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

Tutoraufgabe 1 (Überblickswissen):

a) In Haskell haben Funktionen keine Seiteneffekte. Welche Vorteile ergeben sich daraus?

b) Angenommen Haskell würde Seiteneffekte erlauben, sehen Sie Probleme, die bei der Auswertungsstrategie auftreten könnten?

Lösung:

a) Daraus ergibt sich eine Vielzahl von Vorteilen. Es wird deutlich einfacher, über das Programm nachzudenken, da man sich sicher sein kann, dass die Funktion keine unerwarteten Nebeneffekte hat. Das macht es auch simpler, die Korrektheit eines Programms zu beweisen. Außerdem erleichtert eine Garantie der Seiteneffektfreiheit die Parallelisierung von Programmcode, da sich so der Code nicht gegenseitig beeinflusst.

b) Wenn Seiteneffekte erlaubt wären, so würde die Bedarfsevaluation (lazy evaluation) nicht mehr einfach umsetzbar sein. Jeder spätere Ausdruck könnte Seiteneffekte haben, so dass entweder jede*r Programmierer*in genau Bescheid wissen muss, wann welcher Ausdruck ausgewertet werden kann/soll oder das Programm wird nahezu unmöglich vorhersehbar.

Tutoraufgabe 2 (Datenstrukturen):

In dieser Aufgabe beschäftigen wir uns mit *Kindermobiles*, die man beispielsweise über das Kinderbett hängen kann. Ein Kindermobile besteht aus mehreren Figuren, die mit Fäden aneinander aufgehängt sind. Als mögliche Figuren im Mobile beschränken wir uns hier auf *Sterne*, *Seepferdchen*, *Elefanten* und *Kängurus*.
An Sternen und Seepferdchen hängt keine weitere Figur. An jedem Elefant hängt eine weitere Figur, unter jedem Känguru hängen zwei Figuren. Weiterhin hat jedes Känguru einen Beutel, in dem sich etwas befinden kann (z. B. eine Zahl).
In Abbildung 1 finden Sie zwei beispielhafte Mobiles[1].

a) Implementieren Sie in Haskell einen parametrisierten Datentyp `Mobile a` mit vier Konstruktoren (für Sterne, Seepferdchen, Elefanten und Kängurus), mit dem sich die beschriebenen Mobiles darstellen lassen. Verwenden Sie den Typparameter `a` dazu, den Typen der Känguru-Beutelinhalte festzulegen.

Modellieren Sie dann die beiden in Abbildung 1 dargestellten Mobiles als Ausdruck dieses Datentyps in Haskell. Nehmen Sie hierfür an, dass die gezeigten Beutelinhalte vom Typ `Int` sind.

```
mobileLinks :: Mobile Int
mobileLinks = ...
```

```
mobileRechts :: Mobile Int
mobileRechts = ...
```

Hinweise:

- Für Tests der weiteren Teilaufgaben bietet es sich an, die beiden Mobiles als konstante Funktionen im Programm zu deklarieren.
- Schreiben Sie `deriving Show` an das Ende Ihrer Datentyp-Deklaration. Damit können Sie sich in `GHCi` ausgeben lassen, wie ein konkretes Mobile aussieht.

[1] Für die Grafik wurden folgende Bilder von Wikimedia Commons (2012) verwendet:

- Stern `https://commons.wikimedia.org/wiki/File:Crystal_Clear_action_bookmark.png`
- Seepferdchen `https://commons.wikimedia.org/wiki/File:Seahorse.svg`
- Elefant `https://commons.wikimedia.org/wiki/File:African_Elephant_Transparent.png`
- Känguru `https://commons.wikimedia.org/wiki/File:Kangourou.svg`

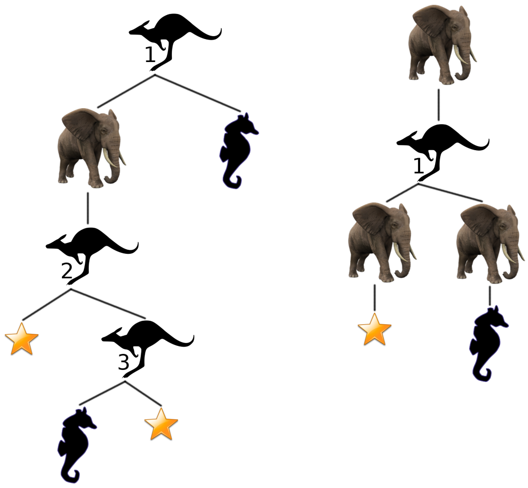

Abbildung 1: Zwei beispielhafte Mobiles.

b) Schreiben Sie in Haskell eine Funktion `count :: Mobile a -> Int`, die die Anzahl der Figuren im Mobile berechnet. Für die beiden gezeigten Mobiles soll also 8 und 6 zurückgegeben werden.

c) Schreiben Sie eine Funktion `liste :: Mobile a -> [a]`, die alle in den Känguru-Beuteln enthaltenen Elemente in einer Liste (mit beliebiger Reihenfolge) zurückgibt. Für das linke Mobile soll also die Liste `[1,2,3]` (oder eine Permutation davon) berechnet werden. Sie dürfen die vordefinierte Funktion `++` verwenden.

d) Schreiben Sie eine Funktion `greife :: Mobile a -> Int -> Mobile a`. Diese Funktion soll für den Aufruf `greife mobile n` die Figur mit Index `n` im Mobile `mobile` zurückgeben.

Wenn man sich das Mobile als Baumstruktur vorstellt, werden die Indizes entsprechend einer *Tiefensuche*[2] berechnet:

Wir definieren, dass die oberste Figur den Index 1 hat. Wenn ein Elefant den Index n hat, so hat die Nachfolgefigur den Index $n + 1$.

Wenn ein Känguru den Index n hat, so hat die linke Nachfolgefigur den Index $n+1$. Wenn entsprechend dieser Regeln alle Figuren im linken Teil-Mobile einen Index haben, hat die rechte Nachfolgefigur den nächsthöheren Index.

Im linken Beispiel-Mobile hat das Känguru mit Beutelinhalt `3` also den Index 5.

Hinweise:

- Benutzen Sie die Funktion `count` aus Aufgabenteil b).
- Falls der übergebene Index kleiner als 1 oder größer als die Anzahl der Figuren im Mobile ist, darf sich Ihre Funktion beliebig verhalten.

Lösung:

a)

```
data Mobile a = Stern | Seepferdchen | Elefant (Mobile a)
                | Kaenguru a (Mobile a) (Mobile a) deriving Show

mobileLinks :: Mobile Int
mobileLinks = Kaenguru 1
                  (Elefant (Kaenguru 2
```

[2]siehe auch Wikipedia: `https://de.wikipedia.org/wiki/Tiefensuche`

```
                              Stern
                              (Kaenguru 3
                                  Seepferdchen
                                  Stern
                              )
                        ))
                        Seepferdchen

mobileRechts :: Mobile Int
mobileRechts = Elefant (Kaenguru 1 (Elefant Stern) (Elefant Seepferdchen))
```

b)
```
count :: Mobile a -> Int
count Stern                = 1
count Seepferdchen         = 1
count (Elefant     m)      = 1 + count m
count (Kaenguru _  m1 m2)  = 1 + count m1 + count m2
```

c)
```
liste :: Mobile a -> [a]
liste Stern                    = []
liste Seepferdchen             = []
liste (Elefant m)              = liste m
liste (Kaenguru inhalt m1 m2)  = inhalt : liste m1 ++ liste m2
```

d)
```
greife :: Mobile a -> Int -> Mobile a
greife x                   1 = x
greife (Elefant m)         x = greife m (x-1)
greife (Kaenguru _ m1 m2)  x
  | x-1 <= count m1   = greife m1 (x-1)
  | otherwise         = greife m2 (x-1 - count m1)
greife _                   _ = Stern -- Gesuchte Figur existiert nicht
```

Tutoraufgabe 3 (Datenstrukturen (Video)):

In dieser Aufgabe geht es darum, arithmetische Ausdrücke auszuwerten. Wir betrachten Ausdrücke auf den ganzen Zahlen (`Int`) mit Variablen sowie den Operationen der Addition und Multiplikation.

a) Wir wollen Variablennamen durch den Datentyp `VariableName` darstellen. In dieser Aufgabe betrachten wir nur die Variablen `X` und `Y`.

Erstellen Sie den Datentyp `VariableName`, sodass er entweder den Wert `X` oder den Wert `Y` annehmen kann. Erstellen Sie außerdem die Funktion `getValue :: VariableName -> Int`, welche der Variablen `X` den Wert `5` und der Variablen `Y` den Wert `13` zuordnet. Die Auswertung des Ausdrucks `getValue X` soll also `5` ergeben.

b) Nun wollen wir arithmetische Ausdrücke durch den Datentyp `Expression` darstellen. Ein arithmetischer Ausdruck ist entweder ein konstanter `Int`-Wert, der Name einer Variablen (`VariableName`), die Addition zweier `Expressions` oder die Multiplikation zweier `Expressions`.

Erstellen Sie den entsprechenden Datentyp `Expression` mit den Datenkonstruktoren `Constant`, `Variable`, `Add` und `Multiply`.

Hinweise:

- Auch hier und bei `VariableName` ist es hilfreich, `deriving Show` an das Ende der Datentyp-Deklaration zu schreiben.

c) Um eine `Expression` zu einem `Int` auszuwerten, benötigen wir die `Expression` selbst sowie die Funktion `getValue`, welche den einzelnen Variablen Werte zuordnet. Falls die `Expression` ein konstanter `Int`-Wert ist, so ist eben dieser `Int`-Wert das Ergebnis. Falls die `Expression` eine Variable ist, so ist das Ergebnis der `Int`-Wert, welcher der Variablen von der Funktion `getValue` zugeordnet wird. Falls die `Expression` die Addition bzw. Multiplikation zweier `Expressions` ist, so werden zunächst diese beiden `Expressions` ausgewertet und die beiden `Int`-Werte anschließend miteinander addiert bzw. multipliziert, um das Ergebnis zu erhalten.

Erstellen Sie die entsprechende Funktion `evaluate :: Expression -> Int`.

Angenommen `exampleExpression :: Expression` sei wie folgt definiert.

```
exampleExpression = Add
                      (Add
                        (Constant 20)
                        (Constant 17))
                      (Add
                        (Variable X)
                        (Multiply
                          (Add
                            (Constant 14)
                            (Constant 7))
                          (Constant 2)))
```

Der Ausdruck `evaluate exampleExpression` würde dann zum Wert `84` ausgewertet.

Hinweise:

- Die `exampleExpression` entspricht dem arithmetischen Ausdruck $(20 + 17) + (x + ((14 + 7) \cdot 2))$.

d) Wird eine `Expression` mehrfach ausgewertet, so ist es wünschenswert, sie möglichst klein zu halten, damit die Auswertung möglichst schnell geht. Wir wollen nun eine gegebene `Expression` unter der Annahme unbekannter Variablenwerte optimieren. In einem ersten Schritt fassen wir dazu die Addition zweier Konstanten zu einer neuen Konstanten zusammen, welche als Wert die Summe der Werte der beiden ursprünglichen Konstanten hat. Mit der Multiplikation gehen wir analog vor. Alle anderen `Expressions` bleiben unverändert.

Erstellen Sie die entsprechende Funktion `tryOptimize :: Expression -> Expression`.

Der Ausdruck `tryOptimize (Add (Constant 20) (Constant 17))` würde beispielsweise zum Wert `Constant 37` ausgewertet. Die Ausdrücke `tryOptimize (Add (Variable X) (Constant 2))` und `tryOptimize (Multiply (Add (Constant 14) (Constant 7)) (Constant 2))` würden hingegen ihren Parameter genau so zurückgeben, d.h. sie werten zu `Add (Variable X) (Constant 2)` bzw. `Multiply (Add (Constant 14) (Constant 7)) (Constant 2)` aus.

e) In komplexen `Expressions` kann es Teilausdrücke geben, welche nur aus der Berechnung von Konstanten bestehen. Diese Teilausdrücke können durch *partielle Auswertung* vollständig durch eine neue Konstante ersetzt werden, welche als Wert die Evaluation des Teilausdrucks hat.

Erstellen Sie die entsprechende Funktion `evaluatePartially :: Expression -> Expression`. Für eine Addition werden zunächst die beiden Teilausdrücke partiell ausgewertet. Das Ergebnis ist nun die mit `tryOptimize` optimierte Addition der partiell ausgewerteten Teilausdrücke. Die Multiplikation arbeitet analog. Alle anderen Ausdrücke bleiben unverändert.

Der Ausdruck `evaluatePartially exampleExpression` würde beispielsweise zum Wert `Add (Constant 37) (Add (Variable X) (Constant 42))` ausgewertet.

Lösung:

a)
```
data VariableName = X | Y deriving Show

getValue :: VariableName -> Int
getValue X = 5
getValue Y = 13
```

b)
```
data Expression = Constant  Int
                | Variable  VariableName
                | Add       Expression Expression
                | Multiply  Expression Expression
                deriving Show
```

c)
```
evaluate :: Expression -> Int
evaluate (Constant c)       = c
evaluate (Variable v)       = getValue v
evaluate (Add e1 e2)        = evaluate e1 + evaluate e2
evaluate (Multiply e1 e2)   = evaluate e1 * evaluate e2
```

d)
```
tryOptimize :: Expression -> Expression
tryOptimize (Add      (Constant c1) (Constant c2)) = Constant (c1 + c2)
tryOptimize (Multiply (Constant c1) (Constant c2)) = Constant (c1 * c2)
tryOptimize e = e
```

e)
```
evaluatePartially :: Expression -> Expression
evaluatePartially (Add e1 e2)      = tryOptimize (Add
                                        (evaluatePartially e1)
                                        (evaluatePartially e2))
evaluatePartially (Multiply e1 e2) = tryOptimize (Multiply
                                        (evaluatePartially e1)
                                        (evaluatePartially e2))
evaluatePartially e                = e
```

Aufgabe 4 (Datenstrukturen): (4+4+4+10+4+10+9+10+10 = 65 Punkte)

In dieser Aufgabe betrachten wir Binärbaume, deren innere Knoten einzelne Werte eines Typs `a` speichern, während die Blätter jeweils einen Wert des Typs `b` speichern. Als Beispiel betrachten wir den Baum in Abbildung 2.
Sie dürfen jederzeit Hilfsfunktionen aus vorherigen Teilaufgaben verwenden, auch wenn Sie diese nicht selbst implementiert haben.
Der Beispielbaum hat fünf Blätter mit den Zeichen `'g'`, `'u'`, `'l'`, `'f'`, `'i'` vom Typ `Char` und vier weiteren inneren Knoten, die eine einstellige Funktion vom Typ `Int -> Bool` enthalten.
Schreiben Sie zu jeder der im Folgenden zu implementierenden Funktionen auch eine Typdeklaration. Beachten Sie, dass auch die Wurzel eines Baums als innerer Knoten gilt.

a) Implementieren Sie in Haskell einen parametrisierten Datentyp `BinTree a b`, mit dem Binärbäume mit Werten vom Datentyp `a` in den inneren Knoten und mit Werten vom Datentyp `b` in den Blättern dargestellt werden können. Dabei soll sichergestellt werden, dass jeder innere Knoten genau zwei Nachfolger hat.

   Hinweise:
   - Ergänzen Sie `deriving Show` am Ende der Datentyp-Deklaration, damit `GHCi` die Bäume auf der Konsole anzeigen kann.

b) Definieren Sie den Beispielbaum aus Abb. 2 als `example`:

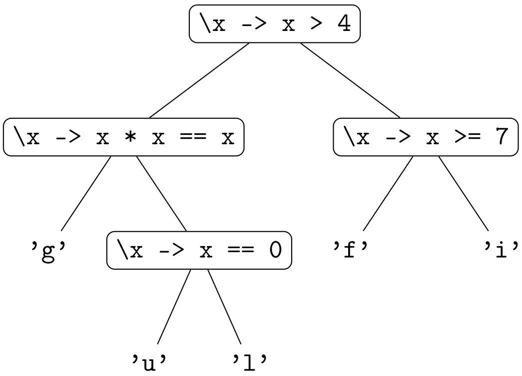

Abbildung 2: Ein beispielhafter Binärbaum.

```
example :: BinTree (Int -> Bool) Char
example = ...
```

Hinweise:

- Importieren Sie das zusätzliche Modul `Text.Show.Functions` durch Einfügen der Zeile `import Text.Show.Functions` am Anfang ihrer Datei, damit `GHCi` auch Bäume wie in Abb. 2 auf der Konsole anzeigen kann, welche Funktionen enthalten.

c) Schreiben Sie die Funktion `countInnerNodes`, die einen Binärbaum vom Typ `BinTree a b` übergeben bekommt und die Anzahl der inneren Knoten als `Int` zurückgibt, d.h. die Anzahl der Knoten, die keine Blätter sind.

Für den Beispielbaum `example` soll der Aufruf `countInnerNodes example` also zu `4` auswerten.

d) Schreiben Sie die Funktion `decodeInt`. Diese bekommt als erstes Argument einen Binärbaum vom Typ `BinTree (Int -> Bool) b` und als zweites Argument einen Wert vom Typ `Int`. Der Rückgabewert dieser Funktion ist vom Typ `b`. Für einen Baum `bt` und eine Zahl `x` gibt `decodeInt bt x` das Zeichen zurück, an das man gelangt, wenn man ausgehend von der Wurzel in jedem inneren Knoten die Funktion vom Typ `Int -> Bool` des jeweiligen Knotens auf die Zahl `x` anwendet, wobei das linke Kind eines Knotens als Nachfolger gewählt wird, falls die Funktion zu `False` auswertet, und das rechte Kind gewählt wird, falls sie zu `True` auswertet. Wird `decodeInt` auf einem Blatt aufgerufen, wird dessen Wert zurückgegeben.

Für den Beispielbaum `example` soll der Aufruf `decodeInt example 0` also `'l'` zurückgeben.

e) Schreiben Sie die Funktion `decode`. Diese bekommt als erstes Argument einen Binärbaum vom Typ `BinTree (Int -> Bool) b` und als zweites Argument eine Liste vom Typ `[Int]` übergeben. Für einen Baum `bt` und eine Liste `xs` sucht `decode bt xs` zu jeder Zahl `x` aus der Liste `xs` den Wert `decodeInt bt x` und fügt die so erhaltenen Werte in einer Liste zusammen.

Für den Beispielbaum `example` soll der Aufruf `decode example [0,1,5,-4,7]` also den `String` `"lufgi"` zurückgeben.

f) Implementieren Sie eine Funktion `insertSorted`. Diese enthält als erstes Argument einen binären Suchbaum `bt` vom Typ `BinTree Int ()` und als zweites Argument eine Zahl vom Typ `Int`. Ein binärer Suchbaum ist ein Binärbaum, sodass für jeden inneren Knoten gilt, dass alle im linken Teilbaum gespeicherten Werte vom Typ `Int` echt kleiner sind als das im Knoten selbst gespeicherte Element, während alle im rechten Teilbaum gespeicherten Werte vom Typ `Int` nicht kleiner sind als der im Knoten selbst gespeicherte Wert. Der Typ `()` ist der Typ des leeren Tupels mit dem einzigen Wert `()::()` und dient hier dazu, zu verhindern, dass Werte vom Typ `Int` auch in den Blättern gespeichert werden. Ein binärer Suchbaum speichert also die in ihm enthaltenen Werte nur in den inneren Knoten, nicht jedoch in den Blättern. Der Aufruf `insertSorted bt x` soll nun den binären Suchbaum berechnen, welcher durch Einfügen des Werts `x` als neuen inneren Knoten aus `bt` entsteht.

Ein solcher binärer Suchbaum `bt` und das Ergebnis des Aufrufs `insertSorted bt 3` sind in Abb. 3 dargestellt.

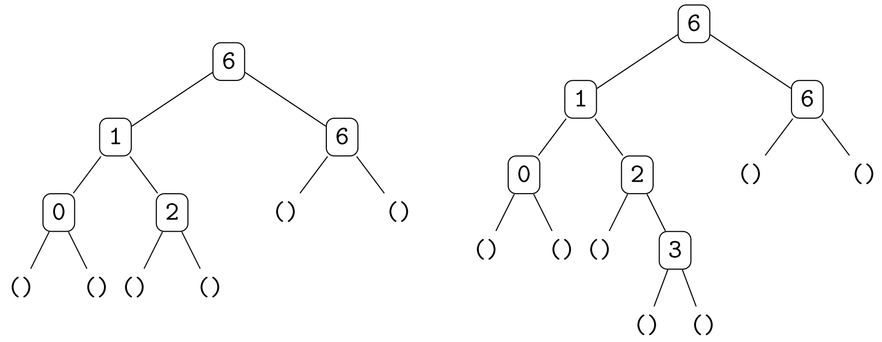

Abbildung 3: Ein binärer Suchbaum `bt` (links), welcher die Zahlen `0,1,2,6,6` speichert und das Ergebnis des Aufrufs `insertSorted bt 3` (rechts).

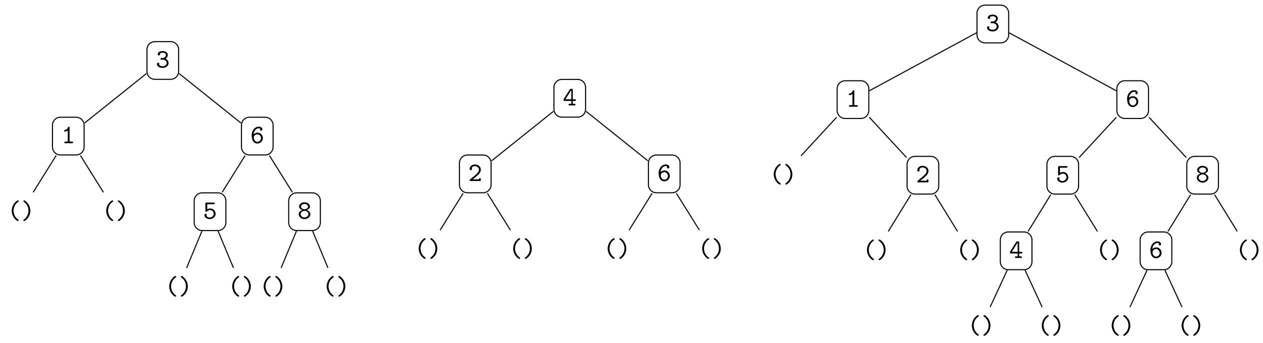

Abbildung 4: Ein Suchbaum `bt1` vom Typ `BinTree Int ()` (links), ein weiterer Suchbaum `bt2` vom Typ `BinTree Int ()` (Mitte) und ein mögliches Ergebnis des Aufrufs `mergeTrees bt1 bt2` (rechts).

g) Schreiben Sie eine Funktion `treeSort`, welche eine Liste vom Typ `[Int]` durch geeignete Aufrufe der Funktion `insertSorted` sortiert.

Der Aufruf `treeSort [3,2,1,3,4,5]` soll zu der Liste `[1,2,3,3,4,5]::[Int]` auswerten.

h) Schreiben Sie eine Funktion `mergeTrees`, welche zwei binäre Suchbäume `bt1` und `bt2` vom Typ `BinTree Int ()` als Argumente erhält. Der Rückgabewert von `mergeTrees` ist ein binärer Suchbaum, welcher alle in `bt1` und `bt2` enthaltenen Werte speichert. Ist ein Wert `x` dabei n_1 mal in `bt1` und n_2 mal in `bt2` enthalten, so soll der Wert `x` anschließend $n_1 + n_2$ mal in dem Ergebnis des Aufrufs `mergeTrees bt1 bt2` enthalten sein.

Abb. 4 stellt zwei Suchbäume `bt1` und `bt2` sowie ein mögliches Ergebnis des Aufrufs `mergeTrees bt1 bt2` dar.

Hinweise:

- Benutzen Sie die Funktion `insertSorted` aus Aufgabenteil f).

i) Implementieren Sie eine Funktion `numberOfOccurrences`, welche einen binären Suchbaum `bt` vom Typ `BinTree Int ()` als erstes Argument und einen Wert `x` vom Typ `Int` als zweites Argument erhält. Die Funktion berechnet dann die Anzahl der Vorkommen des Werts `x` im binären Suchbaum `bt`. Nutzen Sie dabei die Suchbaumeigenschaft von `bt` aus, um eine effiziente Vorgehensweise Ihrer Implementierung sicherzustellen.

Für den Suchbaum `bt` aus Abb. 3 wertet der Aufruf `numberOfOccurrences bt 6` zu `2` aus, da der Wert `6` zweimal im Suchbaum `bt` enthalten ist.

Lösung:

```
import Text.Show.Functions
```

a)
```
data BinTree a b = Node a (BinTree a b) (BinTree a b) | Leaf b
                         deriving Show
```

b)
```
example :: BinTree (Int -> Bool) Char
example = Node (\x -> x > 4)
            (Node (\x -> x * x == x)
                (Leaf 'g')
                (Node (\x -> x == 0) (Leaf 'u') (Leaf 'l'))
            )
            (Node (\x -> x >= 7) (Leaf 'f') (Leaf 'i'))
```

c)
```
countInnerNodes :: BinTree a b -> Int
countInnerNodes (Leaf _) = 0
countInnerNodes (Node _ bt1 bt2) =
  1 + countInnerNodes bt1 + countInnerNodes bt2
```

d)
```
decodeInt :: BinTree (Int -> Bool) b -> Int -> b
decodeInt (Leaf c)        x = c
decodeInt (Node f bt1 bt2) x | f x         = decodeInt bt2 x
                             | otherwise = decodeInt bt1 x
```

e)
```
decode :: BinTree (Int -> Bool) b -> [Int] -> [b]
decode _ []      = []
decode bt (x:xs) = decodeInt bt x : decode bt xs
```

f)
```
examplebt :: BinTree Int ()
examplebt = Node 6 (Node 1
                     (Node 0 (Leaf ()) (Leaf ()))
                     (Node 2 (Leaf ()) (Leaf ())))
                   (Node 6 (Leaf ()) (Leaf ()))

insertSorted :: BinTree Int () -> Int -> BinTree Int ()
insertSorted (Leaf ())     v = Node v (Leaf ()) (Leaf ())
insertSorted (Node x l r) v
  | v < x      = Node x (insertSorted l v) r
  | otherwise = Node x l (insertSorted r v)
```

g)
```
treeSort :: [Int] -> [Int]
treeSort keys = toList (buildTree keys)
  where
    buildTree []     = Leaf ()
    buildTree (x:xs) = insertSorted (buildTree xs) x
    toList (Leaf ())     = []
    toList (Node x l r) = toList l ++ x:toList r
```

h)
```
examplebt1 :: BinTree Int ()
examplebt1 = Node 3
               (Node 1 (Leaf ()) (Leaf ()))
               (Node 6
```

```
                        (Node 5 (Leaf ()) (Leaf ()))
                        (Node 8 (Leaf ()) (Leaf ())))

examplebt2 :: BinTree Int ()
examplebt2 = Node 4
               (Node 2 (Leaf ()) (Leaf ()))
               (Node 6 (Leaf ()) (Leaf ()))


mergeTrees :: BinTree Int () -> BinTree Int () -> BinTree Int ()
mergeTrees bt (Leaf ())    = bt
mergeTrees bt (Node x l r) =
  mergeTrees (mergeTrees (insertSorted bt x) l) r
```

i)
```
numberOfOccurrences :: BinTree Int () -> Int -> Int
numberOfOccurrences (Leaf ()) _ = 0
numberOfOccurrences (Node x l r) v
  | v < x     = numberOfOccurrences l v
  | v > x     = numberOfOccurrences r v
  | otherwise = 1 + numberOfOccurrences r v
```

Tutoraufgabe 5 (Typen):

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g` und `h` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben.

a)
```
f []     x y = y
f [z:zs] x y = f [] (z:x) y
```

b)
```
g x 1 = 1
g x y = (\x -> (g x 0)) y
```

c)
```
h (x:xs) y z = if x then h xs x (y:z) else h xs y z
```

Hinweise:

- Versuchen Sie diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

Lösung:

a)
```
f []     x y = y
f [z:zs] x y = f [] (z:x) y
```

Wir beginnen mit einer generellen Definition `f :: a -> b -> c -> d`.

- Aus der ersten Definition ergibt sich, dass der Typ des dritten Arguments dem Typ des Rückgabewertes von `f` entspricht, d.h. `c = d`.
- Aus der zweiten Definition ergibt sich, dass der Typ des ersten Arguments von `f` eine Liste von Listen ist. Hat das daraus entnommene `z` Typ `e`, hat also das erste Argument den Typ `[[e]]`.

- Aus der zweiten Definition ergibt sich weiterhin, dass `z` in `x` eingefügt werden kann. Daher hat `x` (und damit das zweite Argument von `f`) den Typ `[e]`.

Insgesamt ergibt sich also der Typ `f :: [[e]] -> [e] -> c -> c`.

b)
```
g x 1 = 1
g x y = (\x -> (g x 0)) y
```

Wir beginnen mit einer generellen Definition `g :: a -> b -> c`.

- Aus der ersten Definition ergibt sich, dass der Typ des zweiten Arguments `Int` sein muss, da es auf `1` gematcht wird. Es gilt also `b = Int`.
- Aus der ersten Definition ergibt sich, dass der Typ des Rückgabewertes `Int` ist. Es gilt also `c = Int`.
- Aus der zweiten Definition ergibt sich durch `(\x -> (g x 0)) y = g y 0`, dass der Typ des ersten Arguments den gleichen Typ wie das zweite Argument haben muss, also `a = b`.

Insgesamt ergibt sich also der Typ `g :: Int -> Int -> Int`.

c)
```
h (x:xs) y z = if x then h xs x (y:z) else h xs y z
```

Wir beginnen mit einer generellen Definition `h :: a -> b -> c -> d`.

- Das erste Argument ist eine Liste, aus der `x` entnommen wird. Hat `x` den Typ `e`, gilt also `a = [e]`.
- Der Wert `x` wird als Bedingung in einem `if ... then ... else` verwendet, also ist `e = Bool`.
- Wir verwenden `x` auch als zweites Argument für `h`. Also gilt `b = e = Bool`.
- Wir fügen `y` in die Liste `z` ein. Es gilt also `c = [Bool]`.

Insgesamt ergibt sich also der Typ `h :: [Bool] -> Bool -> [Bool] -> d`.

Aufgabe 6 (Typen): (8 + 7 + 9 + 11 = 35 Punkte)

Bestimmen Sie zu den folgenden Haskell-Funktionen `f`, `g`, `h`, `i`, `j` und `k` den jeweils allgemeinsten Typ. Geben Sie den Typ an und begründen Sie Ihre Antwort. Gehen Sie hierbei davon aus, dass alle Zahlen den Typ `Int` haben, die Funktion `+` den Typ `Int -> Int -> Int` hat, die Funktion `head` den Typ `[a] -> a`, und die Funktion `==` den Typ `a -> a -> Bool` hat.

a)
```
f xs     y []     = []
f (x:xs) y (z:zs) = if x then y + z : f xs z zs else z : f xs z zs
```

b)
```
g x y = g (head y) y
g x y = (\x y -> x) x x
```

c)
```
h w x []     z = if x == [] then head z else w x []
h w x (y:ys) z = if w x ys then head z else y
```

d)
```
i x y z = x z y
j x = x
k = i j
```

Hinweise:

- Versuchen Sie diese Aufgabe ohne Einsatz eines Rechners zu lösen. Bedenken Sie, dass Sie in einer Prüfung ebenfalls keinen Rechner zur Verfügung haben.

Lösung:

a)
```
f xs     y []     = []
f (x:xs) y (z:zs) = if x then y + z : f xs z zs else z : f xs z zs
```

Da `f` drei Parameter bekommt, hat der Typ die Form `f :: a -> b -> c -> d`.

- Aus der ersten Gleichung folgt, dass `c` und `d` Listen sein müssen.
- Aus `(x:xs)` können wir folgern, dass auch `a` eine Liste ist.
- Aus `if x` folgt, dass `x` den Typ `Bool` hat.
- Weiter folgt aus `y + z`, dass `y` und `z` beide den Typ `Int` haben. Damit ergibt sich `f :: [Int] -> Int -> [Bool] -> [e]`.
- Aus `y + z : f xs z zs` bzw. `z : f xs z zs` folgt, dass der Ergebnistyp eine Liste mit Elementen des Typs von `z::Int` ist.

Somit ergibt sich insgesamt

`f :: [Bool] -> Int -> [Int] -> [Int].`

b)
```
g x y = g (head y) y
g x y = (\x y -> x) x x
```

Da `g` zwei Argumente hat, nehmen wir den Typ `g :: a -> b -> c` an.

- Aus der ersten Gleichung folgt, dass `b` eine Liste mit Elementen des Typs `a` ist.
- Der Typ des Lambda-Ausdrucks `\x y -> x` ist `d -> e -> f`. Aus dem Ausdruck folgt weiterhin, dass die Typvariable `f` mit der Typvariablen `d` übereinstimmt.
- Da dieser Lambda-Ausdruck zuerst auf den ersten Parameter der Funktion angewendet wird, folgt aus der zweiten Gleichung, dass dieser Parametertyp dem Ergebnistyp entspricht: `g :: a -> [a] -> a`.

c)
```
h w x []     z = if x == [] then head z else w x []
h w x (y:ys) z = if w x ys then head z else y
```

Die Funktion `h` hat 4 Parameter. Wir gehen also erstmal vom Typ `h :: a -> b -> c -> d -> e` aus.

- Aus der ersten Gleichung erfahren wir, dass der zweite Parameter (`x`) einen Listentyp (`[f]`) hat.
- Aus `head z` folgt, dass der vierte Parameter ebenfalls einen Listentyp `[g]` hat und dass der Elementtyp `g` dem Ergebnistyp `e` der Funktion `h` entspricht.
- Aus `w x []` folgt, dass `w` eine Funktion mit 2 Parametern ist. Wir nehmen hier also eine Funktion `w :: b -> i -> e` an, da der erste Parameter von `w` der zweite Parameter der ersten Gleichung ist und der Ergebnistyp von `w` mit dem Ergebnistyp von `h` übereinstimmt. Der erste Parameter der Funktion `w` ist wie `x` also ebenfalls vom Typ `[f]`.
- Aus `if w x ys` in der zweiten Gleichung ergibt sich, dass der Rückgabetyp von `w`, und damit auch von `h`, `Bool` ist.
- Damit folgern wir, dass die Elemente der Liste `z` und `y` vom Typ `Bool` sein müssen. Das dritte Argument von `h` (und das zweite Argument von `w`) ist damit selbst vom Typ `[Bool]`.

Also ergibt sich insgesamt der Typ

`h :: ([f] -> -> [Bool] -> Bool) -> [f] -> [Bool] -> [Bool] -> Bool.`

d)
```
i x y z = x z y
j x = x
k = i j
```

Da die erste Gleichung für die Funktion `i` 3 Parameter besitzt, nehmen wir `i :: a -> b -> c -> d` an.

- Der Parameter `x` ist jedoch selbst eine Funktion, welche mit den beiden Parametern `z` und `y` von `i` aufgerufen wird und dessen Rückgabetyp dem Ergebnistyp von `i` entspricht. Wir nehmen daher `x :: c -> b -> d` an.

Damit ergibt sich `i :: (c -> b -> d) -> b -> c -> d`.

Mithilfe der zweiten Gleichung stellen wir fest, dass `j` einen Parameter erwartet, dessen Typ auch gleich wieder der Rückgabetyp von `j` selbst ist. Wir erhalten daher `j :: e -> e`.

Da nach der dritten Gleichung `k` keine Parameter erwartet, nehmen wir `k :: f` an, wobei `f` ebenfalls der Rückgabetyp von `i j` ist.

- Um den Typ `f` von `i j` zu bestimmen, stellen wir fest, dass die Typvariable `e` dem Typen `b -> d` entsprechen muss, denn `(->)` assoziert nach rechts. Damit ergibt sich auch, dass `e` und `c` identisch sein müssen.

Wir erhalten daher `k :: b -> (b -> d) -> d` als allgemeinsten Typ von `k`.