Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 0 | 16.10.2009

Aufgabenblatt für die Präsenzübung in der zweiten Vorlesungswoche

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)
```

Reduzieren Sie die Ausdrücke:

(a) `twice succ 0` und

(b) `twice twice succ 0`!

Beachten Sie: der Ausdruck `f a b` kürzt `(f a) b` ab, da die Funktionsanwendung von links assoziiert.

Aufgabe 0.2. [Binärbaum]

Das Einfügen von Elementen in einen binären Suchbaum wird durch folgende Funktion `insert` realisiert:

```
insert :: Int -> IntTree -> IntTree
insert a Empty = Node Empty a Empty
insert a (Node l b r) = if a ≤ b then Node (insert a l) b r else Node l b (insert a r)
```

Es sei

```
atree :: IntTree
atree = Node (Node Empty 2 Empty) 5 Empty.
```

Reduzieren Sie den Ausdruck `insert 7 atree`!

Aufgabe 0.3. [Berechnung auf Binärbaum]

Die beiden Funktionen `big` und `prune` sind folgendermaßen definiert:

```
big :: Int -> IntTree
big n = Node (big (2 * n)) n (big (2 * n + 1))

prune :: Int -> IntTree -> IntTree
prune 0 t = Empty
prune (n + 1) Empty = Empty
prune (n + 1) (Node l a r) = Node (prune n l) a (prune n r)
```

Reduzieren Sie den Ausdruck `prune 2 (big 1)`! Worauf müssen Sie achten?

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Präsenzaufgabenblatt 1 | 29.10.2009

Aufgabe 1.1.

(a) Implementieren Sie eine Funktion

```
removeMults :: Eq a => [a] -> [a]
```

die aus einer Liste eventuelle Wiederholungen von Elementen streicht.

(b) Schreiben Sie eine Haskell-Funktion

```
splitEO :: [a] -> ([a],[a])
```

die eine Liste L so in 2 Listen L_1 und L_2 zerlegt, dass L_1 alle Elemente von L enthält, die an geraden Postionen in L stehen, und L_2 diejenigen, die an ungeraden Positionen vorkommen. Die Reihenfolge in L soll dabei erhalten bleiben. Z.B. soll also gelten `splitEO [1,3,1,5,7,3,6] =([1,1,7,6],[3,5,3])`.

Aufgabe 1.2.

(a) Schreiben Sie eine Haskell-Funktion `pSelect :: (a-> Bool) -> [a] -> [a]`, die bei Anwendung auf eine Funktion f vom Typ `a -> Bool` und auf eine Liste xs vom Typ `[a]` die Liste der Elemente x in xs ausgibt, für die $f\ x$ den Wert True hat. Es darf keine Listenkomprehension verwendet werden.

(b) Eine *ideale Zahl* ist eine Zahl, die genauso groß ist wie die Summe ihrer *echten* Teiler (d.h. aller Teiler, die ungleich der Zahl selbst sind). So ist z.B. 6 ideal, da für die echten Teiler $1, 2$ und 3 gilt: $1+2+3 = 6$.

Schreiben sie eine Haskell-Funktion `ideal`, die für eine positive ganze Zahl $n > 1$ die Liste aller idealen Zahlen aus $\{1, \ldots, n\}$ liefert.

Aufgabe 1.3.
Gegeben sei der folgende Datentyp für Binärbäume:

```
data Bintree a = Leaf a | Node (Bintree a) a (Bintree a) deriving (Read, Show)
```

Der Binärbaum `Node (Leaf 'D') 'B' (Leaf 'C')` hätte damit die Wurzel 'B' und das Blatt 'D' als linken und das Blatt 'C' als rechten Unterbaum.
Schreiben Sie eine Haskell-Funktion `countLeafs :: Bintree Int -> Int -> Int`, die bei Eingabe eines Binärbaums und einer ganzen Zahl n in dem Binärbaum die Anzahl der Blätter zählt, die einen Wert größer n haben.

technische universität dortmund

Peter Padawitz
Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

Aufgabe 2.1.
Implementieren Sie die Funktion `f :: Integer -> Integer -> Integer` mit

$$f\ p\ k := \sum_{i=1}^{k} i \cdot p \cdot (1-p)^{i-1}$$

einmal rekursiv und einmal unter Verwendung von Listenkomprehension.

Aufgabe 2.2.
Durch Einbinden des Moduls `Ratio` mit dem Befehl `import Ratio` zu Beginn eines Haskell-Programms stehen in diesem Programm die rationalen Zahlen zur Verfügung. Die Haskell-Notation für eine rationale Zahl $\frac{n}{m}$ ist dabei `n % m`.

Die n-te Harmonische Zahl ist definiert durch $H_n = \sum_{k=1}^{n} \frac{1}{k}$. Implementieren Sie unter Verwendung einer Listenkomprehension die Haskell-Funktion `h :: Integer -> Ratio Integer`, die bei Eingabe n die n-te Harmonische Zahl H_n berechnet.

technische universität dortmund

Peter Padawitz
Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

Aufgabe 3.1.
Die Catalan-Zahlen, oder Catalansche Zahlen, benannt nach dem belgischen Mathematiker Eugène Charles Catalan (1814–1894), stellen eine Folge natürlicher Zahlen dar, die in vielen Problemen der Kombinatorik auftaucht. Die n-te Catalan-Zahl C_n ist z.B. die Anzahl der verschiedenen Möglichkeiten, ein konvexes $(n+2)$-Eck durch Diagonalen in Dreiecke zu zerteilen (Triangulation).Für die Catalan-Zahlen gilt die folgende Rekursionsformel

$$\begin{aligned} C_0 &= 1 \\ C_{n+1} &= \sum_{k=0}^{n} C_k \cdot C_{n-k}. \end{aligned}$$

Geben Sie auf der Basis dieser Rekursionsformel eine rekursive Definition der Catalan-Zahlen in Haskell.

Aufgabe 3.2.
Ausgehend von einer vorgegebenen Menge AV von Aussagenvariablen wird in der Aussagenlogik die Menge AL der aussagenlogischen Formeln in der folgenden Weise als kleinste Menge, die die beiden folgenden Bedingungen erfüllt, definiert:

- Jede Aussagenvariable $A \in AV$ ist eine aussagenlogische Formel.
- Sind F und G aussagenlogische Formeln, so auch $\neg F$, $(F \wedge G)$ und $(F \vee G)$.

(a) Geben Sie einen geeigneten Datentyp `data AL` für die aussagenlogischen Formeln an.

(b) Schreiben Sie eine Funktion `showForm :: AL -> String`, mit der aussagenlogische Formeln in einer Form als String dargestellt werden, die der üblichen Schreibweise nahe kommt. (In der Wahl der Symbole für die Repräsentation von $\neg, \wedge, \vee$ sind Sie frei.)

(c) Schreiben Sie eine Funktion `nnf :: AL -> AL`, die eine aussagenlogische Formeln in ihre Negationsnormalform überführt. Hierbei ist eine aussagenlogische Formeln in Negationsnormalform, wenn ein Negationssymbol höchstens vor einer Aussagenvariablen steht.

technische universität dortmund

Peter Padawitz
Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

Aufgabe 4.1.
Ein sehr einfaches, inzwischen aber auch nicht mehr ganz aktuelles Dateiformat für Graphen ist das GML-Format. Da mit yEd ein frei verfügbarer Graph-Editor existiert, mit dem Graphen in diesem Format eingelesen und dann weiter verarbeitet, insbesondere auch in einem moderneren Format abgespeichert werden können, soll in dieser Aufgabe ein Haskell-Programm geschrieben werden, das einen Graphen in der Adjazenzlistenrepräsentation vom Typ

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

im GML-Format in eine Datei schreibt.
Das folgende einfache Beispiel demonstriert die Repräsentation in diesem GML-Format. Auf eine formale Definition verzichten wir hier. In kursiver Schreibweise sind aber Bemerkungen zum Verständnis eingefügt.

```
graph [
  comment <String>
  directed 1                    1 für gerichtete Graphen, 0 für ungerichtete
  id 42                         Kanten- und Knotenbezeichner als ganze Zahl
  label <String>                label muss ein String sein
  node [                        erst alle Knoten, dann die Kanten
    id 1
    label <String>
  ]
  node [
    id 2
    label <String>
  ]
  node [
    id 3
    label <String>
  ]
  edge [
     source 1
    target 2
    label <String>
  ]
  edge [
    source 2
    target 3
    label <String>
  ]
]
```

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 1 | 19.10.2009

Abgabefrist: 26.10.2009, **Abgabe per Email**

Ihre Lösungen schicken Sie bitte per Email an `Hubert.Wagner@udo.edu`. Die Betreff-Zeile muss dabei für Teilnehmer z.B. der Gruppe 3 folgendermaßen aussehen:

```
FP-Abgabe Gruppe 3
```

Beachten Sie, dass Gruppenabgaben mit einer Gruppengröße von 2 oder 3 Personen gefordert sind. (Ausnahmen nur nach Rücksprache mit dem Betreuer der Übungsgruppe.)

Wichtig:
Ihre Abgaben müssen als Textdatei oder als Haskell-Programm vorliegen.
Am Anfang der Datei sollten die Übungsgruppennummer sowie die Autoren der Lösung aufgeführt werden, im Falle eines Haskell-Programms in einer Kommentarzeile. Der Dateiname der Abgabe sollte dabei wie folgt aussehen:

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

z.B. also `fp1Meier.hs`

Für jede selbstdefinierte Funktion ist der Typ der Funktion anzugeben, ferner ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. In ganz einfachen Fällen darf beides auch schon einmal weggelassen werden.

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) In Haskell können wir Datentypen `D1` und `D2` in der folgenden Form zu einem neuen Datentyp `D` vereinigen:

```
data D = D1 | D2
```

(d) Neue Datentypen werden unter Verwendung von Konstruktoren definiert.

Aufgabe 1.1. [Binärbaum] (6 Punkte)
In dem Datentyp `IntTree` war ein Blatt `a` eines Baumes repräsentiert durch `Node Empty a Empty`. Wir definieren nun einen Datentyp für Bäume mit ganzen Zahlen, in dem ein Produkttyp verwendet wird und Blätter explizit über einen Konstruktor `Leaf` gekennzeichnet werden:

```
data IntTree1 = Leaf Int | Node (IntTree1, Int, IntTree1)
```

`Leaf 1` ist dann ein Baum, der nur aus dem Blatt 1 besteht, `Node ((Leaf 2), 1, (Leaf 3))` ist ein Baum mit Wurzel 1 und einem linken Teilbaum, der aus dem Blatt 2 besteht, und einem rechten Teilbaum, der aus dem Blatt 3 besteht.

(a) Geben Sie eine Haskell-Funktion `tiefe :: IntTree1 -> Int` an, die die Tiefe eines Binärbaums `t` vom Typ `IntTree1` berechnet. (1 Punkt)

Verwenden Sie dazu die Funktion `max`. `max a b` berechnet das Maximum der beiden Zahlen `a` und `b`.

(b) Schreiben Sie die Funktionen `big` und `prune` in Funktionen `big1 :: Int -> IntTree1` und `prune1 :: Int -> IntTree1 -> IntTree1` um, so dass `prune1 n (big1 1)` einen vollständig ausgeglichenen Binärbaum des Datentyps `IntTree1` der Tiefe n konstruiert. (3 Punkte)

(c) Reduzieren Sie `prune1 2 (big1 5)`. (2 Punkte)

Aufgabe 1.2. [Berechnung im Binärbaum] (4 Punkte)
Schreiben Sie eine Haskell-Funktion `replace :: Int -> Int -> IntTree1 -> IntTree1`, so dass `replace n m t` im Binärbaum `t` jedes Vorkommen einer Knotenmarkierung `n` durch die Knotenmarkierung `m` ersetzt.

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 2 | 26.10.2009

Abgabefrist: Abgabe per Email bis 02.11.2009

Ihre Lösungen schicken Sie bitte an `Hubert.Wagner@udo.edu`. Die Betreff-Zeile muss dabei für Teilnehmer z.B. der Gruppe 3 folgendermaßen aussehen:

```
FP-Abgabe Gruppe 3
```

Beachten Sie, dass Gruppenabgaben mit einer Gruppengröße von 2 oder 3 Personen gefordert sind. (Ausnahmen nur nach Rücksprache mit dem Betreuer der Übungsgruppe.)

Wichtig:
Ihre Abgaben müssen als Haskell-Programm vorliegen, d.h. Sie müssen eine Haskell-Datei zuschicken, die vom Haskell-Interpreter (ghci) ohne Fehlermeldung geladen werden kann. In Kommentarzeilen sollen zu Beginn die Übungsgruppennummer sowie die Autoren der Lösung aufgeführt werden. Der Dateiname der Abgabe sollte dabei wie folgt aussehen:

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

z.B. also `fp02Meier.hs`

Für jede selbstdefinierte Funktion ist der Typ der Funktion anzugeben, ferner ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. In ganz einfachen Fällen darf beides auch schon einmal weggelassen werden.

Quizfragen zu Haskell:

(a) Welche der folgenden Datentypdefinitonen sind korrekt?

1) `type ListList = [Int,[Int]]`
2) `type ListPair = [(Int,[Int])]`
3) `data Pair = (Int,Bool)`
4) `type PairTreeBool a = (Tree a, Bool)`
5) `type PairTree a = (Tree a, Tree b)`

(b) Welche der folgenden Aussagen sind wahr, welche falsch?

1) Bäume, die unendlich viele Söhne haben, können nicht als Datentyp definiert werden.
2) In einer Typdefinition mit `type` kann der Datentyp auch rekursiv definiert werden. So ist Z.B. `type PairType a = S a | T (a, PairType a)` ein korrekter Datentyp.
3) In Datentypdefinitionen mit `type` können auch Standardfunktionen vorkommen. Z.B ist die Typdefinition `type ListElem a = [a] !! 2` korrekt.

Aufgabe 2.1. [Listen und Binärbäume] (6 Punkte)

(a) Definieren Sie eine Haskell-Funktion `select :: (Ord a) => a -> [a] ->[a]`, so dass `select x xs` die Liste derjenigen Elemente in xs liefert, die $< x$ sind. (2 Punkte)

(b) Für binäre Suchbäume legen wir den Datentyp

```
data Tree lab = Empty | Node (Tree lab) lab (Tree lab) deriving (Read, Show)
```

zu Grunde.

Geben Sie eine Haskell-Funktion `preOrder :: (Ord a) => [a] -> Tree a` an, die zu einer Folge $x_1, \ldots, x_n$ von paarweise verschiedenen Schlüsseln eines binären Suchbaumes, die aus einer Pre-Order-Traversierung des binären Suchbaumes hervorgegangen ist, den binären Suchbaum rekonstruiert.

(2 Punkte)

(c) Ein beliebtes Kinderspiel zur Auswahl eines Kindes ist das folgende:

- Eine Gruppe von n Kindern bildet einen geschlossenen Kreis.
- Mit einem ersten Kind beginnend wird jeweils bis zum k-ten Kind weitergezählt. Dieses Kind scheidet aus, der Kreis wird wieder geschlossen.
- Vom Nachbarn (Nachfolger) des ausgeschiedenen Kindes beginnend verfährt man wie zuvor, so lange bis nur noch 1 Kind übrigbleibt.

Schreiben Sie eine Haskell-Funktion, die bei Eingabe einer Liste von Kindern (z.B. mit den Nummern $1, \ldots, n$) und einer positiven ganzen Zahl k errechnet, welches Kind am Ende übrigbleibt. (Das Ergebnis darf auch in Form einer Einerliste vorliegen.) (2 Punkte)

Aufgabe 2.2. [Wörterbuch] (4 Punkte)

Implementieren Sie den Datentyp *Wörterbuch* unter Verwendung binärer Suchbäume:

```
type Dict attr val = Tree (attr,val)
empty :: Dict a b
add :: (Ord a) => (a,b) -> Dict a b -> Dict a b
delete :: (Ord a) => a -> Dict a b -> Dict a b
lookupWithDefault :: (Ord a) => Dict a b -> b -> a -> b
```

- `empty` erzeugt ein leeres Wörterbuch.
- `add (x,y) w` fügt in das Wörterbuch w das Paar (x, y) ein. Ist für den Schlüssel x schon ein Eintrag vorhanden, so wird dieser Eintrag überschrieben.
- `delete x w` löscht im Wörterbuch w einen eventuell vorhandenen Eintrag mit Schlüssel x.
- `lookupWithDefault w y x` schaut, ob im Wörterbuch w ein Eintrag zum Schlüssel x vorhanden ist, wenn ja, gibt es diesen Wert aus, ansonsten gibt es y aus.

Übungen zur Vorlesung Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1 Logik in der Informatik

WS 09/10 | Übungsblatt 3 | 02.11.2009

Abgabefrist: Abgabe per Email bis 09.11.2009

Ihre Lösungen schicken Sie bitte an `Hubert.Wagner@udo.edu`. Die Betreff-Zeile muss dabei für Teilnehmer z.B. der Gruppe 3 folgendermaßen aussehen:

`FP-Abgabe Gruppe 3`

Beachten Sie, dass Gruppenabgaben mit einer Gruppengröße von 2 oder 3 Personen gefordert sind. (Ausnahmen nur nach Rücksprache mit dem Betreuer der Übungsgruppe.)

Wichtig:
Ihre Abgaben müssen als Haskell-Programm vorliegen, d.h. Sie müssen eine Haskell-Datei zuschicken, die vom Haskell-Interpreter (ghci) ohne Fehlermeldung geladen werden kann. In Kommentarzeilen sollen zu Beginn die Übungsgruppennummer sowie die Autoren der Lösung aufgeführt werden. Der Dateiname der Abgabe sollte dabei wie folgt aussehen:

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

z.B. also `fp02Meier.hs`

Für jede selbstdefinierte Funktion ist der Typ der Funktion anzugeben, ferner ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. In ganz einfachen Fällen darf beides auch schon einmal weggelassen werden.

Quizfragen zu Haskell:

Welche der folgenden Aussagen sind wahr, welche falsch?

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

(b) `(id . f)` und `id f` liefern bei Anwendung immer dasselbe Ergebnis.

(c) `(4 *) $ (2 +) $ (5 +)` ist gleich der Funktion `\ n -> 28 + 4 * n`.

(d) `map uncurry (:)` liefert einen Typfehler.

Aufgabe 3.1. [Rekursive Funktion auf Listen] **(3 Punkte)**

Implementieren Sie in Haskell den Sortieralgorithmus Insertionsort durch die Funktion

`insertionSort :: Ord a => [a] -> [a]`,

die eine Liste von Elementen aufsteigend sortiert. Sortieren Sie damit 2 Listen bestehend aus 50 bzw. 1000 Zufallszahlen.

Zur Generierung von Zufallszahlen setzen Sie dazu an den Anfang Ihres Programms die Zeile

```
import Random
```

mit der das Modul Random.hs geladen wird. In diesem Modul sind Zufallszahlengeneratoren definiert. Eine unendliche Liste von Zufallszahlen erzeugen Sie dann z.B. mit der folgenden Funktion

```
randomInts :: Int -> Int -> Int -> [Int]
randomInts lowerBound upperBound seed = randomRs (lowerBound,upperBound) (mkStdGen seed)
```

wobei ganzzahlige Zufallszahlen aus dem Bereich von `lowerBound` bis `upperBound` erzeugt werden. `seed` legt unterschiedliche Startwerte für den Zufallszahlengenerator fest, so dass durch Wahl von unterschiedlichen Werten für `seed` unterschiedliche Listen generiert werden.
Mit `take n (randomInts 100 200 50)` wird dann z.B. eine Liste von n ganzzahligen Zufallszahlen aus dem Intervall von 100 bis 200 erzeugt.

Aufgabe 3.2. [Erreichbarkeit in Graphen] **(3 Punkte)**
Als Datentyp für gerichtete Graphen verwenden wir Adjazenzlisten.
Wir führen hierzu als Abkürzung ein:

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

Schreiben Sie eine Haskell-Funktion `reach :: Graph a -> a -> [a]`, die zu einem gerichteten Graphen G und einem Knoten a die Liste der Knoten berechnet, die von a aus über einen Pfad in G erreichbar sind.

Aufgabe 3.3. [First-Fit-Heuristik für das Bin-Packing-Problem] **(4 Punkte)**
In dieser Aufgabe soll die First-Fit Heuristik für das Bin-Packing Problem implementiert werden:
Das Bin-Packing Problem:

Gegeben: n Gegenstände der Größen $w_1, \ldots, w_n$ und beliebig viele Kisten der Größe k. Es gilt $w_i \leq k$ für alle $i = 1, \ldots, n$.

Gesucht: Die kleinste Anzahl von Kisten, mit der alle Gegenstände aufgenommen werden können.

Die First-Fit Heuristik geht so vor, dass die Gegenstände und auch die Kisten in einer festen Reihenfolge betrachtet werden und dann jeder Gegenstand in die erste Kiste gelegt wird, in die er passt.
Testen Sie Ihre Implementierung, indem Sie mit dem Zufallszahlengenerator eine Liste von 1000 Größen zwischen 5 und 100 erzeugen und als Größe der Kisten 150 wählen.

Übungen zur Vorlesung Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1 Logik in der Informatik

WS 09/10 | Übungsblatt 4 | 09.11.2009

Abgabefrist: Abgabe per Email bis 16.11.2009

Ihre Lösungen schicken Sie bitte an `Hubert.Wagner@udo.edu`. Die Betreff-Zeile muss dabei für Teilnehmer z.B. der Gruppe 3 folgendermaßen aussehen:

```
FP-Abgabe Gruppe 3
```

Beachten Sie, dass Gruppenabgaben mit einer Gruppengröße von 2 oder 3 Personen gefordert sind. (Ausnahmen nur nach Rücksprache mit dem Betreuer der Übungsgruppe.)

Wichtig:
Ihre Abgaben müssen als Haskell-Programm vorliegen, d.h. Sie müssen eine Haskell-Datei zuschicken, die vom Haskell-Interpreter (ghci) ohne Fehlermeldung geladen werden kann. In Kommentarzeilen sollen zu Beginn die Übungsgruppennummer sowie die Autoren der Lösung aufgeführt werden. Der Dateiname der Abgabe sollte dabei wie folgt aussehen:

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

z.B. also `fp02Meier.hs`

Für jede selbstdefinierte Funktion ist der Typ der Funktion anzugeben, ferner ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. In ganz einfachen Fällen darf beides auch schon einmal weggelassen werden.

Aufgabe 4.1. [Listenlogik, map] **(5 Punkte)**

(a) Eine Funktion $f : \mathbb{Z} \to \mathbb{Z}$ ist im Intervall $[n, m]$ genau dann streng monoton wachsend, wenn für alle $x, y \in \{n, \ldots, m\}$ gilt:

$$x < y \Rightarrow f(x) < f(y).$$

Schreiben Sie eine Haskell-Funktion

```
monoton :: (Integer -> Integer) -> (Integer,Integer)-> Bool,
```

die feststellt, ob eine totale (d.h. überall definierte) Funktion $f : \mathbb{Z} \to \mathbb{Z}$ im Intervall $[n, m]$ streng monoton wachsend ist. (2 Punkte)

(b) Wir verwenden die schon in Aufgabenblatt 3 eingeführte Adjazenzlisten-Repräsentation für gerichtete Graphen:

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

Implementieren Sie mit Hilfe der map-Funktion eine Haskell-Funktion

```
delEdge :: Eq a => a -> Graph a -> Graph a ,
```

die bei Eingabe eines Knoten x und eines gerichteten Graphen g alle Kanten des Graphen, die zu diesem Knoten hinführen, löscht.

(3 Punkte)

Aufgabe 4.2. [Listenkomprehension, fold] (5 Punkte)

(a) Schreiben Sie eine Haskell-Funktion, die zu einer ganzen Zahl ≥ 0 die Liste ihrer ganzzahligen Teiler > 0 ausgibt. (1 Punkt)

(b) Ein Pythagoräisches Tripel ist ein Tripel (a, b, c) mit positiven ganzen Zahlen a, b und c, so dass $a^2 + b^2 = c^2$.

Geben Sie eine Haskell-Funktion `pTriple :: Integer -> [(Integer, Integer, Integer)]` an, die bei Eingabe n alle pythagoräischen Tripel (a, b, c) mit $c \leq n$ und ungeradem a berechnet. (2 Punkte)

(c) Definieren Sie die Funktion `reverse :: [a] -> [a]` unter Verwendung von foldl. (2 Punkte)

Übungen zur Vorlesung Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1 Logik in der Informatik

WS 09/10 | Übungsblatt 5 | 16.11.2009

Abgabefrist: Abgabe per Email bis 23.11.2009

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 5.1. [Datenstruktur] **(4 Punkte)**

Ausgehend von einer vorgegebenen Menge AV von Aussagenvariablen wird in der Aussagenlogik die Menge AL der aussagenlogischen Formeln in der folgenden Weise als kleinste Menge, die die beiden folgenden Bedingungen erfüllt, definiert:

- Jede Aussagenvariable $A \in AV$ ist eine aussagenlogische Formel.
- Sind F und G aussagenlogische Formeln, so auch $\neg F$, $(F \wedge G)$, $(F \vee G)$ und $(F \rightarrow G)$.

(a) Geben Sie einen geeigneten Datentyp `data AL` für die aussagenlogischen Formeln an. (2 Punkte)

(b) Es sei

```
type Bel = String -> Bool
```

der Datentyp für die Belegungen der aussagenlogischen Variablen. Geben Sie eine Haskell-Funktion `val :: AL -> Bel -> Bool` an, die den Wert einer aussagenlogischen Formel unter einer Belegung ermittelt. (2 Punkte)

Aufgabe 5.2. [Faltung auf Bäumen] **(6 Punkte)**

Auf dem Foliensatz „Modellieren und Implementieren in Haskell" ist die Faltungsfunktion `foldT` zur Berechnung auf Termen definiert worden. Es soll nun die Faltung für Binärbäume betrachtet werden. Für Binärbäume wird dabei der folgende Datentyp zu Grunde gelegt:

```
data BinTree a = Leaf a | T (BinTree a) a (BinTree a) deriving (Read, Show)
```

(a) Implementieren Sie in Analogie zu `foldT` eine Faltungsfunktion `foldBinTree` auf Binärbäumen.

(2 Punkte)

(b) Entwickeln Sie mit Hilfe von `foldBinTree` die Funktionen

1) `blaetter :: BinTree a -> [a]` sammelt die Blätter des Binärbaums

(2 Punkte)

2) `anzahl :: Int -> BinTree Int -> Int`
`anzahl n t` berechnet die Anzahl der Knoten in `t` mit einer Knotenmarkierung $> n$. (2 Punkte)

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 6 | 23.11.2009

Abgabefrist: Abgabe per Email bis 30.11.2009

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 6.1. [Datentyp, Typklasse] **(5 Punkte)**
Für natürliche Zahlen $n, m \in \mathbb{N}$ ist die arithmetische Differenz von n und m gleich $n - m$, falls $m < n$ und 0 sonst.
Gegeben sei nun der Datentyp für die natürlichen Zahlen

```
data Nat = Zero | S Nat deriving (Eq, Ord, Read)
```

(a) Schreiben Sie Haskell-Funktionen `nat2Int :: Nat -> Int` und `int2Nat :: Int -> Nat`, die Umwandlungen von natürlichen Zahlen n in der Repräsentation durch den Datentyp `Nat` in die entsprechenden nicht negativen ganzen Zahlen des Datentyps `Int` und umgekehrt durchführen. Z.B. soll `nat2Int (S ( S Zero))` den Wert 2 und `int2Nat 3` den Wert `S (S (S Zero)))` ergeben.

(1 Punkt)

(b) Deklarieren Sie den Datentyp `Nat` als Instanz der Typklasse `Show` so, dass die `show`-Funktion Elemente des Datentyps `Nat` immer als ganze Zahlen anzeigt, d.h. der Interpreter soll z.B. bei Eingabe `S (S Zero))` 2 und nicht `S (S Zero))` ausgeben. (1 Punkt)

(c) Addition und arithmetische Differenz werden für `Nat` wie folgt definiert:

```
(<+>) :: Nat -> Nat -> Nat
n <+> Zero = n
n <+> (S m) = S (n <+> m)
```

und

```
(<->) :: Nat -> Nat -> Nat
n <-> Zero = n
Zero <-> m = Zero
(S n') <-> (S m') = n' <-> m'
```

Schreiben Sie Haskell-Funktionen für die Multiplikation und die ganzzahlige Division von natürlichen Zahlen des Datentyps `Nat`. (3 Punkte)

Bemerkung: Eine Verwendung der entsprechenden Funktionen von `Int, Integer` usw. zur Definition der oben genannten Funktionen ist nicht erlaubt.

Aufgabe 6.2. [`readFile` **und** `writeFile`] **(5 Punkte)**
Für gerichtete Graphen sei der Datentyp

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

angenommen.

(a) Schreiben Sie eine Haskell-Funktion `tc :: Eq a => Graph a -> Graph a` , die zu einem gerichteten Graphen $G = (V, E)$ die reflexive und transitive Hülle berechnet, d.h. den gerichteten Graphen $G' = (V, E')$, für den $(a, b) \in E'$ genau dann gilt, wenn b von a aus über einen Pfad (im Graphen G) erreichbar ist.

(2 Punkte)

(b) Erstellen Sie mit einem Texteditor die Datei *graph1*, in der ein Graph (z.B. der Graph

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

in Adjazenzlistenrepräsentation gespeichert wird.

Schreiben Sie eine Haskell-Funktion, die zunächst diesen Graphen aus der Datei liest, für diesen dann die reflexive und transitive Hülle berechnet und schließlich das Resultat in der Datei *tcGraph1* speichert.

(3 Punkte)

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 7 | 30.11.2009

Abgabefrist: Abgabe per Email bis 07.12.2009

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 7.1. [Typklasse] **(5 Punkte)**

(a) Definieren Sie die Typklasse `PartOrd` der Halbordnungen. Diese Typklasse soll Funktionen für die Beziehungen *kleiner oder gleich*, *größer oder gleich* und *unvergleichbar* enthalten. (1 Punkt)

(b) In Aufgabe 6.1 war der Datentyp der natürlichen Zahlen definiert worden durch

```
data Nat = Zero | S Nat deriving (Eq, Ord, Read)
```

Auf den natürlichen Zahlen kann nun eine Halbordnung definiert werden durch

$$n \preceq m \text{ genau dann, wenn } n \text{ teilt } m \quad (\forall n, m \in \mathbb{N}).$$

Machen Sie die Implementierung der natürlichen Zahlen durch `Nat` zu einer Instanz von `PartOrd`.

(4 Punkte)

Aufgabe 7.2. [Fixpunktberechnung] **(5 Punkte)**
Für gerichtete Graphen ist wieder der Datentyp

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

angenommen.
Auf Folie 31 des aktuellen Foliensatzes „Modellieren und Implementieren in Haskell" ist die Funktion

```
lfp :: Lattice a => (a -> a) -> a -> a
```

angegeben, so dass `lfp f start` für eine aufwärtsstetige Funktion $f : V \to V$ in dem vollständigen Verband V den kleinsten Fixpunkt von `f` berechnet. Mit `start` sei dabei das kleinste Element in V bezeichnet.
Zeigen Sie, wie die Fixpunktkonstruktion zur Berechnung der reflexiven und transitiven Hülle eines Graphen genutzt werden kann. Definieren Sie dazu

(a) zunächst eine geeignete Verbandsstruktur auf Graphen, d.h., machen Sie `Graph a` zu einer Instanz von `Lattice a` und (2 Punkte)

(b) definieren Sie dann eine Funktion `f`, so dass `lfp f g` zu dem Graphen `g` seine reflexive und transitive Hülle liefert. (3 Punkte)

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 8 | 07.12.2009

Abgabefrist: Abgabe per Email bis 14.12.2009

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 8.1. [Graph-Repräsentationen] **(4 Punkte)**
In den Aufgaben, die sich mit gerichteten Graphen beschäftigten, wurde bisher die Repräsentation über Adjazenzlisten verwendet. Wir hatten dazu den Typ

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

eingeführt. Im Folgenden wollen wir eine Repräsentation über die Adjazenzmatrix betrachten. Wir führen dazu das folgende Typsynonym ein:

```
type GraphAM a = ([a],a -> a -> Bool) .
```

In der Adjazenzmatrix-Repräsentation wird dann ein Graph durch die Liste seiner Knoten und durch eine Boolesche Funktion repräsentiert, die für Knoten a und b genau dann den Wert True hat, wenn eine Kante von a nach b im Graphen vorhanden ist.
Schreiben Sie Haskell-Funktionen `list2am :: Graph a -> GraphAM a`, mit denen Graphen von der Repräsentation über eine Adjazenzliste in die Repräsentationsform einer Adjazenzmatrix überführt werden können, und `am2list :: GraphAM a -> Graph a`, die eine Repräsentation über eine Adjazenzmatrix in eine Adjazenzlistendarstellung umrechnet.

Aufgabe 8.2. [Markierte gerichtete Graphen] **(6 Punkte)**
Wir verallgemeinern die Adjazenzmatrix-Darstellung aus der vorangegangenen Aufgabe auf kantenbewertete *ungerichtete* Graphen. Wir führen dazu den Typ

```
type GraphAML a b = ([a], a -> a-> b)
```

ein, bei denen die Kanten eine Markierung (Kantengewicht) vom Typ `b` erhalten.
Für das Folgende betrachten wir solche ungerichtete Graphen, bei denen die Kanten eine positive ganze Zahl, die die Entfernung zwischen den anliegenden Knoten angibt, als Kantengewicht haben.
Schreiben Sie eine Haskell-Funktion, die zu je zwei Knoten a und c eines solchen Graphen einen kürzesten Weg von a nach c berechnet, sofern ein solcher existiert, und andernfalls eine Fehlermeldung ausgibt.

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 9 | 17.12.2009

Abgabefrist: Abgabe per Email bis 05.01.2010

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 9.1. [ExprC: Arithmetische Ausdrücke über den komplexen Zahlen] (6 Punkte)

Durch den Import des Moduls `Complex` stehen in Haskell die komplexen Zahlen zur Verfügung. Diese werden in der Form `a :+ b` notiert, beispielsweise würde die komplexe Zahl $2 + 3i$ in Haskell-Notation als `2 :+ 3` geschrieben. Weitere Informationen zu dem Modul `Complex` findet man unter

`http://www.haskell.org/onlinereport/complex.html`.

Im Abschnitt *Datentypen* von *Modellieren und Implementieren in Haskell* ist der Datentyp Expr definiert. Wir wollen nun diesen Datentyp zu arithmetischen Ausdrücken über den komplexen Zahlen erweitern, wobei wir zusätzlich noch die Division als arithmetische Operation mit aufnehmen:

```
data ExprC = Con (Complex Float) | Var String | Sum [ExprC] | Prod [ExprC] |
      ExprC :- ExprC | (Complex Float) :* ExprC | ExprC :/ (Complex Float) | ExprC :^ Int
```

(a) Formulieren Sie eine Instanziierung von `ExprC` als Instanz von `Show` in der iterativen Version. Die komplexen Zahlen `a :+ b` sollen dabei als $a + b\ i$ dargestellt werden.

(b) Verallgemeinern Sie die in den Folien angegebene Normalisierungsfunktion `reduce` zu einer Normalisierungsfunktion `reduce :: ExprC -> ExprC` auf `ExprC`.

(c) Erweitern Sie schließlich den Expr-Compiler zu einem ExprC-Compiler. Beachten Sie, dass hier Typ- und Datentyp-Definitionen in geeigneter Form anzupassen sind.

(Jeweils 2 Punkte)

Aufgabe 9.2. [Simulation einer Registermaschine] (6 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`

Es gibt in diesem Fall also keine indirekte Adressierung.

(a) Geben Sie einen geeigneten Datentyp für Registermaschinen-Programme an. (1 Punkt)

(b) Schreiben Sie dann eine Haskell-Funktion `step`, die angewandt auf ein RM-Programm den nächsten Befehl des RM-Programms ausführt und die Registerinhalte nach Ausführung des Befehls ausgibt. (Es sind natürlich nur Registerinhalte von Registern auszugeben, die in der RM-Berechnung benutzt werden.) (3 Punkte)

(c) Schreiben Sie schließlich eine Haskell-Funktion `rm`, die bei Eingabe des RM-Programms dieses ausführt und das Resultat der Berechnung ausgibt. (2 Punkte)

Aufgabe 9.3. [Zusatzaufgabe] (8 Punkte)

In dieser Aufgabe sind ungerichtete Graphen ohne Schlingen vorausgesetzt.
Eine *Euler-Tour* (auch *Euler-Kreis* genannt) in einem Graphen ist ein geschlossener Pfad durch den Graphen, der jede Kante des Graphen genau einmal durchläuft (und jeden Knoten mindestens einmal). Ein (ungerichteter) Graph hat genau dann eine Euler-Tour, wenn der Graph zusammenhängend ist und jeder Knoten des Graphen einen geraden Grad hat, d.h. an jedem Knoten eine gerade Anzahl von Kanten anliegt. Ein Graph mit diesen Eigenschaften wird *Euler-Graph* genannt.

(a) Schreiben Sie eine Haskell-Funktion `euler :: Graph a -> Bool`, die für einen zusammenhängenden Graphen ermittelt, ob es ein Euler-Graph ist. (2 Punkte)

(b) Ein elegantes und auch effizientes Verfahren zur Bestimmung einer Euler-Tour basiert auf der folgenden Beobachtung: Entfernt man die Kanten eines Kreises in einem Euler-Graphen, so sind die verbleibenden Zusammenhangskomponenten wiederum Euler-Graphen. Das Verfahren besteht nun darin, einen Kreis im Graphen zu finden, dessen Kanten zu entfernen, um dann rekursiv für jede Zusammenhangskomponte diese Schritte durchzuführen. Algorithmisch formuliert, in freier Formulierung nach `http://www.algorithmist.com/index.php/Euler_tour`, sieht das Verfahren wie folgt aus:

Es sei `tour` die bisher erstellte Tour, zu Beginn die leere Liste.

```
findTour u:
    für jede Kante e=(u,v) in der aktuellen Kantenmenge E:
        entferne e aus E
        findTour v
    hänge u als Kopfelement an tour an.
```

Implementieren Sie auf der Basis dieses rekursiven Verfahrens eine Haskell-Funktion `eulerTour :: Eq a => Graph a -> [a]`, die zu einem Euler-Graphen eine Euler-Tour berechnet. (6 Punkte)

Ein frohes Weihnachtsfest und einen guten Übergang in das Neue Jahr!

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 10 | 05.01.2010

Abgabefrist: Abgabe per Email bis 12.01.2010

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 10.1. [do-Notation] **(2 Punkte)**

Im Abschnitt *Monaden* von *Modellieren und Implementieren in Haskell* ist das Beispiel der Termunifikation aufgeführt. Unter anderem sind dort die beiden Haskell-Funktionen unify und unifyall unter Verwendung der do-Notation definiert:

```
unify :: Eq a => Term a -> Term a -> Maybe (Substitution a)
unify (V a) (V b)       = Just $ if a == b then V else update V a (V b)
unify (V a) t           = do guard $ a `notIn` t; Just $ update V a t
unify t (V a)           = unify (V a) t
unify (F a ts) (F b us) = do guard $ a == b; unifyall ts us

unifyall :: Eq a => [Term a] -> [Term a] -> Maybe (Substitution a)
unifyall [] us     = do guard $ null us; Just V
unifyall (t:ts) us = do u:us <- return us; f <- unify t u
                        g <- unifyall (map (>>>f) ts) $ map (>>>f) us
                        Just $ (>>> g) . f
```

Schreiben Sie die monadischen Definitionen der beiden Funktionen so um, dass die do-Notation darin nicht mehr vorkommt.

Aufgabe 10.2. [**Listenmonade**] **(6 Punkte)**

(a) Wir repräsentieren Mengen als Listen. Importieren Sie dazu das Haskell-Modul Data.List. Die Mengenoperationen Vereinigung und Durchschnitt stehen uns dann in Form der Haskell-Listenfunktionen `union` und `intersect` zur Verfügung. Diese sind so definiert, dass die resultierenden Listen Elemente nicht mehrfach enthalten.

Schreiben Sie zunächst eine Haskell-Funktion `powerset :: [a] -> [[a]]`, die zu einer Menge die Potenzmenge berechnet.

Schreiben Sie dann eine Haskell-Funktion

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

so dass `f (m,n) (k,p)` die Liste der Teilmengen C von $\{k, k+1, \ldots, p\}$ ist, für die $C \cap \{m, m+1, \ldots, n\} \neq \emptyset$. (3 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. (3 Punkte)

Bemerkung: Listenkomprehension ist bei der Bearbeitung dieser Aufgabe nicht zugelassen! Verwenden Sie stattdessen in beiden Teilaufgaben Funktionen der Listenmonade.

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 11 | 12.01.2010

Abgabefrist: Abgabe per Email bis 19.01.2010

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 11.1. [**Listenmonade**] **(5 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"`.

Schreiben Sie eine Haskell-Funktion `pattern :: [String] -> Bool`, die bei Eingabe einer Liste *ps* von Patterns gleicher Länge n prüft, ob jeder 0-1-String der Länge n durch ein in der Liste *ps* vorkommendes Muster überdeckt wird.

Aufgabe 11.2. [**Maybe-Monade**] **(5 Punkte)**

Editieren Sie mit einem Texteditor ein Telefonverzeichnis vom Typ `[(String,String)]`. Schreiben Sie zunächst eine Haskell-Funktion `lk :: [(String,String)] -> String -> Maybe String`, die bei Eingabe des Telefonverzeichnisses und einer Telefonnummer den zugehörigen Namen, eingebettet in die Maybe-Monade, ausgibt, sofern im Telefonverzeichnis der entsprechende Eintrag vorhanden ist. Schreiben Sie dann eine Funktion `lkList :: [(String,String)] -> [String] -> [String]`, die bei Eingabe des Telefonverzeichnisses und einer Liste von Telefonnummern die Liste der zugehörigen Personen ausgibt, deren Namen mit dem Buchstaben M beginnt. Für Telefonnummern, für die es in der Telefonliste keinen Eintrag gibt, soll die Information „zu der Telefonnummer ... ist kein Eintrag vorhanden" erscheinen.

Übungen zur Vorlesung
Funktionale Programmierung

technische universität dortmund

Peter Padawitz

Hubert Wagner

Lehrstuhl 1
Logik in der Informatik

WS 09/10 | Übungsblatt 12 | 19.01.2010

Abgabefrist: Abgabe per Email bis 26.01.2010

Lösungen bitte an `Hubert.Wagner@udo.edu` mit folgender Betreff-Zeile:

`FP-Abgabe Gruppe <Gruppennummer>`

Wichtig:
Abgaben in Form einer Haskell-Datei, wobei der Dateiname der Abgabe wie folgt aussehen sollte:

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

Für jede selbstdefinierte Funktion ist eine informelle Erläuterung der Argumente sowie der Definition gefordert. Höchstens in ganz einfachen Fällen darf beides weggelassen werden.

Aufgabe 12.1. [**Zustandsmonade**] **(5 Punkte)**

Gegeben sei der Datentyp `data IntTree = Leaf Integer | Node IntTree Integer IntTree` für Binärbäume mit beliebig großen Zahlen als Knoteneinträgen.

Wir definieren die folgende Haskell-Funktion:

```
fibLabel :: IntTree -> IntTree
fiblabel t = fst $ label t 1
label :: IntTree -> Integer -> (IntTree, Integer)
label (Leaf k) n = (Leaf $ fib n, n+1)
label (Node t1 k t2) n = (Node s1 (fib m) s2, r)
   where
      (s1,m) = label t1 n
      (s2,r) = label t2 (m+1)
```

`fib` ist hier die Fibonacci-Funktion, die bekannt sein sollte.
`fibLabel` ersetzt in einem Binärbaum t vom Typ IntTree Knoteneinträge nach folgendem Verfahren: bei einer Inorder-Traversierung von t erhält der n-te Knoten den Knoteneintrag $fib\ n$. Z.B. würde `fibLabel` für den Baum

```
Node (Node (Leaf 2) 5 (Leaf 4)) 8 (Leaf 1)
```

den Baum

```
Node (Node (Leaf 1) 1 (Leaf 2)) 3 (Leaf 5)
```

als Ergebnis liefern.

Schreiben Sie für `fibLabel` eine monadische Version, die die Knoten eines Baumes in der oben genannten Art markiert. Definieren Sie dazu zuerst
`newtype Mark a = Label {trans :: Integer -> (a, Integer)}`
und führen Sie eine geeignete Instanziierung von Mark als Instanz von Monad durch.

Aufgabe 12.2. [IO-Monade] (5 Punkte)
Schreiben Sie ein Haskell-Programm, das mit Hilfe von binären Suchbäumen einfache Telefonlisten realisiert, in denen nur Name und Telefonnumer gespeichert sind. Menugesteuert oder per Abfrage sollen die folgenden Funktionen unterstützt werden:

- eine neue Telefonliste anlegen
- eine vorhandene Telefonliste öffnen
- einen Eintrag (Name, Telefonnumer) in eine geöffnete Telefonliste einfügen
- einen Eintrag in einer geöffneten Telefonliste löschen
- im Falle einer geöffneten Telefonliste zu einem Namen die Telefonnumer finden
- Ausgabe der Telefonliste

Für dieses Programm darf außer dem Standard-Modul Prelude kein weiteres Modul verwendet werden.