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

Aufgabe 1 (Programmanalyse): (10 Punkte)

Gegeben sei das folgende Java-Programm. Das Programm gibt einige Zeilen Text aus. Dabei wird für Objekte vom Typ `A` jeweils der Wert des Attributes `s` in der Form `s=12.3` ausgegeben. Bei Objekten des Typs `B` wird zuerst die Ausgabe von `A` angezeigt und dann der Rückgabewert der Methode `get` in der Form `value=123` ausgegeben. Tragen Sie die Ausgabe an den markierten Stellen in den Kommentaren ein.

```
class A {
  double s;
  A()                { set(0); }
  A(double x)        { set(x); }
  A(A x)             { set(x.get()); }
  A f(A x)           { set(s + x.get()); return this; }
  A f(double x)      { set(s + x); return this; }
  void set(double x) { s = x; }
  double get()       { return s; }
  public String toString() {
    return "A(s=" + s + ")";
  }
}

class B extends A {
  int s, t;
  B(int x, int y) { s = x; t = y; }
  double get()    { return s / (double)t; }
  B f(A x)        { s += t * x.get(); return this; }
  B f(int x)      { s += t * x; return this; }
  public String toString() {
    return "B(" + super.toString() + ", value=" + get() + ")";
  }
}

class Programm {
  public static void main(String[] p) {
    A z1 = new A(3);
    System.out.println("z1: " + z1); // z1: A(s=     )

    A z2 = new B(12, 4);
    System.out.println("z2: " + z2); // z2: B(A(s=     ), value=     )

    A z3 = new A(z2.f(z1));
    System.out.println("z2: " + z2); // z2: B(A(s=     ), value=     )
    System.out.println("z3: " + z3); // z3: A(s=     )

    A z4 = new A(z2.f(1.0).f(z3));
    System.out.println("z2: " + z2); // z2: B(A(s=     ), value=     )
    System.out.println("z4: " + z4); // z4: A(s=     )

    z4.f(4);
    System.out.println("z4: " + z4); // z4: A(s=     )
  }
}
```

Lösung (Aufgabe 1):

Das Programm gibt folgendes aus:

```
z1: A(s=3.0)
z2: B(A(s=0.0), value=3.0)
z2: B(A(s=0.0), value=6.0)
z3: A(s=6.0)
z2: B(A(s=1.0), value=12.0)
z4: A(s=12.0)
z4: A(s=16.0)
```

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

Gegeben sei folgender Java-Algorithmus P zur Berechnung der Division mit Rest der natürlichen Zahlen x und y, sodass für den berechneten Quotienten e und den Rest r am Ende $e \cdot y + r = x$ und $0 \leq r < y$ gilt.

$\langle\varphi\rangle$ (Vorbedingung)

```
e = 0;
r = x;
while (r >= y) {
   e = e + 1;
   r = r - y;
}
```

$\langle\psi\rangle$ (Nachbedingung)

a) Berechnen Sie zunächst anhand einiger einfacher Beispiele, in welcher Beziehung `e`, `r`, `x` und `y` 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. Sie benötigen hierbei nicht immer alle Zeilen.

`int x = 9; int y = 3;`

e	r

`int x = 5; int y = 2;`

e	r

Invariante:

b) Als Vorbedingung für den oben aufgeführten Algorithmus P gelte $y > 0 \wedge x \geq 0$ und als Nachbedingung

$$e = \lfloor x/y \rfloor \wedge r = x \bmod y.$$

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:

- $x \bmod y$ bezeichnet den Rest der Division von x durch y.
- Es gilt $e = \lfloor x/y \rfloor \wedge r = x \bmod y$ gdw. $e \cdot y + r = x \wedge 0 \leq r < y$ für alle $x, y, e, r \in \mathbb{N}$.

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 $y > 0 \wedge x \geq 0$.

Lösung (Aufgabe 2):

a)

`int x = 9; int y = 3;`

e	r
0	9
1	6
2	3
3	0

`int x = 5; int y = 2;`

e	r
0	5
1	3
2	1

Invariante: $e \cdot y + r = x \wedge 0 \leq r$

b)

$\langle y > 0 \wedge x \geq 0 \rangle$
$\langle y > 0 \wedge x \geq 0 \wedge 0 = 0 \rangle$

```
e = 0;
```

$\langle y > 0 \wedge x \geq 0 \wedge e = 0 \rangle$
$\langle y > 0 \wedge x \geq 0 \wedge e = 0 \wedge x = x \rangle$

```
r = x;
```

$\langle y > 0 \wedge x \geq 0 \wedge e = 0 \wedge r = x \rangle$
$\langle e \cdot y + r = x \wedge 0 \leq r \rangle$

```
while (r >= y) {
```

$\langle r \geq y \wedge e \cdot y + r = x \wedge 0 \leq r \rangle$
$\langle (e+1) \cdot y + r - y = x \wedge 0 \leq r - y \rangle$

```
    e = e + 1;
```

$\langle e \cdot y + r - y = x \wedge 0 \leq r - y \rangle$

```
    r = r - y;
```

$\langle e \cdot y + r = x \wedge 0 \leq r \rangle$

```
}
```

$\langle r \not\geq y \wedge e \cdot y + r = x \wedge 0 \leq r \rangle$
$\langle e = \lfloor x/y \rfloor \wedge r = x \mod y \rangle$

c) Der dritte Aufgabenteil wurde komplett aus der Wertung genommen, da er mit den Mitteln der Vorlesung nicht lösbar ist.

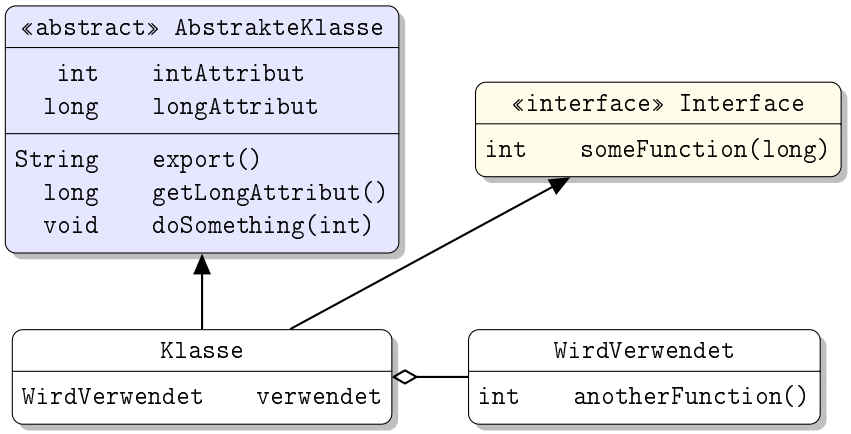
Aufgabe 3 (Stifte): (8 + 6 = 14 Punkte)

In dieser Aufgabe betrachten wir verschiedene Arten von Stiften. Sie sollen in den folgenden Teilaufgaben eine Klassenhierarchie erstellen und eine Methode implementieren.

a) In dieser Teilaufgabe geht es um den Entwurf einer entsprechenden Klassenhierarchie, mit der die verschiedenen Arten von Stiften, ihre Eigenschaften und ein Benutzer dieser Klassen sinnvoll in einem Programm gehandhabt werden können.

- Für alle Stifte ist die Farbe bekannt, die durch ihren Farbcode, eine natürliche Zahl, dargestellt wird.
- Es gibt nachfüllbare Stifte. Für diese ist die Methode `nachfuellen()` implementiert. Diese gibt zurück, ob das Nachfüllen erfolgreich war. Zudem wird im Attribut `nachfuelltyp` als `String` gespeichert, was getauscht (z.B. eine Kugelschreibermine) bzw. nachgefüllt (z.B. Tinte) wird.
- Jeder dokumentenechte Stift muss die Methode `unterschreiben()` implementieren. Diese soll keinen Rückgabewert haben.
- Ein Füller ist ein nachfüllbarer Stift. Es soll gespeichert werden, ob es sich bei dem Füller um einen speziellen Linkshänderfüller handelt.
- Im Gegensatz zu den nachfüllbaren Stiften gibt es auch die nicht nachfüllbaren Stifte. Für diese Stifte soll gespeichert werden, wieviele Tage sie bereits in Benutzung sind.
- Jeder Füller ist dokumentenecht und darf somit zum Unterschreiben genutzt werden.
- Ein Etui enthält Stifte in einem Array vom Typ `Stift[]`.
- Ein Kugelschreiber ist ein nachfüllbarer Stift und auch dokumentenecht.
- Bleistifte sind weder nachfüllbar noch dokumentenecht. Für jeden Bleistift ist bekannt, ob am Bleistiftende ein Radiergummi befestigt ist.
- Filzstifte sind nicht nachfüllbar, aber dokumentenecht. Die Stärke des Strichs ist bekannt.
- Druckbleistifte sind wie Bleistifte, aber im Gegensatz zu diesen nachfüllbar. Der Durchmesser der verwendeten Mine soll gespeichert werden.

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 $B \longrightarrow A$, 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) Ein Etui enthält ein Array von Stiften. Es soll nun ein Stift zurückgegeben werden, mit dem Sie die Klausur schreiben dürfen. Implementieren Sie in Java eine Methode `getKlausurstift()` in der Klasse `Etui`.

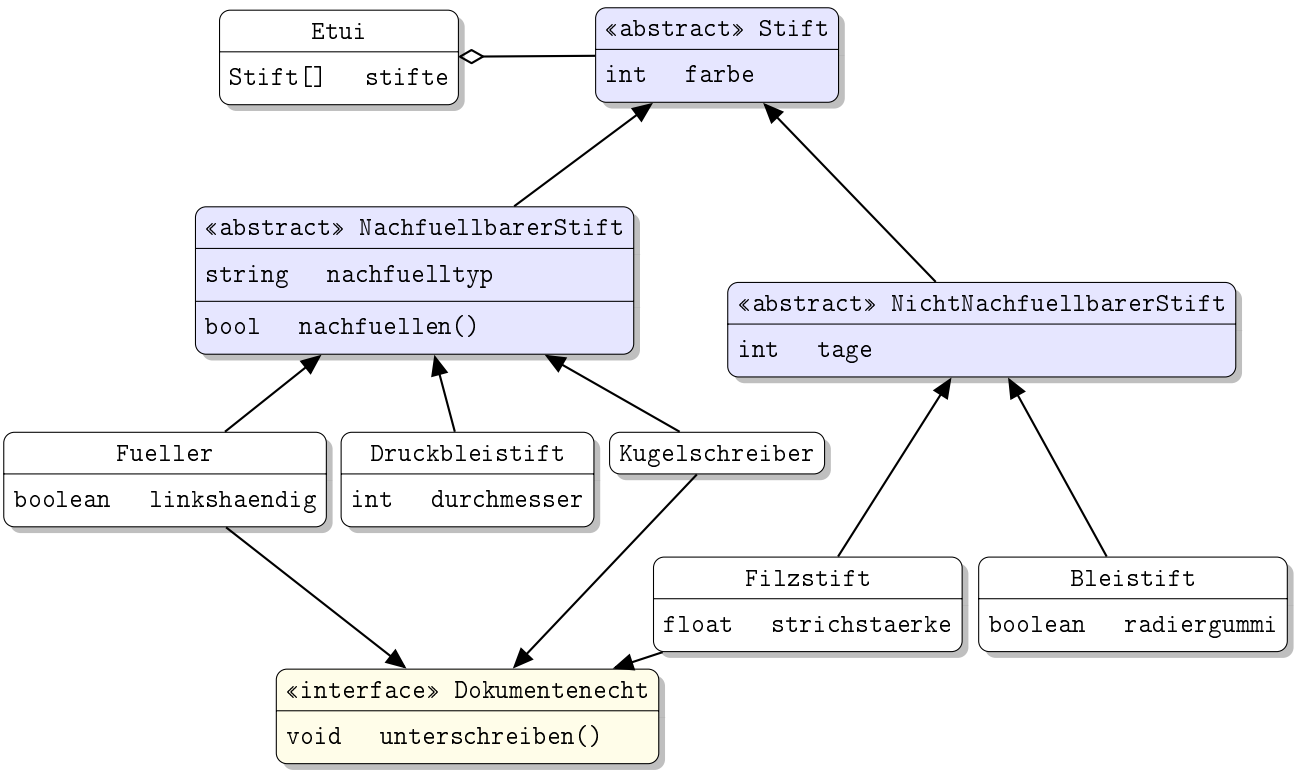
Diese Methode überprüft die im `Etui`-Objekt gespeicherten Stifte und gibt dann den ersten Stift zurück, der dokumentenecht ist und nicht die Farbe rot (Farbcode 3) oder grün (Farbcode 5) hat.

Bevor ein nachfüllbarer Stift zurückgegeben wird, soll er nachgefüllt werden. Gehen Sie davon aus, dass die Methode `nachfuellen()` für die nachfüllbaren Stifte bereits implementiert wurde und immer erfolgreich ist; Sie können diese Methode einfach verwenden.

Setzen Sie voraus, dass das Array immer existiert und nie die `null`-Referenz ist. Falls es keinen entsprechenden Stift gibt, soll die Methode `null` zurückgeben. Kennzeichnen Sie die Methode mit dem Schlüsselwort `static`, falls dies angebracht ist.

Lösung (Aufgabe 3):

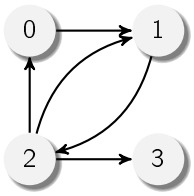
a) Eine mögliche Modellierung der geschilderten Zusammenhänge ist diese:



b)
```
public Stift getKlausurstift() {
  for (int i = 0; i < this.stifte.length; i++) {
    Stift stift = this.stifte[i];
    if (stift instanceof dokumentenecht) {
      if (stift.farbe!=3 || stift.farbe!=5) {
        if (stift instanceof Nachfuellbarer_Stift) {
          ((Nachfuellbarer_Stift) stift).nachfuellen();
        }
        return stift;
      }
    }
  }
  return null;
}
```

Aufgabe 4 (Graphen): (5 + 9 + (5 + 8) = 27 Punkte)

In dieser Aufgabe wollen wir gerichtete Graphen darstellen und zwei Algorithmen implementieren. Ein gerichteter Graph besteht aus einer Menge von Knoten und einer Menge von Kanten, die diese Knoten verbinden. In dieser Aufgabe werden wir die Knoten von 0 bis n durchnummerieren. Ein Graph mit der Knotenmenge $\{0, 1, 2, 3\}$ kann zum Beispiel wie folgt aussehen:



Hier gibt es also eine Kante von Knoten 0 nach 1 (wir schreiben dafür $0 \to 1$) und weitere Kanten $1 \to 2$, $2 \to 0$, $2 \to 1$ und $2 \to 3$. Wir stellen die Graphen als eine (verkettete) Liste von Kanten dar und speichern die Knotenmenge nicht explizit. Wir verwenden die folgenden Klassen, um die Datenstruktur dazustellen:

```
public class Graph {
  EdgeList edges;

  public Graph(EdgeList e) {
    this.edges = e;
  }
}

public class EdgeList {
  Edge edge;
  EdgeList next;

  public EdgeList(Edge e, EdgeList n) {
    this.edge = e;
    this.next = n;
  }
}

public class Edge {
  int start;
  int end;

  public Edge(int s, int e) {
    this.start = s;
    this.end = e;
  }
}
```

a) Schreiben Sie eine Methode `anzahlKanten`, die für ein `Graph`-Objekt die Anzahl der Kanten bestimmt. Für den Beispielgraphen soll `5` zurückgegeben werden. Verwenden Sie dazu **ausschließlich Rekursion**, also keine Schleifenkonstrukte. Sie dürfen aber zusätzliche (rekursive) Methoden schreiben. Sie dürfen davon ausgehen, dass das Feld `edges` des `Graph`-Objekts und das Feld `edge` eines `EdgeList`-Objekts nie `null` ist. Gehen Sie auch davon aus, dass keine Kante mehrfach in `edges` vorkommt. Geben Sie zu jeder Methode an, in welcher Klasse Sie sie implementieren. Kennzeichnen Sie die Methoden mit dem Schlüsselwort `static`, falls dies angebracht ist.

b) Schreiben Sie eine Methode `hatPfad`, die für ein `Graph`-Objekt und zwei (als `int`s angegebene) Knoten `s` und `e` genau dann `true` zurückgibt, wenn es im Graphen einen Pfad von `s` nach `e` gibt. Ein Pfad von `s` nach `e` ist eine Folge von n Kanten $s_1 \to e_1, \ldots, s_n \to e_n$, sodass $\texttt{s} = s_1$, $e_i = s_{i+1}$ und $e_n = \texttt{e}$ ist. Wir wollen auch leere Pfade zulassen, das heißt, dass ebenfalls `true` zurückgegeben werden soll, falls `s` und `e` gleich sind.

Im Beispielgraphen soll also für `s = 0` und `e = 2` `true` zurückgegeben werden, weil es den Pfad $0 \to 1, 1 \to 2$ gibt. Auch für 3 und 3 soll `true` zurückgegeben werden. Für 3 und 2 hingegen soll `false` zurückgegeben werden, da es keine von 3 ausgehende Kante gibt.

Verwenden Sie dazu **ausschließlich Rekursion**, also keine Schleifenkonstrukte. Sie dürfen aber (rekursive) Hilfsmethoden schreiben. Sie dürfen davon ausgehen, dass das Feld `edges` des `Graph`-Objekts und das Feld `edge` eines `EdgeList`-Objekts nie `null` ist. Geben Sie zu jeder Methode an, in welcher Klasse Sie sie implementieren. Kennzeichnen Sie die Methoden mit dem Schlüsselwort `static`, falls dies angebracht ist.

Hinweise:

- Unterscheiden Sie bei der Betrachtung einer Kantenliste $s_1 \to e_1, \ldots, s_n \to e_n$ der Länge n fünf Fälle:
  1) `s` und `e` sind gleich.
  2) $n = 0$, das heißt, es gibt keine Kanten mehr.
  3) $\texttt{s} = s_1$ und $e_1 = \texttt{e}$, das heißt die erste Kante der Liste verbindet `s` und `e` direkt.
  4) Es gibt einen Pfad zwischen `s` und `e` in der Restliste $s_2 \to e_2, \ldots, s_n \to e_n$, der also die Kante $s_1 \to e_1$ nicht verwendet.
  5) Es gibt jeweils einen Pfad von `s` zu s_1 und von e_1 zu `e` in der Restliste $s_2 \to e_2, \ldots, s_n \to e_n$. Dann gibt es natürlich auch einen Pfad von `s` und `e` in Kantenliste $s_1 \to e_1, \ldots, s_n \to e_n$.

c) Wir wollen nun `EdgeList` zu `List<E>` verallgemeinern. Dabei ist `E` Typparameter der Klasse, der den Typen der enthaltenen Objekte festlegt. So soll dann `List<Edge>` Elemente vom Typ `Edge` speichern.

i) Implementieren Sie die Klasse `List<E>` mit einem Konstruktor, der beide Felder setzt. Orientieren Sie sich dabei an `EdgeList`.

ii) Implementieren Sie dann in der Klasse `List<E>` eine Methode `contains`, die ein Element `e` vom Typ `E` übergeben bekommt und genau dann `true` zurückgibt, wenn die Liste ein Element enthält, das gleich zu `e` ist. Verwenden Sie dazu die für jeden Typen vordefinierte Methode `equals(Object o)`, die für `o1.equals(o2)` genau dann `true` zurückgibt, wenn die beiden Objekte `o1` und `o2` gleich sind.

Wird `contains` mit `null` aufgerufen, soll genau dann `true` zurückgegeben werden, wenn die Liste ein Element hat, dessen Wert ebenfalls `null` ist.

Verwenden Sie dazu **keine Rekursion**, sondern nur Schleifen. Sie dürfen aber (iterative) Hilfsmethoden schreiben. Geben Sie zu jeder Methode an, in welcher Klasse Sie sie implementieren. Kennzeichnen Sie die Methoden mit dem Schlüsselwort `static`, falls dies angebracht ist.

Lösung (Aufgabe 4):

a) Statische Version:

```
public class GraphSolution {
  public static int anzahlKanten(Graph g) {
    return anzahlKanten(g.edges);
  }

  public static int anzahlKanten(EdgeList l) {
    if (l == null) {
      return 0;
    } else {
      return 1 + anzahlKanten(l.next);
    }
  }
}
```

Non-static Version:

```
public class Graph {
  public int anzahlKanten() {
    return this.edges.length();
  }
}

public class EdgeList {
  public int length() {
    if (this.next == null) {
      return 1;
    } else {
      return 1 + this.next.length();
    }
  }
}
```

b) Statische Version:

```
public class GraphSolution {
  public static boolean hatPfad(Graph g, int start, int end) {
    return hatPfad(g.edges, start, end);
  }

  public static boolean hatPfad(EdgeList l, int s, int e) {
    if (s == e) {
      return true;
    }
    if (l == null) {
      return false;
    }
    Edge edge = l.edge;
    if (edge.start == s && edge.end == e) {
      return true;
    }
    return hatPfad(l.next, s, e)
      || (hatPfad(l.next, s, edge.start) && hatPfad(l.next, edge.end, e));
  }
}
```

Non-static Version:

```
public class Graph {
  public boolean hatPfad(int e, int s) {
    return this.edges.hatPfad(e, s);
  }
}

public class EdgeList {
  public boolean hatPfad(int e, int s) {
    if (s == e) {
      return true;
    }
    Edge edge = this.edge;
    if (edge.start == s && edge.end == e) {
      return true;
    }
    if (this.next == null) {
      return false;
    }
    return this.next.hatPfad(s, e)
      || (this.next.hatPfad(s, edge.start) && this.next.hatPfad(edge.end, e));
  }
}
```

c)

```
public class List<E> {
  E ele;
  List<E> next;

  public List(E e, List<E> n) {
    this.ele = e;
    this.next = n;
  }

  //non-static version:
  public boolean contains(E e) {
    List<E> cur = this;
    while (cur != null) {
      if (cur.ele != null && cur.ele.equals(e)) {
        return true;
      } else if (cur.ele == null && e == null) {
        return true;
      }
      cur = cur.next;
    }
    return false;
  }

  //static version:
  public static <T> boolean contains(List<T> l, T e) {
    while (l != null) {
      if (l.ele != null && l.ele.equals(e)) {
        return true;
      } else if (l.ele == null && l.ele == null) {
```

```
            return true;
          }
          l = l.next;
        }
        return false;
      }
    }
```

Aufgabe 5 (Haskell): (4 + 1 + (2 + 1 + 5) + 6 = 19 Punkte)

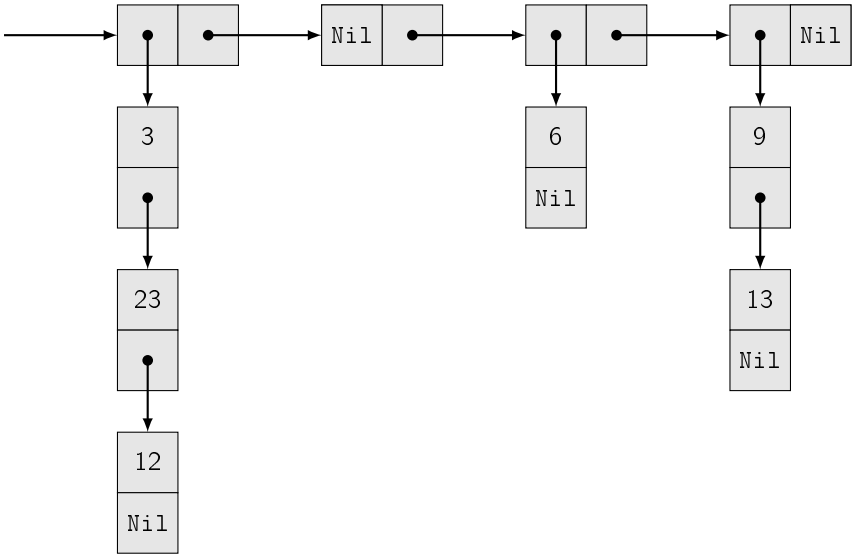
a) Geben Sie den allgemeinsten Typ der Funktionen `f` und `g` an, die wie folgt definiert sind. Gehen Sie davon aus, dass `1` den Typ `Int` hat.

```
f x y = \z -> ((z 1 y) ++ x)

g x y = g y t where t = y + 1
```

b) Gegeben sei das folgende Programm in Haskell:
`h x y = foldr (\u v -> x + y) (x + y)`
Geben Sie das Ergebnis für den Aufruf `h 1 2 [3, 4, 5]` an:

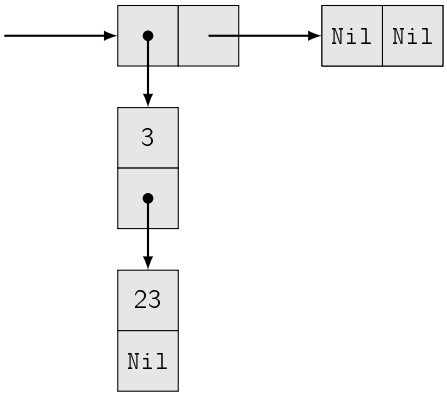
c) Wir wollen Listen von Listen von `Int`-Werten betrachten. Betrachten Sie dazu das folgende Bild:



Dargestellt ist eine Liste mit vier Elementen, welche jeweils eine Liste von `Int`s enthalten. Hierbei steht `Nil` für das Ende einer Liste. Die erste Teilliste enthält die drei Zahlen 3, 23 und 12. Die zweite Teilliste ist leer, die dritte Teilliste enthält eine 6 und die vierte Teilliste eine 9 und dann eine 13.

i) Geben Sie die Definition eines parametrischen Datentyps `List a` an, der sowohl für die Darstellung der Listen von `Int`s (als `List Int`) und für die Darstellung von Listen von Listen von `Int`s (als `List (List Int)`) geeignet ist. Verwenden Sie dazu jeweils einen eigenen Konstruktor für leere Listen und einen für Elemente mit einem Wert und einem Nachfolger. Verwenden Sie **nicht** die in Haskell vordefinierten Listen.

ii) Geben Sie für die folgende Liste einen `List`-Ausdruck an, der die Liste in Ihrer Datenstruktur kodiert:

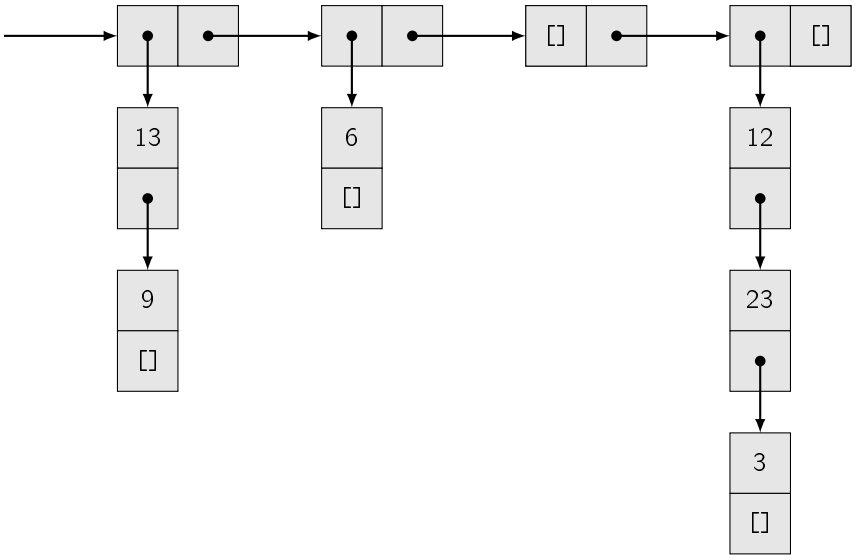


iii) Implementieren Sie eine Funktion `getMax :: List Int -> Int`, die aus einer Liste von `Int`s das Maximum bestimmt. Ihre Funktion darf sich auf der leeren Liste beliebig verhalten. Sie dürfen beliebige Hilfsfunktionen schreiben.

Hinweise:

- Beachten Sie, dass eine Liste auch nur negative Werte enthalten kann.

d) Wir wollen nun wieder vordefinierte Listen aus Haskell verwenden. Dort würde zum Beispiel der Baum aus c)ii) als `[[3, 23], []]` dargestellt werden. Implementieren Sie eine Funktion `revRevList :: [[a]] -> [[a]]`, die sowohl die Unterlisten als auch die Liste von Listen umdreht. Sie dürfen beliebige Hilfsfunktionen schreiben und beliebige vordefinierte Funktionen verwenden. Im Beispiel vom Anfang der Aufgabe c) würde die folgende Liste zurückgegeben werden:



(Als Haskell-Ausdruck: `[[13,9],[6],[],[12,23,3]]`)

Es soll also `revRevList [[3,23,12],[],[6],[9,13] = [[13,9],[6],[],[12,23,3]]]` gelten.

Hinweise:

- Implementieren Sie `revRevList`, indem Sie eine Funktion zum einfachen Umdrehen einer Liste mehrfach verwenden.

Lösung (Aufgabe 5):

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

g :: Int -> Int -> a
g x y = g y t where t = y + 1

-- b)
h x y = foldr (\u v -> x + y) (x + y)
-- h 1 2 [3, 4, 5] = foldr (\u v -> 1 + 2) (1 + 2) [3, 4, 5] = 3
```

```
-- c) i)
data List a = Nil | Cons a (List a) deriving Show

-- c) ii)
testList :: List (List Int)
testList = Cons (Cons 3 (Cons 23 Nil)) (Cons Nil Nil)
fullTestList = Cons (Cons 3 (Cons 23 (Cons 12 Nil)))
                (Cons Nil
                 (Cons (Cons 6 Nil)
                  (Cons (Cons 9 (Cons 13 Nil)) Nil)))

-- c) iii)
getMax :: List Int -> Int
getMax (Cons x Nil) = x
getMax (Cons x xs)  = if x > restMax then x else restMax
  where restMax = getMax xs

-- d)
-- higher order, Prelude variant
revRevList :: [[a]] -> [[a]]
revRevList = reverse . map reverse

-- plain variant:
revRevList' :: [[a]] -> [[a]]
revRevList' xs = revList [] (revSubLists xs)
  where revList a    []   = a
        revList a (y:ys) = revList (y:a) ys
        revSubLists    []   = []
        revSubLists (l:ls) = (revList [] l):(revSubLists ls)
```

Aufgabe 6 (Prolog): (2 + 4 + (2 + 1 + 3 + 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.

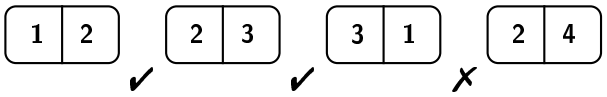
`f(c(a,X),Y,b)` und `f(Y,c(a,Z),Z)`

`h(i(s(X)),i(Y),Y)` und `h(Z,Z,i(X))`

b) Erstellen Sie für das folgende Logikprogramm zur Anfrage `?- q(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 „↯“.

```
q(a,b).
q(a,Y) :- q(Y,b), f(s(Y)).
f(s(X)) :- f(X).
f(a).
```

c) In dieser Aufgabe geht es darum, ein Prolog Programm zu schreiben, das eine Menge von Dominosteinen passend aneinander legen kann. Ein Dominostein hat zwei Seiten, auf denen jeweils eine Zahl steht. Eine Reihe von Dominosteinen — die sich jeweils mit der kurzen Seite berühren — heißt *korrekt*, falls die berührenden Flächen zweier Steine jeweils die gleiche Zahl zeigen.



In diesem Beispiel sind die ersten zwei Übergänge richtig, der vierte Stein passt allerdings nicht zum dritten.

Ein Dominostein lässt sich in Prolog als zweistelliges Symbol `stein` darstellen. Die vier Steine aus dem Beispiel hießen damit `stein(1,2)`, `stein(2,3)`, `stein(3,1)` und `stein(2,4)`.

i) Ein Dominostein kann gedreht werden. Programmieren Sie ein zweistelliges Prädikat `alleRichtungen`, welches zwei Steine als Argumente übergeben bekommt und genau dann erfüllt ist, falls der erste und der zweite Stein exakt gleich, oder gedrehte Varianten voneinander sind.

Beispielanfragen:

```
?- alleRichtungen(stein(2, 3), X).
X = stein(3, 2) ;
X = stein(2, 3).
```

ii) Programmieren Sie ein zweistelliges Prädikat `aneinanderLegbar`, welches zwei Steine als Argumente übergeben bekommt und genau dann erfüllt ist, falls der erste und der zweite — in der gegeben Orientierung — eine *korrekte Reihe* bilden.

Beispielanfragen:

```
?- aneinanderLegbar(stein(1, 2), stein(2, 3)).
true.

?- aneinanderLegbar(stein(1, 2), stein(3, 1)).
false.
```

iii) Programmieren Sie ein dreistelliges Prädikat `steinEntnehmen`, welches eine (nicht leere) Liste von Steinen als erstes Argument übergeben bekommt und einen Stein als zweites sowie eine Liste von Steinen als drittes Argument zurückgibt. Eine Anfrage `steinEntnehmen(`*`eingabe`*`, `*`stein`*`, `*`ausgabe`*`)` soll genau dann erfüllt sein, falls die Liste *eingabe* der Liste *ausgabe* entspricht, bei der ein beliebiger Stein *stein* entnommen wurde.

Beispielanfragen:

```
?- steinEntnehmen([stein(1, 2), stein(2, 3)], X, [stein(1, 2)]).
X = stein(2, 3) ;
false.

?- steinEntnehmen([stein(1, 2),stein(2, 3)], X, L).
X = stein(1, 2),
L = [stein(2, 3)] ;
X = stein(2, 3),
L = [stein(1, 2)] ;
false.
```

iv) Programmieren Sie ein zweistelliges Prädikat `korrekteReihe`, welches zwei Listen von Steinen als Argumente übergeben bekommt und genau dann erfüllt ist, falls beide die gleichen Steine enthalten (von denen möglicherweise einige gedreht wurden) und die zweite Liste eine *korrekte Reihe* darstellt.

```
?- korrekteReihe([stein(1, 2), stein(1, 3),stein(3, 2)], L).
L = [stein(2, 1), stein(1, 3), stein(3, 2)] ;
L = [stein(2, 3), stein(3, 1), stein(1, 2)] ;
L = [stein(3, 2), stein(2, 1), stein(1, 3)] ;
false.

?- korrekteReihe([stein(1, 2), stein(3, 4)], L).
false.
```

Hinweise:

- Sie dürfen die zu programmierenden Funktionen der ersten drei Aufgabenteile verwenden.
- Eine Möglichkeit, die Funktion für eine Eingabeliste mit mehr als einem Stein zu programmieren, ist die folgende:
  - Einen Stein aus der Eingabeliste entnehmen.
  - Den Rest der Liste in eine korrekte Reihenfolge bringen.
  - Testen, ob der Stein in einer der beiden Richtungen korrekt vor die Liste passt.

Lösung (Aufgabe 6):

a) (i) $\sigma_1 = \{Y = c(a, X)\}$

$\sigma_2 = \{X = Z\}$

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

$\sigma = \{X/b, Y/c(a, b), Z/b\}$

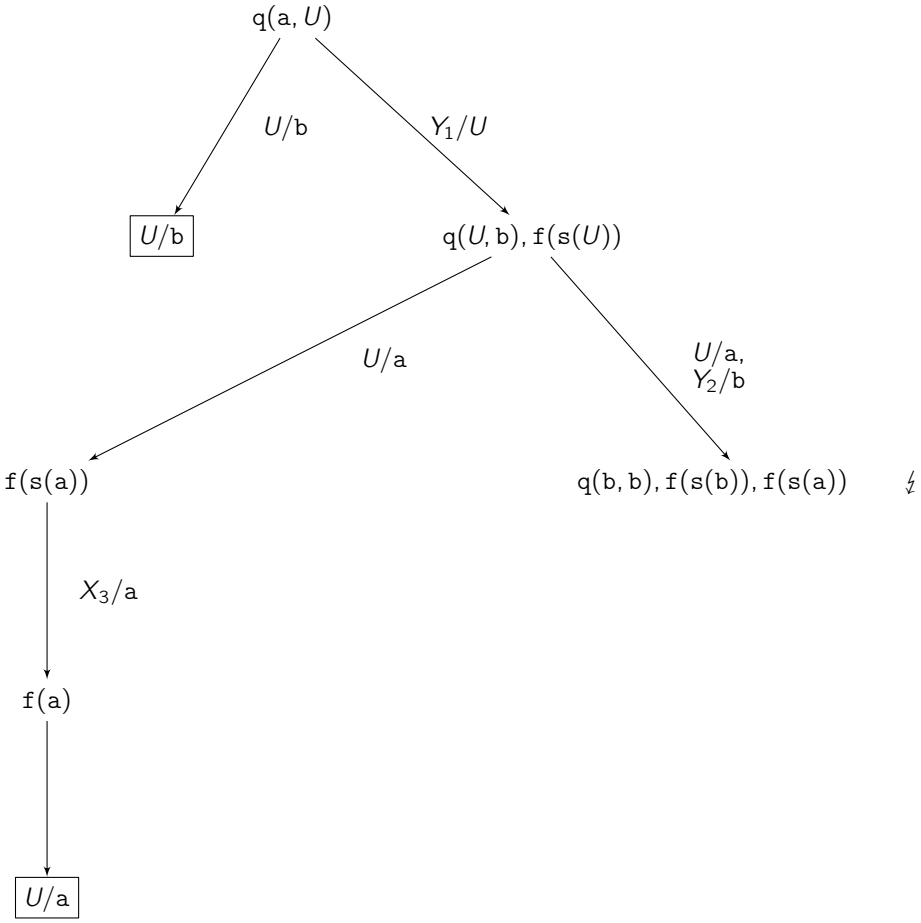
(ii) $\sigma_1 = \{Z = i(s(X))\}$

$\sigma_2 = \{Y = s(X)\}$

σ_3 existiert nicht, da $mgu(i(X), s(X))$ einen *clash failure* zurück gibt.

b) Die Lösung sind die Substitutionen:

- U = a
- U = b



c)

```
%  i)

alleRichtungen(stein(X,Y),stein(Y,X)).
alleRichtungen(stein(X,Y),stein(X,Y)).


%  ii)

aneinanderLegbar(stein(_,X),stein(X,_)).


%  iii)

steinEntnehmen([X|XS], X, XS).
steinEntnehmen([X|XS], Y, [X|YS]) :- steinEntnehmen(XS, Y, YS).


%  iv)

korrekteReihe([],[]).
```

```
korrekteReihe([X],[X]).
korrekteReihe(XS,[SR,Y|YS]) :-
    steinEntnehmen(XS, S, Rest),
    alleRichtungen(S, SR),
    aneinanderLegbar(SR, Y),
    korrekteReihe(Rest, [Y|YS]).
```