Prof. Dr. J. Giesl
M. Hark, J. Hensel, D. Korzeniewski

Allgemeine Hinweise:

- Die **Hausaufgaben** sollen in Gruppen von je **2 Studierenden** aus der **gleichen Kleingruppenübung (Tutorium)** bearbeitet werden. **Namen und Matrikelnummern** der Studierenden sind auf jedes Blatt der Abgabe zu schreiben. **Heften bzw. tackern Sie die Blätter!**
- Die **Nummer der Übungsgruppe** muss **links oben** auf das **erste Blatt** der Abgabe geschrieben werden. Notieren Sie die Gruppennummer gut sichtbar, damit wir besser sortieren können.
- Die Lösungen müssen **bis Dienstag, den 09.01.2018 um 08:00 Uhr** in den entsprechenden Übungskasten eingeworfen werden. Sie finden die Kästen am Eingang Halifaxstr. des Informatikzentrums (Ahornstr. 55). Alternativ können Sie die Lösungen auch vor der Abgabefrist direkt bei Ihrer Tutorin/Ihrem Tutor oder unmittelbar vor Beginn in der Globalübung abgeben.
- In einigen Aufgaben müssen Sie in Haskell programmieren und `.hs`-Dateien anlegen. **Drucken** Sie diese aus **und** schicken Sie sie per **E-Mail** vor Dienstag, dem 09.01.2018 um 08:00 Uhr an Ihre Tutorin/Ihren Tutor.
  Stellen Sie sicher, dass Ihr Programm von `GHC` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.

Tutoraufgabe 1 (Auswertungsstrategie):

Gegeben sei das folgende Haskell-Programm:

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

everySecond :: [Int] -> [Int]
everySecond [] = []
everySecond [x] = [x]
everySecond (x:_:xs) = x : everySecond xs

minus10 :: [Int] -> [Int]
minus10 [] = []
minus10 (x:xs) = x - 10 : minus10 xs

double :: Int -> Int
double x = if x > 0 then x + x else 0
```

Die Funktion `listProd` multipliziert die Elemente einer Liste. Beispielsweise ergibt `listProd [3,5,2,1]` die Zahl `30`. Die Funktion `everySecond` bekommt eine Liste als Eingabe und gibt die gleiche Liste zurück, wobei jedes zweite Element gelöscht wurde. So ergibt `everySecond [1,2,3]` die Liste `[1,3]`. Die Funktion `minus10` gibt seine Eingabeliste zurück, wobei von jedem Element 10 subtrahiert wurde.

Geben Sie alle Zwischenschritte bei der Auswertung der Ausdrücke

`listProd (everySecond (minus10 [3,2,1]))` und `double (listProd [])`

an. Schreiben Sie hierbei (um Platz zu sparen) `p`, `s`, `m` und `d` statt `listProd`, `everySecond`, `minus10` und `double`.

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).

Aufgabe 2 (Auswertungsstrategie): (6 Punkte)

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. 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).

Tutoraufgabe 3 (Listen):

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, 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.

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]:[])`

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.

Aufgabe 4 (Listen): (8 Punkte)

Seien `x`, `y` und `z` ganze Zahlen vom Typ `Int` und `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.

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)`

Hinweise:

- Falls linke und rechte Seite gleich sind, genügt wiederum **eine** Angabe des Typs und der Elementzahl.

Tutoraufgabe 5 (Haskell-Programmierung):

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Verwenden Sie außer Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise) und Vergleichsoperatoren wie `<=`, `==`, ... **keine** vordefinierten Funktionen (dies schließt auch arithmetische Operatoren ein), außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden.

a) `mult x y`
Berechnet `x` · `y`. Sie dürfen dazu `+` und `-` verwenden. Die Funktion darf sich auf negativen Eingaben beliebig verhalten.

b) `bLog x`
Berechnet den aufgerundeten Logarithmus zur Basis 2 von `x`. Somit liefert `bLog 1` den Wert `0`, `bLog 5` den Wert `3` und `bLog 32` den Wert `5`. Sie dürfen hier `+` und `*` verwenden. Die Funktion darf sich auf negativen Eingaben oder der Eingabe 0 beliebig verhalten.

c) `getLastTwo xs`
Berechnet die Teilliste der letzten zwei Elemente der `Int`-Liste `xs`. Beispielsweise berechnet `getLastTwo [12, 7, 23]` den Wert `[7, 23]`. Die Funktion darf sich auf Listen der Länge 0 und 1 beliebig verhalten.

d) `singletons x`
Berechnet eine Liste mit `x` Elementen, wobei jedes Listenelement eine einelementige Liste ist. Die Elemente der einelementigen Listen sind die Zahlen `x`, `x-1`, ..., `1`. Beispielsweise berechnet `singletons 3` die Liste `[[3],[2],[1]]`. Die Funktion darf sich auf negativen Eingaben beliebig verhalten. Sie dürfen hier `-` verwenden.

e) `packing xs ys`
Das erste Argument `xs` ist eine Liste von Listen von Zahlen (vom Typ `[[Int]]`), das zweite Argument ist eine einfache Liste von Zahlen (vom Typ `[Int]`). Die Funktion ersetzt leere Listen in der ersten Eingabeliste durch einelementige Listen. Der Inhalt dieser einelementigen Listen ist die jeweils nächste Zahl aus der zweiten Eingabeliste. Beispielsweise berechnet `packing [[5,6],[],[8],[]] [1,2]` die Liste `[[5,6],[1],[8],[2]]`. Wenn im zweiten Argument nicht genug Zahlen zur Verfügung stehen, werden zusätzliche leere Listen im ersten Argument leer gelassen. Wenn im zweiten Argument zu viele Zahlen zur Verfügung stehen, werden diese ignoriert.

f) `listAdd x xs`
Addiert jeweils das n-te Listenelement auf das $(n+1)$-te Listenelement, wobei auf das erste Listenelement `x` addiert wird. Beispielsweise berechnet `listAdd 5 [1,9,3]` die Liste `[6,10,12]`. Sie dürfen hier `+` verwenden.

Aufgabe 6 (Haskell-Programmierung): (1.5 + 1.5 + 2 + 1.5 + 5.5 = 12 Punkte)

Implementieren Sie alle der im Folgenden beschriebenen Funktionen in Haskell. Geben Sie jeweils auch die Typdeklarationen an. Verwenden Sie außer Listenkonstruktoren `[]` und `:` (und deren Kurzschreibweise), der Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ... und arithmetischen Operatoren wie `+`, `*`, ... **keine** vordefinierten Funktionen außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden.

a) `isEven x`
Berechnet ob `x` gerade ist. Die Funktion darf sich auf negativen Eingaben beliebig verhalten.

b) `arithSeries x d`
Berechnet die Summe aller positiven Zahlen `x`, `x-d`, `x-2d`, `x-3d`, ... Für `arithSeries 6 2` berechnet die Funktion also $6 + 4 + 2$ und für `arithSeries 4 1` wird $4 + 3 + 2 + 1$ berechnet.

Die Funktion darf sich beliebig verhalten, falls `x` oder `d` nicht positiv sind.

Verwenden Sie *nicht* die geschlossene Form der arithmetischen Reihe, sondern lösen Sie diese Aufgabe mittels einer rekursiven Funktion!

c) `isSorted xs`
Berechnet, ob die `Int`-Liste `xs` aufsteigend sortiert ist. Falls ja wird `True` zurückgegeben, andernfalls `False`. Beispielsweise liefert `isSorted [-5, 0, 1, 1, 17]` das Ergebnis `True`.

d) `interval a b`
Gibt eine Liste zurück, die alle ganzen Zahlen zwischen `a` und `b` (jeweils einschließlich) enthält, d.h alle Zahlen x, mit `a` $\leq x \leq$ `b`.

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. Wenn `k` kleiner als 1 oder größer als die Länge von `xs` ist, darf sich die Funktion beliebig verhalten. Der Aufruf `selectKsmallest 3 [4, 2, 15, -3, 5]` würde also `4` zurück geben und `selectKsmallest 1 [5, 17, 1, 3, 9]` würde `1` zurück geben.

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 ys` verwenden, die zu einer Liste `ys` die Anzahl enthaltener Elemente zurückgibt.
- Die Funktion `isSorted` hilft Ihnen in dieser Aufgabe nicht.