Prof. Dr. E.-E. Doberkat

Christian Bockermann
Jan Bessai
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 1

Installieren Sie die Haskell-Platform (`http://www.haskell.org/platform/`) auf ihrem Rechner. Stellen Sie zudem sicher, dass `ghci` zu ihrer Pfadvariablen hinzugefügt wird. Unter Windows finden Sie im Startmenü nach der Installation unter Programme -> Haskell Platform versionsnummer den Eintrag WinGHCi, der eine grafische Oberfläche für den Haskell Interpreter GHCi bereitstellt.

Aufgabe 1.1 *GHCi-Einführung*

(a) Öffnen Sie den Texteditor Ihrer Wahl und tippen Sie folgendes Programm ab:

```
add :: Int -> Int -> Int
add x y = x + y
```

Die Funktion `add` addiert zwei ganze Zahlen.

(b) Speichern Sie das Programm in einer Datei mit der Endung `.hs`. Den Pfad zu der Datei nennen wir im Folgenden `file`.

(c) Öffnen Sie die Kommandozeile und laden Sie die Datei mit dem interaktiven Modus des GHC (GHCi genannt), wie folgt: `ghci file`
Sie sollten nun die folgende Ausgabe erhalten:

```
[1 of 1] Compiling Main             ( file, interpreted )
Ok, modules loaded: Main.
*Main>
```

(d) Rufen Sie nun die Funktion `add` auf, indem Sie `add 1 2` eingeben und mit ENTER bestätigen. Das Ergebnis wird ausgegeben und Sie können weitere Funktionsaufrufe auswerten lassen. Um die Funktion `add` mehrfach zu nutzen, müssen die Parameter geklammert werden, z.B: `add 3 (add 5 7)`.

Folgende Kommandos des GHCi haben sich als nützlich erwiesen:

- `:load file` (kurz `:l`) läd die Datei `file` in den GHCi.
- `:reload` (kurz `:r`) läd die aktuelle Datei neu ein. Nachdem Änderungen an dem Quelltext vorgenommen wurden, kann die aktuelle Datei mit `:r` leicht neu geladen werden.
- `:type ausdruck` (kurz `:t`) zeigt den Typ des Ausdruckes `ausdruck` an, z.B. `:t add` oder `:t add 1 2`.
- `:help` (kurz `:h`) öffnet die Hilfe mit weiteren nützlichen Befehlen.
- `:quit` (kurz `:q`) beendet den GHCi.

Aufgabe 1.2 *Fehlermeldungen des GHCi* Die folgende Aufgabe enthält eine Reihe von fehlerhaften Haskell-Ausdrücken. Ziel dieser Aufgabe ist, dass Sie sich mit den Fehlermeldungen des GHCi vertraut machen. Laden Sie dazu die Datei aus Aufgabe 1 und interpretieren Sie die folgenden Ausdrücke mit dem GHCi. Versuchen Sie die Fehlermeldungen nachzuvollziehen.

(a) `add 3 '2'`

(b) `add 3 2 1`

(c) `Add 3 2`

(d) `ad 3 2`

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 2

Aufgabe 2.1 *Konstanten*

Gegeben sei die Funktion `add2` von den Folien (dort `r` genannt) und die konstante Funktion `one`.

```
add2 :: Int -> Int
add2 = (+) 2

one :: Int
one = 1
```

(a) Definieren Sie eine Konstante `k :: Int`, deren Auswertung die Zahl `11` liefert. Nutzen Sie dabei ausschließlich die Funktionen `add2` und `one`.

(b) Definieren Sie eine Funktion `add6 :: Int -> Int`, die eine ganze Zahl als Parameter erhält und die Summe aus 6 und der Zahl berechnet. Nutzen Sie dafür ausschließlich die Funktion `add2`.

Aufgabe 2.2 *(`$`)-Operator*

Machen Sie sich mit dem sogenannten Dollar-Operator `($) :: (a -> b) -> a -> b` vertraut.

(a) Definieren Sie die Konstante `k` aus Aufgabe 1.*a*) mit Hilfe des Dollar-Operators.

(b) Definieren Sie die Funktion `add6` aus Aufgabe 1.*b*) mit Hilfe des Dollar-Operators.

Aufgabe 2.3 *Partielle Funktionsanwendung*

Gegeben sei die folgende Funktion `add4Ints`:

```
add4Ints :: Int -> Int -> Int -> Int -> Int
add4Ints v w x y = v + w + x + y
```

(a) Die Funktionsanwendung in Haskell ist per Definition bekanntermaßen linksassoziativ. Erweitern Sie dem Ausdruck `add4Ints 1 2 3 4` um die implizit vorhandenen Klammern.

(b) Linksassoziativität der Funktionsanwendung hat direkt Folgen für den Typ von Funktionen mit mehreren Argumenten. Wie sieht die explizite Klammerung für den Typ der Funktion `add4Ints` (also `Int -> Int -> Int -> Int -> Int`) aus?

Hinweis: Die Explikation der Klammerung erhält natürlich die Semantik des Ausdrucks.

technische universität dortmund

Fakultät für Informatik
Lehrstuhl für Software-Technologie

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 2

```
module Loesung02 where

add2 :: Int -> Int
add2 = (+) 2

one :: Int
one = 1

-- 2.1.a
k :: Int
k = add2 ( add2 ( add2 (add2 (add2 one))))

-- 2.1.b
add6 :: Int -> Int
add6 x = add2 (add2 (add2 x))


-- 2.2.a
k_dollar :: Int
k_dollar = add2 $ add2 $ add2 $ add2 $ add2 one

-- 2.2.b
add6_dollar :: Int -> Int
add6_dollar x = add2 $ add2 $ add2 x


-- 2.3.a
--((((add4Ints 1) 2) 3) 4)

--2.3.b
-- Int -> (Int -> (Int -> (Int -> Int)))
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 3

Aufgabe 3.1 *Vorfahrtsregeln*

Gegeben sei eine Funktion `add3` die folgendermaßen definiert ist:

```
add3 x = (+) 3 x
```

(a) Die Ableitung des Funktionstyps sollte keine Schwierigkeiten bereiten. Die Funktion wird in folgendem Kontext aufgerufen:

```
add3 3 * 4
```

Was ist das Ergebnis dieses Aufrufs - und warum?

(b) Nun wird die Funktion leicht verändert aufgerufen und zwar:

```
add3 $ 3 * 4
```

Wo ist der Unterschied zum vorherigen Aufruf? Welches Ergebnis liefert dieser Aufruf zurück und warum?

Aufgabe 3.2 *Auswertung anonymer Funktionen*

Reduzieren Sie die folgendenden Ausdrücke. Geben Sie dabei auch die Zwischenergebnisse an.

(a) `(\x -> 4 * x) $ (\y -> 3 + y) 1`

(b) `(\x -> 4 + x + (\x -> 3 + x) 3) 1`

(c) `(\f -> f (f 2)) (*3)`

(d) `(\x f -> x + f x ) 5 (*3)`

Aufgabe 3.3 *Klassische Logik mit anonymen Funktionen*

Im folgenden soll gezeigt werden, dass mit anonymen Funktionen klassische Logik (1. Ordnung) realisiert werden kann.
Hierzu werden zunächst die beiden Ausdrücke

```
true = (\ x y -> x)
false = (\ x y -> y)
```

definiert.

(a) Bestimmen sie die Typen der Ausdrücke `true` und `false`.

Es wird nun der Implikationsoperator als logische Verknüpfung definiert:

```
(==>) = (\ x y -> x y true)
```

(b) Bestimmen sie den Typ von `(==>)`.

(c) Wenden sie `(==>)` auf alle Kombinationen von `true` und `false` an, und erstellen sie eine Wertetabelle, um zu verifizieren, dass es sich wirklich um die Implikation handelt.

(d) Zeigen sie, dass mit λ-Ausdrücken alle klassichen logischen Verknüpfungen definierbar sind (**Hinweis:** Informieren sie sich darüber, welche Funktionen man hierfür mindestens definieren muss).

technische universität dortmund

Fakultät für Informatik
Lehrstuhl für Software-Technologie

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 3

Aufgabe 3.1 *Vorfahrtsregeln*

(a) Addition und Multiplikation

```
add3 :: (Num a) => a -> a
add3 x = (+) 3 x

add3 3 * 4
 = ((+) 3 3) * 4
 = 6 * 4
 = 24
```

(b) Addition mit Dollar

```
add3 $ 3 * 4
 = add3 $ 12
 = 15
```

Aufgabe 3.2 *Auswertung anonymer Funktionen*

(a) Mit Dollar

```
(\ x -> 4 * x) $ (\ y -> 3 + y) 1
 = (\ x -> 4 * x) $ 3 + 1
 = (\ x -> 4 * x) 4
 = 4 * 4
 = 16
```

(b) Verschachtelt

```
(\ x -> 4 + x + (\ x -> 3 + x) 3) 1
 =  4 + 1 + (\ x -> 3 + x) 3
 = 5 + (3 + 3)
 = 5 + 6
 = 11
```

(c) Mit Funktionsargument

```
(\ f -> f (f 2)) (*3)
 = (*3) ((*3) 2)
 = (*3) (6)
 = 6 * 3
 = 18
```

(d) Mit mehreren Argumenten

```
(\ x f -> x + f x) 5 (*3)
 = (\ f -> 5 + f 5) (*3)
 = 5 + (*3) 5
 = 5 + 15
 = 20
```

Aufgabe 3.3 *Klassische Logik mit anonymen Funktionen*

(a) Typen von Wahrheitswerten

```
true :: t -> f -> t
true = (\ x y -> x)
false :: t -> f -> f
false = (\ x y -> y)
```

(b) Implikationstyp

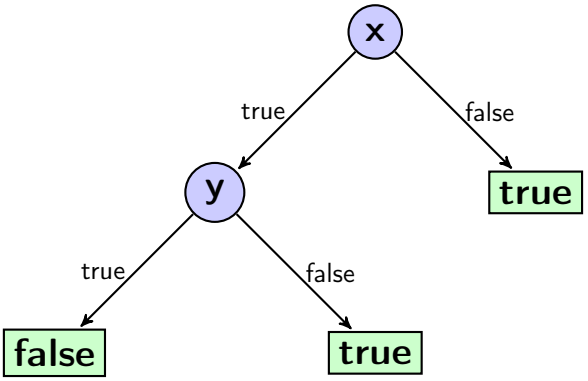
```
(==>) :: (a -> (t -> f -> t) -> r) -> a -> r
(==>) = (\ x y -> x y true)
```

(c) Wertetabelle

```
false ==> false
 = (\ x y -> x y true) false false
 = false false true
 = (\ x y -> y) false true
 = (\ y -> y) true = true

false ==> true
 = (\ x y -> x y true) false true
 = false true true
 = (\ x y -> y) true true
 = (\ y -> y) true
 = true

true ==> false
 = (\ x y -> x y true) true false
 = true false true
 = (\ x y -> x) false true
 = (\ y -> false) true
 = false

true ==> true
 = (\ x y -> x y true) true true
 = true true true
 = (\ x y -> x) true true
 = (\ y -> true) true
 = true
```

(d) NAND-Vollständigkeit

OBDD:



Teilbäume links und rechts in x und y einsetzen ergibt:

```
nand ::
  (r' -> (t -> f -> t) -> r)
  -> ((t -> f -> f) -> (t -> f -> t) -> r')
  -> r
nand = (\ x y -> x (y false true) true)
```

Ausfürhlich:

```
nand false
 = (\ x y -> x (y false true) true) false
 = (\ y -> false (y false true) true)
 = (\ y -> true)

nand true false
 = (\ x y -> x (y false true) true) true false
 = (\ y -> true (y false true) true) false
 = (\ y -> y false true) false
 = false false true
 = true

nand true true
 = (\ x y -> x (y false true) true) true true
 = (\ y -> true (y false true) true) true
 = (\ y -> y false true) true
 = true false true
 = false
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 4

Aufgabe 4.1 *Listen*

(a) Definieren Sie eine Haskell-Funktion `removeOddValues :: [Int] -> [Int]`, die alle ungeraden Listenelemente entfernt.

(b) Definieren Sie eine Funktion `collapse :: (a -> a -> a) -> a -> [a] -> a`, die einen kommutativen Operator `op :: a -> a -> a`, einen Startwert vom Typ `e :: a` und eine Eingabeliste vom Typ `xs :: [a]` erhält. Für die leere Liste soll der Startwert `e` zurückgegeben werden, während bei einer nichtleeren Liste der Operator auf das erste Listenelement und den Wert von `collapse` für die Restliste angewandt wird.

Beispielsweise sollte der Ausdruck `collapse (+) 0 [1,2,3,4]` zu dem Wert `10` reduzieren.

Aufgabe 4.2 *Unendliche Listen*

(a) Definieren Sie eine unendliche Liste `squares :: [Integer]`, die alle Quadratzahlen enthält. Hinweis: mit der Funktion `take :: Int -> [a] -> [a]` können Sie auf die ersten Listenelemente von Listen zugreifen – dies gilt auch für unendliche Listen.

(b) Definieren Sie eine unendliche Liste `triangle :: [[Int]]`, die sich zu einer Liste von Listen von der folgenden Form reduziert:

```
[]:[1]:[1,2]:[1,2,3]:[1,2,3,4]...
```

Hinweis: bei dieser Aufgabe könnte die Funktion `map :: (a -> b) -> [a] -> [b]` hilfreich sein.

Quizfragen:
Beantworten Sie die folgenden Fragen ohne Compiler-Unterstützung. Beachten Sie, dass die leere Liste `[]` vom Typ `[a]` und der sogenannte Cons-Operator `(:)` vom Typ `a -> [a] -> [a]` ist.

(a) Von welchem Typ ist der Ausdruck `[]:[[]]:[[[]]]`? Was ändert sich, wenn wir die Konstruktion unendlich oft fortführen?

(b) Was passiert, wenn die inneren Listen mit Werten gefüllt werden, also zum Beispiel der folgende Fall eintritt: `['a']:[['a']]:[[['a']]]`?

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 4

Aufgabe 4.1 *Listen*

(a)
```
removeOddValues :: [Int] -> [Int]
removeOddValues []     = []
removeOddValues (x:xs)
  | odd x     = removeOddValues xs
  | otherwise = x : removeOddValues xs
```

(b) Es existieren zwei mögliche Definitionen für `collapse`, die sich in der Anwendungsreihenfolge des Operators unterscheiden:

```
-- Operator linksassoziativ anwenden
collapseL :: (a -> a -> a) -> a -> [a] -> a
collapseL _  e []     = e
collapseL op e (x:xs) = collapseL op (op e x) xs

-- Operator rechtsassoziativ anwenden
collapseR :: (a -> a -> a) -> a -> [a] -> a
collapseR _  e []     = e
collapseR op e (x:xs) = op x $ collapseR op e xs
```

Aufgabe 4.2 *Unendliche Listen*

(a) Quadratzahlen

```
squares :: [Int]
squares = map (\x -> x*x) [1,2..]

-- list comprehension
squaresCompr :: [Int]
squaresCompr = [ x*x | x <- [1,2..]]

-- Rekursion
squaresRec :: [Int]
squaresRec = squaresRec' 1 where
  squaresRec' :: Int -> [Int]
  squaresRec' n = (n * n) : squaresRec' (n+1)
```

(b) `triangle`

```
triangle :: [[Int]]
triangle = map (\x -> [1..x]) [0,1..]

triangleCompr :: [[Int]]
triangleCompr = [ [1..x] | x <- [0,1..]]

triangleRec :: [[Int]]
triangleRec = triangleRec' 0 where
  triangleRec' :: Int -> [[Int]]
  triangleRec' 0 = []: triangleRec' 1
  triangleRec' n = [1..n]: triangleRec' (n+1)
```

Quizfragen:

(a) Der Typ von `[]:[[]]:[[[]]]` ist `[[[a]]]`. Der folgende Haskell-Ausdruck definiert die unendliche Fortsetzung dieser Konstruktion:

```
iterate (\x -> [x]) []
```

(b) Wir erhalten einen Typfehler, weil kein größtes Element existiert, das die maximale Verschachtelungstiefe beschränkt. Der Trick, dass das erste Element `[]` genau die leere Liste vom Typ `[[a]]` ist, funktioniert nur wenn endlich viele Listen verschachtelt werden.

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 5

Aufgabe 5.1 *Listen*

(a) Definieren Sie eine Funktion `deleteChar :: Char -> String -> String`, die eine neue Liste erzeugt, die keine Vorkommen des übergebenen Zeichens enthält.

(b) Welche Gemeinsamkeiten haben die Funktionen `removeOddValues` von Blatt 4 und die Funktion `deleteChar`? Lässt sich von einer gemeinsamen Struktur abstrahieren?

Aufgabe 5.2 *Komposition*

(a) Informieren Sie sich über die Funktionskomposition in Haskell. Wie ist der Typ der Funktionskomposition und worin bestehen Unterschiede zum Dollar-Operator? Formulieren Sie dazu die folgende Lösung von Aufgabe 2.2.b) (Blatt 2) unter Verwendung der Komposition neu:

```
add2 :: Int -> Int
add2 = (+2)

add6 :: Int -> Int
add6 x = add2 $ add2 $ add2 x
```

(b) Reduzieren Sie den folgenden Haskell-Ausdruck: `((\x -> 4 * x) . (\y -> 3 + y)) 1`

Aufgabe 5.3 *Listen*

(a) Schreiben Sie eine Haskell-Funktion `asciiQuersumme :: String -> Int`, die zuerst jedes Zeichen in seinen ASCII-Code überführt. Im Anschluss daran soll die Summe von allen ASCII-Codes berechnet werden.

Nutzen Sie zur Umformung eines Zeichens in seinen ASCII-Code die Funktion `ord :: Char -> Int`, die von dem Modul `Data.Char` angeboten wird. Sie binden das Modul mit dem Befehl `import Data.Char` am Anfang ihrer Quellcodedatei ein.

Zur Berechnung der Summe eignet sich die Funktion `collapse` von Blatt 4.

(b) Schreiben Sie eine Funktion, die eine Liste von als `String` abgespeicherten Studiengängen und eine Liste von als `Int` abgespeicherten Matrikelnummern erhält. Die Funktion soll zunächst die beiden Listen mit `zip` zusammenfügen, um die Zuordnung von Matrikelnummern zu Studiengängen zu erzeugen. Anschließend soll sie eine Liste zurückgeben, die alle Matrikelnummern enthält, denen der Studiengang Diplom zugeordnet wurde.

Beispiel:

```
studiengaenge :: [String]
studiengaenge = ["Bachelor","Diplom","Bachelor","Diplom","Diplom"]

matrikelnr :: [Int]
matrikelnr = [31,42,312,89,55,62]
```

Die Funktion soll beispielsweise für die Argumente `studiengaenge` und `matrikelnr` zu der Liste `[42,89,55]` reduzieren.

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 5

Aufgabe 5.1

(a)
```
deleteChar :: Char -> String -> String
deleteChar _ []     = []
deleteChar c (x:xs) | c==x      = deleteChar c xs
                    | otherwise = x : deleteChar c xs
```

(b)
```
removeOddValues :: [Int] -> [Int]
removeOddValues []     = []
removeOddValues (x:xs)
  | odd x     = removeOddValues xs
  | otherwise = x : removeOddValues xs

          filt :: (a -> Bool) -> [a] -> [a]
filt _ [] = []
filt p (x:xs) | p x       = filt p xs
              | otherwise = x : filt p xs

deleteCharF :: Char -> String -> String
deleteCharF char = filt (\x -> x==char)

removeOddValuesF :: [Int] -> [Int]
removeOddValuesF = filt odd
```

Funktion `filter :: (a -> Bool) -> [a] -> [a]` existiert bereits - jedoch mit umgekehrter Semantik, so dass sich die Lösungen mit `filter` wie folgt ergeben:

```
deleteCharFilter :: Char -> String -> String
deleteCharFilter char = filter (\x -> x/=char)

removeOddValuesFilter :: [Int] -> [Int]
removeOddValuesFilter = filter even
```

Aufgabe 5.2 *Komposition*

```
-- Definition der Komposition:
(.)      :: (b -> c) -> (a -> b) -> (a -> c)
f . g    = \ x -> f (g x)
```

(a)
```
add2 :: Int -> Int
add2 = (+2)

add6 :: Int -> Int
add6 = add2 . add2 . add2
```

(b)
```
     ((\x -> 4 * x) . (\y -> 3 + y)) 1
==> (\x -> 4 * x)  ( (\y -> 3 + y)  1)
==> 4 *            ( (\y -> 3 + y)  1)
==> 4 *             (3+1)
==> 16
```

Aufgabe 5.3 *Listen*

(a)
```
-- Funktion collapse ist von Blatt 4 bekannt
collapse :: (a -> a -> a) -> a -> [a] -> a
collapse _  e []     = e
collapse op e (x:xs) = op x $ collapse op e xs

asciiQuersumme :: String -> Int
asciiQuersumme xs = summe $ map ord xs

summe :: [Int] -> Int
summe xs = collapse (+) 0 xs

-- kürzere Variante mit Benutzung von Funktion sum
-- aus der Standardbibliothek
ascii :: String -> Int
ascii = sum . map ord
```

(b)
```
studiengaenge :: [String]
studiengaenge = ["Bachelor","Diplom","Bachelor","Diplom","Diplom"]

matrikelnr :: [Int]
matrikelnr = [31,42,312,89,55,62]

getDiplomer :: [String] -> [Int] -> [Int]
getDiplomer ss ms = map snd $ filter (\(s,_) ->s=="Diplom") $ zip ss ms
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 6

Aufgabe 6.1 *Reduktion*

```
f x y z = (take x . map (\a -> (a,y a)) . filter even)  z
```

(a) Bestimmen Sie den Typ der Funktion `f`.

(b) Verdeutlichen Sie sich das Verhalten bei der Reduktion dieser Funktion, indem Sie sich beispielhaft Werte für die Argumente überlegen.

Aufgabe 6.2 *Punktfreie Notation*

Betrachten sie den λ-Ausdruck `(\ x -> f x) 3`. Die bekannten Reduktionsregeln ergeben offensichtlich, dass der Ausdruck reduziert identisch ist zu `f 3`. Die benannte (gebundene) Variable x war also nicht zwingend erforderlich.
Wir wissen bereits, dass Funktionen in klassischer Notation oder in λ-Notation definiert werden können. Dementsprechend kann man die obige Beobachtung analog für benannte Funktionen machen:

```
f :: a -> b
f x = ...

g :: a -> b
g x = f x

(g x) 3 = g 3 = f 3
```

Lassen wir die gebundene Variable x weg, so erhalten wir

```
g :: a -> b
g = f
```

Diese Notation wird in der Literatur als /Pointfree/-Notation bezeichnet.

(a) Formen Sie die folgenden Ausdrücke in Pointfree-Notation um:

```
manyOdd :: [Int] -> [Bool]
manyOdd xs = map (\ x -> odd x) xs

plusDrei :: [Int] -> [Int]
plusDrei xs = map (\ x -> add 3 x) xs
```

(b) Implementieren sie eine Funktion `sum :: [Int] -> Int` unter verwendung der Funktion `collapse` von Übungsblatt 4, die eine Liste aufsummiert. Verwenden sie bei ihrer Implementierung die Pointfree-Notation.

Aufgabe 6.3 *Binärkodierung*

(a) Schreiben Sie eine Haskell-Funktion `stringToBinary :: String -> [Int]`, die eine Zeichenkette in ihre Binärkodierung basierend auf dem ASCII-Wert der Zeichen umformt. Die Binärkodierung wird als `Int`-Liste mit 0 und 1 als Werten repräsentiert. Gehen Sie davon aus, dass die Funktion `ord :: Char -> Int` nur Werte zwischen 0 und 127 liefert.

Um die Funktion `stringToBinary` später umkehren zu können, muss sichergestellt werden, dass jede Kodierung eines Zeichens mit führenden Nullen aufgefüllt wird, sofern die Kodierung eine kürzere Länge als 7 ($2^7 = 128$) hat. Die Länge der durch die Funktion `stringToBinary` erzeugten Listen ist somit immer durch 7 teilbar.

Beispiel:

```
stringToBinary "a␣b" ==> [1,1,0,0,0,0,1,0,1,0,0,0,0,0,1,1,0,0,0,1,0]
```

(b) Schreiben Sie eine Funktion `binaryToString :: [Int] -> String`, die genau die Umkehrfunktion zu `stringToBinary` darstellt. Folgende Bedingung muss also gelten:

```
binaryToString . stringToBinary = id
```

Die Bedingung `stringToBinary . binaryToString = id` gilt nicht nur für Listen, deren Länge ein Vielfaches von 7 ist.

Folgende Funktionen könnten sich bei dieser Aufgabe als hilfreich herausstellen:

```
mod :: Int -> Int -> Int -- Modulo-Operation
div :: Int -> Int -> Int -- ganzzahlige Division

ord :: Char -> Int       -- ASCII-Repräsentation eines Zeichens
chr :: Int -> Char       -- Umkehrfunktion zu ord
```

Die letzten beiden Funktionen finden Sie im Modul `Data.Char`, das Sie mit `import Data.Char` zu Beginn der Haskell-Datei importieren.

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 6

Aufgabe 6.1 *Typisierung und Reduktion*

(a)
```
f x y z = (take x . map (\a -> (a,y a)) . filter even) z

-- Annahme: even :: Int -> Bool
f :: Int -> (Int -> a) -> [Int] -> [(Int, a)]

-- Annahme: even :: Integral a => a -> Bool
f :: Integral t1 => Int -> (t1 -> t2) -> [t1] -> [(t1, t2)]
```

(b)
```
    f 0 id [0..]
==> (take 0 . map (\a -> (a,id a)) . filter even) [0..]
==> take 0 (map (\a -> (a,id a)) (filter even [0..]) ) -- Definition der Komposition
==> []                                                 -- Definition von take


    f 2 (+7)  [1,2,3,4]
==> (take 2 . map (\a -> (a,(+7) a)) . filter even) [0,1,2,3,4]
==> take 2 (map (\a -> (a,(+7) a)) (filter even [0,1,2,3,4]))
==> (0,(+7) 0) : take 1   (map (\a -> (a,(+7) a)) (filter even [1,2,3,4]))
==> (0,(+7) 0) : (2,(+7) 2) : take 0 (map (\a -> (a,(+7) a)) (filter even [3,4]))
==> (0,(+7) 0) : (2,(+7) 2) : []
==> [(0,7),(2,9)]

-- Angenommen wir entfernen den Teil mit filter aus der Definition:
    f 2 (+7)  [1,2,3,4]
==> (take 2 . map (\a -> (a,(+7) a))) [0,1,2,3,4]
==> take 2 (map (\a -> (a,(+7) a)) [0,1,2,3,4])
==> (0,(+7) 0) : take 1   (map (\a -> (a,(+7) a)) [1,2,3,4])
==> (0,(+7) 0) : (1,(+7) 1) : take 0   (map (\a -> (a,(+7) a)) [2,3,4])
==> (0,(+7) 0) : (1,(+7) 1) : []
==> [(0,7),(1,8)]
```

Aufgabe 6.2 *Punktfreie Notation*

(a)
```
manyOdd :: [Int] -> [Bool]
manyOdd = map odd

plusDrei :: [Int] -> [Int]
plusDrei = map (+3)
```

(b) `sum` in punktfreier Notation:
```
sum :: [Int] -> Int
sum = collapse (+) 0
```

Aufgabe 6.3 *Binärcodierung*

(a) Umformung einer Zeichenkette in ihre Binärkodierung, wobei jedes Zeichen mit einem Bitstring der Länge 7 repräsentiert wird.

```
-- überführt Zahl in Dezimalsystem in Bitstring
toBinary :: Int -> [Int]
toBinary i | i > 1     = toBinary (div i 2) ++ [i `mod` 2]
           | otherwise = [i `mod` 2]

-- füllt die Eingabeliste mit führenden Nullen auf, falls Länge kleiner als n
fill :: Int -> [Int] -> [Int]
fill n xs | length xs < n = take (n-length xs) [0,0..]  ++ xs
          | otherwise     = xs

stringToBinary :: String -> [Int]
stringToBinary = concatMap (fill 7 . toBinary . ord)

-- Alternative: erzeugt Repräsentation der gegebenen Länge (kein fill mehr nötig)
toBinary2 :: Int -> Int -> [Int]
toBinary2 stellen x = map (\ i -> (x `div` 2^i) `mod` 2) [stellen-1, stellen-2 .. 0]

-- mit alternativer toBinary Implementierung:
stringToBinary :: String -> [Int]
stringToBinary = concatMap (toBinary2 7 . ord)
```

(b) Umkehrfunktion zu `stringToBinary`.

```
binaryToString :: [Int] -> String
binaryToString = map (chr . fromBinary) . splitBuckets 7

-- zerlegt die Eingabeliste in Listen von Listen der Länge n
splitBuckets :: Int -> [Int] -> [[Int]]
splitBuckets _ [] = []
splitBuckets n xs = take n xs : splitBuckets n (drop n xs)

-- überführt Bitstring in Dezimalsystem
fromBinary :: [Int] -> Int
fromBinary is = sum $ zipWith (\p i -> i * 2^p) [0,1..] (reverse is)
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 7

Aufgabe 7.1 *Die Funktion `flip`*

(a) Welchen Typ hat die Funktion `flip`?

(b) Was ist der Typ von `flip take`?

(c) Wozu könnte die Funktion `flip` hilfreich sein?

Aufgabe 7.2 *Umkehren von Listen als Faltung*

In dieser Aufgabe soll eine Funktion zur Umkehrung einer Liste (Typ: `[a] -> [a]`) auf zwei verschiedene Arten definiert werden.

(a) Definieren Sie eine Funktion `revL :: [a] -> [a]`, indem Sie `foldl` nutzen.

(b) Definieren Sie eine Funktion `revR :: [a] -> [a]`, indem Sie `foldr` nutzen.

(c) Vergleichen Sie beide Implementierungen – welche würden Sie bevorzugen und warum?

Aufgabe 7.3 *Faltungen von Funktionslisten*

(a) Definieren Sie eine Konstante `fs :: [Int -> Int]` mit der folgenden Struktur: Abwechselnd wird die Summe und das Produkt gebildet. Diese Funktionen werden partiell auf die entsprechenden Elemente der Liste `[1,2..]` angewandt, so dass die ungeraden Elemente summiert und die gerade Elemente multipliziert. Die ersten Listenelemente sollen sich wie folgt ergeben:

```
[(+1),(*2),(+3),(*4),(+5),(*6),..]
```

Diese Liste kann als Beispieleingabe für die folgenden beiden Aufgabenteile benutzt werden.

(b) Schreiben Sie eine Funktion `foldLFunc :: [a -> a] -> a -> a`, die eine Liste von Funktionen von links auf einen Startwert anwendet. Nutzen Sie `foldl` für Ihre Implementierung. Beispiel:

```
foldLFunc [f,g,h] x ==> (h (g (f x)))
```

(c) Schreiben Sie eine Funktion `foldRFunc :: [a -> a] -> a -> a`, die eine Liste von Funktionen von rechts auf einen Startwert anwendet. Nutzen Sie `foldr` für Ihre Implementierung. Beispiel:

```
foldRFunc [f,g,h] x ==> (f (g (h x)))
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 7

Aufgabe 7.1 *Die Funktion `flip`*

```
flip :: (a -> b -> c) -> b -> a -> c

take :: Int -> [a] -> [a]

flip take :: [a] -> Int -> [a]
```

Die Funktion `flip` wird häufig genutzt, um Funktionen in punktfreier Notation zu definieren. Vergleiche dazu die folgenden beiden (semantisch äquivalenten) Funktionen `f` und `g`:

```
f :: Int -> [Int]
f x = take x [1,2..]

g :: Int -> [Int]
g = flip take [1,2..]
```

Aufgabe 7.2 *Listen umkehren mit Faltungen*

(a)
```
revFoldl :: [a] -> [a]
revFoldl = foldl (\xs x -> x : xs) []

-- alternativ mit flip
revFoldl = foldl (flip (:)) []
```

(b)
```
revFoldr :: [a] -> [a]
revFoldr = foldr (\x xs -> xs++[x]) []
```

(c) Die Funktion `revFoldl` ist effizienter, weil die Operation `(:)` konstante Zeit benötigt, während `xs++[x]` die komplette Liste `xs` durchlaufen muss.

Aufgabe 7.3 *Faltungen von Funktionsanwendungen*

(a)
```
-- [(+1),(*2),(+3),(*4),(+5),(*6),..]
fs :: [Int -> Int]
fs =  zipWith (\op x -> op x) ops [1,2..]

-- kurz:
fs = zipWith ($) ops [1,2..]

ops :: [Int -> Int -> Int]
ops = [(+),(*)]++ops

-- alternative
ops = cycle [(+),(*)]

-- cycle ist eine Funktion aus der Haskell-Prelude
cycle :: [a] -> [a]
cycle []       = []
cycle xs@(_:_) = xs ++ cycle xs
```

(b)
```
-- Gewünschtes Verhalten:
foldLFunc [f,g,h] x ==> (h (g (f x)))

foldLFunc :: [a -> a] -> a -> a
foldLFunc fs x = foldl (\x f -> f x) x fs

-- alternativ:
foldLFunc fs x = foldl (flip ($)) x fs
```

(c)
```
-- Gewünschtes Verhalten:
foldRFunc [f,g,h] x ==> (f (g (h x)))

foldRFunc :: [a -> a] -> a -> a
foldRFunc fs x = foldr (\f x -> f x) x fs

--alternativ:
foldRFunc fs x = foldr ($) x fs
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 8

Aufgabe 8.1 *Modellierung*

(a) Modellieren Sie einen Datentyp mit den folgenden Eigenschaften. Definieren Sie geschickt eigene Typen, um die Modellierung besser zu strukturieren.

- Eine Firma hat mehrere Abteilungen.
- Jede Abteilung hat einen Manager und mehrere Mitarbeiter.
- Manager und Mitarbeiter haben einen Namen und ein Wert für das monatliche Gehalt.

(b) Geben Sie eine nichttriviale Beispielinstanz des Datentyps an.

(c) Schreiben Sie eine Funktion, die einen Wert des Typs Firma erhält und das Gehalt der Mitarbeiter verdoppelt.

Aufgabe 8.2 *Binärbäume*

(a) Definieren Sie einen rekursiven Datentypen `BinTree` für Binärbäume nach den folgenden Regeln: Ein Binärbaum ist entweder leer oder ein Knoten mit einem `Int`-Wert und zwei Binärbäumen als Nachfolger.

(b) Schreiben Sie eine Haskell-Funktion `sumAll :: BinTree -> Int`, die die Summe aller Werte in dem Baum berechnet.

(c) Schreiben Sie eine Haskell-Funktion `preorder :: BinTree -> [Int]`, die den Baum in Preorder-Reihenfolge durchläuft und die Werte der Knoten in dieser Reihenfolge zurückgibt.

(d) Schreiben Sie eine Haskell-Funktion `mapBinTree :: (Int -> Int) -> BinTree -> BinTree`, die die übergebene Funktion an jedem Knoten anwendet.

(e) Schreiben Sie eine Haskell-Funktion `holdsEverywhere :: (Int -> Bool) -> BinTree -> Bool`, die überprüft, ob das Prädikat an jedem Knoten gilt.

(f) Schreiben Sie eine Faltungsfunktion

```
foldBinTree :: (Int -> a -> a -> a) -> a -> BinTree -> a
```

so dass der erste Parameter das Verhalten für innere Knoten festlegt, während der zweite Parameter angibt was mit einem leeren Baum passiert. Überlegen Sie sich, wie `foldBinTree` zur Lösung der vorherigen Teilaufgaben genutzt werden kann.

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 8

Aufgabe 8.1 *Modellierung*

```
data Firma = Firma { abteilungen :: [Abteilung] } deriving Show

data Abteilung = Abteilung { manager     :: Manager
                           , mitarbeiter :: [Mitarbeiter] }
                           deriving Show

data Manager = Manager { managerName   :: String
                       , managerGehalt :: Float }
                       deriving Show

data Mitarbeiter = Mitarbeiter { mitarbeiterName   :: String
                               , mitarbeiterGehalt :: Float}
                               deriving Show


meineFirma :: Firma
meineFirma = Firma [Abteilung (Manager "Manager" 1000)
                              [Mitarbeiter "Mitarbeiter" 800]
                   ]


incSalary :: Firma -> Firma
incSalary (Firma as) = Firma $ map incSalaryAbt as

incSalaryAbt :: Abteilung -> Abteilung
incSalaryAbt (Abteilung manager ms) = Abteilung manager $ map incSalaryMit ms

incSalaryMit :: Mitarbeiter -> Mitarbeiter
incSalaryMit (Mitarbeiter n s) = Mitarbeiter n (s*2)
```

Aufgabe 8.2 *Binärbäume*

```
data BinTree = Empty
             | Branch Int BinTree BinTree
             deriving Show

-- rekursive Lösungen
sumAll :: BinTree -> Int
sumAll Empty            = 0
sumAll (Branch i t1 t2) = i + sumAll t1 + sumAll t2

preorder :: BinTree -> [Int]
preorder Empty            = []
preorder (Branch i t1 t2) = i : preorder t1 ++ preorder t2

mapBinTree :: (Int -> Int) -> BinTree -> BinTree
mapBinTree _ Empty            = Empty
mapBinTree f (Branch i t1 t2) = Branch (f i) (mapBinTree f t1) (mapBinTree f t2)

holdsEverywhere :: (Int -> Bool) -> BinTree -> Bool
holdsEverywhere _ Empty            = True
holdsEverywhere p (Branch i t1 t2) = p i && holdsEverywhere p t1
                                         && holdsEverywhere p t2

-- Definition von fold
fold :: (Int -> a -> a -> a) -> a -> BinTree -> a
fold _ e Empty            = e
fold f e (Branch i t1 t2) = f i (fold f e t1) (fold f e t2)

-- fold-basierte Lösung
sumFold :: BinTree -> Int
sumFold = fold (\i x1 x2 -> i+x1+x2) 0

preorderFold :: BinTree -> [Int]
preorderFold = fold (\i x1 x2 -> i : x1 ++ x2) []

mapFold :: (Int -> Int) -> BinTree -> BinTree
mapFold f = fold (\i x1 x2 -> Branch (f i) x1 x2) Empty

holdsEverywhereFold :: (Int -> Bool) -> BinTree -> Bool
holdsEverywhereFold p = fold (\i x1 x2 -> (p i) && x1 && x2) True
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 9

Aufgabe 9.1 *Intervalle*

(a) Definieren Sie einen Datentyp `Interval` für Intervalle $[x, y)$, wobei die Intervallgrenzen vom Typ `Float` sein sollen. Geben Sie eine Instanz für die Typklasse `Show` an, um Intervalle anzeigen zu können.

(b) Definieren Sie eine Funktion `compatible :: Interval -> Interval -> Bool`. Zwei Intervalle sind kompatibel, wenn sie sich nicht überlappen.

(c) Definieren Sie eine Funktion `split :: Interval -> (Interval,Interval)`, die ein Interval in zwei gleich große Intervalle aufteilt. Beispiel:

```
split (Interval 0 1.0) ==> (Interval 0 0.5, Interval 0.5 1)
```

Aufgabe 9.2 *unendliche Datentypen*

(a) Definieren Sie einen Datentyp `BinTreeInf a` für unendliche Bäume, die Elemente vom Typ `a` enthalten.

(b) Gegeben sei der Typ für endliche Binärbäume `BinTree a`:

```
data BinTree a = Empty | BinTree a (BinTree a) (BinTree a)
```

Definieren Sie eine Funktion `takeInf :: Int -> BinTreeInf a -> Bintree a`, die die ersten `n` Ebenen eines unendlichen Baumes zurückgibt.

(c) Definieren Sie eine Funktion `buildTree :: (a -> (a,a)) -> a -> BinTreeInf a`, die einen unendlichen Baum erzeugt, wobei die übergebene Funktion aus einem Wert die Werte für den linken bzw. rechten Nachfolger bestimmt. Beispielsweise soll die Funktion `buildTree split (Interval 0 1.0)` zu einem unendlichen Baum reduzieren, bei dem an jedem inneren Knoten die Intervalle geteilt werden.

(d) Definieren Sie Instanzen der Typklasse `Functor` für `BinTree` und `BinTreeInf`

Aufgabe 9.3 *Collections*

(a) Definieren Sie einen Datentyp `Collection a`, der als Attribute eine Liste `[a]` und ein Prädikat `a -> [a] -> Bool` hat. Die Semantik der `Collection`s ist, dass Werte `x` nur dann in die Liste `xs` aufgenommen werden, wenn das Prädikat `p` für `x` und `xs` als Argumente `True` liefert. Das Prädikat gibt also an, ob ein Wert einen Informationsgewinn liefert. Geben Sie eine Instanz der Typklasse `Show` für den Typ `Collection` an.

(b) Definieren Sie eine Funktion `empty :: (a -> [a] -> Bool) -> Collection a`, die eine leere `Collection` mit einem gegebenen Prädikat erzeugt.

(c) Definieren Sie eine Funktion `ins :: a -> Collection a -> Collection a`, die einen Wert in die `Collection` aufnimmt, wenn das Prädikat für die aktuelle Liste und den einzufügenden Wert `True` liefert.

(d) Definieren Sie ein Modul `Collection`, das den Typ `Collection` und die beiden Funktionen `empty` und `insert` exportiert. Der Konstruktor für den Typ `Collection` wird nicht exportiert, da neue `Collection`s durch die Funktionen `empty` und `ins` definiert werden sollen.

(e) Definieren Sie ein neues Modul, das das Modul `Collection` importiert.

- Definieren Sie in diesem Modul eine leere `Collection Int`, die einen Wert nur dann aufnimmt, wenn dieser noch nicht enthalten ist.
- Definieren Sie in diesem Modul eine leere `Collection Int`, die solange neue Werte aufnimmt wie die Summe der Werte in der Liste kleiner als `50` ist.
- Definieren Sie zudem eine leere `Collection Interval`, die ein Intervall nur dann aufnimmt, wenn es kompatibel zu allen Intervallen in der ursprünglichen `Collection` ist.

Überprüfen Sie das Verhalten der `Collection`s, indem Sie beispielhaft Werte einfügen.

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 9

Aufgabe 9.1 *Intervalle*

```
data Interval = Interval { lower:: Float , upper :: Float }

interval :: Float -> Float -> Interval
interval a b
  | a < b = Interval a b
  | a > b = Interval b a

instance Show Interval where
  show (Interval l u) = "["++show l++","++show u++")"

overlaps :: Interval -> Interval -> Bool
overlaps (Interval a b) (Interval x y) = x < b && a < y

compatible :: Interval -> Interval -> Bool
compatible i j = not (overlaps i j)

split :: Interval -> (Interval, Interval)
split (Interval a b) = let m = (a + b) / 2 in (Interval a m , Interval m b)
```

Aufgabe 9.2 *unendliche Binärbäume*

```
data BinTreeInf a = InfBranch a (BinTreeInf a) (BinTreeInf a)

data BinTree a = Empty
               | Branch a (BinTree a) (BinTree a)
              deriving Show

takeInf :: Int -> BinTreeInf a -> BinTree a
takeInf 0 _ = Empty
takeInf n (InfBranch v t1 t2) =
  Branch v (takeInf (n-1) t1) (takeInf (n-1) t2)


buildTree :: (a -> (a,a)) -> a -> BinTreeInf a
buildTree f x =
  InfBranch x (buildTree f (fst $ f x)) (buildTree f (snd $ f x))

instance Functor BinTreeInf where
  fmap f (InfBranch x t1 t2) = InfBranch (f x) (fmap f t1) (fmap f t2)

instance Functor BinTree where
  fmap _ Empty = Empty
  fmap f (Branch x t1 t2) = Branch (f x) (fmap f t1) (fmap f t2)
```

Aufgabe 9.3 *Der Datentyp* `Collection`

```
data Collection a = Collection [a] (a -> [a] -> Bool)

instance Show a => Show (Collection a) where
  show (Collection xs _) = show xs

ins :: a -> Collection a -> Collection a
ins x (Collection xs new)
  | new x xs  = Collection (x:xs) new
  | otherwise = Collection xs new

empty :: (a -> [a] -> Bool) -> Collection a
empty p = Collection [] p

intervalCompatibleSet :: Collection Interval
intervalCompatibleSet =
 ins (Interval 0.7 0.9) $
 ins (Interval 0.2 0.3) $
 ins (Interval 0.1 0.5) $
 empty (\x xs -> all (compatible x) xs)

eqBasedSet :: Collection Int
eqBasedSet =
  ins 4 $
  ins 4 $
  ins 1 $
  empty notElem

sumLessThan :: Collection Int
sumLessThan =
 ins 10 $
 ins 20 $
 ins 5 $
 ins 30 $
 empty (\x xs -> 50 > sum (x:xs))

list :: Collection Int
list =
  ins 4 $
  ins 4 $
  ins 1 $
  empty (\_ _ -> True)
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 10

Aufgabe 10.1 *λ-Kalkül*

In dieser Aufgabe sollen einzelne Funktionalitäten des λ-Kalküls implementiert werden. Gegeben sei dazu folgender Datentyp für Termvariablen:

```
data Var = Var { var :: String }
```

(a) Definieren Sie einen Datentyp `Lambda` für λ-Terme nach folgenden Vorgaben:

- jede Termvariable ist ein $\lambda-$Term.
- für zwei λ-Terme P und Q ist (PQ) ein λ-Term (Applikation).
- für eine Termvariable x und einen $\lambda-$Term P ist $(\lambda x.P)$ ein λ-Term ($\lambda-$Abstraktion).

Implementieren Sie zudem eine Instanz der Typklasse `Show` für den Typ `Lambda`.

(b) Repräsentieren Sie die Funktionen `true`, `false` und die Implikation von Übungsaufgabe 3.3 als Werte des Typs `Lambda`.

(c) Implementieren Sie eine Funktion `freshTermVar :: Lambda -> Var`, die für einen gegebenen λ-Ausdruck eine Variable zurückgibt, die nicht in dem λ-Term vorkommt.

(d) Die freien Variablen $FV(P)$ eines λ-Terms P sind die Termvariablen in P, die nicht durch eine λ-Abstraktion gebunden sind. Implementieren Sie eine Haskell-Funktion `freeVars :: Lambda -> [Var]`, die die freien Variablen eines λ-Terms bestimmt.

(e) Die Substitution $[P/x]Q$ ersetzt P für jedes freie Vorkommen der Variable x in Q.

$$
\begin{aligned}
[P/x]x &= N && && (1)\\
[P/x]y &= y, && \textit{mit } x \neq y && (2)\\
[P/x](M\ N) &= (([P/x]M)\ ([P/x]N)) && && (3)\\
[P/x](\lambda x.N) &= \lambda x.N && && (4)\\
[P/x](\lambda y.N) &= \lambda y.N, && \textit{falls } x \neq y \textit{ und } x \notin FV(N) && (5)\\
[P/x](\lambda y.N) &= \lambda y.[P/x]N, && \textit{falls } x \in FV(N) \textit{ und } y \notin FV(P) && (6)\\
[P/x](\lambda y.N) &= \lambda z.[P/x][z/y]N, && \textit{falls } x \in FV(N) \textit{ und } y \in FV(P) && (7)
\end{aligned}
$$

In Fall (7) ist z eine frische Termvariable, die nicht in (PN) vorkommt (vgl. Aufgabenteil 10.1.c).

Hinweis: der Fall (7) fängt die Situation ab, dass eine ehemals freie Variable durch die Substitution gebunden wird: durch $[y/x]\lambda y.x$ darf nicht die Identitätsfunktion $\lambda y.y$ entstehen, sondern wir ersetzen vor der eigentlichen Substitution alle Vorkommen von y durch eine frische Variable z: Anwendung der Substitution $[y/x][z/y]\lambda y.x$ liefert $[y/x]\lambda z.x$. Weitere Anwendung der Definition führt zu $\lambda z.y$.

(f) Ein β-Redex ist ein Teilterm eines λ-Terms der Form $(\lambda x.P)Q$. Ein λ-Term kann durchaus mehrere β-Redexe enthalten. Die Ersetzung eines β-Redexes $(\lambda x.P)Q$ durch $[Q/x]P$ nennen wir β-Reduktion. Es hat sich als sinnvoll erwiesen, dass immer der äußerste, linke β-Redex reduziert wird.

Definieren Sie eine Haskell-Funktion `betaReduction :: Lambda -> Lambda`, die in einem λ-Term den äußersten, linken β-Redex wie oben gezeigt ersetzt. Achten Sie bei der Applikation (siehe. Teil (a)) darauf, dass `betaReduction` nur maximal einen β-Redex substituiert.

(g) Ein λ-Term ist in β-Normalform, wenn er keine β-Redexe enthält. Schreiben Sie eine Haskell-Funktion `betaNormalForm :: Lambda -> Lambda`, die die β-Normalform eines λ-Terms berechnet.

(h) Lassen Sie von der Funktion `betaNormalForm` die β-Normalform der Anwendungen der Implikation auf die vier Kombinationen der Wahrheitswerte berechnen, zum Beispiel `betaNormalForm (impl `App` true `App` false)` , wobei `impl, true, false :: Lambda` aus Aufgabe 10.1.*b* und `App :: Lambda -> Lambda -> Lambda` der Konstruktor für die Applikation sei.

technische universität dortmund

Fakultät für Informatik
Lehrstuhl für Software-Technologie

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 10

Aufgabe 10.1 *Lambda-Kalkül*

```
module Lambda where
import Data.List

data Var = Var { var :: String } deriving Eq

instance Show Var where
  show (Var v) = v

-- a) datentyp für lambda-formeln
data Lambda = Lambda Var Lambda
            | App Lambda Lambda
            | LVar Var
            deriving Eq

instance Show Lambda where
  show (LVar v) = show v
  show (App (LVar v1) (LVar v2)) = show v1 ++"␣"++ show v2
  show (App e1 e2) = "("++show e1++")␣("++show e2++")"
  show t@(Lambda _ _) =
    let (vs,p) = multipleAbstractions t
    in"␣\\"++(concatMap (\v -> show v ++"␣") vs)
           ++ ".␣("++show p++")"

-- /hilfsfunktion, um \x \y . ... als \x y . darzustellen
multipleAbstractions :: Lambda -> ([Var],Lambda)
multipleAbstractions = helper [] where
  helper :: [Var] -> Lambda -> ([Var],Lambda)
  helper vs (Lambda v e) = helper (vs++[v]) e
  helper vs e            = (vs,e)

-- b) repräsentation von boolescher algebra
x,v,z :: Var
x = Var "x"
z = Var "z"
v = Var "v"

true :: Lambda
true = Lambda x (Lambda v (LVar x))

false :: Lambda
false = Lambda x (Lambda v (LVar v))

implication :: Lambda
implication = Lambda x $ Lambda z $ (LVar x) `App` (LVar z) `App` true

-- c) liefert eine variable zurück, die nicht im lambda-term vorkommt
freshTermVar :: Lambda -> Var
freshTermVar = Var . (++"'") . maximumBy len  . map var . vars where
 len :: [a] -> [a] -> Ordering
 len x y = compare (length x) (length y)

vars :: Lambda -> [Var]
vars (LVar v)     = [v]
vars (App p q)    = union (vars p) (vars q)
vars (Lambda v e) = union [v] (vars e)
```

```
-- d) freie variablen eines lambda-terms
freeVars :: Lambda -> [Var]
freeVars (LVar v)     = [v]
freeVars (App p q)    = union (freeVars p) (freeVars q)
freeVars (Lambda v e) = delete v $ freeVars e

-- e) substitution
sub :: Var -> Lambda -> Lambda -> Lambda
sub x n (LVar y) | x == y = n
                 | x /= y = LVar y
sub x n (App p q) = App (sub x n p) (sub x n q)
sub x n (Lambda y p)
  | x == y                                        = Lambda x p
  | not $ elem x (freeVars p)                     = Lambda y p
  | elem x (freeVars p) && notElem y (freeVars n) = Lambda y (sub x n p)
  | elem x (freeVars p) && elem y (freeVars n)    =
    let z = freshTermVar $ App n p
    in Lambda z $ sub x n $ sub y (LVar z) p

-- f) beta-reduktion
betaReduction :: Lambda -> Lambda
betaReduction (App (Lambda v e) a) = sub v a e
betaReduction (App e1 e2)          =
  if e1 == (betaReduction e1)
  then App e1 (betaReduction e2)
  else App (betaReduction e1) e2
betaReduction (Lambda v e)         = Lambda v $ betaReduction e
betaReduction (LVar v)             = LVar v

-- g) beta-normalform
betaNF :: Lambda -> Lambda
betaNF t
  | t /= (betaReduction t) = betaNF $ betaReduction t
  | otherwise              = t

-- h)
applyToImpl :: Lambda -> Lambda -> Lambda
applyToImpl x y = betaNF $ implication `App` x `App` y

ff = applyToImpl false false
ft = applyToImpl false true
tf = applyToImpl true false
tt = applyToImpl true true

-- alternative: applyToImpl auf alle kombinationen von true und false anwenden
applyAll :: [Lambda]
applyAll = let args = [false,true] in [applyToImpl x y | x <- args , y <- args]
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 11

Aufgabe 11.1 *Visualisierung der β-Reduktion*

(a) Definieren Sie eine Funktion `betaNFSteps :: Lambda -> [Lambda]`, die alle Zwischenergebnisse bei der β-Reduktion zur Bestimmung der Normalform (Übungsblatt 10) zurückgibt.

(b) Implementieren Sie eine Funktion `visualizeBetaReduction :: Lambda -> IO ()`, die den λ-Ausdruck mit der Funktion aus Aufgabenteil (a) in die β-Normalform überführt und die Zwischenergebnisse in der folgenden Form ausgibt:

```
    (\x .  (\x . x) x) y
==> (\x . x) y
==> y
```

Aufgabe 11.2 *Hangman*

Implementieren Sie das Spiel „Hangman“. Eine Beschreibung des Spiels finden Sie auf folgender Webseite: `http://de.wikipedia.org/wiki/Galgenmännchen`. Das Spiel soll interaktiv in der Konsole spielbar sein und selbstständig auf Tastatureingaben vom Benutzer reagieren.
Beim Programmstart soll eine Datei mit den möglichen Wörtern eingelesen werden. Nutzen Sie die Funktion `randomIO :: IO a` aus dem Modul `System.Random`, um eine Zufallszahl zu erhalten, mit der Sie zufällig ein Wort aus der Datei auswählen.

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 11

Aufgabe 11.1 *Lambda-Kalkül*

```
betaNFSteps :: Lambda -> [Lambda]
betaNFSteps t
  | t /= (betaReduction t) = t : (betaNFSteps $ betaReduction t)
  | t == (betaReduction t) = [t]

visualize :: [Lambda] -> IO ()
visualize [] = return ()
visualize (x:xs) = do
  putstrLn $ "==>" ++ show x
  visualize xs

visualizeBeta :: Lambda -> IO ()
visualizeBeta = visualize . betaNFSteps
```

Aufgabe 11.2 *Hangman*

```
module Hangman where
import System.Random (randomIO)
import Data.Char (toUpper)

data Character
  = Hidden Char
  | Visible Char

instance Show Character where
  show (Hidden _) = "_␣"
  show (Visible c) = show c ++ "␣"

data Word = Word [Character]

fromString :: String -> Word
fromString = Word . map (Hidden . toUpper)

instance Show Word where
  show (Word cs) = foldr (\c acc -> show c ++ "␣" ++ acc) "" cs

unshadowChar :: Char -> Word -> Word
unshadowChar c (Word cs) = Word $ map f cs where
  f :: Character -> Character
  f (Visible x) = Visible x
  f (Hidden x) | x == c = Visible x
               | x /= c = Hidden x

successfulGuess :: Char -> Word -> Bool
successfulGuess c (Word cs) = any p cs where
  p :: Character -> Bool
  p (Visible x) | x == c = True
                | x /= c = False
  p (Hidden x) | x == c = True
               | x /= c = False

isWordSolved:: Word -> Bool
isWordSolved (Word cs) = all p cs where
  p :: Character -> Bool
  p (Visible _) = True
  p (Hidden _)  = False

unshadowAll :: Word -> Word
unshadowAll (Word cs) = Word $ map f cs where
  f :: Character -> Character
  f (Visible x) = Visible x
  f (Hidden x)  = Visible x

-- * IO-basierte funktionen

randomWord :: IO Word
randomWord = do
  contents <- readFile "dictionary"
  let words = lines contents
  randomInt <- randomIO :: IO Int
  let index = randomInt `mod` length words
  return $ fromString $ words !! index
```

```
allowedNumberOfMistakes :: IO Int
allowedNumberOfMistakes = do
  putStrLn "How␣many␣mistakes␣should␣be␣allowed?"
  strMistakes <- getLine
  return $ (read strMistakes :: Int)

main :: IO ()
main = do
  n <- allowedNumberOfMistakes
  word <- randomWord
  print word
  play n word

play :: Int -> Word -> IO ()
play 0 word | isWordSolved word = congrats
            | otherwise         = loss $ unshadowAll word
play n word = do
  putStrLn $ "Guess␣a␣character:␣[" ++ show n ++ "␣mistakes␣allowed]"
  c' <- getChar
  putStrLn ""
  let c = toUpper c'
      newWord = unshadowChar c word
      n' | successfulGuess c word = n
         | otherwise              = n-1
  print newWord
  if isWordSolved newWord
  then congrats
  else play n' newWord

congrats :: IO ()
congrats = putStrLn "Congratulations!␣You␣won!"

loss :: Word -> IO ()
loss word = do
  putStrLn $ "You␣lost!␣The␣word␣was:"
  print word
```

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Übungsblatt 12

Aufgabe 12.1 *Partielle Funktionen*

Gegeben seien die drei partiellen Funktionen

```
logarithmus :: Float -> Maybe Float
logarithmus x
  | x <= 0 = Nothing
  | otherwise = Just (log x)

quadratwurzel :: Float -> Maybe Float
quadratwurzel x
  | x < 0 = Nothing
  | otherwise = Just (sqrt x)

kehrwert :: Float -> Maybe Float
kehrwert x
  | x == 0 = Nothing
  | otherwise = Just (1 / x)
```

Schreiben Sie partielle Funktionen `ergebnis1, ergebnis2, ergebnis3 :: Float -> Maybe Float`, die den Kehrwert der Wurzel des Logarithmus von x berechnen. Formulieren Sie die Funktionen auf unterschiedliche Weise:

(a) Benutzen Sie die `case of` Formulierung.

(b) Schreiben Sie die Funktion als `bind`, also mit `>>=`.

(c) Formulieren Sie in der `do`-Notation.

Prof. Dr. E.-E. Doberkat

Jan Bessai
Christian Bockermann
Pascal Hof

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2012/2013

Musterlösung 12

Aufgabe 12.1 *Maybe-Monade*

```
module Loesung12 where

logarithmus :: Float -> Maybe Float
logarithmus x
  | x <= 0 = Nothing
  | x >  0 = Just $ log x


wurzel :: Float -> Maybe Float
wurzel x
  | x < 0 = Nothing
  | x>= 0 = Just $ sqrt x

kehrwert :: Float -> Maybe Float
kehrwert x
  | x == 0 = Nothing
  | x /= 0 = Just $ 1 / x


ergebnis1 :: Float -> Maybe Float
ergebnis1 x =
  case logarithmus x of
    Nothing -> Nothing
    Just y  -> case wurzel y of
       Nothing -> Nothing
       Just z  -> kehrwert z

ergebnis2 :: Float -> Maybe Float
ergebnis2 x = logarithmus x >>= \y -> wurzel y >>= \z -> kehrwert z

ergebnis2_kurz x = logarithmus x >>= wurzel >>= kehrwert


ergebnis3 :: Float -> Maybe Float
ergebnis3 x = do
  y <- logarithmus x
  z <- wurzel y
  kehrwert z
```