Prof. C. Bischof, Ph.D. M. Brockschmidt, F. Emmes, M. Lülfesmann, J. Willkomm

Aufgabe 1 (Programmanalyse): (10 Punkte)

Gegeben sei das folgende Java-Programm. Dieses Programm gibt sechs Zeilen Text in der Form
`x: Erweitert: 12.3, Basis: 4, 56.7`
sowie zwei weitere Zeilen der Form
`e.val: 12.3`
und
`b.val: 12.3`
auf der Konsole aus. Tragen Sie die Ausgabe an den markierten Stellen in den Kommentaren ein.

```
class Basis {
    static int a = 0;
    double val;
    Basis() {         a = a + 1;     val = 1; }
    Basis(int x) {    a = x;         val = 2; }
    void f(int x) {
        a = a + 1;
        val = val * x;
    }
    void f(double x) {
        a = a + 1;
        val = val * x;
    }
    public String toString() { return "Basis: " + a + ", " + val; }
}

class Erweitert extends Basis {
    double val;
    Erweitert(double x) {  val = x;   }
    void f(double x) {     super.f(x); val = val * x;  }
    public String toString() {
        return "Erweitert: " + val + ", " + super.toString();
    }
}

class Programm {
    public static void main(String[] p) {
        Erweitert e = new Erweitert(12);
        System.out.println("e: " + e); // e: Erweitert:     , Basis:   ,
        Basis b = new Erweitert(12);
        System.out.println("b: " + b); // b: Erweitert:     , Basis:   ,

        e.f(2.0);
        System.out.println("e: " + e); // e: Erweitert:     , Basis:   ,
        System.out.println("b: " + b); // b: Erweitert:     , Basis:   ,

        b.f(3);
        System.out.println("e: " + e); // e: Erweitert:     , Basis:   ,
        System.out.println("b: " + b); // b: Erweitert:     , Basis:   ,

        System.out.println("e.val: " + e.val); // e.val:
        System.out.println("b.val: " + b.val); // b.val:
    }
```

```
}
```

Lösung (Aufgabe 1):

Das Programm gibt folgendes aus:

```
e: Erweitert: 12.0, Basis: 1, 1.0
b: Erweitert: 12.0, Basis: 2, 1.0
e: Erweitert: 24.0, Basis: 3, 2.0
b: Erweitert: 12.0, Basis: 3, 1.0
e: Erweitert: 24.0, Basis: 4, 2.0
b: Erweitert: 12.0, Basis: 4, 3.0
e.val: 24.0
b.val: 3.0
```

Aufgabe 2 (Hoare-Kalkül): (2 + 10 + 2 = 14 Punkte)

Gegeben sei folgender *Java*-Algorithmus P zur Berechnung der doppelten Summe eines `int`-Arrays `a`.

$\langle\varphi\rangle$ (Vorbedingung)

```
i = 0;
res = 0;
while (i < a.length) {
   res = res + a[i];
   i = i + 1;
   res = res + a[i-1];
}
```

$\langle\psi\rangle$ (Nachbedingung)

a) Berechnen Sie zunächst anhand einiger einfacher Beispiele, in welcher Beziehung `res` und `i` bei Überprüfung der Schleifenbedingung zueinander stehen, und geben Sie anschließend eine Schleifeninvariante für die gegebene `while`-Schleife an, die diese Beziehung formal beschreibt. Verwenden Sie dafür die vorgefertigten Tabellen und die gegebene Belegung des Arrays `a`. Sie benötigen hierbei nicht immer alle Zeilen.

`int a[] = {1,2,3};`

res	i

`int a[] = {1,1,1,1};`

res	i

Invariante:

b) Als Vorbedingung für den oben aufgeführten Algorithmus P gelte $true$ und als Nachbedingung

$$res = 2 \cdot \Sigma_{j=0}^{a.length-1} a[j].$$

Vervollständigen Sie auf der nächsten Seite die 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 dann eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

Hinweise:

- Da Arrays keine negative Länge haben können, gilt **immer** $a.length \geq 0$.

c) Beweisen Sie die Terminierung des Algorithmus P. Geben Sie hierzu eine Variante für die `while`-Schleife an. Zeigen Sie, dass es sich tatsächlich um eine Variante handelt, und beweisen Sie damit die Terminierung unter Verwendung des Hoare-Kalküls mit der Voraussetzung $true$.

Lösung (Aufgabe 2):

a)

`int a[] = {1,2,3};`

res	i
0	0
2	1
6	2
12	3

`int a[] = {1,1,1,1};`

res	i
0	0
2	1
4	2
6	3
8	4

Invariante: $i < a.length + 1 \wedge res = 2 \cdot \Sigma_{j=0}^{i-1} a[j]$

b)

$\langle true \rangle$

$\langle 0 = 0 \rangle$

```
i = 0;
```

$\langle i = 0 \rangle$

$\langle i = 0 \wedge 0 = 0 \rangle$

```
res = 0;
```

$\langle i = 0 \wedge res = 0 \rangle$

$\langle i < a.length + 1 \wedge res = 2 \cdot \Sigma_{j=0}^{i-1} a[j] \rangle$

```
while (i < a.length) {
```

$\langle i < a.length \wedge i < a.length + 1 \wedge res = 2 \cdot \Sigma_{j=0}^{i-1} a[j] \rangle$

$\langle i < a.length \wedge res + a[i] + a[i] = 2 \cdot \Sigma_{j=0}^{i} a[j] \rangle$

```
    res = res + a[i];
```

$\langle i < a.length \wedge res + a[i] = 2 \cdot \Sigma_{j=0}^{i} a[j] \rangle$

$\langle i + 1 < a.length + 1 \wedge res + a[(i+1) - 1] = 2 \cdot \Sigma_{j=0}^{(i+1)-1} a[j] \rangle$

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

$\langle i < a.length + 1 \wedge res + a[i - 1] = 2 \cdot \Sigma_{j=0}^{i-1} a[j] \rangle$

```
    res = res + a[i-1];
```

$\langle i < a.length + 1 \wedge res = 2 \cdot \Sigma_{j=0}^{i-1} a[j] \rangle$

```
}
```

$\langle i \not< a.length \wedge i < a.length + 1 \wedge res = 2 \cdot \Sigma_{j=0}^{i-1} a[j] \rangle$

$\langle res = 2 \cdot \Sigma_{j=0}^{a.length-1} a[j] \rangle$

c) Eine Variante ist $V = a.length - i$, denn aus der Schleifenbedingung $i < a.length$ folgt $V = a.length - i \geq 0$. Somit:

$\langle a.length - i = m \wedge i < a.length \rangle$

$\langle a.length - i = m \rangle$

```
    res = res + a[i];
```

$\langle a.length - i = m \rangle$

$\langle a.length - (i + 1) = m - 1 \rangle$

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

$\langle a.length - i = m - 1 \rangle$

$\langle a.length - i < m \rangle$

```
    res = res + a[i];
```

$\langle a.length - i < m \rangle$

Damit ist die Terminierung der einzigen Schleife in P gezeigt.

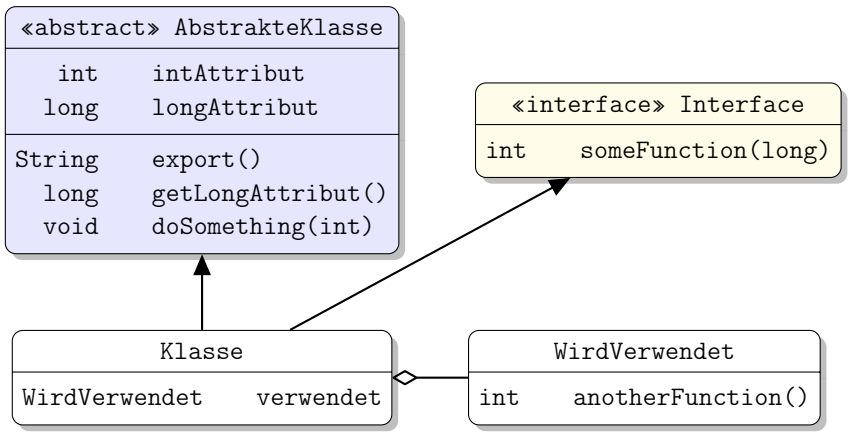
Aufgabe 3 (Telefone): (6 + 8 + 5 = 19 Punkte)

In dieser Aufgabe betrachten wir verschiedene Arten von Telefonen. Sie sollen in den folgenden Teilaufgaben eine Klassenhierarchie erstellen sowie eine Ausnahmeklasse und zwei Methoden implementieren.

a) In dieser Teilaufgabe geht es um den Entwurf einer entsprechenden Klassenhierarchie, mit der die verschiedenen Arten von Telefonen und ihren Komponenten sinnvoll in einem Programm gehandhabt werden können.

- Festnetztelefone sind Telefone, die nur an einem bestimmten Standort genutzt werden können, da sie mit dem Festnetz verbunden sind. Für diese Telefone ist der Standort bekannt.
- Ein Akku speichert seinen Füllstand in Prozent.
- Bei allen Telefonen wird die Lautstärke ganzzahlig in Dezibel gespeichert.
- Ein Teil der Festnetztelefone ist stationär, d.h., sie sind fest an einem Standort, wie z.B. der Wohnung oder dem Büro, installiert und können nicht mitgenommen werden.
- Als Gegensatz zu Festnetztelefonen gibt es Mobiltelefone. Jedes Mobiltelefon benutzt einen Akku und die Anzahl der Tasten ist bekannt.
- Es gibt auch Mobiltelefone mit Touchscreen. Bei diesen kann man die Eingaben per Touchscreen und per Tasten vornehmen. Für diese Telefone ist die Diagonale des Touchscreens bekannt.
- Die Festnetztelefone, die nicht stationär sind, sind drahtlose Telefone. Solche Telefone können in einem bestimmten Radius um den Standort herum genutzt werden und benutzen einen Akku. Die Reichweite in Metern soll gespeichert werden.
- Die Telefone mit Akku sind aufladbar und stellen die Methode getLadestand() bereit, mit der der Ladestand (0-100) des Akkus abgefragt werden kann. Außerdem haben sie eine Methode aufladen(), die den Akku auflädt und zurückgibt, ob das Aufladen erfolgreich war.
- Ein öffentliches Telefon ist ein stationäres Telefon, das jeder benutzen kann. Für dieses Telefon sind die Kosten pro Zeiteinheit in Cent bekannt.

Entwerfen Sie eine geeignete Klassenhierarchie für die beschriebenen Sachverhalte. Notieren Sie **keine Konstruktoren, Getter und Setter**. Sie müssen **nicht markieren**, ob Attribute `final` sein sollen oder welche Zugriffsrechte (also z.B. `public` oder `private`) für sie gelten sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen zusammengefasst werden. Notieren Sie Ihren Entwurf graphisch und verwenden Sie dazu die folgende Notation:



Eine Klasse wird hier durch einen Kasten beschrieben, in dem der Name der Klasse sowie Felder und Methoden in einzelnen Abschnitten beschrieben werden. Weiterhin bedeutet der Pfeil $B \rightarrow A$, dass A die Oberklasse von B ist (also `class B extends A` bzw. `class B implements A`, falls A ein Interface ist) und $A \diamond\!\!- B$, dass A den Typ B verwendet (z.B. als Typ eines Feldes oder in einem Array).

Tragen Sie keine vordefinierten Klassen (String, etc.) als Klassen in Ihr Diagramm ein, Sie dürfen sie aber als Attributtyp verwenden. Geben Sie für jede Klasse ausschließlich den jeweiligen Namen und die Namen und Datentypen ihrer Attribute an. Methoden müssen nur in dem allgemeinsten Typen (d.h. Klasse oder Interface) angegeben werden, in dem sie deklariert werden. Implementierungen dieser Methoden müssen dann nicht mehr explizit angegeben werden.

b) Alle Telefone mit Akku sollen aufgeladen werden. Implementieren Sie in Java eine Methode `alleAufladen`, die ein Array vom Typ `Telefon[]` übergeben bekommt und dann alle übergebenen Telefon auflädt, wenn sie einen Akku enthalten und wenn der Ladestand (0-100) niedriger als 20 ist. Es soll zurückgegeben werden, wie viele Ladevorgänge erfolgreich waren.

Gehen Sie davon aus, dass die Methoden `getLadestand()` und `aufladen()` für die Telefone mit Akku bereits implementiert wurden; Sie können diese einfach verwenden. Setzen Sie voraus, dass das übergebene Array immer existiert und nie die `null`-Referenz ist. Kennzeichnen Sie die Methode mit dem Schlüsselwort `static`, falls dies angebracht ist.

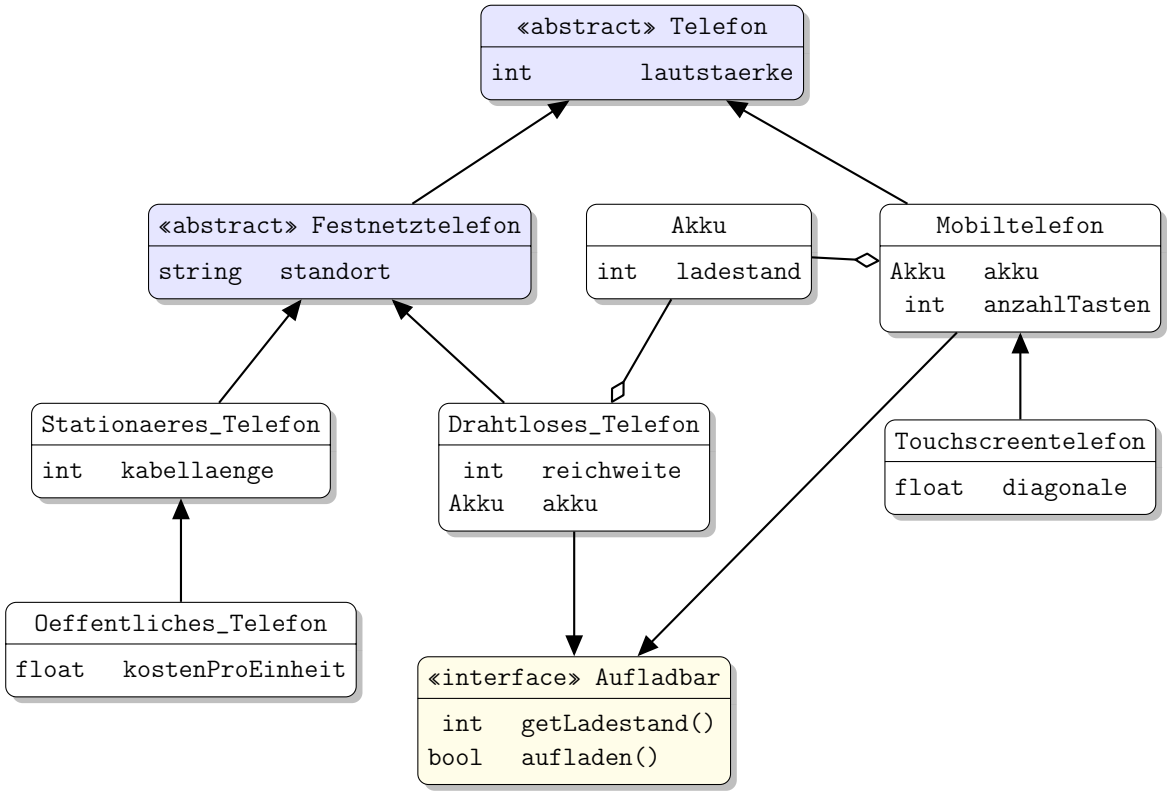
c) Wenn man ein öffentliches Telefon benutzen möchte, muss man Münzen einwerfen. Das Telefon überprüft, ob diese Münzen für mindestens eine Gesprächseinheit ausreichen.

i) Implementieren Sie eine Ausnahme `BetragZuGeringException`. Diese Klasse erweitert die Ausnahmeklasse `Exception` und soll die Differenz speichern, die zum Mindestbetrag fehlt. Erstellen Sie einen passenden Konstruktor, der diesen Wert setzt.

ii) Implementieren Sie die Methode `bezahlen` in der Klasse für öffentliche Telefone. Dieser Methode wird ein Array von Centbeträgen übergeben und sie hat keinen Rückgabewert. Wenn die Summe des Betrags geringer als der Mindestbetrag (Kosten für eine Einheit) ist, dann soll diese Methode die Ausnahme `BetragZuGeringException` werfen, sodass die Differenz zum Mindestbetrag gespeichert wird. Verwenden Sie den Konstruktor aus i). Setzen Sie voraus, dass das übergebene Array immer existiert und nie die `null`-Referenz ist. Kennzeichnen Sie die Methode mit dem Schlüsselwort `static`, falls dies angebracht ist.

Lösung (Aufgabe 3):

a) Eine mögliche Modellierung der geschilderten Zusammenhänge ist diese:



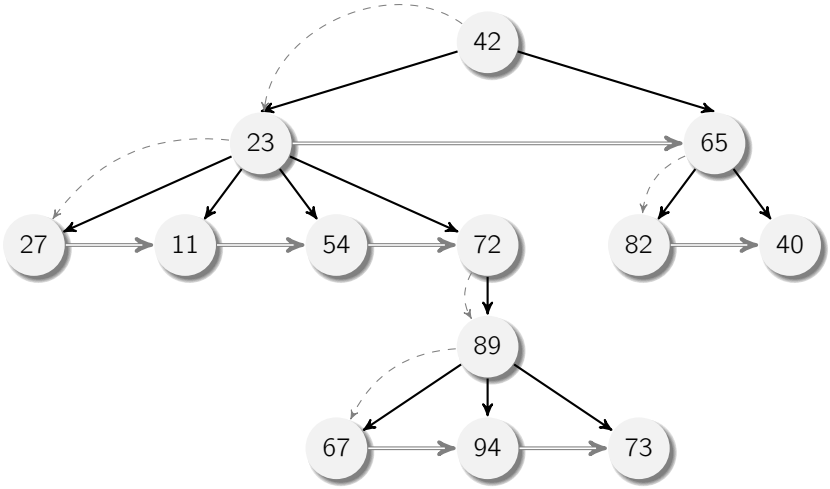
b)
```
public static bool alleAufladen(Telefon[] telefon) {
  int res = 0;
  for (int i = 0; i < telefon.length; i++) {
    if (telefon[i] instanceof Aufladbar) {
      if ((Aufladbar) telefon[i]).getLadestand()<20) {
        if (((Aufladbar) telefon[i]).aufladen()) {
          res = res + 1;
        }
      }
    }
  }
  return res;
}
```

c)
```
public class BetragZuGeringException extends Exception {
  int differenz;

  BetragZuGeringException(int diff) {
    this.differenz = diff;
  }
}

public void bezahlen(int[] muenzen) throws BetragZuGeringException {
  int betrag = 0;

  for (int i=0; i < muenzen.length; ++i) {
    betrag = betrag + muenzen[i];
  }
```

```
    if (betrag < this.kostenProEinheit) {
      throw new BetragZuGeringException(this.kostenProEinheit - betrag);
    }
  }
```

Aufgabe 4 (Bäume mit beliebig vielen Kindern): (6 + 8 + 9 = 23 Punkte)

Bäume, in denen jeder Knoten beliebig viele Nachfolger haben kann, sind eine Verallgemeinerung der Binärbäume, die in der Vorlesung vorgestellt wurden. Diese Bäume werden Vielwegebäume genannt. Ein solcher Baum ist in der folgenden Graphik illustriert:



Wir stellen solche Bäume mit der folgenden Datenstruktur dar:

```
public class MTree {
  public MTree nextSibling;
  public MTree firstChild;
  public int value;
}
```

Wir speichern also jeweils nur einen Verweis auf das erste der Kinder eines Knoten (in `firstChild`, in der Graphik als grauer, gestrichelter Pfeil dargestellt) und zusätzliches einen Verweis auf den nächsten Geschwisterknoten (in `nextSibling`, in der Graphik als grauer, doppelter Pfeil dargestellt).

a) Implementieren Sie in Java eine Methode `size` in der Klasse `MTree`, die die Anzahl der Knoten eines Baums zurückgibt. Für den Beispielbaum soll `size` also 13 zurückgeben. Verwenden Sie dazu **ausschließlich Rekursion**, also keine Schleifenkonstrukte.

b) Betrachten Sie folgende Klasse für einfach verkettete Listen von `int`-Werten:

```
public class List {
  int val;
  List next;

  public List(int v, List n) {
    this.val = v;
    this.next = n;
  }

  public boolean contains(int v) {
    ...
  }
}
```

Wir wollen eine Liste aller in einem Baum vorkommenden Werte bilden, wobei keiner der Werte doppelt auftauchen soll. Sie dürfen dazu die Methode `public boolean contains(int v)` in der Klasse `List` verwenden, die genau dann `true` zurückgibt, wenn der Wert `v` bereits in der Liste enthalten ist.

Implementieren Sie in Java eine Methode `List getValues()` in der Klasse `MTree`, die eine Liste vom Typ `List` zurückgibt. Diese Liste soll jeden Wert, der im Baum gespeichert ist, genau einmal enthalten. Die Reihenfolge der Elemente ist egal. Verwenden Sie dazu **ausschließlich Rekursion**, also keine Schleifenkonstrukte.

Hinweise:

- Implementieren und verwenden Sie eine Hilfsmethode `List getValues(List res)`, die den bereits berechneten Teil der Ergebnisliste übergeben bekommt und eine gegebenenfalls angepasste (erweiterte) Liste zurückgibt.
- Fügen Sie Werte immer an den Anfang der Ergebnisliste ein.
- Überprüfen Sie vor dem Hinzufügen, ob der hinzuzufügende Wert bereits in der Liste enthalten ist.
- Beachten Sie, dass die Ergebnisliste anfangs `null` ist!

c) Implementieren Sie in Java eine Methode `findValue`, welche einen Pfad übergeben bekommt. Dieser Pfad beschreibt, in welcher Reihenfolge Knoten im `MTree` besucht werden. Der Pfad wird als `int`-Array $\{c_1, \ldots, c_n\}$ übergeben, wobei der Wert $c_i, 1 \leq i \leq n$, jeweils spezifiziert, in welches Kind abgestiegen werden soll. Die Methode `findValue` soll den entsprechenden Wert des letzten besuchten Knotens zurückgeben.

So soll für den Beispielbaum für den leeren Pfad $\{\}$ das Wurzelelement 42 zurückgegeben werden. Für den Pfad $\{1, 4, 1, 2\}$ soll zuerst in den ersten Nachfolgerknoten mit Wert 23 abgestiegen werden, dann in den vierten (72), dann in den ersten (89) und zuletzt in den zweiten (94). Im Beispiel würde also der Wert 94 zurückgegeben werden.

Gibt es kein entsprechendes Kind, soll der Wert des letzten auf dem Pfad existierenden Elementes zurückgegeben werden. So soll beispielsweise für den Pfad $\{2, 3\}$ der Wert 65 zurückgegeben werden, da der Pfad $\{2, 3\}$ nicht existiert, aber $\{2\}$ noch existiert.

Verwenden Sie zur Lösung dieser Aufgabe **keine Rekursion**, sondern nur Schleifen. Sie dürfen aber geeignete iterative Hilfsmethoden schreiben. Setzen Sie voraus, dass das übergebene Array immer existiert und nie die `null`-Referenz ist. Kennzeichnen Sie die Methode mit dem Schlüsselwort static, falls dies angebracht ist.

Hinweise:

- Sie benötigen eine Schleife, um alle Elemente c_i des Pfades zu bearbeiten und eine weitere Schleife, um das c_i-te Kind eines Knoten zu ermitteln.
- Ein Pfad existiert dann nicht in einem Baum, wenn auf diesem Pfad ein Element 0 oder negativ ist, er noch weitere Elemente enthält, obwohl bereits ein Blatt erreicht wurde oder ein Element c_i größer ist als die Zahl der Kinder, die der Knoten hat, der durch $\{c_1, \ldots, c_{i-1}\}$ erreicht wird.

Lösung (Aufgabe 4):

a)
```
public static int size(final MTree t) {
  if (t == null) {
    return 0;
  }

  int size = 1;
  size += size(this.firstChild);
  size += size(this.nextSibling);
  return size;
}
```

```
  public int size() {
    int size = 1;
    if (this.firstChild != null) {
      size += this.firstChild.size();
    }
    if (this.nextSibling != null) {
      size += this.nextSibling.size();
    }
    return size;
  }
```

b)
```
  public List getValues() {
    return this.getValues(null);
  }

  public List getValues(List res) {
    if (this.firstChild != null) {
      res = this.firstChild.getValues(res);
    }
    if (this.nextSibling != null) {
      res = this.nextSibling.getValues(res);
    }

    if (res == null || !res.contains(this.value)) {
      return new List(this.value, res);
    } else {
      return res;
    }
  }
```

c)
```
  public int findValue(int[] path) {
    MTree resNode = this;
    for (int i = 0; i < path.length; ++i) {
      int whichChild = path[i];
      if (whichChild < 0 || resNode.firstChild == null) {
        return resNode.value;
      }

      MTree childI = resNode.firstChild;
      for (int childIdx = 0; childIdx < whichChild; ++childIdx) {
        if (childI.nextSibling == null) {
          return resNode.value;
        }
        childI = childI.nextSibling;
      }
      resNode = childI;
    }
    return resNode.value;
  }
```

Aufgabe 5 (Haskell): (3.5 + 1.5 + (2 + 1 + 5 + 5) = 18 Punkte)

a) Geben Sie den allgemeinsten Typ der Funktionen `f` und `g` an, die wie folgt definiert sind. Gehen Sie davon aus, dass `1` den Typ `Int` hat.

```
f x y z = (x (y + 1)) ++ z
```

```
g x y = \y -> [x]
```

b) Gegeben sei das folgende Programm in Haskell:

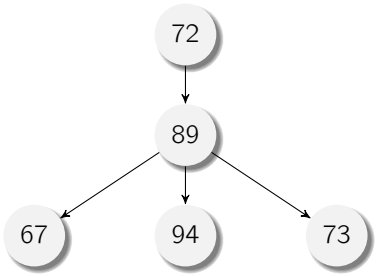
```
h xs = foldr (\x y -> x + y) 0 (map (\x -> 2 * x) xs)
```

Geben Sie das Ergebnis für den Aufruf `h [1,2,3]` an:

c) Wir betrachten Vielwegebäume wie in Aufgabe 4.

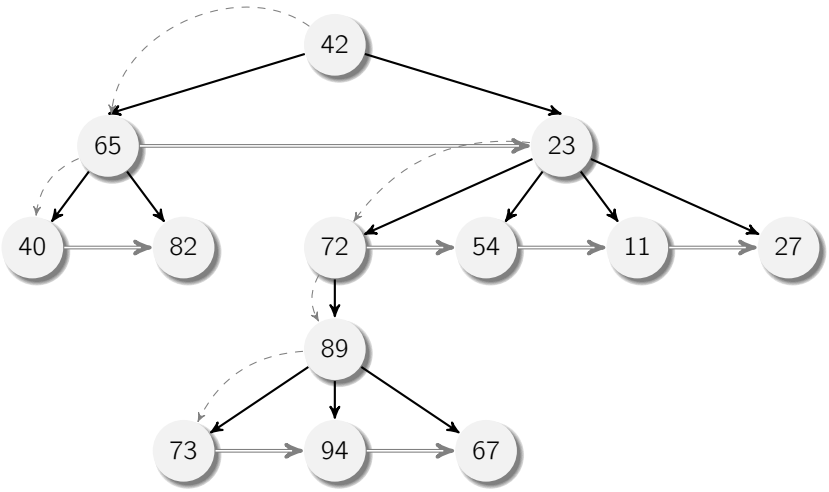
i) Geben Sie die Definition eines Datentyps `MTree` für solche Bäume in Haskell an. Verwenden Sie dazu jeweils einen eigenen Konstruktor für leere Bäume und einen für Knoten mit einem `Int`-Wert und beliebig vielen Nachfolgern. Verwenden Sie die in Haskell vordefinierten Listen.

ii) Geben Sie für den folgenden Baum einen `MTree`-Ausdruck an, der den Baum in Ihrer Datenstruktur kodiert.



iii) Implementieren Sie in Haskell eine Funktion `size :: MTree -> Int`, die die Anzahl der Knoten eines Baums bestimmt. Sie dürfen dazu beliebige vordefinierte Funktionen verwenden, müssen dies aber nicht tun.

iv) Implementieren Sie in Haskell eine Funktion `mirror :: MTree -> MTree`, die in jedem Knoten des Baums die Reihenfolge der Elemente vertauscht. Sie dürfen dazu beliebige vordefinierte Funktionen verwenden, müssen dies aber nicht tun. Der Baum aus Aufgabe 4 soll nach Anwendung von `mirror` wie folgt aussehen:



Lösung (Aufgabe 5):

a)
```
f :: (Int -> [a]) -> Int -> [a] -> [a]
f x y z = (x (y + 1)) ++ z

g :: a -> b -> c -> [a]
g x y = \y -> [x]
```

In `f` muss `y` den Typ `Int` haben, da es ein Operand von + ist. Der Ausdruck `y + 1` hat ebenfalls den Typ `Int`. Da `x` diesen Ausdruck als Argument bekommt, muss `x` eine Funktion sein, die einen `Int`-Parameter erwartet. Weiterhin wird das Ergebnis von `x` mit `z` mittels ++ verbunden, daher müssen beide eine Liste vom selben Typ `[a]` sein. Damit ist `x` eine Funktion `Int -> [a]` und der Rückgabewert der Funktion `[a]`.

In `g` wird eine Funktion zurückgegeben, die einen nicht näher bestimmten Parameter erwartet und als Resultat eine Liste zurückgibt, die `x` enthält.

b) Die Auswertung des ersten Ausdrucks erfordert das Anwenden von `map (\x -> 2 * x)` auf `[1,2,3]`, d.h., jedes Element wird verdoppelt und das Resultat ist `[2,4,6]`. Der `foldr`-Ausdruck bildet dann die Summe der Elemente, d.h. 12.

c)
```
-- i)
data MTree = Node Int [MTree] | Empty deriving Show

-- ii)
bsp :: MTree
bsp = Node 72 [Node 89 [Node 67 [], Node 94 [], Node 73 []]]

-- iii)
size :: MTree -> Int
size Empty = 0
size (Node _ []) = 1
size (Node v (child:children)) = (size child) + (size (Node v children))

-- iii) alternative (using the Prelude):
size' :: MTree -> Int
size' Empty = 0
size' (Node _ children) = sum (map size children) + 1

-- iv)
mirror :: MTree -> MTree
mirror Empty = Empty
mirror (Node i children) = Node i (revMirror children)
  where
    revMirror [] = []
    revMirror (m:ms) = revMirror ms ++ [mirror m]

-- iv) alternative (using the Prelude):
mirror' :: MTree -> MTree
mirror' Empty = Empty
mirror' (Node i ms) = Node i (reverse (map mirror' ms))
```

Aufgabe 6 (Prolog): (2 + 4 + (2 + 4 + 4) = 16 Punkte)

a) Geben Sie für die folgenden Paare von Termen einen allgemeinsten Unifikator an oder begründen Sie kurz, warum dieser nicht existiert. Verwenden Sie dazu den Algorithmus aus der Vorlesung und geben Sie Teilsubstitutionen σ_i an.

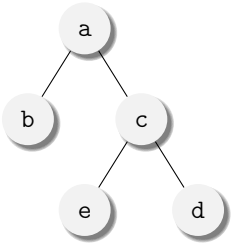
`f(Z,Y,X)` und `f(X,b,g(Z))`

`o(X,h(Z),Z,Y)` und `o(a,Y,b,h(g(X)))`

b) Erstellen Sie für das folgende Logikprogramm zur Anfrage `?- s(a, U).` den Beweisbaum und geben Sie alle Antwortsubstitutionen an. Sie dürfen dabei abbrechen, sobald die Anfrage aus mindestens vier Teilen (Atomen) bestehen würde. Kennzeichnen Sie solche Knoten durch „...“. Kennzeichnen Sie abgebrochene Pfade durch „$\lightning$“.

```
t(a,c).
s(X,Y) :- s(Y,X),t(X,Y).
s(X,Z) :- t(X,Z).
```

c) Binäre Bäume lassen sich in Prolog durch ein dreistelliges Funktionssymbol `baum` darstellen, welches als erstes und drittes Argument den linken bzw. rechten Teilbaum und als zweites Argument einen Wert bekommt. Ein leerer Baum kann als `nil` repräsentiert werden. Auf diese Weise, entspricht der unten dargestellte Baum dem Prolog-Term `baum(baum(nil,b,nil),a,baum(baum(nil,e,nil),c,baum(nil,d,nil)))`. Ein Pfad in einem solchen Baum lässt sich durch eine Liste von `links`/`rechts` Symbolen darstellen, die beschreiben, wie man von der Wurzel aus zu einer Position im Baum gelangt. So kommt man in diesem Baum durch den Pfad `[links]` zu dem Teilbaum `baum(nil,b,nil)` und durch den Pfad `[rechts, links]` zu dem Teilbaum `baum(nil,e,nil)`.



i) Programmieren Sie das dreistellige Prädikat `teilbaum`, welches den Teilbaum an einer bestimmten Position eines Baumes ausrechnen kann. Das Prädikat bekommt als erstes Argument einen Baum, als zweites einen Pfad und als drittes einen weiteren Baum. Es ist erfüllt, falls der zweite Baum an der durch den Pfad gekennzeichneten Position des ersten Baumes steht.

Beispielanfragen:

```
?- teilbaum(baum(baum(Y,a,Z),b,A),[links],X).
X = baum(Y, a, Z).

?- teilbaum(baum(nil,b,R),[links, links],X).
false.
```

ii) Programmieren Sie das vierstellige Prädikat `ersetzeTeilbaum`, so dass ein Aufruf von `ersetzeTeilbaum` mit den Argumenten (*Eingabe*, *Pfad*, *Ersetzung*, *Ergebnis*) genau dann erfolgreich ist, falls *Ergebnis* der Baum *Eingabe* ist, bei dem der Teilbaum an Position *Pfad* durch den Baum *Ersetzung* ersetzt wurde.

Beispielanfragen:

```
?- ersetzeTeilbaum(baum(nil,b,X), baum(nil,a,nil), [links], R).
R = baum(baum(nil, a, nil), b, X).

?- ersetzeTeilbaum(baum(nil,b,X), baum(nil,a,nil), [links, rechts], R).
false.
```

iii) Programmieren Sie das dreistellige Prädikat `werteAnPositionen`, welches einen Baum, eine Liste von Pfaden sowie eine gleichlange Liste von Werten bekommt und überprüft, ob an den durch die Pfade beschriebenen Stellen im Baum die jeweiligen Werte der Liste stehen. Das heißt, dass wenn an der n-ten Stelle der Liste der Pfad p steht, das Prädikat überprüfen soll, ob im übergebenen Baum an der durch den Pfad p beschriebenen Position der n-te Wert der Werteliste steht.

Hinweise:

- Verwenden Sie hierbei das Prädikat `teilbaum` aus Aufgabenteil i).

Beispielanfragen:

```
?- werteAnPositionen(baum(baum(a, X, Y), Z, baum(b, W, c)),
                     [[links,links],[rechts,rechts]], [a,c]).
true.

?- werteAnPositionen(baum(baum(a, X, Y), Z, baum(b, W, c)),
                     [[links,links],[rechts,Wo]], [a,b]).
Wo = links.
```

Lösung (Aufgabe 6):

a) (i) $\sigma_1 = \{Z = X\}$

$\sigma_2 = \{Y = b\}$

σ_3 existiert nicht, da X und $g(X)$ nicht unifiziert werden können. Folglich liegt ein *occur failure* vor.

(ii) $\sigma_1 = \{X = a\}$

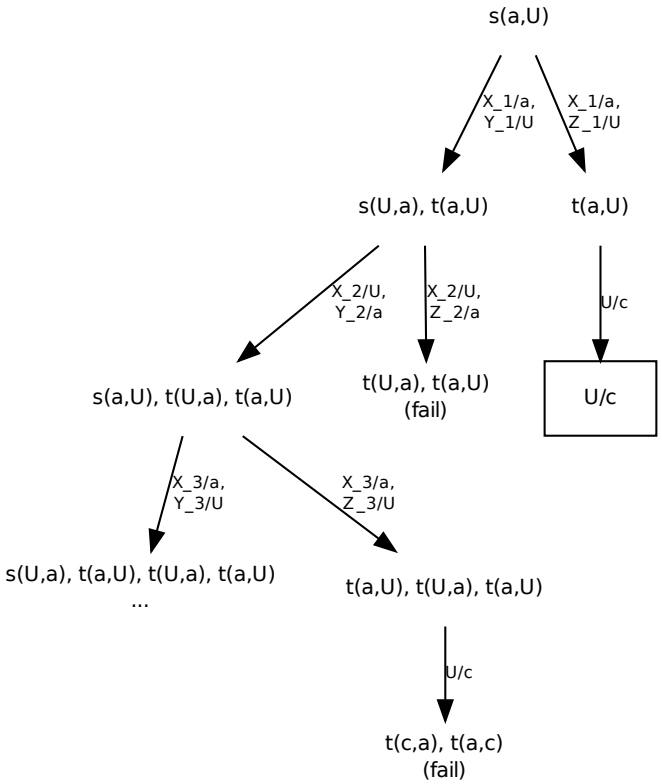
$\sigma_2 = \{Y = h(Z)\}$

$\sigma_3 = \{Z = b\}$

σ_4 existiert nicht, da $h(b)$ und $h(g(a))$ zu einem *clash failure* führen.

b) Die Lösung ist die Substitution:

- X = c



c)
```
teilbaum(X,[],X).
teilbaum(baum(L,_,_), [links|P], Z) :- teilbaum(L,P,Z).
teilbaum(baum(_,_,R), [rechts|P], Z) :- teilbaum(R,P,Z).

ersetzeTeilbaum(_, E, [], E).
ersetzeTeilbaum(baum(L,W,R), E, [links|P], baum(L2,W,R)) :-
    ersetzeTeilbaum(L, E, P, L2).
ersetzeTeilbaum(baum(L,W,R), E, [rechts|P], baum(L,W,R2)) :-
    ersetzeTeilbaum(R, E, P, R2).

werteAnPositionen(_,[],[]).
werteAnPositionen(B,[P|PS],[W|WS]) :-
    teilbaum(B,P,W), werteAnPositionen(B,PS,WS).
```