Prof. Dr. J. Giesl
J.-C. Kassing, M. Leven Rosarius, N. Lommen, E. Meyer

Allgemeine Hinweise:

- Die **Deadline** zur **Abgabe** der Hausaufgaben ist am **Donnerstag, den 18.12.2025, um 14 Uhr**.
- Der **Workflow** sieht wie folgt aus. Die Abgabe der Hausaufgaben erfolgt **im Moodle-Lernraum** und kann nur in **Zweiergruppen** stattfinden. Dabei müssen die Abgabepartner*innen **dasselbe Tutorium** besuchen. Nutzen Sie ggf. das entsprechende **Forum** im Moodle-Lernraum, um eine*n Abgabepartner*in zu finden. Es darf **nur ein*e** Abgabepartner*in die Abgabe hochladen. Diese*r muss sowohl die **Lösung** als auch den **Quellcode** der Programmieraufgaben hochladen. Die Bepunktung wird dann von uns für **beide** Abgabepartner*innen **separat** im Lernraum eingetragen. Die Feedbackdatei ist jedoch nur dort sichtbar, wo die Abgabe hochgeladen wurde und muss innerhalb des Abgabepaars **weitergeleitet** werden.
- Die **Lösung** muss als PDF-Datei hochgeladen werden. Damit die Punkte beiden Abgabepartner*innen zugeordnet werden können, müssen **oben** auf der **ersten Seite** Ihrer Lösung die **Namen**, die **Matrikelnummern** sowie die **Nummer des Tutoriums** von **beiden** Abgabepartner*innen angegeben sein.
- Der **Quellcode** der Programmieraufgaben muss als `.zip`-Datei hochgeladen werden und **zusätzlich** in der PDF-Datei mit Ihrer Lösung enthalten sein, sodass unsere Hiwis ihn mit Feedback versehen können. Auf diesem Blatt muss Ihre Codeabgabe Ihren vollständigen Java-Code in Form von `.java`-Dateien enthalten. Aus dem Lernraum heruntergeladene Klassen dürfen nicht mit abgegeben werden.

  Stellen Sie sicher, dass Ihr Programm von `javac` **in der Version 25 akzeptiert** wird. Generell sollten alle Programme für alle Eingaben terminieren, solange in der Spezifikation (bzw. der Aufgabenstellung) nicht explizit etwas anderes verlangt wird!

Übungsaufgabe 1 (Überblickswissen):

a) In der Vorlesung haben Sie zwei Arten von allgemeinen Listen kennengelernt. Einmal können generische Typen benutzt werden, um eine solche allgemeine Liste zu realisieren. Die zweite Implementierung hingegen benutzt Werte vom Typ `Object`. Was sind die Nachteile dieser Umsetzung?

b) Woraus besteht das Collection-Framework?[1] Warum ist es in den meisten Fällen sinnvoll, ein solches Framework zu verwenden und nicht eigene Implementierungen?

[1] `https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/doc-files/coll-index.html`

Übungsaufgabe 2 (Collections, Exceptions, Generics und Module):

Im Film "Monty Python and the Holy Grail"[2][3] wird an verschiedenen Stellen über Schwalben, ihre Fluggeschwindigkeit und die Frage diskutiert, ob sie als Zugvögel Kokosnüsse aus Afrika nach England gebracht haben könnten. In dieser Aufgabe werden wir uns damit beschäftigen, wie man einige dieser Zusammenhänge in Java darstellen könnte.

Hinweise:

- Falls Sie zur Laufzeit Fehler mit der Beschreibung `java.lang.NoClassDefFoundError` erhalten, prüfen Sie, ob alle Klassen compiliert wurden und im richtigen Pfad liegen.

a) Schreiben Sie eine Klasse `Nut` in dem Paket `cargo`, um eine Nuss (zum Beispiel eine Kokosnuss) zu repräsentieren. Eine Nuss hat die Attribute `name` vom Typ `String` und `weight` vom Typ `int`. Beide Werte sollen auf einen beliebigen Wert außer `null` initialisiert werden. Schreiben Sie außerdem Selektoren zum Lesen und Schreiben beider Werte.

b) Schreiben Sie eine abstrakte Klasse `Swallow` im Paket `bird`, um eine allgemeine Schwalbe darzustellen. Jede Schwalbe kann ein Objekt vom Typ `Object` als Fracht tragen. Diese Fracht wird dem Konstruktor übergeben und kann mit der öffentlichen Methode `getCargo` abgefragt werden. Außerdem hat jede Schwalbe eine Methode `isLadden`, die `true` zurück gibt, falls die Fracht nicht `null` ist, und sonst `false`. Schließlich gibt es noch eine abstrakte Methode `getAirspeedVelocity`,[4] die ein `int` zurückliefert und nur im gleichen Paket und aus Unterklassen genutzt werden darf.

c) Schreiben sie zwei Unterklassen der Klasse `Swallow`, nämlich `EuropeanSwallow` und `AfricanSwallow` im Paket `bird.swallows` mit entsprechenden Konstruktoren.

Die Airspeed Velocity einer europäischen Schwalbe ist $11\frac{m}{s}$, die einer afrikanischen Schwalbe ist $12\frac{m}{s}$. Die entsprechende Methode liefert nur den Betrag zurück, die Einheit können Sie ignorieren. Falls die Schwalbe mit einer `Nut` beladen ist, dann reduziert sich die Geschwindigkeit jedoch um den Betrag des Gewichts der Nuss. Falls sich dadurch ein negativer Wert ergeben würde, wird die Geschwindigkeit stattdessen auf 0 gesetzt. Eine europäische Schwalbe, die mit einer Nuss mit Gewicht 5 beladen ist, hätte also nur eine Airspeed Velocity von 6. Falls die Schwalbe mit einer Fracht beladen ist, die keine Nuss ist, so halbiert sich die Geschwindigkeit, wobei nach unten abgerundet wird.
Schreiben Sie hierzu eine geschützte Hilfsmethode in der Klasse `Swallow`, die gleiche Teile der beiden Implementierungen von `getAirspeedVelocity` enthält.

Schreiben Sie außerdem je eine statische Methode `createAfricanSwallow` und `createEuropeanSwallow` in der Klasse `Swallow`, die jeweils eine Fracht `cargo` vom Typ `Object` übergeben bekommen und ein `Swallow`-Objekt zurück geben. Die Methode soll ein neues Objekt der entsprechenden Klasse mit der Fracht `cargo` erstellen und zurück geben.

d) Schwalben leben und wandern in Schwärmen. Schreiben Sie im Paket `bird` eine generische Klasse `Flock` mit dem Typ-Parameter `S`, um einen Schwarm Schwalben zu repräsentieren. Ein Schwarm kann aber nicht aus anderen Dingen als aus Schwalben bestehen. Ein Schwarm besteht aus einer `List` von Schwalben. Diese Liste darf nur Objekte vom Typ `S` enthalten. Ein Schwarm afrikanischer Schwalben kann also keine europäischen Schwalben enthalten. Ein Schwarm allgemeiner Schwalben kann jedoch beide Schwalbenarten enthalten.

Erstellen Sie einen Konstruktor ohne Parameter sowie eine Methode `join`, die eine Schwalbe passenden Typs als Parameter übergeben bekommt und zum Schwarm hinzufügt. Die Klasse `java.util.ArrayList` ist eine Implementierung des Interfaces `List`, die sich für diese Aufgabe anbietet.

Ein Schwarm hat eine Methode `double getAverageCruiseAirspeedVelocity()`, um die durchschnittliche Airspeed Velocity zu berechnen. Die hinteren Schwalben fliegen im Windschatten der vorderen, daher müssen sie nicht gegen eventuellen Gegenwind anfliegen. Ziehen Sie also von der Airspeed Velocity

[2] Deutscher Titel: "Die Ritter der Kokosnuß" *(noch in alter Rechtschreibung)*
[3] Es ist keine Voraussetzung, den Film gesehen zu haben, um die Aufgabe bearbeiten zu können. Wir empfehlen jedoch, sich den Film anzuschauen (besonders, wenn man Komödien mag).
[4] Airspeed ist die Fluggeschwindigkeit relativ zur umgebenden Luft. Diese Größe ist wichtiger als die Geschwindigkeit über Grund, da sie maßgeblich für den Auftrieb ist.

der ersten Schwalbe 2 ab, ziehen Sie bei der zweiten Schwalbe 1 ab und berechnen Sie erst dann den Durchschnitt.

Schreiben Sie ein Interface `FlockInterface`, das von `Flock` implementiert wird und nur die öffentliche Methode `getAverageCruiseAirspeedVelocity` besitzt. Dieses Interface dient als gemeinsame Oberklasse aller generischen Varianten von `Flock`, mit allen wichtigen, vom Typparameter unabhängigen Methoden, und kann statt des Raw-Typs verwendet werden.[5]

e) Erstellen Sie eine Exception-Klasse `UnspecificQuestionException` im Paket `people`. Es soll sich um eine *checked* Exception handeln.

f) Erstellen Sie im Paket `people` eine Klasse `Troll`. Ein Troll kann verwirrt sein oder nicht und hat daher ein Attribut `confused`, das auf `false` initialisiert wird. Verwirrtheit sieht man einem Troll aber nicht an, daher gibt es für dieses Attribut keine öffentlichen Selektoren.

Ein Troll hat eine Methode `pass`, mit der man versuchen kann, an ihm vorbei zu gehen. Das funktioniert nur, wenn er verwirrt ist. Dann passiert in dieser Methode nichts. Andernfalls beendet der Troll das Programm mit Hilfe der Methode `java.lang.System.exit(-1)`.

Ein Troll hat eine Methode `askAboutAirspeedVelocity(Object)`. Falls es sich bei dem Objekt um ein Objekt vom Typ `FlockInterface` handelt, gibt die Methode einfach die durchschnittliche Airspeed Velocity zurück. Ist es ein `Swallow`-Objekt, erstellt der Troll einen Schwarm allgemeiner Schwalben vom Typ `Swallow` mit genau diesem einen Schwarmmitglied, nimmt die durschnittliche Airspeed Velocity, addiert 2 (um den Gegenwind herauszurechnen) und gibt das Ergebnis zurück.

Für alle anderen Objekte gibt die Methode 0 zurück, mit einer Ausnahme: Ist es ein String, der den Wert `"Swallow"`, `"Unladden Swallow"`, `"European Swallow"` oder `"African Swallow"` hat, so ist der Troll verwirrt und wirft eine `UnspecificQuestionException`.

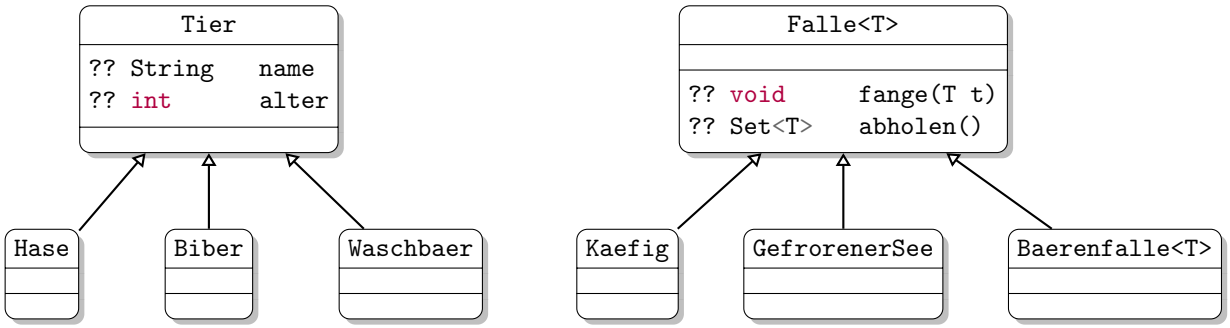
g) Teilen Sie ihr Projekt in drei Module auf. Das Modul `cargo` enthält nur das Paket `cargo`. Das Modul `bird` enthält die Pakete `bird` und `bird.swallows`, stellt aber nur das erste Paket für andere Module zur Verfügung. Das Modul `people` enthält nur das Paket `people`.

h) Erweitern Sie die `main`-Methode in der Klasse `people.KingArthur` um einen Abschnitt, in dem der Troll eine Frage gestellt bekommt, die er nicht beantworten kann. Fangen Sie die auftretende Exception ab und passieren sie anschließend den Troll.

[5] Dies führt zu einem besseren Programmierstil als `instanceof` mit dem Raw-Typ zu verwenden.

Hausaufgabe 3 (Collections, Exceptions und Generics): (5 + 2 + 3 + 6 + 7 + 5 = 28 Punkte)

In dieser Aufgabe modellieren wir eine vereinfachte Version der Welt des Films "Hundreds of Beavers",[6] indem wir Tiere und Fallen mithilfe von Vererbung, abstrakten Klassen, Generics und `Java`-Collections abbilden. Die verschiedenen Klassen und ihre Vererbungshierarchie werden durch das folgende Klassendiagramm dargestellt:



In dieser Klassenhierarchie wurden einige Details ausgelassen. Für die ganze Aufgabe gilt:

- (Pflicht) Nutzen Sie an geeigneten Stellen die folgenden Modifizierer für Methoden und Attribute: `static`, `private`, `protected`, `public`, `final` und `abstract` (bei Methoden).
- (Pflicht) Nutzen Sie an geeigneten Stellen Modifizierer für Klassen wie `abstract`, `sealed` und `final`. Entscheiden Sie, ob es sich jeweils um Klassen oder Interfaces handeln soll.
- (Pflicht) Wenn ein Typ Typparameter erwartet, so nutzen Sie ihn nicht ohne Typparameter, d.h. verwenden Sie keine Raw-Typen.
- (Optional) Nutzen Sie an geeigneten Stellen Annotationen wie `@Override`.

Hinweise:

- Sie dürfen in der gesamten Aufagbe beliebige Klassen aus dem `Java`-Collection Framework verwenden und ihre bereitgestellten Methoden, siehe `https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/doc-files/coll-index.html`.

a) Implementieren Sie die dargestellte Vererbungshierarchie für die verschiedene Tierarten. Jedes Tier hat einen Namen vom Typ `String` und ein Alter vom Typ `int`. Es soll nicht möglich sein, Objekte nur der Klasse `Tier` zu erstellen. Schreiben Sie für jede der vier Klassen einen Konstruktor, der die beiden Attribute initialisiert. Dieser bekommt also als Parameter einen `String` und einen `int`-Wert und setzt dann die Attribute auf die übergebenen Parameter-Werte. Außerdem soll es entsprechende Get-Methoden geben, sowie eine sinnvolle `toString()`-Methode, die die Tierart, den Namen und das Alter ausgibt. Versiegeln Sie die Vererbungshierarchie so, dass es keine weiteren Unterklassen von den vier implementierten Klassen geben kann.

b) Implementieren Sie die Klasse `LeereFalleException`, welche von der Klasse `RuntimeException` erben soll.

c) Implementieren Sie die Klasse `Falle<T>`[7] für beliebige Fallen, die Tiere vom Typ `T` fangen können. Auch hier soll es nicht möglich sein, Objekte nur der Klasse `Falle<T>` zu erstellen. Fallen stellen spezielle Container dar, auf die mittels der zwei folgenden Methoden zugegriffen werden kann:

- `void fange(T t)`, welche das Objekt `t` vom Typ `T` in die Falle einfügt. Die genaue Art, wie dies geschieht, hängt von der konkreten Art der Falle ab.

[6] Es ist keine Voraussetzung, den Film gesehen zu haben, um die Aufgabe bearbeiten zu können. Wir empfehlen jedoch, sich den Film anzuschauen (besonders, wenn man Komödien oder Biber mag). Siehe: `https://www.hundredsofbeavers.com/`

[7] Um noch konkreter zu sein, könnte man auch `Falle<T extends Tier>` verwenden. Dies sagt dem Compiler, dass jede Instantiierung des generischen Typs `T` die Klasse `Tier` oder eine ihrer Unterklassen sein muss.

- `Set<T> abholen()`, welche alle Objekte vom Typ `T` in einer Menge zurückgibt, die momentan in der Falle gespeichert sind. Die Methode soll eine `LeereFalleException` werfen, wenn die Methode aufgerufen wird, obwohl kein Tier in der Falle gefangen wurde. Nach dem Aufruf der Methode soll die Falle wieder leer sein. Wie das `abholen` genau funktioniert, hängt wieder von der konkreten Art der Falle ab.

d) Implementieren Sie die Klasse `Baerenfalle<T>`. Die `Baerenfalle<T>` ist eine Falle, die nur ein einziges Objekt vom Typ `T` speichern kann. Falls die `fange`-Methode aufgerufen wird, obwohl bereits ein Tier gespeichert ist, passiert nichts. Die `abholen`-Methode wirft entweder eine `LeereFalleException` (wenn kein Tier gefangen wurde) oder gibt ein `Set` mit genau einem Element zurück.

e) Implementieren Sie die Klasse `Kaefig`. Die Klasse `Kaefig` ist nicht mehr generisch und nimmt also immer Tiere vom Typ `Tier` (oder Unterklassen) auf. Das können wir darstellen, indem `Kaefig` von der Klasse `Falle<Tier>` erbt, in der wir den generischen Parameter auf eine feste Klasse setzen. Der Käfig verhält sich wie ein unbegrenzter Container für alle Tierarten. Die `fange`-Methode fügt ein neues Tier ein und die `abholen`-Methode gibt das gesamte `Set` der gespeicherten Tiere zurück. Falls sich allerdings ein Waschbär im Käfig befindet, dann frisst dieser alle Hasen, sobald die `abholen`-Methode aufgerufen wird, und es werden entsprechend keine Hasen im `Set` mit zurückgegeben. Falls die Falle leer ist, wirft `abholen` wieder eine `LeereFalleException`.

f) Implementieren Sie die Klasse `GefrorenerSee`. Die Klasse `GefrorenerSee` ist auch nicht generisch und nimmt nur Objekte vom Typ `Tier` auf. Die Klasse `GefrorenerSee` verhält sich ähnlich zu dem Käfig, allerdings schaffen es die Waschbären hier nicht, die Hasen zu fressen. Alle Tiere, die einmal auf dem See gefangen sind, können sich aufgrund des Eises nicht mehr bewegen und kommen auch nicht an die anderen Tiere heran. Die `fange`-Methode fügt also ein neues Tier ein und die `abholen`-Methode gibt das gesamte `Set` der gespeicherten Tiere zurück. Falls die Falle leer ist, wirft `abholen` wieder eine `LeereFalleException`.

Sie können ihre Implementierung mit der Klasse `Jean` testen. Die Ausführung der `main`-Methode sollte die folgende Ausgabe produzieren, wobei vor allem die Anzahl und die Tiere, welche ausgegeben werden, relevant sind (die genaue Ausgabe kommt hier auf die `toString()`-Implementierung der verschiedenen Tiere an und die Reihenfolge, die bei der `abholen()`-Methode erstellt wird):

```
Dann schauen wir doch mal nach was wir alles gefangen haben.
Bei 100 Bibern erhalte ich endlich meinen Hauptgewinn!

Baerenfalle:
[Hase Fluffy (Alter: 5)]

GefrorenerSee:
[Biber Sherlock Holmes (Alter: 14), Hase Fluffy (Alter: 5), Waschbaer Raccoonster (Alter: 4)]

Kaefig:
[Biber Sherlock Holmes (Alter: 14), Waschbaer Raccoonster (Alter: 4)]

Leider ist die Falle Nummer 4 leer...
```

Übungsaufgabe 4 (Auswertungsstrategie):

Gegeben sei das folgende Haskell-Programm:

```
descending :: Int -> [Int]
descending 0 = []
descending n = n : descending (n-1)

listProd :: [Int] -> Int
listProd [] = 1
listProd (x:xs) = x * listProd xs

listSum :: [Int] -> Int
listSum xs = listSum' xs 0
    where listSum' []     a = a
          listSum' (x:xs) a = listSum' xs (a+x)
```

Die Funktion `descending` berechnet die absteigende Liste der natürlichen Zahlen bis hinunter zu `1`. Zum Beispiel berechnet `descending 5` die Liste `[5,4,3,2,1]`. Die Funktion `listProd` multipliziert die Elemente einer Liste, beispielsweise ergibt `listProd [3,5,2,1]` die Zahl `30`. Die Funktion `listSum` addiert die Elemente einer Liste. Zum Beispiel liefert `listSum [5,2,7]` den Wert `14`.
Geben Sie alle Zwischenschritte bei der Auswertung der folgenden Ausdrücke an:

1. `listProd (descending (1+1))`
2. `listSum (descending (1+1))`

Hinweise:

- Beachten Sie, dass Haskell eine Leftmost-Outermost Auswertungsstrategie besitzt. Allerdings sind Operatoren wie `*` und `+`, die auf eingebauten Zahlen arbeiten, strikt, d.h. hier müssen vor Anwendung des Operators seine Argumente vollständig ausgewertet worden sein (wobei zunächst das linke Argument ausgewertet wird).
- Bedenken Sie, dass duplizierte Ausdrücke von Haskell "simultan" ausgewertet werden.

Hausaufgabe 5 (Auswertungsstrategie): (33 Punkte)

Gegeben sei das folgende Haskell-Programm:

```
second :: [Int] -> Int
second (x:y:xs) = y
second xs       = 0

naturalsFromTo :: Int -> Int -> [Int]
naturalsFromTo n m = if n <= m then n:naturalsFromTo (n+1) m else []

append :: [Int] -> [Int] -> [Int]
append []     ys = ys
append (x:xs) ys = x:append xs ys

doubleList :: [Int] -> [Int]
doubleList xs = append xs xs
```

Die Funktion `second` gibt das zweite Element einer Liste zurück oder `0` falls kein solches existiert. Die Funktion `naturalsFromTo` berechnet eine aufsteigende Liste aller natürlichen Zahlen, welche nicht kleiner sind als das erste Argument und nicht größer als das zweite. Die Funktion `append` konkateniert zwei Listen, während die Funktion `doubleListe` die Konkatenation einer Liste mit sich selbst berechnet.
Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

```
second (doubleList (naturalsFromTo 1 2))
```

an. Unterstreichen Sie vor jedem Auswertungsschritt den Teil des Ausdrucks, der als Nächstes an seiner äußersten Position ausgewertet wird. Schreiben Sie hierbei (um Platz zu sparen) `s`, `nFT`, `a` und `dL` statt `second`, `naturalsFromTo`, `append` und `doubleList`.

Hinweise:

- Beachten Sie die Hinweise zur Übungsaufgabe.
- Falls eine Funktion mit mehreren Argumenten aufgerufen wird, dürfen Sie diese direkt ohne Zwischenschritte einsetzen. Zum Beispiel ist der folgende Auswertungsschritt gültig:

  `nFT 1 3` → `if 1 <= 3 then 1:nFT (1+1) 3 else []`

Übungsaufgabe 6 (Listen in Haskell):

Seien `x`, `y` und `z` ganze Zahlen vom Typ `Int` und seien `xs` und `ys` Listen der Längen n und m vom Typ `[Int]`. Welche der folgenden Gleichungen zwischen Listen sind richtig und welche nicht? Begründen Sie Ihre Antwort. Falls es sich um syntaktisch korrekte Ausdrücke handelt, geben Sie für jede linke und rechte Seite auch an, wieviele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.
Beispiel: Die Liste `[[1,2,3],[4,5]]` hat den Typ `[[Int]]` und enthält 2 Elemente.

Hinweise:

- Hierbei steht `++` für den Verkettungsoperator für Listen. Das Resultat von `xs ++ ys` ist die Liste, die entsteht, wenn die Elemente aus `ys` — in der Reihenfolge wie sie in `ys` stehen — an das Ende von `xs` angefügt werden.
  Beispiel: `[1,2] ++ [1,2,3]` = `[1,2,1,2,3]`
- Falls linke und rechte Seite gleich sind, genügt **eine** Angabe des Typs und der Elementzahl.

a) `[x : [y] : []]` = `[[x] ++ [y]]`

b) `[x, y] ++ xs` = `[x] ++ [y] ++ xs`

c) `[x, y, z]` = `([x] ++ [y]) : [[z]]`

d) `(x:[]):[]` = `[x:[]] ++ [[]]`

e) `x:y:z:xs` = `(x:[y]) ++ (z:xs)`

Hausaufgabe 7 (Listen in Haskell): (4 + 5 + 5 = 14 Punkte)

Seien `x`, `y` und `z` ganze Zahlen vom Typ `Int` und seien `xs` und `ys` Listen der Längen n und m vom Typ `[Int]`. Welche der folgenden Gleichungen zwischen Listen sind richtig und welche nicht? Begründen Sie Ihre Antwort. Falls es sich um syntaktisch korrekte Ausdrücke handelt, geben Sie für jede linke und rechte Seite auch an, wie viele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.
Beispiel: Die Liste `[[x,y],ys]` hat den Typ `[[Int]]` und enthält 2 Elemente.

1. `((x:[ys]) : []) : [] = (([x] ++ ys) : [] : []) ++ [[]]`
2. `(x:[y]):[xs] = [[x] ++ [y]] ++ [xs]`
3. `((x:[y]):[[z]++ys]):[] = x:y:z:ys`

Hinweise:

- Beachten Sie die Hinweise zur Übungsaufgabe.