## Aufgabe 1 (Programmanalyse): (15 Punkte)

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

```
public class A {

  double x = 2;

  static int y = 0;

  public A() {
    y += 1;
  }

  public A(double x) {
    y -= 1;
    this.x = x;
  }

  public void f(int a) {
    this.x = a * 3;
  }

  public void f(float a) {
    this.x = a;
  }
}
```

```
public class B extends A {

  static int y = 0;

  public B() {
    super(4.2);
    y += 1;
  }

  public B(double x) {
    y -= 1;
    this.x = x * 2;
  }

  public void f(Float a) {
    this.x = (double)a * 2;
  }

  public void f(float a) {
    this.x = a * 4;
  }

}
```

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

        B b = new B();
        System.out.println(b.x + " " + B.y);    // OUT: [     ]   [     ]

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

        B b2 = new B(2.3);
        System.out.println(b2.x + " " + B.y);   // OUT: [     ]   [     ]

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

        b.f(5);
        System.out.println(b.x);                // OUT: [     ]

        A ab = b;
        Float k = 7.3f;
        ab.f(k);
        System.out.println(ab.x);               // OUT: [     ]
    }
}
```

Lösung:

```
public class M {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x + " " + A.y);    // OUT: [ 2.0 ]   [  1  ]

        B b = new B();
        System.out.println(b.x + " " + B.y);    // OUT: [ 4.2 ]   [  1  ]

        System.out.println(A.y);                // OUT: [  0  ]

        B b2 = new B(2.3);
        System.out.println(b2.x + " " + B.y);   // OUT: [ 4.6 ]   [  0  ]

        System.out.println(A.y);                // OUT: [  1  ]

        b.f(5);
        System.out.println(b.x);                // OUT: [ 15  ]

        A ab = b;
        Float k = 7.3f;
        ab.f(k);
        System.out.println(ab.x);               // OUT: [29.2 ]
    }
}
```

## Aufgabe 2 (Hoare-Kalkül): (11 Punkte)

Die Fibonacci-Folge ist für nicht-negative ganze Zahlen $x$ wie folgt definiert:

$$fib(x) = \begin{cases} 1 & \text{falls } x = 0 \text{ oder } x = 1 \\ fib(x-1) + fib(x-2) & \text{sonst} \end{cases}$$

Gegeben sei folgendes Java-Programm $P$.

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

```
if (x <= 1) {
    z = 1;
} else {
    y = 1;
    z = 1;
    i = 1;
    while (i < x) {
        z = y + z;
        y = z - y;
        i = i + 1;
    }
}
```

$\langle\, \texttt{z} = fib(\texttt{x}) \,\rangle$ (Nachbedingung)

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:

- 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. Benötigte Klammern dürfen Sie auch ohne Konsequenzregel ergänzen.

Lösung:

$\langle \texttt{x} \geq 0 \rangle$

```
if (x <= 1) {
```

$\langle \texttt{x} \geq 0 \wedge \texttt{x} \leq 1 \rangle$
$\langle \texttt{1} = fib(\texttt{x}) \rangle$

```
    z = 1;
```

$\langle \texttt{z} = fib(\texttt{x}) \rangle$

```
} else {
```

$\langle \texttt{x} \geq 0 \wedge \texttt{x} > 1 \rangle$
$\langle 1 = 1 \wedge 1 = 1 \wedge 1 = 1 \wedge \texttt{x} > 1 \rangle$

```
    y = 1;
```

$\langle \texttt{y} = 1 \wedge 1 = 1 \wedge 1 = 1 \wedge \texttt{x} > 1 \rangle$

```
    z = 1;
```

$\langle \texttt{y} = 1 \wedge \texttt{z} = 1 \wedge 1 = 1 \wedge \texttt{x} > 1 \rangle$

```
    i = 1;
```
$\langle \texttt{y} = 1 \land \texttt{z} = 1 \land \texttt{i} = 1 \land \texttt{x} > 1 \rangle$
$\langle \texttt{z} = \mathit{fib}(\texttt{i}) \land \texttt{y} = \mathit{fib}(\texttt{i} - 1) \land \texttt{i} \leq \texttt{x} \rangle$
```
    while (i < x) {
```
$\langle \texttt{z} = \mathit{fib}(\texttt{i}) \land \texttt{y} = \mathit{fib}(\texttt{i} - 1) \land \texttt{i} \leq \texttt{x} \land \texttt{i} < \texttt{x} \rangle$
$\langle \texttt{y} + \texttt{z} = \mathit{fib}(\texttt{i} + 1) \land \texttt{y} + \texttt{z} - \texttt{y} = \mathit{fib}(\texttt{i} + 1 - 1) \land \texttt{i} + 1 \leq \texttt{x} \rangle$
```
        z = y + z;
```
$\langle \texttt{z} = \mathit{fib}(\texttt{i} + 1) \land \texttt{z} - \texttt{y} = \mathit{fib}(\texttt{i} + 1 - 1) \land \texttt{i} + 1 \leq \texttt{x} \rangle$
```
        y = z - y;
```
$\langle \texttt{z} = \mathit{fib}(\texttt{i} + 1) \land \texttt{y} = \mathit{fib}(\texttt{i} + 1 - 1) \land \texttt{i} + 1 \leq \texttt{x} \rangle$
```
        i = i + 1;
```
$\langle \texttt{z} = \mathit{fib}(\texttt{i}) \land \texttt{y} = \mathit{fib}(\texttt{i} - 1) \land \texttt{i} \leq \texttt{x} \rangle$
```
    }
```
$\langle \texttt{z} = \mathit{fib}(\texttt{i}) \land \texttt{y} = \mathit{fib}(\texttt{i} - 1) \land \texttt{i} \leq \texttt{x} \land \texttt{i} \geq \texttt{x} \rangle$
$\langle \texttt{z} = \mathit{fib}(\texttt{x}) \rangle$
```
}
```
$\langle \texttt{z} = \mathit{fib}(\texttt{x}) \rangle$

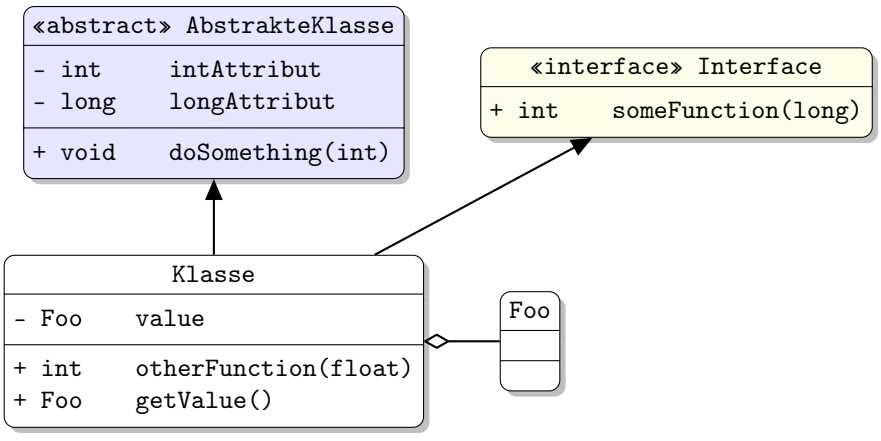
## Aufgabe 3 (Klassen-Hierarchie): (9 + 9 = 18 Punkte)

In dieser Aufgabe geht es darum, eine geeignete Klassenhierarchie zur Repräsentation eines Schuhschranks zu modellieren.

- In einem Schuhschrank befinden sich mehrere Schuhe.
- Jeder Schuh hat eine Größe und einen Abnutzungsgrad. Der Abnutzungsgrad wird durch eine ganze Zahl repräsentiert, wobei "1" für neuwertig steht und größere Werte stärkere Abnutzung bedeuten.
- Jeder Schuh ist entweder ein Sportschuh, ein Hausschuh, oder ein schicker Schuh. Andere Arten von Schuhen gibt es nicht.
- Sportschuhe neigen dazu, unangenehm zu riechen. Ihr Geruch wird ebenfalls durch eine ganze Zahl beschrieben, wobei "1" für geruchsneutral steht und größere Werte stärkeren Geruch bedeuten.
- Fußballschuhe und Rollschuhe sind spezielle Sportschuhe.
- Die Anzahl der Stollen ist ein wichtiges Merkmal eines Fußballschuhs.
- Elektrisch beheizbare Hausschuhe sind spezielle Hausschuhe. Diese bieten die Möglichkeit, die Heizfunktion ein- und auszuschalten.
- Da die Optik schicker Schuhe besonders wichtig ist, muss die Möglichkeit vorgesehen werden, schicke Schuhe zu polieren.
- Schicke Schuhe sind entweder Lackschuhe, Lederschuhe, oder High Heels. Andere schicke Schuhe gibt es nicht.
- Aufgrund der verwendeten Bauteile bietet sich das Recycling von beheizbaren Hausschuhen und Rollschuhen an. Deshalb soll eine Möglichkeit vorgesehen werden, diese Schuhe zu recyceln.

**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 $B \blacktriangleright A$, dass $A$ die Oberklasse von $B$ ist (also `class B extends A` bzw. `class B implements A`, falls $A$ ein Interface ist). Der Pfeil $B \multimap A$ 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)** Ergänzen Sie die Klasse Schuhschrank um eine Methode mit folgender Signatur:

```
public int aufraeumen()
```

Diese Methode soll für jeden Schuh, der in dem Schuhschrank steht, wie folgt verfahren: Falls der Abnutzungsgrad des Schuhs größer als drei ist oder es sich um einen Sportschuh handelt, dessen Geruch den Wert drei überschreitet, dann ist der Schuh nicht mehr akzeptabel. Er wird daher recycelt, falls dies möglich ist.

Falls es sich um einen akzeptablen beheizbaren Hausschuh handelt, wird die Heizfunktion ausgeschaltet. (Sie können davon ausgehen, dass das Ausschalten der Heizfunktion keinen Effekt hat, falls diese bereits deaktiviert war.) Wenn es sich um einen akzeptablen schicken Schuh handelt, wird dieser poliert.

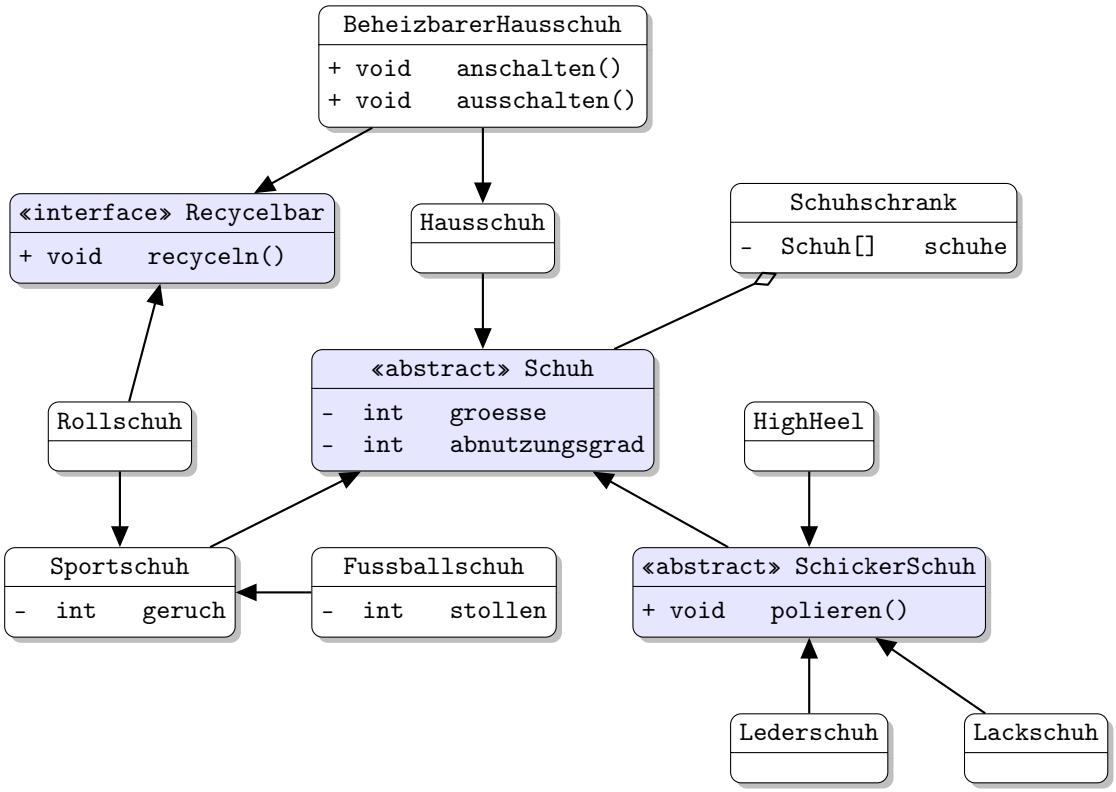
Die Rückgabe der Methode gibt an, wie viele Schuhe *nicht* mehr akzeptabel sind.

Hinweise:

- Gehen Sie hierbei davon aus, dass es für alle Attribute geeignete Selektoren (get- und set-Methoden) gibt.
- Sie können davon ausgehen, dass die Datenstruktur, die zur Speicherung von Schuhen in Schuhschränken genutzt wird, nicht `null` ist und auch nicht den Wert `null` enthält.

## Lösung:

**a)** Die Zusammenhänge können wie folgt modelliert werden:

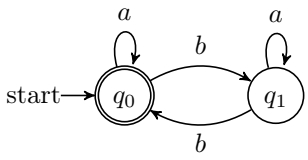



**b)**
```
public int aufraeumen() {
  int n = 0;
  for (Schuh s: schuhe) {
    if (s.getAbnutzungsgrad() > 3 ||
       (s instanceof Sportschuh && ((Sportschuh)s).getGeruch() > 3)) {
```

```
        if (s instanceof Recycelbar) {
          ((Recycelbar)s).recyceln();
        }
        n++;
      } else {
        if (s instanceof BeheizbarerHausschuh) {
          ((BeheizbarerHausschuh)s).ausschalten();
        } else if (s instanceof SchickerSchuh) {
          ((SchickerSchuh)s).polieren();
        }
      }
    }
    return n;
  }
```

## Aufgabe 4 (Programmieren in Java): (6 + 13 + 13 + 4 = 36 Punkte)

In dieser Aufgabe beschäftigen wir uns mit endlichen Automaten. Ein endlicher Automat besteht aus einer Menge von Zuständen (states) und einer Menge von Transitionen, die jeweils von einem Zustand in einen anderen führen. Jede Transition ist mit einem einzelnen Buchstaben markiert. Ein Automat hat einen eindeutigen Startzustand und einen oder mehrere Endzustände, die durch eine doppelte Umrandung gekennzeichnet werden. Im Beispiel ist ein Automat dargestellt, der die beiden Zustände $q_0$ und $q_1$ enthält. Hierbei ist $q_0$ der Startzustand und gleichzeitig der einzige Endzustand. Es gibt eine Transition von $q_0$ zu $q_1$, die mit dem Buchstaben $b$ markiert ist, eine Transition von $q_0$ zu $q_0$, die mit $a$ markiert ist, eine Transition von $q_1$ zu $q_0$, die mit $b$ markiert ist, und eine Transition von $q_1$ zu $q_1$, die mit $a$ markiert ist.




In dieser Aufgabe gehen wir davon aus, dass die Datenstruktur mit den folgenden Klassen im gleichen Paket realisiert ist:

```
public class Automaton {
  State start;                          // Startzustand des Automaten
}

public class State {
  boolean isFinal;                      // Gibt an, ob dieser Zustand ein Endzustand ist
  LinkedList<Transition> transitions; // Liste der von diesem Zustand ausgehenden
}                                       // Transitionen


public class Transition {
  State destination;                    // Der Zielzustand, zu dem die Transition führt
  char letter;                          // Der Buchstabe, mit dem die Transition markiert ist
}
```

Hinweise:

- Sie dürfen in allen Teilaufgaben zusätzliche Hilfsmethoden schreiben. Geben Sie bei diesen Hilfsmethoden jeweils an, in welcher Klasse diese implementiert sind.
- Sie können davon ausgehen, dass bei allen Automaten der Startzustand `start` nicht `null` ist.

**a)** Implementieren Sie eine öffentliche Methode `State read(char c)` in der Klasse `State`. Diese soll den Zustand zurückgeben, zu dem man gelangt, wenn man vom aktuellen Zustand aus der Transition folgt, die mit dem Buchstaben `c` markiert ist. Sie können hierbei davon ausgehen, dass von einem Zustand aus niemals mehrere Transitionen ausgehen, die mit dem gleichen Buchstaben markiert sind. Falls keine entsprechende Transition existiert, soll eine `InvalidLetterException` geworfen und hierbei der ungültige Buchstabe `c` übergeben werden. Die Exception ist wie folgt definiert:

```
public class InvalidLetterException extends Exception {
  private char letter;
  public InvalidLetterException(char l) {
    this.letter = l;
  }
  public String toString() {
    return "Invalid Letter: " + this.letter;
  }
}
```

**b)** Schreiben Sie eine öffentliche Methode `boolean read(String word)` in der Klasse `Automaton`. Diese soll den Zustand berechnen, zu dem man gelangt, wenn man ausgehend vom Startzustand nacheinander den Transitionen folgt, die die Buchstaben von `word` enthalten. Falls dieser Zustand ein Endzustand ist, soll `true` zurückgegeben werden und sonst `false`. Beispielsweise soll der Aufruf von `read("baba")` auf obigem Beispielautomaten den Rückgabewert `true` liefern, während `read("baa")` `false` zurückgeben soll. Sie können wieder davon ausgehen, dass von einem Zustand aus niemals mehrere Transitionen ausgehen, die mit dem gleichen Buchstaben markiert sind. Verwenden Sie die Methode `read` aus Teilaufgabe a). Falls eine `InvalidLetterException` auftritt, soll diese gefangen werden, eine Fehlermeldung ausgegeben werden und `false` zurückgegeben werden.

Verwenden Sie in dieser Aufgabe **nur Rekursion** und **keine Schleifen**.

Hinweise:

- Sie können davon ausgehen, dass `word` nicht `null` ist.
- Sie dürfen die Methode `charAt` der Klasse `String` verwenden, die einen Index $i$ übergeben bekommt und das Zeichen an der Stelle $i$ zurückgibt. Hierbei hat das Zeichen an erster Stelle den Index 0. Ruft man beispielsweise `charAt(2)` auf `"abcde"` auf, erhält man als Ergebnis `'c'`. Hingegen führt `charAt(5)` zu einer Exception.
- Sie dürfen die Methode `substring` der Klasse `String` verwenden, die einen Index $i$ übergeben bekommt und den Teil-String zurückgibt, der an der Stelle $i$ beginnt. Ruft man beispielsweise `substring(2)` auf `"abcde"` auf, erhält man als Ergebnis `"cde"`. Das Ergebnis von `substring(5)` ist `""`.

**c)** Schreiben Sie eine öffentliche Methode `LinkedList<State> getReachableStates()` in der Klasse `Automaton`. Diese soll eine Liste mit allen Zuständen zurückgeben, zu denen man ausgehend vom Startzustand gelangen kann, wenn man beliebig vielen Transitionen folgt. Beispielsweise soll der Aufruf der Methode auf obigem Beispielautomaten eine zweielementige Liste mit den Zuständen $q_0$ und $q_1$ zurückliefern.

Verwenden Sie in dieser Aufgabe **nur Schleifen** und **keine Rekursion**.

Hinweise:

- Sie dürfen die Methode `size()` der Klasse `LinkedList` verwenden, die die Anzahl der Elemente der Liste zurückgibt.
- Außerdem dürfen Sie die Methode `contains` verwenden. Hierbei gibt `contains(x)` an, ob das Element `x` in der aktuellen Liste enthalten ist.

**d)** Passen Sie die Deklarationen der Klassen `Automaton`, `State` und `Transition` so an, dass anstelle von `char`-Werten beliebige Objekte als Markierung der Transitionen verwendet werden können. Stellen Sie dabei sicher, dass diese Objekte innerhalb eines Automaten von der gleichen Klasse stammen.

Sie brauchen **keine Methoden** anpassen, **nur** die **Klassen und Attribute**.

Lösung:

**a)**
```
public State read(char c) throws InvalidLetterException {
  if (this.transitions != null) {
    for (Transition t : this.transitions) {
      if (t.letter == c) {
        return t.destination;
      }
    }
  }
  throw new InvalidLetterException(c);
}
```

**b)** In der Klasse `Automaton`:

```
public boolean read(String word) {
  return this.start.read(word);
}
```

In der Klasse `State`:

```
protected boolean read(String word) {
  if (word.equals("")) {
    return this.isFinal;
  }
  State dest;
  try {
    dest = this.read(word.charAt(0));
  } catch (InvalidLetterException e) {
    System.out.println(e);
    return false;
  }
  return dest.read(word.substring(1));
}
```

**c)**
```
public LinkedList<State> getReachableStates() {
  LinkedList<State> oldStates = new LinkedList<State>();
  LinkedList<State> newStates = new LinkedList<State>();
  newStates.add(this.start);
  while (newStates.size() != oldStates.size()) {
    oldStates = newStates;
    newStates = new LinkedList<State>();
    for (State s : oldStates) {
      if (!newStates.contains(s)) {
        newStates.add(s);
      }
      for (Transition t : s.transitions) {
        if (!newStates.contains(t.destination)) {
          newStates.add(t.destination);
        }
      }
    }
  }
  return newStates;
}
```

**d)**
```
public class Automaton<T> {
  State<T> start;
}

public class State<T> {
  boolean isFinal;
  LinkedList<Transition<T>> transitions;
}

public class Transition<T> {
  State<T> destination;
  T letter;
}
```

## Aufgabe 5 (Haskell): (4 + 5 + 2 + 1 + 4 + 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 y [] = [x]
f x y (z:zs) = if y > 0 then f z y zs else f x (y-1) zs

g x 0 z = z
g x y z = g 0 z x
```

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

```
i :: [Int]
i = (\a b -> b ++ a) (filter (\x -> x > 2) [1,2,3]) (filter (\x -> x < 2) [1,2,3])

j :: [Int]
j = map (\a -> a 1) (map (\a x -> x + a) [1,2,3])
```

**c)** Geben Sie die Definition einer Datenstruktur `Q` zur exakten Darstellung von rationalen Zahlen an. Diese soll ganze Zahlen durch eine Zahl vom Typ `Int` repräsentieren. Alle anderen rationalen Zahlen werden als zwei Zahlen vom Typ `Int` dargestellt, die für den Zähler und Nenner des Bruchs stehen.

**d)** Implementieren Sie die Funktion `definiert :: Q -> Bool`. Diese soll für alle Brüche mit Nenner `0` den Wert `False` zurück geben. In jedem anderen Fall soll die Rückgabe `True` sein.

**e)** Implementieren Sie die Funktion `vereinfache :: Q -> Q`. Diese soll Brüche so weit wie möglich kürzen, d.h. Zähler und Nenner durch den größten gemeinsamen Teiler teilen. Außerdem sollen nach dem Kürzen alle Brüche mit Nenner `1` oder `-1` in die entsprechende Darstellung mit nur einer Zahl umgewandelt werden. Brüche mit Nenner `0` sollen nicht verändert werden.

Hinweise:

Sie dürfen in dieser Aufgabe die folgenden vordefinierten Funktionen benutzen:

- `gcd :: Int -> Int -> Int` berechnet den größten gemeinsamen Teiler zweier Zahlen. Das Ergebnis ist 0, falls eine der Zahlen 0 ist. Sonst ist das Ergebnis immer $\geq 1$.
- `div :: Int -> Int -> Int` berechnet die Ganzzahldivision. Beachten Sie, dass Ganzzahldivision durch 0 zu einem Fehler führt.

**f)** Implementieren Sie die Funktion `vereinfacheListe :: [Q] -> [Q]`. Diese soll alle rationalen Zahlen in der Eingabeliste mittels `vereinfache` vereinfachen und alle undefinierten Werte, also alle Brüche mit Nenner `0`, entfernen.

### Lösung:

**a)** i) `f :: a -> Int -> [a] -> [a]`

ii) `g :: Int -> Int -> Int -> Int`

**b)**
```
i = [1,3]
j = [2,3,4]
```

**c)**
```
data Q = Bruch Int Int | Z Int
```

**d)**
```
definiert :: Q -> Bool
definiert (Bruch x 0) = False
definiert _ = True
```

**e)**
```
vereinfache :: Q -> Q
vereinfache (Z x) = Z x
vereinfache (Bruch x 1) = Z x
vereinfache (Bruch x (-1)) = Z (-x)
vereinfache (Bruch x 0) = Bruch x 0
vereinfache (Bruch x y) = if g > 1 then vereinfache (Bruch (div x g) (div y g))
                          else Bruch x y
                            where g = gcd x y
```

**f)**
```
vereinfacheListe :: [Q] -> [Q]
vereinfacheListe xs = filter definiert (map vereinfache xs)
```

## Aufgabe 6 (Prolog): (2 + 8 + 2 + 3 + 5 = 20 Punkte)

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

i) $f(b, X, s(a)), f(X, Z, s(Z))$

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

**b)** Gegeben sei folgendes Prolog-Programm $P$.

```
p(X,s(X)) :- p(s(X),Z), p(s(Z),0).
p(s(0),0).
```

Erstellen Sie für das Programm $P$ den Beweisbaum zur Anfrage "`?- p(A,B).`" bis zur Höhe 4 (die Wurzel hat dabei die Höhe 1). Markieren Sie Pfade, die zu einer unendlichen Auswertung führen, mit $\infty$ und geben Sie alle Antwortsubstitutionen zur Anfrage "`?- p(A,B).`" an, die im Beweisbaum bis zur Höhe 4 enthalten sind. Geben Sie außerdem zu jeder dieser Antwortsubstitutionen an, ob sie von Prolog gefunden wird. Geben Sie schließlich noch ein logisch äquivalentes Programm an, das durch eine einzige Vertauschung entsteht und bei dem der Beweisbaum zur obigen Anfrage endlich ist.

**c)** Natürliche Zahlen lassen sich mit Hilfe der Funktionssymbole `0` und `s` in sogenannter *Peano-Notation* darstellen (d. h., der Term `s(0)` stellt die Zahl 1 dar, `s(s(0))` stellt 2 dar, etc.). Implementieren Sie ein Prädikat `countUp` mit Stelligkeit 2 in Prolog, sodass die Anfrage `countUp(`$n$`,X)` für eine gegebene natürliche Zahl $n$ in Peano-Notation nacheinander alle Zahlen von 0 bis $n$ in Peano-Notation aufzählt. Beispielsweise soll die Anfrage `countUp(s(s(s(0))),X)` folgende Antworten liefern.

```
X = 0 ;
X = s(0) ;
X = s(s(0)) ;
X = s(s(s(0))).
```

Alle Anfragen der Form `countUp(`$n$`,`$x$`)`, bei denen $n$ keine Variablen enthält, sollen terminieren.

**d)** Implementieren Sie ein Prädikat `sum` mit Stelligkeit 2 in Prolog, wobei `sum(`$t$`,`$n$`)` genau dann gilt, wenn $t$ eine Liste natürlicher Zahlen in Peano-Notation ist und $n$ die Summe aller Zahlen in $t$ ist. Beispielsweise soll die Anfrage `sum([s(s(0)),0,s(0)],N)` die einzige Antwort `N = s(s(s(0)))` liefern. Alle Anfragen der Form `sum(`$t$`,`$n$`)`, bei denen $t$ keine Variablen enthält, sollen terminieren.

**e)** Implementieren Sie ein Prädikat `firstGreater` mit Stelligkeit 3 in Prolog, wobei `firstGreater(`$t$`,`$n$`,`$i$`)` genau dann gilt, wenn $t$ eine Liste vordefinierter Prolog-Zahlen und $i$ der Index der ersten Zahl in $t$ ist, die strikt größer ist als $n$. Hierbei ist 0 der Index der ersten Zahl in einer Liste. Beispielsweise soll die Anfrage `firstGreater([5,6,7,5],6,I)` die einzige Antwort `I = 2` liefern. Die Anfrage `firstGreater([5,6,7,5],7,I)` ergibt `false`. Alle Anfragen der Form `firstGreater(`$t$`,`$n$`,`$i$`)`, bei denen $t$ und $n$ keine Variablen enthalten, sollen terminieren.

Hinweise:

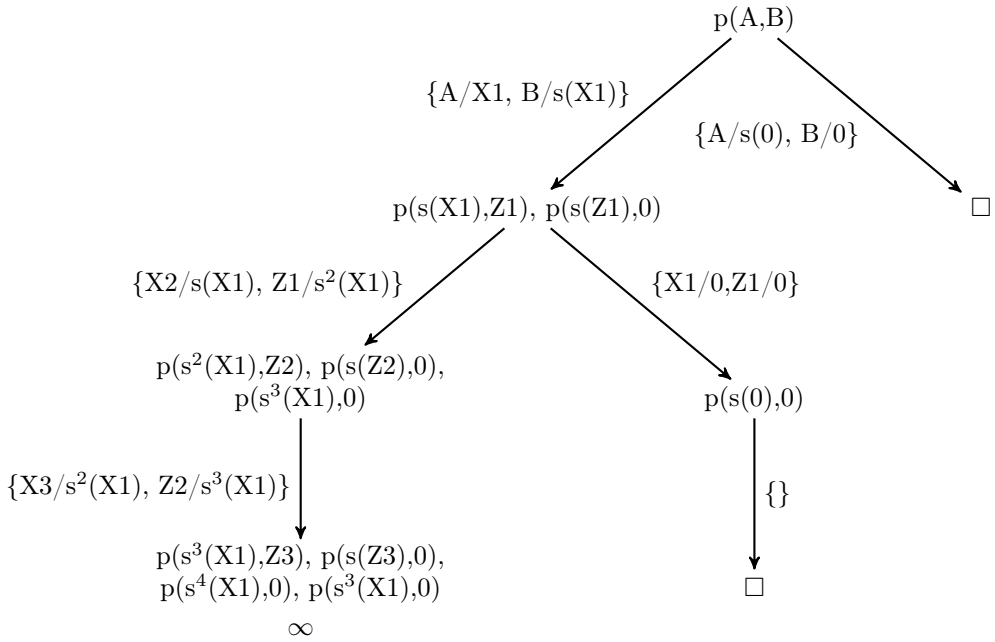
- In dieser Teilaufgabe sollen keine Zahlen in Peano-Notation, sondern die vordefinierten ganzen Zahlen von Prolog verwendet werden.

### Lösung:

**a)** i) $f(b, X, s(a)), f(X, Z, s(Z))$: clash failure $a \neq b$

ii) $g(X, Y, X), g(Y, Y, s(Z))$: $X/s(Z), Y/s(Z)$

**b)**




Die beiden Antwortsubstitutionen innerhalb des Beweisbaums sind {`A/s(0),B/0`} und {`A/0,B/s(0)`}. Diese werden von Prolog nicht gefunden.

Mit dem folgenden logisch äquivalenten Programm ist der Beweisbaum zur Anfrage endlich:

```
p(X,s(X)) :- p(s(Z),0), p(s(X),Z).
p(s(0),0).
```

**c)**
```
countUp(_,0).
countUp(s(X),s(Y)) :- countUp(X,Y).
```

Alternative Lösung:

```
countUp(s(X),Y) :- countUp(X,Y).
countUp(X,X).
```

**d)**
```
sum([],0).
sum([0|XS],R) :- sum(XS,R).
sum([s(X)|XS],s(Y)) :- sum([X|XS],Y).
```

**e)**
```
firstGreater([X|_],Y,0) :- X > Y.
firstGreater([X|XS],Y,I) :- X =< Y, firstGreater(XS,Y,I2), I is I2+1.
```