Prof. Dr. J. Giesl

S. Dollase, M. Hark, 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 oben links!**
- 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 Freitag, den 21.12.2018 um 12:00** 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 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 Freitag, den 21.12.2018 um 12:00 an Ihre Tutorin/Ihren Tutor. Stellen Sie sicher, dass Ihr Programm von `GHC` **akzeptiert** wird, ansonsten werden keine Punkte vergeben.
- Benutzen Sie in Ihrem Code keine Umlaute, auch nicht in Strings und Kommentaren. Diese führen oft zu Problemen, da diese Zeichen auf verschiedenen Betriebssystemen unterschiedlich behandelt werden. Dadurch kann es dazu führen, dass Ihr Programm bei Ihrer Tutorin/Ihrem Tutor bei Verwendung von Umlauten nicht mehr von `GHC` akzeptiert wird.
- Halten Sie sich beim Lösen von Programmieraufgaben an die auf der Website zur Vorlesung verfügbaren Codekonventionen. Verstöße, die zu unleserlichem Code führen, können zu Punktabzug führen.
- Einige Hausaufgaben müssen im Spiel Codescape gelöst werden:
  https://codescape.medien.rwth-aachen.de/progra/
  Diese Aufgaben werden getrennt von den anderen Hausaufgaben gewertet.

Tutoraufgabe 1 (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. 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).

Aufgabe 2 (Auswertungsstrategie): (4 Punkte)

Gegeben sei das folgende Haskell-Programm:

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

tillN::Int -> [Int]
tillN n = if (n <= 0) then []
          else n : tillN (n-1)

greaterThan::[Int] -> Int -> [Int]
greaterThan [] _ = []
greaterThan (x:xs) n = if(x > n) then x : (greaterThan xs n)
                       else (greaterThan xs n)
```

Die Funktion `first` gibt zu jeder Eingabeliste mit mindestens einem Element den ersten Eintrag der Liste zurück. Im Falle der leeren Liste wird `0` zurückgegeben. Die Funktion `tillN` erzeugt die absteigende Liste aller positiven ganzen Zahlen beginnend mit dem Eingabewert. Bei nicht-positiven Eingaben wird die leere Liste zurückgegeben. Beispielsweise erhält man beim Aufruf `tillN 3` die Liste [3, 2, 1] als Rückgabe. Die Funktion `greaterThan` erhält als Eingabe eine Integer–Liste `xs` und eine ganze Zahl `n`. Der Rückgabewert ist die Liste, die sich ergibt, wenn man alle Einträge löscht, die kleiner oder gleich `n` sind. Rufen wir `greaterThan [-1, 7, 0, 5, -9] 0` auf, so ergibt sich `[7, 5]`.

Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

```
first (greaterThan (tillN 1) (first []))
```

an. Schreiben Sie hierbei (um Platz zu sparen) `f`, `tN` und `gT` statt `first`, `tillN` und `greaterThan`.

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 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.

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:

- 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 in Haskell): (1.5 + 1.5 + 2 + 1.5 + 1.5 = 8 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, wieviele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.

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

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

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

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

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

Hinweise:

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

Tutoraufgabe 5 (Programmieren in Haskell):

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.

Aufgabe 6 (Programmieren in Haskell): (1.5 + 1 + 3 + 1 + 1.5 + 4 = 12 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 `<=`, `==`, ... und die arithmetischen Operatoren `+`, `*`, `-` verwenden, aber **keine** vordefinierten Funktionen außer denen, die in den jeweiligen Teilaufgaben explizit erlaubt werden.

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

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) `pow a b`
Berechnet die Potenz a^b für ganze Zahlen a und b. Ist der Wert von `b` negativ, darf sich die Funktion beliebig verhalten. Hierbei dürfen Sie keine in Haskell vordefinierten Funktionen zur Exponentiation verwenden.

c) `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 sich eine geeignete Hilfsfunktion, die die Division mit Rest durchführt.

d) `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`.

e) `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]`.

f) `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`.
- Benutzen sie die Hilfsmethode aus Teil c).

Aufgabe 7 (Codescape): (Codescape)

Schließen Sie das Spiel Codescape ab, indem Sie den letzten Raum auf Deck 7 auf eine der drei möglichen Arten lösen. Genießen Sie anschließend das Outro. Dieses Deck enthält keine für die Zulassung relevanten Missionen.

Hinweise:

- Es gibt verschiedene Möglichkeiten wie die Story endet, abhängig von Ihrer Entscheidung im finalen Raum.
- Verraten Sie Ihren Kommilitonen nicht, welche Auswirkungen Ihre Entscheidung hatte, bevor diese selbst das Spiel abgeschlossen haben.