technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Präsenzblatt

Ausgabe: 13.10.2017, **Abgabe:** keine Abgabe, **Block:** N.N.

Aufgabe 0.1 *Einführung in den* `GHCi`

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

a) Öffnen Sie den Texteditor Ihrer Wahl und tippen Sie folgendes Programm ab:

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

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

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

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

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

Folgende Kommandos des GHCi haben sich als nützlich erwiesen:

- `:load file` (kurz `:l`) lädt die Datei `file` in den GHCi.
- `:reload` (kurz `:r`) lädt die aktuelle Datei neu ein. Nachdem Änderungen an dem Quelltext vorgenommen wurden, kann die aktuelle Datei mit `:r` leicht neu geladen werden.
- `:type ausdruck` (kurz `:t`) zeigt den Typ des Ausdruckes `ausdruck` an, z. B. `:t f` oder `:t f 1 2 3`.
- `:kind typ` (kurz `:k`) zeigt den Kind des Typs `typ` an, z. B. `:k Int` oder `:k []`.
- `:info name` (kurz `:i`) zeigt umfangreiche Informationen zu `name` an, z. B. `:i True`, `:i Bool` oder `:i Eq`.
- `:help` (kurz `:h`) öffnet die Hilfe mit weiteren nützlichen Befehlen.
- `:quit` (kurz `:q`) beendet den GHCi.

Aufgabe 0.2 *Fehlermeldungen des GHCi*

Die folgende Aufgabe enthält eine Reihe von fehlerhaften Haskell-Ausdrücken. Ziel dieser Aufgabe ist, dass Sie sich mit den Fehlermeldungen des GHCi vertraut machen. Laden Sie dazu die Datei aus Aufgabe 1 und interpretieren Sie die folgenden Ausdrücke mit dem GHCi. Versuchen Sie die Fehlermeldungen nachzuvollziehen.

a) `f 3 1 True`

b) `f 4 3 2 1`

c) `f 3 2 1`

d) `foo 3 2 1`

Aufgabe 0.3 *Painter-Paket*
Laden und Entpacken Sie das *Painter-Paket* von der Vorlesungsseite (`https://fldit-www.cs.uni-dortmund.de/fpba.html`). Das Modul `Examples` stellt die meisten in der Vorlesung vorgestellten Definitionen bereit.

a) Laden Sie die Datei `Examples.hs` in den GHCi.

b) Sie können Module mit der Anweisung `import` laden. Diese Anweisung kann im GHCi ausgeführt werden oder am Anfang einer Haskell-Datei (`.hs`) stehen. Legen Sie eine Haskell-Datei an. Beginnen Sie die Datei mit `import Examples` und laden Sie die Datei in GHCi.

Aufgabe 0.4 *Einführung in den GHC*
Mit dem Glasgow Haskell Compiler kann man auch ausführbare Dateien erzeugen. Dazu *muss* eine Funktion `main` vom Typ `IO ()` als Einstiegspunkt existieren. Speichern Sie folgendes Programm in einer Haskell-Datei:

```
main :: IO ()
main = putStrLn "Hello, world!"
```

Übersetzen Sie das Programm mit `ghc file.hs`. Es entsteht eine ausführbare Datei mit gleichem Namen (`file`) und der Dateiendung `.exe` bzw. keiner Endung, je nach Betriebssystem. Führen Sie diese Datei aus.

Aufgabe 0.5 *Hackage*
Besuchen Sie die Seite `https://hackage.haskell.org/`. Suchen Sie dort nach dem Paket `base`. Finden Sie in dem Paket das Modul `Prelude`. Hier finden Sie die Dokumentationen zu allen Funktionen, Datentypen, etc. die Ihnen automatisch in Haskell zur Verfügung stehen.

Sie können weitere Module aus dem Paket `base` oder anderen Paketen mit der Anweisung `import` nutzen. Eine vollständige Liste aller Pakete und damit aller Module der Haksell-Plattform finden Sie unter `https://www.haskell.org/platform/contents.html`. In der Veranstaltung werden lediglich die Module aus dem *Painter-Paket* und das Modul `Prelude` benutzt.

Aufgabe 0.6 *Hoogle*
Besuchen Sie die Seite `https://www.haskell.org/hoogle/`. Hier können Sie nach Funktionen, Datentypen und mehr suchen. Finden Sie heraus, was der Operator `$` macht.

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 1

Ausgabe: 13.10.2017, **Abgabe:** 20.10.2017 – 16:00 Uhr, **Block:** 1

> Das `case`-Konstrukt wird im Abschnitt *Gleichungen, die Funktionen definieren* auf den Folien 19–22 vorgestellt. Bitte lesen Sie diesen Abschnitt selbstständig.

Hinweis: Um dieses Übungsblatt zu lösen, sind folgende Äquivalenzen hilfreich:

```
x ⊗ y ⇔ (⊗) x y                                   (Operator als Funktion)
f x y ⇔ x `f` y                                   (Funktion als Operator)
\x -> ... \z -> e ⇔ \x ... z -> e                 (λ-Ausdrücke zusammenfassen)
f = \x ... z -> e ⇔ f x ... z = e                 (Applikative Definition)

f x1 ... z1 | g1 = e1  ⇔ f x ... z =              (Patternmatching Umformung)
          ⋮                  case (x, ..., z) of
f xN ... zN | gN = eN          (x1,...,z1) | g1 -> e1
                                       ⋮
                               (xN,...,zN) | gN -> eN
```

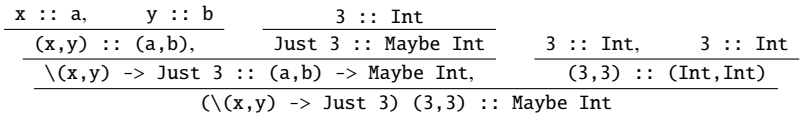
Aufgabe 1.1 (3 Punkte) *Typeinferenz*
Berechnen Sie die Typen der folgenden Ausdrücke mithilfe der Typinferenzregeln.

a) `(\(x,y) -> Just 3) (3,3)` mit `3 :: Int`

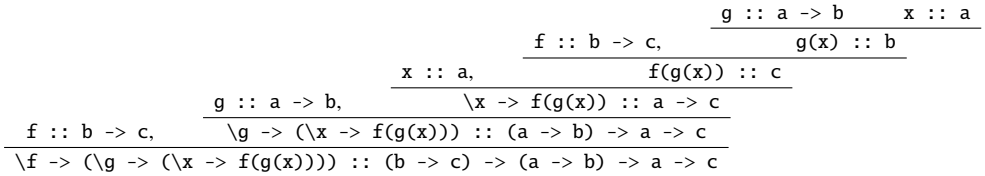
b) `\f g x -> f (g x)`

Lösungsvorschlag

a) `(\(x,y) -> Just 3) (3,3)`



b) `\f g x -> f (g x)` ⇔ `\f -> (\g -> (\x -> f(g(x))))`



Aufgabe 1.2 (3 Punkte) *λ-Ausdrücke Auswerten*
Werten Sie folgende Ausdrücke schrittweise aus.

a) `(\x y -> x * y) 3 2`

b) `(\f g x -> f (g x)) (\y -> y * 2) (\z -> z + 1)`

Lösungsvorschlag

a)

```
(\x y -> x * y) 3 2
<=> (\x -> \y -> x * y) 3 2
~> (\y -> 3 * y) 2
~> 3 * 2 ~> 6
```

b)

```
(\f g x -> f (g x)) (\y -> y * 2) (\z -> z + 1)
<=>(\f -> \g -> \x -> f (g x)) (\y -> y * 2) (\z -> z + 1)
~>(\g -> \x -> (\y -> y * 2) (g x)) (\z -> z + 1)
~>\x -> (\y -> y * 2) ((\z -> z + 1) x)
~>\x -> (\y -> y * 2) (x + 1)
~>\x -> (x + 1) * 2
```

Aufgabe 1.3 (3 Punkte) *Haskell-Funktion Auswerten*
Gegeben sei folgende Haskell-Funktion:

```
and' :: Bool -> Bool -> Bool
and' False _ = False
and' True  b = b
```

Werten Sie den Ausdruck `and' True True` aus, indem Sie erst `and'` in einen λ-Ausdruck umformen und dann schrittweise auswerten.

Lösungsvorschlag
Als λ-Ausdruck:

```
and' :: Bool -> Bool -> Bool
and' False _ = False
and' True  b = b
<=>
and' b1 b2 = case (b1,b2) of
  (False,_) -> False
  (True,b) -> b
<=>
and' = \b1 b2 -> case (b1,b2) of
  (False,_) -> False
  (True,b) -> b
<=>
and' = \b1 -> \b2 -> case (b1,b2) of
  (False,_) -> False
  (True,b) -> b
```

Auswertung:

```
and' True True
~> (\b1 -> \b2 -> case (b1,b2) of
      (False,_) -> False
      (True,b) -> b) True True
```

```
~> (\b2 -> case (True,b2) of
      (False,_) -> False
      (True,b) -> b) True
~> case (True,True) of
     (False,_) -> False
     (True,b) -> b
~> True
```

Aufgabe 1.4 (3 Punkte) *Haskell-Funktionen definieren*
Schreiben Sie eine Haskell-Funktionen `ite` vom Typ `Bool -> Int -> Int -> Int`, welche die erste Ganzzahl (`Int`) zurückgibt, falls der erste Parameter vom Wert `True` ist und die zweite Ganzzahl sonst.
Beispiele:

```
ite True 3 9 ~> 3
ite False 3 9 ~> 9
```

a) Definieren Sie die Funktion als λ-Ausdruck mit `case`.

b) Definieren Sie die Funktion applikativ.

Definieren Sie die Funktionen ohne `if_then_else_`-Ausdruck.

Lösungsvorschlag

a) λ-Ausdruck mit `case`:

```
ite = \b -> \x -> \y -> case (b,x,y) of
  (True,x,_) -> x
  (False,_,y) -> y
```

Einfacher:

```
ite = \b -> \x -> \y -> case b of
  True -> x
  False -> y
```

b) applikativ:

```
ite True  x _ = x
ite False _ y = y
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 2

Ausgabe: 20.10.2017, **Abgabe:** 27.10.2017 – 16:00 Uhr, **Block:** 1

Aufgabe 2.1 (4 Punkte) *Datentypen*

a) Modellieren folgende Eigenschaften mit Datentypen. Geben Sie den Attributen Namen.

- Ein Konto hat einen Kontostand und einen Kunden als Besitzer.
- Für einen Kunden werden die Daten Vorname, Name und Adresse (`String`) gespeichert.

b) Legen Sie ein Beispielkonto als Tupel mit Konstruktor an (ohne Attribute).

c) Legen Sie ein weiteres Beispielkonto als attributiertes Tupel an.

Lösungsvorschlag

```
data Account = Account { balance :: Int, owner :: Client }
  deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show

client1 :: Client
client1 = Client "Max" "Mustermann" "Musterhausen"
acc1 :: Account
acc1 = Account 100 client1

client2 :: Client
client2 = Client
  { name = "John"
  , surname = "Doe"
  , address = "Somewhere"
  }
acc2 :: Account
acc2 = Account{balance = 0, owner = client2}
```

Aufgabe 2.2 (2 Punkte) *Maybe*
Mit dem Datentyp `Maybe` können partielle Funktionen definiert werden. Definieren Sie eine Divisionsfunktion `div' :: Int -> Int -> Maybe Int`, welche bei der Division durch Null `Nothing` ausgibt.

Hinweis: Sie dürfen die Funktion `div` wiederverwenden.

Lösungsvorschlag

```
div' :: Int -> Int -> Maybe Int
_ `div'` 0 = Nothing
x `div'` y = Just (x `div` y)
```

Aufgabe 2.3 (3 Punkte) *Fallunterscheidungen nach Bedingungen*
Implementieren Sie folgende Funktionen in Haskell und geben Sie die Typen der Funktionen an.

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

Sie können die Haskell-Funktion `div` und `even` benutzen.

b) $f(x,y) = \begin{cases} y*2, & \text{falls } x = 0,\ y > 50 \\ y+2, & \text{falls } x = 0,\ y \leq 50 \\ x*2, & \text{falls } y = 0,\ x < 100 \\ x+y, & \text{sonst} \end{cases}$

Lösungsvorschlag

```
collatz :: Int -> Int
collatz n
  | even n    = div n 2
  | otherwise = 3 * n + 1

f :: (Int,Int) -> Int
f (0,y)
  | y > 50         = y*2
  | otherwise      = y+2
f (x,0) | x < 100 = x * 2
f (x,y)            = x + y
```

Aufgabe 2.4 (3 Punkte) *Präfixdarstellung*
Fügen Sie die impliziten Klammern in folgende Haskell-Ausdrücke ein. Wandeln Sie danach den Ausdruck in seine Präfixdarstellung.

a) `x + 3 * y * z`

b) `add3 1 2 3`

c) `f $ g . h x`

Lösungsvorschlag

a) Wie in der Mathematik gilt Punkt vor Strich. Bei Haskell sind Additionsoperator und Multiplikationsoperator linksassoziativ, d.h. die Operatoren werden von links nach rechts ausgewertet.

Klammern: `x + ((3 * y) * z)`
Präfix: `(+) x ((*) ((*) 3 y) z)`

b) Funktionsanwendungen sind auch linksassoziativ, werden also auch von links nach rechts ausgewertet. Funktionsanwendungen besitzen außerdem automatisch die höchste Priorität und werden vor allen anderen Operatoren ausgeführt.

Klammern: `((add3 1) 2) 3`
Der Ausdruck ist bereits in Präfixdarstellung.

c) Der Applikationsoperator (`$`) und die Komposition (`.`) sind beide rechtsassoziativ. Der Ausdruck wird also von rechts nach links ausgewertet. Die Funktionsanwendungen besitzt dabei die höchste Priorität, dann folgt die Komposition und dann der Applikationsoperator.

Klammern: `f $ (g . (h x))`
Präfix: `($) f ((.) g (h x))`

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 3

Ausgabe: 3.11.2017, **Abgabe:** 10.11.2017 – 16:00 Uhr, **Block:** 2

Aufgabe 3.1 (4 Punkte) *Endrekursion*
Formen Sie folgende Funktionen, die Schleifen enthalten, in *endrekursive* Funktionen um.

a) Eine Potenzfunktion, die mit einer Multiplikation arbeitet.

```
int power(int base, int expo) {
   int state = 1;
   while (expo > 0) {
       state = state * base;
       expo = expo - 1;
   }
   return state;
}
```

b) Eine Funktion die alle Zahlen in einem Feld summiert. Benutzen Sie in Haskell eine Liste anstelle des Feldes.

```
int summe(int[] ls) {
    int state = 0;
    int i = 0;
    while (i < ls.length) {
        state = state + ls[i];
        i = i + 1;
    }
    return state;
}
```

Lösungsvorschlag

```
power :: Int -> Int -> Int
power base expo = loop 1 expo
  where
    loop state expo
      | expo > 0  = loop (state * base) (expo - 1)
      | otherwise = state

summe :: [Int] -> Int
summe ls = loop 0 ls
  where
    loop state (x:xs) = loop (state + x) xs
    loop state []     = state
```

Aufgabe 3.2 (4 Punkte) *Listenfunktionen auswerten*
Werten Sie folgende Haskell-Ausdrücke schrittweise aus.

a) `take 2 $ tail [2,3,5,4,1]`

b) `head $ drop 2 [1,4,5,3,2]`

c) `foldl (-) 8 [5, 2]`

d) `foldr (-) 8 [5, 2]`

Lösungsvorschlag

a) `take 2 $ tail [2,3,5,4,1]`
   ⇝ `take 2 [3,5,4,1]`
   ⇝ `3 : take 1 [5,4,1]`
   ⇝ `3 : 5 : take 0 [4,1]`
   ⇝ `3 : 5 : []`
   ⇝ `[3,5]`

b) `head $ drop 2 [1,4,5,3,2]`
   ⇝ `head $ drop 1 [4,5,3,2]`
   ⇝ `head $ drop 0 [5,3,2]`
   ⇝ `head [5,3,2]`
   ⇝ `5`

c) `foldl (-) 8 [5, 2]` = $(8 - 5) - 2$
   `foldl (-) 8 [5, 2]`
   ⇝ `foldl (-) ((-) 8 5) [2]`
   ⇝ `foldl (-) 3 [2]`
   ⇝ `foldl (-) ((-) 3 2) []`
   ⇝ `foldl (-) 1 []`
   ⇝ `1`

d) `foldr (-) 8 [5, 2]` = $(5 - (2 - 8))$
   `foldr (-) 8 [5, 2]`
   ⇝ `foldr (-) 8 [5, 2]`
   ⇝ `(-) 5 $ foldr (-) 8 [2]`
   ⇝ `(-) 5 $ (-) 2 $ foldr (-) 8 []`
   ⇝ `(-) 5 $ (-) 2 $ 8`
   ⇝ `(-) 5 ((-) 2 8)`
   ⇝ `(-) 5 (-6)`
   ⇝ `11`

Aufgabe 3.3 (4 Punkte) *Listenfunktionen implementieren*
Implementieren Sie folgende Listenfunktionen in Haskell und geben Sie die Typen der Funktionen an. Es dürfen keine Hilfsfunktionen benutzt werden. Die Typen sollten möglichst allgemein sein.

a) Die Funktionen `safeHead` und `safeTail` sollen als absturzsichere Versionen der Funktionen `head` und `tail` implementiert werden. Machen Sie gebrauch vom `Maybe`-Datentyp.

b) Die Funktion `listEven` soll prüfen, ob die Anzahl der Elemente in einer Liste gerade sind.

Lösungsvorschlag

```
safeHead :: [a] -> Maybe a
safeHead (a:_) = Just a
safeHead []    = Nothing

safeTail :: [a] -> Maybe [a]
safeTail (_:as) = Just as
safeTail []     = Nothing

listEven :: [a] -> Bool
listEven []     = True
listEven (_:[]) = False
listEven (_:_:as) = listEven as
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 5

Ausgabe: 17.11.2017, **Abgabe:** 24.11.2017 – 16:00 Uhr, **Block:** 3

Aufgabe 5.1 (2 Punkte) *Unendliche Listen*
Schreiben Sie die angegebene Liste `solutions :: [(Int, Int, Int)]` so um, dass sie *alle* Lösungen der Gleichung $2x^3 + 5y + 2 = z^2$ enthält. Die Liste wird dadurch unendlich lang.

```
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z)
  | z <- [0..100]
  , y <- [0..100]
  , x <- [0..100]
  , 2*x^3 + 5*y + 2 == z^2
  ]
```

Lösungsvorschlag

Um eine Endlosschleife zu vermeiden, darf nur die erste Liste in der Komprehension unendlich sein. Da für $x, y > z$ keine Lösungen mehr existieren können, ist dies eine sinnvolle Einschränkung.

```
solutions :: [(Int , Int , Int )]
solutions = [ (x,y,z)
  | z <- [0..]
  , y <- [0..z^2]
  , x <- [0..z^2]
  , 2*x^3 + 5*y + 2 == z^2
  ]
```

Aufgabe 5.2 (3 Punkte) *Zahlen als Datentypelemente*
Diese Aufgabe bezieht sich auf die in der Vorlesung vorgestellten rekursiven Datentypen `Nat`, `Int'` und `PosNat`.

a) Definieren Sie eine Konstante *drei* = 3 für den Datentyp `Int'` in Haskell.

b) Erweitern Sie die Datentypen für Zahlen um einen Datentyp für rationale Zahlen. Basieren Sie den Datentyp nur auf den Datentypen `Nat`, `Int'` und `PosNat`.

c) Definieren Sie eine Konstante $c = -\frac{3}{2}$ für Ihren Datentyp in Haskell.

Lösungsvorschlag

```
drei :: Int'
drei = Plus (Succ' (Succ' One))

data Rat = Rat Int' PosNat

c :: Rat
c = Rat (Minus (Succ' (Succ' One))) (Succ' One)
```

Aufgabe 5.3 (4 Punkte) *Rekursive Datentypen*
Definieren Sie folgende Haskell-Funktionen.

a) `natLength :: [a] -> Nat`, wie `length` für den Datentyp `Nat`.

b) `natDrop :: Nat -> [a] -> [a]`, wie `drop` für den Datentyp `Nat` anstatt `Int`.

c) `colistIndex :: Colist a -> Int -> a`, wie `(!!)` für `Colist a` anstatt `[a]`.

d) `streamTake :: Int -> Stream a -> [a]`, wie `take` für `Stream a` anstatt `[a]`.

Lösungsvorschlag

```
natLength :: [a] -> Nat
natLength (_:s) = Succ (natLength s)
natLength []    = Zero

natDrop :: Nat -> [a] -> [a]
natDrop (Succ n) (a:s) = natDrop n s
natDrop Zero     s     = s
natDrop _        []    = []

colistIndex :: Colist a -> Int -> a
colistIndex (Colist (Just (a,_))) 0         = a
colistIndex (Colist(Just (_,s))) n | n > 0 = colistIndex s (n-1)

streamTake :: Int -> Stream a -> [a]
streamTake 0 _ = []
streamTake n (a :< s) | n > 0 = a : streamTake (n-1) s
```

Aufgabe 5.4 (3 Punkte) *Modellierung*
Gegeben seien folgende Datentypen:

```
type ID = Int
data Bank = Bank [(ID,Account)] deriving Show
data Account = Account { balance :: Int, owner :: Client } deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show
```

Definieren Sie folgende Funktionen.

a) `credit :: Int -> ID -> Bank -> Bank` – Addiert den angegebenen Betrag auf das angegebene Konto.

b) `debit :: Int -> ID -> Bank -> Bank` – Subtrahiert den angegebenen Betrag von dem angegebenen Konto.

c) `transfer :: Int -> ID -> ID -> Bank -> Bank` – Überweist den angegebenen Betrag vom ersten Konto auf das zweite.

Lösungsvorschlag

```
credit :: Int -> ID -> Bank -> Bank
credit amount id (Bank ls)
  = Bank (updRel ls id entry{ balance = oldBalance + amount})
  where
    Just entry = lookup id ls
    oldBalance = balance entry

debit :: Int -> ID -> Bank -> Bank
debit amount = credit (-amount)

transfer :: Int -> ID -> ID -> Bank -> Bank
transfer amount id1 id2 = debit amount id1 . credit amount id2
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 6

Ausgabe: 24.11.2017, **Abgabe:** 1.12.2017 – 16:00 Uhr, **Block:** 3

Aufgabe 6.1 (3 Punkte) *Arithmetische Ausdrücke*

a) Stellen Sie den Ausdruck $2x^3 + 5y + 2$ und z^2 als Elemente vom Typ `Exp String` da.

b) Schreiben Sie die Listenkomprehension `solutions :: [(Int,Int,Int)]` um. Machen Sie sinvollen gebrauch von den beiden Ausdrücken und `exp2store`.

```
solutions :: [(Int,Int,Int)]
solutions = [ (x,y,z)
  | z <- [0..]
  , y <- [0..z^2]
  , x <- [0..z^2]
  , 2*x^3 + 5*y + 2 == z^2
  ]
```

Hinweis: Importieren Sie das Modul `Expr`.

Lösungsvorschlag

a)
```
expr1, expr2 :: Exp String
expr1 = Sum [2 :* (Var "x" :^ 3), 5 :* Var "y", Con 2]
expr2 = Var "z" :^ 2
```

b)
```
solutions :: [(Int,Int,Int)]
solutions = [(x,y,z)
  | z <- [0..], x <- [0..z^2], y <- [0..z^2]
  , let st "x" = x
        st "y" = y
        st "z" = z
  , exp2store expr1 st == exp2store expr2 st
  ]
```

Aufgabe 6.2 (3 Punkte) *Boolesche Ausdrücke*
Schreiben Sie eine Funktion `bexp2store`, welche sich ähnlich wie `exp2store` verhält. Anstelle arithmetischer Ausdrücke sollen boolesche Ausdrücke vom Typ `BExp x` ausgewertet werden. Diese Funktion benötigt zwei Variablenbelegungen. Eine für boolesche Ausdrücke und die andere für arithmetische Ausdrücke.

Benutzen Sie folgende Typen:

```
type BStore x = x -> Bool
bexp2store :: BExp x -> Store x -> BStore x -> Bool
```

Hinweis: Importieren Sie das Modul `Expr`.

Lösungsvorschlag

```
bexp2store :: BExp x -> Store x -> BStore x -> Bool
bexp2store True_ _ _ = True
bexp2store False_ _ _ = False
bexp2store (BVar x) _ bst = bst x
bexp2store (Or bs) st bst = or $ map (\x -> bexp2store x st bst) bs
bexp2store (And bs) st bst = and $ map (\x -> bexp2store x st bst) bs
bexp2store (Not bs) st bst = not $ bexp2store bs st bst
bexp2store (e1 := e2) st _ = exp2store e1 st == exp2store e2 st
bexp2store (e1 :<= e2) st _ = exp2store e1 st <= exp2store e2 st
```

Aufgabe 6.3 (3 Punkte) *Typklassen*
Schreiben Sie eine Klasse für eine überladene Funktion `drop'`. Diese soll sich wie `drop` verhalten, aber nicht auf den Listentyp `[a]` beschränkt sein. Instanziieren Sie die Klasse für `[a]`, `Colist a` und `Stream a`.

Lösungsvorschlag

```
class Drop a where
  drop' :: Int -> a -> a

instance Drop [a] where
  drop' = drop

instance Drop (Colist a) where
  drop' 0 s = s
  drop' n ls = case split ls of
    Just (_,s) | n > 0 -> drop' (n-1) s
    Nothing            -> Colist Nothing

instance Drop (Stream a) where
  drop' 0 s = s
  drop' n (_:<s) | n > 0 = drop' (n-1) s
```

Aufgabe 6.4 (3 Punkte) *Binäre Bäume*
Gegeben seien folgende Datentypen:

```
data Bintree a = Empty | Fork a (Bintree a) (Bintree a) deriving Show
data Edge = Links | Rechts deriving Show
type Node = [Edge]
```

Definieren Sie folgende Funktionen.

a) `value :: Node -> Bintree a -> Maybe a` – Gibt den Wert des Knoten zurück. Falls der Knoten nicht existiert, wird `Nothing` ausgegeben.

b) `search :: Eq a => a -> Bintree a -> Maybe Node` – Durchsucht den Baum nach dem angegebenen Wert. Falls Knoten mit dem Wert existieren, wird der Erste ausgegeben. Ansonsten wird `Nothing` zurückgegeben. Die Suchreihenfolge ist beliebig.

Lösungsvorschlag

a)
```
value :: Node -> Bintree a -> Maybe a
value (Rechts:nodes) (Fork _ _ r) = value nodes r
value (Links:nodes) (Fork _ l _) = value nodes l
value [] (Fork a _ _) = Just a
value _ Empty = Nothing
```

b)
```
search :: Eq a => a -> Bintree a -> Maybe Node
search _ Empty = Nothing
search a (Fork b l r)
  | a == b    = Just []
  | otherwise = f (search a l) (search a r) where
      f (Just l) _ = Just $ Links:l
      f Nothing (Just r) = Just $ Rechts:r
      f _ _ = Nothing
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 7

Ausgabe: 1.12.2017, **Abgabe:** 8.12.2017 – 16:00 Uhr, **Block:** 4

Aufgabe 7.1 (6 Punkte)

a) Schreiben Sie eine Instanz der Klasse `Eq` für den Typ `Nat`.

b) Schreiben Sie eine Instanz der Klasse `Ord` für den Typ `Nat`. Es ist ausreichend den Operator `(<=)` zu definieren.

c) Schreiben Sie eine Instanz der Klasse `Enum` für den Typ `Nat`. Es ist ausreichend die Funktionen `toEnum` und `fromEnum` zu definieren.
Beispiel:
`take 3 $ map fromEnum [Zero .. ] ⇝ [0,1,2]`

d) Schreiben Sie eine Instanz der Klasse `Show` für den Typ `Nat`. Die Ausgabe soll sich ähnlich wie vom Typ `Int` verhalten:

```
show Zero ~> "0"
show (Succ Zero) ~> "1"
show (Succ (Succ (Succ Zero))) ~> "3"
```

e) Schreiben Sie eine Instanz der Klasse `Num` für den Typ `Nat`. Nutzen Sie folgende Vorgabe:

```
instance Num Nat where
  negate = undefined
  abs n  = n
  signum Zero = Zero
  signum n    = Succ Zero
  fromInteger = toEnum . fromInteger
```

Sie müssen lediglich die fehlenden Operatoren `(+)` und `(*)` definieren (Stichwort: Peano-Axiome). Sie dürfen nur die beiden Operatoren `(+)` und `(*)` benutzen. Weitere Hilfsfunktionen sind nicht erlaubt.

f) Ändern Sie den Typ der Liste `solutions` in `[(Nat,Nat,Nat)]` und passen Sie die Definition entsprechend an.

```
solutions :: [(Int,Int,Int)]
solutions = [ (x,y,z)
  | z <- [0..] , y <- [0..z^2], x <- [0..z^2]
  , 2*x^3 + 5*y + 2 == z^2 ]
```

Lösungsvorschlag

```
instance Eq Nat where
  Zero == Zero     = True
  Succ n == Succ m = n == m
  _ == _           = False

instance Ord Nat where
  Succ n <= Succ m = n <= m
  Zero <= _        = True
  _ <= _           = False

instance Enum Nat where
  toEnum 0          = Zero
  toEnum n | n > 0  = Succ (toEnum (n - 1))
  fromEnum Zero     = 0
  fromEnum (Succ n) = fromEnum n + 1

instance Show Nat where
  showsPrec _ n = shows (fromEnum n)

instance Num Nat where
  Zero + n     = n
  (Succ n) + m = Succ (n + m)
  Zero * n     = Zero
  (Succ n) * m = m + (n * m)
  negate = undefined
  abs n  = n
  signum Zero = Zero
  signum n    = Succ Zero
  fromInteger = toEnum . fromInteger

solutions :: [(Nat,Nat,Nat)]
solutions = [ (x,y,z)
  | z <- [0..] , y <- [0..z^2], x <- [0..z^2]
  , 2*x^3 + 5*y + 2 == z^2 ]
```

Aufgabe 7.2 (3 Punkte) *Ausgabe*
Schreiben Sie eine Instanz der Klasse `Show` für den folgenden Datentyp für nichtleere binäre Bäume:

```
data STree a = BinS (STree a) a (STree a) | LeftS  (STree a) a
             | RightS  a (STree a)  | LeafS a
```

Die Ausgabe soll sich an den binären Bäumen aus der Vorlesung orientieren:
`BinS (LeftS (LeafS 9) 2) 4 (RightS 7 (LeafS 3))` $\leadsto$ `4(2(9,),7(,3))`

Lösungsvorschlag

```
instance Show a => Show (STree a) where
  showsPrec _ (LeafS a) = shows a
  showsPrec _ (LeftS l a)
    = shows a . showChar '(' . shows l . showString ",)"
  showsPrec _ (RightS a r)
    = shows a . showString "(," . shows r . showChar ')'
  showsPrec _ (BinS l a r) = shows a . showChar '('
    . shows l . showChar ',' . shows r . showChar ')'
```

Aufgabe 7.3 (3 Punkte) *Bäume mit beliebigem Ausgrad*
Definieren Sie folgende Funktionen.

a) `treeAnd :: Tree Bool -> Bool` – Verhält sich wie `and`. Ergibt `True`, falls alle Knoten den Wert `True` enthalten.

b) `treeZip :: Tree a -> Tree b -> Tree (a,b)` – Eine Variante von `zip` für Bäume.

Lösungsvorschlag

```
treeAnd :: Tree Bool -> Bool
treeAnd (F b as) = b && all treeAnd as
treeAnd (V b) = b

treeZip :: Tree a -> Tree b -> Tree (a,b)
treeZip (F a as) (F b bs) = F (a,b) $ zipWith treeZip as bs
treeZip a b = V (root a,root b)
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 8

Ausgabe: 8.12.2017, **Abgabe:** 15.12.2017 – 16:00 Uhr, **Block:** 4

Aufgabe 8.1 (3 Punkte) *Baumfaltungen*
Definieren Sie folgende Funktionen als Baumfaltungen (`foldBtree` bzw. `foldTree`) und geben Sie den Typ an. Wählen Sie den Typ möglichst allgemein.

a) `or_ :: Tree Bool -> Bool` – Verhält sich wie `or`. Enthält ein Knoten `True`, ist das Ergebnis `True`, sonst `False`.

b) `preorderB :: Bintree a -> [a]` – Die Knoten des Binärbaumes (`Bintree`) werden als Liste in Hauptreihenfolge (pre-order) wiedergegeben.

Lösungsvorschlag

```
or_ :: Tree Bool -> Bool
or_ = foldTree id (||) False (||)

preorderB :: Bintree a -> [a]
preorderB = foldBtree [] (\a l r -> a : l ++ r)
```

Aufgabe 8.2 (3 Punkte) *Faltungen*

a) Schreiben Sie die Faltung `foldPosNat` für den Typen `PosNat`.

b) Schreiben Sie eine Funktion `toInt` die Werte vom Typ `PosNat` in entsprechende Werte vom Typ `Int` wandelt. Benutzen Sie dazu die Faltung `foldPosNat`.

Lösungsvorschlag

```
foldPosNat :: val -> (val -> val) -> PosNat -> val
foldPosNat val _ One = val
foldPosNat val f (Succ' n) = f (foldPosNat val f n)

toInt :: PosNat -> Int
toInt = foldPosNat 1 (+1)
```

Aufgabe 8.3 (3 Punkte) *Arithmetische Ausdrücke kompilieren*
Geben Sie die Kommandosequenz für die Auswertung `execute (exp2code expr) ([],vars)` an. Zu jedem Kommando soll auch der Stapelinhalt (Stack) nach der Ausführung angegeben werden. Dabei sei der Ausdruck `expr` und die Belegungsfunktion `vars` wie folgt definiert:

```
expr :: Exp String
expr = Prod [Con 2, Con 9 :- Var "x"]

vars :: Store String
vars "x" = 6
```

Mit `getResult (execute (exp2code expr) ([],vars))` kann das Ergebnis angezeigt werden. Diese Hilfsfunktion wird für die Bearbeitung der Aufgabe nicht benötigt.

```
getResult :: Estate x -> Int
getResult = head . fst
```

Lösungsvorschlag

Kommando	Stapel
`Push 2`	`[2]`
`Push 9`	`[9,2]`
`Load "x"`	`[6,9,2]`
`Sub`	`[3,2]`
`Mul 2`	`[6]`

Aufgabe 8.4 (3 Punkte) *Fixpunkte*
Gegeben sei folgender Datentyp für den Restklassenring $\mathbb{Z}_{10}$:

```
data Mod10 = Z0 | Z1 | Z2 | Z3 | Z4
           | Z5 | Z6 | Z7 | Z8 | Z9
           deriving (Show, Eq, Ord, Enum)

succ10 Z9 = Z0
succ10 x  = succ x

pred10 Z0 = Z9
pred10 x  = pred x
```

Definieren Sie folgende Funktionen mithilfe der Fixpunktfunktion `fixpt`.

a) `lfp, gfp :: Mod10` – Der kleinste bzw. größte Fixpunkt der Funktion `f`.

```
f :: Mod10 -> Mod10
f x | x < Z4 = succ10 x
    | x > Z6 = pred10 x
    | otherwise      = x
```

Aufgrund der Typklasse `Ord` wird `Mod10` ein CPO mit

$$\texttt{Z}i \texttt{ <= Z}j \equiv i \leq j \text{ für } i, j \in \mathbb{Z}_{10}$$

und `maximum` als Supremumsbildung.

b) `evens :: [Mod10]` – Die kleinste Lösung der Gleichung

$$evens = \{0\} \cup \{x + 2 \mid x \in evens\}\,.$$

Dabei entspricht `[Mod10]` dem Potenzmengenverband $\mathcal{P}(\mathbb{Z}_{10})$ (Folie 143). Es wird eine Schrittfunktion $\Phi : \mathcal{P}(\mathbb{Z}_{10}) \to \mathcal{P}(\mathbb{Z}_{10})$ bzw. `phi :: [Mod10] -> [Mod10]` benötigt. Die Fixpunkte der Funktion müssen den Lösungen der Gleichung entsprechen.

Lösungsvorschlag

```
lfp, gfp :: Mod10
lfp = fixpt (<=) f Z0
gfp = fixpt (>=) f Z9

evens :: [Mod10]
evens = fixpt subset phi [] where
  phi :: [Mod10] -> [Mod10]
  phi m = [Z0] `union` [succ10 (succ10 x) | x <- m]
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 9

Ausgabe: 15.12.2017, **Abgabe:** 22.12.2017 – 16:00 Uhr, **Block:** 5

Aufgabe 9.1 (4 Punkte)

a) Zeigen Sie, dass die Rekursionsgleichung `fib` eine Funktion definiert. Definieren Sie dazu eine Schrittfunktion Φ analog zu der auf Folie 146.

b) Beweisen Sie durch Induktion, dass $\mathrm{lfp}(\Phi)$ keine natürliche Zahl auf $\bot$ abbildet.

Lösungsvorschlag

a)

$$\begin{aligned}\Phi : (\mathbb{N} \to \mathbb{N}_\bot) &\to (\mathbb{N} \to \mathbb{N}_\bot)\\ f &\mapsto \lambda n.\text{if } n > 1 \text{ then } f(n-1) + f(n-2) \text{ else } 1\end{aligned}$$

Φ ist stetig, wenn man die Addition zur *strikten* Funktion auf $\mathbb{N}_\bot$ erweitert, d. h. $n + \bot$ und $\bot + n$ auf $\bot$ setzt.

b) Induktionsvoraussetzung: $\mathrm{lfp}(\Phi)(n) \neq \bot$ und $\mathrm{lfp}(\Phi)(m) \neq \bot$ für alle $m < n$.
Induktionsanfang:

$$\begin{aligned}&\mathrm{lfp}(\Phi)(0)\\ =&\Phi(\mathrm{lfp}(\Phi))(0)\\ =&(\lambda n.\text{if } n > 1 \text{ then } \mathrm{lfp}(\Phi)(n-1) + \mathrm{lfp}(\Phi)(n-2) \text{ else } 1)(0)\\ =&1 \neq \bot\end{aligned}$$

$$\begin{aligned}&\mathrm{lfp}(\Phi)(1)\\ =&\Phi(\mathrm{lfp}(\Phi))(1)\\ =&(\lambda n.\text{if } n > 1 \text{ then } \mathrm{lfp}(\Phi)(n-1) + \mathrm{lfp}(\Phi)(n-2) \text{ else } 1)(1)\\ =&1 \neq \bot\end{aligned}$$

Induktionsschritt:

$$\begin{aligned}&\mathrm{lfp}(\Phi)(n+1)\\ =&\Phi(\mathrm{lfp}(\Phi))(n+1)\\ =&(\lambda n.\text{if } n > 1 \text{ then } \mathrm{lfp}(\Phi)(n-1) + \mathrm{lfp}(\Phi)(n-2) \text{ else } 1)(n+1)\\ =&\mathrm{lfp}(\Phi)(n) + \mathrm{lfp}(\Phi)(n-1)\\ &(\textit{Induktionsvoraussetzung})\\ \neq&\bot\end{aligned}$$

Aufgabe 9.2 (3 Punkte)
Definieren Sie folgende Haskell-Funktionen.

a) `isCyclic :: Eq a => Graph a -> Bool` – Erkennt, ob ein Graph zyklisch ist. Sie können hier den transitiven Abschluss nutzen.
Beispiele:
`isCyclic graph1` ⇝ `True`
`isCyclic graph2` ⇝ `False`

b) `depthFirst :: Eq a => a -> Graph a -> [a]` – Ähnlich wie `preorder` und `postorder` auf Bäumen, sollen die Knoten des Graphen in einer bestimmten Reihenfolge als Liste ausgegeben werden. Die Funktion erhält einen Startknoten und gibt dann weitere Knoten durch Tiefensuche aus.

Lösungsvorschlag

```
isCyclic :: Eq a => Graph a -> Bool
isCyclic graph = any self nodes where
  G nodes closure = closureF graph
  self node = node `elem` closure node

depthFirst :: Eq a => a -> Graph a -> [a]
depthFirst start (G _ adj) = df [start] [] where
  df (a:as) visited
    | a `elem` visited = df as visited
    | otherwise        = df (adj a ++ as) (visited ++ [a])
  df [] visited        = visited
```

Aufgabe 9.3 (2 Punkte) *Kinds*
Bestimmen Sie den Kind folgender Typkonstruktoren.

a)
```
class C f where
  comp :: f b c -> f a b -> f a c
```
Bestimmen Sie den Kind von `f`.

b) `data T f g = T (f String Int) (g Bool)`
Bestimmen Sie den Kind von `T`.

Lösungsvorschlag

a) `f :: * -> * -> *`

b) `T :: (* -> * -> *) -> (* -> *) -> *`

Aufgabe 9.4 (3 Punkte) *Typfamilien*
Gegeben sei folgende Typklasse:

```
class Listable l where
  type Item l :: *
  toList :: l -> [Item l]
```

Die Funktion `toList` wandelt eine Eingabe in eine Liste um. Überladen Sie die Funktion für die angegebenen Typen.

a) `Colist a` – Gibt die zugehörige Liste aus.

b) `data Map a b = Map [(a,b)]` – Ein Datentyp für eine Assoziationsliste (siehe Folie 53). Es sollen in der Liste nur die Werte ausgegeben werden. Die Argumente bzw. Schlüssel entfallen.

c) `Nat` – Die bijektive Funktion der Isomorphie von `Nat` und `[()]`.

Hinweis: Die Typklasse macht gebrauch von einer Typfamilie. Um Typfamilien zu nutzen, muss die Spracherweiterung *TypeFamilies* aktiviert werden. Fügen Sie das Pragma

```
{-# LANGUAGE TypeFamilies #-}
```

an den Kopf Ihrer Haskell-Datei ein.

Lösungsvorschlag

```
instance Listable (Colist a) where
  type Item (Colist a) = a
  toList ls = case split ls of
    Just (a,as) -> a : toList as
    Nothing -> []

instance Listable (Map a b) where
  type Item (Map a b) = b
  toList (Map ((a,b):as)) = b : toList (Map as)
  toList (Map [])         = []

instance Listable Nat where
  type Item Nat = ()
  toList Zero     = []
  toList (Succ n) = ():toList n
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 10

Ausgabe: 22.12.2017, **Abgabe:** 12.1.2017 – 16:00 Uhr, **Block:** 5

Aufgabe 10.1 (2 Punkte) *Funktor*
Schreiben Sie eine Instanz der Klasse `Functor` für den folgenden Datentyp für nichtleere binäre Bäume:

```
data STree a = BinS (STree a) a (STree a) | LeftS (STree a) a
             | RightS a (STree a) | LeafS a deriving Show
```

Beispiel: `fmap succ $ BinS (LeftS (LeafS 9) 2) 4 (RightS 7 (LeafS 3))`
`  ⇝ BinS (LeftS (LeafS 10) 3) 5 (RightS 8 (LeafS 4))`

Lösungsvorschlag

```
instance Functor STree where
  fmap f t = case t of
    BinS l a r -> BinS (fmap f l) (f a) (fmap f r)
    LeftS l a -> LeftS (fmap f l) (f a)
    RightS a r -> RightS (f a) (fmap f r)
    LeafS a -> LeafS (f a)
```

Aufgabe 10.2 (3 Punkte) *do-Notation*
Gegeben sei folgende Listenkomprehension:

```
solutions :: [(Int , Int , Int )]
solutions = [ (x,y,z) | z <- [0..] , y <- [0..z^2] , x <- [0..z^2]
            , 2*x^3 + 5*y + 2 == z^2]
```

a) Überführen Sie die Listenkomprehension in die do-Notation.

b) Überführen Sie die do-Notation in monadische Operatoren und Funktionen (>>=,>>,`return`).

Lösungsvorschlag

a)
```
solutions' :: [(Int , Int , Int )]
solutions' = do
   z <- [0..]
   y <- [0..z^2]
   x <- [0..z^2]
   guard $ 2*x^3 + 5*y + 2 == z^2
   return (x,y,z)
```

b)
```
solutions'' :: [(Int , Int , Int )]
solutions'' =
  [0..] >>= \z ->
  [0..z^2] >>= \y ->
  [0..z^2] >>= \x ->
  (guard $ 2*x^3 + 5*y + 2 == z^2) >>
  return (x,y,z)
```

Aufgabe 10.3 (3 Punkte) *Plusmonade*
Implementieren Sie die Funktion `preorderM` als Verallgemeinerung der Funktion `preorderB`:

`preorderM :: MonadPlus m => Bintree a -> m a`.

Die Funktion `preorderM` soll sich bei einer Festlegung des Rückgabetyps auf eine Liste von Werten wie `preorderB` verhalten:

`preorderM $ Fork 3 (leaf 4) (leaf 5) :: [Int]` $\rightsquigarrow$ `[3,4,5]`

Darüber hinaus soll `preorderM` aber beispielsweise auch für einen in `Maybe` eingebetteten Wert funktionieren und dann den Wert des Wurzelknotens zurückgeben:

`preorderM $ Fork 3 (leaf 4) (leaf 5) :: Maybe Int` $\rightsquigarrow$ `Just 3`

Ist der Baum leer, gibt die Funktion für den Rückgabetyp `Maybe Int` den Fehlerwert `Nothing` zurück.

Lösungsvorschlag

```
preorderM :: MonadPlus m => Bintree a -> m a
preorderM Empty = mzero
preorderM (Fork a l r) = msum [return a, preorderM l, preorderM r]
```

> Für die folgenden Aufgaben lesen Sie bitte selbstständig den Abschnitt *Maybe- und Listenmonade* auf den Folien 189 und 190.

Aufgabe 10.4 (2 Punkte) *`Maybe`-Monade*
Gegeben sei die sichere Division `sdiv`.

```
sdiv :: Int -> Int -> Maybe Int
_ `sdiv` 0 = Nothing
x `sdiv` y = Just $ x `div` y
```

Definieren Sie die Funktion `f` vom Typ `Int -> Int -> Int -> Maybe Int`, welche die Gleichung

$$f(x,y,z) = \frac{\frac{18}{x}}{y} + \frac{6}{z}$$

erfüllt. Für die Divisionen muss die Funktion `sdiv` benutzt werden. Nutzen Sie nur die Monadeneigenschaft von `Maybe` und lösen Sie die Aufgabe in der `do`-Notation.
Beispiel: `f 2 3 2` $\rightsquigarrow$ `Just 6`

Lösungsvorschlag

```
f :: Int -> Int -> Int -> Maybe Int
f x y z = do
  r1 <- 18 `sdiv` x
  r2 <- r1 `sdiv` y
  r3 <- 6 `sdiv` z
  return $ r2 + r3
```

Aufgabe 10.5 (2 Punkte) *Listenmonade*
Gegeben sei die nichtdeterministische Transitionsfunktion `delta`, welche auf `graph1` basiert.

```
G _ delta = graph1
```

Definieren Sie die Funktion `g` vom Typ `Int -> Int -> [Int]`, welche die Gleichung

$$g(x, y) = \delta(x) * \delta(\delta(y))$$

erfüllt. Nutzen Sie nur die Monadeneigenschaft der Liste und lösen Sie die Aufgabe in der `do`-Notation.
Beispiel: `g 1 3` $\rightsquigarrow$ `[4,6,2,4,8,10,6,9,3,6,12,15]`

Lösungsvorschlag

```
g :: Int -> Int -> [Int]
g x y = do
  r1 <- delta x
  r2 <- delta y
  r3 <- delta r2
  return $ r1 * r3
```

Frohes Fest und einen guten Rutsch ins neue Jahr

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 11

Ausgabe: 12.2.2017, **Abgabe:** 19.2.2017 – 16:00 Uhr, **Block:** 6

Lesen Sie bitte selbstständig die Abschnitte *Lesermonaden* und *Schreibermonaden* auf den Folien 205–208 und den Abschnitt *Zustandsmonaden* auf den Folien 213–215.

Aufgabe 11.1 (4 Punkte) *Lesermonade*
Schreiben Sie die Funktion `bexp2store` so um, dass sie Gebrauch von der Lesermonade macht. Verwenden Sie die do-Notation.

```
type BStore x = x -> Bool

bexp2store :: BExp x -> Store x -> BStore x -> Bool
bexp2store True_ _ _ = True
bexp2store False_ _ _ = False
bexp2store (BVar x) _ bst = bst x
bexp2store (Or bs) st bst = or $ map (\x -> bexp2store x st bst) bs
bexp2store (And bs) st bst = and $ map (\x -> bexp2store x st bst) bs
bexp2store (Not b) st bst = not $ bexp2store b st bst
bexp2store (e1 := e2) st _ = exp2store e1 st == exp2store e2 st
bexp2store (e1 :<= e2) st _ = exp2store e1 st <= exp2store e2 st
```

Lösungsvorschlag

```
bexp2store :: BExp x -> Store x -> BStore x -> Bool
bexp2store True_ _ = return True
bexp2store False_ _ = return False
bexp2store (BVar x) _ = do
  bst <- id
  return $ bst x
bexp2store (Or bs) st = do
  is <- mapM (\x -> bexp2store x st) bs
  return $ or is
bexp2store (And bs) st = do
  is <- mapM (\x -> bexp2store x st) bs
  return $ and is
bexp2store (Not b) st = do
  i <- bexp2store b st
  return $ not i
bexp2store (e1 := e2) st = return $ exp2store e1 st == exp2store e2 st
bexp2store (e1 :<= e2) st = return $ exp2store e1 st <= exp2store e2 st
```

Aufgabe 11.2 (4 Punkte) *Schreibermonade*
Gegeben sei folgende Vorlage.

```
type ID = Int
type Bank = [(ID,Account)]
data Account = Account { balance :: Int, owner :: Client } deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show

own1, own2, own3 :: Client
own1 = Client "Max" "Mustermann" "Musterhausen"
own2 = Client "John" "Doe" "Somewhere"
own3 = Client "Erika" "Mustermann" "Musterhausen"

acc1, acc2, acc3 :: Account
acc1 = Account 100 own1
acc2 = Account 0 own2
acc3 = Account 50 own3

bank :: Bank
bank = [(1,acc1), (2,acc2), (3,acc3)]

credit :: Int -> ID -> Bank -> Bank
credit amount id ls
  = updRel ls id entry{ balance = oldBalance + amount} where
    Just entry = lookup id ls
    oldBalance = balance entry

debit :: Int -> ID -> Bank -> Bank
debit amount = credit (-amount)
```

Erweitern Sie die Vorlage um eine Transferfunktion mit Protokollierung (Log). Machen Sie hierfür Gebrauch von einer Schreibermonade mit `String` für die Protokollierung (`(,) String`).

a) Eine Loggerfunktion `logMsg`, welche einen beliebigen `String` an das Protokoll anhängt. Geben Sie auch den Typ der Funktion an.

b) `transferLog :: Int -> ID -> ID -> Bank -> (String,Bank)` – Überweist den angegebenen Betrag vom ersten Konto auf das zweite und schreibt einen Eintrag in das Protokoll. Der Eintrag soll folgendes Format besitzen.

   "Der Betrag *<ammount>* wurde von Konto *<id1>* auf Konto *<id2>* übertragen."

   Verwenden Sie die do-Notation. Der Tupelkonstruktor `(,)` darf *nicht* benutzt werden.

c) `transactions :: Bank -> (String,Bank)` – Soll Transaktionen durchführen, welche zu folgenden Protokolleinträgen passen.

```
putStrLn $ fst $ transactions bank ↪

Der Betrag 50 wurde von Konto 1 auf Konto 2 übertragen.
Der Betrag 25 wurde von Konto 1 auf Konto 3 übertragen.
Der Betrag 25 wurde von Konto 2 auf Konto 3 übertragen.
```

Verwenden Sie die do-Notation. Der Tupelkonstruktor `(,)` darf *nicht* benutzt werden.

Lösungsvorschlag

```
logMsg :: String -> (String,())
logMsg msg = (msg,())

transferLog :: Int -> ID -> ID -> Bank -> (String,Bank)
transferLog amount id1 id2 bank = do
  logMsg "Der Betrag "
  logMsg $ show amount
  logMsg " wurde von Konto "
  logMsg $ show id1
  logMsg " auf Konto "
  logMsg $ show id2
  logMsg " übertragen.\n"
  return $ debit amount id1 $ credit amount id2 bank

transactions :: Bank -> (String,Bank)
transactions bank = do
  bank2 <- transferLog 50 1 2 bank
  bank3 <- transferLog 25 1 3 bank2
  transferLog 25 2 3 bank3
```

Aufgabe 11.3 (4 Punkte) *Zustandsmonade*
Benutzen Sie die Vorlage aus Aufgabe 11.2 und erweitern Sie diese um die folgenden Funktionen.

a) `putAccount :: ID -> Account -> State Bank ()` – Erstellt ein neues Konto mit angegebener `ID`. Ist bereits ein Konto mit der `ID` vorhanden, wird dieses überschrieben.

b) `getAccount :: ID -> State Bank (Maybe Account)` – Falls vorhanden, wird das Konto mit der `ID` ausgegeben.

c) `creditS :: Int -> ID -> State Bank ()` – Addiert den angegebenen Betrag auf das angegebene Konto. Verwenden Sie die do-Notation. Konstruktor und Destruktor von `State` (`State`, `runS`) dürfen *nicht* verwendet werden.

d) `debitS :: Int -> ID -> State Bank ()` – Subtrahiert den angegebenen Betrag von dem angegebenen Konto. Verwenden Sie die do-Notation. Konstruktor und Destruktor von `State` dürfen *nicht* verwendet werden.

e) `transferS :: Int -> ID -> ID -> State Bank ()` – Überweist den angegebenen Betrag vom ersten Konto auf das zweite. Verwenden Sie die `do`-Notation. Konstruktor und Destruktor von `State` dürfen *nicht* verwendet werden.

Beispiel: Mit `transferS 25 1 3` werden 25 Geldeinheiten von Konto 1 auf Konto 3 übertragen.

```
map (fmap balance) $ snd $ runS (transferS 25 1 3) bank
  ⇝[(1,75),(2,0),(3,75)]
```

Lösungsvorschlag

```
putAccount :: ID -> Account -> State Bank ()
putAccount id acc = State $ \bank -> ((), updRel bank id acc)

getAccount :: ID -> State Bank (Maybe Account)
getAccount id = State $ \bank -> (lookup id bank,bank)

creditS  :: Int -> ID -> State Bank ()
creditS amount id = do
  Just entry <- getAccount id
  let oldBalance = balance entry
  putAccount id entry{ balance = oldBalance + amount }

debitS :: Int -> ID -> State Bank ()
debitS amount = creditS (-amount)

transferS :: Int -> ID -> ID -> State Bank ()
transferS amount id1 id2 = do
  debitS amount id1
  creditS amount id2
```

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2017/2018

Übungen zu Funktionaler Programmierung
Übungsblatt 12

Ausgabe: 19.2.2017, **Abgabe:** 26.2.2017 – 16:00 Uhr, **Block:** 6

Importieren Sie die Module `Examples`, `Coalg` und `Data.Array`.

Aufgabe 12.1 (3 Punkte) *Zustandsmonade*
Schreiben Sie folgende Funktionen.

a) `getX, getY :: State Point Float` – Liest die Koordinate *x* bzw. *y* eines Punktes aus.

b) `setX, setY :: Float -> State Point ()` – Setzt die Koordinate *x* bzw. *y* eines Punktes.

c) `rotate :: (Float,Float) -> Float -> State Point ()` – Eine zustandsbasierte Variante der Funktion `rotate` von Folie 29. Der erste Parameter gibt die Koordinaten an, um die gedreht werden soll. Der zweite Parameter gibt den Winkel an.
Beispiel: `runS (rotate (4,5) 180) $ Point 5 8` ⇝ `((),(3.0000002,2.0))`

Lösungsvorschlag

```
getX, getY :: State Point Float
getX = State $ \pt -> (x pt, pt)
getY = State $ \pt -> (y pt, pt)

setX, setY :: Float -> State Point ()
setX x = State $ \pt -> ((),pt{x = x})
setY y = State $ \pt -> ((),pt{y = y})

rotate :: (Float,Float) -> Float -> State Point ()
rotate _ 0 = return ()
rotate (i,j) a = do
  x <- getX
  y <- getY
  let x1 = x - i
      y1 = y - j
      s = sin rad
      c = cos rad
      rad = a * pi / 180
  when (not $ x == i && y == j) $ do
    setX $ i+x1*c-y1*s
    setY $ j+x1*s+y1*c
```

Aufgabe 12.2 (4 Punkte) *IO-Monade*
Schreiben Sie eine Funktion `main :: IO ()`, welches eine Textdatei einliest und den Inhalt in Groß- oder Kleinbuchstaben in eine andere Datei schreibt. Das Programm soll interaktiv sein. Der Benutzer kann die Eingabedatei, die Ausgabedatei und die Art der Änderung bestimmen. Ein Ablauf soll wie folgt aussehen:

```
Eingabedatei: <Eingabe>
Ausgabedatei: <Eingabe>
Ändere Eingabe in...
1. Großbuchstaben
2. Kleinbuchstaben
<Eingabe>
```

Übersetzen Sie das Programm mit dem Befehl

```
ghc <filename>.hs
```

in eine ausführbare Datei und führen Sie diese aus.

Hinweis: Sie können die Funktionen `toUpper` und `toLower` für die Umwandlung benutzen. Importieren Sie dazu das Modul `Data.Char`.

Lösungsvorschlag

```
main :: IO ()
main = do
  putStr "Eingabedatei: "
  inputFile <- getLine
  putStr "Ausgabedatei: "
  outputFile <- getLine
  putStrLn "Ändere Eingabe in..."
  putStrLn "1. Großbuchstaben"
  putStrLn "2. Kleinbuchstaben"
  putStr "Auswahl (1-2):"
  option <- getLine
  let function = case read option of
        1 -> toUpper
        2 -> toLower
  input <- readFile inputFile
  writeFile outputFile $ map function input
```

Aufgabe 12.3 (2 Punkte) *Felder*
Gegeben sei die bekannte Funktion `credit`.

```
type ID = Int
type Bank = [(ID,Account)]
data Account = Account { balance :: Int, owner :: Client } deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show

credit :: Int -> ID -> Bank -> Bank
credit amount id ls
  = updRel ls id entry{ balance = oldBalance + amount} where
    Just entry = lookup id ls
    oldBalance = balance entry
```

Ändern Sie das Typsynonym `Bank` in

```
type Bank = Array ID Account
```

und schreiben Sie die Funktion `credit` entsprechend um.

Lösungsvorschlag

```
credit :: Int -> ID -> Bank -> Bank
credit amount id bank
  = bank // [(id, entry{ balance = oldBalance + amount})]where
    entry = bank ! id
    oldBalance = balance entry
```

Aufgabe 12.4 (3 Punkte) *Dynamische Programmierung*
Gegeben sei folgende rekursive Berechnung des Binomialkoeffizienten:

```
bincoeff :: (Int,Int) -> Int
bincoeff(n,k)
  | k == 0 || k == n = 1
  | 0 < k, k < n = bincoeff(n-1,k-1) + bincoeff(n-1,k)
  | otherwise = 0
```

Definieren Sie eine Funktion `bincoeffDyn` vom Typ `(Int,Int) -> Int`, welche das Problem mithilfe der dynamischen Programmierung löst.

Hinweis: Tupel sind Instanzen der Klasse `Ix` und können daher auch zur Indizierung von Feldern genutzt werden.

Lösungsvorschlag

```
bincoeffDyn :: (Int,Int) -> Int
bincoeffDyn t = bincoeffArr ! t where
  bincoeffArr = mkArray ((0,0),t) bincoeffFun
  bincoeffFun (n,k)
    | k == 0 || k == n = 1
    | 0 < k, k < n = bincoeffArr ! (n-1,k-1) + bincoeffArr ! (n-1,k)
    | otherwise = 0
```