Prof. Dr. J. Giesl

D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Aufgabe 1 (Programmanalyse): (9 + 4 = 13 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 sealed class A permits B {
  public int n = 17;
  public static int k = -1;

  public A(int m) {
      n += m;
  }

  public A f(int m) {
    A a = this;
    n = m;
    return a;
  }

  public static int g(int m) {
      int ac = k;
      k = m;
      return ac;
  }
}
```

```
public final class B extends A {
  public int n = 2;

  public B(int n) {
    super(0);
    this.n = 5;
  }
}
```

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 {
  public int n = 0;

  public C() {
    super(0);
    super(1);
    n = 1.0;
  }
}
```

Lösung:

a)
```
public class M {
  public static void main(String[] args) {

    A a1 = new A(18);
    System.out.println(a1.n);                          // OUT: [ 35]

    A a2 = a1.f(1);
    System.out.println(a1.n + " " + a2.n);             // OUT: [ 1 ] [ 1 ]

    int n = a1.g(100);
    System.out.println(a1.k + " " + n);                // OUT: [100] [ -1]
```

```
    System.out.println(a2.k + " " + A.k);           // OUT: [100] [100]

    B b = new B(3);
    System.out.println(b.n + " " + ((A)b).n);       // OUT: [ 5 ] [ 17]
  }
}
```

In der Musterlösung werden konkrete Zahlenwerte genutzt. Die Klausur war an diesen Stellen jedoch parametrisiert, sodass Sie hier die entsprechenden Werte aus Ihrer persönlichen Klausur einsetzen müssen.

b)

- Zeile 1: Die mit `final` versehene Klasse `B` darf nicht als Oberklasse genutzt werden.
- Zeile 5: `super(...)` muss der erste Aufruf in einem Konstruktor sein (und darf deshalb nur einmal vorkommen).
- Zeile 7: Das Attribut `n` ist vom Typ `int`, aber `1.0` ist vom Typ `double`. Von `double` zu `int` kann aber nicht automatisch umgewandelt werden.

Aufgabe 2 (Hoare-Kalkül): (12 + 1 = 13 Punkte)

Gegeben sei folgendes Java-Programm P über den `int`-Variablen `n`, `i`, `p` und `res`:

$\langle\, \mathtt{n} \geq 0 \,\rangle$ (Vorbedingung)

```
i = 0;
p = 1;
res = 0;
while (i < 2 * n) {
    if (i % 2 == 0) {
      res = res + p;
    }
    p = 3 * p;
    i = i + 1;
}
```

$\langle\, \mathtt{res} = gs(2 \cdot \mathtt{n} - 1) \,\rangle$
$\langle\, \mathtt{res} = \frac{1}{8}(9^{\mathtt{n}} - 1) \,\rangle$ (Nachbedingung)

a) Vervollständigen Sie die folgende Verifikation des Programms 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. Geben Sie bei der Anwendung der Bedingungsregel (zur Behandlung der „if"-Anweisung) an, welche Formel der Art „$\varphi \land \neg B \Rightarrow \psi$" hierbei gezeigt werden muss. Hierzu können Sie auch die Nummerierung der Zeilen verwenden. Sie müssen diese Formel hier *nicht* beweisen.

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.
- Für alle $k \in \mathbb{N}$ sei $gs(k) = \sum\limits_{\substack{j=0 \\ j \bmod 2 = 0}}^{k} 3^j$. Insbesondere gilt also $gs(-1) = 0$.

  Sie können die Funktion gs in Ihren Zusicherungen benutzen, um Schreibarbeit zu sparen.
- Sie können verwenden, dass für alle $\mathtt{n} \in \mathbb{N}$ gilt:

$$gs(2 \cdot \mathtt{n} - 1) = \sum_{\substack{j=0 \\ j \bmod 2 = 0}}^{2\cdot\mathtt{n}-1} 3^j = \sum_{\substack{j=0 \\ j \bmod 2 = 0}}^{2\cdot\mathtt{n}-2} 3^j = \frac{1}{8}(9^{\mathtt{n}} - 1)$$

b) Geben Sie eine Variante an, mit der man die Terminierung des Programms P beweisen könnte. Sie müssen den Terminierungsbeweis aber *nicht* durchführen (d.h. hier brauchen Sie den Hoare-Kalkül nicht zu verwenden).

Lösung:

a)

$\langle \mathtt{n} \geq 0 \rangle$
$1 : \langle \mathtt{n} \geq 0 \land 0 = 0 \land 1 = 1 \land 0 = 0 \rangle$

```
i = 0;
```

$2 : \langle \mathtt{n} \geq 0 \land \mathtt{i} = 0 \land 1 = 1 \land 0 = 0 \rangle$

```
p = 1;
```

$3 : \langle \mathtt{n} \geq 0 \land \mathtt{i} = 0 \land \mathtt{p} = 1 \land 0 = 0 \rangle$

```
res = 0;
```

$4 : \langle \mathtt{n} \geq 0 \land \mathtt{i} = 0 \land \mathtt{p} = 1 \land \mathtt{res} = 0 \rangle$
$5 : \langle \mathtt{res} = gs(\mathtt{i} - 1) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} \leq 2 \cdot \mathtt{n} \rangle$

```
while (i < 2 * n) {
```

$6 : \langle \mathtt{res} = gs(\mathtt{i} - 1) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} \leq 2 \cdot \mathtt{n} \land \mathtt{i} < 2 \cdot \mathtt{n} \rangle$
$7 : \langle \mathtt{res} = gs(\mathtt{i} - 1) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} < 2 \cdot \mathtt{n} \rangle$

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

$8 : \langle \mathtt{res} = gs(\mathtt{i} - 1) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} < 2 \cdot \mathtt{n} \land \mathtt{i} \mod 2 = 0 \rangle$
$9 : \langle \mathtt{res} + \mathtt{p} = gs(\mathtt{i}) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} < 2 \cdot \mathtt{n} \rangle$

```
        res = res + p;
```

$10 : \langle \mathtt{res} = gs(\mathtt{i}) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} < 2 \cdot \mathtt{n} \rangle$

```
    }
```

$11 : \langle \mathtt{res} = gs(\mathtt{i}) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} < 2 \cdot \mathtt{n} \rangle$
$12 : \langle \mathtt{res} = gs((\mathtt{i} + 1) - 1) \land 3 \cdot \mathtt{p} = 3^{\mathtt{i}+1} \land \mathtt{i} + 1 \leq 2 \cdot \mathtt{n} \rangle$

```
    p = 3 * p;
```

$13 : \langle \mathtt{res} = gs((\mathtt{i} + 1) - 1) \land \mathtt{p} = 3^{\mathtt{i}+1} \land \mathtt{i} + 1 \leq 2 \cdot \mathtt{n} \rangle$

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

$14 : \langle \mathtt{res} = gs(\mathtt{i} - 1) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} \leq 2 \cdot \mathtt{n} \rangle$

```
}
```

$15 : \langle \mathtt{res} = gs(\mathtt{i} - 1) \land \mathtt{p} = 3^{\mathtt{i}} \land \mathtt{i} \leq 2 \cdot \mathtt{n} \land \neg(\mathtt{i} < 2 \cdot \mathtt{n}) \rangle$
$\langle \mathtt{res} = gs(2 \cdot \mathtt{n} - 1) \rangle$
$\langle \mathtt{res} = \frac{1}{8}(9^{\mathtt{n}} - 1) \rangle$

Für die Anwendung der Bedingungsregel muss die folgende Implikation gezeigt werden:

- Zeile 7 *und* $\neg(\mathtt{i} \mod 2 = 0)$ *impliziert* Zeile 11.

b) Wir wählen als Variante $V = 2 \cdot \mathtt{n} - \mathtt{i}$.

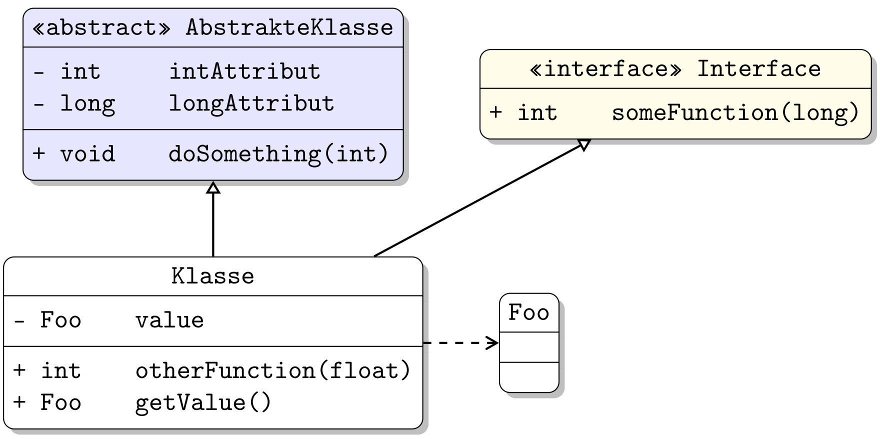
Aufgabe 3 (Klassenhierarchie): (10 + 6 = 16 Punkte)

Ziel dieser Aufgabe ist die Erstellung einer Hierarchie zum Umgang mit verschiedenen Haushaltsgeräten.

- Haushaltsgeräte haben einen Namen. Jedes Haushaltsgerät ist stets auch von mindestens einem weiteren (Unter-)Typ.
- Ein Staubsauger ist ein Haushaltsgerät, welches ein Attribut `beutelVoll` vom Typ `boolean` besitzt.
- Ein Küchengerät ist ein Haushaltsgerät, welches ein Attribut vom Typ `boolean` besitzt, um die Sauberkeit des Gerätes zu speichern. Jedes Küchengerät ist stets auch von mindestens einem weiteren (Unter-)Typ.
- Ein Backofen ist ein Küchengerät mit einem weiteren Attribut, welches angibt, ob der Backofen umluftfähig ist oder nicht.
- Ein Wasserkocher ist ein Küchengerät mit einem zusätzlichen Gleitkomma-Attribut zum Speichern des Wasserkochervolumens in Litern.
- Ein Kaffeebereiter ist ein Küchengerät, welches ein Gleitkomma-Attribut zum Speichern des Volumens des zubereiteten Kaffees in Litern bereitstellt. Zusätzlich kann ein Kaffeebereiter mittels einer entsprechenden Methode `zubereiten` zum Kochen von Kaffee verwendet werden. Hierzu werden keine weiteren Informationen benötigt und es wird nichts zurückgegeben.
- Eine Moka-Kanne, nicht zu verwechseln mit dem türkischen Kaffee *Mokka*, ist ein Kaffeebereiter. Neben der normalen Zubereitungsart, bei welcher die Moka-Kanne (wie jeder Kaffeebereiter) mit kaltem Wasser befüllt wird, gibt es für jede Moka-Kanne außerdem noch eine spezielle Methode zur schnelleren Zubereitung, bei der die Moka-Kanne mit bereits kochendem Wasser befüllt wird. Hierzu wird ein Wasserkocher benötigt, es wird jedoch nichts zurückgegeben. Diese Methode soll unter dem gleichen Namen wie die normale Zubereitungsmethode von Kaffeebereitern verfügbar sein.
- Ein Vollautomat ist ebenfalls ein Kaffeebereiter, welcher überdies eine Methode zum Berechnen der Tage bis zur nächsten Reinigung besitzt. Hierzu sind keine weiteren Informationen notwendig, es wird jedoch eine ganze Zahl zurückgegeben.
- Backöfen und Vollautomaten sind vorheizbar und stellen daher eine Methode `void vorheizen()` zur Verfügung.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Klassen. 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:



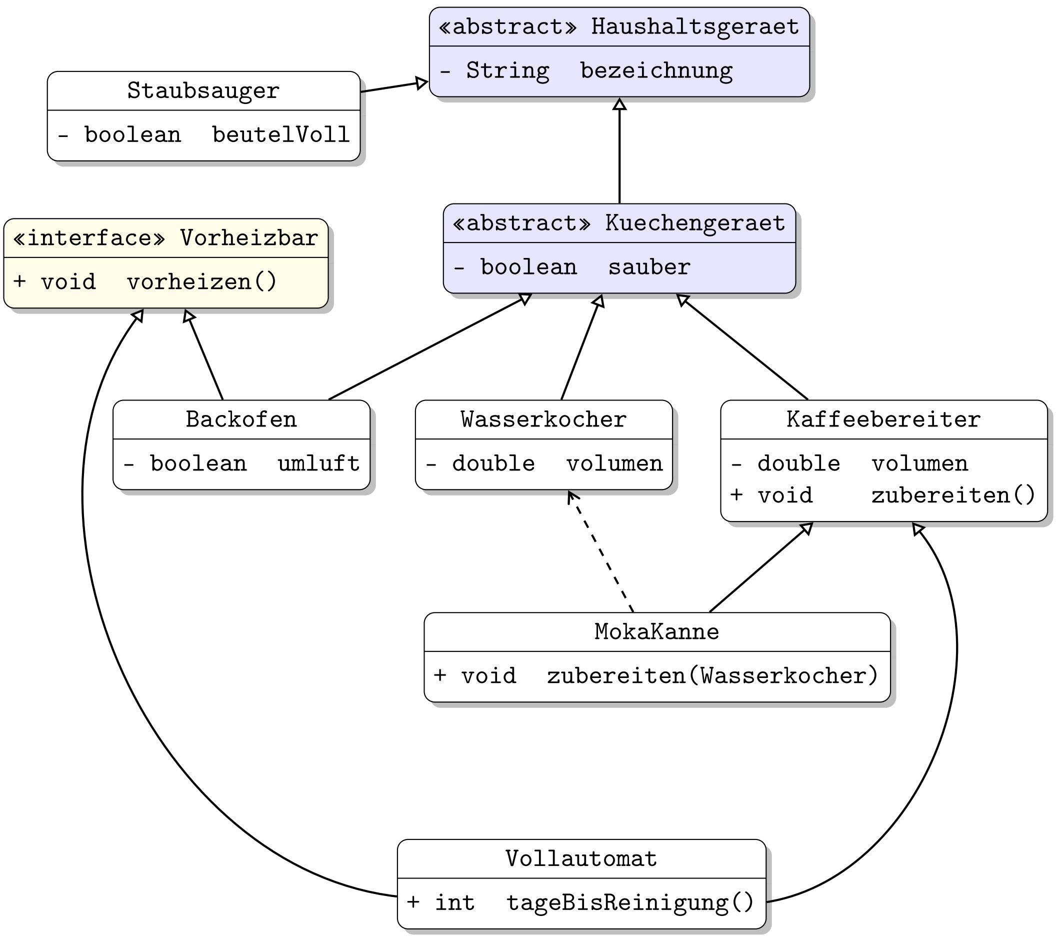
Eine Klasse wird hier durch einen Kasten dargestellt, in dem der Name der Klasse sowie alle in der Klasse definierten Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Überschriebene Methoden müssen nicht angegeben werden. Weiterhin bedeutet der Pfeil $B \mathrel{-\!\!\triangleright} A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist). Der Pfeil $A \dashrightarrow B$ bedeutet, dass A den Typ B in den Typen seiner Attribute oder in den Ein- oder Ausgabeparametern seiner Methoden verwendet. 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) Johanna plant ein großes Mittagsessen, für welches ihre Haushaltsgeräte vorgeheizt werden müssen. Schreiben Sie außerhalb der in Teilaufgabe (a) modellierten Hierarchie eine `Java`-Methode `vorbereiten`. Beim Aufruf von `vorbereiten(geraete)` werden alle Haushaltsgerät im Array `geraete` vorgeheizt, soweit dies möglich ist. Zusätzlich möchte Johanna den erforderlichen Kaffee bereits im Voraus zubereiten. Dazu ruft sie die entsprechende Methode aller in `geraete` enthaltenen und zur Kaffeebereitung geeigneten Geräte auf. Auf welche Art sie die Moka-Kannen zur Kaffeebereitung einsetzt, ist *unerheblich*.

```
public void vorbereiten(Haushaltsgeraet[] geraete) {
```

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:



b)
```
public void vorbereiten(Haushaltsgeraet[] geraete) {
```

```
        for (Haushaltsgeraet h : geraete) {
            if (h instanceof Vorheizbar v) {
                v.vorheizen();
            }
            if (h instanceof Kaffeebereiter k) {
                k.zubereiten();
            }
        }
    }
```

Aufgabe 4 (Programmierung in Java): (6 + 13 + 7 + 8 + 4 = 38 Punkte)

In dieser Aufgabe entwerfen wir eine Datenstruktur, um die Verästelungen eines Baums darzustellen und dessen Wachstum zu simulieren. Dazu nutzen wir die Klasse `Branch`.

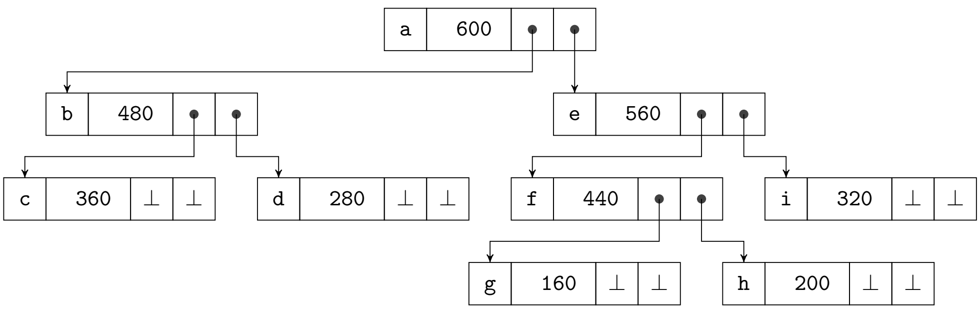
```
public class Branch {
    private int length = 40;
    private Branch left = null;
    private Branch right = null;
}
```

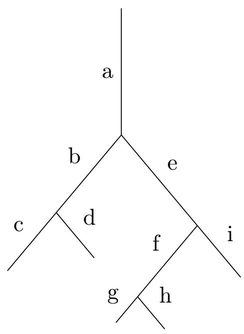
Ein `Branch`-Objekt stellt einen Ast dar. Es enthält die Länge des Asts in cm im Attribut `length`. Ein `Branch`-Objekt kann entweder ein Endstück oder ein Mittelstück sein. Ist es ein Endstück, dann haben beide Attribute `left` und `right` den Wert `null`. Ist es ein Mittelstück, dann hat keins der Attribute `left` oder `right` den Wert `null`. Es ist nie der Fall, dass nur eins dieser beiden Attribute den Wert `null` hat. (Die einzige Ausnahme von dieser Regel tritt auf, wenn ein Endstück zu einem Mittelstück umgewandelt wird, oder umgekehrt, denn in diesem Fall können die beiden Attribute natürlich nur nacheinander neu zugewiesen werden.)

Ist ein `Branch`-Objekt `x` ein Mittelstück, welches die beiden `Branch`-Objekte `l` und `r` in den Attributen `left` und `right` enthält, so nennen wir `x` den Vorgänger von `l` und `r`.
Jeder Ast mit einem Vorgänger startet an dessen Ende, d.h. es gibt keine Äste, die z.B. aus der Mitte eines vorigen Asts herauswachsen.

Beispiel:

In der folgenden Grafik ist beispielhaft eine Verästelung dargestellt.





Links sehen wir neun `Branch`-Objekte `a` bis `i`, dargestellt durch je vier nebeneinander liegende Rechtecke. Das linke Rechteck enthält dabei immer einen Bezeichner, damit wir in den Beispielen der folgenden Teilaufgaben auf die hier dargestellten Objekte verweisen können. Insbesondere sollte Ihre Lösung die hier genannten Beispielobjekte nicht explizit erwähnen. Das zweite Rechteck enthält den Wert des Attributs `length`. Das vorletzte bzw. letzte Rechteck enthält den Wert des Attributs `left` bzw. `right`. Der Wert `null` wird durch ⊥ dargestellt.
Rechts sehen wir eine maßstabsgetreue grafische Repräsentation der links dargestellten Verästelung. Hier ist jeder Ast durch eine Linie dargestellt, deren Länge der links beschriebenen Länge entspricht.

Hinweise:

- Sie dürfen in allen Teilaufgaben davon ausgehen, dass nur auf azyklischen Verästelungen gearbeitet wird (d.h. man erreicht keinen Zyklus, wenn man jeweils einem beliebigen Attribut des `Branch`-Objekts folgt). Außerdem dürfen Sie davon ausgehen, dass es für jeden Ast höchstens einen Vorgänger gibt. Daher existiert zwischen zwei `Branch`-Objekten höchstens ein Pfad.
- Achten Sie darauf, bei allen Teilaufgaben nicht nur die Beispiele korrekt zu behandeln, sondern den allgemeinen Fall zu lösen.
- Sie dürfen in allen Teilaufgaben Klassen und Methoden aus vorherigen Teilaufgaben verwenden, auch wenn Sie diese nicht implementiert haben.
- Sie dürfen eigene Hilfsmethoden implementieren und nutzen.
- Es ist nicht erlaubt, von `Java` vorgegebene Methoden zu nutzen. Ausnahmen dieser Regel finden sich in den Hinweisen der Teilaufgaben.

a) Implementieren Sie in der Klasse `Branch` die Methode `countBranches`, welche die Anzahl der vom aktuellen Ast erreichbaren Äste zurückgibt. Dabei sollen sowohl der aktuelle Ast als auch alle Mittel- und Endstücke mitgezählt werden.

Beispiele:

- Der Ausdruck `a.countBranches()` wird zu 9 ausgewertet.
- Der Ausdruck `c.countBranches()` wird zu 1 ausgewertet.
- Der Ausdruck `e.countBranches()` wird zu 5 ausgewertet.

```
public int countBranches() {
```

b) Implementieren Sie in der Klasse `Branch` die Methode `grow`, welche den aktuellen `Branch` sowie alle von ihm erreichbaren Äste wachsen lässt. Entsteht bei diesem Wachstum ein neuer Ast, so soll auch dieser wachsen.

Im Folgenden nennen wir das `Branch`-Objekt, auf dem die `grow`-Methode aufgerufen wurde, den *Stamm*. Für jeden Ast berechnet sich die *Entfernung von der Wurzel*, indem die Entfernung von der Wurzel des Vorgängers um die Länge des aktuellen Asts erhöht wird. Für den Stamm ist die Entfernung von der Wurzel gleich der Länge des Stamms.

Wenn ein Ast wächst, passiert Folgendes: Die Länge des Asts erhöht sich um einen zufälligen Wert von x%, mit $10 \leq x < 50$. Falls der Ast anschließend mehr als 2000cm (20m) von der Wurzel entfernt ist, so kann er nicht mehr mit Wasser versorgt werden. Es wird eine `BranchTooLongException` geworfen, deren Definition unten zu finden ist. Dadurch wird das weitere Wachstum der gesamten Verästelung gestoppt. Falls der Ast nach dem Wachsen höchstens 2000cm (20m) von der Wurzel entfernt, mindestens 360cm (3,6m) lang und ein Endstück ist, so wird er in ein Mittelstück umgewandelt, indem den beiden Attributen `left` und `right` je ein neues `Branch`-Objekt zugewiesen wird. Beide neuen `Branch`-Objekte sind Endstücke mit einer Länge von 40cm, die nach ihrer Erstellung ebenfalls wachsen.

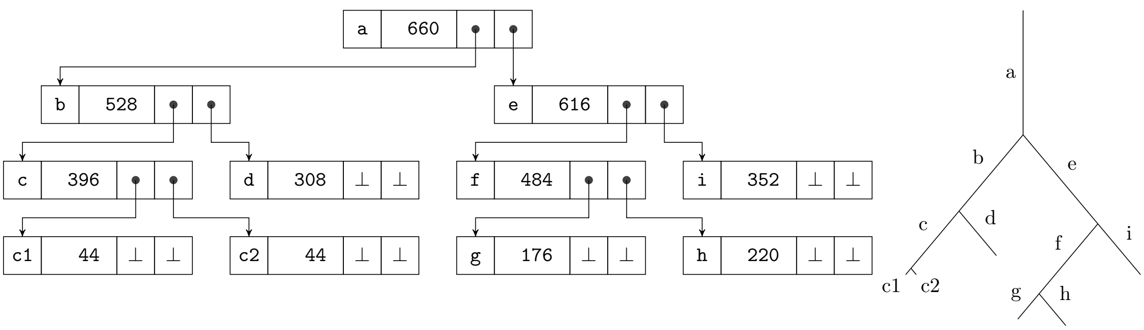
Verwenden Sie in dieser Teilaufgabe keine Schleifen, sondern **ausschließlich Rekursion**.

```
public class BranchTooLongException extends Exception {}
```

Beispiele:

- Im oben dargestellten Beispiel ist `h` mit 1800cm der Ast mit der größten Entfernung von der Wurzel. Falls `a.grow()` aufgerufen wird und jeder Ast nur um 10 Prozent wächst, so würde die folgende Verästelung entstehen. Weiterhin ist `h` mit 1980cm der Ast mit der größten Entfernung von der Wurzel.

  Da `c` das einzige Endstück ist, dass dann eine Länge von mindestens 360cm hat, wird es in ein Mittelstück umgewandelt und erhält die neuen Nachfolger `c1` und `c2` mit der Länge 40cm. Da diese ebenfalls wachsen, haben sie zum Schluss die Länge 44cm.



- Falls `a.grow()` aufgerufen wird und jeder Ast um mindestens 20 Prozent wächst, so würde eine `BranchTooLongException` geworfen werden, da der Ast mit der größten Entfernung von der Wurzel nun mehr als 2000cm von der Wurzel entfernt wäre.

Hinweise:

- Sie dürfen die Methode `Math.random()` nutzen, welche bei jedem Aufruf einen zufällig gewählten `double`-Wert x zurückgibt, mit $0 \leq x < 1$. Damit gibt `1.1 + 0.4 * Math.random()` also einen Wert x mit $1.1 \leq x < 1.5$ zurück.

```
public void grow() throws BranchTooLongException {
```

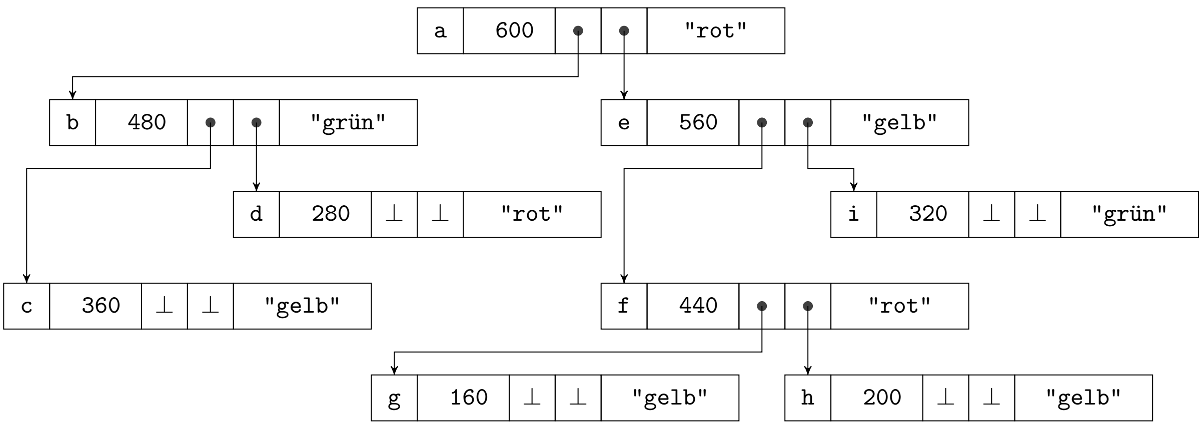
c) In der echten Welt können an einem Ast auch Blätter hängen. Dabei sind alle Blätter derselben Verästelung vom selben Typ, beispielsweise Birkenblätter, Eichenblätter, Tannennadeln, `String` (z.B. für bunte Blätter), etc. Je nach Typ des Blatts können für ein konkretes Blatt verschiedene Eigenschaften interessant sein. Daher ist es sinnvoll, für jeden Blatttyp eine eigene Klasse anzulegen, um die konkreten Blätter anschließend als Objekte dieser Blatttyp-Klasse zu erzeugen.

In dieser Aufgabe soll die Klasse `Branch` so angepasst werden, dass jeder Ast ein Blatt speichern kann und alle Blätter derselben Verästelung vom selben Typ sind. Schreiben Sie dazu die ganz oben in der Aufgabe gegebene Definition der Klasse `Branch` erneut auf, inklusive der Attribute. Es ist nicht notwendig, die Methoden aus den vorigen Aufgabenteilen anzupassen. Fügen Sie einen generischen Typparameter für die Klasse `Branch` ein, welcher den Typ der Blätter angibt. Ergänzen Sie die Klasse `Branch` außerdem um ein Attribut `leaf`, in dem ein Blatt-Objekt gespeichert werden kann. Wählen sie dazu den passenden Typ. Falls nötig, nehmen Sie weitere Anpassungen an den Typen der anderen Attribute vor.

Implementieren Sie außerdem einen Konstruktor, welcher als Parameter ein Blatt des passenden Typs erhält und dieses Blatt dem Attribut `leaf` zuweist.

Beispiel:

Im Folgenden ist dieselbe Verästelung dargestellt, wie zu Beginn der Aufgabe, jedoch wurde jedes `Branch`-Objekt in ein `Branch<String>`-Objekt umgewandelt. Die Blätter haben also den Typ `String`, wobei der `String`-Wert die Blattfarbe angibt. Für alle Äste wurde das Blatt mit dem Wert `"grün"`, `"gelb"` oder `"rot"` initialisiert. Es ist Herbst.



Hinweise:

- Sie dürfen davon ausgehen, dass der Parameter des Konstruktors nicht den Wert `null` hat.
- Sie dürfen in allen folgenden Teilaufgaben davon ausgehen, dass das Attribut `leaf` jedes `Branch`-Objekts nicht `null` ist.

d) Implementieren Sie in der Klasse `Branch` die Methode `leaves`, welche alle Blätter der Verästelung in eine `java.util.List` einfügt und diese anschließend zurückgibt. Dabei dürfen die Blätter in einer beliebigen Reihenfolge in die Liste eingefügt werden.

Beispiel:

Sei `a` das `Branch<String>`-Objekt aus Aufgabenteil c), dann wird der Ausdruck `a.leaves()` zu einer `List<String>` ausgewertet, welche zweimal den Wert `"grün"`, viermal den Wert `"gelb"` und dreimal den Wert `"rot"` in einer beliebigen Reihenfolge enthält.

Hinweise:

- Sie dürfen den parameterlosen Konstruktor der Klasse `java.util.LinkedList<T>` nutzen, um eine neue Liste vom Typ `java.util.List<T>` zu erzeugen.
- Sie dürfen für eine Liste `xs` vom Typ `List<T>` die Methode `xs.add(e)` nutzen, welche das Element `e` vom Typ `T` in die Liste `xs` einfügt.
- Sie dürfen für zwei Listen `xs` und `ys` vom Typ `List<T>` die Methode `xs.addAll(ys)` nutzen, welche alle Elemente aus `ys` in die Liste `xs` einfügt.
- Gehen Sie davon aus, dass die Datei `Branch.java` mit folgender Zeile beginnt:
  `import java.util.*;`

```
public List<L> leaves() {
```

e) Implementieren Sie in der Klasse `Branch` die Methode `print`, welche jedes Blatt der Verästelung in einer neuen Zeile auf dem Bildschirm ausgibt.

Verwenden Sie in dieser Teilaufgabe keine Rekursion, sondern **ausschließlich Schleifen**. Falls Sie Methoden aus vorherigen Teilaufgaben aufrufen möchten, so dürfen diese natürlich weiterhin Rekursion nutzen.

Beispiel:

Sei `a` das `Branch<String>`-Objekt aus Aufgabenteil c), dann wird der Ausdruck `a.print()` zweimal den Wert `"grün"`, viermal den Wert `"gelb"` und dreimal den Wert `"rot"` in einer beliebigen Reihenfolge auf dem Bildschirm ausgeben.

Hinweise:

- Sie dürfen für ein Objekt `x` die Methode `System.out.println(x)` nutzen, um `x` auf dem Bildschirm auszugeben.
- Gehen Sie davon aus, dass die Datei `Branch.java` mit folgender Zeile beginnt:
  `import java.util.*;`

```
public void print() {
```

Lösung:

a)
```java
public int countBranches() {
  if (left != null && right != null) {
    return 1 + left.countBranches() + right.countBranches();
  } else {
    return 1;
  }
}
```

b)
```
public void grow() throws BranchTooLongException {
  doGrow(0);
}

public void doGrow(int previousRootDistance) throws BranchTooLongException {
  length *= 1.1 + 0.4 * Math.random();
  int newRootDistance = previousRootDistance + length;
  if (newRootDistance > 2000) {
    throw new BranchTooLongException();
  }
  if (length >= 360) {
    if (left == null || right == null) {
      left = new Branch();
      right = new Branch();
    }
  }
  if (left != null && right != null) {
    left.doGrow(newRootDistance);
    right.doGrow(newRootDistance);
  }
}
```

c)
```
public class Branch<L> {
  private int length = 40;
  private Branch<L> left = null;
  private Branch<L> right = null;
  private L leaf;

  public Branch(L leaf) {
    this.leaf = leaf;
  }
}
```

d)
```
public List<L> leaves() {
  List<L> result = new LinkedList<>();
  collectLeavesToList(result);
  return result;
}

private void collectLeavesToList(List<L> result) {
  result.add(leaf);
  if (left != null && right != null) {
    left.collectLeavesToList(result);
    right.collectLeavesToList(result);
  }
}
```

e)
```
public void print() {
  for (L leaf : leaves()) {
    System.out.println(leaf);
  }
}
```

Aufgabe 5 (Haskell): (2 + 4 + 3 + 3 + 8 = 20 Punkte)

a) Geben Sie zur folgenden Haskell–Funktion `f` den allgemeinsten Typ an.

```
f h (x:xs) = if x then [h x] else h x : f h xs
```

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

```
i = map (\x -> x /= x) (map odd [2,5])

j = let f (x:y:_) = x + y in
      filter (odd . f) [[9,1,1,2],[1,4,2]]
```

Hinweise:

- Die Funktion `odd` vom Typ `Int -> Bool` untersucht, ob ihr Argument ungerade ist. Es gilt also `odd 0 == False` und `odd 1 == True`.
- Der Operator `.` vom Typ `(b -> c) -> (a -> b) -> a -> c` entspricht der Funktionskomposition. Es gilt also `(not . odd) 1 == not (odd 1) == False`.

c) Wir verwenden die folgende Datenstrukturen, um arithmetische Ausdrücke und eine einfache imperative Programmiersprache zu modellieren:

```
data Expr = Var String
          | Number Int
          | Minus Expr Expr

data Program = NoOp
             | Assgn String Expr
             | While Expr Program
             | Seq Program Program
```

In dieser Programmiersprache werden Variablen durch ihre Namen vom Typ `String` repräsentiert. Ausdrücke sind entweder (Integer-) Variablen, Integer-Zahlen oder Subtraktionen von Ausdrücken.

Das leere Programm wird durch den Konstruktor `NoOp` repräsentiert. `Assgn x e` repräsentiert ein Programm, welches der Variablen `x` den Wert des Ausdrucks `e` zuweist. `While e p` stellt ein Programm dar, welches einer While-Schleife entspricht, bei welcher der Schleifenrumpf `p` solange ausgeführt wird, wie der Ausdruck `e` positiv ist. `Seq p q` ist ein Programm, welches der Hintereinanderausführung der Programme `p` und `q` entspricht.

Der folgende Haskell-Ausdruck `p1` repräsentiert ein Programm, welches der Variablen `"x"` zuerst den Wert `10` zuweist, und diese dann in einer Schleife solange verkleinert, bis sie nicht mehr positiv ist.

```
p1 :: Program
p1 = Seq (Assgn "x" (Number 10))
         (While (Var "x")
                (Assgn "x" (Minus (Var "x")
                                  (Number 1))))
```

```
// Entsprechendes Java-Programm
x = 10;
while (x > 0) {
  x = x-1;
}
```

Für alle folgenden Teilaufgaben gelten die folgenden Hinweise:

Hinweise:

- Sie können jeweils Funktionen aus vorherigen Teilaufgaben verwenden, unabhängig davon, ob Sie diese implementiert haben.
- Sie dürfen beliebige vordefinierte Funktionen aus der Standardbibliothek verwenden, wie z.B. `map`.
- Sie dürfen eigene Hilfsfunktionen implementieren und nutzen.

Schreiben Sie eine Funktion `mkSeq :: [Program] -> Program`, sodass der Aufruf `mkSeq ps` ein Programm erzeugt, welches alle Programme aus `ps` nacheinander ausführt. Ihre Implementierung darf hierbei beliebig viele neue `NoOp` Anweisungen einfügen.

So kann der Aufruf `mkSeq [NoOp, Assgn "x" (Number 0)]` beispielsweise zu dem Programm `Seq NoOp (Seq (Assgn "x" (Number 0)) NoOp)` auswerten.

d) Implementieren Sie die Funktion `eval :: Expr -> (String -> Int) -> Int`. Ein Programmzustand wird durch eine Funktion vom Typ `String -> Int` festgelegt, welche jeder Variablen ihren Wert zuweist. Für eine Funktion `s :: String -> Int` und einen Ausdruck `e :: Expr` soll dann `eval e s` der Wert von `e` im Programmzustand `s` sein.

Falls also `s "x" == 10` gilt, so soll `eval (Minus (Var "x") (Number 1)) s == 9` gelten.

e) Implementieren Sie eine Funktion `run :: Program -> (String -> Int) -> (String -> Int)`, sodass `run p s` zu dem Zustand des Programms `p` nach seiner Ausführung im Programmzustand `s` auswertet. Wenn die Ausführung von `p` im Zustand `s` nicht terminiert, so darf sich Ihre Implementierung beliebig verhalten.

Beispielsweise wertet `(run p1 s) "x"` für alle `s :: String -> Int` zu `0` aus.

Lösung:

a)
```
f :: (Bool -> a)-> [Bool] -> [a]
```

b)
```
i = [False,False]
j = [[1,4,2]]
```

c)
```
-- c)
-- mit foldr
mkSeq :: [Program] -> Program
mkSeq = foldr Seq NoOp

-- explizite "foldr" Version
mkSeq' :: [Program] -> Program
mkSeq' []     = NoOp
mkSeq' (p:ps) = Seq p (mkSeq' ps)


-- explizite "foldl" Version mit Akkumulator
mkSeq'' :: [Program] -> Program
mkSeq'' = helper NoOp
  where
    helper acc []     = acc
    helper acc (p:ps) = helper (Seq acc p) ps
```

d)
```
-- d)
eval :: Expr -> (String -> Int) -> Int
eval (Var v)       s = s v
eval (Number i)    _ = i
eval (Minus e1 e2) s = eval e1 s - eval e2 s
```

e)
```
-- e)
run :: Program -> (String -> Int) -> (String -> Int)
run NoOp         s = s
run (Assgn v e) s = \v' -> if v == v' then eval e s else s v'
run (While e p) s
  | eval e s   > 0 = run (While e p) (run p s)
  | otherwise      = s
run (Seq p q)   s = run q (run p s)
```

Aufgabe 6 (Prolog): (2 + 8 + (3 + 4 + 3) = 20 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) $f(g(a, X), g(Z, s(X))), f(Y, g(a, Y))$

ii) $h(s(a), s(X), s(Y)), h(X, s(s(Y)), X)$

b) Gegeben sei folgendes Prolog-Programm P.

```
f(N,M,[N|XS]) :- f(s(N),M,XS).
f(N,N,[N]).
f(N,s(M),[s(M)|XS]) :- f(N,M,XS).
```

- Erstellen Sie für das Programm P den Beweisbaum zur Anfrage `?- f(0,s(0),Res).` bis zur Höhe 3 **(die Wurzel hat dabei die Höhe 1)**. Die Pfade haben also maximal die Länge 2.
- Markieren Sie die Knoten in Höhe 3, die zu einer unendlichen Auswertung führen können, mit ∞.
- Geben Sie alle Antwortsubstitutionen zur Anfrage `?- f(0,s(0),Res).` an, die im Beweisbaum bis zur Höhe 3 enthalten sind.
- Geben Sie außerdem zu jeder dieser Antwortsubstitutionen an, ob sie von Prolog gefunden wird.
- Knoten, von denen keine weitere Ausführung aus möglich ist, sollen mit *fail* markiert werden.
- Wie muss das Programm durch Verschiebung von Klauseln abgeändert werden, damit Prolog mindestens eine Antwortsubstitution bis zur Höhe 3 findet?

c) In dieser Aufgabe betrachten wir das sog. Klappenspiel, ein einfaches Würfelspiel. Die neun namensgebenden Klappen sind mit den Ziffern von 1 bis 9 beschriftet. Ein Spiel besteht aus dem wiederholten Wurf von zwei sechsseitigen Würfeln. Nach jedem Wurf werden eine oder mehrere der neun anfänglich offenen Klappen geschlossen, wenn dies noch möglich ist. Die Summe der nach einem Wurf zu schließenden Klappe(n) muss dabei stets genau der gewürfelten Augenzahl entsprechen. Mit einer im ersten Wurf gewürfelten 6 ergeben sich also genau die folgenden vier Möglichkeiten. Geschlossen werden können entweder

- die drei Klappen 1, 2 und 3 oder
- die beiden Klappen 1 und 5 oder
- die beiden Klappen 2 und 4 oder
- nur die Klappe 6.

Ist es nach einem Wurf nicht möglich, gemäß der gerade beschriebenen Regeln Klappen zu schließen, endet das Spiel und es werden Minuspunkte in Höhe der Summe der noch offenen Klappen vergeben. Ansonsten wird ein weiterer Wurf durchgeführt.

Hinweise:

- Sie dürfen Prädikate aus vorigen Teilaufgaben verwenden, auch wenn Sie diese Prädikate nicht implementiert haben.
- Sie dürfen eigene Hilfsprädikate implementieren und nutzen.
- Sie dürfen beliebige vordefinierte Prädikate für Arithmetik in Prolog benutzen, wie z.B. das zweistellige Prädikat `>` für *größer*. So wertet `4 > 3` zu `true` aus, `5 > 6` zu `false`.

i) Implementieren Sie ein Prädikat `fromNToM` mit Stelligkeit 3 in Prolog. Wenn `N` und `M` ganze Zahlen sind, so soll `fromNToM(N,M,XS)` genau dann wahr sein, wenn `M` $\geq$ `N` gilt und `XS` die aufsteigende Liste mit allen ganzen Zahlen von `N` bis einschließlich `M` ist. Wenn `N` und `M` ganze Zahlen sind, so soll die Anfrage `fromNToM(N,M,XS)` für beliebige Argumente `XS` stets terminieren, d.h. der zugehörige Beweisbaum soll endlich sein. Beispielsweise ist `fromNToM(0,2,[0,1,2])` wahr und `fromNToM(5,3,XS)` falsch.

ii) Implementieren Sie ein Prädikat `klappbar` mit Stelligkeit 3 in Prolog. Wenn `X` eine ganze Zahl ist und `XS` und `YS` Listen von ganzen Zahlen sind (die die noch offenen Klappen des Klappenspiels darstellen), so soll `klappbar(X,XS,YS)` genau dann wahr sein, wenn `YS` sich aus `XS` durch das regelkonforme Schließen von Klappen mit Summe `X` ergibt. Beispielsweise hat die Anfrage `?- klappbar(3,[1,2,3,4],Rest).` genau die beiden Antwortsubstitutionen `Rest = [1, 2, 4]` und `Rest = [3, 4]`.

Hinweise:

- Das Prädikat `klappbar` muss nicht überprüfen, ob $2 \leq$ `X` ≤ 12 gilt und ob `XS` nur Zahlen von 1 bis 9 ohne Duplikate enthält.

iii) Implementieren Sie ein Prädikat `perfektesSpiel` mit Stelligkeit 1 in Prolog. Wenn `XS` eine Liste von Würfelergebnissen (also ganzen Zahlen) ist, soll `perfektesSpiel(XS)` genau dann wahr sein, wenn mit den Würfelergebnissen aus `XS` ein Spiel möglich ist, an dessen Ende alle neun Klappen geschlossen sind.

Hinweise:

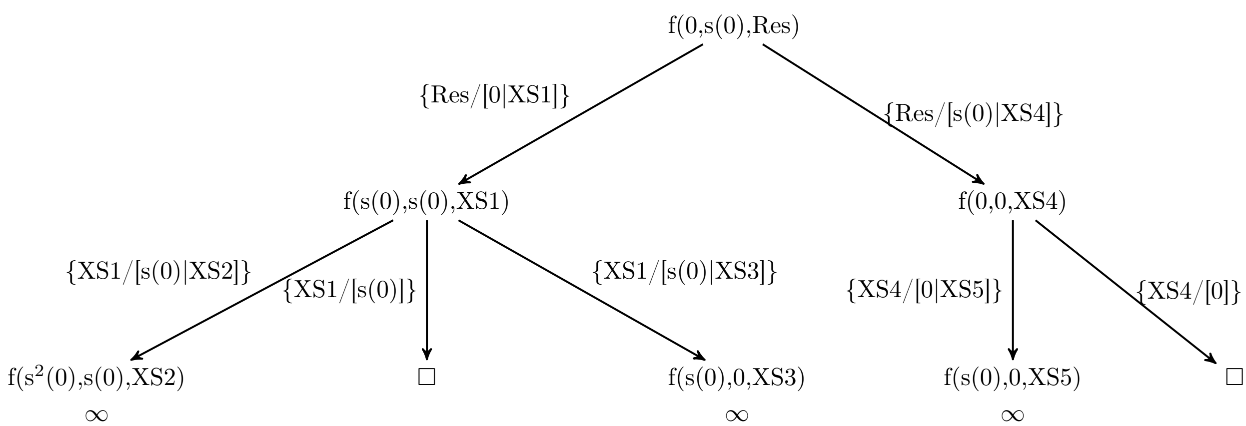
- Das Prädikat `perfektesSpiel` muss nicht überprüfen, ob `XS` nur Zahlen von 2 bis 12 enthält.

Lösung:

a) i) $f(g(a,X), g(Z,s(X))), f(Y,g(a,Y))$ kann nicht unifiziert werden. Nach der Unifikation des ersten Arguments ist das zweite Argument des zweiten inneren g-Symbols $s(X)$ bzw. $g(a,X)$. Es liegt ein Clash Failure vor.

ii) $h(s(a), s(X), s(Y)), h(X, s(s(Y)), X)$ hat als MGU $\sigma = \{X = s(a), Y = a\}$

b)



Es gibt zwei Antwortsubstitution innerhalb des Beweisbaums: {Res/[0,s(0)]} und {Res/[s(0),0]}. Diese werden von Prolog beide nicht gefunden.

Die erste Regel muss hinter das Faktum verschoben werden, damit Prolog eine Antwortsubstitutionen bis zur Höhe 3 findet. Eine Möglichkeit, mit der {Res/[0,s(0)]} gefunden wird, wäre z.B.:

```
f(N,N,[N]).
f(N,M,[N|XS]) :- f(s(N),M,XS).
f(N,s(M),[s(M)|XS]) :- f(N,M,XS).
```

c)
```
fromNToM(N,N,[N]).
fromNToM(N,M,[N|XS]) :- M > N, N1 is N + 1, fromNToM(N1,M,XS).
```

```
klappbar(X,[X|XS],XS).
klappbar(X,[Y|YS],[Y|ZS]) :- klappbar(X,YS,ZS).
klappbar(X,[Y|YS],ZS) :- X > Y, Rest is X - Y, klappbar(Rest,YS,ZS).

perfektesSpiel(XS) :- initKlappen(KS), perfektesSpielH(XS,KS).

initKlappen(XS) :- fromNToM(1,9,XS).

perfektesSpielH([],[]).
perfektesSpielH([X|XS],KS) :- klappbar(X,KS,KSRest), perfektesSpielH(XS,KSRest).
```