Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 0

Dieses Übungsblatt ist ein Präsenzblatt, d.h. die Aufgaben werden in den Übungen besprochen, aber es erfolgt keine Abgabe.

Aufgabe 0.1 *Berechnungsmodell*

Die Funktionen `succ` und `twice` sind wie folgt definiert:

```
succ :: Int -> Int
succ n = n + 1
twice f a = f (f a)
```

(a) Geben Sie den allgemeinsten Typ von `twice` an.

(b) Reduzieren Sie die Ausdrücke `twice succ 0` und `twice twice succ 0`. Beachten Sie: Der Ausdruck `f a b` kürzt `(f a) b` ab, da die Funktionsanwendung von links assoziiert.

(c) Überführen Sie `succ` und `twice` in λ-Notation.

(d) Reduzieren Sie den λ-Ausdruck `(\x -> (\y -> succ y+x+2)) 6 2`.

Aufgabe 0.2 *Listen*

Listen stelle eine grundlegende Datenstruktur in Haskell dar. Im Folgenden sollen Sie einige einfache Funktionen für Liste selbst entwickeln:

(a) Entwickeln Sie eine Funktion `idx`, die den kleinsten Index eines Elementes innerhalb einer Liste bestimmt. Wenn das Element in der Liste vorhanden ist, soll -1 zurückgeliefert werden.

  a) Geben Sie den Funktionstyp von `idx` an!

  b) Geben Sie eine Implementierung der Funktion an!

(b) Definieren Sie eine Funktion `insert`, die ein Element an einem angegeben Index in eine Liste einsortiert. Falls der Index größer als der maximale Index aller Elemente der Liste ist, soll das Element am Ende einsortiert werden.

  a) Welchen Funktionstyp hat die Funktion?

  b) Geben Sie die Definition der Funktion an!

(c) Gegeben sei die folgende Liste

`[ 4 2 5 6 7 1 ]`

Fügen Sie das Element 3 mit Hilfe von `idx`, `insert` und `succ` nach dem Element 6 in die Liste ein!

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 1

Abgabefrist: 28.10.2010, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Quizfragen:

Welche der folgenden Aussagen sind richtig, welche sind falsch? Warum?

(a) Eine Haskell-Funktion muss mit einem Kleinbuchstaben beginnen.

(b) In der Definition einer Haskell-Funktion muss immer der Typ angegeben werden.

(c) Die Listenfunktionen `head` und `last` haben das gleiche Laufzeitverhalten.

Aufgabe 1.1 *Funktionen* (10 Punkte)

(a) Definieren Sie eine Haskell-Funktion `umdrehen :: [a] -> [a]`, die eine Liste umdreht. Bei Eingabe der Liste $[1, 2, 3]$ soll also die Liste $[3, 2, 1]$ ausgegeben werden. (Es darf nicht die Haskell-Funktion `reverse` verwendet werden.) (1 Punkt)

(b) Definieren Sie die `umdrehen`-Funktion nun mit Hilfe der Funktion `foldl`. (Nennen Sie die neue Funktion `umdrehen2`) *(Tipp: Die Haskell-Funktion* `flip` *könnte hilfreich sein.)* (3 Punkte)

(c) 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. (3 Punkte)

(d) Die kleinste Zahl, die sich durch die Zahlen 1 bis 10 ohne Rest teilen lässt, ist 2520. Finden Sie mit Hilfe einer Haskell-Funktion die kleinste Zahl, die ohne Rest durch die Zahlen 1 bis 20 teilbar ist. (3 Punkte)

Aufgabe 1.2 *Listen* (10 Punkte)

(a) Schreiben Sie eine Haskell-Funktion `multReverse :: [a] -> [a]`, die eine Liste umkehrt und dabei die Listenelemente i mal hintereinander schreibt, wobei i der Index $+1$ des entsprechenden Listenelementes ist. Z.B. würde `multReverse [1,2,3,4]` die Liste `[4,4,4,4,3,3,3,2,2,1]` als Ergebnis haben.
(Es darf nicht die Haskell-Funktion `reverse` verwendet werden.) (3 Punkte)

(b) Implementieren Sie eine Haskell-Funktion

```
graphF :: (Int -> Int) -> Int -> Int -> [(Int,Int)]
```

so dass `graphF` bei Anwendung auf eine Funktion `f :: Int -> Int` und zwei ganze Zahlen n und m die Liste `[(n, f n), (n+1, f (n+1)), ... , (m, f m)]` berechnet. Für $n > m$ soll das Ergebnis die leere Liste sein. (3 Punkte)

(c) Drücken Sie die `filter`-Funktion

```
filter :: (a -> Bool) -> [a] -> [a]
filter f (a:s) = if f a then a:filter f s else filter f s
filter f _ = []
```

mit Hilfe der Funktionen `takeWhile` und `dropWhile` aus. Schreiben Sie also eine Haskell-Funktion `filt`, die sich wie obige Funktion verhält und benutzen Sie dazu die beiden genannten Funktionen. (2 Punkte)

(d) Entwickeln Sie eine Haskell-Funktion `listSum :: [Int] -> [Int]`. Diese Funktion soll eine Liste von Integern L auf eine Liste von Integern $listSum(L)$ abbilden, wobei $listSum(L)$ als i-tes Element die Summe der ersten $n - i$ Elemente von L enthält. Dabei ist n die Länge von L und $0 \leq i \leq n - 1$. Z.B. wäre `listSum [1,2,3,4,5]` die Liste `[15,10,6,3,1]`. (Selbstverständlich darf auch hier die Haskell-Funktion `reverse` nicht verwendet werden, genausowenig wie die Haskell-Funktion `sum`.) (2 Punkte)

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 2

Abgabefrist: 4.11.2010, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 2.1 *Reduktion* (4 Punkte)

Gegeben seien die beiden Funktionen:

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

(a) Reduzieren Sie den Ausdruck `sumr [1,2,3,4]`. 2 Punkte

(b) Reduzieren Sie den Ausdruck `suml [1,2,3,4]`. 2 Punkte

Geben Sie aussagekräftige Zwischenergebnisse an, die den Lösungsweg dokumentieren.

Aufgabe 2.2 *Schleifen* (9 Punkte)

(a) For-Schleife
Gegeben sei folgende for-Schleife.

```
x = x0
for (i = i0, i ≤ i1, i++) x = f(x,i)
```

Geben Sie einen Haskell-Ausdruck mit `foldl` an, der x berechnet. 3 Punkte

(b) While-Schleife (1)
Gegeben sei folgende while-Schleife.

```
x = x0, i = i0
while a(i) do {
    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. 3 Punkte

(c) While-Schleife (2)
Gegeben sei folgende while-Schleife.

```
x = x0, i = i0
while a(i) do {
    x = f(x,i), i = g(i)}
```

Benutzen Sie `foldl` jetzt, um eine Funktion

```
foldWhile1 ::
(a -> Int -> a) -> (Int -> Bool) -> (Int -> Int) -> a -> Int -> a
```

zu definieren, so dass sich `x = foldWhile1 f a g x0 i0` berechnen lässt. 3 Punkte

Aufgabe 2.3 *Collatz-Problem* (7 Punkte)

Informieren Sie sich über das Collatz-Problem (`http://de.wikipedia.org/wiki/Collatz-Problem`).

(a) Schreiben Sie eine Haskell-Funktion `collatzFolge :: Integer -> [Integer]`, die für eine natürliche Zahl die (unendliche) Collatz-Folge liefert. 4 Punkte

(b) Schreiben Sie eine Haskell-Funktion `max10Schritte :: [Integer]`, die alle natürlichen Zahlen zurückgibt, für die die Collatz-Folge weniger als 10 Schritte benötigt, bis zuerst eine 1 auftaucht. 3 Punkte

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 3

Abgabefrist: 11.11.2010, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Quizfragen:

Welche der folgenden Aussagen sind richtig, welche sind falsch? Warum?

(a) `foldl` ist vom Speicherplatzverhalten effizienter als `foldr`.

(b) `not . not True` liefert den Booleschen Wert `True`.

(c) `1 + succ $ 3` liefert einen Typfehler.

Aufgabe 3.1 *Rekursion und Listen* (10 Punkte)

(a) Gegeben seien zwei Funktionen

```
f :: Int -> Int
f 0 = 0
f n = f (n-1) + n
g :: Int -> Int
g n = div (n*(n+1)) 2
```

Zeigen Sie, dass `f n = g n` für $n \geq 0$ gilt. (2 Punkte)

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

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

Zum Beispiel ist $3^2 + 4^2 = 9 + 16 = 25 = 5^2$. Es gibt genau ein pythagoreisches Tripel, für das $a+b+c = 1000$. Schreiben Sie eine Haskell-Funktion `tripel :: (Integer,Integer,Integer)`, die dieses Tripel (a, b, c) ausgibt. (2 Punkte)

(c) 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. Erzeugen Sie mittels Listenkomprehension die unendliche Liste aller Mirp-Zahlen.

(3 Punkte)

(d) 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. *(Tipp: Benutzen Sie Tupel, um bei der Faltung eine Zustandsinformation mitzuführen.)*

(3 Punkte)

Aufgabe 3.2 *Datentypen* (10 Punkte)

(a) Definieren Sie einen Datentyp `BoolExp`, der aussagenlogische Formeln modelliert. Dabei sind aussagenlogische Formeln wie folgt definiert:

- Jede Variable $X_0, X_1 \ldots$ ist eine aussagenlogische Formel.
- Sind φ, ψ aussagenlogische Formeln, dann sind auch $(\neg\varphi), (\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi)$ und $(\varphi \leftrightarrow \psi)$ aussagenlogische Formeln.

(4 Punkte)

(b) Schreiben Sie eine Funktion `knf :: BoolExp -> BoolExp`, die eine aussagenlogische Formel in eine äquivalente aussagenlogische Formel in konjunktiver Normalform umwandelt. Ein Algorithmus, der das gewährleistet, sieht für eine Formel F folgendermaßen aus :

- Ersetze in F jedes Vorkommen einer Teilformel der Bauart

$$(G \rightarrow H) \text{ durch } ((\neg G) \vee H)$$

$$(G \leftrightarrow H) \text{ durch } ((G \wedge H) \vee ((\neg G) \wedge (\neg H)))$$

  bis keine derartige Formel mehr vorkommt.

- Ersetze jedes Vorkommen einer Teilformel der Bauart

$$(\neg(\neg G)) \text{ durch } G$$

$$(\neg(G \wedge H)) \text{ durch } ((\neg G) \vee (\neg H))$$

$$(\neg(G \vee H)) \text{ durch } ((\neg G) \wedge (\neg H))$$

  bis keine derartige Teilformel mehr vorkommt.

- Ersetze jedes Vorkommen einer Teilformel der Bauart

$$(G \vee (H \wedge I)) \text{ durch } ((G \vee H) \wedge (G \vee I))$$

$$((H \wedge I) \vee G) \text{ durch } ((H \vee G) \wedge (I \vee G))$$

  bis keine derartige Teilformel mehr vorkommt.

(3 Punkte)

(c) Schreiben Sie einen `BoolExp`-Interpreter `evalBE :: BoolExp -> [Bool] -> Bool`.

(3 Punkte)

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 4

Abgabefrist: 18.11.2010, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Quizfragen:

Welche der folgenden Aussagen sind richtig, welche sind falsch? Warum?

(a) Für Infix-Konstruktoren gelten dieselben syntaktischen Regeln wie für Infix-Funktionen.

(b) Jeder Aufruf der Funktionen `map` oder `filter` lässt sich äquivalent als Listenkomprehension schreiben.

Aufgabe 4.1 *Boolsche Ausdrücke und Induktion* (11 Punkte)

Gegeben sei folgender Datentyp aussagenlogischer Formeln

```
data BoolExp = True_ | False_ | X Int | Not BoolExp | BoolExp :& BoolExp |
               BoolExp :| BoolExp | BoolExp :=> BoolExp | BoolExp :<=> BoolExp
               deriving Show

infixr 1 :=>, :<=>
infixr 2 :|
infixr 3 :&
```

Die Syntax `infixr ...` gibt hier an, dass die Infix-Konstruktoren `:=>`, `:<=>`, `:|` und `:&` rechtsassoziativ sind und dass `:&` unter ihnen die höchste Priorität hat, `:=>` und `:<=>` die niedrigste. Dadurch können also Klammern gespart werden.
Weiterhin sei die Auswertung einer Formel gegeben:

```
evalBE :: BoolExp -> [Bool] -> Bool
evalBE True_ b = True
evalBE False_ b = False
evalBE (X i) b = b!!i
evalBE (Not e) b = not $ evalBE e b
evalBE (e :& e') b = evalBE e b && evalBE e' b
evalBE (e :| e') b = evalBE e b || evalBE e' b
evalBE (e :=> e') b = evalBE (Not e :| e') b
evalBE (e :<=> e') b = evalBE ((e :& e') :| (Not e :& Not e')) b
```

(a) Es sei $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ der Binominalkoeffizient. Der Wert dieser Funktion gibt die Anzahl der k-elementigen Teilmengen einer n-elementigen Menge an. Zeigen Sie, durch Induktion über n, dass die Summe der Binomialkoeffizienten $\binom{n}{k}$ über alle $0 \leq k \leq n$ mit der Anzahl 2^n aller Teilmengen einer n-elementigen Menge übereinstimmt. (3 Punkte)

(b) Schreiben Sie eine Funktion `funcToFormel :: Int -> ([Bool] -> Bool) -> BoolExp`, sodass `funcToFormel k f` für die Einschränkung einer Funktion `f :: [Bool] -> Bool` auf k-elementige Listen eine ihr entsprechende aussagenlogische Formel zurückgibt.

Sei z. B.

```
f :: [Bool] -> Bool
f [x0,x1,x2] = x0 || (x1 && x2)
f _ = False
```

Dann soll `funcToFormel 3 f` den Ausdruck `X 0 :| (X 1 :& X 2)` oder einen hiermit äquivalenten Ausdruck liefern.

(4 Punkte)

(c) Schreiben Sie einen Äquivalenztest für Boolesche Formeln, also eine Funktion

`aequTest :: BoolExp -> BoolExp -> Bool`,

die `True` ausgibt, wenn die beiden Formeln semantisch äquivalent sind und sonst `False`.

(4 Punkte)

Aufgabe 4.2 *Bäume* (9 Punkte)

Folgender Datentyp eignet sich, um Bäume mit beliebigem Knotengrad zu modellieren.

```
data Tree a = T a [Tree a]
```

(a) Schreiben Sie eine Haskell-Funktion `filterKnoten :: (a -> Bool) -> Tree a -> [a]`, sodass `filterKnoten p tree` alle Knoteneinträge des Baums tree ausgibt, die das Prädikat `p` erfüllen.

(3 Punkte)

(b) Schreiben Sie eine Haskell-Funktion `zaehleBlaetter :: Tree a -> Integer`, die alle Blätter des Baumes zählt. Ein Blatt ist ein Knoten im Baum, der keine Kinder hat. (3 Punkte)

(c) Ein Baum gilt als balanciert, wenn sich die Höhen aller Teilbäume, die einen gemeinsamen Vaterknoten haben, um höchstens 1 unterscheiden. Schreiben Sie eine Haskell-Funktion `isBalanced :: Tree a -> Bool`, die für einen Baum angibt, ob dieser balanciert ist.

(3 Punkte)

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 5

Abgabefrist: 25.11.2011, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heien. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erluterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 5.1 *Boolsche Ausdrücke* (6 Punkte)

Gegeben sei folgender Datentyp aussagenlogischer Formeln

```
data BoolExp = True_ | False_ | X Int | Not BoolExp | BoolExp :& BoolExp |
               BoolExp :| BoolExp | BoolExp :=> BoolExp | BoolExp :<=> BoolExp

infixr 1 :=>, :<=>
infixr 2 :|
infixr 3 :&
```

(a) Definieren Sie für den Datentyp `BoolExp` eine Instanz der Typklasse `Show`. (2 Punkte)

(b) Definieren Sie für den Datentyp `BoolExp` eine Instanz der Typklasse `Read`. (4 Punkte)

Aufgabe 5.2 *Bäume* (6 Punkte)

Folgender Datentyp eignet sich, um Bäume mit beliebigem Knotengrad zu modellieren.

```
data Tree a = T a [Tree a]
```

(a) Schreiben Sie eine Haskell-Funktion `foldTree :: (a -> [b] -> b) -> Tree a -> b` zur Faltung von Bäumen. Die Faltungsfunktion vom Typ `a -> [b] -> b` soll den Wert am aktuellen Knoten `a` und die Ergebnisse für die Nachfolger des Knotens `[b]` zu einem neuen Ergebnis `b` zusammensetzen. (2 Punkte)

(b) Schreiben Sie eine Haskell-Funktion `blattString :: Show a => Tree a -> String` mit Hilfe von `foldTree`, die jedes Blatt in einen `String` überführt und diese `String`s konkateniert. (2 Punkte)

(c) Schreiben Sie eine Haskell-Funktion `maxTree :: Ord a => Tree a -> a` mit Hilfe von `foldTree`, die den maximalen Wert in dem Baum zurückgibt. (2 Punkte)

Aufgabe 5.3 *Natürliche Zahlen und Typklassen* (8 Punkte)

Gegeben sei folgender Datentyp natürlicher Zahlen

```
data Nat = Null | Succ Nat
```

(a) Definieren Sie eine Instanz der Typklasse `Show` für `Nat`, die die repräsentierte natürliche Zahl ausgibt. Zum Beispiel soll `show (Succ Null)` den Wert `"1"` ausgeben. (1 Punkt)

(b) Definieren Sie eine Instanz der Typklasse `Num` für `Nat`, indem Sie die Funktionen `(+)`,`(*)`,`(-)`, `abs`, `signum` und `fromInteger` implementieren. Dazu müssen Sie auch noch eine Instanz der Typklasse `Eq` definieren. Details über die Funktionen entnehmen Sie der Dokumentation[1]. In dem Fall eines negativen Ergebnisses bei der Subtraktion soll `Null` zurückgegeben werden, genauso sollen negative Zahlen bei `fromInteger` auf `Null` abgebildet werden. (3 Punkte)

(c) Definieren Sie eine Instanz der Typklasse `Read` für `Nat`, so dass sowohl Strings der Form `"17"` als auch `"Succ (Succ Null)"` eingelesen werden können. (4 Punkte)

[1] http://haskell.org/ghc/docs/latest/html/libraries/base/Prelude.html

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 6

Abgabefrist: 2.12.2011, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heien. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erluterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.
Ebenfalls im EWS (fp6.hs) findet sich ein Frame des aktuellen Zettels.

Aufgabe 6.1 *Natürliche Zahlen und Typklassen (1)* (7 Punkte)

Gegeben sei folgender Datentyp natürlicher Zahlen vom letzten Übungszettel:

```
data Nat = Null | Succ Nat
```

(a) 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. (2 Punkte)

(b) Informieren Sie sich in der Dokumentation[1] über die Typklasse `Enum`. Definieren Sie eine Instanz der Typklasse `Enum` für `Nat`, indem Sie alle 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`.

Die folgenden Beispielaufrufe deuten das gewünschte Verhalten der Enumerationsfunktionen an:

```
enumFrom (Succ Null) ==>
  [Succ Null
  ,Succ (Succ Null)
  ,Succ (Succ (Succ Null)),..

enumFromTo (Succ Null) (Succ (Succ (Succ Null))) ==>
  [Succ Null
  ,Succ (Succ Null)
  ,Succ (Succ (Succ Null))]

enumFromThen Null (Succ (Succ Null)) ==>
  [Null
  ,Succ (Succ Null)
  ,Succ (Succ (Succ (Succ Null)))
  ,Succ (Succ (Succ (Succ (Succ (Succ Null))))),...

enumFromThenTo Null
  (Succ (Succ (Null)))
  (Succ (Succ (Succ(Succ (Succ (Succ Null))))))
  ==>
  [Null
  ,Succ (Succ Null)
  ,Succ (Succ (Succ (Succ Null)))
  ,Succ (Succ (Succ (Succ (Succ (Succ Null)))))]
```

(5 Punkte)

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

Aufgabe 6.2 *Natürliche Zahlen und Typklassen (2)* (7 Punkte)

Definiert man auf einer Halbordnung $(A, \preceq)$ die zweistelligen Operationen $\wedge$ (meet) und $\vee$ (join) durch die Festlegungen:

- $a \wedge b =$ Infimum von a und b bezüglich $\preceq$
- $a \vee b =$ Supremum von a und b bezüglich $\preceq$

So bildet A zusammen mit diesen Operationen ein Verband. Wir definieren eine Typklasse für Verbände:

```
class Lattice a where
  (<:) :: a -> a -> Bool
  join :: a -> a -> a
  meet :: a -> a -> a
```

Ein solcher Verband heißt beschränkt, wenn er bezüglich $\preceq$ zusätzlich ein kleinstes ($\bot$) und ein größtes ($\top$) Element besitzt. Wir definieren eine Typklasse für beschränkte Verbände:

```
class Lattice a => BoundedLattice a where
  top :: a
  bot :: a
```

Auf den natürlichen Zahlen aus Aufgabe 1 kann nun eine Halbordnung $\preceq$ definiert werden durch:

$$\forall n, m \in \mathbb{N} : n \preceq m \text{ genau dann, wenn } \exists k \in \mathbb{N} : n \cdot k = m$$

Definieren Sie Instanzen von `Lattice` und `BoundedLattice` für die natürlichen Zahlen `Nat`. Beachten Sie, dass 0 bezüglich $\preceq$ das größte Element von $\mathbb{N}$ ist.

Aufgabe 6.3 *Graphen* (6 Punkte)

Gegeben sei folgender Datentyp für die Adjazenzlistendarstellung von Graphen:

```
type Graph a = [(a,[a])]
```

(a) Definieren Sie eine Funktion `isSymm :: Eq a => Graph a -> Bool`, die überprüft, ob ein Graph symmetrisch ist. Ein Graph ist symmetrisch, wenn für je zwei Knoten u und v entweder keine oder beide Kanten (u, v) und (v, u) existieren. (3 Punkte)

(b) Definieren Sie eine Funktion `mkSymm :: Eq a => Graph a -> Graph a`, die aus einem beliebigen Graphen einen symmetrischen erzeugt, indem die fehlenden Kanten eingefügt werden. (3 Punkte)

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 7

Abgabefrist: 09.12.2011, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 7.1 *Modallogik und Kripke-Strukturen* (10 Punkte)

(a) Geben Sie einen Datentyp für MF (Modallogische Formeln) an und Typen für Kripke-Strukturen und die Menge Store. (5 Punkte)

(b) Programmieren Sie die Auswertungsfunktion `eval` unter Verwendung der Funktionen `lfp` und `gfp`. Mengen sollen dabei stets als Listen realisiert werden. (5 Punkte)

Aufgabe 7.2 *Minimum Spanning Tree* (10 Punkte)

Ein Graph mit ganzzahligen Kantengewichten kann durch folgenden Datentyp modelliert werden.

```
type LGraph a = ([a], [(a, Integer, a)])
```

Hier ist die erste Komponente des Tupels die Liste der Knoten des Graphen und die zweite Komponente die gewichteten Kanten des Graphen. Da die Graphen ungerichtet sein sollen, können Sie davon ausgehen, dass zwischen zwei Knoten `a1` und `a2` nur jeweils höchstens eine Kante angegeben ist (also entweder `(a1,i,a2)` oder `(a2,i,a1)` oder gar keine Kante). Außerdem kann vorausgesetzt werden, dass die Algorithmen nur mit zusammenhängenden Graphen aufgerufen werden, das muss also vorher nicht noch überprüft werden.

(a) Implementieren Sie den Algorithmus von Kruskal für ungerichtete, zusammenhängende, endliche und kantengewichtete Graphen.

```
kruskal :: (Ord a, Eq a) => LGraph a -> [(a, Integer, a)].
```

(5 Punkte)

(b) Implementieren Sie den Algorithmus von Prim für ungerichtete, zusammenhängende, endliche und kantengewichtete Graphen.

```
prim :: (Ord a, Eq a) => LGraph a -> [(a, Integer, a)].
```

(5 Punkte)

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 8

Abgabefrist: 16.12.2011, 10:00 Uhr

Für dieses Übungsblatt kann eine Frame-Datei geladen werden. Sie befindet sich im EWS unter fp8.hs. Ebenfalls dort findet sich auch die Datei Painter.hs, die für Aufgabe 1 benötigt wird.
Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

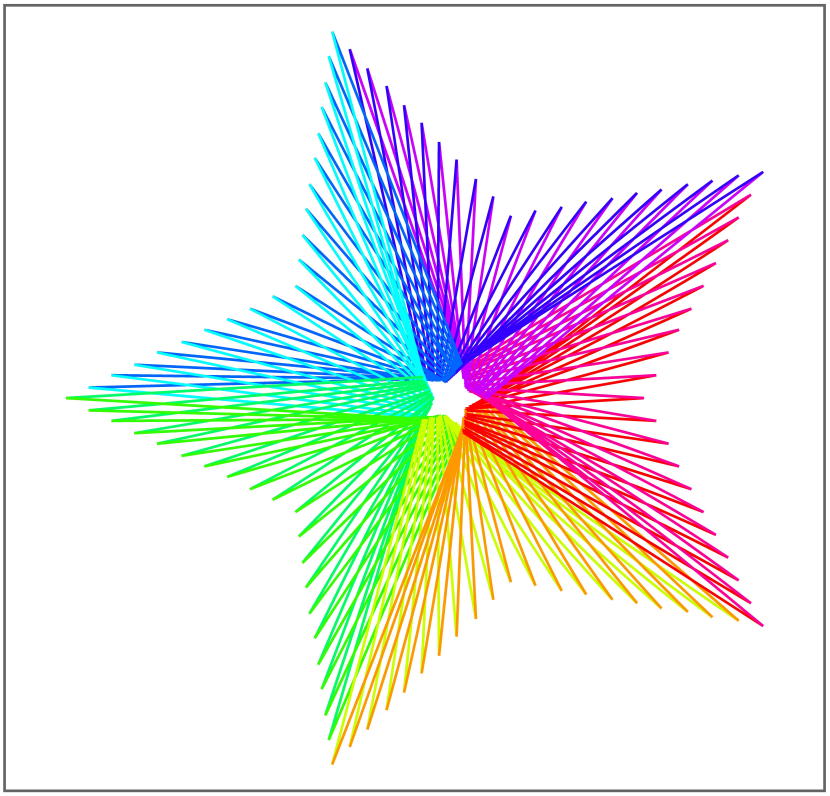
In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 8.1 *Painter* (10 Punkte)

Programmieren Sie eine Haskell-Funktion `morphing :: Int -> Curves -> Curves -> Curves` mit folgender Bedeutung: `morphing(n)(g)(g')` erzeugt aus zwei Graphen `g` und `g'` eine Graphenliste $gs = [g0, g1, \ldots gn]$, die aus $g0 = g, gn = g'$ und äquidistanten Zwischenstufen zwischen `g` und `g'` besteht, die mit `combine` (siehe Painter.hs) zu einer Kurve zusammengefügt werden soll.
Es wird vorausgesetzt, dass `paths(g)` und `paths(g')` aus gleichvielen Wegen bestehen und für alle $0 \leq i < |paths(g)|$ die Wege `paths(g)!!i` und `paths(g')!!i` aus gleichvielen Punkten. Das gilt dann auch für je zwei Graphen der Liste `gs`, die wie folgt berechnet wird:
Für alle $0 \leq i \leq n$ und alle Paare (p, q) sich entsprechender Punkte von `g` bzw. `g'` hat der p (oder q) entsprechende Punkt von `gi` den Wert $(1-\frac{i}{n})*p+\frac{i}{n}*q$. Da p und q den Typ `Point` haben, bezeichnen $+$ und $*$ hier die Addition bzw. Skalarmultiplikation im $\mathbb{R}^2$. Außerdem muss die Standardfunktion `float : Int -> Float` verwendet werden, um ganze in reelle Zahlen umzuwandeln.
Legen Sie in Ihrem Homeverzeichnis einen Ordner mit Namen PainterPix an, importieren Sie Painter.hs in Ihr Programm und verwenden Sie combine und zipCurves in morphing (siehe Painter.pdf). Testen Sie morphing auf Graphen, welche die o.g. Voraussetzungen erfüllen. Z.B. sollte mit

```
morph n = combine [morphing n poly1 poly2, morphing n poly2 $ turn 72 poly1]
          where poly1 = poly 5 [4,44] 12111
                poly2 = turn 36 $ scale 0.5 poly1
```

der Aufruf `drawC (morph 10)` (im folgenden Dialog kann mit z.B. 5 5 svg bestätigt werden, dann nur noch Enter) folgenden Graphen in die Datei PainterPix/poly.svg malen:



Aufgabe 8.2 *Anzahl der Wege in Graphen* (10 Punkte)

(a) Schreiben Sie eine zu `paths` analoge Funktion `numbers :: Mat Bool -> Mat Int` einschließlich einer zu `iniPaths` analogen Funktion `iniNums :: Pos -> Int` mit folgender Bedeutung: Für alle Knoten i, j eines als Boolesche Matrix dargestellten Graphen `g` liefert `mat(numbers(g))(i,j)` die Anzahl der Wege von i nach j in `g`.

Es wird vorausgesetzt, dass `g` kreisfrei ist und die Knoten von `g` positive natürliche Zahlen sind. Die Lösung erfordert eine andere `Int`-Instanz der Typklasse Semiring als die auf den Vorlesungsfolien genannte: `add` und `mul` müssen als ganzzahlige Addition bzw. Multiplikation definiert werden. (6 Punkte)

(b) Schreiben Sie eine zur `Mat(Path)`-Instanz von `Show` (s. Vorlesungsfolien) analoge `Mat(Int)`-Instanz von `Show`, so dass z.B. der Aufruf `numbers $ Mat(5,5) $ \ (i,j) -> i < j` die folgende Ausgabe liefert:

```
1 - 1 -> 1
1 - 1 -> 2
1 - 2 -> 3
1 - 4 -> 4
1 - 8 -> 5
2 - 1 -> 2
2 - 1 -> 3
2 - 2 -> 4
2 - 4 -> 5
3 - 1 -> 3
3 - 1 -> 4
3 - 2 -> 5
4 - 1 -> 4
4 - 1 -> 5
5 - 1 -> 5
```

(4 Punkte)

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 9

Abgabefrist: 06.01.2012, 10:00 Uhr

Für dieses Übungsblatt kann eine Frame-Datei geladen werden. Sie befindet sich im EWS unter fp9.hs.
Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Grobuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heien. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 9.1 *Funktoren* (6 Punkte)

Vereinfacht dargestellt sind Funktoren Container, auf deren Elemente Funktionen angewandt werden können. Die Typklasse `Functor` bietet eine Schnittstelle für Funktoren:

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

(a) Gegeben sei der folgende Datentyp für nichtleere Listen:

```
data NonEmptyList a = Singleton a | Cons a (NonEmptyList a)
```

Definieren Sie eine Instanz der Typklasse `Functor` für `NonEmptyList`, so dass die Funktion `fmap f` die Funktion `f` auf jedes Listenelement anwendet. (3 Punkte)

(b) Gegeben sei der folgende Datentyp für Binärbäume:

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

Definieren Sie eine Instanz der Typklasse `Functor` für `BinTree`, so dass die Funktion `fmap f` die Funktion `f` auf jeden Knotenwert anwendet. (3 Punkte)

Aufgabe 9.2 *Monaden* (4 Punkte)

Die mathematische Struktur einer Monade wird in Haskell durch die folgende Typklasse `Monad` abgebildet:

```
infixl 1  >>, >>=
class  Monad m  where
    (>>=)            :: m a -> (a -> m b) -> m b
    (>>)             :: m a -> m b -> m b
    return           :: a -> m a
    fail             :: String -> m a

    m >> k           =  m >>= \_ -> k
```

Für die Definition einer Instanz müssen die Funktionen so implementiert werden, dass die folgenden Regeln (Monadengesetze) eingehalten werden:

```
return a >>= k              = k a
m >>= return                = m
m >>= (\x -> k x >>= h)     = (m >>= k) >>= h
```

Gegeben sei nun der Datentyp `Maybe a` inklusive der Instanz für die Typklasse `Monad`:

```
data Maybe a = Nothing | Just a

instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing

    (Just _) >>  k      = k
    Nothing  >>  _      = Nothing

    return              = Just
    fail _              = Nothing
```

Zeigen Sie, dass diese Typklasseninstanz wirklich eine monadische Struktur darstellt, indem Sie die Gültigkeit der drei Monadengesetze zeigen. (4 Punkte)

Aufgabe 9.3 *Simulation einer Registermaschine* (10 Punkte)

Eine Registermaschine (kurz: RM) besteht aus einer zentralen Recheneinheit, einem Speicher und aus einem Programm.
Die zentrale Recheneinheit ihrerseits besteht aus zwei Registern: dem Befehlszähler und dem Akkumulator.
Der Speicher enthält unendlich viele Register: $R_1, R_2, \ldots$ mit den Adressen $1, 2, \ldots$. Der Akkumulator hat die Adresse 0.
Ein RM-Programm besteht aus einer endlichen Folge von RM-Befehlen, die mit 1 beginnend aufsteigend durchnummeriert sind. Wir legen die folgenden RM-Befehle zugrunde:

(a) Ein- und Ausgabebefehle:

`LOAD` i

`STORE` i

(b) Arithmetische Befehle mit Konstanten:

`CLOAD` i

`CADD` i

`CSUB` i

`CMULT` i

`CDIV` i

(c) Arithmetische Befehle:

`ADD` i

`SUB` i

`MULT` i

`DIV` i

(d) Sprungbefehle:

`GOTO` j

`JZERO` j

`END`

```
data RMcom =
     LOAD Int | STORE Int | CLOAD Integer | CADD Integer | CSUB Integer |
     CMULT Integer | CDIV Integer | ADD Int | SUB Int | MULT Int |
     DIV Int | GOTO Integer | JZERO Integer | END
     deriving(Eq, Show, Read)

type RMprog = Integer -> RMcom
```

Für die Datentypen der Register und des Registermaschinenzustands vereinbaren wir:

```
type Register = Int -> Integer

data RMstate = State {progr :: RMprog, pc :: Integer, reg :: Register, maxReg :: Int}
```

`maxReg` ist dabei die größte im RM-Programm verwendete Registernummer, `pc` ist die Nummer des nächsten auszuführenden Programmbefehls.

(a) Schreiben Sie eine Instanz von `Show` für `RMstate`, mit der die Registerinhalte eines Registermaschinenzustands ausgegeben werden können. (Es sind natürlich nur Registerinhalte von Registern auszugeben, die im RM-Programm benutzt werden.) (2 Punkte)

(b) Schreiben Sie nun eine Haskell-Funktion `step: RMstate -> RMstate`, die den nächsten Befehl des RM-Programms ausführt. (4 Punkte)

(c) Schreiben Sie schließlich eine Haskell-Funktion `execute: RMprog -> Integer`, die bei Eingabe eines RM-Programms dieses ausführt und das Resultat der Berechnung (d. h. den Inhalt des Akkumulators) ausgibt. (4 Punkte)

(In der Datei `fp9.hs` wird ein RM-Programm zu Berechnung der Potenz n^m für $m \geq 0$ zur Verfügung gestellt.)

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 10

Abgabefrist: 13.01.2012, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 10.1 *Partielle Funktionen* (9 Punkte)

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) Formulieren Sie in der `do`-Notation.

(c) Schreiben Sie die Funktion als `Bind`, also mit `>>=`.

Aufgabe 10.2 *Partielles Einfügen in einen Suchbaum* (4 Punkte)

Folgender Datentyp eignet sich, um Suchbäume darzustellen.

```
data Bintree a = Empty | Fork (Bintree a) a (Bintree a)
```

Gegeben seien die partiellen Funktionen, die die Elemente 1, bzw. 10 in einen Suchbaum vom Typ `Bintree Int` einfügen. Dabei soll das Ergebnis undefiniert sein, wenn das Element bereits im Suchbaum enthalten ist.

```
insertTree :: Ord a => a -> Bintree a -> Bintree a
insertTree a t@(Fork t1 b t2)
  | a == b = t
  | a < b = Fork (insertTree a t1) b t2
  | otherwise = Fork t1 b $ insertTree a t2
insertTree a _ = Fork Empty a Empty

enthalten :: Ord a => a -> Bintree a -> Bool
enthalten a t@(Fork t1 b t2)
  | a == b = True
  | a < b = enthalten a t1
  | otherwise = enthalten a t2
enthalten a _ = False

insertPartial :: Ord a => a -> Bintree a -> Maybe (Bintree a)
insertPartial a t
  | enthalten a t = Nothing
  | otherwise = Just (insertTree a t)
```

Schreiben Sie eine partielle Funktion `insert1_10_4 :: Bintree Int -> Maybe (Bintree Int)`, die die Elemente 1, 10 und 4 (in dieser Reihenfolge) einfügt, falls sie noch nicht im Suchbaum enthalten sind und für den Fall, dass eines der Elemente im Suchbaum enthalten ist, undefiniert ist (d. h. Nothing zurückgegeben wird). Machen Sie sich dafür zu Nutze, dass `Maybe` eine Monade ist.

Aufgabe 10.3 *Identitätsmonade* (7 Punkte)

Der Typ `newtype Id a = Id {run :: a}` sei wie in der Vorlesung als Monade instantiiert. Schreiben Sie eine prozedurale Version der Funktion

```
insertTree :: Ord a => a -> Bintree a -> Bintree a
```

aus Aufgabe 2. Dazu müssen Sie – analog zu `sumTree` auf Folie 123 – eine Hilfsfunktion

```
f :: Ord a => a -> Bintree a -> Id (Bintree a)
```

definieren und deren Ergebnis mit Hilfe von `run` in ein Element von `Bintree a` umwandeln.

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 11

Abgabefrist: 20.01.2012, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 11.1 *Listenmonade* (8 Punkte)

Ein 0-1-String ist ein endlicher String, in dem jedes Zeichen eine 0 oder eine 1 ist. Ein *Muster* ist ein endlicher String, in dem nur die Symbole 0,1 und `*` vorkommen dürfen. Ein Muster σ *überdeckt* einen 0-1-String x, falls x aus σ dadurch erhalten werden kann, dass man in σ jedes Vorkommen von `*` entweder durch 0 oder durch 1 ersetzt. Z.B. überdeckt das Muster `"0**0"` die vier Strings `"0000"`, `"0010"`, `"0100"`, `"0110"`.

(a) Schreiben Sie unter Verwendung der Listenmonade eine Haskell-Funktion `bitStr :: String -> [String]`, die zu einem Muster die Liste der von ihm überdeckten 0-1-Strings berechnet. (4 Punkte)

(b) Schreiben Sie nun eine Haskell-Funktion `pattern :: [String] -> Bool`, die bei Eingabe einer Liste ps von Mustern gleicher Länge n prüft, ob durch die Muster in dieser Liste alle 0-1-Strings der Länge n überdeckt werden, mit anderen Worten, ob es zu jedem 0-1-String der Länge n wenigstens 1 Muster in ps gibt, das diesen String überdeckt.
Testen Sie Ihre Funktion `pattern` für die beiden Listen:

`muster1 = ["**1*1", "01***", "*1**0", "***10","**0**","**100"]` und

`muster2 = ["*1***", "***1*"]` (4 Punkte)

Aufgabe 11.2 *Listenmonade* (10 Punkte)

(a) Schreiben Sie eine Funktion `tripel :: [(Integer, Integer, Integer)]`, die alle pythagoreischen Tripel ausgibt. Verwenden Sie hierzu die Funktionen der Listenmonade, Listenkomprehension ist dabei nicht erlaubt. Zur Erinnerung, ein pythagoreisches Tripel ist ein Tripel natürlicher Zahlen $a < b < c$, so dass $a^2 + b^2 = c^2$. (4 Punkte)

(b) Wir betrachten die Springerfigur auf einem $n \times n$-Schachbrett. Schreiben Sie eine Haskell-Funktion `positions :: Int -> (Int,Int) -> Int -> [(Int,Int)]`, so dass `positions n (i,j) k` die Liste (ohne Wiederholungen) aller Positionen (u, v) angibt, die die Springerfigur in Position (i, j) des $n \times n$-Schachbretts startend in k Zügen erreicht.

a) Lösen Sie das Springerproblem zunächst mit Hilfe von Listenkomprehension. (2 Punkte)

b) Und dann ohne Zuhilfenahme von Listenkomprehension, sondern unter Verwendung der Funktionen der Listenmonade. (4 Punkte)

Aufgabe 11.3 *Monaden-Kombinatoren* (2 Punkte)

Schreiben Sie einen Monaden-Kombinator `seqList :: Monad m => [a -> m a] -> a -> m a`, der zu einer Liste xs von Funktionen und einem Startwert a, die Komposition der Funktionen aus xs (in der gegebenen Reihenfolge) zum Startwert a berechnet.

Prof. Dr. P. Padawitz

Pascal Hof
Jens Lechner
Christian Cöster

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2011/12

Übungsblatt 12

Abgabefrist: 27.01.2012, 10:00 Uhr

Es ist möglich und erwünscht, dass in maximal Dreiergruppen abgegeben wird.
Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,3 und 4	`Jens.Lechner@cs.tu-dortmund.de`
Gruppen 5,6,7 und 8	`Pascal.Hof@tu-dortmund.de`
Gruppen 9 und 10	`Christian.Coester@tu-dortmund.de`

Geben Sie dazu in der Betreff-Zeile an:

`FP_<Gruppennummer>`

Falls Sie z.B. in Gruppe 3 sind, also: `FP_3`.

Wichtig:

Die Abgaben müssen in Form einer vom Interpreter fehlerfrei lesbaren Haskell-Datei vorliegen (Empfehlung: verwenden Sie das "literate Haskell"-Format siehe `www.haskell.org/haskellwiki/Literate_programming`, wobei der Dateiname der Abgabe in diesem Fall wie folgt aussehen sollte:

`fp<Nr. des Blattes><Name eines Authors, mit Großbuchstaben beginnend>.lhs`,

z.B. also `fp3Meier.lhs`, falls es sich um das Aufgabenblatt 3 handelt und Sie Meier heißen. Verwenden Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.

In der Datei geben Sie zunächst die Namen, Vornamen und Matrikelnummern der Personen an, die an dieser Lösung beteiligt waren.
Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.
Schreiben Sie kurze Definitionen und achten Sie darauf, dass die Quellcode-Zeilen nicht mehr als 80 Zeichen haben.

Aufgabe 12.1 *Transitionsmonade* (5 Punkte)

Formulieren Sie eine Funktion `traceM' :: [DefUseE] -> [(String,Int)]`, die eine Variante von `traceM :: [DefUse] -> [(String,Int)]` sein soll, aber auf Listen vom Typ `[DefUseE]` anwendbar ist. Dabei sei `DefUseE` wie folgt definiert:

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

Aufgabe 12.2 *Monadische Parser* (5 Punkte)

Erweitern Sie den Datentyp `Expr` und den Parser `expr` um eine ganzzahlige Division `(:/)`.

Aufgabe 12.3 *IO-Monade* (10 Punkte)

Wir betrachten eine einfache Variante des Galgenmännchen-Spiels. Ein Wort, dessen Länge bekannt ist, soll mit höchstens n Rateversuchen, bei denen der Spieler Buchstaben des Worts rät, gefunden werden. Dabei wird ein Buchstabe, der richtig geraten wurde, in allen im Wort vorkommenden Positionen aufgezeigt.
Dieses Spiel soll nun als Haskell-Programm realisiert werden. Nach Aufruf des Programms, wobei die Anzahl n der Rateversuche als Argument mit übergeben wird, soll das Programm aus einer bestehenden Datei mit einer Liste von Wörtern (möglichst ohne Umlaute) zufällig (siehe System.Random) eines auswählen und verschlüsselt – für jedes Zeichen des Wortes erscheint ein Bindestrich – auf dem Bildschirm ausgeben. In höchstens n Rateversuchen soll der Benutzer Buchstaben des Wortes raten. Korrekt geratene Buchstaben ersetzen im noch verschlüsselten Wort dann den jeweiligen Bindestrich. Hat der Benutzer nach höchstens n Versuchen das Wort vollständig erraten, so wird er zum Gewinner erklärt. Im anderen Fall ist er der Verlierer und das richtige Wort wird aufgezeigt.

Hinweis: Der folgende Codeschnipsel verdeutlicht, wie mit Haskell Zufallszahlen erzeugt werden können:

```
import Random

guess :: IO ()
guess = do x <- getStdRandom $ randomR (0,10)
           putStr "rate Zufallszahl zwischen 0 und 10: "
           n <- readLn :: IO Int
           if n > 10 then return () else do
             putStrLn ("Richtig geraten? "++(show(x == n)))
             guess
```