technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 0

Ausgabe: 10.10.2014, **freiwillige Abgabe:** keine Abgabe

Das vorliegende Übungsblatt 0 soll mit dem grundlegenden Handgriffen bei der Programmierung in Haskell vertraut machen. Dem entsprechend ist für dieses Übungsblatt **keine Abgabe** vorgesehen.

Aufgabe 0.1

Installieren Sie die Haskell-Platform (`http://www.haskell.org/platform/`) auf ihrem Rechner. Stellen Sie dabei sicher, dass `ghc` und `ghci` zu ihrer Pfadvariablen hinzugefügt sind.

1. Öffnen Sie den Texteditor Ihrer Wahl und tippen Sie folgendes Programm ab:

   ```
   f :: Int -> Int -> Int -> Int
   f x y z = x + y + z * z
   ```

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

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

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

4. Rufen Sie nun die Funktion `f` auf, indem Sie zum Beispiel `f 1 2 3` eingeben und mit ENTER bestätigen. Das Ergebnis wird ausgegeben und Sie können weitere Funktionsaufrufe auswerten lassen.

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 f` oder `:t f 1 2 3`.
- `:help` (kurz `:h`) öffnet die Hilfe mit weiteren nützlichen Befehlen.
- `:quit` (kurz `:q`) beendet den GHCi.

Aufgabe 0.2

Die folgende Aufgabe entählt 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.

1. `f 3 1 True`
2. `f 4 3 2 1`
3. `f 3 2 1`
4. `foo 3 2 1`

Aufgabe 0.3

Gegeben sei die Funktion `addFive` und die konstante Funktion `one`:

```
addFive :: Int -> Int
addFive x = x + 5

one :: Int
one = 1
```

1. Definieren Sie eine Konstante `k :: Int`, deren Auswertung die Zahl `11` liefert. Nutzen Sie dabei ausschließlich die Funktionen `addFive` und `one`.
2. Definieren Sie eine Funktion `addTen :: Int -> Int`, die eine ganze Zahl als Parameter erhält und die Summe aus Zehn und der übergebenen Zahl berechnet. Nutzen Sie dafür die Funktion `addFive` und den Dollar-Operator.
3. Definieren Sie eine Funktion `addTenComposition :: Int -> Int`, die semantisch äquivalent zu der Funktion `addTen` ist, jedoch nur die Funktion `addFive` und die Funktionskomposition nutzt.

Aufgabe 0.4

1. Implementieren Sie die eine Funktion `max3 :: Int -> Int -> Int -> Int`, die das Maximum dreier Zahlen bestimmt. Dabei können Sie die Funktion `max :: Int -> Int -> Int` aus der Standardbibliothek nutzen, die das Maximum zweier Zahlen bestimmt.
2. Eine quadratische Gleichung $x^2 + ax + b = 0$ mit ganzzahligen Koeffizienten a und b kann durch das Tupel (a, b) mit Integer-Werten a und b repräsentiert werden.

   Schreiben Sie eine Haskell-Funktion `loesung :: Int -> (Int,Int) -> Bool`, die als Eingabe eine ganze Zahl n und ein Tupel (a, b) mit ganzen Zahlen a und b erhält und die testet, ob n Lösung der Gleichung $x^2 + ax + b = 0$ ist.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 1

Ausgabe: 10.10.2014, **freiwillige Abgabe:** 17.10.2014

Aufgabe 1.1 Die folgenden Funktionen in Lambda-Notation sind gegeben.

```
f1 = \x -> x + 1

f2 = \(x,y) -> (x * y + 1 + x)

f3 = \x -> \y -> y (x * x)

f4 = \f g -> f . g . f
```

1. Formen Sie die Funktionen in die applikative Notation um.
2. Bestimmen Sie die Typen der Funktionen ohne Compilerunterstützung. Nehmen Sie dabei an, dass `1 :: Int` und `(+), (*) :: Int -> Int -> Int` gilt.
3. Entscheiden Sie für die in der vorherigen Aufgabe bestimmten Typen, ob es sich jeweils um einen monomorphen oder polymorphen Typen handelt.

Aufgabe 1.2 Gegeben sei die folgende Funktion `add4Ints`:

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

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

Aufgabe 1.3 Die Collatz-Funktion ist wie folgt definiert:

$$c(n) = \begin{cases} n/2, & \text{falls n gerade} \\ 3n+1, & \text{falls n ungerade} \end{cases}$$

1. Implementieren Sie eine Haskell-Funktion `collatz :: Int -> Int`, die für eine natürliche Zahl das nächste Element in der Collatz-Folge berechnet.

   Gehen Sie also davon aus, dass die Funktion `collatz` nur natürliche Zahlen als Argument erhält. Die Funktion `div :: Int -> Int -> Int` implementiert die Integerdivision in Haskell.

2. Die Collatz-Folge ergibt sich aus der mehrfachen Anwendung der Funktion c auf eine natürliche Zahl n. Man nimmt an, dass die Collatz-Folge für jede natürliche Zahl irgendwann einmal den Wert 1 annimmt.

   Implementieren Sie eine Funktion `collatzMax :: Int -> Int`, die für eine gegebene natürliche Zahl n die größte Zahl zurückgibt, die die Collatz-Folge für Startwert n annimmt.

   Beispiele:

   - $n = 4 : (4, 2, 1)$, `collatzMax 4` $\rightsquigarrow$ `4`
   - $n = 21 : (21, 64, 32, 16, 8, 4, 2, 1)$, `collatzMax 21` $\rightsquigarrow$ `64`

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 2

Ausgabe: 17.10.2014, **freiwillige Abgabe:** 24.10.2014

Aufgabe 2.1 Reduzieren Sie die folgenden λ-Ausdrücke ohne Compilerunterstützung, indem Sie immer den **äußersten** λ-Ausdruck auflösen. Geben Sie für jede Auflösung eines λ-Ausdrucks ein Zwischenergebnis an.

1. `(λx -> 4 * x) $ (λy -> 3 + y) 1`
2. `(λf -> (f . f) 2) (*3)`
3. `(λx f -> x + f x) 5 (*3)`

Aufgabe 2.2 Bestimmen Sie die Typen der folgenden Ausdrücke ohne Compilerunterstützung. Gehen Sie bei der Typinferenz davon aus, dass `2 :: Int` und `(+) :: Int -> Int -> Int` gelten.

1. `λx y -> if x y then y else y+2`
2. `λf -> (f . f) 2`

Aufgabe 2.3 Implementieren Sie zwei Funktionen `sumRec :: Int -> Float` und `sumIter :: Int -> Float`, die die folgende Summe rekursiv bzw. iterativ berechnen:

$$\sum_{1 \le i \le n} \frac{1}{i}$$

Hinweis: Sie können die folgende Funktion zur Konvertierung von `Int` nach `Float` nutzen:

```
intToFloat :: Int -> Float
intToFloat = fromIntegral
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 3

Ausgabe: 24.10.2014, **freiwillige Abgabe:** 31.10.2014

Aufgabe 3.1 Typinferenz

Diese Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass `4 :: Int` gilt.

1. `dropWhile (/=4)`
2. `last . ([4]++)`
3. `λx -> (reverse . map (λx -> 4*x)) x`

Aufgabe 3.2 Listen

1. Definieren Sie eine Haskell-Funktion `fromTo :: Int -> Int -> [Int]`, die für zwei Parameter `begin` und `end` die sortierte Liste der ganzen Zahlen zwischen `begin` und `end` erzeugt. Die beiden Grenzen `begin` und `end` sollen in der Liste enthalten sein.

   Beispielsweise führt der Ausdruck `fromTo (-3) 6` zu folgendem Ergebnis:

   `[-3,-2,-1,0,1,2,3,4,5,6]`

2. Definieren Sie eine Haskell-Funktion `squares :: Int -> [Int]`, die für ein positive natürliche Zahl n die ersten n Quadratzahlen berechnet. Gehen Sie davon aus, dass `squares` nur mit natürlichen Zahlen als Parameter aufgerufen wird. Die Funktion `fromTo` könnte bei der Definition von `squares` hilfreich sein.

3. Definieren Sie eine Haskell-Funktion `conjunction :: [Bool] -> Bool`, die genau dann den Wert `True` zurückgibt, wenn die übergebene Liste ausschließlich den Wert `True` enthält. Gehen Sie davon aus, dass die übergebene Liste nicht leer ist.

Aufgabe 3.3 Collatz-Folge

Definieren Sie eine Haskell-Funktion `collatzSequence :: Int -> [Int]`, die für einen Startwert die Collatz-Folge berechnet. Sobald ein Folgeglied den Wert `1` erreicht, soll die Berechnung terminieren. Beispielsweise wird `collatzSequence 10` zu `[10,5,16,8,4,2,1]` ausgewertet.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 4

Ausgabe: 31.10.2014, **freiwillige Abgabe:** 07.11.2014

Aufgabe 4.1 Typinferenz

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass `4 :: Int` gilt.

1. `span (λy -> y /= '4')`
2. `head . map (λc -> c+4) . takeWhile (λc -> c<4)`
3. `map (map (λx -> 4*x))`

Aufgabe 4.2 Reduktion

Gegeben seien die beiden Funktionen:

```
sumr , suml :: [Int] -> Int
sumr = foldr (+) 0
suml = foldl (+) 0
```

1. Reduzieren Sie den Ausdruck `sumr [1,2,3,4]`.
2. Reduzieren Sie den Ausdruck `suml [1,2,3,4]`.

Geben Sie aussagekräftige Zwischenergebnisse an, die den Lösungsweg dokumentieren.

Aufgabe 4.3 Listenfunktionen

1. Implementieren Sie eine Funktion `multiply :: [Int] -> [Int]`, so dass hinter jedem Wert n aus der Liste genau $n - 1$ Kopien von n eingefügt werden. Gehen Sie dabei davon aus, dass die Argumentliste ausschließlich echt positive Werte enthält.

   Beispiel: `multiply [1,2,1,4,1,3]` $\rightsquigarrow$ `[1,2,2,1,4,4,4,4,1,3,3,3]`

2. Implementieren Sie eine Haskell-Funktion `removeDuplicates :: [Int] -> [Int]`. Die Auswertung von `removeDuplicates xs` soll eine Liste zurückgeben, in der kein Wert aus `xs` mehrfach vorkommt.

   Beispiel: `removeDuplicates [1,2,1,4,1,3]` $\rightsquigarrow$ `[1,2,3,4]`

3. Implementieren Sie eine Funktion `mapUsingFoldr :: (a -> b) -> [a] -> [b]`, die semantisch äquivalent zu der Funktion `map` ist. Ihre Definition soll jedoch mit Hilfe von `foldr` geschehen. Es ist also der folgende Coderahmen zu vervollständigen:

   ```
   mapUsingFoldr :: (a -> b) -> [a] -> [b]
   mapUsingFoldr f xs = foldr ...
   ```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 5

Ausgabe: 07.11.2014, **freiwillige Abgabe:** 14.11.2014

Aufgabe 5.1 Typinferenz

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass alle Zahlen vom Typ `Int` und die Funktionen `(+),(*)` vom Typ `Int -> Int -> Int` sind.

1. `map (foldr (+) 0)`
2. `any (<100) . map (λx -> (x - 1)*6)`
3. `zipWith ($) [(+),(*)]`

Aufgabe 5.2

Wir betrachten das kartesische Koordinatensystem, das durch die horizontal verlaufende x-Achse und die vertikal verlaufende y-Achse gegeben ist. Eine Gerade g in diesem Koordinatensystem ist durch die Angabe eines Punktepaares (a, b) durch die folgende Festlegung eindeutig bestimmt: $(a, 0)$ sei der Schnittpunkt von g mit der x-Achse und $(0, b)$ sei der Schnittpunkt von g mit der y-Achse (Achsenabschnittsform einer Geraden).

Definieren Sie eine Haskell-Funktion `solution :: (Float, Float) -> [(Int,Int)]`, die bei Eingabe eines Paares (a, b) von positiven Dezimalzahlen a und b die Liste der ganzzahligen Punktepaare berechnet, die in dem Dreieck liegen, das durch die y-Achse, die x-Achse und die Gerade g eingeschlossen wird.

Hinweis: Die Funktion `floor :: Float -> Int` rundet den Wert vom Typ `Float` ab.

Aufgabe 5.3 Pythagoreische Tripel

Ein Tripel $(a, b, c) \in \mathbb{N}^3$ ist ein pythagoreisches Tripel genau dann, wenn $a < b < c$ und $a^2+b^2 = c^2$ gelten. Zum Beispiel handelt es sich bei $(3, 4, 5)$ um ein pythagoreisches Tripel.

Implementieren Sie eine Haskell-Funktion `pyTriples :: [(Int,Int,Int)]`, die die unendliche Liste aller pythagoreischen Tripel erzeugt.

Aufgabe 5.4 Modellierung

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

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

   Hinweis: Fügen Sie jeder Datentypdefinition ein `deriving Show` an, um Werte des Datentyps anzeigen zu können.

2. Definieren Sie eine nichttriviale Beispielinstanz des Datentyps.

3. Implementieren Sie eine Funktion, die eine Firma als Argument erhält und das Gehalt aller Mitarbeiter verdoppelt.

4. Implementieren Sie eine Funktion, die eine Firma als Argument erhält und die Namen aller Abteilungsleiter zurückgibt.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 6

Ausgabe: 14.11.2014, **freiwillige Abgabe:** 21.11.2014

Gegeben sei der aus der Vorlesung bekannte Datentyp für Listen:

```
data List a
  = Nil
  | Cons a (List a)
  deriving Show
```

Die natürlichen Zahlen $\mathbb{N}$ können induktiv wie folgt gebildert werden:

- Die Zahl `Zero` ist eine natürliche Zahl.
- Ist `n` eine natürliche Zahl, dann ist auch `Succ n` eine natürliche Zahl.

Folgender Haskell-Datentyp repräsentiert also natürliche Zahlen:

```
data Nat = Zero | Succ Nat deriving Show
```

Aufgabe 6.1 Typinferenz Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass `6 :: Int` ist.

1. `foldl (λacc x -> acc || (x ‘mod‘ 6 == 0)) False`
2. `Cons (Succ Zero)`
3. `foldr Cons Nil`

Aufgabe 6.2 Listen

Die folgenden Funktionen sollen rekursiv über die Struktur der Argumente implementiert werden.

1. Definieren Sie eine Haskell-Funktion `mapList :: (a -> b) -> List a -> List b`, die die übergebene Funktion auf jedes Listenelement anwendet.
2. Definieren Sie eine Haskell-Funktion `foldrList :: (a -> b -> b) -> b -> List a -> b`, die eine Faltung von rechts auf einer Liste vom Typ `List a` realisiert.
3. Definieren Sie eine Haskell-Funktion `get :: Int -> List a -> Maybe a`, die einen indexbasierten Zugriff auf Listenelemente implementiert. Sorgen Sie mit Hilfe des Datentyps `Maybe a` dafür, dass die Funktion `get` für **alle** möglichen Eingaben einen Wert (vom Typ `Maybe a`) zurückgibt.

Aufgabe 6.3 Natürliche Zahlen und Listen

1. Definieren Sie eine Haskell-Funktion `toInt :: Nat -> Int`, die eine natürliche Zahl in den repräsentierten `Int` überführt.

2. Definieren Sie eine Haskell-Funktion `getNat :: Nat -> List a -> Maybe a` analog zu Aufgabe 6.2.3, die jedoch statt einem Wert vom Typ `Int` einen Wert vom Typ `Nat` als Index erhält. Ein Aufruf der Funktion `get` aus Aufgabe 6.2.3 ist nicht erlaubt.

3. Definieren Sie eine Haskell-Funktion `len :: List a -> Nat`, die die Länge einer Liste bestimmt.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 7

Ausgabe: 21.11.2014, **freiwillige Abgabe:** 28.11.2014

Aufgabe 7.1 Typinferenz

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmen Sie die Typen der folgenden Ausdrücke.

1. `zipWith (/=)`
2. `λy -> map (λx -> x == y)`

Aufgabe 7.2 Typklassen `Eq` und `Ord`

Wir betrachten hier erneut die Typen `Nat` und `List a` von Präsenzblatt 6:

```
data Nat = Zero | Succ Nat deriving Show

data List a = Nil | Cons a (List a) deriving Show
```

Eine Instanz von `Eq` muss entweder `(==)` oder `(/=)` implementieren. Eine Instanz von `Ord` muss entweder `(<=)` oder `compare` implementieren.

1. Definieren Sie eine Instanz der Typklasse `Eq` für den Datentyp `Nat`.
2. Definieren Sie eine Instanz der Typklasse `Ord` für den Datentyp `Nat`.
3. Definieren Sie eine Instanz der Typklasse `Eq` für den Datentyp `List a`.

Aufgabe 7.3 Typklasse `Show`

In der Prelude ist die Typklasse `Show` wie folgt (etwas vereinfacht) definiert:

```
class Show a where
  show :: a -> String
```

Die Funktion `show` gibt an, wie ein Wert ausgegeben wird.

1. Definieren Sie eine `Show`-Instanz für `Nat`, indem Sie die Funktion `show` mit folgendem Verhalten implementieren: `show (Succ (Succ Zero))` ⇝ `"2"`
2. Definieren Sie eine `Show`-Instanz für `List a`, indem Sie die Funktion `show` mit folgendem Verhalten implementieren: `show (Cons 3 $ Cons 5 $ Cons 9 Nil)` ⇝ `"3 , 5 , 9"`

Hinweis: Um nicht zwei Instanzen der Typklasse `Show` vorliegen zu haben, müssen Sie vor Bearbeitung der Aufgabe die automatisch erzeugte `Show`-Instanz (`deriving Show`) für die entsprechenden Typen entfernen.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 8

Ausgabe: 28.11.2014, **freiwillige Abgabe:** 5.12.2014

Aufgabe 8.1 `Collections`

1. Definieren Sie einen Datentyp `Collection a`, der als Attribute eine Liste `[a]` und ein Prädikat `a -> [a] -> Bool` hat.

   Die Semantik des Datentyps ist, dass nur Werte `x` in die Liste `xs` aufgenommen werden, wenn das Prädikat `p` für `x` und `xs` als Argumente `True` liefert.

2. Geben Sie eine Instanz der Typklasse `Show` für `Collection a` an.

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

4. Definieren Sie eine Funktion `insert :: 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.

5. Definieren Sie als Beispiel für eine `Collection` den Wert `intSet :: Collection Int`, der Mengen implementiert.

   Beispiel: `insert 3 $ insert 1 $ insert 1 intSet` $\rightsquigarrow$ `[3,1]`

6. Definieren Sie als weiteres Beispiel eine `Collection` `palindromes :: Collection String`, die nur Palindrome aufnimmt. Ein Palindrom ist ein Wort, das von vorne und hinten gelesen dasselbe ergibt.

Aufgabe 8.2 Typklasse `Monoid` und Endofunktionen

Die Typklasse `Monoid` aus dem Modul `Data.Monoid` ist vereinfacht wie folgt definiert:

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
```

Die Typklasse bietet noch eine weitere Funktion, die wir für unsere Zwecke jedoch nicht benötigen. Sie können für die Aufgabe entweder die Typklasse mit `import Data.Monoid` aus der Standardbibliothek importieren oder die obige Definition nutzen.

1. Definieren Sie für den Datentyp `data Nat = Zero | Succ Nat` eine `Monoid`-Instanz, so dass `mempty` die Null und `mappend` die Addition zweier natürlicher Zahlen darstellt.

2. Gegeben sei der Datentyp `Endo a` zur Repräsentation von Endofunktionen:

   ```
   data Endo a = Endo { getEndo :: a -> a }
   ```

   Definieren Sie eine `Monoid`-Instanz für `Endo a`.

3. Die Instanzen von `Monoid` sollten folgende drei Regeln erfüllen:

   - `mapppend mempty x = x`
   - `mappend x mempty = x`
   - `mappend x (mappend y z) = mappend (mappend x y) z`

   Beweisen Sie, dass Ihre Implementierung diese Regeln befolgt. Sie können dabei davon ausgehen, dass die Funktionskomposition assoziativ ist.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 9

Ausgabe: 5.12.2014, **freiwillige Abgabe:** 12.12.2014

Aufgabe 9.1 Rationale Zahlen

Informieren Sie sich in der Dokumentation[1] über den Datentyp `Ratio a`.

Das Heron-Verfahren ist ein Algorithmus zur Approximation der Quadratwurzel einer Zahl q.

$$\begin{aligned} x_0 &= q \\ x_n &= \tfrac{1}{2}\left(x_{n-1} + \tfrac{q}{x_{n-1}}\right) \end{aligned}$$

Dieses Verfahren kann man über ein zusätzliches Argument ϵ parametrisieren, so dass das Ergebnis zurückgegeben wird, sobald die Abweichung vom tatsächlichen Ergebnis kleiner als ϵ ist.

Implementieren Sie also die Funktion

```
heronsMethod :: Rational -> Rational -> Rational
```

, so dass `heronsMethod q epsilon` die Quadratwurzel von `q` bis auf eine Abweichung von `epsilon` approximiert.

Folgende Beispiele zeigen das Verhalten der Funktion:

- `heronsMethod 4 (1 % 10)` ⇝ `3281 % 1640`
- `heronsMethod 5 (1 % 1000)` ⇝ `2207 % 987`

[1] http://hackage.haskell.org/package/base-4.7.0.1/docs/Data-Ratio.html

Aufgabe 9.2 Erzeugung zufälliger natürlicher Zahlen

Wir betrachten den Datentyp für natürliche Zahlen

```
data Nat = Zero | Succ Nat
```

sowie die beiden Funktionen `toInt :: Nat -> Int8` und `toNat :: Int8 -> Nat`. Sie finden diese Funktionen bereits in der Vorlagedatei `Vorlage09.hs` im EWS.

Beachten Sie, dass wir hier den Datentyp `Int8` für ganze Zahlen in der Repräsentation mit 8-Bit aus dem Modul `Data.Int` nutzen. Wir nutzen hier einen kleineren Datentyp als `Int`, um die Erzeugung von zufälligen Werten effizient zu halten.

Definieren Sie eine Instanz der Typklasse `Random`, die Sie in dem Modul `System.Random`[2] finden, indem Sie die Funktionen

```
randomR :: RandomGen g => (a,a) -> (a,g)
```

und

```
random RandomGen g => g -> (a,g)
```

implementieren. Details zur Funktionsweise der beiden Funktionen entnehmen Sie der entsprechenden Dokumentation.

Hinweis: Nutzen Sie dabei die Instanz von `Random` für den Datentyp `Int8` und die Tatsache, dass für `Int8` eine Instanz der Typklasse `Bounded`[3] existiert.

[2] http://hackage.haskell.org/packages/archive/random/latest/doc/html/System-Random.html

[3] http://hackage.haskell.org/packages/archive/ base/latest/doc/html/Prelude.html#t:Bounded

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 10

Ausgabe: 19.12.2014, **freiwillige Abgabe:** keine Abgabe

Aufgabe 10.1 Kinds

Gegeben seien folgende Datentypen:

```
data Bool = True | False
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data WrapInt f = WrapInt (f Int)
data T f g = T (f Bool) (g Int Bool)
```

1. Bestimmen Sie den Kind der folgenden Typen bzw. Typkonstruktoren:

   `Bool`, `Maybe`, `Either`, `WrapInt`, `T`.

2. Geben Sie für die Typen `WrapInt f` und `T f g` je zwei beliebige Werte an.

Aufgabe 10.2 Funktoren

Wir betrachten die aus der Prelude bekannte Typklasse `Functor`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Zudem sei folgender Datentyp für Binärbäume gegeben:

```
data Bintree a = Leaf | Branch a (Bintree a) (Bintree a)
```

1. Instanziieren Sie die Typklasse `Functor` für den Typ `Bintree`.

2. Beweisen Sie, dass Ihre Definition aus Aufgabenteil 1 die beiden Regeln für Funktoren erfüllt. Jede Instanz der Typklasse `Functor` muss folgende Regeln erfüllen:

   ```
   fmap id = id
   fmap (f . g) = fmap f . fmap g
   ```

   Alternativ kann man auch sagen, dass für alle `x :: Bintree a` folgendes gelten muss:

   ```
   fmap id x = x
   fmap (f . g) x = fmap f (fmap g x)
   ```

   Hinweis: Nutzen Sie eine vollständige Induktion über die Tiefe des Baumes als Beweistechnik.

3. Definieren Sie eine Funktion `doubleValues :: Functor f => f Int -> f Int`, die die Funktion `fmap` nutzt, um alle Werte in einem beliebigen Funktor zu verdoppeln. Überprüfen Sie die Funktionsweise der Funktion `doubleValues` an den Funktoren `Bintree`, `[]` und `Maybe`.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 11

Ausgabe: 05.01.2015, **freiwillige Abgabe:** 09.01.2015

Aufgabe 11.1 Partielle Funktionen

Gegeben seien folgende drei partielle 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
  | x /= 0 = Just (1 / x)
```

Implementieren Sie partielle Funktionen `fCase, fBind, fDo, fKleisli :: Float -> Maybe Float`, die jeweils den Kehrwert der Wurzel des Logarithmus' ihres Arguments berechnen. Formulieren Sie die vier semantisch äquivalenten Funktionen wie folgt:

1. Benutzen Sie die `case`-Syntax zur Fehlerbehandlung in der Funktion `fCase`.
2. Benutzen Sie die »=-Notation zur Fehlerbehandlung in der Funktion `fBind`.
3. Benutzen Sie die `do`-Notation zur Fehlerbehandlung in der Funktion `fDo`.
4. Benutzen Sie die Kleisli-Komposition in der Funktion `fKleisli`.

Aufgabe 11.2 `[]-Monade`

Reduzieren Sie folgenden Ausdruck, indem Sie

1. die Listenkomprehension in die do-Notation überführen,
2. dann in die »=-Notation wechseln
3. und die Definition von »= einsetzen.

```
xs :: [(Int, Int)]
xs = [ (x,y) | x <- [1,2] , y <- [6,7] , x + y /= 8]
```

Aufgabe 11.3 Programmbeweis

Gegeben seien folgende Funktionen:

```
sdiv :: Int -> Int -> Maybe Int
sdiv x y
  | y == 0 = Nothing
  | y /= 0 = Just (x `div` y)
```

```
monadPlusSafeDiv :: MonadPlus m => Int -> Int -> m Int
monadPlusSafeDiv x y = guard (y /= 0) >> return (x `div` y)
```

Beweisen Sie durch Programmtransformationen, dass die Funktionen `sdiv` und `monadPlusSafeDiv` äquivalent sind, wenn wir bei `monadPlusSafeDiv` `Maybe` als Datentyp für `MonadPlus` wählen.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 12

Ausgabe: 09.01.2015, **freiwillige Abgabe:** 06.01.2015

Aufgabe 12.1 Lifting

Implementieren Sie eine Funktion

```
lift :: Monad m => (a -> b -> c) -> m a -> m b -> m c
```

die eine binäre Funktion in eine Monade liftet.

Die folgenden Auswertungen zeigen beispielhaft, wie sich die Funktion `lift` verhält:

- `lift (+) (Just 3) (Just 1)` ⇝ `Just 4`
- `lift (+) Nothing (Just 1)` ⇝ `Nothing`
- `lift (*) [1,2,3] [4,5]` ⇝ `[4,5,8,10,12,15]`

Aufgabe 12.2 Polymorphe Rückgabetypen

Implementieren Sie eine Funktion

```
filt :: MonadPlus m => (a -> Bool) -> [a] -> m a
```

die die Funktion `filter` aus der *Prelude* verallgemeinert. Die Funktion `filt` soll sich bei einer Festlegung des Rückgabetyps auf eine Liste von Werten wie `filter` verhalten:

```
filt (<1) [1,-2,3,-4,5] :: [Int] ⇝ [-2,-4]
```

Darüber hinaus soll `filt` aber beispielsweise auch für einen in `Maybe` eingebetteten Wert funktionieren und dann den ersten Wert in der Liste zurückgeben, der das Prädikat erfüllt:

```
filt (<1) [1,-2,3,-4,5] :: Maybe Int ⇝ Just (-2)
```

Gibt es keinen Wert, für den das Prädikat gilt, gibt die Funktion für den Rückgabetyp `Maybe Int` den Fehlerwert `Nothing` zurück.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Pascal Hof (*pascal.hof@tu-dortmund.de*)
Dr. Hubert Wagner (*hubert.wagner@tu-dortmund.de*)

Wintersemester 2014/2015

Übungen zu Funktionaler Programmierung
Präsenzblatt 13

Ausgabe: 16.01.2015, **freiwillige Abgabe:** 23.01.2015

Zu diesem Präsenzblatt liegt im EWS eine Vorlagedatei mit dem Namen `Vorlage13.hs`, die die Definition der Zustandstransformationsmonade sowie einen Coderahmen für die Aufgaben liefert.

Aufgabe 13.1 Markieren von Bäumen

Diese Aufgabe behandelt noch einmal die Nummerierung von Bäumen, die wir auf dem zweiten Aufgabenblatt bereits bearbeiteten. Sie finden eine Beispiellösung für diese Aufgabe im EWS. Wir wollen Teile eine Funktion `numbering :: Int -> Forest () -> Forest Int` definieren, die die k Knoten eines Waldes beginnend bei einem Wert n mit den Werten von n bis $n + k - 1$ markiert.

In dieser Aufgabe soll die Lösung dieser Aufgabe mit der Zustandstransitionsmonade geschehen.

1. Zunächst wird eine Hilfsfunktion `get :: Trans [a] a` benötigt, die von der Liste im Zustand, das erste Element zurückgibt und die Restliste als Nachfolgezustand festlegt. Sie können dabei davon ausgehen, dass die Liste niemals leer ist, da die Funktion`numbering` immer unendliche Listen als Zustand erzeugt (siehe Vorlage). Definieren Sie die Funktion `get`, indem Sie eine Zustandstransition mit dem Konstruktor `T` erzeugen.

2. Definieren Sie nun die Funktionen

   ```
   numberTree :: Tree () -> Trans [Int] (Tree Int)
   ```

   und

   ```
   numberForest :: Forest () -> Trans [Int] (Tree Int)
   ```

   als Zustandstransitionen, wobei in dem Zustand stets eine unendliche Liste der als nächstes zu vergebenen Markierungen steht. Wie auf dem Aufgabenblatt rufen sich diese Funktionen auch hier wechselseitig rekursiv auf. Definieren Sie diese Funktionen in »=- oder `do`-Notation.

Aufgabe 13.2 Dynamische Programmierung mit Feldern

Gegeben sei folgende rekursive Berechnung des *i*-ten Gliedes der Catalan-Folge:

```
catalan :: Int -> Int
catalan 0 = 1
catalan n = sum (map (\i -> catalan i * catalan (n-1-i)) [0..n-1])
```

Mit Hilfe von dynamischer Programmierung soll nun ein Feld `catalanArr :: Array Int Int` mit den Gliedern der Catalan-Folge definiert werden. Bei der Berechnung der Glieder sollen Zwischenergebnisse durch Zugriffe auf das Feld anstelle durch Rekursion berechnet werden.

```
catalanDyn :: Int -> Int
catalanDyn n = catalanArr ! n where

  catalanArr :: Array Int Int
  catalanArr = mkArray ...
```

Die Funktion `catalanDyn :: Int -> Int` sollte durch die Nutzung der dynamischen Programmierung wesentlich effizienter als die rekursive Variante `catalan` sein.