Prof. Dr. J. Giesl
S. Dollase, M. Hark, D. Korzeniewski

Klausur Programmierung WS 2018/2019

Vorname: _____

Nachname: _____

Matrikelnummer: _____

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

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

	Anzahl Punkte	Erreichte Punkte
Aufgabe 1	**15**	
Aufgabe 2	**15**	
Aufgabe 3	**19**	
Aufgabe 4	**33**	
Aufgabe 5	**20**	
Aufgabe 6	**18**	
Summe	**120**	

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.
- Bei **Täuschungsversuchen** wird die Klausur mit **0 Punkten** bewertet.
- Geben Sie am Ende der Klausur **alle Blätter zusammen mit den Aufgabenblättern ab**.

Mit meiner Unterschrift bestätige ich, dass ich mich gesund und in der Lage fühle, an dieser Klausur teilzunehmen.

_____
Unterschrift

Aufgabe 1 (Programmanalyse): (9 + 6 = 15 Punkte)

a) 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 {

    int x = 8;

    public A() {
        x = 6;
    }
    public A(int x) {
    }
    public A(Integer x) {
      this.x = 7;
    }

    int getX() {
        return this.x;
    }

    public int f(int a) {
      return 1;
    }
    public int f(Float a) {
      return 2;
    }
    public int f(double a) {
      return 9;
    }
}
```

```
public class B extends A {

    int x = 10;

    public B(int x) {
      super(Integer.valueOf(12));
      this.x = x;
    }
    public B() {
      super(12);
      this.x = this.x + 1;
    }

    int getX() {
        return this.x;
    }

    public int f(double a) {
      return 3;
    }
    public int f(int a) {
      return 4;
    }
}
```

```
public class M {
    public static void main(String[] args) {
        A a1 = new A();
        System.out.println(a1.x);                    // OUT: [    ]

        B b1 = new B();
        System.out.println(b1.x + " " + ((A)b1).x);// OUT: [    ] [    ]

        A ab = new B(5);
        System.out.println(ab.x + " " + ((A)ab).x);// OUT: [    ] [    ]

        System.out.println(ab.getX());               // OUT: [    ]

        System.out.println(a1.f((long)1));           // OUT: [    ]

        System.out.println(b1.f(Long.valueOf(42))); // OUT: [    ]

        System.out.println(ab.f(Short.valueOf("7")));//OUT: [    ]
    }
}
```

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

```
1  public class C extends A {
2      int y = 12;
3      float a;
4
5      public C() {
6        int value = this.y;
7        super(value);
8      }
9
10     public int f(float a) {
11       return this.y;
12     }
13
14     public int g(int b) {
15       return 17;
16     }
17
18     public int g(Float c) {
19       return 4;
20     }
21
22     public static void main(String[] args) {
23       C c1 = C();
24
25       System.out.println(c1.g((long)23));
26       System.out.println(c1.g(Integer.valueOf(23)));
27     }
28 }
```

Aufgabe 2 (Hoare-Kalkül): (14 + 1 = 15 Punkte)

Gegeben sei folgendes Java-Programm P über den Variablen `a`, `x`, `y` und `res`.

$\langle\, \texttt{a} \geq 0 \,\rangle$ (Vorbedingung)

```
x = a;
y = 1;
res = 0;
while (x > 0) {
    if (x % 2 == 0) {
      x = x / 2;
    } else {
      res = res + y;
      x = (x - 1)/2;
    }
    y = -y;
}
```

$\langle\, 3 \,|\, (\texttt{res - a}) \,\rangle$ (Nachbedingung)

Hinweise:

- Die Programmanweisung `x / 2` berechnet das abgerundete Ergebnis der Ganzzahldivision, d.h. $\lfloor \frac{x}{2} \rfloor$. Die Programmanweisung `x % 2` berechnet den Rest von `x` bei Ganzzahldivision durch `2`, z.B. ist `3 % 2 == 1`.
- Die Schreibweise $3 \,|\, v$ (sprich "3 teilt v") für eine ganze Zahl `v` bedeutet: Es gibt eine ganze Zahl `w`, sodass $v = 3 \cdot w$.
- Bedenken Sie, dass aus $3 \,|\, v$ und $3 \,|\, w$ immer $3 \,|\, (v + w)$ folgt und aus $3 \,|\, v$ immer $3 \,|\, (-v)$ folgt.
- Sie dürfen in beiden Teilaufgaben 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. Klammern dürfen und müssen Sie jedoch eventuell bei der Anwendung der Zuweisungsregel setzen.
- Benutzen Sie die folgenden Wertetabellen zum Finden der Schleifeninvariante. Die Schleifeninvariante enthält unter anderem die Aussage `3|exp` für einen geeigneten Ausdruck `exp`. Dieser Ausdruck `exp` enthält alle Programmvariablen und setzt sich aus den Ausdrücken in der Tabelle zusammen. Beim Verlassen der Schleife hat `exp` den Wert `res-a`, vor Betreten der Schleife den Wert `0`.

Anzahl Schleifendurchläufe	a	x	y	res	res-a	x·y
0	7	7	1	0	-7	7
1	7	3	-1	1	-6	-3
2	7	1	1	0	-7	1
3	7	0	-1	1	-6	0

Anzahl Schleifendurchläufe	a	x	y	res	res-a	x·y
0	6	6	1	0	-6	6
1	6	3	-1	0	-6	-3
2	6	1	1	-1	-7	1
3	6	0	-1	0	-6	0

- Wenn Sie die Konsequenzregel anwenden, müssen Sie nicht beweisen, warum aus der oberen Zusicherung die untere Zusicherung folgt.

Name: **Matrikelnummer:**

a) Vervollständigen Sie die Verifikation des Algorithmus P auf dieser Seite 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.

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

$\langle$ _____ $\rangle$

```
x = a
```

$\langle$ _____ $\rangle$

```
y = 1
```

$\langle$ _____ $\rangle$

```
res = 0;
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
while (x > 0) {
```

$\langle$ _____ $\rangle$

```
    if (x % 2 == 0) {
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
        x = x / 2;
```

$\langle$ _____ $\rangle$

```
    } else {
```

$\langle$ _____ $\rangle$

$\langle$ _____ $\rangle$

```
        res = res + y;
```

$\langle$ _____ $\rangle$

```
        x = (x - 1) / 2;
```

$\langle$ _____ $\rangle$

```
    }
```

$\langle$ _____ $\rangle$

```
    y = -y;
```

$\langle$ _____ $\rangle$

```
}
```

$\langle$ _____ $\rangle$

$\langle 3 \mid (\mathtt{res} - \mathtt{a}) \rangle$

b) Geben Sie eine gültige Variante an, mit deren Hilfe die Terminierung des Algorithmus P bewiesen werden kann. Sie brauchen den eigentlichen Terminierungsbeweis nicht durchzuführen – die Angabe einer geeigneten Variante für die `while`-Schleife genügt.

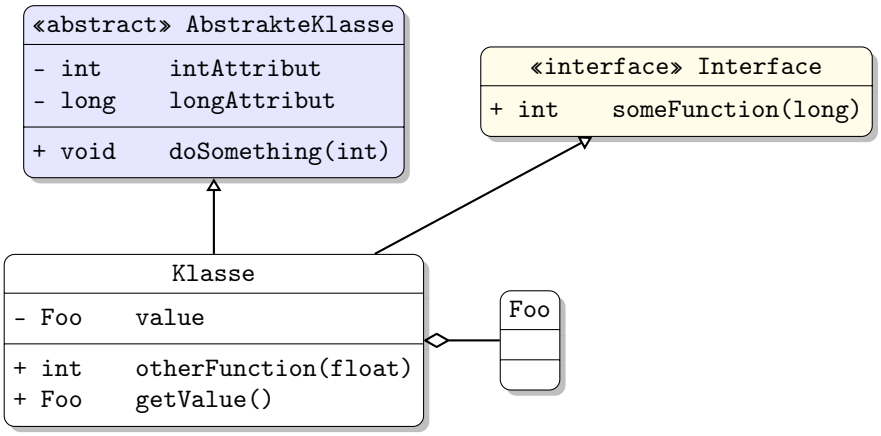
Aufgabe 3 (Klassenhierarchie): (11 + 8 = 19 Punkte)

Ziel dieser Aufgabe ist die Erstellung einer Klassenhierarchie zur Verwaltung von Raumfahrzeugen.

- Ein Antriebsmodul (für ein Raumfahrzeug) hat als wichtigste Eigenschaft den maximalen Schub. Der Wert des maximalen Schubs wird inklusive Nachkommastellen gespeichert.
- Feststoffbooster und Raketenstufen sind spezielle Antriebsmodule. Falconbooster sind spezielle Raketenstufen.
- Ein Raumfahrzeug hat eine Nutzlast und beliebig viele Antriebsmodule. Außerdem kann jedes Raumfahrzeug starten und bietet daher eine entsprechende Methode ohne Parameter oder Rückgabewert an.
- Frachtkapseln sind Raumfahrzeuge, bei denen das Volumen des Frachtraums interessant ist.
- Raumschiffe sind Raumfahrzeuge, die Menschen transportieren können. Daher soll bei Raumschiffen gespeichert werden, wie viele Passagiere sie aufnehmen können.
- Jedes Raumfahrzeug ist entweder eine Frachtkapsel oder ein Raumschiff.
- Spaceshuttles und Sojus sind spezielle Raumschiffe. Spaceshuttles bieten eine Methode `void frachtraumOeffnen()` und Sojus bietet eine Methode `void solarpaneleAusfahren()` an.
- Alle Raumschiffe, sowie Feststoffbooster und Falconbooster sind wiederverwendbar. Diese bieten alle die Methode `float kostenErmitteln()` an, die zurück gibt, wie viel Prozent der Herstellungskosten eine Reparatur kosten würde. Außerdem haben alle wiederverwendbaren Raumfahrzeuge und Antriebsmodule eine Methode `void reparieren()`.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Sachverhalte. 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 dargestellt, in dem der Name der Klasse sowie alle in der Klasse definierten bzw. überschriebenen Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \multimap\!\triangleright A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist). Der Pfeil $B \multimap\!\diamond A$ bedeutet, dass A ein Objekt vom Typ B benutzt. Benutzen Sie `-`, um `private` abzukürzen, und `+` für alle anderen Sichtbarkeiten (wie z. B. `public`). Fügen Sie Ihrem Diagramm keine Kästen für vordefinierte Klassen wie `String` hinzu.

b) Schreiben Sie eine Java-Methode
`public static void raumfahrtprogramm(Raumfahrzeug[] raumfahrzeuge)`. Für jedes Raumfahrzeug soll Folgendes durchgeführt werden: Zunächst soll das Raumfahrzeug gestartet werden. Falls es ein Sojus ist, so sollen danach die Solarpanele ausgefahren werden. Danach sollen für das Raumfahrzeug (falls es wiederverwendbar ist) und für alle seine wiederverwendbaren Antriebsmodule jeweils die Kosten einer Reparatur ermittelt werden. Sofern diese Kosten bei höchstens `90` (Prozent) liegen, soll die Repatur ausgeführt werden.

Gehen Sie davon aus, dass es zu jedem Attribut geeignete Selektoren gibt und alle Klassen einen Konstruktor besitzen, der für jedes Attribut einen Parameter hat. Eine Klasse `A` mit Attributen `int x` und `float y` hätte beispielsweise einen Konstruktor `A(int x, float y)`.

Hinweise:

- Gehen Sie davon aus, dass alle vorkommenden Arrays weder `null` sind noch `null` enthalten.
- Sie dürfen Hilfsmethoden schreiben.

Aufgabe 4 (Programmierung in Java): (7 + 5 + 2 + 11 + 8 = 33 Punkte)

In dieser Aufgabe wollen wir einfache boolesche Ausdrücke in Java modellieren. Ein einfacher boolescher Ausdruck besteht aus zwei booleschen Ausdrücken `left` und `right`, die durch den logischen Operator $\wedge$ (für die Konjunktion) oder den logischen Operator $\vee$ (für die Disjunktion) verknüpft sind oder er ist ein Atom, d.h. eine Variable, die durch `<`,`>`,`<=` oder `>=` mit einer Gleitkommazahl verglichen wird. Ein Beispiel für einen solchen Ausdruck ist $((x <= 3.5) \wedge (y > 1.7)) \vee (z < -2.0)$. Die Ausdrücke werden als Bäume modelliert. Ein Blatt eines Baumes repräsentiert ein Atom. Innere Knoten repräsentieren die Konjunktion (wenn `isConjunction == true`) bzw. die Disjunktion (wenn `isConjunction == false`) zweier einfacher boolescher Ausdrücke `left` und `right`. Das `enum Comparator` repräsentiert die Vergleichsoperatoren: `GT` entspricht `>`, `GE` `>=`, `LT` `<` und `LE` `<=`. So wird z.B der obige Ausdruck durch den Baum in Abb. 1 dargestellt. Objekte der Klasse `InnerNode` werden als Kreise dargestellt, in deren Zentrum der Wert des Attributs `isConjunction` steht. Objekte der Klasse `Leaf` werden als Rechtecke dargestellt, in denen von links nach rechts die Werte der Attribute `comp`, `var` und `bound` zu finden sind.

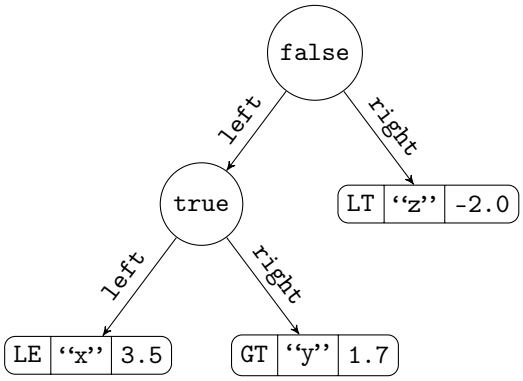


Abbildung 1: Boolescher Ausdruck als Baum

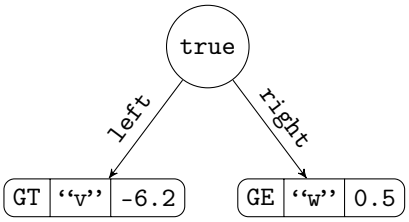


Abbildung 2: Weiterer boolescher Ausdruck

```
public abstract class TreeNode{
  private final TreeNode left;
  private final TreeNode right;

  public TreeNode getLeft(){
    return this.left;
  }

  public TreeNode getRight(){
    return this.right;
  }
}
```

```
public class InnerNode extends TreeNode{
  private final boolean isConjunction;

  public boolean getIsConjunction(){
    return this.isConjunction;
  }
}
```

```
public enum Comparator{
  GT,GE,LT,LE;

  public String toString(){
    String result = "";
    switch (this){
      case GE:
        result += ">=";
        break;
      case GT:
        result += ">";
        break;
      case LE:
        result += "<=";
        break;
      case LT:
        result += "<";
        break;
    }
    return result;
  }
}
```

```
public class Leaf extends TreeNode{
  private final Comparator comp;
  private final String var;
  private final double bound;

  public Comparator getComp(){
    return this.comp;
  }

  public String getVar(){
    return this.var;
  }

  public double getBound(){
    return this.bound;
  }
}
```

Sie können in den folgenden Aufgaben davon ausgehen, dass bei Objekten der Klasse `InnerNode` die Argumente `left` und `right` **nicht** `null` sind.
Machen Sie in jeder Aufgabe deutlich, welche Teile ihrer Lösung zu welcher der Klassen `TreeNode`, `InnerNode` und `Leaf` gehören. Hierzu reicht es, wenn Sie über den entsprechenden Teil „in der Klasse ...“ schreiben.

a) In der Klassendeklaration der Klasse `TreeNode` sei die Deklaration `public abstract String toString();` gegeben. Implementieren Sie die Methode in den Klassen `InnerNode` und `Leaf` so, dass der Baum als boolescher Ausdruck in Infix Notation ausgegeben wird. Dabei soll für die Konjunktion das Symbol `&&` bzw. für die Disjunktion das Symbol `||` benutzt werden. Rufen wir `toString` auf dem Objekt aus Abb. 1 auf, so erhalten wir den `String`

‘‘`((x<=3.5) && (y>1.7)) || (z<-2.0)`’’.

Rufen wir `toString()` auf dem Objekt aus Abb. 2 auf, so ergibt sich der String

‘‘`(v>-6.2) && (w>=0.5)`’’.

b) In Java ist das folgende Interface vordefiniert:

```
public interface Comparable<T>{

  public int compareTo(T toBeComparedTo);

}
```

Ändern Sie die Deklarationen der Klassen `TreeNode`, `InnerNode` und `Leaf` so ab, dass das Attribut `bound` in der Klasse `Leaf` von einem beliegen Typ `T` ist, der das Interface `Comparable<T>` implementiert.

Get–Methoden müssen Sie **nicht** anpassen, können diese aber in den nachfolgenden Teilaufgaben benutzen, d.h. Sie können davon ausgehen, dass diese entsprechend angepasst wurden.

Die Objekte aus Abb. 1 und 2 gehören somit zur Klasse `InnerNode<Double>`.

Hinweise:

- In dieser Aufgabe sollen **alle** Klassen generisch werden.
- Das Interface `Comparable<T>` hängt generisch vom Typen `T` ab, da der Typ in der Vergleichsmethode `compareTo` benutzt wird. Möchten Sie, dass `T` das Interface `Comparable<T>` implementiert, so schreiben Sie `T extends Comparable<T>` in die entsprechenden Klassenköpfe in Ihrer Lösung.

c) Schreiben Sie eine `Exception`–Klasse `UndefinedVariableException`. Diese soll ein Attribut vom Typ `String` haben (den Variablennamen). Schreiben Sie einen entsprechenden Konstruktor, der den Namen der Variablen als Parameter enthält.

d) In der folgenden Aufgabe benutzen wir das generische Interface `Map<K,V>` aus dem `Collections Framework`. Hierbei ist `K` der Typ der `keys` und `V` der Typ der `values`. Sie können davon ausgehen, dass `import java.util.*;` an den entsprechenden Stellen in den Deklarationen der Klassen `TreeNode`, `InnerNode` und `Leaf` steht.

In der Klasse `TreeNode` sei die Deklaration `public abstract boolean evaluate(Map<String,T> varAssignment) throws UndefinedVariableException;` gegeben. Implementieren Sie die Methode in den Klassen `InnerNode` und `Leaf` so, dass die Variablen mit den entsprechenden Werten aus der `Map varAssignment` instanziiert werden und anschließend der boolesche Ausdruck ausgewertet wird. Kommt in einem Blatt eine Variable vor, der kein oder der Wert `null` zugewiesen wurde, so soll eine `UndefinedVariableException` geworfen werden. Das `String`–Attribut der `UndefinedVariableException` soll der Name der entsprechenden Variable sein.

Rufen wir die Methode `evaluate` auf dem Objekt aus Abb. 1 mit einer `Map` mit den key $\mapsto$ value–Paaren {`"x"` $\mapsto$ `3.0`, `"y"` $\mapsto$ `1.0`, `"z"` $\mapsto$ `-4.0`, `"v"` $\mapsto$ `0.7`} auf, so ergibt sich `true`. Rufen wir die Methode mit demselben Argument hingegen auf dem Objekt aus Abb. 2 auf, so wird eine `UndefinedVariableException` mit Attribut `"w"` geworfen.

Hinweise:

- Eine `Map` speichert zu einem `key` *genau einen* `value`. Sie entspricht also einer Abbildung (engl. „map“) aus der Mathematik.
- Das Interface `Map<K,V>` stellt die Methode `public boolean containsKey(K key)` zur Verfügung. Diese gibt genau dann `true` zurück, wenn zu dem Argument `key` ein entsprechender `value` in der `Map` gespeichert ist.
- Das Interface `Map<K,V>` stellt die Methode `public V get(K key)` zur Verfügung.
  Wenn `containsKey(key) == true`, so liefert diese Methode den zugehörigen `value` zurück.
- Gehen Sie davon aus, dass in jeder Klasse `T`, die das Interface `Comparable<T>` implementiert, die Methode `compareTo` wie folgt implementiert ist. Falls zwei Objekte `t1` und `t2` der Klasse `T` beide *nicht* `null` sind, so gilt:
  - `t1` ist echt kleiner als `t2` $\Rightarrow$ `t1.compareTo(t2) < 0`
  - `t1` ist gleich `t2` $\Rightarrow$ `t1.compareTo(t2) = 0`
  - `t1` ist echt größer als `t2` $\Rightarrow$ `t1.compareTo(t2) > 0`

e) Schreiben Sie in der Klasse `TreeNode` eine Methode `public boolean forAll(List<Map<String,T>> varAssignments)`, die den Wahrheitswert der Aussage „Für jedes Element `e` in `varAssignments` gilt `this.evaluate(e)`“ berechnet. Falls in einer `Map` in der Liste eine Variable nicht oder mit `null` instanziiert ist, so soll `false` zurückgegeben werden.

Hinweise:

- Die leere Aussage ist wahr.
- Benutzen Sie die Methode `evaluate` aus der vorherigen Teilaufgabe. Es ist unerheblich, ob Sie diese selbst implementiert haben.

Aufgabe 5 (Haskell): (4 + 4 + 3 + 2 + 2 + 5 = 20 Punkte)

a) Geben Sie zu den folgenden Haskell-Funktionen `f` und `g` jeweils den allgemeinsten Typ an. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben.

```
f n x y = if n > 0 then y:(f (n - 1) x x) else [False]
```

```
g x y z = z:(g x (x:z) (y++z))
```

b) Bestimmen Sie, zu welchem Ergebnis die Ausdrücke `i` und `j` jeweils auswerten.

```
i :: [Bool]
i = filter (\x->x) (map even [1,2,3,4,5,6])
```

Hinweise:

- Für die vordefinierte Funktion `even::Int -> Bool` ist `even x = True` genau dann, wenn `x` eine gerade ganze Zahl ist.

```
j :: Int
j = (\f g x -> if (g x) then 3 * (f x) else 4 + (f x)) (\x->1) even 8
```

c) Schreiben Sie eine Funktion `removeDuplicates::[Int] -> [Int]` die aus einer gegebenen Liste `xs` eine Liste `ys` macht, sodass `ys` jeden Eintrag aus `xs` **genau einmal** enthält. Die Sortierung der Elemente ist irrelevant. So ergibt der Aufruf `removeDuplicates [1,1,2,1,3,2,1,3]` eine Liste, die genau einmal die ganzen Zahlen `1`,`2` und `3` enthält (also z.B. `[1,2,3]`).

Hinweise:

- Sie dürfen beliebige vordefinierte Funktionen aus der Standardbibliothek verwenden, wie den vordefinierten Ungleich–Operator `(/=)::Int -> Int -> Bool`. Außerdem dürfen Sie Funktionen höherer Ordnung wie `filter` benutzen.

d) Wir verwenden die folgende Datenstruktur `Graph`, um Graphen in Haskell durch eine Liste von Kanten zu respräsentieren.

```
data Graph = Edges [(Int,Int)] deriving Show
```

Die Kanten sind also als Paare vom Typ `Int` dargestellt. Beispielsweise kann der Graph aus Abb. 3 durch folgenden Haskell-Code dargestellt werden:

```
g1::Graph
g1 = Edges [(1,2),(1,4),(1,5),(2,3),(2,4),(3,4),(4,5),(4,6)]
```

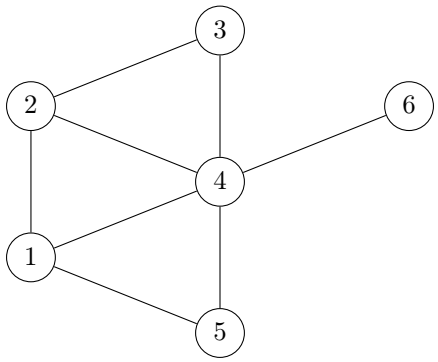


Abbildung 3: Graph des Typs `Graph Int`

Desweiteren ist die folgende Funktion gegeben.

```
flatten::[(a,a)] -> [a]
flatten [] = []
flatten ((m,n):xs) = [m,n] ++ (flatten xs)
```

Der Aufruf `flatten [(1,2),(3,4),(-5,-6)]` ergibt also die Liste `[1,2,3,4,-5,-6]`.

Schreiben Sie eine Haskell–Funktion `knoten::Graph -> [Int]`, die eine Liste der im Graphen enthaltenen Knoten berechnet. Dabei soll jeder enthaltene Knoten genau einmal vorkommen. Der Aufruf `knoten g1` soll also eine Liste zurückgeben, die genau einmal die Werte `1,2,3,4,5` und `6` enthält, also z.B. `[1,2,4,5,3,6]`. Die Sortierung der Liste ist irrelevant.

Hinweise:

- Sie können die Funktionen `flatten` und `removeDuplicates` verwenden. Es ist unerheblich, ob Sie `removeDuplicates` selbst implementiert haben.

e) Schreiben Sie eine Haskell–Funktion `istEnthalten::Int -> Graph -> Bool`. Der Ausdruck `istEnthalten x g` wertet genau dann zu `True` aus, wenn `x` in `knoten g` enthalten ist.

Hinweise:

- Benutzen Sie die Funktion `knoten` aus der vorherigen Aufgabe. Es ist unerheblich, ob Sie diese selbst implementiert haben.
- Sie dürfen beliebige vordefinierte Funktionen aus der Standardbibliothek verwenden, wie den vordefinierten Gleichheits–Operator `(==)::Int -> Int -> Bool` oder die Funktion `length::[a] -> Int`, die die Anzahl der Elemente einer Liste berechnet. Außerdem dürfen Sie Funktionen höherer Ordnung wie `filter` benutzen.

f) Ein *Wald* ist ein Graph, in dem es keine Kreise gibt, d.h. von jedem Knoten gibt es höchstens eine Möglichkeit über Kanten einen anderen Knoten zu erreichen. Der Graph in Abb. 3 ist kein Wald, denn z.B. formen die Knoten `1,2` und `4` einen Kreis. Schreiben Sie eine Haskell–Funktion `spannWald::Graph -> Graph`, die für einen gegebenen Graphen `g` einen Spannwald `w` berechnet, d.h.

- `knoten g == knoten w`
- `w` ist ein Wald.
- Jede Kante in `w` ist auch in `g` enthalten.

Für den Graphen `g1` aus Abb. 3 könnte `spannwald g1` den Graph `Edges [(4,6),(2,3),(1,5),(1,4),(1,2)]` berechnen. Dieser ist in Abb. 4 dargestellt.

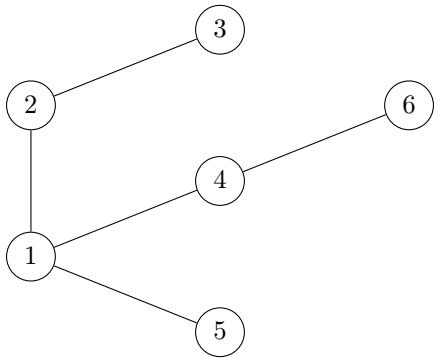


Abbildung 4: Spannwald für den Graphen aus Abb. 3

Hinweise:

- Der leere Graph `Edges []` ist ein Wald. Fügt man eine Kante `(m,n)` in einen Wald `w` ein, der **nicht** sowohl `m` als auch `n` enthält, so ergibt sich wieder ein Wald.
- Benutzen sie die Funktion `istEnthalten` aus der vorherigen Aufgabe. Es ist unerheblich, ob Sie diese selbst implementiert haben.

Aufgabe 6 (Prolog): (2 + 7 + (1 + 2.5 + 2.5 + 3) = 18 Punkte)

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

i) $f(s(X), X, s(Y)), f(s(Y), Z, Z)$

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

b) Gegeben sei folgendes Prolog-Programm P.

```
p(s(s(X)), 0) :- p(X, s(s(0))).
p(s(X), s(Y)) :- p(X, Y).
p(0, s(0)).
```

Erstellen Sie für das Programm P den Beweisbaum zur Anfrage "`?- p(R, s(0)).`" bis zur Höhe 5 (die Wurzel hat dabei die Höhe 1). Markieren Sie Pfade, die zu einer unendlichen Auswertung führen, mit ∞ und geben Sie alle Antwortsubstitutionen zur Anfrage "`?- p(R, s(0)).`" an, die im Beweisbaum bis zur Höhe 5 enthalten sind. Geben Sie außerdem zu jeder dieser Antwortsubstitutionen an, ob sie von Prolog gefunden wird. Welche der beiden Regeln muss ans Ende des Programms verschoben werden, damit Prolog bei obiger Anfrage alle Antwortsubstitutionen bis zur Höhe 5 findet?

c) In dieser Aufgabe geht es darum, Terminkonflikte in einem Stundenplan zu finden. Ein Termin besteht aus einem Titel, einem Wochentag und einer Uhrzeit, wobei nur die Stunde relevant ist, da alle Termine in einem festen Stundenraster liegen. In Prolog stellen wir einen Termin beispielsweise durch den Term `termin('VL Progra', 'MO', 10)` dar. Ziel dieser Aufgabe ist es, zu einer Liste von Terminen die Liste von Terminkonflikten zu finden.

Hinweise:

- Sie dürfen in allen Teilaufgaben Prädikate aus den vorherigen Teilaufgaben verwenden, auch wenn Sie diese Prädikate nicht implementiert haben.
- Sie dürfen sich Hilfsprädikate definieren.

i) Implementieren Sie ein Prädikat `conflict` mit Stelligkeit 2 in Prolog, das wahr ist, wenn beide Argumente Termine sind und die Termine den gleichen Wochentag und die gleiche Stunde haben. Beispielsweise soll
`conflict(termin('Progra', 'MO', 10), termin('Ausschlafen', 'MO', 10))` wahr sein. Die Anfragen `conflict(termin('Progra', 'MO', 10), termin('AfI', 'MI', 16))` und `conflict (0, termin('Ausschlafen', 'MO', 10))`
sollen hingegen falsch sein.

ii) Implementieren Sie ein Prädikat `combineWithAll` mit Stelligkeit 3 in Prolog. Beim Aufruf von `combineWithAll(`X`,` *list*`, ZS)` soll die Liste aller Paare von X und einem Element aus *list* berechnet werden. Ein Paar wird dabei als Liste mit genau zwei Elementen dargestellt. Beispielsweise soll `?- combineWithAll(1, [2,3,4], ZS)` als erste Antwort
`ZS= [[1,2][1,3][1,4]]` liefern.

iii) Implementieren Sie ein Prädikat `allPairs` mit Stelligkeit 2 in Prolog, das wahr ist, wenn das zweite Argument eine Liste aller Paare aus Elementen der ersten Liste enthält. Beispielsweise soll die Anfrage `?- allPairs([1,2,3], PS)` als erste Antwort `PS = [[1,2],[1,3],[2,3]]` zurückliefern. (Die Reihenfolge der Paare ist jedoch nicht wichtig.)

Hinweise:

- Sie dürfen das vordefinierte Prädikat `append` mit Stelligkeit 3 nutzen. Dies kann genutzt werden, um Listen zu verketten. Beispielsweise ist `append([1,2], [3,4], [1,2,3,4])` wahr.
- Außerdem dürfen Sie das Prädikat `combineWithAll` aus ii) verwenden.

iv) Implementieren Sie ein Prädikat `getConflicts` mit Stelligkeit 2 in Prolog, das zu einer Liste von Terminen die Liste aller Terminkonflikte berechnet. Beispielsweise soll ein Aufruf von `getConflicts([ termin('Progra', 'MO', 10), termin('AfI', 'Mi', 16), termin('Ausschlafen', 'MO', 10) ], Conflicts)` als erste Antwort
`Conflicts= [[termin('Progra', 'MO', 10), termin('Ausschlafen', 'MO', 10)]]` liefern.

Hinweise:

- Berechnen Sie zunächst die Liste aller Paare von Terminen aus der Liste mit Hilfe des Prädikats `allPairs`. Anschließend prüfen Sie mit `conflict` für jedes Paar, ob es einen Terminkonflikt darstellt.