Prof. Dr. M. Müller C. Aschermann, J. Hensel, J. Protze, P. Reble

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
```

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 des Ausdrucks

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

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

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).

Lösung:

```
p (s (m [3,2,1]))
→ p (s (3 - 10 : m [2,1]))
→ p (s (3 - 10 : 2 - 10 : m [1]))
→ p (3 - 10 : s (m [1]))
→ (3 - 10) * p (s (m [1]))
→ (-7) * p (s (m [1]))
→ (-7) * p (s (1 - 10 : m []))
→ (-7) * p (s (1 - 10 : []))
→ (-7) * p (1 - 10 : [])
→ (-7) * ((1 - 10) * p [])
→ (-7) * ((-9) * p [])
→ (-7) * ((-9) * 1)
→ (-7) * (-9)
→ 63
```

Aufgabe 2 (Auswertungsstrategie): (5 Punkte)

Gegeben sei das folgende Haskell-Programm:

```
ofSizeTwo :: [Int] -> Int
ofSizeTwo (_:_:[]) = 1
ofSizeTwo _ = 0

removeOnes :: [Int] -> [Int]
removeOnes [] = []
removeOnes (x:xs) | x == 1    = removeOnes xs
                  | otherwise = x : (removeOnes xs)

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

Die Funktion `ofSizeTwo` gibt zu einer Eingabeliste den Wert `1` zurück, falls diese zwei Elemente enthält. Sonst gibt sie `0` zurück. Die Funktion `removeOnes` entfernt alle Einträge der Zahl `1` aus der Liste. Beispielsweise ergibt `removeTwos [1,2,3,1,4]` die Liste `[2,3,4]`. Die Funktion `listSum` addiert die Elemente einer Liste. Zum Beispiel liefert `listSum [3,5,2,1]` die Zahl `11`.

Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

```
listSum [ ofSizeTwo (removeOnes [1,2,3]), ofSizeTwo [listSum [1,2],2,1] ]
```

an. Schreiben Sie hierbei (um Platz zu sparen) `t`, `r` und `s` statt `ofSizeTwo`, `removeOnes` und `listSum`.

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).

Lösung:

```
s [t (r [1,2,3]), t [s [1,2],2,1]]
→ t (r [1,2,3]) + (s [t [s [1,2],2,1]])
→ t (r [2,3]) + (s [t [s [1,2],2,1]])
→ t (2 : r [3]) + (s [t [s [1,2],2,1]])
→ t (2 : 3 : r []) + (s [t [s [1,2],2,1]])
→ t (2 : 3 : []) + (s [t [s [1,2],2,1]])
→ 1 + (s [t [s [1,2],2,1]])
→ 1 + ((t [s [1,2],2,1]) + (s []))
→ 1 + (0 + (s []))
→ 1 + (0 + 0)
(→ 1 + 0)
→ 1
```

Der kursive Schritt in Klammern wird von vielen Interpretern übersprungen, da reine arithmetische Operationen gemeinsam ausgewertet werden. Wir akzeptieren beides (mit und ohne diesen Schritt) als korrekte Lösung.

Tutoraufgabe 3 (Listen in Haskell):

Seien `x`, `y`, `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, 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]:[])`

Lösung:

a) Beide Ausdrücke haben den Typ `[[Int]]`. Jedoch hat die erste Liste ein Element, während die zweite Liste zwei Elemente besitzt. Die Ausdrücke sind also nicht gleich.

b) Beide Ausdrücke sind nicht typkorrekt. Daher würde die Gleichung in Haskell nicht gelten. Allerdings würden beide Ausdrücke die gleiche (ungültige) Liste darstellen, nämlich `[[],x]` (somit sind beide Antworten, ob die Gleichung gilt oder nicht, zulässig). Diese Liste ist nicht typkorrekt, da sie sowohl eine Liste, als auch einen `Int` Wert enthält.

c) Die Gleichung gilt, denn beide Ausdrücke repräsentieren die Liste `[x,y]` vom Typ `[Int]`, welche zwei Elemente enthält.

d) Die Gleichung gilt, denn beide Ausdrücke repräsentieren die gleiche Liste, welche zuerst die Elemente `x`, `y`, `z`, anschließend die n Elemente der Liste `xs` und schließlich die m Elemente der Liste `ys` enthält. Diese Liste enthält also insgesamt $3 + n + m$ Elemente und ist vom Typ `[Int]`.

e) Beide Ausdrücke sind vom gleichen Typ `[[[Int]]]` und sind Listen mit zwei Elementen. Diese Elemente sind auch noch gleich (einerseits die Liste `[[]]` und andererseits die Liste `[x:xs,ys]`), allerdings ist ihre Reihenfolge in den beiden Ausdrücken unterschiedlich, sodass die Gleichung nicht gilt.

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 `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], xs] = [] ++ [[]] ++ [[x]] ++ [xs]`

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

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

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

e) `[([] : []) ++ [xs] ++ [(y:[])]] = [xs] : [[y]] : ([[]] : [])`

Lösung:

a) Beide Ausdrücke repräsentieren die gleichen Listen der Länge 3 und vom Typ `[[Int]]`.

b) Der rechte Ausdruck ist nicht typkorrekt, da `[xs]` vom Typ `[[Int]]` ist aber `[x]` vom Typ `[Int]`. Der Operator `++` kann daher nicht genutzt werden. Der linke Ausdruck ist typkorrekt und repäsentiert eine Liste der Länge 2 vom Typ `[[Int]]`. Die Ausdrücke sind also nicht gleich.

c) Der linke Ausdruck ist nicht typkorrekt, da `x` und `ys` nicht den gleichen Typ haben, aber in der gleichen Liste enthalten sein sollen, und da `ys` nicht den gleichen Typ hat wie die Elemente von `xs`, aber ein Element vpn `xs` sein soll. Der rechte Ausdruck repräsentiert eine Liste der Länge $n + m + 1$ und ist vom Typ `[Int]`. Die Ausdrücke sind also nicht gleich.

d) Beides sind typkorrekte Listenausdrücke (äquivalent zu `[x,y,z,x,y] ++ xs` der Länge $n+5$ und ebenfalls vom Typ `[Int]`. Die Ausdrücke sind also gleich.

e) Der linke Ausdruck ergibt ausgeschrieben die Liste `[[[],xs,[y]]]`, also eine Liste mit einem Element, welches mehrere Listen enthält. Der Typ des gesamten Ausdrucks ist daher `[[[Int]]]`. Der zweite Ausdruck ergibt ausgeschrieben die Liste `[[xs], [[y]], [[]]]`, also eine Liste mit 3 Elementen, deren Einträge Listen von Listen enthält. Der gesamte Ausdruck hat daher den Typen `[[[Int]]]`. Da beide Ausdrücke zwar den gleichen Typen haben, aber nicht die gleichen Listen darstellen, sind die Ausdrücke also nicht gleich.

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 $x, x-1, \ldots, 1$. 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.

Lösung:

```
-- a
mult :: Int -> Int -> Int
mult _ 0 = 0
mult x y = x + mult x (y-1)

-- b
bLog :: Int -> Int
bLog 1 = 0
bLog x = bLogH 2 1
  where
    bLogH :: Int -> Int -> Int
    bLogH y r | x > y     = bLogH (y*2) (r+1)
              | otherwise = r
```

```
-- c
getLastTwo :: [Int] -> [Int]
getLastTwo [x, y]   = [x, y]
getLastTwo (_:x:xs) = getLastTwo (x:xs)

-- d
singletons :: Int -> [[Int]]
singletons 0 = []
singletons n = [n] : singletons (n-1)

-- e
packing :: [[Int]] -> [Int] -> [[Int]]
packing []       _      = []
packing xs       []     = xs
packing ([]:xs) (y:ys) = [y] : packing xs ys
packing (x:xs)  (y:ys) = x   : packing xs (y:ys)

-- f
listAdd :: Int -> [Int] -> [Int]
listAdd _ []     = []
listAdd z (x:xs) = (z+x) : listAdd x xs
```

Aufgabe 6 (Haskell-Programmierung): (2 + 2 + 2 + 2 + 3 + 2,5 + 1,5 + 2 + 2 + 2 = 21 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 früheren Teilaufgaben implementiert wurden.

a) `fibon x`
Gibt die entsprechende Fibonacci-Zahl zurück. Die erste Zahl lautet 1, die zweite 1, die dritte 2 $(1+1)$, die vierte 3 $(1+2)$, usw. Der Aufruf von `fibon 4` soll also z.B. die Zahl 3 zurückgeben. Die Funktion darf sich auf Eingaben von $x < 1$ beliebig verhalten.

b) `containsFour xs`
Überprüft ob sich in der übergebenen Liste (vom Typ `Int`) die Zahl 4 befindet. Ist dies der Fall, gibt die Funktion `True` zurück, sonst `False`. Beispielsweise liefert der Aufruf von `containsFour [3,4]` als Ergebnis `True`.

c) `isEven x`
Liefert `True` zurück, wenn es sich bei der übergebenen Zahl (vom Typ `Int`) um eine gerade Zahl handelt. Beispielsweise geben `isEven 0` und `-2` als Ergebnis `True` zurück. Der Aufruf `isEven 5` führt zu `False`.

d) `countEven xs`
Liefert die Anzahl der geraden Zahlen in der übergebenen `Int`-Liste zurück. Zum Beispiel liefert `countEven [1,2,3,4,5]` als Ergebnis 2. Nutzen Sie zur Umsetzung die Funktion `isEven` aus der vorherigen Aufgabe.

e) `minimalNumber xs`
Berechnet das kleinste Element, das in der `Int`-Liste xs vorkommt. Zum Beispiel liefert der Aufruf kleinstes [42, 7, 23] den Wert 7. Die Funktion darf sich auf leeren Listen beliebig verhalten.

f) `getFirstAndLast xs`
Liefert eine Liste mit genau zwei Element zurück. Hierbei ist das erste Element das erste Element der übergebenen `Int`-Liste. Das zweite Element ist das letzte Element der übergebenen Liste. Beispielsweise liefert ein Aufruf von `getFirstAndLast [1,2,3,4]` als Ergebnis `[1,4]` zurück.

g) `doubleList xs`
Verdoppelte die übergebene `Int`-Liste, in dem es eine Liste zurückgibt, die jedes Element dupliziert. Zum Beispiel gibt der Aufruf `doubleList [1,5,2]` als Ergebnis `[1,1,5,5,2,2]` zurück.

h) `merge xs ys`
Die Funktion bekommt zwei *aufsteigend sortierte* Listen `xs` und `ys` als Argumente und gibt eine aufsteigend sortierte Liste zurück, die alle Elemente von `xs` und `ys` enhält. Ein Aufruf `merge [1,3,5] [2,4,4,8]` liefert somit `[1,2,3,4,4,5,8]`. Ein Aufruf `merge [] [1,2,3,4]` liefert hingegen `[1,2,3,4]`.

Hinweise:

- Vergleichen Sie Listenelemente mit der Funktion `<=`.

i) `split xs`
Die Funktion bekommt eine Liste `xs` als Argument und gibt ein Tupel von zwei Listen zurück. Hierbei werden die Elemente der Liste `xs` abwechselnd auf die beiden Ergebnislisten verteilt, so dass das erste Element der Liste `xs` das erste Element der ersten Ergebnisliste ist und das zweite Element der `xs` das erste Element der zweiten Ergebnisliste ist. Ein Aufruf `split [1,2,3,4,5]` liefert das Ergebnis `([1,3,5],[2,4])`.

Hinweise:

- Konstruieren Sie 3 Fälle: Die Liste `xs` hat kein, ein oder mindestens zwei Elemente.
- Im dritten Fall hilft es, erst den Rest der Liste (ohne die ersten zwei Elemente) aufzuteilen.

j) `sort xs`
Die Funktion bekommt eine Liste `xs` und gibt ihre Elemente in aufsteigend sortierter Reihenfolge aus. Listen der Länge höchstens eins sind schon sortiert. Ansonsten soll die Liste mit der Funktion `split` in zwei kleinere Listen zerteilt werden, diese dann rekursiv sortiert und mit der Funktion `merge` zu einer sortierten Liste zusammengefasst werden.

Lösung:

```
-- a
fibon :: Int -> Int
fibon 1 = 1
fibon 2 = 1
fibon n = fibon (n-1) + fibon (n-2)

-- b
containsFour :: [Int] -> Bool
containsFour [] = False
containsFour (x:xs) | x == 4    = True
                    | otherwise = containsFour xs

-- c
isEven :: Int -> Bool
isEven 0 = True
isEven 1 = False
isEven x | x > 0     = isEven (x-2)
         | otherwise = isEven (x+2)

-- d
countEven :: [Int] -> Int
countEven [] = 0
countEven (x:xs) | isEven x = 1 + countEven(xs)
                 | otherwise  = countEven(xs)
```

```
-- e
minimalNumber :: [Int] -> Int
minimalNumber (x:xs) = mHelper x xs
    where mHelper :: Int -> [ Int ] -> Int
          mHelper m [] = m
          mHelper m (y:ys) | m < y = mHelper m ys
                           | otherwise = mHelper y ys

-- f
getFirstAndLast :: [Int] -> [Int]
getFirstAndLast (x:xs) = [x, falHelper xs]
    where falHelper :: [Int] -> Int
          falHelper [y] = y
          falHelper (_:xs) = falHelper xs

-- g
doubleList :: [Int] -> [Int]
doubleList [] = []
doubleList (x:xs) = x:x:(doubleList xs)

-- h
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys


-- i
split [] = ([], [])
split [x] = ([x], [])
split (x : y : rest) = (x : links, y : rechts)
  where (links, rechts) = split rest

-- j
sort [] = []
sort [x] = [x]
sort xs = merge (sort links) (sort rechts)
  where (links, rechts) = split xs
```