technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 0

Ausgabe: 18.10.2013, **Abgabe:** N.N.

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.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 1

Ausgabe: 18.10.2013, **Abgabe:** 25.10.2013

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.

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.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 2

Ausgabe: 25.10.2013, **Abgabe:** 01.11.2013

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. `(λx -> 4 + x + (λx -> 3 + x) 3) 1`
3. `(λf -> (f . f) 2) (*3)`
4. `(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`
3. `flip (.)`
4. `uncurry (+)`

Aufgabe 2.3 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 `countStepsToOne :: Int -> Int`, die für eine gegebene natürliche Zahl n die Anzahl der Schritte berechnet, bis die Collatz-Folge für Startwert n den Wert 1 annimmt.

   Beispiele:

   - $4 \longrightarrow 2 \longrightarrow 1$, also gibt `countSteps 4` den Wert `2` zurück.
   - $21 \longrightarrow 64 \longrightarrow 32 \longrightarrow 16 \longrightarrow 8 \longrightarrow 4 \longrightarrow 2 \longrightarrow 1$, also gibt `countSteps 21` den Wert `7` zurück.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 4

Ausgabe: 08.11.2013, **Abgabe:** 15.11.2013

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

Aufgabe 4.2 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. 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 eine Konjunktion beliebiger Stelligkeit implementiert. Die Funktion `conjunction` gibt also genau dann den Wert `True` zurück, wenn die übergebene Liste ausschließlich den Wert `True` enthält. Gehen Sie davon aus, dass die übergebene Liste nicht leer ist.

Aufgabe 4.3 Collatz-Folge

Definieren Sie eine Haskell-Funktion `collatzSequence :: Int -> [Int]`, die für einen Startwert die Collatz-Folge berechnet. Sie können dabei die Funktion aus Aufgabe 2.3.1 verwenden. Sobald ein Folgeglied den Wert `1` erreicht, soll die Berechnung terminieren. Beispielsweise wird `collatzSequence 10` zu `[10,5,16,8,4,2,1]` ausgewertet.

Aufgabe 4.4 Listen von Funktionen

Definieren Sie eine Haskell-Funktion `applyToValue :: [a -> a -> a] -> a -> [a -> a]`, die eine Liste von zweistelligen Funktionen über einem Typ `a` und einen Wert vom Typ `a` enthält. Die Funktion `applyToValue` gibt eine Liste von einstelligen Funktionen zurück, so dass jede Funktion aus der Eingabeliste als erstes Argument den Wert vom Typ `a` erhält.

Die Reduktion von `applyToValue [\x y -> (x+y) * x, (+) , \x y -> y ‘mod‘ x ] 2` soll beispielsweise die Liste `[\y -> (2+y) * 2, (2+) , \y -> y ‘mod‘ 2]` erzeugen. Überlegen Sie sich wie Sie die Funktionsweise der Funktion `applyToValue` ausprobieren können.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 3

Ausgabe: 01.11.2013, **Abgabe:** 08.11.2013

Aufgabe 3.1 Funktionstypen

Fügen Sie zu den folgenden Typen die implizit vorhandenen Klammern ein:

1. `Int -> (Bool,a) -> (a,Int)`
2. `Int -> Bool -> (Int,Int) -> Int`
3. `a -> (a -> b -> a) -> b -> a`

Aufgabe 3.2 Reduktion von Ausdrücken

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

Aufgabe 3.3 Typinferenz

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass `3 :: Int` gilt.

1. `λ(f,g) p x -> if p x then f x else g x`
2. `(curry (λ(x,y) -> if x < y then y else x)) 3`

Aufgabe 3.4 Update von Funktionen

1. Definieren Sie eine Haskell-Funktion `k :: Int -> Int`, die alle Zahlen ungleich 0 auf 1 und die 0 auf 0 abbildet.
2. Ändern Sie mit Hilfe der update-Funktion die Funktion `k` zu einer Funktion `k1 :: Int -> Int` so ab, dass sie für das Argument 2 ebenfalls den Wert 0 annimmt.

   (Beachten Sie: `update` ist keine Standard-Haskell Funktion, so dass Sie die Funktion `update` in Ihrem Haskell-Programm selbst definieren müssen. Die Definition der Funktion finden Sie auf den Folien.)
3. Nun möchten Sie mit Hilfe der update-Funktion die Funktion `k` zu einer Funktion `k2 :: Int -> Int` so abändern, dass sie für die Argumente 2 und 5 den Wert 0 annimmt. Geben Sie die Definition von `k2` an.
4. Schreiben Sie eine Haskell-Funktion `symm :: (Int -> Int) -> Int -> Int`, die eine Funktion `f` so abändert, dass sie für ein negatives Argument `x` den Funktionswert `f (-x)` als Ergebnis hat.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 5

Ausgabe: 15.11.2013, **Abgabe:** 22.11.2013

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 `5 :: Int` gilt.

1. `span (λy -> y /= '5')`
2. `head . map (λc -> c+1) . takeWhile (λc -> c<5)`
3. `map (map (λx -> 5*x))`

Aufgabe 5.2 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 5.3 Schleifen als Faltungen

1. Gegeben sei folgende for-Schleife:

   ```
   x = x0;
   for (i = i0, i <= i1, i++){
      x = f(x,i);
   }
   ```

   Definieren Sie eine Haskell-Funktion mit `foldl`, die `x` in Abhängigkeit von `x0,i0,i1` und `f` berechnet.

2. Gegeben sei folgende while-Schleife:

   ```
   x = x0, i = i0;
   while a(i){
     x = f(x,i);
     i++;
   }
   ```

Benutzen Sie die Funktion `foldl`, um eine Funktion

```
foldWhile :: (a -> Int -> a) -> (Int -> Bool) -> a -> Int -> a
```

zu definieren, so dass sich mit ihr `x = foldWhile f a x0 i0` berechnen läßt.

Aufgabe 5.4 Binärcodierung einer Zeichenkette

Definieren Sie eine Haskell-Funktion `binaryEncoding :: String -> [Int]`, die die Bitstring-Repräsentation einer Zeichenkette berechnet. Achten Sie bei der Erzeugung darauf, dass jedes Zeichen durch genau 8 Bit repräsentiert wird. Zu kurze Repräsentationen eines Zeichens sind mit führenden Nullen aufzufüllen.

Beispiel:

```
    binaryEncoding "hallo"
==> [0,1,1,0,1,0,0,0
    ,0,1,1,0,0,0,0,1
    ,0,1,1,0,1,1,0,0
    ,0,1,1,0,1,1,0,0
    ,0,1,1,0,1,1,1,1]
```

Die Zeilenumbrüche nach je acht Bit dienen ausschließlich zur besseren Visualisierung des Beispiels.

Nutzen Sie zur Lösung der Aufgabe die Funktion `ord :: Char -> Int`, die einem Zeichen seinen ASCII-Code zuordnet. Die Funktion `ord` ist nicht Teil der Haskell-Prelude und muss daher mit dem Befehl `import Data.Char(ord)` zu Beginn der Haskell-Datei importiert werden.

Hinweis: Zerlegen Sie das Problem geschickt in kleinere Teilprobleme und lösen Sie diese durch entsprechende Funktionen.

Aufgabe 5.5 Zipper

1. Definieren Sie eine Funktion `toZipper :: [a] -> ListZipper a`, die für eine Liste den das erste Listenelement referenzierende Zipper erzeugt.

2. Definieren Sie eine Funktion `fromZipper :: ListZipper a -> [a]`, die aus einem Zipper die repräsentierte Liste erzeugt.

3. Definieren Sie eine Funktion `update :: (a -> a) -> ListZipper a -> ListZipper a`, die den aktuell von dem Zipper referenzierten Wert mit der übergebenen Funktion anpasst.

Beispiel:

```
    fromZipper $ update (\_ -> 'u') $ forth $ forth $ toZipper "hallo"
==> "haulo"
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 6

Ausgabe: 22.11.2013, **Abgabe:** 29.11.2013

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 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 6.2 Listen

1. Implementieren Sie eine Haskell-Funktion `isSorted :: [Int] -> Bool`, die entscheidet, ob für die übergebene Liste aufsteigend sortiert ist.
2. Für jede Programmiersprache gibt es Konventionen, die die Benennung und Formatierung von Bezeichnern empfehlen. Zum Beispiel ist es in C üblich, Funktionsnamen aus Worten in Kleinbuchstaben zu bilden, die durch Unterstriche getrennt sind (z. B. `print_hello_world`).

   In Haskell und Java hat sich der sogenannte CamelCase durchgesetzt, bei dem die Worte nicht durch ein festes Zeichen getrennt, sondern die folgenden Worte mit einem Großbuchstaben beginnen. Das erste Wort beginnt jedoch immer mit einem Kleinbuchstaben (z.B. `printHelloWorld`).

   Offensichtlich sind beide Konventionen für nur aus einem Wort gebildete Bezeichner äquivalent.

   Implementieren Sie eine Haskell-Funktion `camelCase :: String -> String`, die einen Bezeichner in korrekter C-Syntax als Eingabe erhält und einen entsprechenden Bezeichner im CamelCase zurückgibt.

   Hinweis: Nutzen Sie die Funktion `toUpper :: Char -> Char`*, um einen Kleinbuchstaben in einen Großbuchstaben umzuformen. Diese Funktion muss am Anfang der Haskell-Datei mit dem Befehl* `import Data.Char(toUpper)` *importiert werden.*

Aufgabe 6.3 Faltungen

Schreiben Sie mit Hilfe von `foldr` eine Haskell-Funktion `filterLast :: (a -> Bool) -> [a] -> [a]`, so dass `filterLast p xs` in der Liste `xs` das letzte Element, auf das die Eigenschaft `p` nicht zutrifft, löscht.

Hinweis: Benutzen Sie Tupel, um bei der Faltung eine Zustandsinformation mitzuführen.

Aufgabe 6.4 Unendliche Listen

1. Ein pythagoreisches Tripel ist eine Menge von drei natürlichen Zahlen $a < b < c$, so dass

$$a^2 + b^2 = c^2.$$

   Zum Beispiel ist $3^2 + 4^2 = 9 + 16 = 25 = 5^2$. Schreiben Sie eine Haskell-Funktion `pyTriples :: [(Int,Int,Int)]`, die die unendliche Liste aller pythagoreischen Tripel erzeugt.

2. Mirp-Zahlen sind Primzahlen, die rückwärts gelesen ebenfalls eine Primzahl darstellen. Zum Beispiel ist 149 eine Mirp-Zahl, da sowohl 149 als auch 941 Primzahlen sind. Schreiben Sie eine Haskell-Funktion `mirp :: [Int]`, die die unendliche Liste aller Mirp-Zahlen berechnet.

   Hinweis: Auf den Vorlesungsfolien finden Sie eine Funktion `primes :: [Int]`, die die unendliche Liste aller Primzahlen erzeugt.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 7

Ausgabe: 29.11.2013, **Abgabe:** 06.12.2013

Aufgabe 7.1 Typinferenz

Gegeben sei folgender Datentyp:

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

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die Typen der folgenden Ausdrücke. Gehen Sie bei der Typbestimmung davon aus, dass `7 :: Int` ist.

1. `foldl (λacc x -> acc || (x `mod` 7 == 0)) False`
2. `Cons 7`
3. `foldr Cons Nil`

Aufgabe 7.2 Listen

Gegeben sei wieder der Datentyp `List a` aus Aufgabe 7.1. Für die Lösung dieser Aufgabe dürfen die Funktionen `map` und `foldr` aus der Standardbibliothek nicht benutzt 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.

Aufgabe 7.3 Modellierung

1. 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.

   Hinweis: Fügen Sie (analog zu dem Datentyp `List a` in Aufgabe 7.1) jeder Datentypdefinition ein `deriving Show` an, um Werte des Datentyps anzeigen zu können.
2. Geben Sie eine nichttriviale Beispielinstanz des Datentyps an.
3. Schreiben Sie eine Funktion, die einen Wert des Typs Firma erhält und das Gehalt der Mitarbeiter verdoppelt.

Aufgabe 7.4 Boolsche Algebra

Gegeben sei folgender Datentyp zur Repräsentation von boolschen Ausdrücken. Bei der Definition der folgenden Funktionen kann davon ausgegangen werden, dass für alle Variablen `Var i` die Bedingung $i \geq 0$ gilt.

```
data BExp = T                   -- True
          | F                   -- False
          | Var Int             -- Variable
          | Neg BExp            -- Negation
          | Disj BExp BExp      -- Disjunktion
          | Conj BExp BExp      -- Konjunktion
          deriving Show         -- (notwenig zur Anzeige)
```

1. Definieren Sie eine Haskell-Funktion `eval :: (Int -> Bool) -> BExp -> Bool`, die einen boolschen Ausdruck in Abhängigkeit einer Variablenbelegung auswertet.

2. Eine andere Möglichkeit zur Kodierung der Variablenbelegung erfolgt mit einer Liste von boolschen Werten. Der Wert der Variablen `Var i` wird durch den Wert an der i-ten Stelle in der Liste bestimmt.

   Schreiben Sie eine Haskell-Funktion `evalListEnv :: [Bool] -> BExp -> Bool`, die eine Auswertung basierend auf einer Variablenbelegung in der beschriebenen Listenrepräsentation durchführt. Nutzen Sie zur Definition von `evalListEnv` die in Aufgabe 7.4.1 definierte Funktion `eval`.

3. Implementiern Sie eine Haskell-Funktion `environments :: Int -> [[Bool]]`, die für eine natürliche Zahl `n` alle möglichen Variablenbelegungen mit genau `n` Variablen erzeugt. Die Variablenbelegungen sollen wieder in der Listenrepräsentation erzeugt werden.

4. Implementieren Sie eine Haskell-Funktion `equiv :: BExp -> BExp -> Bool`, die überprüft, ob die beiden boolschen Ausdrücke für alle Variablenbelegungen zu dem gleichen Ergebnis ausgewertet werden.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 8

Ausgabe: 06.12.2013, **Abgabe:** 13.12.2013

Aufgabe 8.1 Typinferenz

Die folgende Aufgabe soll ohne Compilerunterstützung gelöst werden. Bestimmten Sie die allgemeinsten Typen der folgenden Ausdrücke bzw. Funktionen. Gehen Sie bei der Typbestimmung davon aus, dass `8 :: Int` ist.

1. `(/=8)`
2. `f v = foldl (λacc x -> acc || (x == v)) False`

Aufgabe 8.2 Liste von Punkten in Aktionsfolge überführen

Die folgenden Datentypen für Punkte, Pfade und Aktionen sind aus der Vorlesung bekannt:

```
type Point = (Float,Float)
type Path [Point]

data Action = Turn Float | Move Float deriving Show
```

Zudem liegen Funktionen zur Berechnung des Winkels relativ zu einem Winkel bzw. der Distanz zwischen zwei Punkten vor:

```
angle :: Float -> Point -> Point -> Float
angle winkel (x1,y1) (x2,y2) = (f (y2-y1) (x2-x1)*180/pi) - winkel where
  f 0 0 = atan2 0 1
  f x y = atan2 x y

distance :: Point -> Point -> Float
distance (x1,y1) (x2,y2) = sqrt ((x2-x1)^2+(y2-y1)^2)
```

Implementieren Sie eine Haskell-Funktion `makeActions :: Path -> [Action]`, die eine Liste von Punkten in die entsprechende Aktionsfolge überführt.

Aufgabe 8.3 Typklasse `Eq`

Gegeben sei der aus der Vorlesung bekannte Datentyp für Farben:

```
data RGB = RGB Int Int Int
```

Geben Sie eine Instanz der Typklasse `Eq` für den Datentyp `RGB` an, so dass zwei Farben genau dann gleich sind, falls die entsprechenden drei Farbkodierungen beider Farben übereinstimmen.

Aufgabe 8.4 Boolsche Ausdrücke

Wir betrachten in dieser Aufgabe den bereits aus Aufgabe 7.4 bekannten Datentyp für boolsche Ausdrücke:

```
data BExp = T | F | Var Int | Neg BExp | Disj BExp BExp
          | Conj BExp BExp
          deriving Show
```

Aufgabe 7.4 führte eine Form der Variablenbelegung mittels Listen ein. Die Belegung der Variablen *i* ist in dieser Variante an der *i*-ten Stelle in der Liste zu finden. Für beliebige boolsche Ausdrücke ergeben sich jedoch folgende Probleme:

- Mit der in Aufgabe 7.4.2 definierten Funktion `evalListEnv` kann kein boolscher Ausdruck ausgewertet werden, der negative Variablen (z.B. die Variable `Var (-1)`) enthält.
- Darüber hinaus spielt bei dem boolschen Ausdruck `Conj (Var 0) (Var 2)` die Belegung an der Stelle 1 keine Rolle, da keine Variable 1 in dem Ausdruck existiert.

In dieser Aufgabe soll schrittweise eine Funktion `minListRep :: BExp -> BExp` definiert werden, die einen beliebigen boolschen Ausdruck in die *minimale Listenform* bringt. Ein boolscher Ausdruck ist in minimaler Listenform, falls alle Variablen von 0 bis n für eine natürliche Zahl n in dem boolschen Ausdruck vorkommen. Es gibt offensichtlich nicht notwendigerweise eine eindeutige minimale Listenform.

*Hinweis: Für die Lösung dieser Aufgabe könnten sich einige Funktionen aus dem Modul **Data.List** als durchaus nützlich erweisen.*

1. Implementieren Sie eine Haskell-Funktion `vars :: BExp -> [Int]`, die alle in einem boolschen Ausdruck vorkommenden Variablen bestimmt, so dass keine Variable in der Ergebnisliste doppelt vorkommt.
2. Implementieren Sie eine Haskell-Funktion `sub :: Int -> Int -> BExp -> BExp`, so dass der Aufruf `sub x y bexp` in dem Term `bexp` alle Vorkommen der Variablen `Var x` durch `Var y` ersetzt.
3. Implementieren Sie eine Haskell-Funktion `subs :: [(Int,Int)] -> BExp -> BExp`, die eine Liste von Ersetzungen bekommt und diese sukzessive auf den boolschen Ausdruck anwendet.
4. Implementieren Sie eine Haskell-Funktion `mkMapping :: BExp -> [(Int,Int)]`, die für einen boolschen Ausdruck die vorzunehmenden Ersetzungen berechnet, die für eine Umformung des Ausdrucks in eine minimale Listenform nötig sind.

   Algorithmus: Bestimmten Sie zuerst alle Variablen des Eingabeausdrucks und alle Variablen `toBeUsed`, die im korrekt umbenannten Ausdruck vorkommen müssen. Partitionieren Sie die Variablenliste dann in eine Liste mit bereits korrekten und eine Liste `correct` mit noch zu ändernden Variablen `toBeChanged`. Verbinden Sie dann die Listen `toBeChanged` mit der Liste, die aus der Mengendifferenz von `toBeUsed` und `correct` hervorgeht, um die entsprechende Abbildung zu erhalten.
5. Setzen Sie die in dieser Aufgabe definierten Funktionen so zusammen, dass Sie eine Haskell-Funktion `minListRep :: BExp -> BExp` erhalten, die einen beliebigen boolschen Ausdruck in eine minimale Listenform bringt.

Hilfe: Mit dieser Funktion `isInMinimalListForm :: BExp -> Bool` können Sie überprüfen, ob Ihre Implementierung korrekt arbeitet:

```
isInMinimalListForm :: BExp -> Bool
isInMinimalListForm bexp = sort (vars bexp) `isPrefixOf` nats where
  nats = [0,1..]
```

Die Funktionen `sort` und `isPrefixof` befinden sich auch in dem Modul `Data.List`.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 9

Ausgabe: 20.12.2013, **Abgabe:** 10.01.2014

Aufgabe 9.1 Natürliche Zahlen und Typklassen

Die natürlichen Zahlen können wie folgt induktiv definiert werden: Die Null ist eine natürliche Zahl. Für eine natürliche Zahl ist die nächstgrößere ganze Zahl wieder eine natürliche Zahl. Gegeben sei dem entsprechend folgender Datentyp zur repräsentation natürlicher Zahlen:

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

+

1. Definieren Sie eine Instanz der Typklasse `Show` für `Nat`, indem Sie die Funktion `show :: a -> String` implementieren. Die Ausgabe soll die repräsentierte Zahl im Dezimalsystem zurückgeben, z.B. `show (Succ Zero) ==> "1"`.

2. Definieren Sie eine Instanz der Typklasse `Ord` für `Nat`, indem Sie die Funktion `(<=) :: a -> a -> Bool` implementieren. Da `Ord` eine Unterklasse von `Eq` ist, müssen Sie hierzu zunächst eine Instanz der Typklasse `Eq` für `Nat` definieren.

3. Informieren Sie sich in der Dokumentation[1] über die Typklasse `Enum`. Definieren Sie eine Instanz der Typklasse `Enum` für `Nat`, indem Sie die nötigen Funktionen der Typklasse definieren.

   Bei der Konvertierung von `Int` nach `Nat` für negative Zahlen und in dem Fall eines negativen Ergebnisses bei `pred` sollen Laufzeitfehler geworfen werden. Nutzen Sie dazu die Funktion `error :: String -> a`.

   Hinweis: Für den Typ `Int` ist auch eine Instanz der Typklasse `Enum` definiert, d.h. Sie können ihre Implementierungen der Funktionen `enumFrom`,`enumFromTo`, `enumFromThen` und `enumFromThenTo` gegen die Aufrufe der entsprechenden Funktionen für Werte des Typs `Int` testen.

Aufgabe 9.2 Boolsche Ausdrücke

Wir betrachten in dieser Aufgabe den bereits bekannten Datentyp für boolsche Ausdrücke mit folgender kleinen Variation: Die Variablen sind nicht mehr vom Typ `Int`, sondern von enem beliebigen Typ `a`.

```
data BExp a = T | F | Var a | Neg (BExp a) | Disj (BExp a) (BExp a)
            | Conj (BExp a) (BExp a) deriving Show
```

1. Definieren Sie eine Instanz der Typklasse `Eq` für den Typ `BExp a`.

[1] http://hackage.haskell.org/packages/archive/haskell2010/latest/doc/html/Prelude.html

2. Definieren Sie eine Funktion `varMap :: (a -> b) -> BExp a -> BExp b`, die die übergebene Funktion auf jede Variable anwendet.

Aufgabe 9.3 Zipper für Binärbäume

In dieser Aufgabe nutzen wir den aus der Vorlesung bekannten Datentyp `TreeZipper a`. Im EWS finden Sie eine Modulvorlage, die schon einige grundlegenden Typen und Funktionen aus den Vorlesungsfolien enthalt.

1. Definieren Sie Funktionen `getLeftChild, getRightChild, getUpper :: TreeZipper a -> Maybe a`, die den Wert am linken bzw. rechten Nachfolger an einer Zipperposition zurückgibt. Die Funktion `getUpper` gibt den Wert am Vorgängerknoten zurück.

   Bei allen drei Funktionen, kann es vorkommen, dass ein solcher Wert nicht existiert. So zum Beispiel wenn bei dem Aufruf von `getLeftChild` kein linker Nachfolger existiert. Die Funktionen geben in diesen Fällen `Nothing` zurück.

2. Definieren Sie eine Funktion `neighbours :: TreeNode a -> [a]`, die zuerst mit einem Zipper an die übergebene Stelle in dem Baum wandert und dann alle direkten Nachbarn des Knoten zurückgibt.

   Um an die Stelle in dem Baum zu gehen, nutzen Sie die vorgegebene Funktion `treeTozipper`. Nutzen Sie die in Aufgabenteil 1 definierten Funktionen zur Bestimmung der Nachbarn an einer Stelle.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 10

Ausgabe: 10.01.2014, **Abgabe:** 17.01.2014

Aufgabe 10.1 (5 Punkte) 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 Punkte)

2. Geben Sie für die Typen `WrapInt f` und `T f g` je zwei beliebige Werte an. (3 Punkte)

Aufgabe 10.2 (7 Punkte) 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`. (1 Punkt)

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)
   ```

   (4 Punkte)

   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`. (2 Punkte)

Aufgabe 10.3 (8 Punkte) Maybe-Monade

1. Implementieren Sie eine Haskell-Funktion, die eine Liste von `Int`-Paaren und eine Liste von Schlüsseln des Typs `Int` enthält. Die Funktion soll alle mit den Schlüsseln assoziierten Werte aus der Liste der Paare aufsummieren. Gehen Sie dabei davon aus, dass mit jedem Schlüssel nur maximal ein Wert assoziiert wird.

   Dabei kann es vorkommen, dass für einen Schlüssel kein Wert in der Paar-Liste vorkommt. In diesem Fall soll die Funktion `Nothing` zurückgeben.

   Der folgende Codeschnipsel enthält bereits einen Rahmen für die Funktionsdefinition. Die Funktion `sumUpAssociatedToValues` berechnet die Summe für die Schlüssel und eine Paar-Liste, indem die Hilfsfunktion `sumUp` mit den beiden Listen und dem Startwert `0` für die Summe übergeben wird.

   Implementieren Sie die Funktion `sumUp` und nutzen Sie die Monadeneigenschaften von `Maybe` zur Fehlerbehandlung aus.

```
sumUpAssociatedToValues :: [Int] -> [(Int,Int)] -> Maybe Int
sumUpAssociatedToValues = sumUp 0 where

  sumUp :: Int -> [Int] -> [(Int,Int)] -> Maybe Int
```

   Hinweis: Mit der Funktion `lookup :: Eq a => a -> [(a,b)] -> Maybe b` können Sie den assoziierten Wert aus einer Liste von Paaren erhalten. Die Funktion `lookup` gibt `Nothing` zurück, wenn für den Schlüssel kein Paar in der Liste enthalten ist. (4 Punkte)

   Beispielaufrufe:

```
sumUpAssociatedToValues [0]   [(1,2),(3,8)] ==> Nothing
sumUpAssociatedToValues []    [(1,2),(3,8)] ==> Just 0
sumUpAssociatedToValues [3]   [(1,2),(3,8)] ==> Just 8
sumUpAssociatedToValues [3,1] [(1,2),(3,8)] ==> Just 10
```

2. Beweisen Sie die ersten beiden Monadengesetze für den Datentyp `Maybe`. Zeigen Sie also, dass die Implementierung von »= für jeden Wert `x` vom Typ `Maybe a` Folgendes sicherstellt:

   (a) `return x >>= f = f x`

   (1 Punkt)

   (b) `f x >>= return = f x`

   (3 Punkte)

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 11

Ausgabe: 17.01.2014, **Abgabe:** 24.01.2014

Aufgabe 11.1 (5 Punkte) `[]-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.

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

Aufgabe 11.2 (7 Punkte) Fehlerbehandlung mit `MonadPlus`

Gegeben sei die Funktion `lookup`, die einen assoziierten Wert in einer Liste sucht:

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup _ [] = Nothing
lookup k ((a,b):xs)
  | k == a = Just b
  | k /= a = lookup k xs
```

Implementieren Sie eine Funktion

```
lookupMonadPlus :: (MonadPlus m,Eq a) => a -> [(a,b)] -> m b
```

die statt der Fehlerbehandlung mit `Maybe` eine Fehlerbehandlung in `MonadPlus` vornimmt.

Wird der monadische Typ in der Signatur von `lookupMonadPlus` bei einem Aufruf auf `Maybe` gesetzt, soll sich `lookupMonadPlus` wie `lookup` verhalten. Nutzen wir hingegen den Typ `[]` bei `lookupMonadPlus`, so sollen alle mit einem Wert assoziierten Werte zurückgegeben werden.

Beispielaufrufe:

```
lookupMonadPlus 3 [(3,1),(3,2),(5,9)] :: Maybe Int ==> Just 1
lookupMonadPlus 0 [(3,1),(3,2),(5,9)] :: Maybe Int ==> Nothing
lookupMonadPlus 3 [(3,1),(3,2),(5,9)] :: [Int]     ==> [1,2]
lookupMonadPlus 0 [(3,1),(3,2),(5,9)] :: [Int]     ==> []
```

Aufgabe 11.3 (8 Punkte) Programmbeweis

Gegeben seien folgende aus der Vorlesung bekannte 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

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 12

Ausgabe: 24.01.2014, **Abgabe:** 31.01.2014

Aufgabe 12.1 (3 Punkte) `Id`-Monade

Gegeben sei folgender Coderahmen:

```
data Id a = Id { get :: a }

instance Monad Id where
  return = Id
  Id x >>= f = f x

mapp :: (a -> b) -> [a] -> [b]
mapp f = get . mapId f
```

Implementieren Sie die Funktion `mapId :: (a -> b) -> [a] -> Id [b]`, so dass sich `mapp` wie die bekannte Funktion `map` aus der *Prelude* verhält.

Aufgabe 12.2 (5 Punkte) `MonadPlus`

Gegeben sei folgender Datentyp für binäre Bäume:

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

1. Definieren Sie eine Funktion

   ```
   satisfiesFromRootToLeaf :: (a -> Bool) -> Bintree a -> [a]
   ```

   die alle Blätter zurückgibt, auf deren Pfaden zu der Wurzel das übergebene Prädikat gilt.

   ```
   -- Beispiele
   satisfiesFromRootToLeaf (/=7) (Branch 4 (Leaf 1) (Leaf 9)) ==> [1,9]
   satisfiesFromRootToLeaf (/=4) (Branch 4 (Leaf 1) (Leaf 9)) ==> []
   satisfiesFromRootToLeaf (/=1) (Branch 4 (Leaf 1) (Leaf 9)) ==> [9]
   ```

2. Definieren Sie nun eine abstraktere Variante von `satisfiesFromRootToLeaf`, die nicht eine Liste von Werten des Typs `a` zurückgibt, sondern einen Wert vom Typ `m a` für ein `m`, das die Typklasse `MonadPlus` implementiert.

   Definieren Sie also eine Funktion

   ```
   satisfiesFromRootToLeafMplus :: MonadPlus m => (a -> Bool) -> Bintree a -> m a
   ```

   , die sich für `m` als `[]` wie die in Aufgabenteil 1 definierte Funktion verhält. Für `m` als `Maybe` soll die Funktion das erste Blatt zurückgeben mit der Eigenschaft, dass die Knoten auf dem Pfad zu der Wurzel die Bedingung erfüllen. Im dem Fall, dass es kein solches Blatt gibt, soll `Nothing` zurückgegeben werden.

Aufgabe 12.3 (8 Punkte) Galgenmännchen

Implementieren Sie das Spiel *Galgenmännchen*. 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 zur Auswahl eines zu erratenden Wortes erhalten.

Hinweis: Bei der Lösung dieser Aufgabe sind einige vordefinierte Funktionen aus der Haskell-Prelude nützlich.

Aufgabe 12.4 (4 Punkte) `Trans`-Monade

Betrachten Sie die Funktion `traceM :: [DefUse] -> [(String,Int)]` von den Folien. Implementieren Sie nun eine Funktion `traceMExpr :: [DefUseE] -> [(String,Int)]`, die die Verwendungsstellen der Varialben mit dem entsprechenden ausgewerteten arithmetischen Ausdruck zurückgibt. Dabei sei `DefUseE` wie folgt definiert:

```
data DefUseE = Def String Expr
             | Use String
```

Sie müssen dazu auch den Datentyp `Expr` von den Folien in Ihr Modul kopieren.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jens Lechner (*jens.lechner@cs.uni-dortmund.de*)
Pascal Hof (*pascal.hof@tu-dortmund.de*)
Niklas Klocke (*niklas.klocke@tu-dortmund.de*)
Michel Jakob (*michel.jakob@tu-dortmund.de*)

Wintersemester 2013/2014

Übungen zu Funktionaler Programmierung
Übungsblatt 13

Ausgabe: 31.01.2014, **Abgabe:** Besprechung der Lösungen am 7.2. im Tutorium

Für diesen Übungszettel ist im EWS ein Vorlagemodul `vorlage13.hs` zu finden, das den Datentyp `Trans` und die entsprechende Instanz für `Monad` enthält.

Aufgabe 13.1 Ein Stapel als Zustand

Gegeben sei folgender Datentyp für einen Stack. Das erste Listenelement repräsentiere dabei das oberste Element des Stacks.

```
data Stack a = Stack [a] deriving Show
```

In dieser Aufgabe werden die beiden Operationen `pop` und `push` auf Stacks als Zustandstransitionen in der `Trans`-Monade realisiert.

1. Implementieren Sie die Funktion `isEmpty :: Trans (Stack a) Bool`, die zurückgibt, ob der Stack leer ist.

2. Implementieren Sie die Funktion `push :: a -> Trans (Stack a) ()`, die den übergebenen Wert vom Typ `a` auf den Stack legt.

3. Implementieren Sie die Funktion `pop :: Trans (Stack a) a`, die das erste Element des Stacks entfernt und zurückgibt.

   Gehen Sie davon aus, dass diese Operation immer auf einem nicht-leeren Stack ausgeführt wird. Es ist also keine Fehlerbehandlung für den Fall des leeren Stacks durchzuführen. In der Praxis könnte man die Fehlerfälle mit der Funktion `isEmpty` abfangen.

Das folgende Beispiel soll das Zusammenspiel von `push` und `pop` verdeutlichen:

```
example :: Stack Int
example = snd (run m (Stack [1,2,3,4])) where
  m :: Trans (Stack Int) ()
  m = do
    pop
    x <- pop
    push 7
    push 9
    pop
    push x
```

Die Reduktion von `example` ergibt den Wert `Stack [2,7,3,4]`.

Aufgabe 13.2 Markieren von Bäumen

Gegeben sei folgender Datentyp für binäre Bäume:

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

Gegeben sei folgende Funktion `markInfix :: [a] -> Bintree () -> Bintree a`, die eine Liste von gewünschten Knotenwerten und einen Baum mit Knotentyp `()` erhält und den Baum in einer Infix-Traversion mit der Liste der gewünschten Knotenwerte markiert:

```
markInfix :: [a] -> Bintree () -> Bintree a
markInfix labels = fst . mark labels where

  mark :: [a] -> Bintree () -> (Bintree a,[a])
  mark ms     Empty          = (Empty,ms)
  mark (m:ms) (Leaf ())      = (Leaf m,ms)
  mark ms     (Branch () t1 t2) = (t', ms2) where

   t' = Branch m t1' t2'

   (t1',m:ms1) = mark ms t1
   (t2', ms2)  = mark ms1 t2
```

Beachten Sie, dass der Baum vom `Bintree ()` außer seiner Struktur keine Informationen trägt. Der Baum gibt also lediglich die Struktur des auszugebenen Baumes fest, während die Knotenwerte durch die Eingabeliste bestimmt werden.

Wir gehen davon aus, dass die Liste vom Typ `[a]` immer ausreichend viele Elemente enthält, so dass ein Baum beliebiger Größe markiert werden kann.

Implementieren Sie die Funktion `markSt` in folgendem Coderahmen, die eine monadischen Variante von der obigen Funktion `markInfix` darstellt. Sie können also die Funktion `markInfix` zur Überprüfung Ihrer Implementierung benutzen. Die Funktion `markSt` hat als Zustand eine Liste der noch zu vergebenen Knotenwerte.

```
markInfixSt :: [a] -> Bintree () -> Bintree a
markInfixSt labels t = fst (run (markSt t) labels) where

  markSt :: Bintree () -> Trans [a] (Bintree a)
```

Tipp: Implementieren Sie sich zur Definition von `markSt` eine Hilfsfunktion `popList :: Trans [a] a`, die das erste Element der Liste im Zustand entfernt und als Ergebnis zurückgibt.

Aufgabe 13.3 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 ersten 100 Gliedern der Catalan-Folge definiert werden. Bei der Berechnung der Glieder sollen Zwischenergebnisse durch Zugriffe auf das Feld anstelle durch Rekursion berechnet werden. Die folgende Funktion `catalanDyn :: Int -> Int` sollte durch diese Modifikation wesentlich effizienter als die rekursive Variante `catalan` sein.

```
catalanDyn :: Int -> Int
catalanDyn i = catalanArr ! i
```

Hinweis: Importieren Sie das Modul `Data.Array` für die Typklasse `Ix` und die grundlegenen Funktionen auf Feldern. Die Funktion `mkArray` von den Folien ist in diesem Modul jedoch nicht enthalten. Sie müssen diese Funktion selbst definieren oder die Modulvorlage aus dem EWS nutzen.