Prof. Dr. J. Giesl D. Cloerkes, S. Dollase, N. Lommen, D. Meier, F. Meyer

Tobias Grether, (205632), Tutorium 13
Lea-Noora Potdschnig, (434855), Tutorium 13

Aufgabe 3 (Auswertungsstrategie): (26 Punkte)

Gegeben sei das folgende Haskell-Programm:

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

firstN :: Int -> [Int] -> [Int]
firstN n [] = []
firstN n (x:xs) =
  if n > 0 then x : firstN (n-1) xs else []

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

Die Funktion `doubleElements` berechnet eine Liste, welche aus einer gegebenen Liste hervorgeht, indem jedes Element dieser Liste noch einmal hinter sich selbst eingefügt wird. Zum Beispiel wertet der Ausdruck `doubleElements [1,2,3]` zu `[1,1,2,2,3,3]` aus. Die Funktion `firstN` gibt zu jeder Liste `xs` den längsten Anfang `ys` von `xs` zurück, sodass `ys` höchstens eine gegebene Anzahl an Elementen besitzt. Der Ausdruck `firstN 2 [1,2,3]` wertet beispielsweise zu `[1,2]` aus, während hingegen `firstN 5 [1,2,3]` zu `[1,2,3]` auswertet. Die Funktion `listSum` bildet die Summe über alle Elemente einer Liste vom Typ `[Int]`. Wird `listSum [1,2,3]` aufgerufen, so ergibt sich `6`.
Geben Sie alle Zwischenschritte bei der Auswertung des Ausdrucks

`listSum (firstN (1+0) (doubleElements [1+0, 1+0]))`

an. Unterstreichen Sie vor jedem Auswertungsschritt den Teil des Ausdrucks, der als Nächstes an seiner äußersten Position ausgewertet wird. Um Platz zu sparen können Sie hierbei `dE`, `fN` und `lS` statt <u>d</u>ouble<u>E</u>lements, <u>f</u>irst<u>N</u> und <u>l</u>ist<u>S</u>um schreiben.

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 6 (Listen in Haskell): (4+5+5+5+5 = 24 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, wie viele Elemente in der jeweiligen Liste enthalten sind und welchen Typ sie hat.
Beispiel: Die Liste `[[x,y],ys]` hat den Typ `[[Int]]` und enthält 2 Elemente.

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

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

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

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

e) `(x:y:([z]++ys)):[xs]` = `[x,y,z]:[ys] ++ [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 9 (Programmieren in Haskell): (7+6+10+9+11+7 = 50 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), `True` und `False`, Werte des Typs `Int`, die Listenkonkatenation `++`, Vergleichsoperatoren wie `<=`, `==`, ..., boolesche Funktionen wie `&&`, `||`, `not` und die arithmetischen Operatoren `+`, `*`, `-` verwenden, aber **keine** weiteren vordefinierten Funktionen. Schreiben Sie ggf. Hilfsfunktionen, um sich die Lösung der Aufgaben zu vereinfachen. Sie dürfen jederzeit Hilfsfunktionen aus vorherigen Teilaufgaben verwenden, auch wenn Sie diese nicht selbst implementiert haben.

a) `symmetricDifference xs ys`
Diese Funktion berechnet die symmetrische Differenz zweier Listen `xs::[Int]` und `ys::[Int]`. Die symmetrische Differenz ist eine Liste `zs::[Int]`, welche alle Elemente enthält, die entweder nur in `xs` oder nur in `ys` enthalten sind. Die Reihenfolge der Elemente innerhalb der resultierenden Liste ist hierbei unerheblich. Wenn ein Wert sowohl in `xs` als auch in `ys` vorkommt, darf er nicht in `symmetricDifference xs ys` auftreten, auch wenn er unterschiedlich oft in `xs` und `ys` vorkommt.

Die Auswertung von `symmetricDifference [1,1,2,4,5] [2,2,3,4,6]` kann beispielsweise die Liste `[1,1,5,3,6]::[Int]` liefern.

b) `powerlist xs`
Berechnet eine Liste `ps::[[Int]]` aller Teillisten von `xs::[Int]`. Für jede Liste in `ps` soll die Reihenfolge der Listenelemente dieselbe sein wie die Reihenfolge der entsprechenden Elemente in `xs`. Die Reihenfolge der Elemente von `ps` selbst ist hierbei unerheblich. Mehrfach vorkommende Werte in `xs` können auch in den Teillisten entsprechend oft auftreten. Wenn l die Länge von `xs` bezeichnet, so ist 2^l die Länge von `powerlist xs`.

Die Auswertung von `powerlist [1,2,3]` kann beispielsweise die Liste `[[],[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]` der Länge 8 ergeben. Der Ausdruck `powerlist [2,2]` kann hingegen beispielsweise zur Liste `[[],[2],[2],[2,2]]` ausgewertet werden.

c) `permutations xs`
Diese Funktion soll zu einer Liste `xs::[Int]` eine Liste aller Permutationen von `xs` berechnen. Eine Permutation von `xs` ist hierbei eine Liste `ps::[Int]`, welche dieselben Elemente wie `xs` enthält, jedoch kann die Reihenfolge der Elemente in `ps` von der Reihenfolge der Elemente in `xs` abweichen. Die Reihenfolge der Elemente von `permutations xs` kann hierbei beliebig sein. Wenn `xs` eine Liste der Länge l ist, so ist `permutations xs` von der Länge $l!$. Falls `xs` die leere Liste ist, so darf sich die Funktion beliebig verhalten.

Der Ausdruck `permutations [1,2,3]` könnte damit zu der Liste `[[1,2,3],[2,1,3],[3,2,1],[1,3,2],[3,1,2],[2,3,1]]` vom Typ `[Int]` auswerten.

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

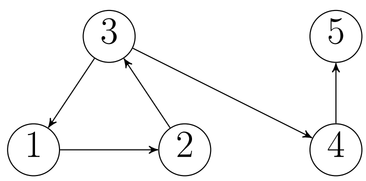

Abbildung 1: Durch die Liste `testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)]::[(Int,Int)]` repräsentierter Graph.

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.

d) `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.

e) `existsPath es a b`
Diese Funktion berechnet für eine gegebene Liste `es::[(Int,Int)]` von Kanten und zwei Knoten `a,b::Int` einen Wahrheitswert vom Typ `Bool`, der angibt, ob der Knoten `b` vom Knoten `a` 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` berechnen 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:

- Überlegen Sie, wie die Liste der Kanten des Graphen in einem rekursiven Aufruf geeignet modifiziert werden kann, um Terminierung bei zyklischen Graphen sicherzustellen.

f) `isConnected es`
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 `a` und `b` der Knoten `b` von `a` aus erreichbar ist, d.h., `existsPath es a b` ist `True` für alle Knoten `a` und `b`.

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

```haskell
permutations :: [Int] -> [[Int]]
permutations [] = [[]]
permutations xs = [y : zs | (y, ys) <- select xs, zs <- permutations ys]
  where
    select [] = []
    select (x : xs) = (x, xs) : [(y, x : ys) | (y, ys) <- select xs]

allConnectionsToNode :: [(Int, Int)] -> Int -> [(Int, Int)]
allConnectionsToNode xs node = [x | x <- xs, tupelSecond x == node]

allConnectionsFromNode :: [(Int, Int)] -> Int -> [(Int, Int)]
allConnectionsFromNode xs node = [x | x <- xs, tupelSecond x == node]

allConnectedNodes :: [(Int, Int)] -> Int -> [Int]
allConnectedNodes edges node = [tupelFirst x | x <- allConnectionsToNode edges node]
```

abgebildete Graph kann als Liste `testGraph = [(1,2),(2,3),(3,1),(4,5),(3,4)]::[(Int,Int)]` seiner Kanten in Haskell dargestellt werden.

d) `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.

e) `existsPath es a b`
Diese Funktion berechnet für eine gegebene Liste `es::[(Int,Int)]` von Kanten und zwei Knoten `a,b::Int` einen Wahrheitswert vom Typ `Bool`, der angibt, ob der Knoten `b` vom Knoten `a` aus mithilfe der Kanten aus der Liste `es` erreicht werden kann.

```haskell
allOfUnique :: [Int] -> [Int]
allOfUnique vals = allOfUniqueHelper vals []

allOfUniqueHelper :: [Int] -> [Int] -> [Int]
allOfUniqueHelper [] vals = vals
allOfUniqueHelper (x: vals) store | containsNumber x store = allOfUniqueHelper vals store -- If it already has the number, skip
                                  | otherwise = allOfUniqueHelper vals (store ++ [x]) -- else add to store and go to next

indexOfInternal :: Int -> Int -> [Int] -> Int
indexOfInternal _ _ [] = 0
indexOfInternal i n (x : xs)
  | n == x = i
  | otherwise = indexOfInternal (i + 1) n xs

listSwap :: Int -> Int -> [Int] -> [Int]
listSwap _ _ [] = []
listSwap i1 i2 xs = [if x == i1 then i2 else if x == i2 then i1 else x | x <- xs]

tupelToList :: (Int, Int) -> [Int]
tupelToList tpl = [tupelFirst tpl, tupelSecond tpl]

symmetricDifference :: [Int] -> [Int] -> [Int]
symmetricDifference [] [] = []
symmetricDifference xs ys = [x | x <- xs ++ ys, (containsNumber x xs && not (containsNumber x ys)) || (containsNumber x ys && not (containsNumber x xs))]

powerlist :: [Int] -> [[Int]]
powerlist [] = [[]]
powerlist (x : xs) = [x : ps | ps <- powerlist xs] ++ powerlist xs

nodes :: [(Int, Int)] -> [Int]
nodes edges = allOfUnique( allOf edges )
```

```haskell
containsNumber :: Int -> [Int] -> Bool
containsNumber _ [] = False
containsNumber n (x : xs)
  | x == n = True
  | otherwise = containsNumber n xs

containsTupel :: (Int, Int) -> [(Int, Int)] -> Bool
containsTupel _ [] = False
containsTupel tupel (t: tupels) | areTupelEqual tupel t = True
                                | otherwise = containsTupel tupel tupels

areTupelEqual :: (Int, Int) -> (Int, Int) -> Bool
areTupelEqual one two = (tupelFirst one == tupelFirst two) && (tupelSecond one == tupelSecond two)

areTupelAndElementsEqual :: (Int, Int) -> Int -> Int -> Bool
areTupelAndElementsEqual tupel a b = (tupelFirst tupel == a) && (tupelSecond tupel == b)

indexOfNormalized :: Int -> [Int] -> Int
indexOfNormalized n xs = indexOf n xs + 1

indexOf :: Int -> [Int] -> Int
indexOf _ [] = 0
indexOf n xs = indexOfInternal 0 n xs

tupelFirst :: (a, b) -> a
tupelFirst (a,b) = a

tupelSecond :: (a, b) -> b
tupelSecond (a,b) = b

allOf :: [(Int, Int)] -> [Int]
allOf tpls = [tupelFirst tpl | tpl <- tpls] ++ [tupelSecond tpl | tpl <- tpls]
```