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 15.01.2026, 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 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. Generell sollten alle Programme für alle Eingaben terminieren, solange in der Spezifikation (bzw. der Aufgabenstellung) nicht explizit etwas anderes verlangt wird!
- Aufgaben, die mit einem * markiert sind, sind Sonderaufgaben mit erhöhtem Schwierigkeitsgrad. Sie tragen nicht zur Summe der erreichbaren Punkte bei, die für die Klausurzulassung relevant ist, jedoch können diese Aufgaben ganz normal im Tutorium vorgerechnet werden und die in solchen Aufgaben erreichten Punkte werden Ihnen ganz normal gutgeschrieben.

Übungsaufgabe 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`?

d) In Haskell haben Funktionen keine Seiteneffekte. Welche Vorteile ergeben sich daraus?

Übungsaufgabe 2 (Programmieren in Haskell):

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), die Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., arithmetische Operatoren wie `+`, `*`, `-`, ... und Konstanten vom Typ `Bool` oder `Int` 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) `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) `powersOfTwo i0 i1`
Gibt eine Integer–Liste zurück, die die Zweierpotenzen $2^{\texttt{i0}}$ bis $2^{\texttt{i1}}$ enthält. Falls $\texttt{i0} > \texttt{i1}$, 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.

d) `intersection xs ys`
Gibt eine Integer–Liste zurück, die je einmal genau die Elemente enthält, die sowohl in `xs` als auch in `ys` enthalten sind. Auf Eingaben, die Dopplungen von Elementen in `xs` oder `ys` enthalten, darf sich die Funktion beliebig verhalten.

Die Auswertung von `intersection [7,3,5,2] [1,2,3,4]` liefert bspw. den Ausgabewert `[3,2]`.

e) `selectKsmallest k xs`
Gibt das Element zurück, das in der `Integer`-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 von `selectKsmallest 1 [5, 17, 1, 3, 9]` 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.

Die folgende Teilaufgabe bezieht sich auf eine Datenstruktur, welche Graphen repräsentiert. Mathematisch ist ein Graph ein Paar $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, 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 (x, y) bedeutet, dass Knoten y von Knoten x erreicht werden kann, jedoch nicht unbedingt auch umgekehrt. Wir nehmen im Folgenden an, dass jeder Knoten mindestens eine eingehende oder ausgehende Kante besitzt.

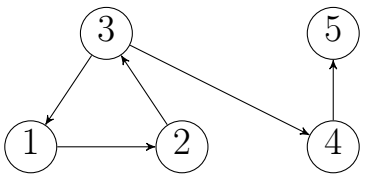

Abbildung 1: Durch die Liste `testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)]::[(Int,Int)]` repräsentierter Graph.

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.

f) `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.

Hausaufgabe 3 (Programmieren in Haskell): (6 + 13 + 6 + 8 + 12 + 10 + 10 = 65 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), die Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., arithmetische Operatoren wie `+`, `*`, `-`, ... und Konstanten vom Typ `Bool` oder `Int` 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.

Hinweise:

- Sie dürfen alle Methoden aus der Übungsaufgabe 2 und aus früheren Teilaufgaben benutzen, auch wenn Sie diese nicht implementiert haben.

a) `pow a b`
Berechnet die Potenz a^b für ganze Zahlen a und b. Ist der Wert von `b` negativ oder `a = b = 0`, darf

sich die Funktion beliebig verhalten. Hierbei dürfen Sie keine in Haskell vordefinierten Funktionen zur Exponentiation verwenden.

b) `isDiv a b`
Gibt genau dann `True` zurück, wenn die ganze Zahl `a` durch die ganze Zahl `b` teilbar ist. Nehmen Sie an, dass 0 durch jede ganze Zahl teilbar ist und keine ganze Zahl durch 0 teilbar ist. Bei negativen Eingaben darf sich die Funktion beliebig verhalten. Hierbei dürfen Sie keine in Haskell vordefinierten Funktionen zur Division oder Modulo-Berechnung verwenden.

> Hinweise:
> - Schreiben Sie eine geeignete Hilfsfunktion, die die Division mit Rest durchführt.

c) `sumUp xs`
Addiert alle Werte eine eingegebenen Integer–Liste auf. Ist die Liste leer, so wird `0` zurückgegeben. Die Auswertung von `sumUp [4, 5, -2]` liefert beispielsweise den Rückgabewert `4 + 5 + (-2) = 7`.

d) `multLists xs ys`
Multipliziert die Listen `xs` und `ys` elementweise. Wenn die Listen nicht die gleiche Länge haben, so sollen nur die Einträge in den Listen bis zum letzten Element der kürzeren Liste beachtet werden. So liefert z.B. `multLists [1, 3, -1] [0, 2, -4]` den Rückgabewert `[0, 6, 4]`. Die Auswertung von `multLists [5] [2, -3, 5]` liefert hingegen `[10]`.

e) `binRep n`
Liefert ein Paar `(e,xs)` so, dass `e` das Vorzeichen repräsentiert, d.h. `0` bei `n = 0`, `1` bei `n > 0` und `-1` bei `n < 0`. Außerdem soll `xs` die Binärdarstellung des Absolutbetrags von `n` als Liste sein. So ist z.B. `binRep 0 = (0,[0])`, `binRep -7 = (-1,[1,1,1])` und `binRep 16 = (1,[1,0,0,0,0])`.

> Hinweise:
> - Sie dürfen die vorimplementierte Methode `div::Int -> Int -> Int` benutzen. Der Aufruf `div a b` liefert als Ergebnis die Ganzzahldivision von `a` durch `b`.
> - Sie dürfen die vorimplementierte Methode `rem::Int -> Int -> Int` benutzen. Der Aufruf `rem a b` liefert als Ergebnis den Rest der Ganzzahldivision von `a` durch `b`.

Die folgenden Teilaufgaben beziehen sich auf die Graphen-Datenstruktur aus der Übungsaufgabe 2.

f) `existsPath es x y`
Diese Funktion berechnet für eine gegebene Liste `es::[(Int,Int)]` von Kanten und zwei Knoten `x,y::Int` einen Wahrheitswert vom Typ `Bool`, der angibt, ob der Knoten `y` vom Knoten `x` 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` ergeben 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:
> - Sie dürfen davon ausgehen, dass die Knoten `x` und `y` im Graphen `es` vorkommen.
> - Überlegen Sie, wie die Liste der Kanten des Graphen in einem rekursiven Aufruf geeignet modifiziert werden kann, um Terminierung des Algorithmus bei zyklischen Graphen sicherzustellen.

g) `isConnected`
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 `x` und `y` der Knoten `y` von `x` aus erreichbar ist, d.h., `existsPath es x y` ist `True` für alle Knoten `x` und `y`.

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

Übungsaufgabe 4 (Typen):

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g`, `h` und `k` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben.

1. ```
   f []     x y = y
   f [z:zs] x y = f [] (z:x) y
   ```

2. ```
   g x 1 = 1
   g x y = (\x -> (g x 0)) y
   ```

3. ```
   h (x:xs) y z = if x then h xs x (y:z) else h xs y z
   ```

4. ```
   i x y z = x z y
   j x = x
   k = i j
   ```

Hinweise:

- Beachten Sie, dass `->` nach rechts assoziiert, d.h., `a -> b -> c` ist identisch zu `a -> (b -> c)`.
- Versuchen Sie diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

Hausaufgabe 5 (Typen): (8 + 6 + 10 + 11 + 10* = 35 + 10* Punkte)

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `h`, `i`, `j` und `m` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben.

1. ```
   f x y z = if y x then x else 0
   f x y z = f x z y
   ```

2. ```
   g = g
   h _ = []
   h _ = g
   h _ = g g
   ```

3. ```
   i []     y = i ((_ -> 0):[]) y
   i (x:xs) y = i (y x:xs) y
   ```

4. ```
   j x      y z = [y:z]
   j (x:xs) y z =  (\(x:xs) -> [y:xs]) xs
   ```

5.* ```
   k f g = \x -> f (g x)
   m = k k k
   ```
   ```

Hinweise:

- Beachten Sie, dass `->` nach rechts assoziiert, während die Funktionsapplikation nach links assoziiert, d.h., der Typ `a -> b -> c` ist identisch zu `a -> (b -> c)` und der Ausdruck `f a b` ist identisch zu `(f a) b`.
- Gehen Sie in der 5. Teilaufgabe wie folgt vor: Überlegen Sie sich zuerst den Typ der Funktion `k` und anschließend den Typ des Ausdrucks `k k`. Bestimmen Sie dann den Typ des Ausdrucks `k k k`.
- Versuchen Sie diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.