;
```

$\langle$ _____ $\rangle$

```
}
```

$\langle$ _____ $\rangle$

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

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung unter der Voraussetzung $0 \leq \mathtt{n}$ bewiesen werden.

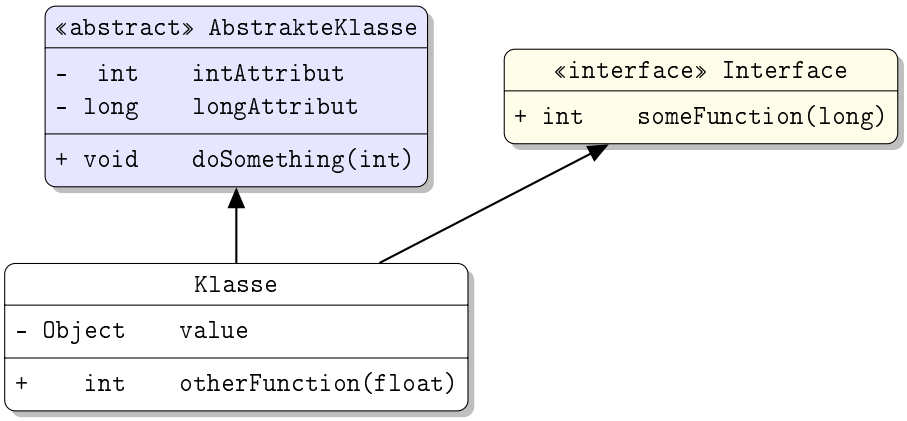
Aufgabe 3 (Klassen-Hierarchie): (6 + 8 = 14 Punkte)

Ziel dieser Aufgabe ist die Erstellung einer Hierarchie zur Verwaltung von Spielkugeln.

- Eine Spielkugel hat einen Durchmesser, der als ganze Zahl in Millimetern angegeben wird.
- Jede Spielkugel soll eine Methode `istDreckig()` zur Verfügung stellen, die genau dann `true` zurückgibt, wenn die Spielkugel schmutzig ist.
- Eine Murmel ist eine Spielkugel, die sich durch ihre Farbe auszeichnet, die als `String` angegeben wird.
- Bälle sind spezielle Spielkugeln, für die gespeichert werden soll, wie groß das Volumen ihres Innenraums in (ganzen) Kubikmillimetern ist.
- Ein Golfball ist ein Ball, der sich durch die Anzahl seiner "Dimples" auszeichnet. Dimples sind die kleinen Dellen auf dem Golfball.
- Tennisbälle sind Bälle, deren Innendruck wir (als `double` in bar) speichern wollen.
- Für diese Aufgabe wollen wir annehmen, dass Murmeln, Golfbälle und Tennisbälle die einzigen Spielkugeln sind, die existieren.
- Golfbälle und Murmeln können poliert werden und bieten dafür die Funktion `polieren()` an, die keinen Rückgabewert hat.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Spielkugeln. Notieren Sie keine Konstruktoren oder Selektoren. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden und markieren Sie alle Klassen als abstrakt, bei denen dies sinnvoll ist.

Verwenden Sie hierbei die folgende Notation:



Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \longrightarrow A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist). Benutzen sie `-` um `private` abzukürzen und `+` für alle anderen Sichtbarkeiten (wie z.B. `public`).

Tragen Sie keine vordefinierten Klassen (`String`, etc.) oder Pfeile dorthin in ihr Diagramm ein.

b) Schreiben Sie eine Methode `putzen`, die ein Array von `Spielkugeln` erhält und dann alle Spielkugeln poliert, die dies erlauben. Ist eine polierte Spielkugel danach weiterhin dreckig, soll eine `FesterDreckException` geworfen werden, die wie folgt definiert ist. Das Attribut `dreckig` der geworfenen `FesterDreckException` soll dabei auf die dreckige Spielkugel verweisen.

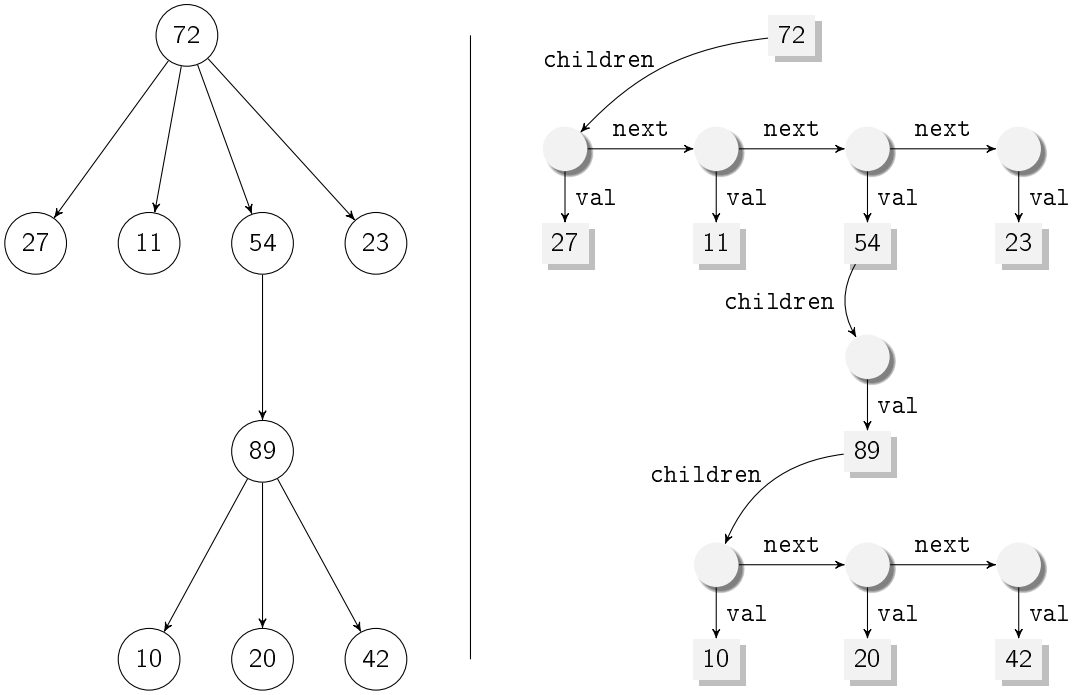
```
public class FesterDreckException extends Exception {
    private Object dreckig;

    public FesterDreckException(Object d) {
        this.dreckig = d;
    }
}
```

Setzen Sie voraus, dass das übergebene Array immer existiert und nie `null` ist. Kennzeichnen Sie die Methode mit dem Schlüsselwort `static`, falls dies angebracht ist. Achten Sie darauf, in der Methodendeklaration zu erklären, dass eine Exception geworfen wird.

Name: **Matrikelnummer:**

Aufgabe 4 (Programmieren in Java): (4 + 5 + 7 + 7 + 7 = 30 Punkte)

Wir betrachten Vielwegbäume, in denen jeder Knoten beliebig viele Nachfolger haben kann. Ein solcher Baum ist in der folgenden Graphik links dargestellt:



Wir verwenden zur Repräsentation der Datenstruktur in Java die beiden folgenden Klassen `Liste` und `Baum`:

```
public class Liste<T> {
  protected T val;
  protected Liste<T> next;
}

public class Baum {
  protected int label;
  protected Liste<Baum> children;
}
```

Eine Darstellung des Beispielbaums in unserer Datenstruktur ist in der Graphik rechts zu sehen. Dort sind Objekte der Klasse `Liste` als (leere) Kreise und Objekte der Klasse `Baum` als Quadrate dargestellt, die den gespeicherten `int`-Wert enthalten. Der dargestellte Baum hat also als Wurzel ein `Baum`-Objekt, dessen `label`-Attribut 72 ist und dessen `children`-Attribut auf eine vierelementige Liste zeigt. Die Elemente dieser Liste sind jeweils durch das Attribut `next` verbunden, während ihre `val`-Attribute jeweils auf `Baum`-Objekte zeigen. Der leere Baum und die leere Liste werden mit `null` dargestellt. Attribute, deren Wert `null` ist, werden in der Graphik nicht gezeigt.

a) Implementieren Sie die Methode `T last()` in der Klasse `Liste<T>`, die den Wert von `val` im letzten Element der Liste zurückgibt. Verwenden Sie dazu **ausschließlich Schleifen** und keine Rekursion. Verändern Sie die Liste nicht durch Schreibzugriffe.

Beispielsweise soll für die unterste Liste in der Graphik (welche die `Baum`-Objekte mit den Werten 10, 20 und 42 enthält) also das `Baum`-Objekt zurückgegeben werden, das 42 enthält.

b) Implementieren Sie die Methode `int rightmostLeaf()` in der Klasse `Baum`, die den Wert von `label` im rechtesten, untersten Blatt des Baumes zurückgibt. Um das rechteste, unterste Blatt eines Baumes zu finden, gehen Sie jeweils zum rechtesten Kind und dann "nach unten" zu den Nachfolgern dieses Kindes. Für den Beispielbaum aus der Graphik soll also 23 (und nicht etwa 42) zurückgegeben werden. Verwenden Sie keine Schleifen, sondern **ausschließlich Rekursion**. Sie dürfen aber `last()` aus Teil a) benutzen.

c) Wir führen nun das Interface `Summierbar` ein, das von solchen Klassen implementiert wird, die eine Methode `int sum()` zum Aufsummieren ihrer Elemente zur Verfügung stellen. Wir passen außerdem unsere Klasse `Liste` so an, dass darin nur Werte gespeichert werden können, die `Summierbar` implementieren:

```
public interface Summierbar {
  int sum();
}
```

```
public class Liste<T extends Summierbar>
 implements Summierbar {
  protected T val;
  protected Liste<T> next;

  ...
}
```

Implementieren Sie die Methode `int sum()` in der Klasse `Liste`. Dabei ergibt sich die Summe einer Liste, indem man `sum()` jeweils auf den Werten der Listenelemente aufruft und die Ergebnisse aufaddiert. Sie dürfen annehmen, dass `val` nie `null` ist.

Beispielsweise soll für die unterste Liste in der Graphik (welche die `Baum`-Objekte mit den Werten 10, 20 und 42 enthält) das Ergebnis 72 zurückgegeben werden, wenn die Klasse `Baum` das Interface `Summierbar` so implementiert, dass `sum()` die Summe der im Baum gespeicherten `label`-Werte zurückgibt.

d) Wie in Teil c) erwähnt, erweitern wir die Deklaration der Klasse `Baum` um `implements Summierbar`. Implementieren Sie dafür die Methode `int sum()` in der Klasse `Baum` so, dass sich die Summe eines Baums ergibt, indem man den Wert von `label` und die Summe aller Kinder addiert.

Hinweise:

- Verwenden Sie auch die Methode `sum()` der Klasse `Liste` aus Teil c).

e) Implementieren Sie eine Methode `LinkedList<Integer> toList()` in der Klasse `Baum`, die alle im Baum enthaltenen Werte in einer Liste vom Typ `LinkedList<Integer>` aus dem `Collections`-Framework zurückgibt. Das heißt, dass Sie **nicht** unseren eigenen Listentyp `Liste` verwenden sollen.

Die Reihenfolge der Elemente in der zurückgegebenen Liste spielt keine Rolle. Für unseren Beispielbaum könnte also z.B. sowohl `[72,27,11,54,23,89,10,20,42]` als auch `[72,27,11,54,89,10,20,42,23]` zurückgegeben werden. Nehmen Sie an, dass das Feld `val` in Objekten vom Typ `Liste` nie `null` ist.

Hinweise:

- Sie dürfen davon ausgehen, dass das `Collections`-Framework mittels `import java.util.*` bereits importiert wurde.
- Die Klasse `LinkedList<T>` stellt eine nicht-statische Methode `add(T element)` zur Verfügung, die ein Element in die Liste einfügt. Zudem gibt es die nicht-statische Methode `addAll(Collection<T> c)`, die alle Elemente von `c` der Liste hinzufügt.

Name: **Matrikelnummer:**

Aufgabe 5 (Haskell): ((1,5 + 1,5) + 1,5 + 2,5 + (3 + 7) = 17 Punkte)

a) Geben Sie zu den folgenden Haskell-Funktionen `f` und `g` jeweils den allgemeinsten Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben und die Funktion + den Typ `Int -> Int -> Int` hat.

i)
```
f x y z   = if y then x == z else y
f x y [u] = (\x -> y == u) x
```

ii)
```
g x = \y -> (\x -> x + 3) x
```

b) Bestimmen Sie, zu welchem Ergebnis der Ausdruck `exp` auswertet.

```
exp :: [Int]
exp = (\x f -> map f x) [1,2] (\y -> y * 3)
```

c) Sei `xs` eine Liste und `f1`, `g1` zwei Funktionen, so dass `filter f1 (map g1 xs)` ein korrekter Haskell-Ausdruck ist. Wie kann man die Funktionen `f2` und `g2` mit Hilfe von `f1` und `g1` definieren, so dass

```
filter f1 (map g1 xs) == map g2 (filter f2 xs)
```

gilt?

d) Eine einfache Methode, um Listen von Zeichen zu komprimieren, ist das sogenannte *R*un-*L*ength *E*ncoding. Hierbei wird eine Liste von Zeichen durch eine Liste von Tupeln dargestellt, wobei eine Sequenz von n aufeinanderfolgenden gleichen Zeichen x innerhalb der Liste durch ein Tupel (n, x) dargestellt wird.

Die Liste `xs = ['a', 'b', 'b', 'b', 'c', 'c']` kann also zu `ys = [(1, 'a'), (3, 'b'), (2, 'c')]` komprimiert werden.

i) Implementieren Sie die Funktion `decode :: [(Int, Char)] -> [Char]`, welche eine komprimierte Liste von Tupeln in die ursprüngliche unkomprimierte Liste umwandelt. Beispielsweise soll `decode ys` die Liste `xs` liefern. Verwenden Sie hierbei die vordefinierten Haskell-Funktionen `++ :: [a] -> [a] -> [a]` und `replicate :: Int -> a -> [a]`, welche Listen einer bestimmten Länge mit gleichen Elementen erstellt. Ein Aufruf `replicate 3 'b'` ergibt beispielsweise `['b', 'b', 'b']`.

ii) Implementieren Sie die Funktion `encode :: [Char] -> [(Int, Char)]`, welche Listen nach dem beschriebenen Verfahren komprimiert. Beispielsweise liefert `encode xs` also das Ergebnis `ys`.

Hinweis: Unterscheiden Sie hierbei 3 Fälle:

- Die Eingabeliste ist leer.
- Die Eingabeliste besteht aus *genau einem* Element.
- Die Eingabeliste enthält *mehr als ein* Element. In diesem Fall können Sie bei einem *rekursiven* Aufruf der `encode`-Funktion (auf dem Teil der Eingabeliste ohne das erste Element) davon ausgehen, dass eine nicht-leere Liste von Tupeln zurückgegeben wird.

Ihre Funktion soll die kürzestmögliche komprimierte Liste zurückgeben.

Aufgabe 6 (Prolog): (3 + (4 + 1) + (2 + 3 + 4) = 17 Punkte)

a) Geben Sie zu den folgenden Termpaaren jeweils einen allgemeinsten Unifikator an oder begründen Sie, warum sie nicht unifizierbar sind. Hierbei beginnen Variablen mit Großbuchstaben und Funktionssymbole mit Kleinbuchstaben.

i) $f(a, Y, Y), f(X, b, X)$

ii) $g(X, a, Y), g(b, Z, Z)$

iii) $h(c(X), X), h(Y, Y)$

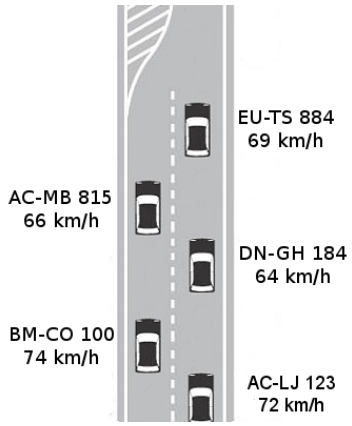
b) Gegeben sei folgendes *Prolog*-Programm P.

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

i) Erstellen Sie für das Programm P den Beweisbaum zur Anfrage `?- p(X, Y).` bis zur Tiefe 4 (die Wurzel hat dabei Tiefe 1). Markieren Sie Pfade, die zu einer unendlichen Auswertung führen, mit ∞.

ii) Geben Sie alle Antwortsubstitutionen zur Anfrage `?- p(X, Y).` an und geben Sie außerdem an, welche dieser Antwortsubstitutionen von *Prolog* gefunden werden.

c) In diesem Aufgabenteil betrachten wir das Reißverschlussprinzip im Straßenverkehr bei einer Verengung einer zweispurigen Straße auf nur noch eine Spur. Ein Auto hat dabei ein Kennzeichen (angegeben als Konstante mit Kleinbuchstaben ohne Leerzeichen - z.B. wird EU-TS 884 als eu-ts884 angegeben) und eine Geschwindigkeit (angegeben als ganze Zahl in km/h - z.B. wird 69 km/h als 69 angegeben). Eine solche Situation ist in der rechten Abbildung zu sehen. Die beiden Spuren werden jeweils als Listen von Autos modelliert, wobei Autos durch zweistellige Terme mit dem Funktionssymbol `auto` repräsentiert werden. Die folgenden Terme repräsentieren die beiden Spuren in der nebenstehenden Abbildung.
Linke Spur: `[auto(ac-mb815,66),auto(bm-co100,74)]`
Rechte Spur: `[auto(eu-ts884,69),auto(dn-gh184,64),auto(ac-lj123,72)]`



i) Schreiben Sie ein zweistelliges Prädikat `kennzeichen`, welches im ersten Argument eine Liste von Autos erhält und im zweiten Argument die Liste der Kennzeichen dieser Autos berechnet. Der Aufruf `kennzeichen([auto(eu-ts884,69),auto(dn-gh184,64),auto(ac-lj123,72)],X)` führt z.B. zur Antwortsubstitution `X = [eu-ts884,dn-gh184,ac-lj123]`.

ii) Schreiben Sie ein dreistelliges Prädikat `einordnen`, welches in den ersten beiden Argumenten jeweils Listen A und B von Autos erhält und im dritten Argument eine Liste C von Autos berechnet, die abwechselnd alle Autos aus den Listen A und B enthält, bis eine dieser Listen leer ist und danach alle übrigen Autos der anderen Liste. Falls beispielsweise A die Liste der Autos auf der linken Spur in der obigen Abbildung und B die Liste der Autos auf der rechten Spur ist, dann ergibt der Aufruf `einordnen([auto(ac-mb815,66),auto(bm-co100,74)],[auto(eu-ts884,69),auto(dn-gh184,64),auto(ac-lj123,72)],X)` die Antwortsubstitution `X = [auto(ac-mb815,66),auto(eu-ts884,69),auto(bm-co100,74),auto(dn-gh184,64),auto(ac-lj123,72)]`.

iii) Schreiben Sie ein dreistelliges Prädikat `durchschnitt`, welches im ersten Argument eine Liste von Autos erhält. Es berechnet dann im zweiten Argument die Anzahl der Autos in der Liste und im dritten Argument die Durchschnittsgeschwindigkeit dieser Autos. Beispielsweise ergibt der Aufruf `durchschnitt([auto(eu-ts884,69),auto(dn-gh184,64),auto(ac-lj123,72)],X,Y)` die Antwortsubstitution `X = 3, Y = 68.3333`. Falls das erste Argument die leere Liste ist, soll das Prädikat fehlschlagen.