Prof. Dr. J. Giesl
S. Dollase, M. Hark, D. Cloerkes

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 **Montag, den 20.01.2020, um 12:00 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 abgeben.
- Auf diesem Blatt müssen Sie in Haskell programmieren und `.hs`-Dateien anlegen. **Drucken** Sie diese aus **und** laden Sie sie fristgerecht im `RWTHmoodle`-Lernraum "Programmierung (Übung - Tutorium)" hoch.

  Stellen Sie sicher, dass Ihr Programm mit `GHCi` **ausgeführt werden kann**, 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 `GHCi` akzeptiert wird.

Tutoraufgabe 1 (Auswertungsstrategie):

Gegeben sei das folgende Haskell-Programm:

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

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

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

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

1. `produkt (absteigend 2)`
2. `summe (absteigend 2)`

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).

Aufgabe 2 (Auswertungsstrategie): (5 Punkte)

Gegeben sei das folgende Haskell-Programm:

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

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

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

Die Funktion `produkt` multipliziert wieder die Elemente einer Liste, beispielsweise ergibt `produkt [3,5,2,1]` die Zahl `30`. Die Funktion `jedesZweite` bekommt eine Liste als Eingabe und gibt die gleiche Liste zurück, wobei jedes zweite Element gelöscht wurde. So ergibt `jedesZweite [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 des Ausdrucks `produkt (jedesZweite (minus10 [3,2,1]))` an. Schreiben Sie hierbei `p`, `j` und `m` statt `produkt`, `jedesZweite` und `minus10`, um Platz zu sparen.

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).
- Beachten Sie, dass der Ausdruck `[3,2,1]` eine Kurzschreibweise für `3 : 2 : 1 : []` ist. Es gilt also `[3,2,1] = 3 : [2,1] = 3 : 2 : [1] = 3 : 2 : 1 : []`. Genauso ist `[x]` eine Kurzschreibweise für `x : []`.

Tutoraufgabe 3 (Listen in Haskell):

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 `[[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) `[] ++ [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 4 (Listen in Haskell): (1.5 + 1.5 + 1.5 + 1.5 + 2 = 8 Punkte)

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.

Hinweise:

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

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]]:[[]]`

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. 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^{i0} bis 2^{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.

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..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.

Aufgabe 6 (Programmieren in Haskell): (2 + 4 + 3 + 3 + 3 + 2 + 3 = 15 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. 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 der Tutoriumsaufgabe 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` ausgwertet 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:

- Für eine Lösung, die Teilergebnisse mehrmals berechnet, werden *keine* Punkte vergeben. Sie können mit obigem Beispiel überprüfen, ob ihre Implementierung effizient ist.

c) `normalize xs`
Gibt eine Integer–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 Integer–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 Integer–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 `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 der Tutoriumsaufgabe verwenden.

g) `multiples xs i0 i1`
Gibt eine Integer–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.