technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Präsenzblatt

Ausgabe: 8.10.2018, **Abgabe:** keine Abgabe, **Block:** N.N.

Aufgabe 0.1 *Einführung in den* `GHCi`

Installieren Sie die Haskell-Platform (`https://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 Moodle-Vorlesungsseite (`https://moodle.tu-dortmund.de/course/view.php?id=12903`) im Abschnitt *Übungen*. 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. Alle Dateien müssen sich in dem gleichen Ordner befinden.

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 dem *base-Paket* (`https://hackage.haskell.org/package/base`) 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*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Übungsblatt 1

Ausgabe: 12.10.2018, **Abgabe:** 19.10.2018 – 16:00 Uhr, **Block:** 1

Das Übungsblatt behandelt Themen bis einschließlich Folie 24.

Aufgabe 1.1 (8 Punkte) *Produkttypen*

a) Modellieren Sie folgende Eigenschaften mit Produkttypen. Geben Sie den Typen und Attributen passende Namen. (4 Punkte)

- 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 mit Besitzer mithilfe der Attribute an. (2 Punkt)

c) Legen Sie ein weiteres Beispielkonto mit Besitzer nur mit Konstruktoren ohne Attribute an. (2 Punkt)

Lösungsvorschlag

```
-- Aufgabe 1.1 a)
data Account = Account { balance :: Int, owner :: Client }
  deriving Show
data Client = Client
  { name :: String
  , surname :: String
  , address :: String
  } deriving Show

-- Aufgabe 1.1 b)
client1 :: Client
client1 = Client
  { name = "John"
  , surname = "Doe"
  , address = "Somewhere"
  }
acc1 :: Account
acc1 = Account{balance = 0, owner = client1}

-- Aufgabe 1.1 c)
client2 :: Client
client2 = Client "Max" "Mustermann" "Musterhausen"
acc2 :: Account
acc2 = Account 100 client2
```

Aufgabe 1.2 (8 Punkte) *Summentypen*

a) Definieren Sie eine dreielementige Menge.

b) Definieren Sie ein Element `eib` vom Typ `Either Int Bool`.

c) Definieren Sie ein Element `mc` vom Typ `Maybe Color` mit einem anderen Wert als `Nothing`.

d) Definieren Sie ein Element `tup` vom Typ `(Bool,())`.

Lösungsvorschlag

```
-- Aufgabe 1.2 a)
data M = X | Y | Z

-- Aufgabe 1.2 b)
eib ::  Either Int Bool
eib = Left 4 -- oder Right True

-- Aufgabe 1.2 c)
data Color = Red | Magenta | Blue | Cyan | Green | Yellow

mc :: Maybe Color
mc = Just Cyan

-- Aufgabe 1.2 d)
tup :: (Bool,())
tup = (True,())
```

Aufgabe 1.3 (8 Punkte) *Typinferenz*
Berechnen Sie die Typen der folgenden Ausdrücke mithilfe der Typinferenzregeln. Sie dürfen die ASCII verträgliche Schreibweise von Folie 34 verwenden.

a) `(\x -> (Left 1, Just x)) 1` mit `1 :: Int`

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

Lösungsvorschlag

```
-- Aufgabe 1.3 a)
```

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{1 :: Int}}\quad \dfrac{}{\texttt{x :: Int}}}{\texttt{Left 1 :: Either Int a,}}\quad \dfrac{}{\texttt{Just x :: Maybe Int}}}{\texttt{(Left 1, Just x) :: (Either Int a,Maybe Int)}}}{\dfrac{\texttt{x :: Int,}}{\texttt{\textbackslash x -> (Left 1, Just x) :: Int -> (Either Int a,Maybe Int),}}\quad \dfrac{}{\texttt{1 :: Int}}}{\texttt{(\textbackslash x -> (Left 1, Just x)) 1 :: (Either Int a,Maybe Int)}}$$

```
-- Aufgabe 1.3 b)
```

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{f :: a -> a -> a,}}\quad \dfrac{}{\texttt{x :: a}}}{\texttt{f(x) :: a -> a,}}\quad \dfrac{}{\texttt{x :: a}}}{\texttt{(f(x))(x) :: a}}}{\texttt{f :: a -> a -> a,}}\quad \dfrac{}{\texttt{f((f(x))(x)) :: a -> a}}}{\texttt{x :: a,}}}{\dfrac{\texttt{f :: a -> a -> a,}}{\texttt{\textbackslash x -> f((f(x))(x)) :: a -> a -> a}}}{\texttt{\textbackslash f -> (\textbackslash x -> f((f(x))(x))) :: (a -> a -> a) -> a -> a -> a}}$$

technische universität dortmund

FLDIT

Lehrstuhl 1
Logik in der Informatik

Prof. Dr. Peter Padawitz

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Übungsblatt 2

Ausgabe: 19.10.2018, **Abgabe:** 26.10.2018 – 16:00 Uhr, **Block:** 1

Das Übungsblatt behandelt Themen bis einschließlich Folie 35.

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 2.1 (6 Punkte) *λ-Ausdrücke Auswerten*
Werten Sie folgende Ausdrücke *schrittweise* und *lazy (leftmost-outermost)* aus.

a) `(\x y -> y x) a b`

b) `(\y z -> z) ((\x -> x x) (\x -> x x)) a`

c) `(\f g x -> f (g x)) (\y -> y y) (\z -> a)`

Lösungsvorschlag

```
-- Aufgabe 2.1 a)
{-
(\x y -> y x) a b
  <=> (\x -> \y -> y x) a b
  ~> (\y -> y a) b
  ~> b a
-}

-- Aufgabe 2.1 b)
{-
(\y z -> z) ((\x -> x x) (\x -> x x)) a
  <=> (\y -> \z -> z) ((\x -> x x) (\x -> x x)) a
  ~> (\z -> z) a
  ~> a
-}
```

```
-- Aufgabe 2.1 c)
{-
(\f g x -> f (g x)) (\y -> y y) (\z -> a)
  <=> (\f -> \g -> \x -> f (g x)) (\y -> y y) (\z -> a)
  ~> (\g -> \x -> (\y -> y y) (g x))  (\z -> a)
  ~> \x -> (\y -> y y) ((\z -> a) x)
  ~> \x -> ((\z -> a) x) ((\z -> a) x)
  ~> \x -> a ((\z -> a) x)
  ~> \x -> a a
-}
```

Aufgabe 2.2 (4 Punkte) *Fallunterscheidungen*
Implementieren Sie folgende Funktionen in Haskell mit der `case_of`-Syntax und geben Sie die Typen der Funktionen an.

a) Benutzen Sie für folgende Funktion die *Fallunterscheidungen nach Muster*.

$$f(e) = \begin{cases} x + y, & \text{falls } e = \mathit{Left}(x, y) \text{ und } x, y \in \mathit{Float} \\ x(pt) + y(pt), & \text{falls } e = \mathit{Right}(pt) \text{ und } pt \in \mathit{Point} \end{cases}$$

b) Benutzen Sie für folgende Funktion die *Fallunterscheidungen nach Bedingung*.

$$g(x, y) = \begin{cases} x * y, & \text{falls } x \text{ gerade} \\ x - y, & \text{falls } x > 50,\ y > 100 \text{ und } x \text{ ungerade} \\ x/y, & \text{falls } y \neq 0 \text{ und } x \text{ ungerade} \\ x + y, & \text{sonst} \end{cases}$$

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

Lösungsvorschlag

```
-- Aufgabe 2.2 a)
f :: Either (Float,Float) Point -> Float
f = \e -> case e of
  Left (x,y) -> x + y
  Right pt -> x pt + y pt

-- Aufgabe 2.2 b)
g :: (Integer,Integer) -> Integer
g = \t -> case t of
  (x,y)
    | even x            -> x * y
    | x > 50, y > 100   -> x - y
    | y /= 0            -> x `div` y
    | otherwise         -> x + y
```

Aufgabe 2.3 (6 Punkte) *Präfixdarstellung*
Fügen Sie die impliziten Klammern in folgende Haskell-Ausdrücke ein. Wandeln Sie danach den Ausdruck in seine Präfixdarstellung.

a) `x + y + 5 * z`

b) `f . g $ h $ f x`

c) `f 5 True 3`

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 + y) + (5 * z)`
Präfix: `(+) ((+) x y) ((*) 5 z)`

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

c) 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: `((f 5) True) 3`
Der Ausdruck ist bereits in Präfixdarstellung.

Aufgabe 2.4 (8 Punkte) *Haskell-Funktion Auswerten*
Werten Sie den folgende Ausdrücke aus, indem Sie erst den Operator *schrittweise* in einen λ-Ausdruck umformen und dann den Ausdruck *schrittweise* auswerten.

a) `False || True` Die Disjunktion ist wie folgt definiert.

```
(||) :: Bool -> Bool -> Bool
True  || _ =  True
False || x =  x
```

b) `h . h'` Der Kompositionsoperator ist auf Folie 32 definiert.

Lösungsvorschlag

```
-- Aufgabe 2.4 a)
{- (||) schrittweise in einen Lambda-Ausdruck umformen.
True || _ = True
False || x = x
<=>
(||) True _  = True
(||) False x = x
<=>
(||) b1 b2 = case (b1,b2) of
  (True,_)  -> True
  (False,x) -> x
<=>
(||) = \b1 b2 -> case (b1,b2) of
  (True,_)  -> True
  (False,x) -> x
<=>
(||) = \b1 -> \b2 -> case (b1,b2) of
  (True,_)  -> True
  (False,x) -> x
-}

{-  False || True schrittwiese auswerten.
False || True
<=> (||) False True
~> (\b1 -> \b2 -> case (b1,b2) of
      (True,_)  -> True
      (False,x) -> x) False True
~> (\b2 -> case (False,b2) of
      (True,_)  -> True
      (False,x) -> x) True
~> case (False,True) of
      (True,_)  -> True
      (False,x) -> x
~> True
-}


--Aufgabe 2.4 b)
{- (.) schrittweise in einen Lambda-Ausdruck umformen.
(g . f) a = g (f a)
<=> (.) g f a = g (f a)
<=> (.) = \g f a -> g (f a)
<=> (.) = \g -> \f -> \a -> g (f a)
-}

{-   h . h' schrittwiese auswerten.
h . h'
<=> (.) h h'
~> (\g -> \f -> \a -> g (f a)) h h'
~> (\f -> \a -> h (f a)) h'
~> \a -> h (h' a)
-}
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 3

Ausgabe: 26.10.2018, **Abgabe:** 2.10.2018 – 16:00 Uhr, **Block:** 3

Das Übungsblatt behandelt Themen bis einschließlich Folie 45.

Aufgabe 3.1 (8 Punkte) *Endrekursion*
Formen Sie folgende Funktionen, die Schleifen enthalten, in *endrekursive* (*iterative*) Haskell-Funktionen um.

a) Eine Multiplikationsfunktion, die mit einer Addition arbeitet.

```
int mult(int x, int y) {
    int state = 0;
    while (y > 0) {
        state = state + x;
        y = y - 1;
    }
    return state;
}
```

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

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

Lösungsvorschlag

```
-- Aufgabe 3.1 a)
mult :: Int -> Int -> Int
mult x y = loop 0 y
    where
      loop state y
        | y > 0  = loop (state + x) (y - 1)
        | otherwise = state
```

```haskell
-- Aufgabe 3.1 b)
prod :: [Int] -> Int
prod ls = loop 1 ls
  where
    loop state (x:xs) = loop (state * x) xs
    loop state []     = state
```

Aufgabe 3.2 (8 Punkte) *Listenfunktionen auswerten*
Werten Sie folgende Haskell-Ausdrücke *schrittweise* und *lazy (leftmost-outermost)* aus. Sie können die Funktionen immer gleich auf alle Parameter anwenden. Daher ist es nicht nötig, die Funktionen erst in λ-Ausdrücke umzuformen.

a) `updList [3,2,8,4] 2 9` (4 Punkte)

b) `foldl (/) 100 [25,10]` Definition von `foldl` auf Folie 54. (2 Punkte)

c) `foldr (/) 100 [25,10]` Definition von `foldr` auf Folie 61. (2 Punkte)

Lösungsvorschlag

```haskell
-- Aufgabe 3.2 a)
{-
updList [3,2,8,4] 2 9
~> take 2 [3,2,8,4] ++ 9 : drop 3 [3,2,8,4]
~> 3 : take 1 [2,8,4] ++ 9 : drop 3 [3,2,8,4]
~> 3 : (take 1 [2,8,4] ++ 9 : drop 3 [3,2,8,4])
~> 3 : (2 : take 0 [8,4] ++ 9 : drop 3 [3,2,8,4])
~> 3 : 2 : (take 0 [8,4] ++ 9 : drop 3 [3,2,8,4])
~> 3 : 2 : ([] ++ 9 : drop 3 [3,2,8,4])
~> 3 : 2 : 9 : drop 3 [3,2,8,4]
~> 3 : 2 : 9 : drop 2 [2,8,4]
~> 3 : 2 : 9 : drop 1 [8,4]
~> 3 : 2 : 9 : drop 0 [4]
~> 3 : 2 : 9 : [4]
= [3,2,9,4]
-}

-- Aufgabe 3.2 b)
{-
foldl (/) 100 [25,10]
~> foldl (/) 4 [10]
~> foldl (/) 0.4 []
~> 0.4
-}

-- Aufgabe 3.2 c)
{-
foldr (/) 100 [25,10]
~> 25 / (foldr (/) 100 [10])
~> 25 / (10 / foldr (/) 100 [])
~> 25 / (10 / 100)
~> 25 / 0.1
~> 250
-}
```

Aufgabe 3.3 (8 Punkte) *Funktionen implementieren*
Implementieren Sie folgende Funktionen in Haskell. Die Funktionen basieren auf partiellen Haskell-Funktionen und sollen mithilfe des Datentyps `Maybe` absturzsicher implementiert werden. Es dürfen nur die angegebenen Hilfsfunktionen benutzt werden.

a) Die Funktion `safeDiv` soll sich ähnlich wie `div` verhalten. Anstelle eines Fehlers soll bei einer Division durch Null der Wert `Nothing` ausgeben werden. Sie dürfen `div` als Hilfsfunktion benutzen.

b) Die Funktion `safeIndex` soll sich ähnlich wie `(!!)` verhalten. Anstelle eines Fehlers soll bei einem Index außerhalb der Liste der Wert `Nothing` ausgeben werden. Sie dürfen `(>)` und `(-)` als Hilfsfunktion benutzen.

Lösungsvorschlag

```haskell
-- Aufgabe 3.3 a)
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv x y = Just (x `div` y)

-- Aufgabe 3.3 b)
safeIndex :: [a] -> Int -> Maybe a
safeIndex [] _     = Nothing
safeIndex (a:_)  0 = Just a
safeIndex (_:as) n
  | n > 0     = safeIndex as (n-1)
  | True = Nothing
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 4

Ausgabe: 2.11.2018, **Abgabe:** 9.11.2018 – 16:00 Uhr, **Block:** 2

Das Übungsblatt behandelt Themen bis einschließlich Folie 72.

Aufgabe 4.1 (4 Punkte) *Listenfunktionen implementieren*
Implementieren Sie folgende Listenfunktionen in Haskell und geben Sie die Typen der Funktionen an. Es dürfen nur die angegebenen Hilfsfunktionen benutzt werden. Die Typen sollten möglichst allgemein sein.

a) Die Funktion `shift` erhält eine Ganzzahl und eine Liste. Die Zahl gibt an, wie viele Elemente vom Anfang der Liste an das Ende angehangen werden. Die Hilfsfunktionen `(-)` und `(++)` dürfen benutzt werden.
Beispiel: `shift 2 [1,2,3,4,5,6]` ⇝ `[3,4,5,6,1,2]`

b) Die Funktion `removeLetterA` entfernt alle Vorkommen des Großbuchstaben A aus einem `String`.
Beispiel: `removeLetterA "BANANA"` ⇝ `"BNN"`

Lösungsvorschlag

```
-- Aufgabe 4.1 a)
shift :: Int -> [a] -> [a]
shift 0 as     = as
shift n (a:as) = shift (n-1) (as ++ [a])
shift _ []     = []

-- Aufgabe 4.1 b)
removeLetterA :: String -> String
removeLetterA ('A':str) = removeLetterA str
removeLetterA (c:str)   = c:removeLetterA str
removeLetterA ""        = ""
```

Aufgabe 4.2 (6 Punkte) *Funktionslifting auf Listen*
Implementieren Sie folgende Aufgaben mithilfe der Funktion `map` oder `zipWith`. Diese müssen sinnvoll eingesetzt werden.

a) `cap :: String -> String` wandelt alle Buchstaben in einem `String` in Großbuchstaben.
Hinweis: Die Funktion `toUpper` wandelt einen einzelnen Buchstaben in einen Großbuchstaben.
Beispiel: `cap "Hello, world"!` ⇝ `"HELLO, WORLD"!`

b) `lesser :: [Int] -> [Int] -> [Int]` vergleicht zwei Listen von Ganzzahlen positionsweise und übernimmt den jeweils kleineren Wert im Ergebnis.
Beispiel: `lesser [5,2,1] [4,2,2]` ⇝ `[4,2,1]`

c) `applyToOne :: [a -> b] -> a -> [b]` wendet alle Funktionen aus einer Liste auf den gleichen Wert an und speichert die einzelnen Ergebnisse in einer Liste.
Beispiel: `applyToOne [(+1),(*2),negate] 2` ⇝ `[3,4,-2]` (= `[2+1,2*2,negate 2]`)

Lösungsvorschlag

```
-- Aufgabe 4.2 a)
cap :: String -> String
cap = map toUpper

-- Aufgabe 4.2 b)
lesser :: [Int] -> [Int] -> [Int]
lesser = zipWith min

-- Aufgabe 4.2 c)
applyToOne :: [a -> b] -> a -> [b]
applyToOne fs a = map ($a) fs
```

Aufgabe 4.3 (4 Punkte) *Listenfaltung auswerten*
Werten Sie folgende Haskell-Ausdrücke *schrittweise* und *lazy (leftmost-outermost)* aus.

a) `foldl (/) 20 [5,4]`

b) `foldr (/) 20 [5,4]`

Lösungsvorschlag

```
-- Aufgabe 4.3 a)
{-
foldl (/) 20 [5,4]
~>  foldl (/) (20 / 5) [4]
~>  foldl (/) 4 [4]
~>  foldl (/) (4 / 4) []
~>  foldl (/) 1 []
~> 1
-}

-- Aufgabe 4.3 b)
{-
foldr (/) 20 [5,4]
~> 5 / foldr (/) 20 [4]
~> 5 / (4 / foldr (/) 20 [])
~> 5 / (4 / 20)
~> 5 / 0.2
~> 25
-}
```

Aufgabe 4.4 (4 Punkte) *Listenfaltung implementieren*
Implementieren Sie folgende Aufgaben mithilfe der Listenfaltungen `foldl` oder `foldr`. Diese müssen sinnvoll eingesetzt werden.

a) `countNothing :: [Maybe a] -> Int` zählt alle Vorkommen von `Nothing` in einer Liste.
Beispiel: `countNothing [Nothing, Just 5, Nothing]` $\rightsquigarrow$ `2`

b) `lefts :: [Either a b] -> [a]` entfernt alle `Right`-Werte und gibt eine Liste aller `Left`-Werte aus.
Beispiel: `lefts [Left 3, Right False, Right True, Left 5]` $\rightsquigarrow$ `[3,5]`

Lösungsvorschlag

```
-- Aufgabe 4.4 a)
countNothing :: [Maybe a] -> Int
countNothing = foldl f 0 where
  f state Nothing = state + 1
  f state _       = state

-- Aufgabe 4.4 b)
lefts :: [Either a b] -> [a]
lefts = foldr f [] where
  f (Left a) as = a:as
  f _        as = as
```

Aufgabe 4.5 (6 Punkte) *Listenkomprehension*
Definieren Sie folgende Funktionen mithilfe der Listenkomprehension.

a) `squares :: [(Int,Int)]` Liste jeder geraden Zahl von 0 bis 10 und ihr Quadrat, also `squares == [(0,0),(2,4),(4,16),(6,36),(8,64),(10,100)]`.

b) `solutions :: [(Int, Int, Int)]` enthält Tripel $(x, y, z) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, welche die Gleichung $5x + 3y^2 + 10 == z$ lösen. Nehmen Sie für x, y und z nur Werte von 0 bis 100.

c) `codes :: [[(Char,Int)]]` gibt alle Lösungen für das Kryptogramm $eins + vier = fuenf$.

Lösungsvorschlag

```
-- Aufgabe 4.5 a)
squares :: [(Int,Int)]
squares = [(val,val^2) | val <- [0..10], even val]

-- Aufgabe 4.5 b)
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z) | x <- [0..100], y <- [0..100], z <- [0..100]
            ,  5*x + 3*y^2 + 10 == z ]

-- Aufgabe 4.5 c)
codes :: [[(Char,Int)]]
codes = [ zip code [0..]
  | code <- perms "einsvrfu01"
  , let [e,i,n,s,v,r,f,u] = map (getIndex code) "einsvrfu"
  , 1000*(e+v)+100*(i+i)+10*(n+e)+(s+r)
    == 10000*f+1000*u+100*e+10*n+f
  ]
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 5

Ausgabe: 9.11.2018, **Abgabe:** 16.11.2018 – 16:00 Uhr, **Block:** 2

Das Übungsblatt behandelt Themen bis einschließlich Folie 83.

Aufgabe 5.1 (12 Punkte) *Unendliche Listen auswerten*
Werten Sie folgende Haskell-Ausdrücke *schrittweise* und *lazy (leftmost-outermost)* aus.

a) `take 2 $ nats 3`

b) `iterate tail [3,4,9,8] !! 2`

c) `fibs !! 2`

Hinweis: Aufgabe 5.1 lässt sich mit Unterstützung der `Painter.hs` lösen. Sie benötigen dazu die neuste Version des Painter-Pakets. Laden Sie das Paket dazu neu von Moodle. Dem Übungsblatt liegt eine Beispieldatei für Aufgabe 5.1 b) bei (`Aufgabe5-1b`). Speichern Sie diese Datei im gleichen Ordner wie das Painter-Paket. Laden Sie dann das Painter-Modul in den GHCi.

```
$ ghci Painter.hs
```

Danach führen Sie die Funktion reduce auf der Datei aus.

```
*Painter> reduce "Aufgabe5-1b" []
```

Dies erzeugt eine Datei im Ordner `Pix` mit dem Namen `Aufgabe5-1bReduction.html`. Die einzelnen Auswertungsschritte lassen sich im Browser anzeigen und können dann mit der eigenen Lösung verglichen werden. Sie können sich selbst Dateien für den Painter schreiben. Eine detaillierte Anleitung finden Sie in der Datei `Painter.pdf` im Painter-Paket. Orientieren Sie sich am Beispiel `Aufgabe5-1b`. Die erste Zeile ist der zu reduzierende Ausdruck und wird mit doppeltem Semikolon beendet. In den anderen Zeilen werden Funktionsgleichungen definiert und werden mit einfachem Semikolon beendet. Die HTML-Datei erzeugen Sie mit: `reduce "dateiname" []` Die Datei heißt dann `Pix/dateinameReduction.html`.

Lösungsvorschlag

```
-- Aufgabe 5.1 a)
{-
take 2 $ nats 3
~> take 2 (nats 3)
~> take 2 (3:map (+1) (nats 3))
~> 3:take 1 (map (+1) (nats 3))
~> 3:take 1 (map (+1) (3:map (+1) nats 3))
~> 3:take 1 (4:map (+1) (map (+1) nats 3))
~> 3:4:take 0 (map (+1) (map (+1) nats 3))
~> 3:4:[]
```

```
-}

-- Aufgabe 5.1 b)
{-
iterate tail [3,4,9,8] !! 2
~> ([3,4,9,8]:iterate tail (tail [3,4,9,8])) !! 2
~> (iterate tail (tail [3,4,9,8])) !! 1
~> ((tail [3,4,9,8]):iterate tail (tail (tail [3,4,9,8]))) !! 1
~> (iterate tail (tail (tail [3,4,9,8]))) !! 0
~> ((tail (tail [3,4,9,8])):iterate tail (tail (tail (tail [3,4,9,8])))) !!
~> tail (tail [3,4,9,8])
~> tail [4,9,8]
~> [9,8]
-}
```

```
-- Aufgabe 5.1 c)
{-
fibs !! 2
~> (1:1:zipWith (+) fibs $ tail fibs) !! 2
~> (1:zipWith (+) fibs $ tail fibs) !! 1
~> (zipWith (+) fibs $ tail fibs) !! 0
~> (zipWith (+) fibs (tail fibs)) !! 0
~> (zipWith (+) (1:1:zipWith (+) fibs $ tail fibs) (tail fibs)) !! 0
~> (zipWith (+) (1:1:zipWith (+) fibs $ tail fibs)
    (tail (1:1:zipWith (+) fibs $ tail fibs))) !! 0
~> (zipWith (+) (1:1:zipWith (+) fibs $ tail fibs)
    (1:zipWith (+) fibs $ tail fibs)) !! 0
~> ((1+1):zipWith (+) (1:zipWith (+) fibs $ tail fibs)
    (zipWith (+) fibs $ tail fibs)) !! 0
~> 1+1
~> 2
-}
```

Aufgabe 5.2 (6 Punkte) *Unendliche Listen implementieren*
Implementieren Sie folgende unendliche Listen. Die Listen dürfen keine Endlosschleifen mit `take` oder `(!!)` erzeugen.

a) `odds :: [Int]` – Liste aller ungeraden Zahlen.
   Beispiel: `take 10 odds` $\rightsquigarrow$ `[1,3,5,7,9,11,13,15,17,19]`

b) `alternate :: [Int]` – Liste aller Ganzzahlen ohne Null als alternierender Reihe. Die Liste soll mit 1 starten.
   Beispiel: `take 10 alternate` $\rightsquigarrow$ `[1,-1,2,-2,3,-3,4,-4,5,-5]`

c) `solutions :: [(Int, Int, Int)]` – Liste *aller* Tripel $(x, y, z) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, welche die Gleichung $5x + 3y^2 + 10 == z$ lösen. Nehmen Sie für x, y und z nur positive Werte.
   Beispiel: `(25,4,183) ``elem`` solutions` $\rightsquigarrow$ `True`

Lösungsvorschlag

```
-- Aufgabe 5.2 a)
odds :: [Int]
odds = filter odd [0..]

-- Aufgabe 5.2 b)
alternate :: [Int]
alternate = concatMap (\x -> [x,-x]) [1..]

-- Aufgabe 5.2 c)
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z) | z <- [0..] , x <- [0..z] , y <- [0..z]
            , 5*x + 3*y^2 + 10 == z ]
```

Aufgabe 5.3 (6 Punkte) *Modellierung*
Gegeben seien folgende Datentypen:

```
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
```

Definieren Sie folgende Funktionen. Fehlerbehandlungen sind nicht notwendig.

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

```
-- Aufgabe 5.3 a)
credit :: Int -> ID -> Bank -> Bank
credit amount accountID bank
  = updRel bank accountID entry{ balance = oldBalance + amount}
  where
    Just entry = lookup accountID bank
    oldBalance = balance entry

-- Aufgabe 5.3 b)
debit :: Int -> ID -> Bank -> Bank
debit amount = credit (-amount)

-- Aufgabe 5.3 c)
transfer :: Int -> ID -> ID -> Bank -> Bank
transfer amount id1 id2 = debit amount id1 . credit amount id2
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 6

Ausgabe: 16.11.2018, **Abgabe:** 23.11.2018 – 16:00 Uhr, **Block:** 2

Aufgabe 6.1 (12 Punkte) *Zahlen als Datentypelemente*
Definieren Sie folgende Konstanten und Datentypen in Haskell mithilfe der in der Vorlesung vorgestellten rekursiven Datentypen `Nat`, `Int'` und `PosNat`.

a) Definieren Sie eine Konstante *drei* = 3 für den Datentyp `Nat`.

b) Definieren Sie eine Konstante *zwei* = 2 für den Datentyp `PosNat`.

c) Definieren Sie eine Konstante *mzwei* = −2 für den Datentyp `Int'`.

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

e) Definieren Sie eine Konstante $c = \frac{1}{3}$ für den Datentyp `Rat`.

f) Definieren Sie eine Konstante $c' = -2$ für den Datentyp `Rat`.

Lösungsvorschlag

```
-- Aufgabe 6.1 a)
drei :: Nat
drei = Succ $ Succ $ Succ Zero

-- Aufgabe 6.1 b)
zwei :: PosNat
zwei = Succ' One

-- Aufgabe 6.1 c)
mzwei :: Int'
mzwei = Minus (Succ' One)

-- Aufgabe 6.1 d)
data Rat
    = Int' :/ PosNat
    deriving Show

-- Aufgabe 6.1 e)
c :: Rat
c = Plus One :/ Succ' (Succ' One)

-- Aufgabe 6.1 f)
c' :: Rat
c' = Minus (Succ' One) :/ One
```

Aufgabe 6.2 (12 Punkte) *Rekursive Datentypen*
Definieren Sie folgende Haskell-Funktionen.

a) `natTake :: Nat -> [a] -> [a]`, wie `take` für den Datentyp `Nat`.

b) `natHoch :: (a -> a) -> Nat -> a -> a`, wie `hoch` (Folie 40) für den Datentyp `Nat`.

c) `colistConc :: Colist a -> Colist a -> Colist a`, wie `(++)` für `Colist a`.

d) `colistReverse :: Colist a -> Colist a`, wie `reverse` für `Colist a`.
Hinweis: Es empfiehlt sich die iterative Variante.

e) `stTakeWhile :: (a -> Bool) -> Stream a -> [a]`, wie `takeWhile` für `Stream a`.

f) `stZipWith :: (a -> b -> c) -> Stream a -> Stream b -> Stream c`,
wie `zipWith` für `Stream a`.

Lösungsvorschlag

```
-- Aufgabe 6.2 a)
natTake :: Nat -> [a] -> [a]
natTake Zero     _      = []
natTake (Succ n) (a:as) = a : natTake n as
natTake _        []     = []

-- Aufgabe 6.2 b)
natHoch :: (a -> a) -> Nat -> a -> a
f `natHoch` Zero     = id
f `natHoch` (Succ n) = f . (f `natHoch` n)

-- Aufgabe 6.2 c)
colistConc :: Colist a -> Colist a -> Colist a
colistConc s s' = case split s of
  Just (a,as) -> Colist (Just (a,colistConc as s'))
  Nothing     -> s'

-- Aufgabe 6.2 d)
colistReverse :: Colist a -> Colist a
colistReverse = loop nil where
  loop state s = case split s of
    Just (a,as) -> loop (Colist (Just (a,state))) as
    Nothing     -> state

-- Aufgabe 6.2 e)
stTakeWhile :: (a -> Bool) -> Stream a -> [a]
stTakeWhile p st
  | p a       = a : stTakeWhile p (tl st)
  | otherwise = []
  where a = hd st

-- Aufgabe 6.2 f)
stZipWith :: (a -> b -> c) -> Stream a -> Stream b -> Stream c
stZipWith f s s' = f (hd s) (hd s') :< stZipWith f (tl s) (tl s')
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 7

Ausgabe: 23.11.2018, **Abgabe:** 30.11.2018 – 16:00 Uhr, **Block:** 3

Aufgabe 7.1 (6 Punkte) *Arithmetische Ausdrücke*

a) Definieren Sie eine Konstante `expr :: Exp String` für den Ausdruck $5x + 3y^2 + 10$. Stellen Sie den Ausdruck als Elemente vom Typ `Exp String` da. (2 Punkte)

b) Schreiben Sie die Listenkomprehension `solutions :: [(Int,Int,Int)]` um. Machen Sie sinvollen gebrauch von dem Ausdruck `expr` und der Funktion `foldArith evalAlg`. (4 Punkte)

Lösungsvorschlag

```
-- Aufgabe 1 a)
expr :: Exp String
expr = Sum [5 :* Var "x", 3 :* Var "y" :^ 2, Con 10]

-- Aufgabe 1 b)
solutions :: [(Int,Int,Int)]
solutions = [ (x,y,z) | z <- [0..] , x <- [0..z] , y <- [0..z]
            , let st "x" = x
                  st "y" = y
            , foldArith evalAlg expr st == z ]
```

Aufgabe 7.2 (6 Punkte) *Boolesche Ausdrücke*
Implementieren Sie folgende boolesche Ausdrücke als Haskell-Konstanten vom Typ `BExp String`.

a) `bexpr1` $= b \vee \neg b$

b) `bexpr2` $= (x \vee \mathit{false}) \wedge y$

c) `bexpr3` $= b \wedge (x \leq x + 10)$

Lösungsvorschlag

```
-- Aufgabe 2 a)
bexpr1 :: BExp String
bexpr1 = Or [BVar "b", Not (BVar "b")]

-- Aufgabe 2 b)
bexpr2 :: BExp String
bexpr2 = And [Or [BVar "x", False_], BVar "y"]

-- Aufgabe 2 c)
bexpr3 :: BExp String
bexpr3 = And [BVar "b", Var "x" :<= Sum [Var "x", Con 10]]
```

Aufgabe 7.3 (6 Punkte) *Abstrakte Datentypen, Algebren und Faltungen*
Die Funktion `toInt` soll einen Wert vom Typ `PosNat` den entsprechenden Wert vom Typ `Int` wandeln. Dabei soll die Funktion Gebrauch von einer Faltung machen.

```
data PosNat = One | Succ' PosNat deriving Show

toInt :: PosNat -> Int
toInt = foldPosNat intAlg
```

Implementieren Sie den abstrakten Datentypen, die Algebra und die Faltungen. Gehen Sie in folgender Reihenfolge vor.

a) Implementieren Sie einen abstrakte Datentypen (eine Signatur) `PosNatSig` für den Datentypen `PosNat`.

b) Implementieren Sie eine Faltung `foldPosNat`, welche mithilfe einer Algebra Werte vom Typ `PosNat` auswertet (faltet).

c) Implementieren Sie die Algebra `intAlg` vom Typ `PosNatSig`, die einen Wert vom Typ `PosNat` den entsprechenden Wert vom Typ `Int` wandeln.

Lösungsvorschlag

```
-- Aufgabe 3 a)
data PosNatSig posnat = PosNatSig
  { pnOne :: posnat
  , pnSucc' :: posnat -> posnat
  }

-- Aufgabe 3 b)
foldPosNat :: PosNatSig posnat -> PosNat -> posnat
foldPosNat alg One       = pnOne alg
foldPosNat alg (Succ' n) = pnSucc' alg $ foldPosNat alg n

-- Aufgabe 3 c)
intAlg :: PosNatSig Int
intAlg = PosNatSig
  { pnOne = 1
  , pnSucc' = (+1)
  }
```

Aufgabe 7.4 (6 Punkte) *Boolesche Algebra*
Es sind die Typen und Funktionen `BStore`, `BExpSig`, `foldBExp` und `evalB` gegeben. Die genaue Definition finden Sie in der Vorgabedatei `Blatt07.hs`.

Definieren Sie eine boolesche Algebra

```
evalBAlg :: BExpSig x (Store x -> Int) (Store x -> BStore x -> Bool).
```

Orientieren Sie sich an der Definition der arithmetischen Algebra `evalAlg` (Folie 103). Bei korrekter Definition verhält sich die Funktion `evalB` ähnlich wie `eval` (Folie 104), bzw. `foldArith evalAlg`. Anstelle arithmetischer Ausdrücke werden aber boolesche Ausdrücke vom Typ `BExp x` ausgewertet. Die Funktion `evalB` benötigt zwei Variablenbelegungen. Eine für boolesche Variablen und die andere für arithmetische Variablen.
Beispiel: `evalB (\"x" -> 5) (\"b" -> True) bexpr3`

Lösungsvorschlag

```
evalBAlg :: BExpSig x (Store x -> Int) (Store x -> BStore x -> Bool)
evalBAlg = BExpSig
  { true = \_ _ -> True
  , false = \_ _ -> False
  , bvar = \x _ bst -> bst x
  , bor = \bs st bst -> or $ map (\b -> b st bst) bs
  , band = \bs st bst -> and $ map (\b -> b st bst) bs
  , bnot = \b st -> not . b st
  , eq = \e1 e2 st _ -> e1 st == e2 st
  , leq = \e1 e2 st _ -> e1 st <= e2 st
  }
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 8

Ausgabe: 30.11.2018, **Abgabe:** 7.12.2018 – 16:00 Uhr, **Block:** 3

Aufgabe 8.1 (6 Punkte) *Typklassen*
Definieren und Instanziieren Sie eine eigene Typklasse.

a) Schreiben Sie eine Klasse für eine überladene Funktion `genDrop`. Diese soll sich wie `drop` verhalten, aber nicht auf den Zahlentyp `Int` beschränkt sein. (2 Punkte)

b) Instanziieren Sie die Klasse für `Int`, `Nat`, `PosNat` und `Int'`. (4 Punkte)

Lösungsvorschlag

```
-- Aufgabe 7.1 a)
class GenDrop n where
  genDrop :: n -> [a] -> [a]

-- Aufgabe 7.1 b)
instance GenDrop Int where
  genDrop = drop

instance GenDrop Nat where
  genDrop Zero     s      = s
  genDrop (Succ n) (_:s)  = genDrop n s
  genDrop _        []     = []

instance GenDrop PosNat where
  genDrop One (_:s)       = s
  genDrop (Succ' n) (_:s) = genDrop n s
  genDrop _          []   = []

instance GenDrop Int' where
  genDrop Zero'    s  = s
  genDrop (Plus n) s  = genDrop n s
  genDrop _        [] = []
```

Aufgabe 8.2 (12 Punkte) *Typklassen*
Instanziieren Sie mehrere Typklassen für den Datentyp `Int'`.

a) Schreiben Sie eine Instanz der Klasse `Enum`. Es ist ausreichend die Funktionen `toEnum` und `fromEnum` zu definieren. (6 Punkte)

   Beispiel: `map fromEnum [Minus One .. Plus (Succ' One)]` ⇝ `[-1,0,1,2]`

b) Schreiben Sie Instanzen für die Klassen `Num`, `Eq`, `Ord` und `Show`. Die Klassenfunktionen sollen sich wie für den Typ `Int` verhalten. (4 Punkte)

   Beispiel: `show $ Minus $ Succ' One` ⇝ `"-2"`

   Hinweis: Bereits definierte Funktionen dürfen wiederverwendet werden. Außerdem kann die Funktion `fromIntegral` genutzt werden, um Werte vom Typ `Integer` in den Typ `Int` zu wandeln.

c) Ändern Sie den Typ der Liste `solutions` von Übungsblatt 5 in `[(Int',Int',Int')]` und passen Sie Ihre Lösung entsprechend an. Sie dürfen nur notwendige Änderungen vornehmen. (2 Punkte)

Lösungsvorschlag

```
-- Aufgabe 7.2 a)
instance Enum Int' where
  toEnum 0 = Zero'
  toEnum i
    | i > 0 = Plus $ f i
    | otherwise = Minus $ f (-i)
    where
      f 1 = One
      f n = Succ' (f (n-1))
  fromEnum i = case i of
    Zero' -> 0
    Plus pn -> f pn
    Minus pn -> -(f pn)
    where
      f One = 1
      f (Succ' n) = f n + 1

-- Aufgabe 7.2 b)
instance Num Int' where
  fromInteger = toEnum . fromIntegral
  negate (Plus n) = Minus n
  negate (Minus n) = Plus n
  negate Zero' = Zero'
  signum (Plus _) = Plus One
  signum (Minus _) = Minus One
  signum Zero' = Zero'
  abs (Minus n) = Plus n
  abs i         = i
  i1 + i2 = toEnum $ fromEnum i1 + fromEnum i2
  i1 * i2 = toEnum $ fromEnum i1 * fromEnum i2

instance Eq Int' where
  x == y = fromEnum x == fromEnum y
```

```
instance Ord Int' where
  x <= y = fromEnum x <= fromEnum y

instance Show Int' where
  show = show . fromEnum

-- Aufgabe 7.2 c)
solutions :: [(Int', Int', Int')]
solutions = [ (x,y,z) | z <- [0..] , x <- [0..z] , y <- [0..z]
            , 5*x + 3*y^2 + 10 == z ]
```

Aufgabe 8.3 (6 Punkte) *Binäre Bäume*
Definieren Sie folgende Funktionen über binäre Bäume.

a) `sizeBintree :: Bintree a -> Int` – Gibt die Anzahl der Knoten in einem binären Baum wieder.

   Beispiel: `sizeBintree btree1` ⇝ `6`

b) `zipBintree :: Bintree a -> Bintree b -> Bintree (a,b)` – Ähnlich wie `zip`, nur auf binären Bäumen. Es werden die Werte mit gleicher Knotenposition zu einem Tupel im Ergebnisbaum zusammengefasst. Gibt es eine Knotenposition nur in einem Baum, entfällt der Knoten im Ergebnis.

   Beispiel: `zipBintree btree1 btree4`
   ⇝ `(6,121)((7,106)((11,99)((55,55),(33,33)),),(9,9))`

c) `getSubbintree :: Bintree a -> Node -> Maybe (Bintree a)` – Gibt den Teilbaum zu einem Knoten aus. Existiert der angegebene Knoten nicht, wird `Nothing` ausgegeben.

   Beispiel: `getSubbintree btree1 [Links,Links]` ⇝ `Just 11(55,33)`

Lösungsvorschlag

```
-- Aufgabe 7.3 a)
sizeBintree :: Bintree a -> Int
sizeBintree (Fork _ l r) = sizeBintree l + sizeBintree r + 1
sizeBintree Empty        = 0

-- Aufgabe 7.3 b)
zipBintree :: Bintree a -> Bintree b -> Bintree (a,b)
zipBintree (Fork a1 l1 r1) (Fork a2 l2 r2)
  = Fork (a1,a2) (zipBintree l1 l2) (zipBintree r1 r2)
zipBintree _ _ = Empty

-- Aufgabe 7.3 c)
getSubbintree :: Bintree a -> Node -> Maybe (Bintree a)
getSubbintree t [] = Just t
getSubbintree (Fork _ l _) (Links:nodes)  = getSubbintree l nodes
getSubbintree (Fork _ _ r) (Rechts:nodes) = getSubbintree r nodes
getSubbintree Empty _ = Nothing
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 9

Ausgabe: 7.12.2018, **Abgabe:** 14.12.2018 – 16:00 Uhr, **Block:** 3

Aufgabe 9.1 (10 Punkte) *Ausgabe*
Instanziieren Sie den Datentyp `BExp` für die Klasse `Show`. Beachten Sie Präzedenzen. Definieren Sie dazu die Funktion `showsPrec`. Orientieren Sie sich dabei an der entsprechenden Instanz für den Datentyp `Exp`. Benutzen Sie für die Ausgabe der Konstruktoren folgende Zeichenketten und Präzedenzen.

Konstruktor	Zeichen	Präzedenz
`True_`	`true`	—
`False_`	`false`	—
`BVar`	Name als String, z. B.: `"x"`	—
`Or`	`|`	2
`And`	`&`	3
`Not`	`!`	10 (Präfix)
`:=`	`=`	4
`:<=`	`<=`	4

Beispiele:

```
bexpr1 ⇝ "b"|!"b"
bexpr2 ⇝ ("x"|false)&"y"
bexpr3 ⇝ "b"&"x"<="x"+10
```

Lösungsvorschlag

```
-- Aufgabe 9.1
instance Show x => Show (BExp x) where
  showsPrec _ True_ = showString "true"
  showsPrec _ False_ = showString "false"
  showsPrec _ (BVar x) = shows x
  showsPrec p (Or bs) = showParen (p > 2) $ showMore 2 '|' bs
  showsPrec p (And bs) = showParen (p > 3) $ showMore 3 '&' bs
  showsPrec _ (Not b) = showChar '!' . showsPrec 11 b
  showsPrec p (e1 := e2) = showParen (p > 4)
    $ showsPrec 5 e1 . showChar '=' . showsPrec 5 e2
  showsPrec p (e1 :<= e2) = showParen (p > 4)
    $ showsPrec 5 e1 . showString "<=" . showsPrec 5 e2
```

Aufgabe 9.2 (4 Punkte) *Bäume mit beliebigem Ausgrad*
Definieren Sie folgende Funktionen für Bäume mit beliebigem Ausgrad (`Tree`) *ohne* Faltung. Sie dürfen die Funktion `foldTree` nicht benutzen.

a) `productA :: Tree Int -> Int` – Produkt aller Zahlen in einem Baum.
   Beispiel: `productA tree1` $\rightsquigarrow$ `-1440`

b) `preorderT :: Tree a -> [a]` – Wie `preorder` (Folie 140) ohne Faltung.
   Beispiel: `preorderT tree1` $\rightsquigarrow$ `[1,2,2,3,-1,-2,4,-3,5]`

Lösungsvorschlag

```
-- Aufgabe 9.2 a)
productA :: Tree Int -> Int
productA (V a) = a
productA (F a ts) = a * product (map productA ts)

--  Aufgabe 9.2 b)
preorderT :: Tree a -> [a]
preorderT (V a) = [a]
preorderT (F a ts) = a : concatMap preorderT ts
```

Aufgabe 9.3 (4 Punkte) *Baumfaltungen*
Definieren Sie folgende Funktionen als Baumfaltungen (`foldBin` bzw. `foldTree`).

a) `sizeBintree :: Bintree a -> Int` – Gibt die Anzahl der Knoten in einem binären Baum (`Bintree`) wieder.
   Beispiel: `sizeBintree btree1` $\rightsquigarrow$ `6`

b) `labelA :: Tree a -> Node -> a` – Wie `label` (Folie 136) mit Faltung.
   Beispiel: `label tree1 [0,0,1]` $\rightsquigarrow$ `-1`

Lösungsvorschlag

```
-- Aufgabe 9.3 a)
sizeBintree :: Bintree a -> Int
sizeBintree = foldBin $ BinSig 0 (\_ l r -> l + r + 1)

-- Aufgabe 9.3 b)
labelA :: Tree a -> Node -> a
labelA = foldTree $ TreeSig
  { var_ = \a [] -> a
  , fun = \_ ts (i:node) -> (ts!!i) node
  }
```

Aufgabe 9.4 (6 Punkte) *Arithmetische Ausdrücke kompilieren*
Geben Sie die Kommandosequenz für die Auswertung `execute (foldArith codeAlg 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 = Sum [5 :* Var "x", 3 :* (Var "y" :^ 2), Con 10]

vars :: Store String
vars "x" = 7
vars "y" = 5
```

Sie können mit `fst` den finalen Stapelinhalt und damit das Ergebnis der Ausführung erhalten.
`fst (execute (exp2code expr) ([],vars))` $\rightsquigarrow$ `[120]`

Lösungsvorschlag

```
Kommando | Stapel
---------|-----------
Push 5   | [5]
Load "x" | [7,5]
Mul 2    | [35]
Push 3   | [3,35]
Load "y" | [5,3,35]
Push 2   | [2,5,3,35]
Up       | [25,3,35]
Mul 2    | [75,35]
Push 10  | [10,75,35]
Add 3    | [120]
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 10

Ausgabe: 14.12.2018, **Abgabe:** 21.12.2018 – 16:00 Uhr, **Block:** 4

Das Übungsblatt behandelt Themen bis einschließlich Folie 175.

Aufgabe 10.1 (12 Punkte) *Fixpunkte*
Gegeben sei folgender Datentyp für den Restklassenring $\mathbb{Z}_{10}$:

```
data Mod10 = Z0 | Z1 | Z2 | Z3 | Z4 | Z5 | Z6 | Z7 | Z8 | Z9
```

Der Datentyp ist instanziiert für die Klassen `Show`, `Eq`, `Ord`, `Read`, `Num`, `Enum` und `Bounded`. Durch die Instanzen der Klassen `Ord` und `Bounded` wird `Mod10` ein Verband und ist damit sowohl ein CPO als auch ein co-CPO.

a) Implementieren Sie die sowohl stetige und als auch co-stetige Funktion f in Haskell.

$$f : \mathbb{Z}_{10} \to \mathbb{Z}_{10}$$
$$f(x) = \begin{cases} x+1 & \text{falls } x < 5 \\ x-1 & \text{falls } x > 7 \\ x & \text{sonst} \end{cases}$$

Berechnen Sie den kleinsten Fixpunkt (`lfp`) und den größten Fixpunkt (`gfp`) von f mithilfe der Funktion `fixpt` (Folie 162).

b) Berechnen Sie `odds :: [Mod10]` mithilfe der Funktion `fixpt` als kleinste Lösung der Gleichung

$$odds = \{1\} \cup \{succ(succ(x)) \mid x \in odds\}\,.$$

Interpretieren Sie `[Mod10]` als den Potenzmengenverband $\mathcal{P}(\mathbb{Z}_{10})$ (Folie 166). Damit ist `odd` eine Teilmenge von $\mathbb{Z}_{10}$ (`odds` $\in \mathcal{P}(\mathbb{Z}_{10})$). 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

```
-- Aufgabe 10.1 a)
f :: Mod10 -> Mod10
f x
  | x < 5 = x + 1
  | x > 7 = x - 1
  | otherwise = x

lfp :: Mod10
lfp = fixpt (<=) f 0
```

```
gfp :: Mod10
gfp = fixpt (>=) f 9


-- Aufgabe 10.1 b)
odds :: [Mod10]
odds = fixpt subset phi [] where
  phi :: [Mod10] -> [Mod10]
  phi m = [1] `union` [succ (succ x) | x <- m]
```

Aufgabe 10.2 (6 Punkte) *Semantik rekursiver Gleichungen*
Die Haskell-Funktion

```
fix :: (a -> a) -> a
fix f = f (fix f)
```

generiert den kleinsten Fixpunkt zu einer Funktion `f`, indem es die Funktion unendlich häufig auf sich selbst anwendet.

a) Zeigen Sie, dass die Rekursionsgleichung `length` eine Funktion definiert. Dazu wird eine Schrittfunktion Φ benötigt (analog zu Folie 163). Mithilfe der Funktion `fix` lässt sich dann der kleinste Fixpunkt für die Schrittfunktion generieren. Definieren Sie in Haskell die Schrittfunktion `phi` so, dass sich

```
lengthF :: [a] -> Int
lengthF = fix phi
```

wie die Funktion `length` verhält. (2 Punkte)

b) Beweisen Sie durch strukturelle Induktion, dass lfp(Φ) (`fix phi`) keine endliche Liste auf $\perp$ abbildet. (4 Punkte)

Lösungsvorschlag

a) Funktionsdefinition:

```
lengthF :: [a] -> Int
lengthF = fix phi where
  phi :: ([a] -> Int) -> ([a] -> Int)
  phi _ []     = 0
  phi f (_:as) = f as + 1
```

b) *Beweis.* Es ist zu Zeigen, dass für ein beliebigen Typ `A` und eine beliebige Liste `ls` $\in$ `[A]` `fix phi ls` $\neq \perp$ gilt.

1. Fall: `ls` = `[]`

```
    fix phi []
↝   phi (fix phi) []
↝   0
```

$0 \neq \perp$

2. Fall: `ls = (a:as)` mit `a` $\in$ `A` und `as` $\in$ `[A]` beliebig.

```
    fix phi (a:as)
↝   phi (fix phi) (a:as)
↝   fix phi as + 1
```

Unter der Annahme, dass `fix phi as` $\neq \perp$ gilt (Induktionsvorraussetzung), gilt auch `fix phi as + 1` $\neq \perp$ und damit `fix phi (a:as)`.

□

Aufgabe 10.3 (6 Punkte) *Graphen*
Definieren Sie folgende Haskell-Funktionen.

a) `reverseGraph :: Eq a => Graph a -> Graph a` – Dreht alle Kanten in einem Graph um.
Beispiel: `reverseGraph graph1` ↝

```
2 -> [1,6]
3 -> [1,5]
1 -> [3,4]
4 -> [3,6]
6 -> [3]
5 -> [5,6]
```

b) `breadthFirst :: Eq a => a -> Graph a -> [a]` – Ähnlich wie bei `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 Breitensuche aus.
Beispiel: `breadthFirst 1 graph1` ↝ `[1,2,3,4,6,5]`

c) `isReachableFrom :: Eq a => a -> a -> Graph a -> Bool` – Gibt aus, ob der erste Knoten von dem zweiten Knoten aus erreichbar ist.
Beispiele:
`(6 `isReachableFrom` 4) graph1` ↝ `True`
`(6 `isReachableFrom` 2) graph1` ↝ `False`

Lösungsvorschlag

```
-- Aufgabe 10.3 a)
reverseGraph :: Eq a => Graph a -> Graph a
reverseGraph = rel2Graph . map swap . graph2Rel

-- Aufgabe 10.3 b)
breadthFirst :: Eq a => a -> Graph a -> [a]
breadthFirst start (G _ adj) = bf [start] [] where
  bf (a:as) visited
    | a `elem` visited = bf as visited
    | otherwise = bf (as ++ adj a) (visited ++ [a])
  bf [] visited = visited

-- Aufgabe 10.3 c)
isReachableFrom :: Eq a => a -> a -> Graph a -> Bool
(goal `isReachableFrom` start) graph = goal `elem` closure start where
  G _ closure = closureW graph
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 11

Ausgabe: 21.12.2018, **Abgabe:** 11.1.2019 – 16:00 Uhr, **Block:** 4

Das Übungsblatt behandelt Themen bis einschließlich Folie 195.

Aufgabe 11.1 (8 Punkte) *Kinds*
Bestimmen Sie den Kind der angegebenen Typkonstruktoren.

a) `data Wrap f = Wrap (f Maybe)` – Bestimmen Sie den Kind von `f`.

b) `data T a f = C (f a)` – Bestimmen Sie den Kind von `T`.

c) `class Cl c where fun :: c a b -> c b a`

Bestimmen Sie den Kind von `c`.

d)
```
class Tf c where
  type F c :: * -> *
  tffun :: g c -> f (F c) g -> a
```

Bestimmen Sie den Kind von `f`.

Lösungsvorschlag

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

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

c) `c :: * -> * -> *`

d) `f :: (* -> *) -> (* -> *) -> *`

Aufgabe 11.2 (8 Punkte) *Typfamilien*
Gegeben sei folgende Typklasse:

```
class FromList a where
  type Item a :: *
  fromList :: [Item a] -> a
```

Die Funktion `fromList` wandelt eine Liste in den Zieltyp um. Überladen Sie die Funktion für die angegebenen Typen.

a) `Colist a` – Gibt die zugehörige Coliste aus.
   Beispiel: `fromList [1,2,3] :: Colist Int` $\leadsto$ `Colist [1,2,3]`

b) `Maybe a` – Gibt das erste Element aus oder `Nothing`, falls die Liste leer ist.
   Beispiel: `fromList [1,2,3] :: Maybe Int` $\leadsto$ `Just 1`

c) `newtype Map a b = Map [(a,b)]` – Ein Datentyp für eine Assoziationsliste (siehe Folie 53). Die Funktion `fromList` entfernt dabei doppelte Einträge für den Typ `a`. Es wird nur das letzte Vorkommen im Ergebnis übernommen.
   Beispiel: `fromList [('X',1),('Y',2),('X',3)] :: Map Char Int`
   $\leadsto$ `Map [('X',3),('Y',2)]`

d) `Nat` – Die bijektive Funktion der Isomorphie von `Nat` und `[()]`. Es werden also die Vorkommen von `()` in der Liste gezählt.
   Beispiel: `fromList [(),(),()] :: Nat` $\leadsto$ `Succ (Succ (Succ Zero))`

Lösungsvorschlag

```
-- Aufgabe 11.2 a)
instance FromList (Colist a) where
  type Item (Colist a) = a
  fromList (a:as) = Colist (Just (a,fromList as))
  fromList [] = nil

-- Aufgabe 11.2 b)
instance FromList (Maybe a) where
  type Item (Maybe a) = a
  fromList (a:_) = Just a
  fromList []    = Nothing

-- Aufgabe 11.2 c)
instance Eq a => FromList (Map a b) where
  type Item (Map a b) = (a,b)
  fromList ts = Map $ foldl (uncurry . updRel) [] ts

-- Aufgabe 11.2 d)
instance FromList Nat where
  type Item Nat = ()
  fromList (_:as) = Succ (fromList as)
  fromList []     = Zero
```

Aufgabe 11.3 (8 Punkte) *Funktor*
Schreiben Sie Instanzen der Klasse `Functor` für die folgenden Datentypen.

a) `BintreeL` (2 Punkte)
Beispiel: `fmap (*2) btl1` $\rightsquigarrow$ `2(4(4(6,-2),-4),8(-6,10))`

b) `BExp` (6 Punkte)
Beispiel: `fmap rename bexpr`
$\rightsquigarrow$ `And [Or [Not (BVar "b"),False_],BVar "b1",("y"+5) := 10]`

Lösungsvorschlag

```
-- Aufgabe 11.3 a)
instance Functor BintreeL where
  fmap f (Leaf a) = Leaf (f a)
  fmap f (Bin a l r) = Bin (f a) (fmap f l) (fmap f r)

-- Aufgabe 11.3 a)
instance Functor BExp where
  fmap f bexpr = case bexpr of
    True_ -> True_
    False_ -> False_
    BVar x -> BVar $ f x
    Or bs -> Or $ map (fmap f) bs
    And bs -> And $ map (fmap f) bs
    Not b -> Not $ fmap f b
    e1 := e2 -> fmap f e1 := fmap f e2
    e1 :<= e2 -> fmap f e1 :<= fmap f e2
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 12

Ausgabe: 11.1.2019, **Abgabe:** 18.1.2019 – 16:00 Uhr, **Block:** 4

Das Übungsblatt behandelt Themen bis einschließlich Folie 210.

Aufgabe 12.1 (4 Punkte) *do-Notation*
Gegeben sei folgende Listenkomprehension:

```
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z) | z <- [0..] , x <- [0..z] , y <- [0..z]
            , 5*x + 3*y^2 + 10 == z ]
```

a) Überführen Sie die Listenkomprehension in die do-Notation (Folie 210).

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

Lösungsvorschlag

```
-- Aufgabe 12.1 a)
solutionsDo :: [(Int, Int, Int)]
solutionsDo = do
  z <- [0..]
  x <- [0..z]
  y <- [0..z]
  guard $ 5*x + 3*y^2 + 10 == z
  return (x,y,z)

-- Aufgabe 12.1 b)
solutionsBind :: [(Int, Int, Int)]
solutionsBind =
  [0..] >>= \z ->
  [0..z] >>= \x ->
  [0..z] >>= \y ->
  (guard $ 5*x + 3*y^2 + 10 == z) >>
  return (x,y,z)
```

Aufgabe 12.2 (8 Punkte) *Maybe- und Listenmonade*
Sie dürfen für diese Aufgabe keine Konstruktoren der Datentypen `Maybe` oder `[]` benutzen. Sie können nach belieben `do`, `(>>=)`, `(<*>)` und weitere Monadenoperatoren und Monadenfunktionen benutzen.

a) `addvals :: Eq a => a -> a -> [(a,Int)] -> Maybe Int` – Mithilfe der Funktion `lookup` sollen die zugehörigen Werte in der Assoziationsliste gesucht werden und dann addiert werden.

$$addvals(a)(b)(ls) = lookup(a)(ls) + lookup(b)(ls)$$

Die Funktion `lookup` ist allerdings partiell (`Maybe`). Die Addition muss also mithilfe der Maybemonade geliftet werden.
Beispiele: `addvals 'a' 'b' [('b',3),('a',7)]` ⇝ `Just 10`
`addvals 'a' 'b' [('b',3),('c',7)]` ⇝ `Nothing`

b) `stepsFrom :: Int -> Int -> [Int]` – Wendet die Funktion `delta` mehrfach auf einen Startwert an. Dabei sei `delta` die Übergangsfunktion von `graph1` und ist bereits vorgegeben. Der erste Parameter gibt die Anzahl der Schritte an und der zweite Parameter den Startwert.

$$stepsFrom(n)(s) = \delta^n(s) \qquad \text{z. B.} \qquad stepsFrom(3)(1) = \delta(\delta(\delta(1)))$$

Ist die Anzahl der Schritte gleich 0, dann wird der Startwert wieder ausgegeben. Da `delta` nichtdeterministisch ist, also eine Liste ausgibt, jedoch als Eingabe eine Zahl verlangt, muss entsprechend mit der Listenmonade gearbeitet werden.
Beispiele: `0 `stepsFrom` 1` ⇝ `[1]`
`1 `stepsFrom` 1` ⇝ `[2,3]`
`2 `stepsFrom` 1` ⇝ `[1,4,6]`
`3 `stepsFrom` 1` ⇝ `[2,3,1,2,4,5]`

Lösungsvorschlag

```
-- Aufgabe 12.2 a)
addvals :: Eq a => a -> a -> [(a,Int)] -> Maybe Int
addvals a b ls = do
  x <- lookup a ls
  y <- lookup b ls
  return $ x + y
-- addvals a b ls = (+) <$> lookup a ls <*> lookup b ls
-- addvals a b ls = liftM2 (+) (lookup a ls) (lookup b ls)

-- Aufgabe 12.2 b)
stepsFrom :: Int -> Int -> [Int]
0 `stepsFrom` s = return s
n `stepsFrom` s = do
  next <- delta s
  (n-1) `stepsFrom` next
```

Aufgabe 12.3 (12 Punkte) *Plusmonade*
Implementieren Sie folgende Funktionen als Verallgemeinerung von gegebenen Listenfunktionen. Dabei soll der Typ von Listen auf beliebige Plusmonaden erweitert werden. Auf Listen sollen sich die neuen Funktionen genau wie ihr Original verhalten.

a) `removeM :: (MonadPlus m, Eq a) => a -> m a -> m a` – Entfernt jedes Vorkommen eines Elementes. (4 Punkte)
   Beispiele: `removeM 0 [0,1,0,2]` ⇝ `[1,2]`
   `removeM 0 (Just 2)` ⇝ `Just 2`
   `removeM 2 (Just 2)` ⇝ `Nothing`

b) `findIndicesM :: MonadPlus m => (a -> Bool) -> [a] -> m Int` – Die Funktion `findIndices` sei wie folgt definiert.

```
findIndices :: (a -> Bool) -> [a] -> [Int]
findIndices p xs = [ i | (x,i) <- zip xs [0..], p x]
```

   Diese kombiniert `filter` und `indices`. Sie gibt nur die Indizes der Werte aus, welche das Prädikat erfüllen. Die Rückgabe der Funktion `findIndicesM` soll für Plusmonaden erweitert werden. Ist die Rückgabe zum Beispiel vom Typ `Maybe`, soll nur der erste Index ausgegeben werden. (8 Punkte)
   Beispiele: `findIndices (>2) [1,5,2,4]` ⇝ `[1,3]`
   `findIndicesM (>2) [1,5,2,4] :: [Int]` ⇝ `[1,3]`
   `findIndicesM (>2) [1,5,2,4] :: Maybe Int` ⇝ `Just 1`

Lösungsvorschlag

```
-- Aufgabe 12.3 a)
removeM :: (MonadPlus m, Eq a) => a -> m a -> m a
removeM a m = do
  b <- m
  guard $ a /= b
  return b

-- Aufgabe 12.3 b)
findIndicesM :: MonadPlus m => (a -> Bool) -> [a] -> m Int
findIndicesM p xs = do
  (x,i) <- filtM $ zip xs [0..]
  guard $ p x
  return i
  where
    filtM [] = mzero
    filtM ((x,i):as) = if p x then return (x,i) `mplus` filtM as else filtM as
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*)
Lukasz Czajka (*lukasz.czajka@tu-dortmund.de*)

Wintersemester 2018/2019

Übungen zu Funktionaler Programmierung
Beispiellösung 13

Ausgabe: 18.1.2019, **Abgabe:** 25.1.2019 – 16:00 Uhr, **Block:** 4

Das Übungsblatt behandelt Themen bis einschließlich Folie 249.

Aufgabe 13.1 (7 Punkte) *Schreibermonade*
Schreiben Sie arithmetische Funktionen, welche ihren Rechenvorgang protokollieren. Machen Sie hierfür Gebrauch von einer Schreibermonade mit einem `String` als Protokoll. *Falls nicht anders angegeben, darf der Tupelkonstruktor `(,)` nicht benutzt werden.* Sie dürfen nur die Monadeneigenschaften von `(,) String` nutzen. Gehen Sie wie folgt vor.

a) Schreiben Sie eine Protokollierungsfunktion `logMsg :: String -> (String,())`, die einen beliebigen `String` an das Protokoll anhängt.
Sie dürfen den Tupelkonstruktor `(,)` für diese Funktion benutzen. (1 Punkt)

b) Schreiben Sie eine Protokollierungsfunktion

```
logOp :: Int -> Int -> Int -> String -> (String,()),
```

die eine arithmetische Operation protokolliert. Die Funktion erhält beide Eingaben für die Operation, die Ausgabe und eine `String`-Repräsentation der Operation. Machen Sie Gebrauch von der Funktion `logMsg`. Der Eintrag soll einen Zeilenumbruch einfügen und exakt folgendem Muster entsprechen. (2 Punkte)
Beispiele: `logOp 3 2 5 "+"` ⇝ `("The value of 3+2 is 5.\n",())`
`logOp 5 2 10 "*"` ⇝ `("The value of 5*2 is 10.\n",())`

c) Schreiben Sie eine Additionsfunktion und eine Multiplikationsfunktion (`add, mult :: Int -> Int -> (String,Int)`), welche ihren Vorgang protokolliert. Das Ausgabetupel enthält das Protokoll und das Ergebnis der Operation. Machen Sie Gebrauch von der Funktion `logOp` für die Protokollierung. (2 Punkte)
Beispiel: `add 3 2` ⇝ `("The value of 3+2 is 5.\n",5)`

d) Schreiben Sie eine Funktion

$$f(x,y,z) = x * (y + z)$$

als Haskell-Funktion, welche die Operationen protokolliert. Machen Sie Gebrauch von den Funktionen `add` und `mult`. (2 Punkte)
Beispiele: `snd $ f 3 1 1` ⇝ `6`
`putStrLn $ fst $ f 3 1 1` ⇝

```
The value of 1+1 is 2.
The value of 3*2 is 6.
```

Lösungsvorschlag

```
-- Aufgabe 13.1 a)
logMsg :: String -> (String,())
logMsg = flip (,) ()

-- Aufgabe 13.1 b)
logOp :: Int -> Int -> Int -> String -> (String,())
logOp x y z op = do
  logMsg "The value of "
  logMsg $ show x
  logMsg op
  logMsg $ show y
  logMsg " is "
  logMsg $ show z
  logMsg ".\n"

-- Aufgabe 13.1 c)
add :: Int -> Int -> (String,Int)
add x y = do
  let z = x + y
  logOp x y z "+"
  return z

mult :: Int -> Int -> (String,Int)
mult x y = do
  let z = x * y
  logOp x y z "*"
  return z

-- Aufgabe 13.1 d)
f :: Int -> Int -> Int -> (String,Int)
f x y z = mult x =<< add y z
```

Aufgabe 13.2 (9 Punkte) *Zustandsmonade*
Simulieren Sie eine zustandsbasierte Programmierung mithilfe einer Assoziationsliste (Folie 53) vom Typ `[(String,Int)]`. Die Assoziationsliste repräsentiert definierte Variablen (`String`) und deren Wert (`Int`). *Falls nicht anders angegeben, dürfen Konstruktor (`State`) und Destruktor (`runS`) der Zustandsmonade nicht benutzt werden.*
Schreiben Sie folgende Funktionen.

a) `(?=) :: String -> Int -> State [(String,Int)] ()` – Speichert das Paar aus einer Variablen und einem Wert in der Assoziationsliste bzw. ändert den Wert der Variablen.
Der Konstruktor `State` darf benutzt werden. (2 Punkte)

b) `get :: String -> State [(String,Int)] Int` – Liest den Wert einer Variablen aus.
Der Konstruktor `State` darf benutzt werden. (2 Punkte)

c) `progWrite :: Int -> Int -> State [(String,Int)] ()` – Speichert die Summe der Eingabe in der Variablen `"x"` und das Produkt in `"y"`. (2 Punkte)
Beispiel: `snd $ runS (progWrite 10 2) []` ⇝ `[("x",12),("y",20)]`

d) `progRead :: State [(String,Int)] Int` – Gibt y/x aus. (2 Punkte)
Beispiel: `fst $ runS progRead [("x",3),("y",12)]` ⇝ `4`

e) `prog :: Int -> Int -> State [(String,Int)] Int` – Führt zuerst `progWrite` mit den Angegebenen Variablen aus und gibt danach das Ergebnis von `progRead` aus. (1 Punkt)
Beispiel: `runS (prog 4 5) []` ⇝ `(2,[("x",9),("y",20)])`

Lösungsvorschlag

```
-- Aufgabe 13.2 a)
infix 0 ?=
(?=) :: String -> Int -> State [(String,Int)] ()
var ?= val = State $ \dict -> ((), updRel dict var val)

-- Aufgabe 13.2 b)
get :: String -> State [(String,Int)] Int
get var = State $ \dict -> (fromJust $ lookup var dict, dict)

-- Aufgabe 13.2 c)
progWrite :: Int -> Int -> State [(String,Int)] ()
progWrite a b = do
  "x" ?= a + b
  "y" ?= a * b

-- Aufgabe 13.2 d)
progRead :: State [(String,Int)] Int
progRead = div <$> get "y" <*> get "x"

-- Aufgabe 13.2 e)
prog :: Int -> Int -> State [(String,Int)] Int
prog a b = do
  progWrite a b
  progRead
```

Aufgabe 13.3 (4 Punkte) *IO-Monade*
Legen Sie eine Datei mit dem Namen `lines.hs` an. Die Datei soll nur die Funktion `main :: IO ()` enthalten, welches eine Textdatei einliest, die Zeilenumbrüche ändert und das Ergebnis in eine andere Datei schreibt. Das Programm soll interaktiv sein. Der Benutzer kann die Eingabedatei, die Art der Änderung und die Ausgabedatei bestimmen. Ein Ablauf soll wie folgt aussehen:

```
Eingabedatei:
<Eingabe>
Ändere Eingabe
  1. Entferne alle Zeilenumbrüche (einzeilig)
  2. Ein Wort je Zeile
Auswahl (1-2):
<Eingabe>
Ausgabedatei:
<Eingabe>
```

Übersetzen Sie das Programm mit dem Befehl

```
ghc lines.hs
```

in eine ausführbare Datei und führen Sie diese aus.

Hinweis: Die Funktionen `words`, `unwords`, `lines` und `unlines` von Folie 50 könnten für diese Aufgabe hilfreich sein.

Lösungsvorschlag

```
-- Aufgabe 13.3
main :: IO ()
main = do
  putStrLn "Eingabedatei:"
  inputFile <- getLine
  putStrLn "Ändere Eingabe"
  putStrLn "  1. Entferne alle Zeilenumbrüche (einzeilig)"
  putStrLn "  2. Ein Wort je Zeile"
  putStrLn "Auswahl (1-2):"
  option <- getLine
  putStrLn "Ausgabedatei:"
  outputFile <- getLine
  let operation = case read option of
        1 -> concat . lines
        2 -> unlines . words
  writeFile outputFile =<< operation <$> readFile inputFile
```

Aufgabe 13.4 (4 Punkte) *Dynamische Programmierung*
Gegeben sei folgende rekursive Funktion, welche die Anzahl aller möglichen booleschen Listen mit Länge `n` ohne aufeinanderfolgende `True`-Elemente ausgibt.

```
numOfString :: Int -> Int
numOfString n = loop(n, False) where
  loop(0, _)     = 0
  loop(1, False) = 2
  loop(1, True)  = 1
  loop(n, False) = loop(n-1, False) + loop(n-1, True)
  loop(n, True)  = loop(n-1, False)
```

Definieren Sie eine Funktion `numOfStringDyn :: 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. Beachten Sie, dass Sie ein Feld generieren müssen, das alle rekursiven Aufrufe von `loop(n, False)` enthält. Sie können die Funktion `range (start,end)` nutzen, um sich die Indizes anzeigen zu lassen, welche `mkArray (start,end)` erzeugt.

Lösungsvorschlag

```
numOfStringDyn :: Int -> Int
numOfStringDyn n = arr ! start where
  start = (n, False)
  arr = mkArray ((0,False),(n,True)) loop
  loop(0, _)     = 0
  loop(1, False) = 2
  loop(1, True)  = 1
  loop(n, False) = arr ! (n-1, False) + arr ! (n-1, True)
  loop(n, True)  = arr ! (n-1, False)
```