Aufgabe 1 (Programmanalyse): (14 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 {

    public int x = 42;

    public static double y = 3;

    public A() {
        y++;
    }

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

    public void f(int y) {
        this.x = 2 * y;
        y = 0;
    }
}
```

```
public class B extends A {

    public double x = 0;

    public B(double x) {
        this.x++;
    }

    public void f(int x) {
        this.x += x;
    }

    public void f(double x) {
        this.x = 3 * x;
    }
}
```

```
public class M {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x + " " + A.y);  // OUT: [ 42   ]   [ 4.0  ]

        a.f(-5);
        System.out.println(a.x + " " + A.y);  // OUT: [ -10  ]   [ 4.0  ]

        B b = new B(4);
        System.out.println(b.x + " " + A.y);  // OUT: [ 1    ]   [ 5.0  ]

        A z = b;
        System.out.println(z.x);              // OUT: [ 42   ]

        z.f(2);
        System.out.println(b.x);              // OUT: [ 6.0  ]

        ((B) z).f(-1.0);
        System.out.println(b.x);              // OUT: [ -6.0 ]
    }
}
```

Lösung:

```
public class M {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.x + " " + A.y);  // OUT: [ 42  ]   [ 4.0 ]

        a.f(-5);
        System.out.println(a.x + " " + A.y);  // OUT: [ -10 ]   [ 4.0 ]

        B b = new B(4);
        System.out.println(b.x + " " + A.y);  // OUT: [ 1.0 ]   [ 5.0 ]

        A z = b;
        System.out.println(z.x);              // OUT: [ 42  ]

        z.f(2);
        System.out.println(b.x);              // OUT: [ 3.0 ]

        ((B) z).f(-1.0);
        System.out.println(b.x);              // OUT: [-3.0 ]
    }
}
```

Aufgabe 2 (Hoare-Kalkül): (10 + 4 = 14 Punkte)

Gegeben sei folgendes Java-Programm P, das zu einer Eingabe $m \leq n$ den Wert $2^{n-m} - 1$ berechnet.

```
⟨ m ≤ n ⟩                    (Vorbedingung)
i = m;
res = 0;
while (i < n) {
    res = 2 * res + 1;
    i = i + 1;
}
⟨ res = 2^(n−m) − 1 ⟩        (Nachbedingung)
```

a) Vervollständigen Sie die Verifikation des Algorithmus P auf der folgenden Seite im Hoare-Kalkül, indem Sie die unterstrichenen Teile ergänzen. Hierbei dürfen zwei Zusicherungen nur dann direkt untereinander stehen, wenn die untere aus der oberen folgt. Hinter einer Programmanweisung darf nur dann 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 $\langle \mathtt{x}+1 = \mathtt{y}+1 \rangle$ zu $\langle \mathtt{x} = \mathtt{y} \rangle$) nur unter Zuhilfenahme der Konsequenzregeln vornehmen dürfen. Dies betrifft allerdings nicht das Setzen von Klammern, wenn dies durch Anwendung der Zuweisungsregel nötig ist (z. B. beim Hoare Tripel $\langle \mathtt{y} = \mathtt{z} - (\mathtt{x}+1) \rangle$ `x = x + 1;` $\langle \mathtt{y} = \mathtt{z} - \mathtt{x} \rangle$).

b) Untersuchen Sie den Algorithmus P auf seine Terminierung. Für einen Beweis der Terminierung muss eine Variante angegeben werden und unter Verwendung des Hoare-Kalküls die Terminierung unter der Voraussetzung $\mathtt{m} \leq \mathtt{n}$ bewiesen werden. Begründen Sie, warum es sich bei der von Ihnen angegebenen Variante tatsächlich um eine gültige Variante handelt.

Lösung:

a)

$\langle \mathtt{m} \leq \mathtt{n} \rangle$
$\langle \mathtt{m} \leq \mathtt{n} \wedge \mathtt{m} = \mathtt{m} \wedge 0 = 0 \rangle$
`i = m;`
$\langle \mathtt{m} \leq \mathtt{n} \wedge \mathtt{i} = \mathtt{m} \wedge 0 = 0 \rangle$
`res = 0;`
$\langle \mathtt{m} \leq \mathtt{n} \wedge \mathtt{i} = \mathtt{m} \wedge \mathtt{res} = 0 \rangle$
$\langle \mathtt{res} = 2^{\mathtt{i}-\mathtt{m}} - 1 \wedge \mathtt{i} \leq \mathtt{n} \rangle$
`while (i < n) {`
$\langle \mathtt{res} = 2^{\mathtt{i}-\mathtt{m}} - 1 \wedge \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} < \mathtt{n} \rangle$
$\langle 2 \cdot \mathtt{res} + 1 = 2^{(\mathtt{i}+1)-\mathtt{m}} - 1 \wedge \mathtt{i} + 1 \leq \mathtt{n} \rangle$
`res = 2 * res + 1;`
$\langle \mathtt{res} = 2^{(\mathtt{i}+1)-\mathtt{m}} - 1 \wedge \mathtt{i} + 1 \leq \mathtt{n} \rangle$
`i = i + 1;`
$\langle \mathtt{res} = 2^{\mathtt{i}-\mathtt{m}} - 1 \wedge \mathtt{i} \leq \mathtt{n} \rangle$
`}`
$\langle \mathtt{res} = 2^{\mathtt{i}-\mathtt{m}} - 1 \wedge \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \not< \mathtt{n} \rangle$
$\langle \mathtt{res} = 2^{\mathtt{n}-\mathtt{m}} - 1 \rangle$

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

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

Damit ist die Terminierung der einzigen Schleife in P gezeigt.

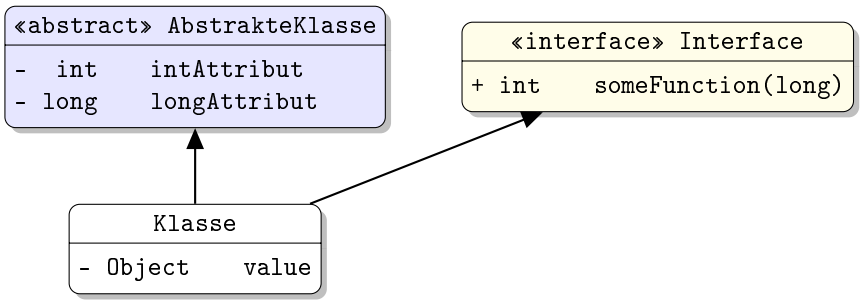
Aufgabe 3 (Klassen-Hierarchie): (6 + 10 = 16 Punkte)

In dieser Aufgabe betrachten wir eine Kontoverwaltung mittels doppelter Buchführung. Dabei betrachten wir verschiedene Sorten von Konten und Transaktionen.

- Ein `Konto` hat immer einen Namen, einen Geldbetrag und ein Array, in dem alle Transaktionen gespeichert sind, die das Konto betreffen.
- Eine `Transaktion` enthält immer einen Wert, der übertragen wird.
- Von einigen Konten aus dürfen Auszahlungen getätigt werden. Dazu stellen diese die Methode `void auszahlung(Transaktion t)` zur Verfügung. Darüber hinaus kann ein solches Konto immer alle seine bereits ausgezahlten Transaktionen mithilfe einer Methode `Transaktion[] getAuszahlungen()` zurückgeben.
- Auf einige Konten dürfen Beträge eingezahlt werden. Diese Konten stellen die Methode `void einzahlung(Transaktion t)` zur Verfügung. Darüber hinaus kann ein solches Konto immer alle seine bereits eingezahlten Transaktionen mithilfe einer Methode `Transaktion[] getEinzahlungen()` zurückgeben.
- Eine `Transaktion` geht immer von einem `Konto` aus, das Auszahlungen tätigen darf. Eine `Transaktion` geht immer zu einem `Konto`, auf das eingezahlt werden darf. Beide Konten sind für eine `Transaktion` interessant.
- `RisikoKapital` ist eine Form von `Konto`, von dem nur Auszahlungen getätigt werden können.
- Ein `AbschreibungsKonto` ist ein `Konto`, auf das nur eingezahlt werden darf.
- Ein `GiroKonto` ist ein `Konto`, das sowohl für Einzahlungen als auch für Auszahlungen genutzt werden kann.
- Es gibt keine Konten, welche weder für Einzahlungen noch für Auszahlungen benutzt werden können.

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

Verwenden Sie hierbei die folgende Notation:



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

Hinweise:

- Sie brauchen keine "Uses" Beziehungen der Form $B \dashrightarrow A$ einzuzeichnen, die aussagen, dass `A` den Typ `B` verwendet.
- Wenn eine Klasse nur die Methoden eines Interfaces implementiert, können Sie diese durch ... abkürzen.

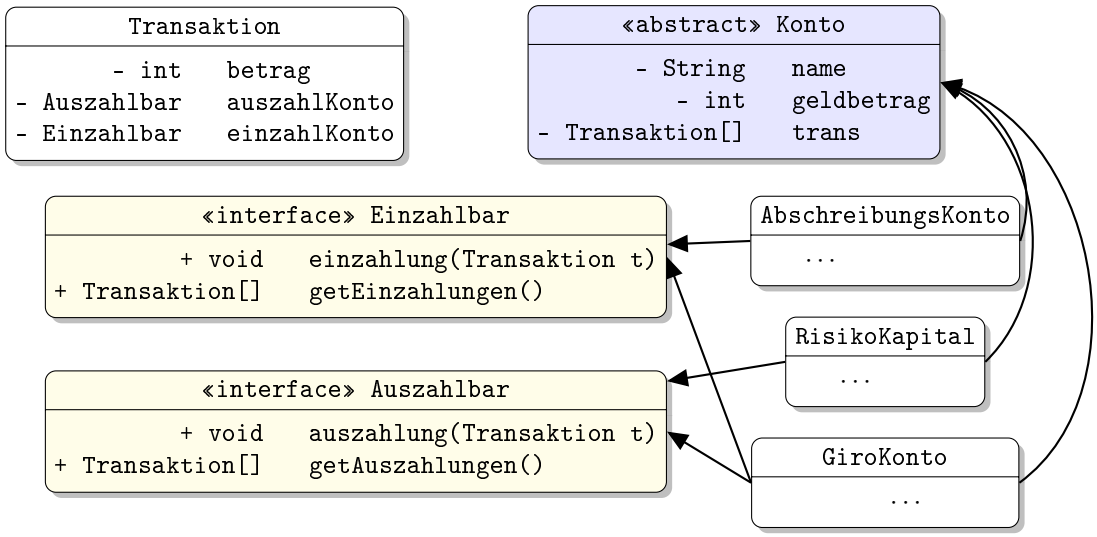
b) Schreiben Sie eine Java-Methode mit der folgenden Signatur:

```
public static boolean wertPruefen(Konto[] konten)
```

Diese Methode soll zurückgeben, ob alle Konten im Array `konten` *korrekt geführt* sind. Hierbei ist ein Konto genau dann *korrekt geführt*, wenn die Summe der Werte seiner bereits eingezahlten Transaktionen minus der Summe der Werte seiner bereits ausgezahlten Transaktionen gleich dem Betrag des Kontos ist. Gehen Sie davon aus, dass das `konten` Array nicht `null` ist, dass kein Konto in diesem Array `null` ist und dass es zu jedem Attribut geeignete Selektoren gibt.

Lösung:

a) Die Zusammenhänge können wie folgt modelliert werden:



b)

```
private static boolean wertPruefen(Konto k) {
  int sum = 0;
  // Addiere alle eingehenden Transaktionen.
  if (k instanceof Einzahlbar) {
    for (Transaktion t : ((Einzahlbar)k).getEinzahlungen()) { sum+= t.getBetrag();}
  }
  // Subtrahiere alle ausgehenden Transaktionen.
  if (k instanceof Auszahlbar) {
    for (Transaktion t : ((Auszahlbar)k).getAuszahlungen()) { sum-= t.getBetrag();}
  }
  return sum == k.getGeldbetrag();
}

public static boolean wertPruefen(Konto[] konten) {
  // Gib false zurueck, falls ein Konto nicht korrekt gefuehrt ist.
  for (Konto k : konten) {
    if (!wertPruefen(k)) {
      return false;
    }
  }
  // Wenn alle Konten korrekt gefuehrt waren, gib true zurueck.
  return true;
}
```

Aufgabe 4 (Programmieren in Java): (4 + 5 + 7 + 10 + 10 = 36 Punkte)

In dieser Aufgabe betrachten wir arithmetische Ausdrücke, welche sich aus Variablen, Konstanten und Operatoren zusammensetzen.

```
public interface Arith {
  public int tiefe();
  public Arith ersetzen(Var var, Arith val);
  //... ggf. weitere Methoden
}

public class Var implements Arith {
  public final String name;
  public Var(String n) {name = n;}
  public int tiefe() {return 1;}
  //... weitere Methoden
}

public class Const implements Arith {
  public final int value;
  public Const(int v) {value = v;}
  public int tiefe() {return 1;}
  //... weitere Methoden
}

public class Op implements Arith {
  public final String operation;
  public final Arith left;
  public final Arith right;
  public Op(String o, Arith l, Arith r) {
    operation = o;
    left = l;
    right = r;
  }
  //... weitere Methoden
}
```

Der Ausdruck:
$(4+3)-x$
würde folgendermaßen als Objekt vom Typ `Arith` dargestellt.

```
Var x = new Var("x");
Arith t =
  new Op(
    "-",
    new Op(
      "+",
      new Const(4),
      new Const(3)
    ),
    x
  );
```

Sie können davon ausgehen, dass die beiden Teilausdrücke, die in den Attributen `left` oder `right` eines `Op`-Objekts gespeichert sind, nie `null` sind. Genauso sind die Attribute `name` in der Klasse `Var` und `operation` in der Klasse `Op` nie `null`. Zur Lösung der folgenden Aufgaben dürfen Sie die gegebenen Klassen und das Interface um beliebige Methoden erweitern.

a) Implementieren Sie eine Methode `tiefe()` in der Klasse `Op`, welche die Schachtelungs-Tiefe des aktuellen Ausdrucks zurückgibt. Verwenden Sie zur Lösung dieser Teilaufgabe **nur Rekursion** und **keine Schleifen**.

Hinweise:

- Die Tiefe von Variablen und Konstanten ist 1. Die Tiefe des Ausdrucks $(4+3)$ ist 2 und die Tiefe von `t` aus dem obigen Beispiel ist 3.

b) Implementieren Sie die statische Methode `public static boolean shallow(Arith[] exs)`, welche genau dann `true` zurückgibt, wenn `exs` nicht `null` ist und jedes Element von `exs` ein Ausdruck der Tiefe 1 oder 2 ist. Ihre Implementierung soll unter keinen Umständen eine `NullPointerException` werfen. Verwenden Sie zur Lösung dieser Teilaufgabe **nur Schleifen** und **keine Rekursion** – Sie dürfen jedoch die Methode `tiefe` aus dem vorigen Aufgabenteil aufrufen.

c) Implementieren Sie die Methode `ersetzen(Var var, Arith val)`, die von dem Interface `Arith` gefordert wird, in allen drei Klassen `Var`, `Const` und `Op`. Diese Methode soll einen neuen Ausdruck zurückgeben,

der dadurch entsteht, dass alle Vorkommen von `var` im aktuellen Ausdruck durch den Teilausdruck `val` ersetzt werden. Der aktuelle Ausdruck soll nicht verändert werden. Sie können davon ausgehen, dass `var` und `val` nicht `null` sind.

Hinweise:

- `t.ersetzen(x, new Const(5))` ergibt ein Objekt, das den Ausdruck $(4 + 3) - 5$ repräsentiert.

d) Implementieren Sie die statische Methode `public static List<Var> variablen(Arith e)`, die eine Liste mit genau den Variablen aus dem übergebenen Ausdruck `e` zurückgibt. Dabei darf jede Variable höchstens einmal in der Ergebnisliste enthalten sein (d. h. die Liste enthält genau die Menge der Variablen in `e`; die Reihenfolge der Variablen in der Liste ist unerheblich). Verwenden Sie das Interface `List<T>` aus dem `Collections`-Framework und instanziieren Sie `T` geeignet. Für unseren Beispielausdruck `t` soll also eine Liste zurückgegeben werden, die nur `x` enthält. Sie können davon ausgehen, dass `e` nicht `null` ist.

Hinweise:

- Die Klasse `LinkedList<T>` implementiert das Interface `List<T>` und verfügt über eine Methode `add(T o)`, die das Element `o` an das *Ende* der aktuellen Liste anhängt.
- Die Klasse `LinkedList<T>` verfügt über eine Methode `boolean contains(Object o)`, die genau dann `true` zurückgibt, wenn `o` in der aktuellen Liste enthalten ist.
- Die Klasse `LinkedList<T>` hat einen Konstruktor ohne Argumente, der eine leere Liste erzeugt.

e) Wir betrachten nun das Interface `Auswerter`, mit dem eine Funktionalität zur Auswertung von Ausdrücken implementiert werden soll.

```
public interface Auswerter {
  public int handleVar(Var v);
  public int handleConst(Const c);
  public int handleOp(Op o, int l, int r);
}
```

Zur Anwendung eines `Auswerters` `f` auf ein Objekt `e` vom Typ `Arith` dient die Funktion

`public static int apply(Auswerter f, Arith e)`.

Um einen Auswerter `f` auf eine Instanz `o` der Klasse `Op` anzuwenden, werden zuerst die Ergebnisse `l` und `r` der Anwendung von `f` auf den linken bzw. rechten Teilausdruck bestimmt. Das Ergebnis der Anwendung von `f` ist dann `f.handleOp(o,l,r)`. Für Instanzen von `Var` und `Const` ergibt sich der Wert der Anwendung von `f` durch `f.handleVar(..)` und `f.handleConst(..)`.

Die Klasse `Compute` ist ein Beispiel für die Verwendung des Interfaces `Auswerter`. Sie dient dazu, einen Ausdruck "auszurechnen", wobei der Wert jeder Variablen 0 ist und man davon ausgeht, dass nur die Operationen "+" und "-" vorkommen.

```
public class Compute implements Auswerter {
  public int handleVar(Var v) {
    return 0;
  }

  public int handleConst(Const c) {
    return c.value;
  }

  public int handleOp(Op o, int l, int r) {
    if (o.operation.equals("+")) {
      return l + r;
    } else {
      return l - r;
    }
  }
}
```

Implementieren Sie die statische Methode `apply` mit der obigen Signatur, die einen `Auswerter` wie oben beschrieben auf einen Ausdruck `e` anwendet. Wenn also `t` der Ausdruck vom Anfang der Aufgabe ist, so soll `apply(new Compute(), t)` die Auswertung des Ausdrucks $(4+3)-0$ (d. h. den Wert 7) ergeben. Sie können davon ausgehen, dass die Parameter `f` und `e` der Methode `apply` nicht mit `null` belegt werden.

Lösung:

a)
```
public int tiefe() {
  // Gib das Maximum der linken bzw. rechten Tiefe plus eins zurueck.
  if (left.tiefe() > right.tiefe()) {
    return left.tiefe() + 1;
  }
  return right.tiefe() + 1;
}
```

b)
```
public static boolean shallow(Arith[] exs) {
  if (exs == null) {
    return false;
  }
  for (Arith e : exs) {
    if (e == null || e.tiefe() > 2) {
      return false;
    }
  }
  return true;
}
```

Alternative mit `try/catch`:

```
public static boolean shallow(Arith[] exs) {
  try {
    for (Arith e : exs) {
      if (e.tiefe() > 2) {
        return false;
      }
    }
  } catch (NullPointerException ex) {
    return false;
  }
  return true;
}
```

c)
```
public class Var implements Arith {
  ....
  public Arith ersetzen(Var var, Arith val) {
    // Eine Variable wird durch val ersetzt, wenn sie gleich var ist.
    if (var == this) {return val;}
    return this;
  }
  ....
}

public class Const implements Arith {
  ....
```

```
  public Arith ersetzen(Var var, Arith val) {
    // Const wird nie ersetzt.
    return this;
  }
  ....
}

public class Op implements Arith {
  ....
  public Arith ersetzen(Var var, Arith val) {
    // Wende die Ersetzung auf beide Teilausdruecke an.
    return
      new Op(
        operation,
        left.ersetzen(var,val),
        right.ersetzen(var,val)
      );
  }
  ....
}
```

d)

```
private static void vars(Arith e, LinkedList<Var> res) {
  if (e instanceof Var) {
    // Fuege Variable zum Ergebnis hinzu, falls nicht enthalten
    if (!res.contains(e)) {
      res.add((Var)e);
    }
  } else if (e instanceof Op) {
    // Betrachte beide Teilausdruecke
    Op op = (Op)e;
    vars(op.left, res);
    vars(op.right, res);
  }
}

public static List<Var> variablen(Arith e) {
  LinkedList<Var> res = new LinkedList<Var>();
  vars(e, res);
  return res;
}
```

Alternative ohne instanceof:

```
// in Arith:
void vars(LinkedList<Var> res);

// in Var:
public void vars(LinkedList<Var> res) {
  // Fuege Variable zum Ergebnis hinzu, falls nicht enthalten
  if (!res.contains(this)) {
    res.add(this);
  }
}

// in Op:
public void vars(LinkedList<Var> res) {
  // Betrachte beide Teilausdruecke
```

```
    this.left.vars(res);
    this.right.vars(res);
  }

  // in Const:
  public void vars(LinkedList<Var> res) {}

  // irgendwo:
  public static List<Var> variablen(Arith e) {
    LinkedList<Var> res = new LinkedList<Var>();
    e.vars(res);
    return res;
  }
```

Alternative, die den `Auswerter` benutzt:

```
public class VarCollector implements Auswerter {

  public LinkedList<Var> res;

  public int handleVar(Var v) {
    // Fuege Variable zum Ergebnis hinzu, falls nicht enthalten
    if (!res.contains(v)) {
      res.add(v);
    }
    return 0;
  }

  public int handleConst(Const c) {
    return 0;
  }

  public int handleOp(Op o, int l, int r) {
    return 0;
  }
}

  public static List<Var> variablen(Arith e) {
    Auswerter f = new VarCollector();
    f.res = new LinkedList<Var>();
    apply(f, e);
    return f.res;
  }
```

e)

```
  public static int apply(Auswerter f, Arith e) {
    if (e instanceof Var) {return f.handleVar((Var)e);}
    if (e instanceof Const) {return f.handleConst((Const)e);}
    Op o = (Op)e;
    return f.handleOp(o, apply(f,o.left), apply(f,o.right));
  }
```

Aufgabe 5 (Haskell): (3 + 6 + 4 + 7 = 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 True x = f True (f False x)
f _    _ = 3

g x y = [] ++ g 5 x
```

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

```
i :: Int
i = ((\g x -> g x + g x) (\y -> y)) 3

j :: [Int]
j = filter (\x -> any (\y -> y*y == x) [1..x]) [1..20]
```

Hinweise:

- Die Funktion `any :: (a -> Bool) -> [a] -> Bool` testet, ob mindestens ein Element der als zweites Argument übergebenen Liste das als erstes Argument übergebene Prädikat erfüllt. Es gilt also beispielsweise `any even [1,3,5] == False` und `any even [1,2,3] == True`.
- `[1..20]` berechnet die Liste `[1,2,3,...,19,20]`.

c) Implementieren Sie die Funktion `containsAll :: [a] -> [a] -> Bool` in Haskell. Dabei wird `containsAll l1 l2` genau dann zu `True` ausgewertet, wenn die Liste `l2` alle Elemente der Liste `l1` enthält.

Hinweise:

- Sie können die vordefinierte Funktion `elem :: a -> [a] -> Bool` in Ihrer Lösung verwenden. Diese testet, ob ihr erstes Argument in der als zweites Argument übergebenen Liste enthalten ist. Der Ausdruck `elem x l` wird also genau dann zu `True` ausgewertet, wenn `x` in `l` enthalten ist.

d) Implementieren Sie die Funktion `choose :: [a] -> Int -> [[a]]` in Haskell, sodass `choose l n` alle Möglichkeiten berechnet, aus der als erstes Argument übergebenen Liste `l` die durch das zweite Argument `n` spezifizierte Anzahl an Elementen auszuwählen. Die Reihenfolge, in der die Elemente von `l` in den Ergebnislisten stehen, ist unerheblich. Jedes Element der Liste `l` kann aber höchstens einmal ausgewählt werden. Wenn die Liste `l` weniger als `n` Elemente enthält, dann soll `[]` als Ergebnis zurückgeliefert werden. Der Ausdruck `choose [1,2,3] 2` könnte also beispielsweise zu `[[1,2],[1,3],[2,3]]` oder `[[2,1],[3,1],[3,2]]` ausgewertet werden.

Hinweise:

- Beachten Sie, dass es für jede Liste `l` genau eine Möglichkeit gibt, 0 Elemente auszuwählen.

Lösung:

```
f :: Bool -> Int -> Int
f True x = f True (f False x)
f _ _ = 3

g :: Int -> Int -> [a]
g x y = [] ++ g 5 x

--i = 6
```

```
--j = [1,4,9,16]

containsAll :: [a] -> [a] -> Bool
containsAll [] _ = True
containsAll (x:xs) ys = (elem x ys) && (containsAll xs ys)

containsAll' :: [a] -> [a] -> Bool
containsAll' xs ys = foldr (\x res -> elem x ys && res) True xs

containsAll'' :: [a] -> [a] -> Bool
containsAll'' xs ys = all (\x -> elem x ys) xs

choose :: [a] -> Int -> [[a]]
choose _ 0 = [[]]
choose [] _ = []
choose (x:xs) n = (map (\ys -> x:ys) (choose xs (n-1))) ++ (choose xs n)

choose' :: [a] -> Int -> [[a]]
choose' xs n = filter (\x -> length x == n) (subsequences xs)
```

Aufgabe 6 (Prolog): (2 + 6 + 2 + 7 + 3 = 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) $g(X, Y, s(X)), g(Z, Z, Z)$

ii) $f(X, Y, s(Y)), f(Z, a, Z)$

b) Gegeben sei folgendes Prolog-Programm P.

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

Erstellen Sie für das Programm P den Beweisbaum zur Anfrage "`?- p(b,A,a).`" bis zur Höhe 3 (die Wurzel hat dabei die Höhe 1). Markieren Sie Pfade, die zu einer unendlichen Auswertung führen, mit ∞ und geben Sie alle Antwortsubstitutionen zur Anfrage "`?- p(b,A,a).`" an, die im Beweisbaum bis zur Höhe 3 enthalten sind. Geben Sie außerdem zu jeder dieser Antwortsubstitutionen an, ob sie von Prolog gefunden wird. Geben Sie schließlich noch eine Reihenfolge der Programmklauseln an, bei der Prolog jede Antwortsubstitution zur obigen Anfrage findet.

c) Implementieren Sie ein Prädikat `last` mit Stelligkeit 2 in Prolog, wobei `last(`t_1`,`t_2`)` genau dann gilt, wenn t_1 eine Liste ist und t_2 das letzte Element dieser Liste. Alle Anfragen der Form `last(`t_1`,`t_2`)`, bei denen t_1 keine Variablen enthält, sollen terminieren.

d) Implementieren Sie ein Prädikat `perm` mit Stelligkeit 2 in Prolog, wobei `perm(`t_1`,`t_2`)` genau dann für eine Liste t_1 gilt, wenn t_2 eine Permutation von t_1 ist, d. h. eine Liste mit genau den gleichen Elementen in der gleichen Anzahl, wobei die Reihenfolge der Elemente unterschiedlich sein darf (aber nicht muss). Beispielsweise gibt es 6 Permutationen der Liste `[1,2,3]`, nämlich `[1,2,3]`, `[1,3,2]`, `[2,1,3]`, `[2,3,1]`, `[3,1,2]` und `[3,2,1]`. Alle Anfragen der Form `perm(`t_1`,`t_2`)`, bei denen t_1 und t_2 keine Variablen enthalten, sollen terminieren.

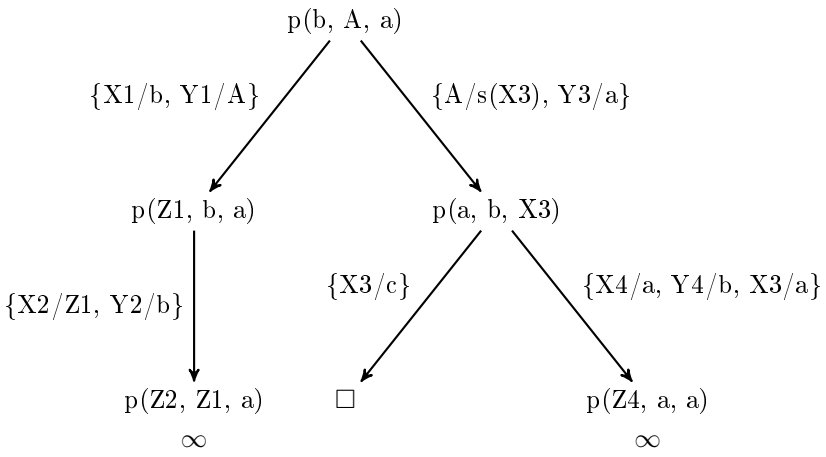
e) Implementieren Sie ein Prädikat `nat` mit Stelligkeit 1 in Prolog, sodass die Anfrage `nat(X)` durch mehrfache Eingabe von `;` nacheinander alle natürlichen Zahlen in aufsteigender Reihenfolge als Belegung für `X` ausgibt (also `X = 0; X = 1; X = 2;` ...).

Lösung:

a) i) $g(X, Y, s(X)), g(Z, Z, Z)$: occur failure X in $s(X)$

ii) $f(X, Y, s(Y)), f(Z, a, Z)$: $X/s(a), Y/a, Z/s(a)$

b)



Die (einzige) Antwortsubstitution innerhalb des Beweisbaums ist $\{A/s(c)\}$. Diese wird von Prolog nicht gefunden.

Mit der folgenden Reihenfolge der Klauseln findet Prolog alle Antwortsubstitutionen:

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

c)
```
last([X],X).
last([_|XS],Y) :- last(XS,Y).
```

d)
```
perm([],[]).
perm([X|XS],ZS) :- perm(XS,YS), insert(X,YS,ZS).

insert(X,XS,[X|XS]).
insert(X,[Y|YS],[Y|ZS]) :- insert(X,YS,ZS).
```

e)
```
nat(0).
nat(X) :- nat(Y), X is Y + 1.
```