Aufgabe 1 (Programmanalyse): (10 + 6 = 16 Punkte)

a) Geben Sie die Ausgabe des Programms für den Aufruf `java M` an. Tragen Sie hierzu jeweils die ausgegebenen Zeichen in die markierten Stellen hinter „`OUT:`“ ein.

```
public class A {
    int x = 101;

    public A(boolean p) {
        this.x = 102;
    }

    public A(int p) {
        this.x = 103;
    }

    public A(String p) {
        this.x = 104;
    }

    public int getY() {
        return 501;
    }

    public int f(double p) {
        return 601;
    }
}
```

```
public class B extends A {
    int x = 201;

    public B() {
        super("15");
        this.x = 202;
    }

    public B(int p) {
        super(true);
        this.x = 203;
    }

    public int getY() {
        return 502;
    }

    public int f(int p) {
        return 602;
    }
}
```

```
public class M {
    public static void main(String[] args) {
        A aa = new A(1);
        System.out.println(aa.x);            // OUT: [     ]
        System.out.println(aa.getY());       // OUT: [     ]

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

        A ab = new B(Integer.valueOf(2));
        System.out.println(ab.x);            // OUT: [     ]
        System.out.println(((B) ab).x);      // OUT: [     ]
        System.out.println(ab.getY());       // OUT: [     ]

        System.out.println(aa.f(3));         // OUT: [     ]
        System.out.println(bb.f(3));         // OUT: [     ]
        System.out.println(ab.f(3));         // OUT: [     ]
    }
}
```

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

```
1  public class C extends A {
2      String x = "";
3
4      public long getY() {
5          return 503;
6      }
7
8      public String getX() {
9          return C.x;
10     }
11 }
```

Lösung:

a)
```
public class M {
    public static void main(String[] args) {
        A aa = new A(1);
        System.out.println(aa.x);            // OUT: [ 103 ]
        System.out.println(aa.getY());       // OUT: [ 501 ]

        B bb = new B();
        System.out.println(((A) bb).x);      // OUT: [ 104 ]
        System.out.println(bb.x);            // OUT: [ 202 ]
        System.out.println(bb.getY());       // OUT: [ 502 ]

        A ab = new B(Integer.valueOf(2));
        System.out.println(ab.x);            // OUT: [ 102 ]
        System.out.println(((B) ab).x);      // OUT: [ 203 ]
        System.out.println(ab.getY());       // OUT: [ 502 ]

        System.out.println(aa.f(3));         // OUT: [ 601 ]
        System.out.println(bb.f(3));         // OUT: [ 602 ]
        System.out.println(ab.f(3));         // OUT: [ 601 ]
    }
}
```

b)
- In der Klasse `A` existiert kein Konstruktor mit leerer Parameterliste. Somit muss ein Konstruktor deklariert werden, welcher als erste Anweisung explizit einen der Konstruktoren von `A` aufruft.
- Der Rückgabetyp der Methode `getY` wurde unzulässigerweise von `int` auf `long` erweitert (Zeile 4).
- Die nicht-statische Variable `x` wird wie eine statische Variable genutzt (Zeile 9).

Aufgabe 2 (Hoare-Kalkül): (12 + 3 = 15 Punkte)

Gegeben sei folgendes Java-Programm P über den Variablen `a`, `i` und `res`, welches die Summe alle ungeraden Zahlen `i` berechnet, die zwischen `0` und `a` liegen d.h., für die `0<i<a` gilt.

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

```
i = 0;
res = 0;
while (i < a) {
    if (i % 2 == 1) {
      res = res + i;
    }
    i = i + 1;
}
```

$\langle\, \texttt{res} = \lfloor \frac{\texttt{a}}{2} \rfloor^2 \,\rangle$ (Nachbedingung)

Hinweise:

- $\lfloor \frac{\texttt{a}}{2} \rfloor$ ist das abgerundete Ergebnis der Division von `a` durch 2. So ist z.B. $\lfloor \frac{9}{2} \rfloor = \lfloor \frac{8}{2} \rfloor = 4$.
- `%` berechnet den Rest bei der Ganzzahldivision. So hat `9 % 2` das Ergebnis `1` und `8 % 2` das Ergebnis `0`.
- Sie dürfen verwenden, dass für ungerade natürliche Zahlen `i` Folgendes gilt: $\lfloor \frac{\texttt{i}+1}{2} \rfloor^2 = \lfloor \frac{\texttt{i}}{2} \rfloor^2 + \texttt{i}$.
- Sie dürfen in beiden Teilaufgaben beliebig viele Zusicherungs-Zeilen ergänzen oder streichen. In der Musterlösung werden allerdings genau die angegebenen Zusicherungen benutzt.
- Bedenken Sie, dass die Regeln des Kalküls syntaktisch sind, weshalb Sie semantische Änderungen (beispielsweise von $x+1 = y+1$ zu $x = y$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen. Klammern dürfen und müssen Sie jedoch eventuell bei der Anwendung der Zuweisungsregel setzen.
- Wenn Sie die Konsequenzregel anwenden, müssen Sie nicht beweisen, warum aus der oberen Zusicherung die untere Zusicherung folgt.
- Bei der Anwendung der Bedingungsregel 1 müssen sie **nicht** zeigen, warum aus der Bedingung vor dem `if`-Block und der negierten `if`-Bedingung die Zusicherung nach dem `if`-Block folgt.

a) Vervollständigen Sie 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 eine Zusicherung stehen, wenn dies aus einer Regel des Hoare-Kalküls folgt.

b) Beweisen Sie die Terminierung des Algorithmus P mit Hilfe des Hoare-Kalküls.

Lösung:

a)

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

```
i = 0
```

$\langle \mathtt{a} \geq 0 \wedge \mathtt{i} = 0 \wedge 0 = 0 \rangle$

```
res = 0
```

$\langle \mathtt{a} \geq 0 \wedge \mathtt{i} = 0 \wedge \mathtt{res} = 0 \rangle$
$\langle \mathtt{i} \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \frac{\mathtt{i}}{2} \rfloor^2 \rangle$

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

$\langle \mathtt{i} \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \frac{\mathtt{i}}{2} \rfloor^2 \wedge \mathtt{i} < \mathtt{a} \rangle$

```
    if (i % 2 == 1) {
```

$\langle \mathtt{i} \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \frac{\mathtt{i}}{2} \rfloor^2 \wedge \mathtt{i} < \mathtt{a} \wedge \mathtt{i} \mod 2 = 1 \rangle$
$\langle \mathtt{i} + 1 \leq \mathtt{a} \wedge \mathtt{res} + \mathtt{i} = \lfloor \frac{\mathtt{i}+1}{2} \rfloor^2 \rangle$

```
        res = res + i;
```

$\langle \mathtt{i} + 1 \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \frac{\mathtt{i}+1}{2} \rfloor^2 \rangle$

```
    }
```

$\langle \mathtt{i} + 1 \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \frac{\mathtt{i}+1}{2} \rfloor^2 \rangle$

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

$\langle \mathtt{i} \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \frac{\mathtt{i}}{2} \rfloor^2 \rangle$

```
}
```

$\langle \mathtt{i} \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \frac{\mathtt{i}}{2} \rfloor^2 \wedge \neg(\mathtt{i} < \mathtt{a}) \rangle$
$\langle \mathtt{res} = \lfloor \frac{\mathtt{a}}{2} \rfloor^2 \rangle$

Wir müssen für die Bedingungsregel 1 noch zeigen, dass aus der Zusicherung vor dem `if`-Block zusammen mit der negierten `if`-Bedingung die Zusicherung hinter dem `if`-Block folgt. In der Aufgabenstellung war dieser Beweis **nicht** gefordert. Die negierte `if`-Bedingung ist $\neg(\mathtt{i} \mod 2 = 1)$, was äquivalent zu $\mathtt{i} \mod 2 = 0$ ist. Also ist `i` in diesem Fall eine gerade Zahl. Für eine gerade Zahl ist aber $\lfloor \frac{\mathtt{i}+1}{2} \rfloor = \frac{\mathtt{i}}{2} = \lfloor \frac{\mathtt{i}}{2} \rfloor$. Also gilt insbesondere:

$$\begin{aligned} & \mathtt{i} \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \tfrac{\mathtt{i}}{2} \rfloor^2 \wedge \mathtt{i} < \mathtt{a} \wedge \neg(\mathtt{i} \mod 2 = 1) \\ \Rightarrow & \mathtt{i} + 1 \leq \mathtt{a} \wedge \mathtt{res} = \lfloor \tfrac{\mathtt{i}+1}{2} \rfloor^2 \end{aligned}$$

b) Eine gültige Variante für die Terminierung ist $V = \mathtt{a} - \mathtt{i}$. Die Schleifenbedingung $\mathtt{i} < \mathtt{a}$ impliziert natürlich $V \geq 0$. Es gilt:

$\langle \mathtt{a} - \mathtt{i} = \mathtt{m} \wedge \mathtt{i} < \mathtt{a} \rangle$

```
    if (i % 2 == 1) {
```

$\langle \mathtt{a} - \mathtt{i} = \mathtt{m} \wedge \mathtt{i} < \mathtt{a} \wedge \mathtt{i} \mod 2 = 1 \rangle$
$\langle \mathtt{a} - (\mathtt{i} + 1) < \mathtt{m} \rangle$

```
        res = res + i;
```

$\langle \mathtt{a} - (\mathtt{i} + 1) < \mathtt{m} \rangle$

```
    }
```

$\langle \mathtt{a} - (\mathtt{i} + 1) < \mathtt{m} \rangle$

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

$\langle \mathtt{a} - \mathtt{i} < \mathtt{m} \rangle$

Damit ist die Terminierung der einzigen Schleife in P gezeigt.

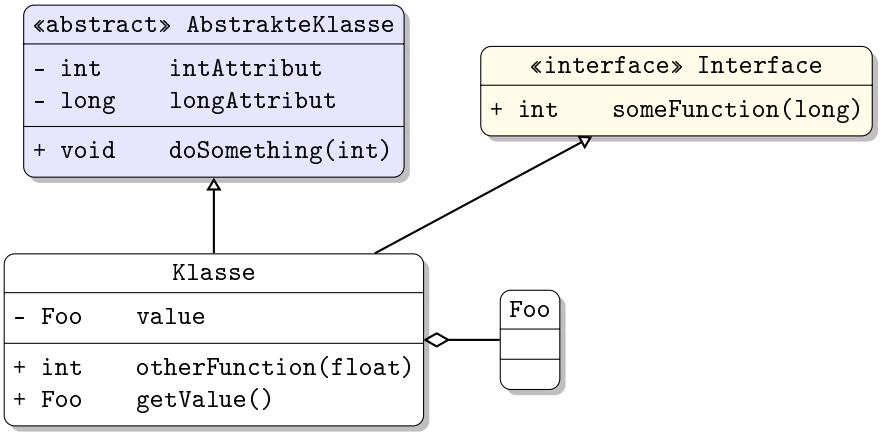
Aufgabe 3 (Klassenhierarchie): (11 + 8 = 19 Punkte)

Ziel dieser Aufgabe ist die Erstellung einer Klassenhierarchie zur Verwaltung von öffentlichen Verkehrsmitteln (kurz *Öffis*).

- Jedes öffentliche Verkehrsmittel ist entweder ein Flugzeug, ein Zug oder ein Bus. Andere öffentliche Verkehrsmittel gibt es bei uns nicht. Für jedes öffentliche Verkehrsmittel ist die maximale Anzahl an Passagieren eine wichtige Eigenschaft.
- Da Flugzeuge häufig für Fernreisen genutzt werden, ist bei Flugzeugen die Beinfreiheit in Zentimetern von besonderem Interesse.
- Ein Zug hat beliebig viele Waggons. Jeder Waggon verfügt über eine Methode `void reinigen()` zum Reinigen des Waggons.
- Speisewagen sind spezielle Waggons. Sie verfügen über eine Methode `void auffuellen()`, um die Bestände des Speisewagens aufzufüllen.
- Reisewaggons sind spezielle Waggons, bei denen die maximale Anzahl an Passagieren eine wichtige Eigenschaft ist.
- Manche öffentliche Verkehrsmittel sind akkubetrieben. Sie verfügen über die Methoden `void aufladen()` und `int getLadezustand()`. Letztere gibt den Ladezustand des Akkus in Prozent zurück.
- Elektrobusse und Hybridbusse sind spezielle Busse, die akkubetrieben sind.

a) Entwerfen Sie unter Berücksichtigung der Prinzipien der Datenkapselung eine geeignete Klassenhierarchie für die oben aufgelisteten Sachverhalte. Notieren Sie keine Konstruktoren oder Selektoren. Sie müssen nicht markieren, ob Attribute `final` sein sollen. Achten Sie darauf, dass gemeinsame Merkmale in Oberklassen bzw. Interfaces zusammengefasst werden und markieren Sie alle Klassen als abstrakt, bei denen dies sinnvoll ist.

Verwenden Sie hierbei die folgende Notation:



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

b) Schreiben Sie eine Java-Methode `public static void aufbereiten(Oeffi[] oeffis)`. Für jeden in `oeffis` enthaltenen Zug soll jeder Waggon gereinigt werden. Außerdem soll jeder Speisewagen aufgefüllt werden, nachdem er gereinigt worden ist.

Jedes in `oeffis` enthaltene akkubetriebene öffentliche Verkehrsmittel soll aufgeladen werden, falls sein Ladezustand niedriger als 50% ist. Bedenken Sie, dass es neben den akkubetriebenen öffentlichen Verkehrsmitteln aus Aufgabenteil **a)** noch weitere akkubetriebene öffentliche Verkehrsmittel geben könnte.

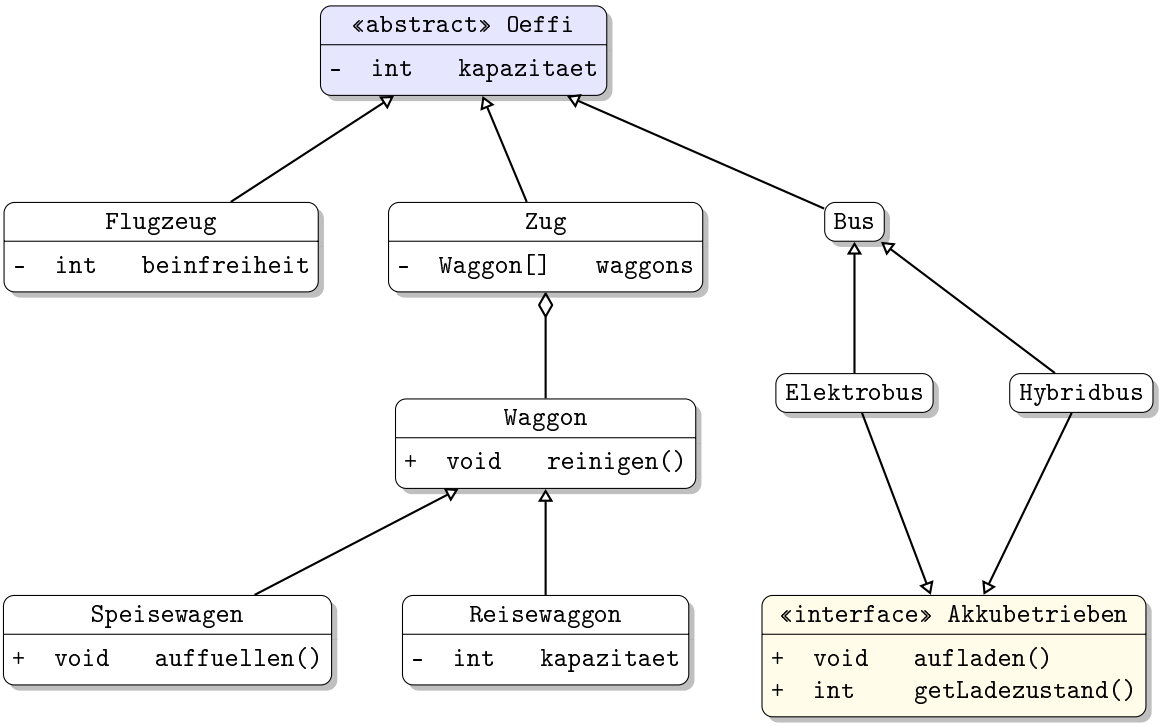
Gehen Sie davon aus, dass es zu jedem Attribut geeignete Selektoren gibt.

Hinweise:

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

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:



b)

```
public static void aufbereiten(Oeffi[] oeffis) {
    for (var o: oeffis) {
        if (o instanceof Zug) {
            for (var w: ((Zug) o).getWaggons()) {
                w.reinigen();
                if (w instanceof Speisewagen) {
                    ((Speisewagen) w).auffuellen();
                }
            }
        }
        if (o instanceof Akkubetrieben && ((Akkubetrieben) o).getLadezustand() < 50) {
            ((Akkubetrieben) o).aufladen();
        }
    }
}
```

Aufgabe 4 (Programmierung in Java): (9 + 3 + 8 + 8 + 4 = 32 Punkte)

In dieser Aufgabe wollen wir eine zweidimensionale Ebene in Java modellieren. Dazu teilen wir die Ebene in Quadrate auf, so wie die Planquadrate auf einer Landkarte. Die auf der Ebene dargestellte Information wird auf die einzelnen Quadrate verteilt. Wir wollen besonders darauf achten, zu einem gegebenen Quadrat direkt angrenzende Quadrate schnell zu erreichen. Um es nicht allzu kompliziert zu machen, beschränken wir uns jedoch darauf, nur das auf der Ebene direkt darunter liegende Quadrat sowie das auf der Ebene direkt rechts davon liegende Quadrat schnell erreichen zu wollen. Das Quadrat links davon und das darüber liegende Quadrat werden also in dieser Aufgabe ignoriert.

Um dieses Ziel zu erreichen, modellieren wir jedes Quadrat als einen `Node`, welcher eine direkte Referenz auf seinen rechten Nachbarn (`right`) und eine direkte Referenz auf seinen unteren Nachbarn (`bottom`) hat. Außerdem hat jeder `Node` einen `value`, welcher die zu diesem Quadrat gehörende Information enthält. Die Klasse `Node` soll Code enthalten, welcher lokal begrenzt auf diesem `Node` oder seinen direkten Nachbarn arbeitet.

Außerdem modellieren wir eine zweidimensionale Ebene durch ein Objekt der Klasse `Grid`, welches eine Referenz auf den `Node` hat, der das obere, linke Quadrat der Ebene repräsentiert.

```
public class Grid {
    private final Node topLeftNode;

    public Grid() {
        topLeftNode = new Node();
    }
}
```

```
public class Node {
    private Node right;
    private Node bottom;
    private int value;

    public Node getRight() {
        return right;
    }

    public Node getBottom() {
        return bottom;
    }

    public int getValue() {
        return value;
    }

    public void setRight(Node right) {
        this.right = right;
    }

    public void setBottom(Node bottom) {
        this.bottom = bottom;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```

In den folgenden Aufgaben dürfen Sie davon ausgehen, dass für jedes Objekt `o` der Klasse `Node` gilt, dass Sie niemals das Objekt `o` erneut besuchen, wenn Sie einer beliebigen Folge von `right`- und `bottom`-Referenzen folgen. Durch `right`- und `bottom`-Referenzen dürfen also **keine** Zyklen gebildet werden.

Es ist sichergestellt, dass bei Objekten der Klasse `Grid` das Attribut `topLeftNode` nicht `null` ist.

a) In dieser Aufgabe wollen wir am unteren Ende der Ebene eine Reihe von Quadraten hinzufügen. Dazu starten wir beim oberen, linken Quadrat, laufen gerade nach unten bis wir die aktuell letzte Reihe erreicht haben und fügen dort die neue Reihe ein.

Gehen Sie davon aus, dass die gegebene Ebene vorher die Form eines Rechtecks hat.

Wenn die Ebene vorher also drei Reihen hoch und fünf Spalten breit ist, so soll sie anschließend vier Reihen hoch und fünf Spalten breit sein.

Implementieren Sie dazu in der Klasse `Grid` die Methode `addRow`, welche keine Parameter erhält und nichts zurückgibt.

b) Implementieren Sie die Klasse `MalformedGridException` als `Exception`-Klasse, welche beim Aufrufen von `toString` den Text `"The Grid is not rectangular!"` zurückgibt.

c) Wir wollen sicherstellen, dass die Ebene tatsächlich die Form eines Rechtecks aufweist. Dazu prüfen wir zunächst für ein Quadrat, ob es zusammen mit seinen direkten Nachbarn diese Eigenschaft verletzt.

Erfüllt ein Quadrat `o` zusammen mit seinen direkten Nachbarn eine der folgenden Bedingungen, so ist es *wohlgeformt*:

- Das Quadrat `o` liegt an der unteren rechten Ecke der Ebene. Sowohl unter `o` als auch rechts von `o` ist also kein weiteres Quadrat.
- Das Quadrat `o` liegt an der rechten Kante der Ebene, aber nicht ganz unten. Es existiert also ein Quadrat `u` direkt unter `o`, jedoch keines rechts daneben. Auch existiert rechts neben dem Quadrat `u` kein weiteres Quadrat.
- Das Quadrat `o` liegt an der unteren Kante der Ebene, aber nicht ganz rechts. Es existiert also ein Quadrat `r` direkt rechts von `o`, jedoch keines darunter. Auch existiert unter dem Quadrat `r` kein weiteres Quadrat.
- Das Quadrat `o` liegt irgendwo auf der Ebene, aber nicht ganz unten und nicht ganz rechts. Es existiert also sowohl ein Quadrat `r` direkt rechts von `o` als auch ein Quadrat `u` direkt unter `o`. Außerdem liegt direkt rechts von `u` dasselbe Quadrat wie unter `r`.

Implementieren Sie in der Klasse `Node` die Methode `ensureWellFormedLocally`, welche überprüft, ob das durch dieses `Node`-Objekt repräsentierte Quadrat wohlgeformt ist. Ist dies nicht der Fall, so soll eine `MalformedGridException` geworfen werden.

d) Nun nutzen wir die in der vorherigen Aufgabe implementierte Methode `ensureWellFormedLocally`, um tatsächlich für die gesamte Ebene sicherzustellen, dass diese die Form eines Rechtecks hat. Dazu beginnen wir in der obersten Reihe, durchlaufen diese bis zum Ende und rufen dabei für jedes Quadrat die Methode `ensureWellFormedLocally` auf. Dies stellt sicher, dass das Quadrat zusammen mit seinen direkten Nachbarn wohlgeformt ist. Anschließend führen wir dieses Verfahren für alle weiteren Reihen aus. Die Ebene hat genau dann die Form eines Rechtecks, wenn alle Quadrate darin wohlgeformt sind.

Implementieren Sie in der Klasse `Grid` die Methode `ensureWellFormed`, welche überprüft, ob die durch dieses `Grid`-Objekt repräsentierte Ebene die Form eines Rechtecks aufweist. Ist dies nicht der Fall, so soll eine `MalformedGridException` geworfen werden.

e) Nun wollen wir uns mit der auf der Ebene dargestellten Information beschäftigen. Bisher ist das `value`-Attribut der `Node`-Objekte ein `int`-Wert, welcher lediglich durch den Compiler auf den Standardwert `0` initialisiert wurde. Ein `int`-Wert ist uns jedoch zu unflexibel. Deswegen wollen wir den Typ von `value` auf den Typparameter `T` ändern.

Fügen Sie daher zur Klasse `Node` den Typparameter `T` hinzu. Ändern Sie den Typ von `value` auf `T`. Falls nötig, passen Sie ebenfalls die Klasse `Grid` an.

Es ist nicht notwendig, die vorgegebenen Getter und Setter oder Ihren Code aus den vorherigen Aufgaben anzupassen.

Lösung:

a) In der Klasse `Grid`

```
public void addRow() {
    Node current = topLeftNode;
    while (current.getBottom() != null) {
        current = current.getBottom();
    }
    current.setBottom(new Node());
    while (current.getRight() != null) {
        Node newNode = new Node();
        current.getRight().setBottom(newNode);
        current.getBottom().setRight(newNode);
        current = current.getRight();
    }
}
```

b)
```
public class MalformedGridException extends Exception {
    @Override
    public String toString() {
        return "The Grid is not rectangular!";
    }
}
```

c) In der Klasse `Node`

```
public void ensureWellFormedLocally() throws MalformedGridException {
    if (right == null && bottom == null) {
        // well formed
    } else if (right == null && bottom != null && bottom.right == null) {
        // well formed
    } else if (right != null && bottom == null && right.bottom == null) {
        // well formed
    } else if (right != null && bottom != null && bottom.right != null
        && bottom.right == right.bottom) {
        // well formed
    } else {
        throw new MalformedGridException();
    }
}
```

d) In der Klasse `Grid`

```
public void ensureWellFormed() throws MalformedGridException {
    Node currentRowHead = topLeftNode;
    while (currentRowHead != null) {
        Node current = currentRowHead;
        while (current != null) {
            current.ensureWellFormedLocally();
            current = current.getRight();
        }
        currentRowHead = currentRowHead.getBottom();
    }
}
```

e) In der Klasse `Grid`

```
public class Grid<T> {
    private final Node<T> topLeftNode;
```

```
    public Grid() {
        this.topLeftNode = new Node<>();
    }
}
```

In der Klasse `Node`

```
public class Node<T> {
    private Node<T> right;
    private Node<T> bottom;
    private T value;
}
```

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

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

```
f [] (x:xs) _ = False
f (x:xs) (y:ys) c = if x > c then y else f xs ys c


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

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

```
i :: [Bool]
i = filter (\x-> (not x)) (map (> 0) [-4,-3,6,1,2])
```

Hinweise:

- Für die vordefinierte Funktion `not::Bool -> Bool` ist `not True = False` und `not False = True`.

```
j :: [Int]
j =  filter (<= 0)  ((map (* (-1))) [-3,-2,7,8,9])
```

c) In dieser Aufgabe betrachten wir sogenannte Operatorbäume, die durch die folgende Datenstruktur `OpTree` repräsentiert sind.

```
data OpTree a = Leaf a | Operator (a -> a -> a) (OpTree a) (OpTree a)
t1:: OpTree Int
t1 = Operator (-) (Operator (*) (Leaf 3) (Leaf 2)) (Leaf 5)
```

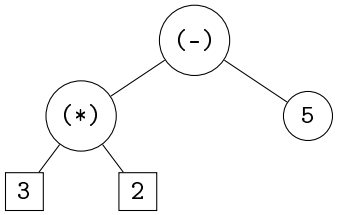


Abbildung 1: Operatorbaum `t1`

Der Operatorbaum `t1` ist grafisch in Abbildung 1 dargestellt. Kreise entsprechen hierbei dem Konstruktor `Operator`, Quadrate dem Konstruktor `Leaf`.

Schreiben Sie eine Funktion `flatten :: OpTree a -> a`, die einen gegebenen `OpTree` auswertet. So soll z.B. der Aufruf `flatten t1` den Wert $3 * 2 - 5 = 1$ haben.

Hinweise:

- Die Funktion `(*) :: Int -> Int -> Int` ist die Multiplikationsfunktion in "Präfix-Schreibweise". Haskell wertet also z.B. `(*) 3 2` zu 6 aus. Analog dazu ist `(-)` die Subtraktionsfunktion in "Präfix-Schreibweise".

d) Schreiben Sie eine Funktion `foldList :: a -> (a -> a -> a) -> [a] -> OpTree a`. Für einen Wert `default :: a`, eine Funktion `op :: a -> a -> a` und eine Liste $[e_1, e_2, e_3, ..., e_n]$`::[a]` soll der Aufruf `foldList default op` $[e_1, e_2, e_3, ..., e_n]$ einen Operatorbaum `t` liefern, sodass `flatten t` den Wert `default` hat, falls `n` $= 0$ ist (d.h., falls die Liste leer ist). Ansonsten soll der Aufruf `flatten t` den Wert `op` e_1 `(op` e_2 `(op` e_3 `(... (op` e_n `default)...)))` haben.

e) Schreiben Sie eine Funktion `listProd :: [Int] -> Int`, die alle Elemente in einer Liste miteinander multipliziert. Ist die Liste leer, so soll die Funktion den Wert `1` zurückgeben. Benutzen Sie hierzu **ausschließlich** die Funktionen `flatten` und `foldList` und die vordefinierte Funktion `(*)`. Für die Lösung dieser Aufgabe ist es unerheblich, ob Sie die beiden vorigen Teilaufgaben lösen konnten.

Lösung:

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

g :: a -> [[a]] -> [a] -> [a]
```

b)
```
i = [False, False]
j = [-7,-8,-9]
```

c)
```
flatten :: OpTree a -> a
flatten (Leaf x) = x
flatten (Operator op left right) = op (flatten left) (flatten right)
```

d)
```
foldList :: a -> (a -> a -> a) -> [a] -> OpTree a
foldList def _ [] = Leaf def
foldList def op (x:xs) = Operator op (Leaf x) (foldList def op xs)
```

e)
```
listProd :: [Int] -> Int
listProd xs = flatten (foldList 1 (*) xs)
```

Aufgabe 6 (Prolog): (2 + 6.5 + (1.5 + 3 + 1.5 + 3.5) = 18 Punkte)

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

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

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

b) Gegeben sei folgendes Prolog-Programm P.

```
p(s(Y)) :- p(q(Y)).
p(q(Y)) :- p(Y).
p(q(0)).
```

Erstellen Sie für das Programm P den Beweisbaum zur Anfrage "`?- p(s(X)).`" bis zur Höhe 4 (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(s(X)).`" an, die im Beweisbaum bis zur Höhe 4 enthalten sind. Welche der beiden Regeln muss ans Ende des Programms verschoben werden, damit Prolog bei obiger Anfrage mindestens eine Antwortsubstitution findet?

c) In dieser Aufgabe geht es darum, die Zulassungen für die Progra-Klausur mit einem Prolog-Programm zu verwalten. Die Zulassungsvoraussetzungen für einen Studierenden bestehen aus

- der prozentualen Punktzahl in den Übungen vor Weihnachten,
- der prozentualen Punktzahl in den Übungen nach Weihnachten,
- einem Boolean (`true` oder `false`), das angibt, ob Codescape bestanden wurde und
- einem Boolean, das angibt, ob die Präsenzübung bestanden wurde.

In Prolog stellen wir die Zulassungsvoraussetzungen für einen Studierenden beispielsweise durch den Term `zv('Hans Meier', 63, 70, true, true)` dar. Dies bedeutet, dass der Student mit Namen Hans Meier 63% in den Übungen vor Weihnachten und 70% in den Übungen nach Weihnachten erreicht hat. Außerdem hat er Codescape und die Präsenzübung bestanden.

Hinweise:

- Sie dürfen in allen Teilaufgaben Prädikate aus den vorherigen Teilaufgaben verwenden, auch wenn Sie diese Prädikate nicht implementiert haben.
- Sie dürfen Hilfsprädikate definieren.
- Strings, die in `' '` eingeschlossen sind, sind in Prolog Funktionssymbole (die ansonsten ja als Strings geschrieben werden, die mit Kleinbuchstaben beginnen).

i) Implementieren Sie ein Prädikat `erfuellt` mit Stelligkeit 1 in Prolog, das wahr ist, wenn das einzige Argument die Zulassungsvoraussetzungen für einen Studierenden in der oben beschriebenen Form darstellt und der Studierende die Zulassung erreicht hat. Dies ist der Fall, wenn in den Übungen vor und nach Weihnachten jeweils mindestens 50% der Punkte erzielt wurden und sowohl Codescape als auch die Präsenzübung bestanden wurden. Beispielsweise soll

```
erfuellt(zv('Hans Meier', 63, 70, true, true))
```

wahr sein. Die Anfragen

"`?- erfuellt(zv('Karl Schneider', 49, 61, true, true)).`" und
"`?- erfuellt(zv('Petra Mueller', 53, 63, false, true)).`"

sollen hingegen falsch sein.

ii) Implementieren Sie ein Prädikat `zugelassen` mit Stelligkeit 2 in Prolog, sodass für die Anfrage

```
?- zugelassen(studis, YS).
```

gilt: Wenn `studis` eine Liste von Zulassungsvoraussetzungen in der oben beschriebenen Form ist, dann findet Prolog mindestens eine Antwort und die erste Antwort für `YS` enthält genau jene Elemente aus `studis`, die die Zulassung erreicht haben. Beispielsweise soll obige Anfrage als erste Antwort

```
YS = [zv('Hans Meier', 63, 70, true, true)]
```

liefern, wenn `studis` ausschließlich die Zulassungsvoraussetzungen für die Studierenden "Hans Meier", "Karl Schneider" und "Petra Mueller" aus Aufgabenteil i) enthält.

iii) Implementieren Sie ein Prädikat `names` mit Stelligkeit 2, sodass für die Anfrage

```
?- names(studis, YS).
```

gilt: Wenn `studis` eine Liste von Zulassungsvoraussetzungen in der oben beschriebenen Form ist, dann findet Prolog mindestens eine Antwort und die erste Antwort für `YS` enthält die Namen aller Elemente aus `studis` in der gleichen Reihenfolge. Beispielsweise soll obige Anfrage als erste Antwort

```
YS = ['Hans Meier', 'Karl Schneider', 'Petra Mueller']
```

liefern, wenn `studis` ausschließlich die Zulassungsvoraussetzungen für die Studierenden "Hans Meier", "Karl Schneider" und "Petra Mueller" aus Aufgabenteil i) enthält.

iv) Leider erlaubt die gewählte Repräsentation uns nicht, die Zulassungsvoraussetzungen für Studierende mit gleichen Namen sinnvoll zu verwalten. Implementieren Sie ein Prädikat `duplicateNames` mit Stelligkeit 2, um daraus resultierende Probleme zu erkennen. Für die Anfrage

```
?- duplicateNames(studis, YS).
```

soll gelten: Wenn `studis` eine Liste von Zulassungsvoraussetzungen in der oben beschriebenen Form ist, dann findet Prolog mindestens eine Antwort und die erste Antwort für `YS` enthält jene Namen, die mehrfach in `studis` vorkommen. Beispielsweise soll Prolog für obige Anfrage als erste Antwort `YS = ['Hans Meier']` liefern, wenn `studis` die Form

```
[zv('Hans Meier', ...), zv('Petra Mueller', ...), zv('Hans Meier', ...)]
```

hat.

Hinweise:

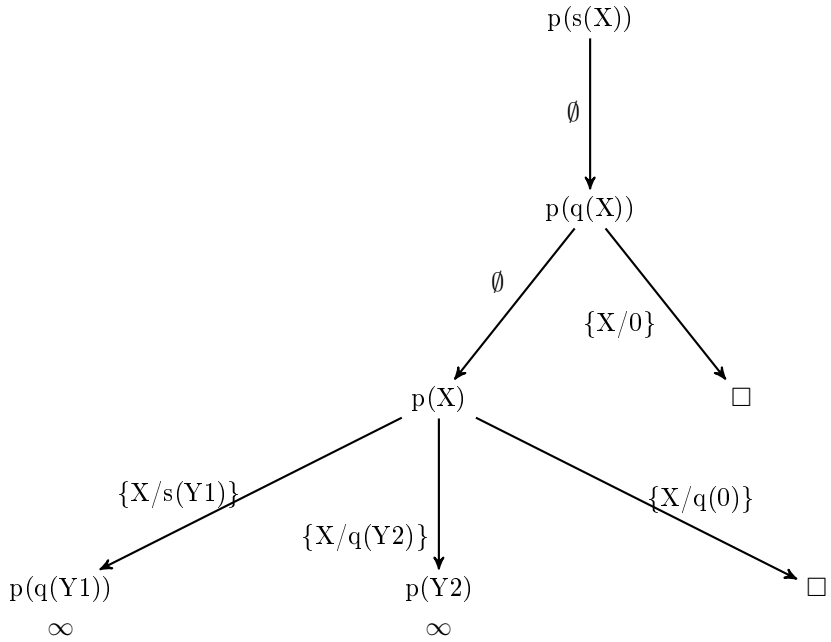
- Sie dürfen das vordefinierte Prädikat `member` mit Stelligkeit 2 verwenden. Dies kann genutzt werden, um zu testen, ob eine Liste ein Element enthält. Beispielsweise ist `member(1, [2,1])` wahr.
- Sie können davon ausgehen, dass in der Liste von Zulassungsvoraussetzungen jeder Name höchstens zwei mal auftritt.

Lösung:

a) i) $f(s(X), X, s(Y)), f(Y, s(Z), s(s(Z)))$: $\sigma_1 = Y/s(X), \sigma_2 = X/s(Z), mgu(s(s(s(Z))), s(s(Z)))$ existiert nicht, occur failure.

ii) $g(Z, s(s(Y))), g(p(X), s(Z))$: $\sigma_1 = Z/p(X), mgu(s(s(Y)), s(p(X)))$ existiert nicht, clash failure.

b)



Die Antwortsubstitutionen innerhalb des Beweisbaums sind $\{X/0\}, \{X/q(0)\}$.

Die zweite Regel muss ans Ende des Programms verschoben werden, damit Prolog mindestens eine Antwortsubstitution findet:

```
p(s(Y)) :- p(q(Y)).
p(q(0)).
p(q(Y)) :- p(Y).
```

c)
```
% i)
erfuellt(zv(_, X, Y, true, true)) :- X >= 50, Y >= 50.

% ii)
zugelassen([], []).
zugelassen([X|XS], [X|YS]) :- erfuellt(X), zugelassen(XS, YS).
zugelassen([X|XS], YS) :- zugelassen(XS, YS).

% iii)
names([], []).
names([zv(X, _, _, _, _)|XS], [X|YS]) :- names(XS, YS).

% iv)
duplicateNames([], []).
duplicateNames([zv(X, _, _, _, _)|XS], [X|YS]) :-
  names(XS, ZS),
  member(X, ZS),
  duplicateNames(XS, YS).
duplicateNames([_|XS], YS) :-
  duplicateNames(XS, YS).
```