Aufgabe 1 (Programmanalyse): (14 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 {
  public long x = 20;

  public A(double x) {
    this((int) x);
  }

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

  public int f(long d) {
    return 1;
  }

  public int f(float i) {
    return 2;
  }
}
```

```
public class B extends A {
  public int x = 7;
  long y = 15;

  public B() {
    this(23);
  }

  public B(int x) {
    super(x);
    this.y = y - 4;
    this.x += 1;
  }

  public int f(int d) {
    return 3;
  }

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

```
public class M {
  public static void main(String[] args) {
    A a = new A(41.75);
    System.out.println(a.x);              // OUT: [     ]

    B b = new B(11);
    System.out.println(((A) b).x);        // OUT: [     ]

    System.out.println(b.y);              // OUT: [     ]

    System.out.println(b.x);              // OUT: [     ]

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

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

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

Lösung:

a)
```
public class M {
    public static void main(String[] args) {
      A a = new A(41.75);
      System.out.println(a.x);              // OUT: [ 42  ]

      B b = new B(11);
      System.out.println(((A) b).x);        // OUT: [ 12  ]

      System.out.println(b.y);              // OUT: [ 11  ]

      System.out.println(b.x);              // OUT: [  8  ]

      System.out.println(b.f(1f));          // OUT: [  2  ]

      System.out.println(b.f(1.));          // OUT: [  4  ]

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

Aufgabe 2 (Hoare-Kalkül): (9 + 3 = 12 Punkte)

Gegeben sei folgendes *Java*-Programm P, das zu einer Eingabe $n \geq 0$ den Wert $2^{(2^n)} + 1$ berechnet.

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

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

$\langle\, \texttt{res} = 2^{(2^n)} + 1 \,\rangle$ (Nachbedingung)

a) Vervollständigen Sie die Verifikation des Algorithmus P auf der folgenden 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.

Hinweise:

- Für alle $i \in \mathbb{N}$ gilt: Falls $r = 2^{(2^i)} + 1$ ist, dann ist $(r-2) \cdot r + 2 = 2^{(2^{i+1})} + 1$.
- 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.

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 $\texttt{n} \geq 0$ bewiesen werden. Begründen Sie, warum es sich bei der von Ihnen angegebenen Variante tatsächlich um eine gültige Variante handelt.

Lösung:

a)

$\langle \texttt{n} \geq 0 \rangle$
$\langle \texttt{n} \geq 0 \wedge 0 = 0 \wedge 3 = 3 \rangle$

```
i = 0;
```

$\langle \texttt{n} \geq 0 \wedge \texttt{i} = 0 \wedge 3 = 3 \rangle$

```
res = 3;
```

$\langle \texttt{n} \geq 0 \wedge \texttt{i} = 0 \wedge \texttt{res} = 3 \rangle$
$\langle \texttt{res} = 2^{(2^\texttt{i})} + 1 \wedge \texttt{i} \leq \texttt{n} \rangle$

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

$\langle \texttt{res} = 2^{(2^\texttt{i})} + 1 \wedge \texttt{i} \leq \texttt{n} \wedge \texttt{i} < \texttt{n} \rangle$
$\langle (\texttt{res} - 2) \cdot \texttt{res} + 2 = 2^{(2^{(\texttt{i}+1)})} + 1 \wedge \texttt{i} + 1 \leq \texttt{n} \rangle$

```
    res = (res - 2) * res + 2;
```

$\langle \texttt{res} = 2^{(2^{(\texttt{i}+1)})} + 1 \wedge \texttt{i} + 1 \leq \texttt{n} \rangle$

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

$\langle \texttt{res} = 2^{(2^\texttt{i})} + 1 \wedge \texttt{i} \leq \texttt{n} \rangle$

```
}
```

$\langle \texttt{res} = 2^{(2^\texttt{i})} + 1 \wedge \texttt{i} \leq \texttt{n} \wedge \texttt{i} \not< \texttt{n} \rangle$
$\langle \texttt{res} = 2^{(2^\texttt{n})} + 1 \rangle$

b) Eine gültige Variante für die Terminierung ist $V = \texttt{n} - \texttt{i}$, denn die Schleifenbedingung $B = \texttt{i} < \texttt{n}$ impliziert $\texttt{n} - \texttt{i} \geq 0$ und es gilt:

$\langle \texttt{n} - \texttt{i} = m \wedge \texttt{i} < \texttt{n} \rangle$
$\langle \texttt{n} - (\texttt{i} + 1) < m \rangle$
`res = (res - 2) * res + 2;`
$\langle \texttt{n} - (\texttt{i} + 1) < m \rangle$
`i = i + 1;`
$\langle \texttt{n} - \texttt{i} < m \rangle$

Damit ist die Terminierung der einzigen Schleife in P gezeigt.

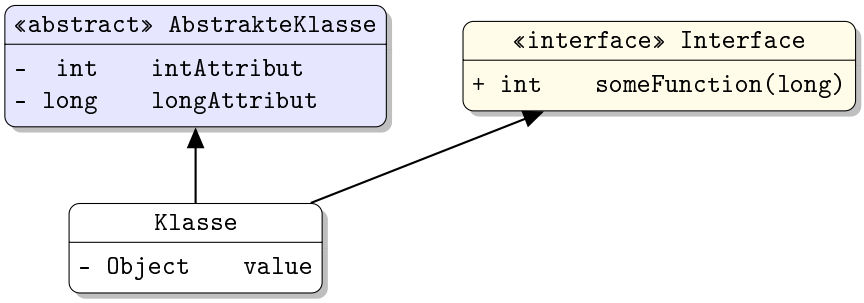
Aufgabe 3 (Klassen-Hierarchie): (4 + 7 = 11 Punkte)

In dieser Aufgabe betrachten wir verschiedene Arten von Kaffeemaschinen und organisieren diese in einer Hierarchie.

- Eine Kaffeemaschine zeichnet sich dadurch aus, wie viel Wasser (in ml) in den Wassertank passt.
- Eine Filter-Kaffeemaschine ist eine Kaffeemaschine und wir speichern, ob diese einen Timer hat.
- Ein Kaffee-Vollautomat ist eine Kaffeemaschine. Hier ist relevant, wie viele verschiedene Stufen das Mahlwerk hat.
- Ein Espresso-Vollautomat ist ein Kaffee-Vollautomat, bei dem der für die Espresso-Zubereitung erzeugte Druck (in bar) wichtig ist.
- Eine Portions-Kaffeemaschine ist eine Kaffeemaschine, bei der die Farbe des Gerätes wichtig ist.
- Mit Espresso-Vollautomaten und Portions-Kaffeemaschinen lässt sich auch Tee kochen. Dazu stellen sie die Methode `void kocheTee()` zur Verfügung.
- Jede Kaffeemaschine ist entweder eine Filter-Kaffeemaschine, ein Kaffee-Vollautomat, ein Espresso-Vollautomat oder eine Portions-Kaffeemaschine.

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

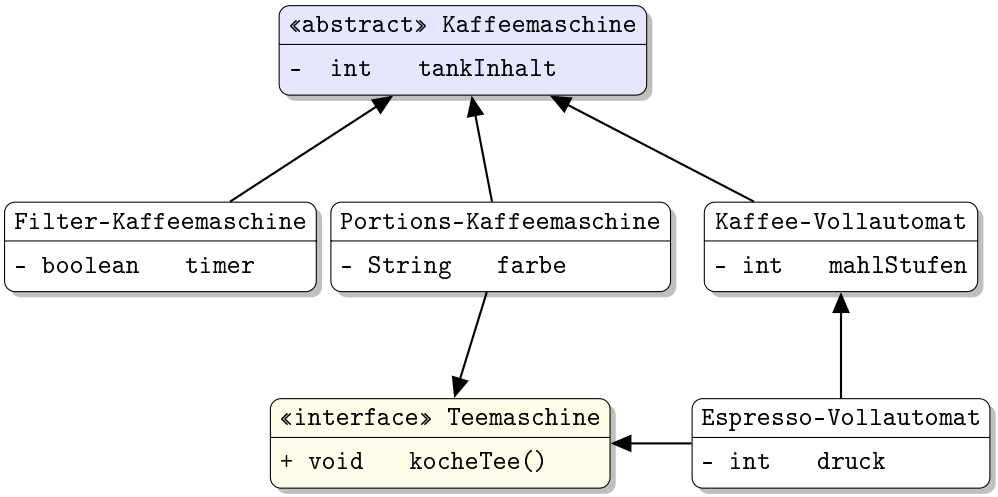


Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Attribute und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \blacktriangleright A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls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

b) Schreiben Sie eine Java-Methode mit der folgenden Signatur:
`public static int vorbereiten(Kaffeemaschine[] km)`.
Diese Methode soll zurückgeben, wie viele ml Wasser insgesamt in die Kaffeemaschinen des Arrays passt. Zusätzlich soll mit jeder im Array enthaltenen Maschine Tee gekocht werden, sofern dies möglich ist. Nehmen Sie dazu an, dass das übergebene Array `km` nicht `null` ist.

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:



b)
```
public static int vorbereiten(Kaffeemaschine[] km) {
    int sum = 0;
    for (Kaffeemaschine k : km) {
        if (k != null) {
          sum += k.tankInhalt;
          if (k instanceof Teemaschine) {
              ((Teemaschine) k).kocheTee();
          }
        }
    }
    return sum;
}
```

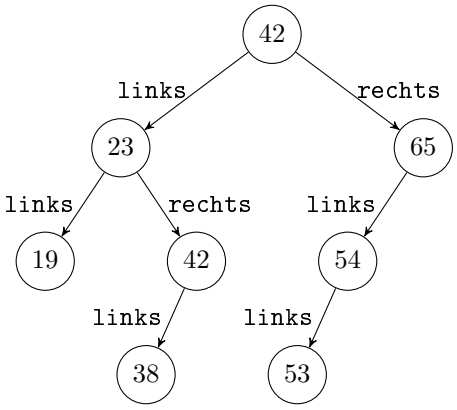
Aufgabe 4 (Programmieren in Java): (10 + 13 + 13 + 8 + 5 = 49 Punkte)

In dieser Aufgabe betrachten wir sortierte binäre Bäume, die Zahlen speichern. In solchen Bäumen gilt immer die folgende Invariante:

> Für jeden Knoten mit Beschriftung x sind alle Knoten im linken Kindbaum mit Zahlen y beschriftet, so dass $y \leq x$ gilt, und alle Knoten im rechten Kindbaum sind mit Zahlen z beschriftet, so dass $z > x$ gilt.

Eine Java-Klasse `Baum`, mit der solche Bäume dargestellt werden können, und ein Beispielbaum sind in der folgenden Grafik zu sehen. Hierbei hat die Wurzel den Wert 42 und die Blätter haben die Werte 19, 38 und 53.

```
public class Baum {
  private int wert;
  private Baum links, rechts;
  public Baum(Baum l, int w, Baum r) {
    this.links = l;
    this.wert = w;
    this.rechts = r;
  }
}
```



a) Implementieren Sie eine Methode `hoehe` in der Klasse `Baum`, die die Höhe des aktuellen Baums zurückgibt. Dabei hat ein Baum, der nur die Wurzel enthält, die Höhe 1 und die Höhe aller anderen Bäume ist 1 plus das Maximum der Höhen des linken und rechten Teilbaums. Der Beispielbaum von oben hat die Höhe 4. Verwenden Sie **nur Rekursion** und **keine Schleifen**.

b) Implementieren Sie eine Methode `einfuegen` in der Klasse `Baum`, die einen `int`-Wert als Blatt in den aktuellen Baum einfügt. Für unseren Baum von oben sollte das Einfügen von 56 einen neuen Knoten als rechten Nachfolger von 54 erzeugen, der mit 56 beschriftet ist. Verwenden Sie **nur Schleifen** und **keine Rekursion**. Sie dürfen dabei annehmen, dass der Baum bereits sortiert ist. Fügen Sie den Wert so ein, dass der Baum nach dem Einfügen immer noch sortiert ist.

c) Implementieren Sie die Methode `toList` in der Klasse `Baum`, die eine **sortierte** Liste mit den Werten aus dem aktuellen Baum zurückgibt. Verwenden Sie das Interface `List<T>` aus dem `Collections`-Framework für das Ergebnis und instanziieren Sie `T` geeignet. Für unseren Beispielbaum soll also eine Liste zurückgegeben werden, die `[19, 23, 38, 42, 42, 53, 54, 65]` entspricht. Sie dürfen dabei annehmen, dass der Baum sortiert ist. Verwenden Sie **nur Rekursion** und **keine Schleifen**.

Hinweise:

- Die Klasse `LinkedList<T>` implementiert das Interface `List<T>` und verfügt über eine Methode `add(T x)`, die ein Element an das *Ende* der Liste anhängt.
- Schreiben und verwenden Sie eine Hilfsmethode.

d) Wir betrachten nun das Interface `BaumFolder<T>`, mit dem eine Funktionalität ähnlich zum `fold`-Konstrukt in funktionalen Sprachen implementiert werden soll:

```
public interface BaumFolder<T> {
  T handleNull();
  T handleBaum(T l, int w, T r);
}
```

Zum Anwenden eines `BaumFolders f` auf ein Objekt vom Typ `Baum` mit Wert `w` werden zuerst die Ergebnisse `l` und `r` der Anwendung von `f` auf den linken bzw. rechten Teilbaum bestimmt. Das Ergebnis der Anwendung auf dem gesamten Baum entspricht dann der Rückgabe eines Aufrufs von `handleBaum` mit den Werten `w`, `l` und `r`. Für `null` ergibt sich der Wert der Anwendung durch `handleNull()`.

Die Klasse `BaumSum` ist ein Beispiel für die Verwendung des Interfaces und berechnet die Summe aller Werte im Baum:

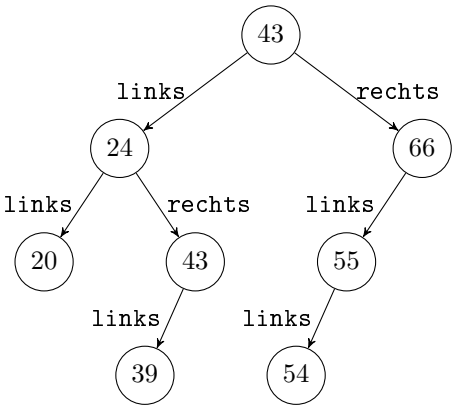
```
class BaumSum implements BaumFolder<Integer> {
  public Integer handleNull() {
    return 0;
  }

  public Integer handleBaum(Integer l, int w, Integer r) {
    return l + w + r;
  }
}
```

Implementieren Sie die generische Methode `apply` in der Klasse `Baum`, die einen `BaumFolder<T>` wie oben beschrieben auf einen Baum anwendet. Wenn also `t` der Baum vom Anfang der Aufgabe ist, so sollte `t.apply(new BaumSum())` die Summe seiner Werte (336) als `Integer`-Objekt ergeben.

```
public <T> T apply(BaumFolder<T> f) {
```

e) Implementieren Sie nun eine Klasse `BaumInc`, die das Interface `BaumFolder<Baum>` so implementiert, dass die Anwendung einen neuen `Baum` ergibt, in dem alle Werte um eins inkrementiert wurden. Für den Baum `t` vom Anfang der Aufgabe soll sich für den Ausdruck `t.apply(new BaumInc())` dieser Baum ergeben:



Lösung:

a) Kurz:

```
public int hoehe() {
  return 1 +
    Math.max(
      links != null ? links.hoehe() : 0,
      rechts != null ? rechts.hoehe() : 0);
}
```

Ordentlich:

```
public int hoehe() {
  int l = 0;
  if (this.links != null)
    l = this.links.hoehe();

  int r = 0;
```

```
    if (this.rechts != null)
      r = this.rechts.hoehe();

    if (l > r)
      return l + 1;
    else
      return r + 1;
  }
```

Alternativ können beide Varianten auch `static` mit explizitem Parameter für den Baum implementiert werden.

b)

```
  public void einfuegen(int x) {
    Baum prev = null;
    Baum cur = this;
    while (cur != null) {
      prev = cur;
      if (x <= cur.wert)
        cur = cur.links;
      else
        cur = cur.rechts;
    }

    Baum neu = new Baum(null, x, null);
    if (x <= prev.wert)
      prev.links = neu;
    else
      prev.rechts = neu;
  }
```

c)

```
  public List<Integer> toList() {
    List<Integer> res = new LinkedList<>();
    this.toList(res);
    return res;
  }

  private void toList(List<Integer> res) {
    if (this.links != null)
      this.links.toList(res);
    res.add(this.wert);
    if (this.rechts != null)
      this.rechts.toList(res);
  }
```

d)

```
  public <T> T apply(BaumFolder<T> f) {
    T l;
    if (this.links != null) {
      l = this.links.apply(f);
    } else {
      l = f.handleNull();
    }

    T r;
    if (this.rechts != null) {
      r = this.rechts.apply(f);
    } else {
```

```
        r = f.handleNull();
      }

      return f.handleBaum(l, this.wert, r);
    }
```

e)
```
  class BaumInc implements BaumFolder<Baum> {
    public Baum handleNull() {
      return null;
    }

    public Baum handleBaum(Baum links, int wert, Baum rechts) {
      return new Baum(links, wert + 1, rechts);
    }
  }
```

Aufgabe 5 (Haskell): (3 + 3 + 4 + 7 = 17 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 und die Funktion `+` den Typ `Int -> Int -> Int` hat.

i)
```
f 0 x = f x x + f x x
f 1 _ = 3
```

ii)
```
g x y = g (y, y) y
```

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

```
i :: Int
i = (\f x -> f x + 1) (\x -> x * x) 3

j :: [Int]
j = map (\x -> x*x) (filter (\x -> x `mod` 2 == 0) [1,2,3])
```

c) Implementieren Sie die Funktion `minim :: [Int] -> Int`, die das Minimum einer Liste (welche nicht leer ist) bestimmt. So liefert der Aufruf `minim [3,1,2]` den Wert `1` zurück. Ein Aufruf Ihrer Funktion auf der leeren Liste darf zu einem beliebigen Ergebnis führen.

d) Implementieren Sie die Funktion `levenshtein :: String -> String -> Int`, welche die Editierdistanz zwischen zwei Wörtern misst. Hierbei gibt ein Aufruf `levenshtein w1 w2` an, wieviele Editierschritte man mindestens braucht, um das Wort `w1` in das Wort `w2` zu überführen. Die drei möglichen Editierschritte sind hierbei: Ein Zeichen *löschen*, ein Zeichen *einfügen* und ein Zeichen *überschreiben*. Beispielsweise kann man das Wort "ruhte" in drei Schritten in das Wort "Route" überführen: Ersetze "r" durch "R", füge hinter dem "R" das Zeichen "o" ein und lösche das "h". Da dies nicht mit weniger Schritten möglich ist, gibt also der Aufruf `levenshtein "ruhte" "Route"` den Wert `3` zurück.

Verwenden Sie folgenden Algorithmus zur Berechnung der Levenshtein-Funktion:

- Falls das erste oder zweite übergebene Wort das leere Wort ist, gibt die Funktion die Länge des jeweils anderen Wortes zurück (da man offensichtlich mindestens diese Zahl an Einfüge- bzw. Löschoperationen benötigt, um das erste Wort in das Zweite zu überführen).
- Anderenfalls sind beide Worte nicht leer. Der Algorithmus vergleicht dann jeweils den ersten Buchstaben beider Wörter miteinander. Sind diese gleich, ist das Ergebnis der rekursive Aufruf der Funktion auf dem Rest (d.h. ohne den ersten Buchstaben) der beiden Wörter. Sind die Buchstaben unterschiedlich, muss mindestens einer der Editierschritte durchgeführt werden. Somit ist das Ergebnis an dieser Stelle Eins mehr als das Minimum folgender Werte:
  - Ein rekursiver Aufruf mit dem ganzen ersten Wort und dem Rest des zweiten Wortes. Dies entspricht dem Einfügen des ersten Buchstabens des zweiten Wortes.
  - Ein rekursiver Aufruf mit dem Rest des ersten Wortes und dem ganzen zweiten Wort. Dies entspricht dem Löschen des ersten Buchstabens des ersten Wortes.
  - Ein rekursiver Aufruf mit dem Rest des ersten Wortes und dem Rest des zweiten Wortes. Dies entspricht dem Überschreiben des ersten Buchstabens des ersten Wortes mit dem ersten Buchstaben des zweiten Wortes.

Verwenden Sie hierbei die vordefinierte Funktionen `length :: [a] -> Int`, die die Länge einer Liste berechnet, und `minim :: [Int] -> Int` aus Aufgabenteil c).

Lösung:

a) i)
```
f :: Int -> Int -> Int
```

ii)
```
g :: (a, a) -> a -> b
```

b)
```
i = 10
j = [4]
```

c)
```
minim :: [Int] -> Int
minim [x] = x
minim (x : xs) = if x < m then x else m
  where m = minim xs
```

d)
```
levenshtein :: String -> String -> Int
levenshtein [] ys = length ys
levenshtein xs [] = length xs
levenshtein (x:xs) (y:ys)
    | x == y    = levenshtein xs ys
    | otherwise = 1 + minim
        [ levenshtein (x:xs) ys -- y einfuegen
        , levenshtein xs (y:ys) -- x loeschen
        , levenshtein xs ys     -- x zu y aendern
        ]
```

Aufgabe 6 (Prolog): (2 + 8 + 7 = 17 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) $f(X, Y, a), f(Z, b, Z)$

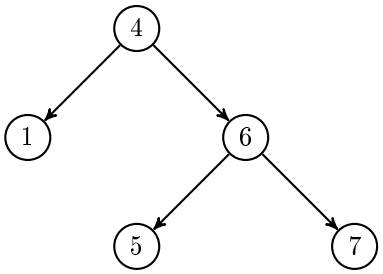
ii) $g(c(X), Y, X), g(Z, Z, Z)$

b) Gegeben sei folgendes Prolog-Programm P.

```
p(a,b).
p(X,X) :- p(s(X),s(X)).
p(a,X) :- p(X,b).
```

Erstellen Sie für das Programm P den Beweisbaum zur Anfrage `?- p(Y,Z).` bis zur Höhe 3 (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(Y,Z).` im Graphen an.

c) Binäre Bäume können in Prolog folgendermaßen als Terme dargestellt werden. Sei `N` ein beliebiger Term in Prolog. Dann repräsentiert der Term `leaf(N)` einen Baum mit nur einem Blatt, welches das Element `N` enthält. Für zwei Bäume `X` und `Y` repräsentiert der Term `node(X,N,Y)` einen binären Baum mit einem Wurzelknoten, der das Element `N` enthält und die Teilbäume `X` und `Y` hat. Als Beispiel ist nachfolgend ein binärer Baum und seine Darstellung als Term angegeben.



`node(leaf(1),4,node(leaf(5),6,leaf(7)))`

Eine Tiefensuche arbeitet folgendermaßen. Bei einem Blatt findet sie genau das Element, das im Blatt enthalten ist. Bei einem inneren Knoten findet sie zunächst rekursiv alle Elemente im ersten (linken) Teilbaum, dann das Element im inneren Knoten und schließlich wieder rekursiv alle Elemente im zweiten (rechten) Teilbaum. Eine Tiefensuche findet die Elemente des Beispielbaumes in folgender Reihenfolge: $1, 4, 5, 6, 7$.

Schreiben Sie ein Prädikat `toList/2` in Prolog, wobei `toList(X,Y)` genau dann wahr sein soll, wenn `X` ein binärer Baum ist und `Y` eine Liste, welche genau die Elemente von `X` in der Reihenfolge enthält, wie sie eine Tiefensuche findet. Beispielsweise gilt also
`toList(node(leaf(1),4,node(leaf(5),6,leaf(7))),[1,4,5,6,7])`.

Hinweise:

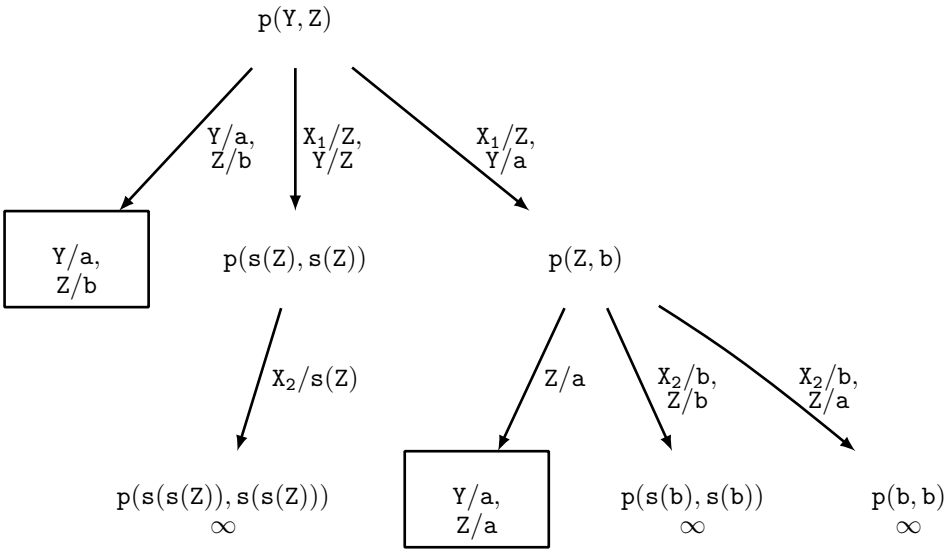
- Sie dürfen das vordefinierte Prädikat `append/3` benutzen, wobei `append(X,Y,Z)` genau dann wahr ist, wenn `X`, `Y` und `Z` Listen sind und `Z` zuerst genau die Elemente von `X` und danach genau die Elemente von `Y` enthält (die Listen `X` und `Y` werden also aneinander gehängt, um `Z` zu berechnen). Beispielsweise gilt `append([1,4],[5,6,7],[1,4,5,6,7])`.

Lösung:

a) i) $f(X, Y, a), f(Z, b, Z)$: $X/a, Y/b, Z/a$

ii) $g(c(X), Y, X), g(Z, Z, Z)$: occur failure X in $c(X)$

b)



Die Antwortsubstitutionen sind $\{\mathtt{Y/a}, \mathtt{Z/b}\}$ und $\{\mathtt{Y/a}, \mathtt{Z/a}\}$.

c)
```
toList(leaf(N),[N]).
toList(node(X,N,Y),ZS) :- toList(X,XS),
                          toList(Y,YS),
                          append(XS,[N|YS],ZS).
```