Prof. Dr. J. Giesl D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Allgemeine Hinweise:

- Die **Deadline** zur **Abgabe** der Hausaufgaben ist am **Donnerstag, den 23.12.2021, um 18:00 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 müssen Sie in Haskell programmieren und `.hs`-Dateien anlegen. Stellen Sie sicher, dass Ihr Programm mit `GHCi` **ausgeführt werden kann**, ansonsten werden keine Punkte vergeben.

Tutoraufgabe 1 (Überblickswissen):

a) In Haskell sind Funktionen gleichberechtigte Datenobjekte. Was bedeutet das? Welche Vorteile bietet es?

b) Typdeklarationen sind in Haskell nicht notwendig. Welche Gründe gibt es, trotzdem eine anzugeben?

c) In der Mathematik sind wir es gewohnt, eine Funktion mit ihrem Definitions- und Zielbereich anzugeben, zum Beispiel $sqrt : \mathbb{N} \to \mathbb{R}$. Die Typdeklaration einer Funktion in Haskell dagegen kann mehrere solcher Pfeile enthalten, zum Beispiel `isSquare :: Int -> Int -> Bool`. Was bedeutet das und welche Vorteile bietet es gegenüber der Typdeklaration `isSquare :: (Int,Int) -> Bool`?

Tutoraufgabe 2 (Auswertungsstrategie):

Gegeben sei das folgende Haskell-Programm:

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

doubleEach :: [Int] -> [Int]
doubleEach []     = []
doubleEach (x:xs) = x * 2 : (doubleEach xs)

repeat :: Int -> Int -> [Int]
repeat x n = if n > 0 then x : (repeat x (n-1)) else []
```

Die Funktion `second` gibt zu jeder Eingabeliste mit mindestens zwei Elementen den zweiten Eintrag der Liste zurück. Für Listen mit weniger als zwei Elementen wird `0` zurückgegeben. Die Funktion `doubleEach` gibt die Liste zurück, die durch Verdoppeln jedes Elements aus der Eingabeliste entsteht. Der Aufruf `doubleEach [1, 2, 3, 1]` würde also `[2, 4, 6, 2]` ergeben. Die Funktion `repeat` erzeugt eine Liste, die das erste Argument

so oft enthält, wie das zweite Argument angibt. Beispielsweise erhält man beim Aufruf `repeat 5 3` die Liste `[5, 5, 5]` als Rückgabe.
Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

```
repeat (second (doubleEach [2, 3, 5])) (second [3, 1, 4])
```

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`, `d` und `r` statt `second`, `doubleEach` und `repeat`.

Hinweise:

- Beachten Sie, dass Haskell eine Leftmost-Outermost Lazy 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).
- `_` bezeichnet den sogenannten Joker-Pattern, der für einen beliebigen Wert steht. Statt des Joker-Patterns könnte man also auch eine neue Variable benutzen. Die zweite definierende Gleichung von `second` könnte also auch wie folgt lauten: `second (y:[]) = 0`.

Aufgabe 3 (Auswertungsstrategie): (26 Punkte)

Gegeben sei das folgende Haskell-Programm:

```
doubleElements :: [Int] -> [Int]
doubleElements [] = []
doubleElements (x:xs) = x : x : doubleElements xs

firstN :: Int -> [Int] -> [Int]
firstN n [] = []
firstN n (x:xs) =
  if n > 0 then x : firstN (n-1) xs else []

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

Die Funktion `doubleElements` berechnet eine Liste, welche aus einer gegebenen Liste hervorgeht, indem jedes Element dieser Liste noch einmal hinter sich selbst eingefügt wird. Zum Beispiel wertet der Ausdruck `doubleElements [1,2,3]` zu `[1,1,2,2,3,3]` aus. Die Funktion `firstN` gibt zu jeder Liste `xs` den längsten Anfang `ys` von `xs` zurück, sodass `ys` höchstens eine gegebene Anzahl an Elementen besitzt. Der Ausdruck `firstN 2 [1,2,3]` wertet beispielsweise zu `[1,2]` aus, während hingegen `firstN 5 [1,2,3]` zu `[1,2,3]` auswertet. Die Funktion `listSum` bildet die Summe über alle Elemente einer Liste vom Typ `[Int]`. Wird `listSum [1,2,3]` aufgerufen, so ergibt sich `6`.
Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

```
listSum (firstN (1+0) (doubleElements [1+0, 1+0]))
```

an. Unterstreichen Sie vor jedem Auswertungsschritt den Teil des Ausdrucks, der als Nächstes an seiner äußersten Position ausgewertet wird. Um Platz zu sparen können Sie hierbei `dE`, `fN` und `lS` statt `doubleElements`, `firstN` und `listSum` schreiben.

Hinweise:

- Beachten Sie, dass Haskell eine Leftmost-Outermost Lazy 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).

Tutoraufgabe 4 (Listen in Haskell):

Seien `x`, `y` 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 `[[1,2,3],[4,5]]` hat den Typ `[[Int]]` und enthält 2 Elemente.

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

b) `x:[y]` = `x:y`

c) `x:ys:xs` = `(x:ys) ++ xs`

d) `[x,x,y] ++ (x:xs)` = `x:x:((y:[x]) ++ xs)`

e) `[]:[[[1]],[]]` = `[[],[1]]:[[]]`

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.

Tutoraufgabe 5 (Listen in Haskell (Video)):

Seien `x`, `y`, `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],[z]]` 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) `[] ++ [xs]` = `[] : [xs]`

b) `[[]] ++ [x]` = `[] : [x]`

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

d) `x:y:z:(xs ++ ys)` = `[x,y,z] ++ xs ++ ys`

e) `[(x:xs):[ys],[[]]]` = `(([]:[]):[]) ++ ([([x] ++ xs),ys]:[])`

Aufgabe 6 (Listen in Haskell): (4+5+5+5+5 = 24 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.

a) `((x:[ys]) : []) : []` = `(([x] ++ ys) : [] : []) ++ [[]]`

b) `(x:[y]):[xs]` = `[[x] ++ [y]] ++ [xs]`

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

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

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

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.

Tutoraufgabe 7 (Programmieren in Haskell):

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Verwenden Sie außer den Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise), `True` und `False`, Werten des Typs `Int`, der Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., booleschen Funktionen wie `&&`, `||`, `not` und arithmetischen Operatoren wie `+`, `*` **keine** vordefinierten Funktionen außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden. Schreiben Sie ggf. Hilfsfunktionen, um sich die Lösung der Aufgaben zu vereinfachen.

a) `fib n`
Berechnet die `n`-te Fibonacci–Zahl. Auf negativen Eingaben darf sich die Funktion beliebig verhalten.

Die Auswertung von `fib 17` liefert bspw. den Ausgabewert `1597`.

Hinweise:

- Die Fibonacci–Zahlen sind durch die rekursive Folge mit den Werten $a_0 = 0$, $a_1 = 1$ und $a_n = a_{n-2} + a_{n-1}$ für $n \geq 2$ beschrieben.

b) `prime n`
Gibt genau dann `True` zurück, wenn die natürliche Zahl `n` eine Primzahl ist. Auf negativen Eingaben darf sich die Funktion beliebig verhalten.

Die Auswertung von `prime 35897` liefert bspw. den Ausgabewert `True`.

Hinweise:

- Sie dürfen die vordefinierte Funktion `rem x y` verwenden, die den Rest der Division `x` / `y` zurückgibt.

c) `nthSmallestPerfectNumber n`
Diese Funktion soll die `n`. kleinste vollkommene (engl. perfect) Zahl berechnen. Bei nicht-positiver Eingabe `n` darf sich die Funktion beliebig verhalten. Eine natürliche Zahl heißt vollkommen, wenn diese die Summe aller ihrer echten Teiler ist, d.h. aller Teiler außer sich selbst. Die kleinsten vollkommenen Zahlen sind $6 = 1 + 2 + 3$, $28 = 1 + 2 + 4 + 7 + 14$ und $496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248$.

Entsprechend sollen die Ausdrücke `nthSmallestPerfectNumber 1`, `nthSmallestPerfectNumber 2` und `nthSmallestPerfectNumber 3` zu 6, 28 und 496 auswerten.

d) `powersOfTwo i0 i1`
Gibt eine `Int`–Liste zurück, die die Zweierpotenzen $2^{\texttt{i0}}$ bis $2^{\texttt{i1}}$ enthält. Falls `i0` $>$ `i1`, soll die leere Liste zurückgegeben werden. Auf negativen Eingaben darf sich die Funktion beliebig verhalten.

Die Auswertung von `powersOfTwo 5 10` liefert bspw. den Ausgabewert `[32,64,128,256,512,1024]`.

Hinweise:

- Sie können die Exponentiation x^y zweier Zahlen `x` und `y` in Haskell mit `x^y` vornehmen.

e) `selectKsmallest k xs`
Gibt das Element zurück, das in der `Int`-Liste `xs` an der Stelle `k` stehen würde, wenn man `xs` aufsteigend sortiert. Hierbei hat das erste Element den Index 1. Wenn `k` kleiner als 1 oder größer als die Länge von `xs` ist, darf sich die Funktion beliebig verhalten.

Die Auswertung von `selectKsmallest 3 [4, 2, 15, -3, 5]` liefert also den Ausgabewert `4` und `selectKsmallest 1 [5, 17, 1, 3, 9]` liefert den Ausgabewert `1`.

Hinweise:

- Sie können die Liste an einem geeigneten Element `x` in zwei Listen teilen, sodass eine der beiden Teillisten nur Elemente enthält, die kleiner oder gleich `x` sind, und die andere Teilliste nur größere Elemente als `x` enthält. Dann können Sie `selectKsmallest` mit geeigneten Parametern rekursiv aufrufen.
- Sie dürfen die vordefinierte Funktion `length` verwenden, wobei `length ys` die Anzahl der Elemente der Liste `ys` zurückgibt.

Tutoraufgabe 8 (Programmieren in Haskell (Video)):

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Sie dürfen die Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise), `True` und `False`, Werte des Typs `Int`, die Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., boolesche Funktionen wie `&&`, `||`, `not` und die arithmetischen Operatoren `+`, `*`, `-` verwenden, aber **keine** vordefinierten Funktionen außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden. Schreiben Sie ggf. Hilfsfunktionen, um sich die Lösung der Aufgaben zu vereinfachen.

a) `fibInit a0 a1 n`
Berechnet die `n`-te Fibonacci–Zahl der Fibonacci-Folge mit den alternativen natürlichen Initialwerten `a0` und `a1`. Auf negativen Eingaben für `a0`, `a1` und `n` darf sich die Funktion beliebig verhalten.

Die Auswertung von `fibInit 1 11 7` liefert bspw. den Rückgabewert `151`.

Hinweise:

- Die Fibonacci–Zahlen mit den Initialwerten `a0` und `a1` sind durch die rekursive Folge mit den Werten $a_0 =$ `a0`, $a_1 =$ `a1` und $a_n = a_{n-2} + a_{n-1}$ für $n \geq 2$ beschrieben.

b) In Aufgabe 7 wurden die Fibonacci–Zahlen mit einer naiven Implementierung berechnet, die schon für $n = 50$ nicht in annehmbarer Zeit zu einem Ergebnis kommt. Das liegt daran, dass für den Aufruf `fib 50` sowohl `fib 49` als auch `fib 48` ausgewertet werden müssen. Für `fib 49` muss aber auch `fib 48` (und `fib 47`) ausgewertet werden. Offensichtlich berechnen wir so dasselbe Ergebnis — `fib 48` — mehrmals. In der Implementierung in dieser Aufgabe soll dieser Mehraufwand umgangen werden.

Dazu gehen wir schrittweise vor: Implementieren Sie zunächst die Funktion `fibInitL a0 a1 n`. Diese berechnet die Liste der nullten, ersten, ..., $(n-1)$-ten, n-ten Fibonacci–Zahl der Fibonacci-Folge mit den alternativen natürlichen Initialwerten `a0` und `a1` auf effiziente Art und Weise. Falls $n = -1$ ist, so liefert `fibInitL a0 a1 (-1)` das Resultat `[]`. Für $n < -1$ darf sich die Funktion beliebig verhalten.

Die Auswertung von `fibInitL 0 1 6` liefert bspw. den Rückgabewert `[0,1,1,2,3,5,8]`.

Implementieren Sie dann die Funktion `fibInit2 a0 a1 n`, die die `n`-te Fibonacci–Zahl der Fibonacci-Folge mit den alternativen natürlichen Initialwerten `a0` und `a1` auf effiziente Art und Weise berechnet. Auf negativen Eingaben für `a0`, `a1` und `n` darf sich die Funktion beliebig verhalten.

Die Auswertung von `fibInit2 1 3 50` liefert bspw. den Rückgabewert `45537549124`.

Hinweise:

- Sie können mit obigem Beispiel überprüfen, ob ihre Implementierung effizient ist.

c) `normalize xs`
Gibt eine `Int`–Liste von derselben Länge wie `xs` zurück, deren kleinster Wert 0 ist. Weiterhin soll die Differenz zwischen zwei benachbarten Zahlen in der Ausgabeliste stets genauso hoch sein wie die Differenz zwischen den beiden Zahlen der Eingabeliste an denselben Positionen.

Die Auswertung von `normalize [15,17,-3,46]` liefert bspw. den Rückgabewert `[18,20,0,49]`.

d) `sumMaxs xs`
Addiert diejenigen Werte der eingegebenen `Int`–Liste `xs` auf, die größer sind als **alle** vorherigen Werte in der Liste.

Die Auswertung von `sumMaxs [2,1,2,5,4]` liefert bspw. den Rückgabewert `7` $(= 2 + 5)$.

e) `sumNonMins xs`
Addiert diejenigen Werte der eingegebenen `Int`–Liste `xs` auf, die größer sind als mindestens **ein** vorheriger Wert in der Liste.

Die Auswertung von `sumNonMins [2,1,2,5,4]` liefert bspw. den Rückgabewert `11` $(= 2 + 5 + 4)$.

f) `primeTwins x`
Gibt den kleinsten Primzahl–Zwilling zurück, dessen beide Elemente größer sind als die `Int`-Zahl `x`.

Die Auswertung von `primeTwins 12` liefert bspw. den Rückgabewert `(17,19)`.

Hinweise:

- Ein Primzahlzwilling ist ein 2-Tupel $(n, n + 2)$, bei dem sowohl n als auch $n + 2$ Primzahlen sind.
- Sie dürfen die Funktion `prime` aus Aufgabe 7 verwenden.

g) `multiples xs i0 i1`
Gibt eine `Int`–Liste zurück, die alle Werte zwischen `i0` und `i1` enthält, die ein Vielfaches einer der Werte aus `xs` sind. Die zurückgegebene Liste soll die Werte in aufsteigender Reihenfolge und jeweils nur einmal enthalten.

Die Auswertung von `multiples [3,5] 5 20` liefert bspw. den Rückgabewert `[5,6,9,10,12,15,18,20]`.

Hinweise:

- Sie dürfen die vordefinierte Funktion `rem x y` verwenden, die den Rest der Division `x` / `y` zurückgibt.

Aufgabe 9 (Programmieren in Haskell): (7+6+10+9+11+7 = 50 Punkte)

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Sie dürfen die Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise), `True` und `False`, Werte des Typs `Int`, die Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., boolesche Funktionen wie `&&`, `||`, `not` und die arithmetischen Operatoren `+`, `*`, `-` verwenden, aber **keine** weiteren vordefinierten Funktionen. Schreiben Sie ggf. Hilfsfunktionen, um sich die Lösung der Aufgaben zu vereinfachen. Sie dürfen jederzeit Hilfsfunktionen aus vorherigen Teilaufgaben verwenden, auch wenn Sie diese nicht selbst implementiert haben.

a) `symmetricDifference xs ys`
Diese Funktion berechnet die symmetrische Differenz zweier Listen `xs::[Int]` und `ys::[Int]`. Die symmetrische Differenz ist eine Liste `zs::[Int]`, welche alle Elemente enthält, die entweder nur in `xs` oder nur in `ys` enthalten sind. Die Reihenfolge der Elemente innerhalb der resultierenden Liste ist hierbei unerheblich. Wenn ein Wert sowohl in `xs` als auch in `ys` vorkommt, darf er nicht in `symmetricDifference xs ys` auftreten, auch wenn er unterschiedlich oft in `xs` und `ys` vorkommt.

Die Auswertung von `symmetricDifference [1,1,2,4,5] [2,2,3,4,6]` kann beispielsweise die Liste `[1,1,5,3,6]::[Int]` liefern.

b) `powerlist xs`
Berechnet eine Liste `ps::[[Int]]` aller Teillisten von `xs::[Int]`. Für jede Liste in `ps` soll die Reihenfolge der Listenelemente dieselbe sein wie die Reihenfolge der entsprechenden Elemente in `xs`. Die Reihenfolge der Elemente von `ps` selbst ist hierbei unerheblich. Mehrfach vorkommende Werte in `xs` können auch in den Teillisten entsprechend oft auftreten. Wenn l die Länge von `xs` bezeichnet, so ist 2^l die Länge von `powerlist xs`.

Die Auswertung von `powerlist [1,2,3]` kann beispielsweise die Liste `[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]` der Länge 8 ergeben. Der Ausdruck `powerlist [2,2]` kann hingegen beispielsweise zur Liste `[[],[2],[2],[2,2]]` ausgewertet werden.

c) `permutations xs`
Diese Funktion soll zu einer Liste `xs::[Int]` eine Liste aller Permutationen von `xs` berechnen. Eine Permutation von `xs` ist hierbei eine Liste `ps::[Int]`, welche dieselben Elemente wie `xs` enthält, jedoch kann die Reihenfolge der Elemente in `ps` von der Reihenfolge der Elemente in `xs` abweichen. Die Reihenfolge der Elemente von `permutations xs` kann hierbei beliebig sein. Wenn `xs` eine Liste der Länge l ist, so ist `permutations xs` von der Länge $l!$. Falls `xs` die leere Liste ist, so darf sich die Funktion beliebig verhalten.

Der Ausdruck `permutations [1,2,3]` könnte damit zu der Liste `[[1,2,3],[2,1,3],[3,2,1],[1,3,2],[3,1,2],[2,3,1]]` vom Typ `[Int]` auswerten.

Die folgenden Teilaufgaben beziehen sich auf eine Datenstruktur, welche Graphen repräsentiert. Mathematisch ist ein Graph ein Tupel $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, wobei $\mathcal{V}$ eine Menge von Knoten (engl. vertices) und $\mathcal{E} \subseteq \mathcal{V}^2$ eine Menge von Kanten (engl. edges) ist, welche die Knoten untereinander verbinden. Die betrachteten Graphen sind gerichtet, d.h., eine Kante (a, b) bedeutet, dass Knoten b von Knoten a erreicht werden kann, jedoch nicht unbedingt auch umgekehrt. Wir nehmen im Folgenden an, dass jeder Knoten mindestens eine eingehende oder ausgehende Kante besitzt.

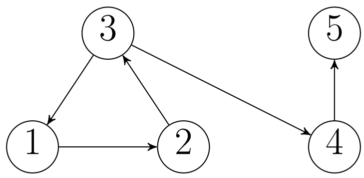

Abbildung 1: Durch die Liste `testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)]::[(Int,Int)]` repräsentierter Graph.

Abb. 1 ist eine graphische Repräsentation des Graphen $(\{1,2,3,4,5\}, \{(1,2),(2,3),(3,1),(4,5),(3,4)\})$. Im Folgenden werden Graphen in Haskell als Liste vom Typ `[(Int,Int)]` ihrer Kanten dargestellt. Der in Abb. 1 abgebildete Graph kann als Liste `testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)]::[(Int,Int)]` seiner Kanten in Haskell dargestellt werden.

d) `nodes es`
Gegeben eine Liste `es::[(Int,Int)]` von Kanten, berechnet diese Funktion eine Liste vom Typ `[Int]` aller im Graph enthaltenen Knoten. Die berechnete Liste soll *keine Duplikate* enthalten. Die Reihenfolge ist irrelevant.

Beispielsweise könnte `nodes testGraph` zu der Liste `[1,2,3,4,5]::[Int]` auswerten.

e) `existsPath es a b`
Diese Funktion berechnet für eine gegebene Liste `es::[(Int,Int)]` von Kanten und zwei Knoten `a,b::Int` einen Wahrheitswert vom Typ `Bool`, der angibt, ob der Knoten `b` vom Knoten `a` aus mithilfe der Kanten aus der Liste `es` erreicht werden kann.

Die Ausdrücke `existsPath testGraph 1 3`, `existsPath testGraph 1 5` und `existsPath testGraph 5 5` berechnen hierbei beispielsweise den Wahrheitswert `True`, während hingegen die Aufrufe `existsPath testGraph 5 1`, `existsPath testGraph 5 4` und `existsPath testGraph 4 3` zu `False` auswerten. Für jeden Knoten `a` gilt hierbei, dass `existsPath es a a` zu `True` auswertet, da man den Knoten `a` immer von sich selbst aus über einen Pfad aus 0 Kanten erreichen kann.

Hinweise:

- Überlegen Sie, wie die Liste der Kanten des Graphen in einem rekursiven Aufruf geeignet modifiziert werden kann, um Terminierung bei zyklischen Graphen sicherzustellen.

f) `isConnected es`
Diese Funktion berechnet, ob ein durch eine Liste von Kanten `es::[(Int,Int)]` dargestellter Graph zusammenhängend ist. Ein Graph heißt zusammenhängend, wenn für alle Knoten `a` und `b` der Knoten `b` von `a` aus erreichbar ist, d.h., `existsPath es a b` ist `True` für alle Knoten `a` und `b`.

Zum Beispiel wertet `isConnected testGraph` zu `False` und `isConnected ((5,1):testGraph)` zu `True` aus.