TECHNISCHE UNIVERSITÄT CAROLO-WILHELMINA ZU BRAUNSCHWEIG

Lösungsvorschlag

Lösungsvorschläge zu: Eine Einführung in die funktionale Programmierung mit Haskell

Jan Oliver Ringert

Wintersemester 07/08

Institut für Programmierung und Reaktive Systeme
Dr. Werner Struckmann

1 Lösungsvorschlag zu Blatt 1

Aufgabe 1.1.

1. Starten Sie die interaktive Shell und probieren Sie ein paar einfache Berechnungen wie im Beispiel der Übung aus.

2. Laden Sie das Modul Simple in der Shell. Benutzen Sie die Funktionen `inc`, `sum'`, `exclaim` und `averageOf2` für einige Beispiele.

Lösung 1.1.

Aufgabe 1.2.

1. Fügen Sie zur Datei `Simple.hs` die Definition aus Listing 1.3 hinzu und vergleichen Sie die Signaturen von `inc` und `inc'`.

2. Erweitern Sie die Definitionen um die Funktion `double`, welche einen Wert verdoppelt, und eine Funktion `square`, welche einen Wert quadriert.

3. Sind die folgenden Konstrukte korrekt? Welche Typen besitzen sie? Prüfen Sie Ihre Vermutungen in der interaktiven Shell.
   a) `sum' 12`
   b) `sum' (square 12)`
   c) `sum' inc 12`

Lösung 1.2.

1. Die Signatur von `inc'` (bestimmt durch `:t inc'`) lautet `inc' :: Int -> Int`. Damit hat die Funktion den gleichen Typen wir `inc`. Um diesen Umstand einzusehen kann man die Funktion in λ-Notation aufschreiben und eine β-Reduktion durchführen (siehe unten). Man nennt diese Methode, von der Haskell hier Gebrauch macht, Currying.

2. Die Implementierung der Funktionen erfolgt analog zu den bestehenden.

```
1 -- Verdoppeln eines Int-Wertes
2 double :: Int -> Int
3 double x = x + x
4
```

```
5 -- Quadrieren eines Int-Wertes
6 square :: Int -> Int
7 square x = x * x
```

Listing 1.1: Ausschnitt aus dem Modul `Simple`

3. Durch einfaches Eingeben der Zeilen in der interaktiven Shell können Sie feststellen, ob die Konstrukte syntaktisch gültig sind. Den Typ bestimmen Sie mit `:t` *Konstrukt*:

   a) `sum' 12` gültig, Typ ist `sum' 12 :: Int -> Int`

   b) `sum' (square 12)` gültig, Typ ist `sum' (square 12) :: Int -> Int`

   c) `sum' inc 12` ungültig.

   Zur Verdeutlichung was hier passiert kann man sich die obigen Funktionen in der λ-Notation aufschreiben und das Currying nachvollziehen, mit dem Haskell aus einer Funktion mit mehreren Elementen mehrere Funktionen mit je einem Element macht. Es ergeben sich die beiden Darstellungen λ_x`(inc x)` und $\lambda_y\lambda_z$`(sum' y z)` und insgesamt:
   $\lambda_y\lambda_z$`(sum' y z)` λ_x`(inc x) 12`
   Nun ist deutlich, was passiert: Es wird in der Funktion `sum' y z` als erster Parameter die Funktion `inc` eingesetzt. Das Argument hat somit den Typ `Int -> Int`, gefordert wird aber `Int`. Es tritt ein Fehler auf.

Aufgabe 1.3.

1. Programmieren Sie eine Funktion `mittelPunkt`, die zwei `Punkte` als Argumente annimmt und einen dritten ausgibt, der den geringsten möglichen Abstand zu beiden besitzt.

2. Geben Sie die Definition einer Funktion an, die für einen `Punkt` entscheidet, in welchem Quadranten er sich befindet (per Fallunterscheidung, `if-then-else` oder einer geeigneten Kombination).

Lösung 1.3.

1. Die Funktion greift auf die Koordinaten der beiden Punkte zu, wie dies schon in der Funktion `norm` realisiert wurde. Die Ausgabe des neuen Punktes funktioniert über das Anlegen eines Tupels wie aus `kreisFlUm` bekannt mit der Berechnung der neuen Koordinaten als lokale Variablen (mit Hilfe von `where`).

```
1 -- Punkt zwischen zwei Punkten berechnen
2 mittelPunkt :: Punkt -> Punkt -> Punkt
3 mittelPunkt (x1, y1) (x2, y2) = (xm, ym) where
4         xm = (x1 + x2)/2
```

```
5         ym = (y1 + y2)/2
```

Listing 1.2: Implementierung der Funktion `mittelPunkt`

2. Die folgende Implementierung verwendet eine Fallunterscheidung wie von der Funktion `signum` bekannt. Eine Behandlung für den Spezialfall x- oder y-Koordinate Null muss gefunden werden. Dieser Fall kann auf verschiedene Weisen behandelt werden, solange er dokumentiert wird.

```
1 -- Bestimmung des Quadranten, in dem ein Punkt liegt
2 -- Nummerierung startet "rechts oben" mathematisch positiv
3 -- Auf Achsen gilt Quadrant ist "0"
4 quadrant :: Punkt -> Int
5 quadrant (x, y) | (x * y == 0)          = 0
6                 | (x > 0 && y > 0)      = 1
7                 | (x < 0 && y > 0)      = 2
8                 | (x < 0 && y < 0)      = 3
9                 | (x > 0 && y < 0)      = 4
```

Listing 1.3: Implementierung der Funktion `quadrant`

Aufgabe 1.4.

1. Schreiben Sie eine Funktion, die eine `Bank` annimmt und eine Liste der Kunden ausgibt.

2. Finden Sie durch Ausprobieren und Betrachten der Signatur heraus, welche Auswirkungen die folgenden Funktionen auf Listen haben: `length`, `head`, `tail`, `sum`, `product`.

Lösung 1.4.

1. Die Funktion funktioniert analog zur Funktion `gesamtBestand`. Anstelle der Addition der Beträge wird hier jedoch eine Liste konstruiert. Der Name des Kunden kann bequem durch die Funktion `name :: Konto -> String` erhalten werden.

```
1 -- Erstellen einer Liste der Kunden einer Bank
2 kundenNamen :: Bank -> [String]
3 kundenNamen [] = []
4 kundenNamen (k:ks) = (name k):(kundenNamen ks)
```

Listing 1.4: Implementierung der Funktion `kundenNamen`

2. Siehe die Referenz unter
`http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html`.

2 Lösungsvorschlag zu Blatt 2

Aufgabe 2.1.

1. Geben Sie eine Funktion an, mit der die Fakultät einer ganzen positiven Zahl n berechnet werden kann (den Typen `Integer` anstelle von `Int` verwenden).
2. Geben Sie eine Funktion an, die für die beiden Argumente `c::Char` und `n::Int` eine Zeichenkette der Länge `n` ausgibt, welche nur aus Wiederholungen des Buchstaben `c` besteht.
3. Erweitern Sie die partielle Funktion `natSum` zu einer totalen Funktion über dem Datentypen `Int`. Geben Sie in vorher undefinierten Fällen eine Fehlermeldung durch die Funktion `error` aus.
4. Setzen Sie die iterative Variante des Euklidischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers in eine funktionale Implementierung in Haskell um.

Lösung 2.1.

1. Analog zu der Funktion `natSum` erhält man die folgende Implementierung.

```
1 -- Fakultät eines Integers
2 fak :: Integer -> Integer
3 fak 0 = 1
4 fak n = n * (fak (n-1))
```

2. Hier ist der Basisfall, dass das Zeichen Null mal kopiert werden soll. Es wird dann eine leere Liste (ein leerer `String`) geliefert. In allen anderen Fällen wird das Zeichen vor eine Liste der rekursiv produzierte Liste von Zeichen gehängt.

```
1 -- Kopieren eines Zeichens c und zwar n mal
2 copyCh :: Int -> Char -> String
3 copyCh 0 c =[]
4 copyCh n c = c:(copyCh (n-1) c)
```

3. Für ein illegales Argument (negative Zahl) wird mit der Funktion `error :: [Char] -> a` eine Fehlermeldung ausgegeben. Man beachte, dass hier `-1` in `natSum -1` als die Infixfunktion `(-) :: (Num a) => a -> a -> a` mit einem Parameter interpretiert wird (`natSum (-1)` wird wie gewünscht behandelt).

```haskell
-- Berechnung der Summe der natürlichen Zahlen 1 bis n
natSum :: Int -> Int
natSum 0 = 0
natSum n | n < 0        = error "natSum: Argument darf nicht
    negativ sein."
         | otherwise    = natSum (n-1) + n
```

4. Der Basisfall dieser rekursiven Funktion ist die Abbruchbedingung in der iterativen Version. In der folgenden Implementierung sind nur positive ganze Zahlen zugelassen.

```haskell
-- Euklid nur für positive ganze Zahlen
ggT :: Int -> Int -> Int
ggT a 0 = a     -- Basisfall
ggT a b = if (a > b) then
                ggT (a - b) b
          else
                ggT a (b - a)
```

Aufgabe 2.2.

1. Geben Sie eine Funktion `listProduct :: Num a => [a] -> a` an, die alle Elemente der Liste multipliziert und das Ergebnis zurück liefert.
2. Modifizieren Sie die Funktion so, dass für eine leere Eingabe `0` das Ergebnis ist.
3. Erstellen Sie eine Funktion `verketten`, die eine Liste von Listen des gleichen Typs zu einer Liste zusammenfasst.

Lösung 2.2.

1. Es wird der Mustervergleich und Rekursion verwendet. Bei dieser einfachen Variante muss für den Basisfall einer leeren Liste eine `1` für die Multiplikation geliefert werden.

```haskell
-- Produkt aller Elemente einer Liste berechnen
listProduct :: Num a => [a] -> a
listProduct [] = 1
listProduct (x:xs) = x * listProduct xs
```

2. Der Basisfall aus der Rekursion muss abgeändert werden. Eine Möglichkeit ist es, den Basisfall in die Zeile des Rekursionsschrittes aufzunehmen (siehe unten). Damit tritt das Muster `[]` nur noch auf, wenn tatsächlich eine leere Liste übergeben wurde.

```haskell
-- Produkt aller Elemente einer Liste berechnen
-- bei [] als Eingabe 0 ausgeben
listProduct :: Num a => [a] -> a
listProduct [] = 0
listProduct (x:xs) = x * if (xs == []) then 1
                         else listProduct xs
```

3. Hier wird wiedereinmal der Mustervergleich für die Konstruktoren von Listen benutzt. Der Typ `[[a]]` ist hier eine Liste von Listen der Typvariablen `a`. Im Muster `verketten (l:ls)` ist `l` eine Liste und `ls` eine Liste von Listen.

```haskell
-- Listen des gleichen Typs verketten
verketten :: [[a]] -> [a]
verketten [] = []
verketten (l:ls) = l ++ verketten ls
```

Aufgabe 2.3. Implementieren Sie die Funktion `isSorted`, die für eine Liste feststellt, ob ihre Elemente aufsteigend sortiert sind. Verwenden Sie hierzu die Typklasse `Ord`.

Lösung 2.3. In der folgenden Implementierung werden werden zwei Basisfälle für die Rekursion benötigt. Der erste Fall gilt wieder für die leere Liste. Der zweite deckt das Vorkommen einelementiger Listen ab, welche durch das Muster `isSorted (x:y:z)` nicht abgedeckt werden (hier ist `z` eine Liste während `x` und `y` einfache Elemente sind).

```haskell
-- Prüft eine Liste, ob sie sortiert ist
isSorted :: Ord a => [a] -> Bool
isSorted [] = True      -- Basisfall 1
isSorted [a] = True     -- Basisfall 2
isSorted (x:y:z) = (x <= y) && isSorted (y:z)
```

Aufgabe 2.4.

1. Geben Sie eine Funktion, die bei einer gegebenen Preisliste den Preis für ein Produkts liefert.

2. Implementieren Sie eine Funktion, die aus einer Einkaufsliste und einer Preisliste eine Liste mit Tupeln (Preis pro Stück, Menge) erstellt.

3. Entwickeln Sie eine Funktion, die aus einer wie im vorherigen Schritt erzeugten Liste eine Endsumme berechnet.

4. Fügen Sie alle entwickelten Funktionen in der Definition der Funktion `kosten :: PreisListe -> EinkaufsListe -> Cent` zusammen, sodass die Endsumme eines Einkaufs berechnet wird.

Lösung 2.4.

1. Die Liste wird nach dem passenden Produk-Preis-Tupel durchsucht. Durch einen `where`-Ausdruck wird das Listenelement in seine beiden Bestandteile zerlegt, die dann als lokale Variablen verwendet werden können.

```
1 -- Zu einem Produkt den Preis herausuchen
2 preis4Produkt :: PreisListe -> Produkt -> Cent
3 preis4Produkt [] p = error "Produkt␣nicht␣in␣Preisliste"
4 preis4Produkt (x:xs) p = if p == prod
5                          then preis
6                          else preis4Produkt xs p
7                          where (prod, preis) = x
```

2. Der Preis für ein Produkt wird mit der im vorherigen Schritt entwickelten Funktion in die lokale Variable `preis` gelesen.

```
1 -- Aus einer Preis- und einer EinkaufsListe Tupel für den
2 -- Einezlpreis und die jeweilige Menge erstellen
3 preisMenge :: PreisListe -> EinkaufsListe -> [(Cent, Int)]
4 preisMenge ps [] = []
5 preisMenge ps (e:es) = (preis, anzahl):(preisMenge ps es)
6                  where   (produkt, anzahl) = e
7                          preis = preis4Produkt ps produkt
```

3. Der Code sollte selbsterklärend sein.

```
1 -- Aus einer Liste mit Einzelpreis und Menge den Gesamtpreis
      berechnen
2 gesamtPreis :: [(Cent, Int)] -> Cent
3 gesamtPreis [] = 0
4 gesamtPreis ((p, a):xs) = a*p + (gesamtPreis xs)
```

4. Mit den zuvor entwickelten Funktionen ist diese Teilaufgabe schnell gelöst.

```
1 -- Berechnet die Gesamtkosten eines Einkaufs
2 kosten :: PreisListe -> EinkaufsListe -> Cent
3 kosten ps es = gesamtPreis (preisMenge ps es)
```

Aufgabe 2.5.

1. Implementieren Sie die Funktion `insertSorted`.
2. Geben sie eine Funktion `iSort` an, die eine Liste nach dem obigen Verfahren unter Benutzung von `insertSorted` sortiert.

Lösung 2.5.

1. Bei diesem Lösungsvorschlag ist zu beachten, dass die Reihenfolge der schon sortierten Elemente vertauscht wird. Wenn in Zeile 5 die Relation `(<=)` verwendet wird, findet kein Vertauschen statt.

```
1 -- Elemente an der richtigen Stelle in eine
2 -- sortierte Liste einfügen
3 insertSorted :: Ord a => a -> [a] -> [a]
4 insertSorted e [] = [e]
5 insertSorted e (x:xs)   | e < x = e:x:xs
6                         | otherwise = x:(insertSorted e xs)
```

2. Siehe die Beschreibung des Algorithmus.

```
1 -- Sortieren einer Liste (insertion sort)
2 iSort :: Ord a => [a] -> [a]
3 iSort [] = []
4 iSort (x:xs) = insertSorted x (iSort xs)
```

Aufgabe 2.6.

1. Implementieren Sie eine weitere Funktion höherer Ordnung mit dem Namen `filter'` und der Signatur `filter' :: (a -> Bool) -> [a] -> [a]`. In der Ausgabeliste sollen nur noch die Elemente vorkommen, für die das übergebene Prädikat `True` ist.
2. Geben Sie eine Implementierung für die Funktion `dropWhile'` an, die vom Anfang einer Liste alle Elemente entfernen, solange das übergebene Prädikat zutrifft. Oder implementieren Sie die Funktion `takeWhile'`, welche das Anfangsstück einer Liste liefert, für dessen Elemente das Prädikat zutrifft. Die Signatur ist wie oben jeweils `:: (a -> Bool) -> [a] -> [a]`.

Lösung 2.6.

1. Wenn das Prädikat zutrifft, wird das Element beibehalten, sonst verschwindet es. Das Prädikat wird in Zeile 3 und 4 mit `p` identifiziert und kann wie jede andere Funktion auf der rechten Seite verwendet werden.

```haskell
-- Filtern einer Liste anhand eines Prädikats
filter' :: (a -> Bool) -> [a] -> [a]
filter' p [] = []
filter' p (x:xs) = o ++ (filter' p xs)
       where o = if (p x)      then [x]
                               else []
```

2. Es sind für die Lösung dieser Aufgabe keine neuen Konzepte nötig.

```haskell
-- Elemente vom Anfang entfernen, solange Prädikat gilt
dropWhile' :: (a -> Bool) -> [a] -> [a]
dropWhile' p [] = []
dropWhile' p (x:xs) = if (p x) then
                        dropWhile' p xs
                      else
                        x:xs

-- Elemente beibehalten, solange Prädikat zutrifft
takeWhile' :: (a -> Bool) -> [a] -> [a]
takeWhile' p [] = []
takeWhile' p (x:xs) = if (p x) then
                        x:(takeWhile' p xs)
                      else
                        []
```

Aufgabe 2.7. Geben Sie die Funktionen `doubleList` und `squareList` an, welche alle Elemente einer Liste verdoppeln bzw. quadrieren. Benutzen Sie für die tatsächlichen Rechenoperationen anonyme Funktionen und für den Rest `map`.

Lösung 2.7. Die beiden Funktionen sind wie folgt definiert:

```haskell
-- Alle Elemente einer Liste verdoppeln
doubleList :: Num a => [a] -> [a]
doubleList = map (\ x -> 2*x)

-- Alle Elemente einer Liste quadrieren
squareList :: Num a => [a] -> [a]
squareList = map (\ x -> x*x)
```

Aufgabe 2.8. Geben Sie mit Hilfe der Listenkomprehension eine Funktion an, die aus zwei Listen mit Zahlen eine Liste mit Paaren erstellt. Es sollen nur solche Paare vorkommen, bei denen die Zahl aus der ersten Liste durch die aus der zweiten teilbar ist.

Lösung 2.8. Die folgende Lösung arbeitet mit Listenkomprehension. Es ist aus diesem Grund hier nicht möglich, für beide Listen unendliche zu verwenden. Wegen der Tiefenauswertung, muss die zweite Liste erst komplett abgearbeitet werden, bevor das nächste Element der ersten Liste betrachtet werden kann. Es ist aber ohne Probleme möglich, die erste Liste unendlich lang zuzulassen.

```
1 -- Siehe Aufgabenbeschreibung: Aufgabe 2.8
2 teilbarePaare :: Integral a => [a] -> [a] -> [(a,a)]
3 teilbarePaare xs ys = [(x,y) | x <- xs, y <- ys, mod x y == 0]
```

3 Lösungsvorschlag zu Blatt 3

Aufgabe 3.1. Implementieren Sie die Funktion `ask :: String -> IO String`, welche die im ersten Parameter übergebene Frage ausgibt und eine eingelesene Zeile als Antwort liefert.

Lösung 3.1.

```
1 ask :: String -> IO String
2 ask frage = do
3         print frage
4         getLine
```

Die Frage wird in einem `do`-Block ausgegeben und danach wird die nächste Zeile vom Benutzer eingelesen. Die Einhaltung dieser Reihenfolge kann entscheidend sein :-).

Aufgabe 3.2.

1. Geben Sie eine Funktion, die eine Liste aus den Paaren `(String, Int)` in eine Datei speichert, nach deren Pfad per `ask` gefragt wird.

2. Schreiben Sie eine Funktion, mit der die gespeicherten Daten wieder gelesen werden können.

Lösung 3.2.

1. Ein Lösungsvorschlag findet sich in dem folgenden Listing.

```
1 speicherePaare :: [(String, Int)] -> IO ()
2 speicherePaare list = do
3                 filename <- ask "Wie␣soll␣die␣Datei␣heissen?"
4                 writeFile filename (show list)
```

2. Mit der Funktion `readFile` können alle Daten aus der Datei gelesen werden. Diese liegen nach dem Lesen jedoch als `String` vor und müssen erst wieder in den passenden Datentypen umgewandelt werden. Dies ist mit der Funktion `readIO` möglich.

```haskell
lesePaare :: IO [(String, Int)]
lesePaare = do
        filename <- ask "Wo kommen die Daten her?"
        inhalt <- readFile filename
        readIO inhalt
```

Aufgabe 3.3. Geben Sie eine Funktion, die für eine konkrete `Figur` den Mittelpunkt dieser berechnet.

Lösung 3.3. Alle Figuren werden nach dem Prinzip des Mustervergleichs unterschiedlich behandelt. Die Behandlung des `Kreis`es ist ziemlich einfach und bei dem `Rechteck` fällt auf, dass sich die Berechnung wie bei der `Strecke` verhält.

```haskell
mitte :: Figur -> Punkt
mitte (Kreis m _) = m
mitte (Rechteck (x1, y1) (x2, y2)) =
                ((x1 + x2) / 2, (y1 + y2) / 2)
mitte (Strecke p1 p2) = mitte (Rechteck p1 p2)
```

Aufgabe 3.4. Geben Sie eine Funktion, die für ein Element vom Typ `T` feststellt, ob es in einer Menge vom Typ `Set T` enthalten ist.

Lösung 3.4. Es ist wieder einmal wichtig, dass der Datentyp der Inhalte der Menge die Vergleichsoperation `(==)` unterstützt. Das gesuchte Element wird einfach rekursiv gesucht.

```haskell
isIn :: Eq a => Set a -> a -> Bool
isIn NilSet x = False
isIn (ConsSet y xs) x = if (x == y)
        then True
        else isIn xs x
```

Aufgabe 3.5.

1. Definieren Sie einen parametrischen Datentypen `Vielleicht a`, der entweder `Nichts` ist oder ein Wert vom Typ `a` enthält.

2. Schreiben Sie ein Prädikat zur Prüfung, ob `Vielleicht a` vielleicht `Nichts` ist.

3. Geben Sie eine Funktion, mit der der Wert von `Vielleicht a` bestimmt werden kann. Falls es `Nichts` ist, benutzen Sie `error`.

Lösung 3.5.

1. Die Definition lautet `data Vielleicht a = Nichts |Einfach a`. Es ist sicher sinnvoll sein, einige Typklassen wie `Eq` und `Show` abzuleiten.

2. ...

```
1 istWas :: Vielleicht a -> Bool
2 istWas Nichts = False
3 istWas (Wert _) = True
```

3. ...

```
1 getWert :: Vielleicht a -> a
2 getWert Nichts = error "Da␣ist␣'Nichts'."
3 getWert (Wert w) = w
```

Aufgabe 3.6.

```
1 -- Eine Assoziationsliste mit Schluessel und Wert
2 type AssocList key value = [(key, value)]
3
4 lookUp :: Eq a => AssocList a b -> a -> b
5 lookUp [] key = error "Schluessel␣nicht␣gefunden."
6 lookUp ((k, v):as) key = if (k == key)
7                          then v
8                          else lookUp as key
```

Listing 3.1: Finden von Elementen in Assoziationslisten

Ändern Sie die gegebene Definiton von `lookUp` so ab, dass auch `Nichts` zurückgegeben werden kann.

Lösung 3.6.

```
1 lookUp :: Eq a => AssocList a b -> a -> Vielleicht b
2 lookUp [] key = Nichts
3 lookUp ((k, v):as) key = if (k == key)
4                          then Wert v
5                          else lookUp as key
```

Listing 3.2: Finden von Elementen in Assoziationslisten

4 Lösungsvorschlag zu Blatt 4

Aufgabe 4.1.

1. Definieren Sie einen parametrischen, rekursiven Datentypen für einen binären Baum, der Werte verschiedener Datentypen in den Knoten halten kann.
2. Erstellen Sie per Hand sortierte Bäume, die die Werte „AUTO“, „BOOT“, „HAUS“, „KAMEL“ und „TIER“ enthalten. Ein Baum soll dabei die minimale Tiefe und einer die maximale Tiefe besitzen.
3. Geben Sie ein Prädikat an, ob ein gegebenes Element in dem Baum vorhanden ist (Klasse `Ord` und `Eq` benötigt).
4. Entwickeln Sie eine Funktion, die Elemente sortiert in einen Baum einfügt.

Lösung 4.1.

1. Es wird der Datentyp `Baum` definiert, für den die Klassen `Show` und `Eq` abgeleitet werden.

```
1 data Baum a = Blatt
2         | Knoten a (Baum a) (Baum a)
3         deriving (Show, Eq)
```

2. Die Definition eines Baumes beginnt jeweils mit dem Wurzelknoten, der die beiden abgehenden Teilbäume oder Blätter enthält. Für einen Baum mit maximaler Tiefe muss ein entarteter Baum gewählt werden, in dem jeder Knoten maximal einen Nachfolger besitzt, der kein Blatt ist.

```
1 Knoten "AUTO" Blatt
2  (Knoten "BOOT" Blatt
3   (Knoten "HAUS" Blatt
4    (Knoten "KAMEL" Blatt
5     (Knoten "TIER" Blatt Blatt)
6    )
7   )
8  )
```

   Bei einem Baum mit minimaler Tiefe muss versucht werden, möglichst viele Ebenen (gemessen am Abstand zur Wurzel) aufzufüllen.

```
Knoten "HAUS"
  (Knoten "BOOT"
    (Knoten "AUTO" Blatt Blatt)
    Blatt
  )
  (Knoten "KAMEL"
    Blatt
    (Knoten "TIER" Blatt Blatt)
  )
```

Es gibt für beide Bäume mehrere Varianten. Es muss jedoch darauf geachtet werden, dass die Elemente richtig sortiert sind. In den obigen Beispielen werden die Elemente beim Absteigen (in Richtung der Wurzel) nach rechts größer.

3. Die Abbruchbedingung für die Suche ist das Finden eines Blattes. Wenn der Baum nicht sortiert ist, muss überall nach dem Element gesucht werden.

```
istDrin :: Eq a => Baum a -> a -> Bool
istDrin Blatt _ = False
istDrin (Knoten x b1 b2) y
        | x == y               = True
        | istDrin b1 y  = True
        | istDrin b2 y  = True
        | otherwise     = False
```

Ist der Baum hingegen sortiert, so muss maximal der Weg bis zu einem Blatt durchlaufen werden:

```
istDrin :: Eq a => Baum a -> a -> Bool
istDrin Blatt _ = False
istDrin (Knoten x b1 b2) y
        | x == y               = True
        | y > x                = istDrin b2 y
        | otherwise            = istDrin b1 y
```

4. Die hier angegebene Funktion fügt ein Element auch dann in den Baum ein, wenn es schon darin vorhanden ist. Alle größeren Elemente werden jeweils in den rechten Teilbaum einsortiert. Gleichgroße (also hier auch gleiche) Elemente werden jeweils in den linken Teilbaum eingefügt. Ein Einfügen findet durch das Ersetzen eines Blattes statt.

```
einfuegen :: (Eq a, Ord a) => Baum a -> a -> Baum a
einfuegen Blatt y = Knoten y Blatt Blatt
```

```
3 einfuegen (Knoten x b1 b2) y
4         | x < y         = Knoten x b1 (einfuegen b2 y)
5         | otherwise     = Knoten x (einfuegen b1 y) b2
```

Durch diese Funktion lassen sich die oben geforderten Bäume auch einfacher aufschreiben. Die Reihenfolge des Einfügens bestimmt das „Aussehen“ des Baumes.

```
1 -- Baum minimaler Tiefe
2 einfuegen (einfuegen (einfuegen (einfuegen (einfuegen
3   Blatt "HAUS")  "BOOT") "AUTO") "KAMEL") "TIER"
4
5 --Baum maximaler Tiefe
6 einfuegen (einfuegen (einfuegen (einfuegen (einfuegen
7   Blatt "AUTO") "BOOT") "HAUS") "KAMEL") "TIER"
```

Aufgabe 4.2.

1. Geben Sie eine Funktion an, die überprüft, ob ein Wort in einem gegebenen `Trie` vorhanden ist.

2. Entwickeln Sie eine Funktion, die Zeichenketten in einen `Trie` einfügt.

Lösung 4.2.

1. Es gibt unterschiedliche Fälle, die betrachtet werden müssen:

```
1  isIn :: Trie -> String -> Bool
2  -- Ende des Wortes gefunden
3  isIn (TrieNode True _) [] = True
4  -- Baum zu Ende, Wort nicht
5  isIn (TrieNode _ []) _ = False
6  -- Pfad des Wortes finden
7  isIn (TrieNode b ((c, t):ts)) (x:xs)
8          -- Pfad gefunden und absteigen
9          | c == x        = isIn t xs
10         -- Pfad nicht gefunden, weiter suchen
11         | otherwise = isIn (TrieNode b ts) (x:xs)
```

In Zeile 3 wird geprüft, ob das Wort zu Ende ist und der aktuelle Knoten das Ende eines Wortes markiert.

In Zeile 5 sind mehrere unbekannte Variablen auf der linken Seite. Einer der Fälle wurde durch Zeile 3 abgedeckt. Es bleiben noch drei Fälle übrig:

- Das Wort ist zu Ende und der Knoten markiert kein Ende.

- Das Wort ist nicht zu Ende, aber der Baum endet.
- Das Wort ist nicht zu Ende, aber keines der Kinder des Knoten passt zu dem Wort (nur erreichbar von Zeile 11).

Wenn ein Kind des Knoten gefunden wird, dessen Pfad mit dem aktuellen Buchstaben markiert ist, wird die Suche eine Ebene tiefer weiter fortgesetzt (Zeile 9). Sonst wird unter den weiteren Kindern des Knotens weiter gesucht (Zeile 11).

2. Ähnlich der Suche läuft auch das Einfügen von Worten in den `Trie` ab.

```
1  insert :: Trie -> String -> Trie
2  -- Notfalls doppelt eingefügt
3  insert (TrieNode _ ts) [] = TrieNode True ts
4  -- Baum zu Ende also Erweiterung
5  insert (TrieNode b []) (x:xs) =
6         TrieNode b [(x, insert (TrieNode False []) xs)]
7  -- Baum durchsuchen für Einfügepunkt
8  insert (TrieNode b ((c, t):ts)) (x:xs)
9          -- Pfad im Baum absteigen
10         | c == x        = TrieNode b ((c, insert t xs):ts)
11         -- Parallel weiter bewegen
12         | otherwise = keepPath (c, t) (insert (TrieNode b ts) (x:xs
               ))
13         where
14                 keepPath path (TrieNode isEnd paths) = TrieNode
                       isEnd (path:paths)
```

Die Unterschiede sind jedoch, dass der Baum erweitert werden muss, wenn nicht weiter abgestiegen werden kann, das Wort aber noch nicht zu ende ist (Zeile 5 und 6).

Eine weitere Abweichung zu dem oberen Listing findet sich in Zeile 12. Hier dürfen bei der Suche nach dem passend markierten Pfad nicht einfach alle anderen Pfade „vergessen“ werden, wie dies bei der Suche der Fall war. Durch die lokal definierte Funktion `keepPath` wird der „vergessene“ Pfad einfach nachträglich wieder an den Knoten angehängt.

Aufgabe 4.3. Geben Sie eine Funktion zur Bestimmung der Menge der gespeicherten Wörter in einem `Trie` an.

Lösung 4.3. Jedes gefundene Blatt steht für ein Wort (Zeile 5), da nach der informellen Definition des `Tries` keine überflüssigen Knoten (solche die sich nach dem Ende des letzten Wortes im Teilbaum befinden) enthalten sein dürfen.

```
1  count :: Trie -> Int
2  -- Fall, der nur durch Konstruktion des Algorithmus vorkommen kann
3  count (TrieNode False []) = 0
4  -- Normalfall bei einem Blatt
5  count (TrieNode True []) = 1
6  -- Alle Wörter im Knoten und Kindern zählen
7  count (TrieNode b ((_, t):ts)) =
8          (if b then 1 else 0)
9          + count t
10         + count (TrieNode False ts)
```

Jeder Knoten, der ein Wort, enthält wird gezählt (in Zeile 8 wird geprüft, ob ein Wort an diesem Knoten endet). Zusätzlich werden alle Worte in dem ersten Teilbaum (Zeile 9) und die in den restlichen Teilbäumen (Zeile 10) gezählt. In Zeile 10 erhält der aktuelle Knoten den Wert `False`, damit ein möglicherweise dort endendes Wort nur einmal gezählt wird.

Aufgabe 4.4. Schreiben Sie eine Funktion zum Löschen von Einträgen aus einem `Trie`.

Lösung 4.4. Die Lösung dieser Aufgabe bleibt dem aufmerksamen Leser überlassen.