Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 0

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:

1. `twice succ 0` und
2. `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 *Listen*

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

1. Geben Sie eine Funktion `indexOf` an, die den kleinsten Index eines Elementes innerhalb einer Liste bestimmt. Wenn das Element nicht in der Liste vorhanden ist, soll -1 zurückgeliefert werden.
   a) Geben Sie den Funktionstyp von `indexOf` an!
   b) Geben Sie eine Implementierung der Funktion an!
2. Definieren Sie eine Funktion `insertAt`, die eine Zahl an eine gegebene Position einer Liste einfügt!
   a) Welchen Funktionstyp hat die Funktion?
   b) Geben Sie die Definition der Funktion an!
3. Geben Sie eine Funktion `insertBehind` an, die eine Zahl y hinter dem ersten Auftreten einer Zahl x in eine Liste einfügt, also z.B.

```
insertBehind 9 6 [ 2, 5, 6, 7, 1 ]    => [ 2, 5, 6, 9, 7, 1 ]
```

Hinweis: Die Listenindizes sollen jeweils natürliche Zahlen, beginnend mit 0 für das erste Element einer Liste, sein.

Prof. Dr. P. Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 1

Abgabefrist: 26.10.2010, 18:00 Uhr

Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,5 und 7	`Christian.Bockermann@cs.tu-dortmund.de`
Gruppen 3 und 4	`Hubert.Wagner@cs.tu-dortmund.de`
Gruppen 6 und 8	`Tristan.Skudlik@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. Verwendung Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.
Für das 1. Aufgabenblatt stehen im EWS-Arbeitsraum im öffentlichen Teil 2 Dateien mit entsprechenden "Masken" zur Verfügung: `fp1Name.hs` und `fp1Name.lhs`. Diese können Sie zur Erstellung Ihres Programms bzw. zur Abgabe verwenden. Ersetzen Sie aber `Name` durch Ihren eigenen Namen.

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 *Update von Funktionen* (10 Punkte)

(a) Definieren Sie eine Haskell-Funktion `k :: Int -> Int`, die alle Zahlen ungleich 0 auf 1 und die 0 auf 0 abbildet. (1 Punkt)

(b) Ändern Sie mit Hilfe der update-Funktion die Funktion `k` zu einer Funktion `k1 :: Int -> Int` so ab, dass sie für das Argument 2 ebenfalls den Wert 0 annimmt. (2 Punkte)

(Beachten Sie: `update` ist keine Standard-Haskell Funktion, so dass die Definition von update in Ihrem Haskell-Programm enthalten sein muss. In den beiden Dateien `fp1Name.hs` und `fp1Name.lhs`, die im EWS-Arbeitsraum zur Verfügung stehen, ist diese Definition schon enthalten.)

(c) Nun möchten Sie mit Hilfe der update-Funktion die Funktion `k` zu einer Funktion `k2 :: Int -> Int` so abändern, dass sie für die Argumente 2 und 5 den Wert 0 annimmt. Geben Sie die Definition von `k2` an. (4 Punkte)

(d) 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)

Aufgabe 1.2 *Listen* (10 Punkte)

(a) Schreiben Sie eine Haskell-Funktion `dupReverse :: [a] -> [a]`, die eine Liste umkehrt und dabei die Listenelemente jeweils zwei mal hintereinander schreibt. Z.B. würde `dupReverse [1,2,3,4]` die Liste `[4,4,3,3,2,2,1,1]` als Ergebnis haben. (Es darf nicht die Haskell-Funktion `reverse` verwendet werden. (4 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. (6 Punkte)

Prof. Dr. P. Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 2

Abgabefrist: 2.11.2010, 18:00 Uhr

Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,5 und 7	`Christian.Bockermann@cs.tu-dortmund.de`
Gruppen 3 und 4	`Hubert.Wagner@cs.tu-dortmund.de`
Gruppen 6 und 8	`Tristan.Skudlik@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. Verwendung Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben.
Für das 1. Aufgabenblatt stehen im EWS-Arbeitsraum im öffentlichen Teil 2 Dateien mit entsprechenden "Masken" zur Verfügung: `fp1Name.hs` und `fp1Name.lhs`. Diese können Sie zur Erstellung Ihres Programms bzw. zur Abgabe verwenden. Ersetzen Sie aber `Name` durch Ihren eigenen Namen.

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 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`.

Aufgabe 2.1 *Listenfunktionen* (15 Punkte)

(a) Geben Sie eine Haskell-Funktion `insertS :: Ord a => a -> [a] -> [a]` an, die ein Element in eine aufsteigend sortierte Liste (sortiert bezüglich der Relation $<$ auf `a`) so einfügt, dass die resultierende Liste immer noch aufsteigend sortiert ist. (3 Punkte)

Die "Voraussetzung" `Ord a =>` bedeutet dabei, dass die Funktion `insertS` nur für Datentypen a definiert wird, für die die Ordnungsrelation $<$ im Datentyp vorhanden ist. Dies ist z.B für `Bool`, für die numerischen Datentypen, für `Char` wie auch für `String` der Fall.

(b) Implementieren Sie in Haskell den Sortieralgorithmus Insertionsort durch die Funktion

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

die eine Liste von Elementen aufsteigend sortiert. (6 Punkte)

(c) Das Produkt von 111 mit sich selbst liefert als Ergebnis die Zahl 12321, die als String betrachtet, ein Palindrom ist. Bestimmen Sie mit Hilfe einer Haskell-Funktion die größte Zahl, die ein Produkt von zwei 3-stelligen ganzen Zahlen ist und ein Palindrom darstellt. (6 Punkte)

Hinweis: Die Haskell-Funktion `show` angewandt auf eine ganze Zahl liefert die Stringdarstellung dieser Zahl.

Aufgabe 2.2 *Ungerichtete Graphen* (5 Punkte)

In dieser Aufgabe verwenden wir Adjazenzlisten zur Repräsentation von ungerichteten Graphen. In dieser Repräsentation wird der Graph dargestellt als eine Liste von Tupeln der Form (Knoten, Liste der direkten Nachbarn des Knoten).
Wir führen hierzu als Abkürzung ein:

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

Schreiben Sie eine Haskell-Funktion `ug :: Eq a => Graph a -> Bool`, die für ein Argument vom Typ `Graph a` feststellt, ob es ein ungerichteter Graph ist.
Die Voraussetzung `Eq a =>` impliziert, dass auf dem Datentyp `a` der Test auf Gleichheit zur Verfügung steht. Für die Basistypen ist diese Voraussetzung erfüllt.

Prof. Dr. P. Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 3

Abgabefrist: 9.11.2010, 18:00 Uhr

Die Abgabe erfolgt per Email an den Betreuer der Übungsgruppe:

Gruppen 1,2,5 und 7	`Christian.Bockermann@cs.tu-dortmund.de`
Gruppen 3 und 4	`Hubert.Wagner@cs.tu-dortmund.de`
Gruppen 6 und 8	`Tristan.Skudlik@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. Verwendung Sie nicht das "literate Haskell"-Format, so muss die Datei die Endung `.hs` haben. Für das 1. Aufgabenblatt stehen im EWS-Arbeitsraum im öffentlichen Teil 2 Dateien mit entsprechenden "Masken" zur Verfügung: `fp1Name.hs` und `fp1Name.lhs`. Diese können Sie zur Erstellung Ihres Programms bzw. zur Abgabe verwenden. Ersetzen Sie aber `Name` durch Ihren eigenen Namen.

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 wahr, welche falsch?

(a) `[ i | i <- [1..], mod 0 i == 0]` ist nicht definiert.

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

Aufgabe 3.1 *fold-Funktionen* (10 Punkte)

(a) Schreiben Sie mit Hilfe von `foldl` eine Haskell-Funktion `filterFirst :: (a -> Bool) -> [a] -> [a]`, so dass `filterFirst p xs` in der Liste `xs` das erste Element, auf das die Eigenschaft `p` nicht zutrifft, löscht. (4 Punkte)

(b) Geben Sie unter Verwendung von `foldl` oder `foldr` die Definition einer Haskell-Funktion `digitProd :: String -> Int` an, so dass für eine als String repräsentierte ganze Zahl n mit mindestens 5 Ziffern `digitProd n` das größte Produkt von 5 aufeinanderfolgenden Ziffern in n liefert. Z.B wäre `digitProd ''2314740''` gleich 336. Im "Projekt Euler" (Problem 21 (`http://projecteuler.net/index.php?section=problems`) ist diese Aufgabenstellung für eine 1000-stellige Zahl zu lösen.

Zur Lösung dieser Aufgabe fügen Sie an den Anfang Ihres Haskell-Programms die Zeile

```
import Char
```

ein und definieren Sie eine Funktion `c2i :: Char -> Int` durch `c2i x = ord x - 48`, die eine Ziffer, als Char betrachtet, in eine ganze Zahl umwandelt. (6 Punkte)

Aufgabe 3.2 *Listenkomprehension* (10 Punkte)

(a) Schreiben Sie unter Verwendung von Listenkomprehension eine Haskell-Funktion, die zu einer positiven ganzen Zahl n die Liste aller echten Teiler berechnet, d.h., die Liste aller ganzen Zahlen i mit $1 \leq i < n$, so dass n ein ganzzahliges Vielfaches von i ist. (3 Punkte)

(b) Zwei verschiedene natürliche Zahlen, von denen wechselseitig jeweils eine Zahl gleich der Summe der echten Teiler der anderen Zahl ist, bilden ein Paar befreundeter Zahlen. Z.B bilden 220 und 284 ein befreundetes Zahlenpaar, da die Summe der echten Teiler von 220 $1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284$ ist und die Summe der echten Teiler von 284 gleich $1 + 2 + 4 + 71 + 142 = 220$ ist. Eine Zahl n, die zusammen mit einer anderen Zahl ein befreundetes Zahlenpaar bildet, bezeichnet man selbst auch als befreundet.

Geben Sie unter Verwendung von Listenkomprehension eine Haskell-Funktion `amicable :: Integer -> Integer`, die zu einer positiven ganzen Zahl n die Summe aller befreundeten Zahlen $\leq n$ berechnet. Testen Sie Ihre Funktion für $n = 10000$.

(7 Punkte)

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 4

Abgabefrist: 16.11.2010, 18:00 Uhr

Anmerkungen

Zur graphischen Wiedergabe von Objekten eines Haskell-Datentyps sowie von Listen von Punktlisten soll der Painter-Modul verwendet werden:

`http://fldit-www.cs.uni-dortmund.de/~peter/Haskellprogs/Painter.pdf`

Dazu muss zunächst der Modul selbst von der Webseite

`http://fldit-www.cs.uni-dortmund.de/~peter/Haskellprogs/Painter.hs`

in das Verzeichnis kopiert werden, indem euer Haskell-Programm steht. Am Anfang desselben ist der Befehl

```
import Painter(drawTerm,drawTrackC)
```

einzufügen. Damit stehen zwei Ausgabeprozeduren zur Verfügung:

`drawTerm :: String -> IO()` und `drawTrackC :: String -> Bool -> Int -> IO()`.

Der Aufruf von `drawTerm file` erwartet ein Datentypobjekt in der Datei `file` und zeichnet dieses als Baum in die Datei `file.svg`. Der Befehl

```
drawTrackC file smooth mode
```

erwartet eine Liste LL des Typs `[[(Float,Float)]]` in der Datei `file` und zeichnet jede Punktliste `L` von `LL` als Linienzug zwischen den Punkten von `L` in die Datei `file.svg`. Das Argument `mode` ist eine Zahl, die angibt, wie die Linien gefärbt werden sollen. Für $mode = 0$ folgt die Färbung eines Linienzuges einem Kreis äquidistanter Farben.
Bei den Modi 2,3,6,7,10 und 11 werden nicht die Linien, sondern die von diesen und dem Mittelpunkt der Graphik aufgespannten Dreiecke gefärbt. Bei $smooth = True$ werden die Ecken der Dreiecke geglättet.
`file.svg` ist eine XML-Datei. Ihr Inhalt wird von jedem Browser als die entsprechende Graphik interpretiert.
Wer noch andere in Painter.pdf beschriebene Funktionen von Painter.hs verwenden möchte, muss diese als weitere Argumente von Painter im obigen `import`-Befehl auflisten.

Aufgabe 4.1 *Datentyp und foldl-Funktion* (10 Punkte)

Gegeben sei ein einfaches Turtle-System (im $\mathbb{R}^2$). Die Turtle befindet sich zunächst im Zentrum $(0.0, 0.0)$ und schaut in Richtung $(1.0, 0.0)$, also "nach rechts".
Anhand einer Liste von Befehlen (vgl. Datentyp `Action` in der Vorlesung) wird die Turtle nun bewegt. Im Folgenden sei `ActionS` definiert als:

```
data ActionS = Move Float | TurnLeft | TurnRight
```

Dabei führt `Move x` zu einer Bewegung um x Einheiten in die aktuelle Blickrichtung, `TurnLeft` dreht die Turtle um 90 Grad entgegen dem Uhrzeigersinn und `TurnRight` dreht die Turtle um 90 Grad im Uhrzeigersinn.

a) Schreiben Sie – unter Verwendung von `foldl` – eine Funktion

```
turtleTrack::[ActionS] -> [(Float,Float)]
```

die für eine Liste von Befehlen die zugehörigen Positionen der Turtle nach jedem Befehl zurückgibt. Dazu soll für jeden Befel die Position der Turtle nach diesem Befehl (in Abhängigkeit von der letzten Turtle-Position) in die Ergebnisliste eingefügt werden.

Beispiel: Die Befehlsfolge

```
[ Move 30.0, TurnLeft, Move 30.0, TurnLeft, Move 30.0, TurnLeft, Move 30.0 ]
```

beschreibt ein Quadrat der Länge 30 und entspricht den Positionen

$$[(0.0, 0.0), (30.0, 0.0), (30.0, 30.0), (0.0, 30.0), (0.0, 0.0)].$$

(5 Punkte)

b) Im Folgenden sollen für den Befehl *Turn* beliebige Winkel erlaubt sein. Wir definieren dazu den Datentyp `Action` durch

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

Geben Sie eine entsprechende Haskell-Funktion an, die Ihre in Teil a) definierte Funktion entsprechend verallgemeinert. (5 Punkte)

Hinweis: *Überlegen Sie, welche Informationen zu jedem Befehl benötigt werden und welche Informationen sich durch einen Befehl ändern können. Entwickeln Sie Ihre Funktionen schrittweise!*

Zum Testen ist es hilfreich, sich eine konstante Funktion zu schreiben, die eine Liste von Befehlen erzeugt, z.B.

```
eingabe = [ Turn 45.0, Move 30.0, Turn 90.0, Move 30.0,
            Turn 90.0, Move 30.0, Turn 90.0, Move 30.0 ]
```

Aufgabe 4.2 *Selbst definierte Datentypen* (10 Punkte)

Binäre Bäume können in Haskell z.B. durch den folgenden Datentyp repräsentiert werden:

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

a) Schreiben Sie eine Haskell-Funktion `countLeaves`, die die Anzahl der Blätter in ein em Binärbaum zählt! (5 Punkte)

b) Geben Sie eine Haskell-Funktion `height` an, die die Höhe eines Binärbaumes berechnet!

(5 Punkte)

(*Die Höhe eines Baumes T ist definiert als die Länge eines längsten Pfades von der Wurzel zu einem Blatt in T.*)

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 5

Abgabefrist: 23.11.2010, 18:00 Uhr

Aufgabe 5.1 *Funktionen auf Binärbäumen* (10 Punkte)

In dieser Aufgabe sollen einige einfache Funktionen für binäre Bäume in Haskell implementiert werden.

a) Definieren Sie einen Datentyp `BinaryTree` der einen binären Baum darstellt und dessen Knoten Elemente vom Typ `Int` enthalten. (1 Punkt)

b) Geben Sie eine Funktion `insert :: BinaryTree -> Int -> BinaryTree` an, die eine Zahl in den Baum einsortiert. Dabei soll der Baum stets die Eigenschaft eines binären Suchbaums haben, d.h. für jeden Knoten gilt, dass alle Elemente im linken Teilbaum echt kleiner und alle Elemente im rechten Teilbaum echt größer als das Wurzelelment sind.

Ist ein Element bereits im Baum enthalten, soll der Baum unverändert zurückgegeben werden. (3 Punkte)

c) Geben Sie ein Funktion `delete :: BinaryTree -> Int -> BinaryTree` an, die eine Zahl aus dem binären Baum entfernt, falls diese im Baum enthalten ist. (6 Punkte)

Aufgabe 5.2 *Faltung auf Bäumen* (10 Punkte)

In dieser Aufgabe legen wir für Binärbäume den folgenden Datentyp zu Grunde:

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

In Analogie zur Faltungsfunktion `foldT` zur Berechnung auf Termen definieren wir eine Faltungsfunktion `foldBinTree` auf Binärbäumen durch

```
foldBinTree :: (a -> b -> b -> b) -> (a -> b) -> (BinTree a) -> b
foldBinTree g f (Leaf x) = f x
foldBinTree g f (T l x r) = g x (foldBinTree g f l) (foldBinTree g f r)
```

Entwickeln Sie mit Hilfe von `foldBinTree` die Funktionen

a) `blaetter :: BinTree a -> (a -> Bool) -> [a]`, so dass `blaetter t p` die Liste der Blätter `x` in `t` berechnet, für die die Bedingung `p x` erfüllt ist. (4 Punkte)

b) `heights :: BinTree a -> BinTree Int`

`heights t` ersetzt in dem Binärbaum `t` jede Knotenmarkierung durch die Höhe des zum Knoten gehörigen Unterbaumes. Z.B. würde

```
heights $ T (Leaf 'D') 'B' (T (Leaf 'C') 'B' (Leaf 'A'))
```

als Ergebnis den Binärbaum `T (Leaf 0) 2 (T (Leaf 0) 1 (Leaf 0))` haben. (6 Punkte)

Aufgabe 5.3 *Präsenzaufgabe*

Implementieren Sie eine Haskell-Funktion `nt :: BinTree a -> BinTree Int`, die in einem Binärbaum die Markierungen der Knoten durch die Nummern ersetzt, die die Knoten bei einem Inorder-Durchlauf erhalten. Z.B. würde sich für den oben genannten Binärbaum der Binärbaum `T (Leaf 1) 2 (T (Leaf 3) 4 (Leaf 5))` ergeben

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 6

Abgabefrist: 30.11.2010, 18:00 Uhr

Aufgabe 6.1 *Datentyp, Typklasse* (8 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.

(5 Punkte)

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. (3 Punkte)

c) Zusatzaufgabe (5 Extrapunkte)

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`.

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`* (12 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. (6 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 dann eine Haskell-Funktion

```
testF :: (Read a, Show b) => (a -> b) -> String -> String -> IO (),
```

so dass durch den Aufruf

```
testF (tc :: Graph Int -> Graph Int) "graph1" "tcGraph1"
```

zunächst dieser Graph aus der Datei eingelesen wird, dann für ihn die reflexive und transitive Hülle berechnet und schließlich das Resultat in der Datei *tcGraph1* gespeichert wird.

(3 Punkte)

c) Schreiben Sie eine Haskell-Funktion `ud :: Eq a => Graph a -> Graph a`, die aus einem gerichteten Graphen einen ungerichteten macht. (In der Adjazenzlistenrepräsentation ist ein ungerichteter Graph dadurch gekennzeichnet, dass im Graphen zu jeder vorhandenen Kante (a, b) auch die Kante (b, a) existiert.) (3 Punkte)

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 7

Abgabefrist: 7.12.2010, 18:00 Uhr

Aufgabe 7.1 *Datentyp, Typklasse* (10 Punkte)

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

$$a \wedge b = \text{ Infimum von } a \text{ und } b$$

und

$$a \vee b = \text{ Supremum von } a \text{ und } b$$

bezüglich $\preceq$, so bildet A zusammen mit diesen Operationen ein Verband. 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 BoundedLattice a where
  top :: a
  bot :: a
  (<:) :: a -> a -> Bool
  join :: a -> a -> a
  meet :: a -> a -> a
```

In Aufgabe 6.1 (Übungsblatt 6) 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 $\preceq$ definiert werden durch:
$\forall n, m \in \mathbb{N}$:

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

Machen Sie die Implementierung der natürlichen Zahlen durch `Nat` zu einer Instanz von `BoundedLattice`. Beachten Sie, dass 0 bezüglich $\preceq$ das größte Element von $\mathbb{N}$ ist.

Aufgabe 7.2 *Read* (10 Punkte)

Wir definieren als Datentyp für die rationalen Zahlen:

```
data RationalZahl = (:%:) Integer Integer deriving Show.
```

Der Konstruktor `:%:` kann dabei als Infix-Konstruktor verwendet werden. Für diesen Datentyp soll nun die `read`-Funktion in der Weise zur Verfügung stehen, dass sowohl ein String der Gestalt

```
"(n/m)"
```

als auch ein String der Gestalt

```
"R(n,m)"
```

mit n und m ganze Zahlen als Rationalzahl `n :%: m` eingelesen wird.
(Hinweis: Definieren Sie `readsPrec` in geeigneter Weise.)

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 8

Abgabefrist: 14.12.2010, 18:00 Uhr

Aufgabe 8.1 *Funktionskomposition* (10 Punkte)

a) Schreiben Sie unter Verwendung von Funktionskomposition eine Funktion

```
charList :: Char -> [String] -> [String],
```

so dass `charList c` angewandt auf eine Liste von Strings als Ergebnis die Liste der Strings liefert, die mit dem Zeichen `c` beginnen. (5 Punkte)

b) Die Funktion `filter . flip elem :: Eq a => [a] -> [a] -> [a]` berechnet, angewandt auf zwei Listen, den "Durchschnitt" der beiden Listen. Beschreiben Sie, wie der Berechnungsvorgang dieser Funktion abläuft, wenn sie auf die beiden Listen [2,3] und [2] angewandt wird. (5 Punkte)

Aufgabe 8.2 *Datentyp Set* (10 Punkte)

In den Vorlesungsfolien ist der Datentyp `Set a` eingeführt worden durch die Typdefinition

```
newtype Set a = Set {list :: [a]}.
```

a) Deklarieren Sie `Set a` als Instanz von `Show` so, dass eine Menge in der üblichen Mengenschreibweise und nicht als Liste angezeigt wird.

(5 Punkte)

b) Schreiben Sie eine Funktion `powerSet :: Eq a => Set a -> Set (Set a)`, die zu einer Menge die Potenzmenge berechnet.

(5 Punkte)

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 9

Abgabefrist: 4.1.2011, 18:00 Uhr

Aufgabe 9.1 *Freiwillige Zusatzaufgabe: Graph-Repräsentationen* (10 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.

a) 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.

(4 Punkte)

b) Da ein Objekt vom Typ `GraphAM a` nicht auf dem Bildschirm ausgegeben werden kann, ist es hilfreich, eine geeignete show-Funktion zur Verfügung zu haben. Zu diesem Zweck definieren wir

```
newtype GraphAMF a = G ([a],a -> a -> Bool) .
```

i) Definieren Sie eine show-Funktion für `GraphAMF a`, so dass der Graph in Form einer Adjazenz-Matrix ausgegeben wird, falls der Graph nicht mehr als 10 Knoten enthält, und zeilenweise mit Angabe des Knotenpaars und Booleschem Wert sonst. (Statt `True` und `False` darf auch 1 bzw. 0 ausgegeben werden.)

ii) Geben Sie eine Funktion `am2amf :: Show a => GraphAM a -> GraphAMF a`, die einen Graphen vom Typ `GraphAM a` in den Typ `GraphAMF a` transformiert, so dass eine Ausgabe auf dem Bildschirm möglich ist.

(6 Punkte)

Aufgabe 9.2 *Kürzeste Wege in einem Graphen* (10 Punkte)

In dieser Aufgabe verwenden wir für kantenbewertete ungerichtete Graphen den Datentyp

```
type Matrix a b = a -> a -> b
```

(siehe Vorlesungsfolien), wobei `b` der Typ der Kantengewichte ist.

a) Für ungerichtete Graphen, bei denen die Kanten eine positive ganze Zahl, die die Entfernung zwischen den anliegenden Knoten angibt, als Kantengewicht haben, wollen wir nun die Länge des kürzesten Weges zwischen 2 Knoten berechnen.

Schreiben Sie unter Verwendung der Haskell-Funktion

```
closure :: Semiring b => Matrix a b -> [a] -> Matrix a b,
```

die in den Vorlesungsfolien angegeben ist, eine Haskell-Funktion

```
csp :: Semiring b => Matrix a b -> [a] -> a -> a -> b,
```

die zu je zwei Knoten c und d eines solchen Graphen die Länge eines kürzesten Weges von c nach d berechnet, sofern ein solcher existiert, und andernfalls eine Fehlermeldung ausgibt.

(Hinweis: Auf der EWS-Seite der Veranstaltung finden Sie die Datei `supportBlatt9.hs`, in der eine Erweiterung des Datentyps `Nat` um ein Element `Infinity` für "unendlich" enthalten ist. Geben Sie eine geeignete Instanzdeklaration von NatInf an, so dass mit `closure` die Längen kürzester Wege berechnet werden können.) (4 Punkte)

b) Sie möchten nun nicht nur die Länge eines kürzesten Weges wissen, sondern auch einen solchen kürzesten Weg ausgegeben bekommen. Schreiben Sie eine Funktion

```
sp :: Semiring b => Matrix a b -> [a] -> a -> a -> (b,[a]),
```

die dies tut. (6 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`

Es gibt in diesem Fall also keine indirekte Adressierung. Den Datentyp für Registermaschinen-Programme legen wir folgendermaßen fest:

```
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 nun 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.) (5 Punkte)

b) Schreiben Sie schließlich eine Haskell-Funktion `rm`, die bei Eingabe des RM-Programms dieses ausführt und das Resultat der Berechnung ausgibt. (5 Punkte)

(In der Datei `supportBlatt9.hs` wird ein RM-Programm zu Berechnung der Potenz n^m für $m \geq 0$ zur Verfügung gestellt.)

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 10

Abgabefrist: 11.1.2011, 18:00 Uhr

Aufgabe 10.1 *Listenmonade* (10 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. (5 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*"]` (5 Punkte)

Aufgabe 10.2 *Listenmonade und Maybe-Monade* (10 Punkte)

a) 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. (6 Punkte)

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

b) Schreiben Sie eine Haskell-Funktion `f :: Integer -> Integer -> Maybe Integer`, die angewandt auf 2 ganze Zahlen n und m das Ergebnis `Nothing` hat, falls $n \ mod \ m = 0$ ist, und sonst das Ergebnis $n^m \ div \ (n \ mod \ m)$, aber gepackt in der Maybe-Monade.

Schreiben Sie dann eine Funktion `testDef :: Integer -> Integer -> Maybe String`, die angewandt auf 2 ganze Zahlen n und m den Funktionswert $f \ n \ m$ berechnet und im Falle, dass das Ergebnis definiert ist, den Funktionswert als String ausgibt. Ist das Ergebnis nicht definiert, dann soll der String

`"Funktionswert nicht definiert, da "++(show n)++" mod "++(show m)++" = 0"`

ausgegeben werden. (4 Punkte)

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 11

Abgabefrist: 18.1.2011, 18:00 Uhr

Aufgabe 11.1 *Zustandsmonade* (10 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 11.2 *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 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.

(Besonders kreative Personen dürfen auf die Eingabe n der Rateversuche verzichten und stattdessen das Galgenmännchen z.B. mit dem ASCII-Zeichensatz erstellen.)

Bemerkung:
Die Lösungshinweise zu Aufgabenblatt 10 werden zusätzlich zu den Lösungen einige Beispiele zu Listen-, Zustands- und IO-Monaden enthalten. Diese Lösungshinweise werden voraussichtlich ab 16.1. online sein.

Peter Padawitz

Christian Bockermann
Hubert Wagner

Übungen zur Vorlesung

Funktionale Programmierung

Wintersemester 2010/11

Übungsblatt 12

Abgabefrist: 25.1.2011, 18:00 Uhr

Aufgabe 12.1 *Parser* (8 Punkte)

Wir betrachten sehr einfache arithmetische Ausdrücke der Programmiersprache Lisp, die von der folgenden Gestalt sind: `( <Operator> Zahl 1 ... Zahl n)` , wobei der Operator entweder `+` oder `*` ist. Z.B. berechnet `(+ 3.14 7.2 -3.1)` die Summe der 3 aufgeführten Zahlen, `(* 3.14 7.2 -3.1)` würde das Produkt dieser 3 Zahlen berechnen. Insbesondere ist `(+) = 0`, `(+ n) = n`, `(*) = 1` und `(* n) = n)` für eine beliebige Dezimalzahl n.

a) Geben Sie zunächst einen geeigneten Haskell-Datentyp `LispExpr` zur Repräsentation dieser arithmetischen Ausdrücke an. (2 Punkte)

b) Schreiben Sie einen Parser `parseLE :: MonadPlus m => Parser m LispExpr`, der arithmetische Ausdrücke der oben genannten Art in den Datentyp LispExpr parst. (6 Punkte)

Aufgabe 12.2 *Parser* (12 Punkte)

Wir definieren den Datentyp der arithmetischen Ausdrücke in einer Variablen X durch

```
data XExpr = Con Integer | X | Sum [XExpr] | Prod [XExpr] |
   XExpr :- XExpr | Integer :* XExpr | XExpr :^ Integer deriving Show
```

a) Für arithmetische Ausdrücke in X kann analog zur Auswertungsfunktion der `evalE` der Vorlesung eine Auswertungsfunktion `evalE :: XExpr -> Integer -> Integer` definiert werden. Geben Sie eine solche Definition an. (2 Punkte)

b) Wir betrachten nun Funktionsdefinitionen von einstelligen Funktion in einer Variablen x, die über arithmetische Terme in x definiert werden. In solchen Termen dürfen nur die arithmetischen Operatoren $+$, $-$, $*$ und `^` und an Variablen höchstens x vorkommen. Ein Beispiel für eine solche Funktionsdefinition ist

```
f x = 23 * (x^5 - (261 + x^2))^4 + (x + 17)^2 + 85
```

Schreiben Sie nun einen Parser

```
parseFunction :: MonadPlus m => Parser m (Integer -> Integer),
```

der einen String mit einer Funktionsdefinition der oben genannten Art in diejenige anonyme einstellige Funktion parst, die durch die Funktionsdefinition beschrieben ist. Z.B. soll `fst $ head $ runM parseFunction "f x = x^5"` die Haskell-Funktion `\ x -> x^5` liefern.

Verwenden Sie zur Definition des Parsers `parseFunction` den Parser `parseE`, der analog zu parseE in den Folien zur Vorlesung definiert ist und den Sie in der Datei Blatt12Vorlage.hs (siehe EWS-Seite zur Vorlesung) finden. In dieser Datei sind auch weitere Parser aufgeführt, die u.U. für die Lösung relevant sind. (10 Punkte)