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	**11**	
Aufgabe 3	**13**	
Aufgabe 4	**31**	
Aufgabe 5	**16**	
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.

```
public class A {
  public static int x = 1;
  public int y = 2;

  public A() {
    this.y += 3;
    this.x += 3;
  }

  public A(int arg) {
    int y;
    x = arg;
    y = arg + 1;
  }

  public int f(A x) {
    return 6;
  }

  public int f(Object x) {
    return 7;
  }
}
```

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

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

  public int f(A x) {
    return 8;
  }

  public int f(B x) {
    return 9;
  }
}
```

```
public class M {
  public static void main(String[] args) {
    A aa = new A(2);
    System.out.println(aa.x + " " + aa.y); // OUT: [   ] [   ]

    System.out.println(aa.f(aa));          // OUT: [   ]

    B bb = new B(4);
    System.out.println(bb.x);              // OUT: [   ]

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

    System.out.println(bb.f(bb));          // OUT: [   ]

    System.out.println(ab.f(bb));          // OUT: [   ]
  }
}
```

Aufgabe 2 (Hoare-Kalkül): (9 + 2 = 11 Punkte)

Gegeben sei untenstehendes *Java*-Programm P, das zu einer Eingabe $\mathtt{x} \geq 0$ den Wert $3 \cdot \mathtt{x}$ 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.

$\langle \mathtt{x} \geq 0 \rangle$

$\langle$ _____ $\rangle$

```
y = 0;
```

$\langle$ _____ $\rangle$

```
z = 0;
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
while (y < x) {
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
    y = y + 1;
```

$\langle$ _____ $\rangle$

```
    z = z + 2;
```

$\langle$ _____ $\rangle$

```
}
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
y = y + z;
```

$\langle \mathtt{y} = 3 \cdot \mathtt{x} \rangle$

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 $\mathrm{x} \geq 0$ bewiesen werden.

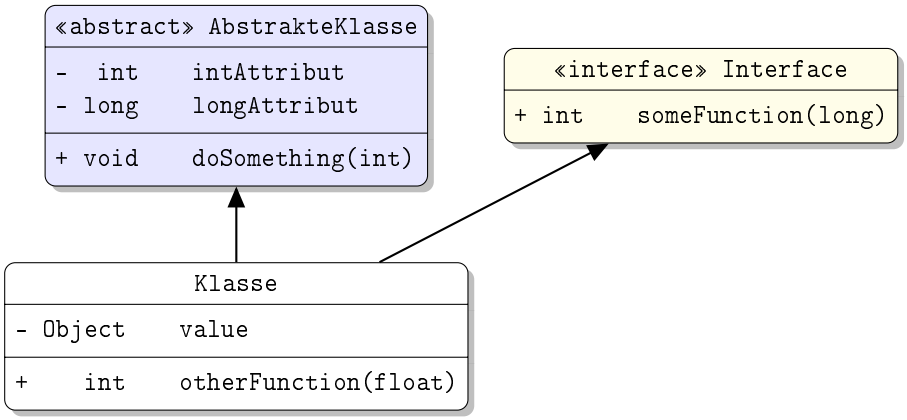
Aufgabe 3 (Klassen-Hierarchie): (6 + 7 = 13 Punkte)

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

- Ein Bodenbelag hat eine Fläche, die als ganze Zahl in Quadratmetern angegeben werden soll.
- Jeder Bodenbelag soll eine Methode `istDreckig()` zur Verfügung stellen, die genau dann `true` zurückgibt, wenn der Boden schmutzig ist.
- Einige Bodenbeläge sind elastisch. Die *Elastizitätsgrenze* dieser Beläge gibt an, wieviel Spannung (in Pascal) ausgeübt werden kann, bevor diese sich dauerhaft verformen. Dieser Wert soll als Gleitkommazahl gespeichert werden.
- Linoleum ist ein elastischer Bodenbelag. Die Dicke des Linoleums soll als ganze Zahl in Millimetern gespeichert werden.
- Teppich ist ebenfalls ein elastischer Bodenbelag.
- Laminat ist ein Bodenbelag, der nicht elastisch ist. Für Laminat wollen wir den Namen der imitierten Oberfläche als `String` speichern.
- Linoleum und Laminat können feucht gesäubert werden und bieten dafür die Funktion `wischen()` an, die keinen Rückgabewert hat.
- Außer Linoleum, Teppich und Laminat gibt es keine weiteren Arten von Bodenbelägen.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Arten von Bodenbelägen. 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 $B \blacktriangleright A$, 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 `wischListe`, die ein Array von Bodenbelägen als Eingabe erhält und eine Liste mit den Belägen aus dem Array zurückgibt, die wischbar sind. Dafür soll die vorgebene Liste `LinkedList<T>` aus dem `Collections`-Framework verwendet werden, deren Typparameter `T` so instantiiert werden soll, dass die Liste nur wischbare Beläge enthalten kann.

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.

Hinweise:

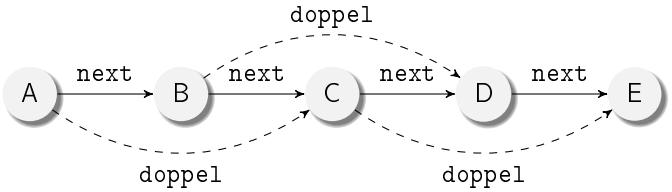
- Sie dürfen davon ausgehen, dass das `Collections`-Framework mittels `import java.util.*` bereits importiert wurde.
- Die Klasse `LinkedList<T>` stellt eine nicht-statische Methode `add(T element)` zur Verfügung, die ein Element in die Liste einfügt.

Aufgabe 4 (Programmieren in Java): (5 + 7 + 4 + 8 + 7 = 31 Punkte)

Die Klasse `DoppelListe<T>` dient zur Repräsentation von Listen. Jedes Listenelement wird als Objekt der Klasse `DoppelListe` dargestellt. Es enthält einen Wert vom generischen Typen `T`, einen Verweis auf den direkten Nachfolger und zusätzlich einen Verweis auf den **übernächsten** Nachfolger. Der Wert wird in dem Attribut `wert` gespeichert, das Attribut `next` zeigt auf das nächste Element der Liste und das Attribut `doppel` zeigt auf das übernächste Element der Liste.
Das letzte Element der Liste hat keinen Nachfolger, so dass dessen Attribut `next` auf `null` zeigt. Aus dem gleichen Grund zeigen die `doppel`-Attribute des letzten und vorletzten Listenelements ebenfalls auf `null`. Die leere Liste stellen wir als `null` dar.

```
public class DoppelListe<T> {
  T wert;
  DoppelListe<T> next;
  DoppelListe<T> doppel;
}
```

In der folgenden Graphik ist dargestellt, wie eine Liste mit den Werten `A, B, C, D, E` mit der Klasse `DoppelListe` repräsentiert wird.



Objekte der Klasse `DoppelListe` sind als Kreise dargestellt, die den im `wert`-Attribut gespeicherten Wert enthalten. Die Elemente dieser Liste sind jeweils durch das Attribut `next` (durchgezogene Pfeile) mit dem direkten Nachfolger verbunden. Das Attribut `doppel` (gestrichelte Pfeile) verbindet jedes Listenelement mit dem übernächsten Listenelement. Attribute, deren Wert `null` ist, werden in der Graphik nicht gezeigt.
Die `doppel`-Attribute der `DoppelListe`-Objekte mit den Werten `D` und `E` zeigen beide auf `null`, da für die entsprechenden Listenelemente kein übernächstes Element existiert.

Wir definieren also, dass ein Objekt `x` vom Typ `DoppelListe<T>` eine korrekte `DoppelListe` ist, falls folgende beiden Eigenschaften gelten:

- Wenn `x.next == null`, dann auch `x.doppel == null`.
- Wenn `x.next != null`, dann `x.doppel == x.next.next`.

Hinweis: Sie dürfen in **allen Teilaufgaben** davon ausgehen, dass nur auf **azyklischen Listen** gearbeitet wird.

a) Implementieren Sie einen Konstruktor für die Klasse `DoppelListe`, der als erstes Argument einen Wert vom Typ `T` und als zweites Argument eine weitere `DoppelListe<T>` bekommt.

Der Konstruktor soll ein neues `DoppelListe`-Objekt erzeugen, dessen `wert`-Attribut auf den Wert des ersten Arguments gesetzt wird und dessen direkter Nachfolger die im zweiten Argument übergebene `DoppelListe` ist. Schreiben Sie den Konstruktor so, dass auch das `doppel`-Attribut entsprechend obiger Erklärung richtig gesetzt wird. Beachten Sie hierbei bitte, dass das zweite Argument auch `null` sein kann!

b) Implementieren Sie die nicht-statische Methode `finde` in der Klasse `DoppelListe<T>`, die ein Argument vom Typ `T` übergeben bekommt. Der Rückgabewert der Methode ist das erste `DoppelListe`-Objekt in der aktuellen `DoppelListe`, dessen Wert gleich dem übergebenen Wert ist. Falls kein solches Element in der `DoppelListe` existiert, soll `null` zurückgegeben werden.

Verwenden Sie dazu **ausschließlich Schleifen** und keine Rekursion. Verändern Sie die `DoppelListe` nicht durch Schreibzugriffe. Verwenden Sie für den Vergleich die in der Klasse `Object` vordefinierte Methode `boolean equals(Object obj)`.

Hinweise:

- Sie dürfen davon ausgehen, dass sowohl die in der `DoppelListe` gespeicherten Werte als auch der übergebene Wert nicht `null` sind.
- Für diese Aufgabe brauchen Sie das `doppel`-Attribut nicht zu verwenden!

c) Implementieren Sie die statische Methode `vorneEinfuegen` ohne Rückgabewert in der Klasse `DoppelListe<T>`, die ein zusätzliches Element **zwischen dem ersten und zweiten** Element einer `DoppelListe` einfügt. Hierfür wird im ersten Argument der Methode `vorneEinfuegen` eine `DoppelListe` `erstes` vom Typ `DoppelListe<T>` übergeben. Im zweiten Argument wird ein Wert vom Typ `T` übergeben. Die Methode soll ein neues `DoppelListe`-Objekt mit dem Wert des zweiten Arguments erzeugen und dieses so in die `DoppelListe` `erstes` einfügen, dass diese anschließend das neue Objekt an der zweiten Position enthält. Hierbei sollen die Attribute so angepasst werden, dass anschließend wieder eine gültige `DoppelListe` entsprechend der Definition am Anfang der Aufgabe vorliegt.

Wenn die `DoppelListe` `erstes` vorher mindestens zwei Elemente hatte, soll das ursprüngliche zweite Element anschließend also das dritte Element sein. Entsprechendes gilt auch für alle folgenden Elemente.

Sie dürfen davon ausgehen, dass die übergebene `DoppelListe` `erstes` nicht `null` ist.

Hinweise:

- Verwenden Sie den Konstruktor aus Aufgabenteil a).
- Verwenden Sie die vorgegebene Signatur.

```
public static <T> void vorneEinfuegen(DoppelListe<T> erstes, T wert) {
```

d) Implementieren Sie die nicht-statische Methode `pruefe` in der Klasse `DoppelListe<T>` ohne Argumente und ohne Rückgabewert, die überprüft, ob die aktuelle `DoppelListe` korrekt ist (wie am Anfang der Aufgabe definiert). Falls sie nicht korrekt ist, soll eine Exception der Klasse `StrukturFehlerException` mit der folgenden Deklaration geworfen werden.

```
public class StrukturFehlerException extends Exception {
}
```

Verwenden Sie keine Schleifen, sondern **ausschließlich Rekursion**.

e) Implementieren Sie die nicht-statische Methode `rueckwaerts` ohne Argumente in der Klasse `DoppelListe<T>`, die eine `DoppelListe<T>` zurückgibt. Das Ergebnis der Methode soll eine `DoppelListe` sein, die die Elemente der aktuellen `DoppelListe` in umgekehrter Reihenfolge enthält. Verändern Sie die aktuelle `DoppelListe` nicht durch Schreibzugriffe, sondern erzeugen Sie für die Rückgabe bei Bedarf eine neue `DoppelListe`.

Hinweise:

- Falls Sie Ihre Lösung mit einer Schleife implementieren, so empfiehlt es sich, die Ergebnisliste schrittweise zu erzeugen, während Sie die vorderen Elemente der Eingabeliste abarbeiten.
- Verwenden Sie den Konstruktor aus Aufgabenteil a).

Aufgabe 5 (Haskell): ((1,5 + 1,5) + 1,5 + 2 + (3 + 5 + 1,5) = 16 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  = if x then (\z -> y ++ z) else (\z -> z ++ y)
```

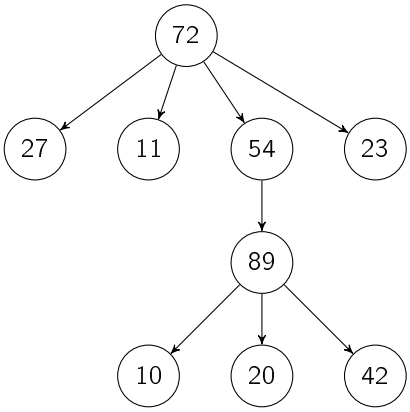
ii)
```
g [] y z        = [z]
g (x : xs) y z = (x + y) : (g xs y z)
```

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

```
exp :: [Int]
exp = (\x f y -> y : filter f x) [4,2,6] (\z -> z > 3) 2
```

c) Implementieren Sie die Funktion `last :: [a] -> a`, die das letzte Element einer Liste zurückgibt. Beispielsweise soll für `[1, 2, 3]` der Wert 3 zurückgegeben werden. Für leere Listen soll das Ergebnis nicht definiert sein.

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



Wir repräsentieren (nicht-leere) solche Bäume in Haskell mit dem folgenden Datentyp:

```
data MTree = Node Int [MTree]
```

Der Baum aus der Graphik wird wie folgt dargestellt:

```
Node 72 [
 Node 27 [],
 Node 11 [],
 Node 54 [
  Node 89 [
   Node 10 [],
   Node 20 [],
   Node 42 []]],
 Node 23 []]
```

i) Implementieren Sie die Funktion `rightmostLeaf :: MTree -> Int`, die den Wert des rechtesten, untersten Blattes 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.

Hinweise:

- Verwenden Sie `last` aus Aufgabenteil c).

ii) Implementieren Sie die Methode `sumTree :: MTree -> Int` so, dass sich die Summe eines Baums ergibt, indem man den Wert des aktuellen Knotens und die Summe aller Kinder addiert. Für den Beispielbaum aus der Graphik ergibt sich also `72 + 27 + 11 ... + 23 = 348`.

Verwenden Sie dazu die vordefinierten Funktionen `map` und `sum`. Dabei berechnet `sum` die Summe einer Liste von Integer-Werten. **Verwenden Sie nicht** `+`.

iii) Wandeln Sie die Definition des Datentyps `MTree` so ab, dass nicht nur `Int`-Werte in den Knoten gespeichert werden können. Verwenden Sie dafür einen Typparameter `a`, der den Typ der Werte in den Knoten angibt.

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

ii) $g(X, Y, X), g(c(Y), c(Z), Z)$

iii) $h(X, b, X), h(Y, Y, a)$

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

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

i) Erstellen Sie für das Programm P den Beweisbaum zur Anfrage `?- p(a, U).` bis zur Tiefe 4 (die Wurzel hat dabei Tiefe 1). Markieren Sie Pfade, die zu einer unendlichen Auswertung führen, mit ∞. Sie dürfen Substitutionen, welche keinen Einfluss auf die Belegung von U haben, weglassen.

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

c) In diesem Aufgabenteil betrachten wir Algorithmen auf Listen. In *Prolog* bezeichnet `[]` die leere Liste und `[X|XS]` ist die Liste, bei der das Element `X` vorne in die Liste `XS` eingefügt wurde.

Weiterhin verwenden wir die Peano-Notation von natürlichen Zahlen. Also wird die Zahl 0 durch den Term `0` dargestellt und der Nachfolger einer natürlichen Zahl `X` wird durch den Term `s(X)` repräsentiert. Damit wird z.B. die Zahl 2 als `s(s(0))` dargestellt.

i) Schreiben Sie ein dreistelliges Prädikat `del`, welches im ersten Argument ein Listenelement `X` und im zweiten Argument eine Liste `XS` erhält. Dieses Prädikat berechnet dann im dritten Argument eine Liste, welche aus der List `XS` entsteht, in dem man ein beliebiges Vorkommen des Elements `X` entfernt. Durch Backtracking sollen so alle Möglichkeiten, ein Element zu löschen, gefunden werden. Der Aufruf `del(2,[3,2,1,2,3],ZS)` führt z.B. zu den Antwortsubstitutionen `ZS = [3,1,2,3]` und `ZS = [3,2,1,3]`.

ii) Schreiben Sie ein dreistelliges Prädikat `delN`, welches im ersten Argument eine natürliche Zahl `N` in Peano-Notation und im zweiten Argument eine Liste `XS` erhält. Dieses Prädikat berechnet dann im dritten Argument eine Liste, welche aus der Liste `XS` entsteht, indem man `N` beliebige Elemente entfernt. Durch Backtracking sollen alle solchen Listen gefunden werden. Der Aufruf `delN(s(s(0)),[1,2,3],ZS)` führt z.B. zu den Antwortsubstitutionen `ZS = [3]`, `ZS = [2]` und `ZS = [1]`, wobei die gleichen Substitutionen auch mehrfach auftreten dürfen.

Hinweis: Verwenden Sie das Prädikat `del` aus Teil i). Überlegen Sie sich, wie man `del` aufrufen muss, damit es ein beliebiges Element löscht.

iii) Schreiben Sie ein zweistelliges Prädikat `min`, welches im ersten Argument eine Liste `XS` von Integern erhält und im zweiten Argument das minimale Element in der Liste `XS` berechnet. In diesem Aufgabenteil werden vordefinierte Integer verwendet (und **nicht** die Peano-Notation). Beispielsweise ergibt der Aufruf `min([3,2,1,2,3],Y)` die Antwortsubstitution `Y = 1`. Falls das erste Argument die leere Liste ist, soll das Prädikat fehlschlagen (und keinen Programmfehler verursachen).