technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Präsenzblatt

Ausgabe: 11.10.2019, **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 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=16366`) 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

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 1

Ausgabe: 11.10.2019, **Abgabe:** 18.10.2019 – 16:00 Uhr, **Block:** 1

Das Übungsblatt behandelt Themen bis einschließlich Folie 17.

Aufgabe 1.1 (8 Punkte) *Summentypen*

a) Definieren Sie einen Typ `Genre` der eine Menge von mindestens drei Computerspielgenres repräsentiert.

b) Definieren Sie ein Element `m` vom Typ `Maybe Int` mit einem anderen Wert als `Nothing`.

c) Definieren Sie ein Element `eith` vom Typ `Either () Int`.

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

Lösungsvorschlag

```
-- Aufgabe 1.1 a)
data Genre = Action | Puzzle | Multiplayer

-- Aufgabe 1.2 b)
m :: Maybe Int
m = Just 4

-- Aufgabe 1.3 c)
eith :: Either () Int
eith = Left ()

-- Aufgabe 1.4 d)
tup :: (Color,Bool)
tup = (Magenta,True)
```

Aufgabe 1.2 (8 Punkte) *Produkttypen*

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

- Ein Computerspiel (`Game`) hat einen Titel, ein Genre und einen Preis.
- Ein Benutzer (`User`) hat einen Benutzernamen, ein Passwort, einen Kontostand und ein oder kein Spiel.

b) Legen Sie ein Beispielbenutzer `user` mit einem Spiel an. (2 Punkt)

c) Addieren Sie 50 zu dem Kontostand des Benutzers. Da es bei der rein funktionalen Programmierung nur Kontostand gibt, müssen Sie ein neuen Benutzer `balancePlus50` anlegen, der sich lediglich im Kontostant von `user` unterscheidet. (2 Punkt)

Lösungsvorschlag

```
-- Aufgabe 1.2 a)
data Game = Game
  { title :: String
  , genre :: Genre
  , price :: Int
  }

data User = User
  { username :: String
  , password :: String
  , balance :: Int
  , game :: Maybe Game
  }

-- Aufgabe 1.2 b)
user :: User
user = User
  { username = "TheLegend27"
  , password = "abc123"
  , balance = 100
  , game = Just Game
    { title = "Super Morio Country"
    , genre = Action
    , price = 10
    }
  }

-- Aufgabe 1.2 c)
balancePlus50 :: User
balancePlus50 = user { balance = balance user + 50 }
```

Aufgabe 1.3 (8 Punkte) *Mustererkennung*
Gegeben ist jeweils ein Muster und ein Ausdruck. Entscheiden Sie zuerst, ob das Muster den Ausdruck erfüllt. Ist dies der Fall, so geben Sie die entsprechende Belegung der Variablen an.

a)

Muster: `Just (c,i)`
Ausdruck: `Just (Red, 4)`

b)

Muster: `(Left x, y)`
Ausdruck: `(Left (Yellow,True), Just 3)`

Lösungsvorschlag

```
-- Aufgabe 1.3 a)
c = Red
i = 4

-- Augabe 1.3 b)
x = (Yellow,True)
y = Just 3
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 2

Ausgabe: 18.10.2019, **Abgabe:** 25.10.2019 – 16:00 Uhr, **Block:** 1

Das Übungsblatt behandelt Themen bis einschließlich Folie 37.

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

```
x   y    ( ) x y                                   (Präfix Operator, Folie 30)
f x y    x `f` y                                   (Infix Funktion, Folie 31)
\x -> ... \z -> e    \x ... z -> e                 (geschachtelte  -Abstraktion, Folie 24)
f = \x ... z -> e    f x ... z = e                 (Applikative Definition, Folie 21)

f x1 ... z1 | g1 = e1    f x ... z =               (case-Konstrukt, Folie 22)
          ⋮                case (x, ..., z) of
f xN ... zN | gN = eN        (x1,...,z1) | g1 -> e1
                                      ⋮
                             (xN,...,zN) | gN -> eN
```

Aufgabe 2.1 (4 Punkte) *-Ausdrücke auswerten*
Werten Sie folgende Ausdrücke *schrittweise* und *lazy* aus.

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

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

Lösungsvorschlag

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

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

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*.

$$f(t) = \begin{cases} x+y, & \text{falls } t = (\mathit{Right}(x), \mathit{Just}(y)) \text{ und } x, y \quad \mathit{Int} \\ x, & \text{falls } t = (\mathit{Left}(x), \mathit{Just}(y)) \text{ und } x, y \quad \mathit{Int} \\ 0, & \text{falls } t = (x, \mathit{Nothing}) \end{cases}$$

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

$$g(x,y) = \begin{cases} x+y, & \text{falls } x \quad y > 100 \\ x-y, & \text{falls } x \quad y \quad 100 \text{ und } x \text{ ungerade} \\ x \quad y, & \text{sonst} \end{cases}$$

Hinweis: Sie können die Haskell-Funktion `odd` benutzen.

Lösungsvorschlag

```
-- Aufgabe 2.2 a)
f :: (Either Int Int, Maybe Int) -> Int
f = \t -> case t of
  (Right x, Just y) -> x + y
  (Left x, _) -> x
  (_,Nothing) -> 0

-- Aufgabe 2.2 b)
g :: (Int,Int) -> Int
g = \t -> case t of
  (x,y)
    | x * y > 100 -> x + y
    | odd (x+y) -> x - y
    | otherwise -> x * y
```

Aufgabe 2.3 (4 Punkte) *Typinferenz*
Berechnen Sie die Typen des folgenden Ausdrucks mithilfe der Typinferenzregeln. Sie dürfen die ASCII-verträgliche Schreibweise von Folie 34 verwenden.

```
(\x -> (x, Right x)) (Just True)
```

Lösungsvorschlag

```
-- Aufgabe 2.3
```

$$\dfrac{\dfrac{x :: b}{\backslash x \to (x, \mathrm{Right}\ x) :: b \to (b, \mathrm{Either}\ a\ b)} \qquad \dfrac{\dfrac{\mathrm{Just} :: a \to \mathrm{Maybe}\ a \qquad \mathrm{True} :: \mathrm{Bool}}{\mathrm{Just}\ \mathrm{True} :: \mathrm{Maybe}\ \mathrm{Bool}}}{(\backslash x \to (x, \mathrm{Right}\ x))\ (\mathrm{Just}\ \mathrm{True}) :: (\mathrm{Maybe}\ \mathrm{Bool}, \mathrm{Either}\ a\ (\mathrm{Maybe}\ \mathrm{Bool}))}$$

$$\dfrac{x :: b \qquad \dfrac{\mathrm{Right} :: b \to \mathrm{Either}\ a\ b \qquad x :: b}{\mathrm{Right}\ x :: \mathrm{Either}\ a\ b}}{(x, \mathrm{Right}\ x) :: (b, \mathrm{Either}\ a\ b)}$$

Aufgabe 2.4 (4 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) `h $ f . g x y $ f`

Lösungsvorschlag

a) Wie in der Mathematik gilt, dass die Potenz stärker bindend, als Addition und Subtraktion sind. Bei Haskell sind die beiden letzteren Operatoren linksassoziativ. Sie werden also von links nach rechts ausgewertet.

Klammern: `(x + (y ^ 5)) - z`
Präfix: `(-) ((+) x ((^) y 5)) z`

b) Die Komposition (`.`) bindet stärker als der Applikationsoperator (`$`). Beide sind rechtsassoziativ, daher wird die zweite Applikation zuerst ausgeführt. Funktionsanwendungen sind linksassoziativ, werden also von links nach rechts ausgewertet. Sie besitzen außerdem automatisch die höchste Priorität und werden vor allen anderen Operatoren ausgeführt.

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

Aufgabe 2.5 (8 Punkte) *Haskell-Funktion auswerten*
Werten Sie den folgende Ausdrücke aus, indem Sie erst den Operator *schrittweise* in einen -Ausdruck umformen und dann den Ausdruck *schrittweise* auswerten.

a) `True == False` Die Gleicheit sei wie folgt definiert.

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

b) `(&&) `on` even` Der Operator `` `on` `` ist im Modul `Data.Function` wie folgt definiert.

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
op `on` f = \x y -> f x `op` f y
```

Lösungsvorschlag

```
-- Aufgabe 2.5 a)
{- (==) schrittweise in einen Lambda-Ausdruck umformen.
True == True = True
False == False = True
_ == _ = False
<=>
(==) True True = True
(==) False False = True
(==) _ _ = False
<=>
(==) b1 b2 = case (b1,b2) of
  (True,True) = True
  (False,False) = True
  (_,_) = False
<=>
(==) = \b1 b2 -> case (b1,b2) of
  (True,True) = True
  (False,False) = True
  (_,_) = False
<=>
(==) = \b1 -> \b2 -> case (b1,b2) of
  (True,True) = True
  (False,False) = True
  (_,_) = False
-}

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

```
--Aufgabe 2.5 b)
{- on schrittweise in einen Lambda-Ausdruck umformen.
op `on` f = \x y -> f x `op` f y
<=> on op f = \x y -> f x `op` f y
<=> on = \op f -> \x y -> f x `op` f y
<=> on = \op -> \f -> \x -> \y -> f x `op` f y
-}

{-   (&&) `on` even schrittwiese auswerten.
(&&) `on` even
<=> on (&&) even
~> (\op -> \f -> \x -> \y -> f x `op` f y) (&&) even
~> (\f -> \x -> \y -> f x && f y) even
~> \x -> \y -> even x && even y
-}
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 3

Ausgabe: 25.10.2019, **Abgabe:** 8.11.2019 – 16:00 Uhr, **Block:** 1

Das Übungsblatt behandelt Themen bis einschließlich Folie 50.

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

a) Eine Funktion zur Berechnung des Binomialkoeffizienten.

```
int binom(int n, int k) {
    int state = 1;
    int i = 1;
    while (i <= k) {
        state = state * (n - k + i) / i;
        i = i + 1;
    }
    return state;
}
```

Hineweis: Die Funktion `div` dividiert Ganzzahlen in Haskell.

b) Diese Funktion testet, ob alle Zahlen in einem Feld gleich Null sind. Benutzen Sie in Haskell eine Liste anstelle des Feldes.

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

Lösungsvorschlag

```
-- Aufgabe 3.1 a)
binom :: Int -> Int -> Int
binom n k = loop 1 1 where
  loop state i
    | i <= k     = loop (state * (n - k + i)  div  i) (i+1)
    | otherwise = state

-- Aufgabe 3.1 b)
```

```
allZero :: [Int] -> Bool
allZero ls = loop True ls where
  loop state (x:xs) = loop (state && x == 0) xs
  loop state [] = state
```

Aufgabe 3.2 (4 Punkte) *Funktionen implementieren*
Implementieren Sie folgende Funktionen in Haskell. Die Funktionen sollen mithilfe des Datentyps `Maybe` absturzsicher implementiert werden. Es dürfen nur die angegebenen Hilfsfunktionen benutzt werden.

a) Die Funktion `safeLast` soll sich ähnlich wie `last` verhalten. Anstelle eines Fehlers bei einer leeren Liste soll `Nothing` ausgeben werden.

b) Die Funktion `safeAdd` soll sich ähnlich wie `(+)` verhalten. Die Funktion soll allerdings mit `Maybe`-Werten umgehen können und `Nothing` ausgeben, falls ein Eingabeparameter `Nothing` ist. Sie dürfen `(+)` als Hilfsfunktion benutzen.
Beispiele: `safeLast [1,2,3]  safeAdd  safeLast [4]  Just 7`
`safeLast [1,2,3]  safeAdd  safeLast []  Nothing`

Lösungsvorschlag

```
-- Aufgabe 3.2 a)
safeLast :: [a] -> Maybe a
safeLast [a] = Just a
safeLast (_ : s) = safeLast s
safeLast [] = Nothing

-- Aufgabe 3.2 b)
safeAdd :: Maybe Int -> Maybe Int -> Maybe Int
safeAdd (Just x) (Just y) = Just (x + y)
safeAdd _ _ = Nothing
```

Aufgabe 3.3 (4 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 -Ausdrücke umzuformen.

a) `drop 2 [3,2,4,8,4,5] !! 1`

b) `takeWhile (<4) [3,2,8,4]`

Lösungsvorschlag

```
-- Aufgabe 3.3 a)
{-
drop 2 [3,2,4,8,4,5] !! 1
~>
drop 1 [2,4,8,4,5] !! 1
~>
drop 0 [4,8,4,5] !! 1
~>
[4,8,4,5] !! 1
~>
[8,4,5] !! 0
~>
```

```
8
-}

-- Aufgabe 3.3 b)
{-
takeWhile (<4) [3,2,8,4]
~>
if 3 < 4 then 3:takeWhile (<4) [2,8,4] else []
~>
3:takeWhile (<4) [2,8,4]
~>
3:if 2 < 4 then 2:takeWhile (<4) [8,4] else []
~>
3:2:takeWhile (<4) [8,4]
~>
3:2:if 8 < 4 then 8:takeWhile (<4) [4] else []
~>
3:2:[] = [3,2]
-}
```

Aufgabe 3.4 (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 `removeEverySecond` entfernt jedes zweite Elment aus einer Liste.
   Beispiele: `removeEverySecond [1,2,3,4,5]  [1,3,5]`
   `removeEverySecond [True]  [True]`

b) Die Funktion `countTrue` zählt, wie häufig der Wert `True` vorkommt.
   Beispiel: `countTrue [True,False,True]  2`

c) Die Funktion `countTrue` zählt, wie häufig der Buchstabe e in einem `String` vorkommt.
   Beispiel: `countE "Event"  2`

Lösungsvorschlag

```
-- Aufgabe 3.4 a)
removeEverySecond :: [a] -> [a]
removeEverySecond (a:b:s) = a : removeEverySecond s
removeEverySecond s       = s

-- Aufgabe 3.4 b)
countTrue :: [Bool] -> Int
countTrue (True : s)  = countTrue s + 1
countTrue (False : s) = countTrue s
countTrue []          = 0

countE :: String -> Int
countE ('E' : s) = countE s + 1
countE ('e' : s) = countE s + 1
countE (_ : s)   = countE s
countE []        = 0
```

Aufgabe 3.5 (4 Punkte) *Funktionslifting auf Listen*
Implementieren Sie folgende Aufgaben mithilfe der Funktion `map` oder `zipWith`. Diese müssen sinnvoll eingesetzt werden.

a) `invert :: [Int] -> [Int]` invertiert das Vorzeichen aller Ganzzahlen in einer Liste.
Beispiel: `invert [5,-2,3]  [-5,2,-3]`

b) `apply :: [a -> b] -> [a] -> [b]` wendet positionsweise eine Liste von Funktionen auf eine Liste von Parametern an und speichert die einzelnen Ergebnisse in einer Liste.
Beispiel: `apply [(+1),(*2),negate] [1,2,3]  [2,4,-3]` (= `[1+1,2*2,negate 3]`)

Lösungsvorschlag

```
-- Aufgabe 3.5 a)
invert :: [Int] -> [Int]
invert xs = map negate xs

-- Aufgabe 3.5 b)
apply :: [a -> b] -> [a] -> [b]
apply fs as = zipWith ($) fs as
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 4

Ausgabe: 8.11.2019, **Abgabe:** 15.11.2019 – 16:00 Uhr, **Block:** 2

Das Übungsblatt behandelt Themen bis einschließlich Folie 73.

Aufgabe 4.1 (8 Punkte) *Assoziationsliste*
Gegeben seien folgende Datentypen:

```
type Username = String
type ID = Int
type Games = [(ID,Game)]
type Users = [(Username,User)]
type Database = (Users,Games)
data Genre = Action | Adventure | Puzzle | SingleP | MultiP
  deriving (Show,Eq,Enum,Bounded)
data Game = Game
  { title :: String
  , genre :: [Genre]
  , price :: Int
  } deriving (Show,Eq)
data User = User
  { password :: String
  , balance :: Int
  , games :: [ID]
  } deriving (Show,Eq)
```

Definieren Sie folgende Funktionen. Fehlerbehandlungen sind nicht notwendig.

a) `updateGame :: ID -> Game -> Database -> Database` – Überschreibt ein Spiel (`Game`) mit angegebener `ID` in der Assoziationsliste `Games`. Falls das Spiel nicht existiert, wird ein neues hinzugefügt. (2 Punkte)

b) `listGames :: Username -> Database -> [Game]` – Listet alle Spiele (Attribut `games`) die ein Benutzer (`User`) mit angegebenem `Username` besitzt. (2 Punkte)

c) `buyGame :: Username -> ID -> Database -> Database` – Der Benutzer mit entsprechendem `Username` möchte das Spiel mit angegebener `ID` kaufen. Dazu muss geprüft werden, ob er genug Geld auf seinem Konto hat (`balance`) und falls ja, der Betrag abgezogen und das Spiel zu seinem Konto hinzugefügt werden. (4 Punkte)

Lösungsvorschlag

```
-- Aufgabe 4.1 a)
updateGame :: ID -> Game -> Database -> Database
updateGame gameID g (userDB,gameDB)
  = (userDB, updRel gameDB gameID g)

-- Aufgabe 4.1 b)
listGames :: Username -> Database -> [Game]
listGames username (uDB,gDB) = map f $ games user where
  Just user = lookup username uDB
  f gameID = fromJust $ lookup gameID gDB

-- Aufgabe 4.1 c)
buyGame :: Username -> ID -> Database -> Database
buyGame name gameID (userDB,gameDB)
  = (updRel userDB name newUser, gameDB) where
    Just g = lookup gameID gameDB
    Just user = lookup name userDB
    newUser = user
      { balance = balance user - price g
      , games = gameID : games user
      }
```

Aufgabe 4.2 (4 Punkte) *Listenfaltung auswerten*
Werten Sie folgende Haskell-Ausdrücke *schrittweise* und *lazy* aus.

a) `foldr (^) 3 [4,1,5]`

b) `foldl (^) 3 [4,1,5]`

Lösungsvorschlag

```
-- Aufgabe 4.2 a)
{-
foldr (^) 3 [4,1,5]
~> 4 ^ foldr (^) 3 [1,5]
~> 4 ^ (1 ^ foldr (^) 3 [5])
~> 4 ^ (1 ^ (5 ^ foldr (^) 3 []))
~> 4 ^ (1 ^ (5 ^ 3))
~> 4 ^ (1 ^ 125)
~> 4 ^ 1
~> 4
-}

-- Aufgabe 4.2 b)
{-
foldl (^) 3 [4,1,5]
~> foldl (^) (3 ^ 4) [1,5]
~> foldl (^) ((3 ^ 4) ^ 1) [5]
~> foldl (^) (((3 ^ 4) ^ 1) ^ 5) []
~> ((3 ^ 4) ^ 1) ^ 5
~> (81 ^ 1) ^ 5
~> 81 ^ 5
~> 3486784401
-}
```

Aufgabe 4.3 (6 Punkte) *Listenfaltung implementieren*
Implementieren Sie folgende Aufgaben mithilfe der Listenfaltungen `foldl` oder `foldr`. Diese müssen sinnvoll eingesetzt werden.

a) `sumMaybe :: [Maybe Int] -> Int` verhält sich wie `sum`, allerdings werden `Nothing`-Werte ignoriert.
Beispiel: `sumMaybe [Just 3, Nothing, Just 2] { 5`

b) `onlyRGB :: [Color] -> [Color]` entfernt alle Werte außer `Red`, `Green` und `Blue` aus einer Liste. Die Reihenfolge bleibt ansonsten erhalten.
Beispiel: `onlyRGB [Red,Magenta,Green,Yellow] { [Red,Green]`

c) `sumPoint :: [Point] -> (Float,Float)` addiert alle Werte für `x` und `y` aus einer Liste von Punkten.
Beispiel: `sumPoint [Point 1 10, Point 2 8, Point 3 (-6)] { (6.0,12.0)`

Lösungsvorschlag

```
-- Aufgabe 4.3 a)
sumMaybe :: [Maybe Int] -> Int
sumMaybe = foldl f 0 where
  f x (Just y) = x + y
  f x Nothing  = x

-- Aufgabe 4.3 b)
onlyRGB :: [Color] -> [Color]
onlyRGB = foldr f [] where
  f color cs
    | color `elem` [Red,Green,Blue] = color : cs
    | otherwise = cs

-- Aufgabe 4.3 c)
sumPoint :: [Point] -> (Float,Float)
sumPoint = foldl f (0,0) where
  f (sumX,sumY) pt = (x pt + sumX, y pt + sumY)
```

Aufgabe 4.4 (6 Punkte) *Listenkomprehension*
Definieren Sie folgende Funktionen mithilfe der Listenkomprehension.

a) `ker :: Eq b => (a -> b) -> [a] -> [(a,a)]` gibt den Kern einer Funktion f aus, also alle $(a,a') \in A \times A$ für die gilt $f(a) = f(a')$. Der zweite Parameter bestimmt die Menge A.
Beispiel: `ker (^2) [-2,0,2] { [(-2,-2),(-2,2),(0,0),(2,-2),(2,2)]`

b) `solutions :: [(Int, Int, Int)]` enthält Tripel $(x,y,z) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, welche die Gleichung $3x^2 + 10z + 7 = y$ lösen. Nehmen Sie für x, y und z nur Werte von 0 bis 100.

c) `codes :: [[(Char,Int)]]` gibt alle Lösungen für das Kryptogramm *sonne* + *strand* = *ferien*.

Lösungsvorschlag

```
-- Aufgabe 4.4 a)
ker :: Eq b => (a -> b) -> [a] -> [(a,a)]
ker f as = [(a,a') | a <- as, a' <- as, f a == f a']

-- Aufgabe 4.4 b)
solutions :: [(Int,Int,Int)]
solutions = [ (x,y,z) | x <- [0..100], y <- [0..100]
            , z <- [0..100] , 3*x^2 + 10*z + 7 == y ]

-- Aufgabe 4.4 c)
codes :: [[(Char,Int)]]
codes = [zip "sonetradfi" nums
  | nums <- perms [0..9], correct nums] where
    correct :: [Int] -> Bool
    correct [s,o,n,e,t,r,a,d,f,i] = f /= 0 && s /= 0
      && 100000*s + 10000*(s+t) + 1000*(o+r) + 100*(n+a)
        + 10*(n+n) + e+d
      == 100000*f + 10000*e + 1000*r + 100*i + 10*e + n
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 5

Das Übungsblatt behandelt Themen bis einschließlich Folie 88.

Aufgabe 5.1 (4 Punkte) *Unendliche Listen auswerten*
Werten Sie folgende Haskell-Ausdrücke *schrittweise* und *lazy* aus.

a) `take 2 (iterate (+2) 10)`

b) `map (\x -> fib (50*x)) blink !! 2` Funktion `fib` auf Folie 254.

Lösungsvorschlag

```
-- Aufgabe 5.1 a)
{-
take 2 (iterate (+2) 10)
~> take 2 (10 : iterate (+2) (10 + 2))
~> take 2 (10 : iterate (+2) (10 + 2))
~> 10 : take (2-1) (iterate (+2) (10 + 2))
~> 10 : take 1 (iterate (+2) (10 + 2))
~> 10 : take 1 (10 + 2 : iterate (+2) (10 + 2 + 2))
~> 10 : 10 + 2 : take 0 (iterate (+2) (10 + 2 + 2))
~> 10 : 10 + 2 : []
~> 10 : 12 : [] = [10,12]
-}

-- Aufgabe 5.1 b)
{-
map (\x -> fib (50*x)) blink !! 2
~> map (\x -> fib (50*x)) (0 : 1 : blink) !! 2
~> ((\x -> fib (50*x)) 0:map (\x -> fib (50*x)) (1 : blink)) !! 2
~> map (\x -> fib (50*x)) (1 : blink) !! (2-1)
~> ((\x -> fib (50*x)) 1:map (\x -> fib (50*x)) blink) !! (2-1)
~> ((\x -> fib (50*x)) 1:map (\x -> fib (50*x)) blink) !! 1
~> map (\x -> fib (50*x)) blink !! (1-1)
~> map (\x -> fib (50*x)) (0 : 1 : blink) !! (1-1)
~> ((\x -> fib(50*x)) 0:map (\x -> fib(50*x)) (1:blink)) !! (1-1)
~> ((\x -> fib(50*x)) 0:map (\x -> fib(50*x)) (1:blink)) !! 0
~> ((\x -> fib (50*x)) 0
~> fib (50*0)
~> fib 0
~> 1
-}
```

```
-- 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 (+) (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) `nums :: [Int]` – Liste aller positiven Zahlen, die durch 3 oder 5 teilbar sind.
Beispiel: `take 10 nums` $\rightsquigarrow$ `[3,5,6,9,10,12,15,18,20,21]`

b) `collatz :: Int -> [Int]` – Liste beginnt mit einer beliebigen nicht negativen Zahl n. Die nächste Zahl ist $n/2$ falls n gerade ist und $3n+1$ sonst. Wiederhole den Vorgang mit der nächsten Zahl. (Collatz-Problem: `https://de.wikipedia.org/wiki/Collatz-Problem`)
Beispiel: `take 10 (collatz 19)` $\rightsquigarrow$ `[19,58,29,88,44,22,11,34,17,52]`

c) `solutions :: [(Int, Int, Int)]` – Liste *aller* Tripel $(x, y, z) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$, welche die Gleichung $3x^2 + 10z + 7 = y$ lösen. Nehmen Sie für x, y und z nur positive Werte.
Beispiel: `(4,115,6) `elem` solutions` $\rightsquigarrow$ `True`

Lösungsvorschlag

```
-- Aufgabe 5.2 a)
nums :: [Int]
nums = [ n | n <- [1..], n `mod` 3 == 0 || n `mod` 5 == 0]

-- Aufgabe 5.2 b)
collatz :: Int -> [Int]
collatz n
  | n > 0 = n : collatz (if even n then n `div` 2 else 3*n + 1)

-- Aufgabe 5.2 c)
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z) | y <- [0..], x <- [0..y]
            , z <- [0..y] , 3*x^2 + 10*z + 7 == y ]
```

Aufgabe 5.3 (6 Punkte) *Rekursive Datentypen*
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 `three :: Int'`, welche den Wert 3 repräsentiert.

b) Definieren Sie einen Datentyp `Bits` der einen Bitvektor (Liste von 0 und 1) in big-endian repräsentiert. Der Datentyp darf keine anderen Datentypen benutzen.

c) Definieren Sie eine Konstante `vec :: Bits`, welche den Bitvektor 100 repräsentiert.

Lösungsvorschlag

```
-- Aufgabe 5.3 a)
three :: Int'
three = Plus $ Succ' $ Succ' One

-- Aufgabe 5.3 b)
data Bits = Nil | ConsZero Bits | ConsOne Bits
  deriving Show

-- Aufgabe 5.3 c)
vec :: Bits
vec = ConsOne $ ConsZero $ ConsZero Nil
```

Aufgabe 5.4 (8 Punkte) *Funktionen auf rekursiven Datentypen*
Definieren Sie folgende Haskell-Funktionen. Ersetzen Sie dabei den Datentyp `Int` durch den hier besser geeigneten Datentyp `Nat` und Listentypen durch Colisten (`Colist`) bzw. Ströme (`Stream`).

a) `colistLength :: Colist a -> Nat`, wie `length` für Colisten.
Beispiel: `colistLength co123` $\rightsquigarrow$ `Succ (Succ (Succ Zero))`

b) `updStream :: Stream a -> Nat -> a -> Stream a`, wie `updList` für Ströme.
Beispiel: `updStream blinkS (Succ Zero) 3` $\rightsquigarrow$ `Stream [0,3,0,1,0,1,0,1,...]`

Lösungsvorschlag

```
-- Aufgabe 5.4 a)
colistLength :: Colist a -> Nat
colistLength ls = case split ls of
  Just (a,s) -> Succ $ colistLength s
  Nothing -> Zero

-- Aufgabe 5.4 b)
updStream :: Stream a -> Nat -> a -> Stream a
updStream (_ :< s) Zero b     = b :< s
updStream (a :< s) (Succ n) b = a :< updStream s n b
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 6

Das Übungsblatt behandelt Themen bis einschließlich Folie 99.

Aufgabe 6.1 (4 Punkte) *Strikte Faltung von links her*
Machen Sie den Unterschied zwischen strikter und nicht-strikter Linksfaltung deutlich. Werten Sie dazu folgende Haskell-Ausdrücke *schrittweise* und *lazy* aus. Arithmetische Ausdrücke müssen in ihrer korrekten Reihenfolge aufgelöst werden. Die Funktion `seq` darf aber sofort ausgerechnet werden und muss nicht mehr in der Lösung auftauchen.

a) `foldl  (+) 5 [3, 4*2, 3-1]`

b) `foldl' (+) 5 [3, 4*2, 3-1]`

Lösungsvorschlag

```
-- Aufgabe 6.1 a)
{-
foldl  (+) 5 [3, 4*2, 3-1]
foldl  (+) (5 + 3) [4*2, 3-1]
foldl  (+) (5 + 3 + 4*2) [3-1]
foldl  (+) (5 + 3 + 4*2 + (3-1)) []
(5 + 3 + 4*2) + (3-1)
8 + 4*2 + (3-1)
8 + 8 + (3-1)
16 + (3-1)
16 + 2
18
-}

-- Aufgabe 6.1 b)
{-
foldl' (+) 5 [3, 4*2, 3-1]
foldl' (+) 8 [4*2, 3-1]
foldl' (+) 16 [3-1]
foldl' (+) 18 []
18
-}
```

Aufgabe 6.2 (12 Punkte) *Funktionen auf rekursiven Datentypen*
Definieren Sie folgende Haskell-Funktionen. Ersetzen Sie dabei den Datentyp `Int` durch den hier besser geeigneten Datentyp `Nat` und Listentypen durch Colisten (`Colist`) bzw. Ströme (`Stream`).

a) `newLength :: Colist a -> Nat`, wie `length` für Colisten.
   Beispiel: `colistLength co123` $\rightsquigarrow$ `Succ (Succ (Succ Zero))`

b) `newUpd :: Stream a -> Nat -> a -> Stream a`, wie `updList` für Ströme.
   Beispiel: `updStream blinkS (Succ Zero) 3` $\rightsquigarrow$ `Stream [0,3,0,1,0,1,0,1,...]`

c) `newTake :: Nat -> Stream a -> Colist a`, wie `take` mit allen Typen ersetzt.
   Beispiel: `newTake (Succ $ Succ Zero) blinkS` $\rightsquigarrow$ `Colist [0,1]`

Lösungsvorschlag

```
-- Aufgabe 6.2 a)
newLength :: Colist a -> Nat
newLength ls = case split ls of
  Just (a,s) -> Succ $ newLength s
  Nothing -> Zero

-- Aufgabe 6.2 b)
newUpd :: Stream a -> Nat -> a -> Stream a
newUpd s Zero b            = b :< tl s
newUpd (a :< s) (Succ n) b = a :< newUpd s n b

-- Aufgabe 6.2 b)
newTake :: Nat -> Stream a -> Colist a
newTake Zero _ = nil
newTake (Succ n) (a :< s) = Colist $ Just (a, newTake n s)
```

Aufgabe 6.3 (8 Punkte) *Arithmetische und Boolesche Ausdrücke*
Implementieren Sie folgende arithmetische und boolesche Ausdrücke als Haskell-Konstanten.

a) `expr1 :: Exp String`
   `expr1` $= 4x^2 + 9x + 2$

b) `expr2 :: Exp String`
   `expr2` $= a^2 - 2ab + b^2$

c) `bexpr1 :: BExp String`
   `bexpr1` $= b \vee \mathit{false} \vee (\neg b \wedge \mathit{true})$

d) `bexpr2 :: BExp String`
   `bexpr2` $= (0 \leq x \wedge x \leq 100) \vee y = 50$

Lösungsvorschlag

```
-- Aufgabe 6.3 a)
expr1 :: Exp String
expr1 = Sum [4 :* (Var "x" :^ 2), 9 :* Var "x", Con 2]

-- Aufgabe 6.3 a)
expr2 :: Exp String
expr2 = Sum [((Var "a" :^ 2) :- Prod [Con 2, Var "a", Var "b"]),
  Var "b" :^ 2]

-- Aufgabe 6.3 a)
bexpr1 :: BExp String
bexpr1 = Or [BVar "b", False_, And [Not (BVar "b"), True_]]

-- Aufgabe 6.3 a)
bexpr2 :: BExp String
bexpr2 = Or [And [Con 0 :<= Var "x", Var "x" :<= Con 100],
  Var "y" := Con 50]
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 7

Das Übungsblatt behandelt Themen bis einschließlich Folie 108.

Aufgabe 7.1 (10 Punkte) *Boolesche Ausdrücke auswerten*

a) Definieren Sie eine Konstante `formula :: BExp String` für den Ausdruck $3x^2 + 10z + 7 = y$. Stellen Sie den Ausdruck als Elemente vom Typ `BExp String` da. (2 Punkte)

b) Schreiben Sie eine Funktion `evalB :: BExp x -> BStore x -> Store x -> Bool`, welche boolesche Ausdrücke auswertet. Sie können sich an der Funktion `eval` (Folie 104) orientieren. Zusätzlich zu einer Variablenbelegung `Store x` für arithmetische Variablen, wird eine weitere Belegung `BStore x` für boolesche Variablen benötigt. (6 Punkte)

```
type BStore x = x -> Bool
```

c) Schreiben Sie die Listenkomprehension `solutions :: [(Int,Int,Int)]` um. Machen Sie sinnvollen Gebrauch von dem Ausdruck `formula` und der Funktion `evalB`. (2 Punkte)

Lösungsvorschlag

```
-- Aufgabe 7.1 a)
formula :: BExp String
formula = Sum [3:*(Var "x" :^ 2), 10:* Var "z", Con 7] := Var "y"

-- Aufgabe 7.1 b)
evalB :: BExp x -> BStore x -> Store x -> Bool
evalB e bst st = case e of
  False_ -> False
  True_ -> True
  BVar x -> bst x
  Or es -> any recursive es
  And es -> all recursive es
  Not e -> not $ recursive e
  e1 := e2 -> eval e1 st == eval e2 st
  e1 :<= e2 -> eval e1 st <= eval e2 st
  where recursive e = evalB e bst st


-- Aufgabe 7.1 c)
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z) | y <- [0..], x <- [0..y] , z <- [0..y]
  , let st "x" = x; st "y" = y; st "z" = z
  , evalB formula (const False) st ]
```

Aufgabe 7.2 (6 Punkte) *Signaturen, Algebren und Faltungen*
Die Funktion `evalNat` soll einen Ausdruck für natürliche Zahlen vom Typ `NatE` in den entsprechenden Wert vom Typ `Int` wandeln. Dabei soll die Funktion Gebrauch von einer Faltung machen.

```
data NatE = Z | S NatE | NatE :+: NatE | NatE :*: NatE
  deriving (Show,Eq)

evalNat :: NatE -> Int
evalNat = foldNat natAlg
```

Beispiel: `evalNat $ S (S Z) :*: (S Z :+: S (S (S Z)))` $\rightsquigarrow$ `8`
Wobei `S (S Z) :*: (S Z :+: S (S (S Z)))` dem Ausdruck $2 * (1 + 3)$ entspricht.

Implementieren Sie sie Signatur, die Algebra und die Faltungen. Gehen Sie in folgender Reihenfolge vor.

a) Implementieren Sie eine Signatur `NatSig` für den Datentypen `NatE`.

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

c) Implementieren Sie die Algebra `natAlg` vom Typ `NatSig`, die einen Wert vom Typ `NatE` den entsprechenden Wert vom Typ `Int` wandeln.

Lösungsvorschlag

```
-- Aufgabe 7.2 a)
data NatSig val = NatSig
  { z :: val
  , s :: val -> val
  , plus :: val -> val -> val
  , mult :: val -> val -> val
  }


-- Aufgabe 7.2 b)
foldNat :: NatSig val -> NatE -> val
foldNat alg Z = z alg
foldNat alg (S n) = s alg $ foldNat alg n
foldNat alg (n :+: m) = plus alg (foldNat alg n) (foldNat alg m)
foldNat alg (n :*: m) = mult alg (foldNat alg n) (foldNat alg m)

-- Aufgabe 7.2 c)
natAlg :: NatSig Int
natAlg = NatSig
  { z = 0
  , s = (+1)
  , plus = (+)
  , mult = (*)
  }
```

Aufgabe 7.3 (8 Punkte) *Isomorphismen*
Die folgenden Datentypen sind isomorph. Geben Sie eine bijektive Funktion `from` und ihr Inverses `to` an, welche die Gleichungen

```
to . from  =  id
from . to  =  id
```

erfüllen. Es ist ausreichend die Funktionen anzugeben. Ein Beweis muss nicht geführt werden.

a) `PosNat` ≅ `[()]`: Definieren Sie die Funktion `fromPosNat` und ihr Inverses `toPosNat`.

b) `Colist a` ≅ `[a]`: Definieren Sie die Funktion `fromColist` und ihr Inverses `toColist`.

Lösungsvorschlag

```
-- Aufgabe 7.3 a)
fromPosNat :: PosNat -> [()]
fromPosNat One = []
fromPosNat (Succ' n) = () : fromPosNat n

toPosNat :: [()] -> PosNat
toPosNat [] = One
toPosNat (() : ls) = Succ' $ toPosNat ls

-- Aufgabe 7.3 b)
fromColist :: Colist a -> [a]
fromColist ls = case split ls of
  Just (a,as) -> a : fromColist as
  Nothing -> []

toColist :: [a] -> Colist a
toColist (a:as) = Colist $ Just (a, toColist as)
toColist [] = Colist Nothing
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 8

Das Übungsblatt behandelt Themen bis einschließlich Folie 123.

Aufgabe 8.1 (8 Punkte) *Typklassen*
Definieren und Instanziieren Sie eine eigene Typklasse.

a) Schreiben Sie eine Klasse für einen überladenen Operator `(!!!)`. Dieser soll sich wie `(!!)` verhalten, aber nicht auf den Zahlentyp `Int` beschränkt sein. (2 Punkte)

b) Instanziieren Sie die Klasse für `Int`, `Nat` und `Int'`. (6 Punkte)

Lösungsvorschlag

```
-- Aufgabe 8.1 a)
class Index i where
  (!!!) ::  [a] -> i -> a

-- Aufgabe 8.1 b)
instance Index Int where
  (!!!) = (!!)

instance Index Nat where
  (a : _) !!! Zero = a
  (_ : s) !!! Succ n = s !!! n

instance Index Int' where
  (a : _) !!! Zero' = a
  (_ : a : _) !!! Plus One = a
  (_ : s) !!! Plus (Succ' n) = s !!! Plus n
```

Aufgabe 8.2 (11 Punkte) *Typklassen*
Instanziieren Sie mehrere Typklassen für den Datentyp `PosNat`. Sie dürfen nur rekursive Aufrufe, Konstruktoren, Literale, arithmetischen Operatoren (`(+)`, `(*)`, etc.) und Vergleichsoperatoren (`(<)`, `(>)`, etc.) benutzen. Andere Funktionen und Operatoren dürfen nicht genutzt werden. Alle Funktionen sollen sich wie für positive `Int` verhalten.

a) Schreiben Sie eine sinnvolle Instanz der Klasse `Num`. Das verhalten der Funktion `negate` ist für den Typ `PosNat` nicht definiert. (4 Punkte)

b) Schreiben Sie Instanzen für die Klassen `Enum` und `Eq`. (4 Punkte)
Beispiel: `map fromEnum [One .. Succ' (Succ' One)]` $\rightsquigarrow$ `[1,2,3]`

c) Schreiben Sie eine Instanz für die Klasse `Show`. Sie *dürfen* beliebige Hilfsfunktion für diese Aufgabe einsetzen. (2 Punkte)
Beispiel: `show $ Succ' $ Succ' One` $\rightsquigarrow$ `"3"`

d) Ändern Sie den Typ der Liste `solutions` von Übungsblatt 5 in `[(PosNat, PosNat, PosNat)]` und passen Sie Ihre Lösung entsprechend an. Sie dürfen nur notwendige Änderungen vornehmen. (1 Punkt)

Lösungsvorschlag

```
-- Aufgabe 8.2 a)
instance Num PosNat where
  negate = undefined
  fromInteger 1 = One
  fromInteger i | i > 1 = Succ' (fromInteger (i-1))
  signum _ = 1
  abs n = n
  One + n = Succ' n
  Succ' n + m = n + Succ' m
  One * n = n
  Succ' n * m = n * m + m

-- Aufgabe 8.2 b)
instance Enum PosNat where
  toEnum 1 = One
  toEnum i | i > 1 = Succ' (toEnum (i-1))
  fromEnum One = 1
  fromEnum (Succ' n) = fromEnum n + 1

instance Eq PosNat where
  One == One = True
  Succ' n == Succ' m = n == m
  _ == _ = False

-- Aufgabe 8.2 c)
instance Show PosNat where
  show = show . fromEnum

-- Aufgabe 8.2 d)
solutions :: [(PosNat, PosNat, PosNat)]
solutions = [ (x,y,z) | y <- [1..] , x <- [1..y] , z <- [1..y]
            , 3*x^2 + 10*z + 7 == y ]
```

Aufgabe 8.3 (3 Punkte) *Binäre Bäume*
Definieren Sie die Funktionen `maxTree`, welche den größten Wert aus den Knoten eines Baumes zurückgibt. Ist der Baum leer, wird `Nothing` ausgegeben. Geben Sie den möglichst allgemeinsten Typ der Funktion an.
Beispiel: `maxTree btree4` $\leadsto$ `Just 121`

Lösungsvorschlag

```
-- Aufgabe 8.3
maxTree :: Ord a => Bintree a -> Maybe a
maxTree Empty = Nothing
maxTree (Fork a l r) = maximum [Just a, maxTree l, maxTree r]
```

Aufgabe 8.4 (2 Punkte) *Fakultätsfunktion*
Gegeben ist folgende rekursive Definition der Fakultätsfunktion:

```
fact :: Nat -> Int
fact Zero = 1
fact (Succ n) = fact n * (mkInt n + 1)

mkInt :: Nat -> Int
mkInt Zero = 0
mkInt (Succ n) = mkInt n + 1
```

Wie muss eine *NatSig*-Algebra

```
factAlg :: NatSig (Int, Int)
```

definiert werden, damit sie die Gleichungen

$$foldNat(factAlg) = \langle fact, mkInt \rangle : Nat \to Int^2$$

erfüllt?

Lösungsvorschlag

```
-- Aufgabe 8.4
factAlg :: NatSig (Int,Int)
factAlg = NatSig
  { zero_ = (1,0)
  , succ_ = \(x,y) -> (x*(y+1), y+1)
  }
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 9

Das Übungsblatt behandelt Themen bis einschließlich Folie 145.

Aufgabe 9.1 (6 Punkte) *Ausgabe*
Instanziieren Sie den Datentyp `NatE` für die Klasse `Show`.

```
data NatE = Z | S NatE | NatE :+: NatE | NatE :*: NatE
```

Benutzen Sie für die Ausgabe der Konstruktoren folgende Zeichenketten.

Konstruktor	Zeichen
`Z`	`0`
`S`	+1 (Suffix)
`:+:`	`+`
`:*:`	`*`

Alle Operatoren sind rechtsassoziativ. Die Multiplikation soll stärker bindend sein als die Addition. Beides gilt auch für +1. Definieren Sie dazu die Funktion `showsPrec`.
Beispiele:

```
S Z                        ⇝  0+1
(Z :+: S Z) :*: S (S Z)    ⇝  (0+0+1)*((0+1)+1)
Z :+: S Z :*: S (S Z)      ⇝  0+(0+1)*((0+1)+1)
```

Lösungsvorschlag

```
-- Aufgabe 9.1
instance Show NatE where
  showsPrec _ Z = showChar '0'
  showsPrec p (S n) = showParen (p > 6)
    $ showsPrec 7 n . showString "+1"
  showsPrec p (n :+: m) = showParen (p > 6)
    $ showsPrec 7 n . showChar '+' . showsPrec 6 m
  showsPrec p (n :*: m) = showParen (p > 7)
    $ showsPrec 8 n . showChar '*' . showsPrec 7 m
```

Aufgabe 9.2 (6 Punkte) *Einlesen*
Schreiben Sie eine Instanz der Klasse `Read` 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 Syntax soll sich an den binären Bäumen aus der Vorlesung orientieren (Folie 135f.):

```
read "4(2(9,),7(,3))" :: STree Int
    ⇝ BinS (LeftS (LeafS 9) 2) 4 (RightS 7 (LeafS 3))
```

Lösungsvorschlag

```
-- Aufgabe 9.2
instance Read a => Read (STree a) where
  readsPrec _ s
    =  [(BinS l a r, s6) | (a, s1) <- reads s, ("(",s2) <- lex s1
      , (l,s3) <- reads s2, (",",s4) <- lex s3
      , (r,s5) <- reads s4, (")",s6) <- lex s5]
    ++ [(LeftS l a, s5) | (a, s1) <- reads s, ("(",s2) <- lex s1
      , (l,s3) <- reads s2, (",",s4) <- lex s3
      , (")",s5) <- lex s4]
    ++ [(RightS a r, s5) | (a, s1) <- reads s, ("(",s2) <- lex s1
      , (",",s3) <- lex s2, (r,s4) <- reads s3
      , (")",s5) <- lex s4]
    ++ [(LeafS a, s1) | (a,s1) <- reads s]
```

Aufgabe 9.3 (6 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) `postorderT :: Tree  a -> [a]` – Wie `postorder` (Folie 144) ohne Faltung.
   Beispiel: `postorderT tree1` ⇝ `[3,-1,2,-2,2,-3,5,4,1]`

b) `maxChildren :: Tree a -> Int` – Gibt die Anzahl der Kinder des Knotens, mit den meisten Kinder an.
   Beispiel: `maxChildren tree2` ⇝ `9`

c) `nodesWith :: Eq a => a -> Tree a -> [Node]` – Die Funktion erhält ein Element und ein Baum und gibt alle Knoten aus, welche das Element enthalten.
   Beispiel: `nodesWith 11 tree2` ⇝ `[[],[7,0],[8]]`

Lösungsvorschlag

```
-- Aufgabe 9.3 a)
postorderT :: Tree  a -> [a]
postorderT (V a) = [a]
postorderT (F a ts) = concatMap postorderT ts ++ [a]

--  Aufgabe 9.3 b)
maxChildren :: Tree a -> Int
maxChildren (V _) = 0
maxChildren (F _ ts) = maximum $ length ts : map maxChildren ts

--  Aufgabe 9.3 c)
nodesWith :: Eq a => a -> Tree a -> [Node]
nodesWith a t = [ node | node <- nodes t, label t node == a ]
```

Aufgabe 9.4 (6 Punkte) *Baumfaltungen*
Definieren Sie folgende Funktionen als Baumfaltungen (`foldBin` bzw. `foldTree`).

a) `bintreeSum :: Num a => Bintree a -> a` – Addiert alle Knoteninhalte.

   Beispiel: `bintreeSum btree1` $\leadsto$ `121`

b) `treeAnd :: Tree Bool -> Bool` – Gibt den Wert `True` aus, falls alle Knoten den Wert `True` enthalten und sonst `False`.

   Beispiele: `treeAnd $ F True [V True]` $\leadsto$ `True`
   `treeAnd $ F False [V True]` $\leadsto$ `False`

c) `heightA :: Tree a -> Int` – Wie `height` (Folie 139) mit Faltung.

   Beispiel: `heightA tree1` $\leadsto$ `4`

Lösungsvorschlag

```
-- Aufgabe 9.4 a)
bintreeSum :: Num a => Bintree a -> a
bintreeSum = foldBin BinSig
  { empty_ = 0
  , fork = \a l r -> a + l + r
  }

-- Aufgabe 9.4 b)
treeAnd :: Tree Bool -> Bool
treeAnd = foldTree TreeSig
  { var = id
  , fun = \t ts -> and (t:ts)
  }

-- Aufgabe 9.4 c)
heightA :: Tree a -> Int
heightA = foldTree TreeSig
  { var = const 1
  , fun = \_ ts -> 1 + maximum (0:ts)
  }
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 10

Das Übungsblatt behandelt Themen bis einschließlich Folie 183.

Aufgabe 10.1 (4 Punkte) *Arithmetische Ausdrücke kompilieren*
Geben Sie die Kommandosequenz für die Auswertung des folgenden Ausruckes an.

```
execute (foldArith codeAlg expr) ([],vars)
```

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 [((Var "a" :^ 2) :- Prod [Con 2, Var "a", Var "b"])
  ,Var "b" :^ 2]

vars :: Store String
vars "a" = 5
vars "b" = 3
```

Tragen Sie die Lösung in die Tabelle `table :: [(StackCom String, [Int])]` ein. Sie können mit `fst` den finalen Stapelinhalt und damit das Ergebnis der Ausführung erhalten.
`fst $ execute (foldArith codeAlg expr) ([],vars)` ⇝ `[4]`

Lösungsvorschlag

```
table :: [(StackCom String, [Int])]
table =
  --(Kommando   , Stapel       )
  [ (Load "a"   , [5]          )
  , (Push 2     , [2,5]        )
  , (Up         , [25]         )
  , (Push 2     , [2,25]       )
  , (Load "a"   , [5,2,25]     )
  , (Load "b"   , [3,5,2,25]   )
  , (Mul 3      , [30,25]      )
  , (Sub        , [-5]         )
  , (Load "b"   , [3,-5]       )
  , (Push 2     , [2,3,-5]     )
  , (Up         , [9,-5]       )
  , (Add 2      , [4]          )
  ]
```

Aufgabe 10.2 (6 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 *searchPrime* in Haskell.

$$\begin{aligned} &\mathit{searchPrime} : \mathbb{Z}_{10} \to \mathbb{Z}_{10} \\ &\mathit{searchPrime}(x) = \begin{cases} x & \text{falls } x \in \{1,2,3,4,7\} \\ x-1 & \text{falls } x \notin \{1,2,3,4,7\} \text{ und } x > 4 \\ x+1 & \text{falls } x \notin \{1,2,3,4,7\} \text{ und } x < 5 \end{cases} \end{aligned}$$

b) Berechnen Sie den kleinsten Fixpunkt (`lfp`) von *searchPrime* mithilfe der Funktion `fixpt` (Folie 166).

c) Berechnen Sie den größten Fixpunkt (`gfp`) von *searchPrime* mithilfe der Funktion `fixpt`.

Lösungsvorschlag

```
-- Aufgabe 10.2 a)
searchPrime :: Mod10 -> Mod10
searchPrime x
  | x `elem` [1,2,3,5,7] = x
  | x > 4 = x - 1
  | x < 5 = x + 1

-- Aufgabe 10.2 b)
lfp :: Mod10
lfp = fixpt (<=) searchPrime minBound

-- Aufgabe 10.2 c)
gfp :: Mod10
gfp = fixpt (>=) searchPrime maxBound
```

Aufgabe 10.3 (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.

Zeigen Sie, dass folgende Rekursionsgleichungen eine Funktion definieren. Dazu wird eine Schrittfunktion Φ benötigt (analog zu Folie 168). Mithilfe der Funktion `fix` lässt sich dann der kleinste Fixpunkt für die Schrittfunktion generieren. Definieren Sie in Haskell die Schrittfunktion `phi` so, dass sich `fix phi` wie die zugehörige Rekursionsgleichung verhält.

a) `sumFix = fix phi` soll sich verhalten wie:

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x:xs) = x + sum xs
```

b) `reverseFix = fix phi` soll sich verhalten wie:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (a:s) = reverse s ++ [a]
```

c) `dropFix = fix phi` soll sich verhalten wie:

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

Lösungsvorschlag

```
-- Aufgabe 10.3 a)
sumFix :: Num a => [a] -> a
sumFix = fix phi where
  phi :: Num a => ([a] -> a) -> ([a] -> a)
  phi _ []     = 0
  phi f (a:s)  = a + f s

-- Aufgabe 10.3 b)
reverseFix :: [a] -> [a]
reverseFix = fix phi where
  phi :: ([a] -> [a]) -> ([a] -> [a])
  phi _ [] = []
  phi f (a:s) = f s ++ [a]

-- Aufgabe 10.3 b)
dropFix :: Int -> [a] -> [a]
dropFix = fix phi where
  phi :: (Int -> [a] -> [a]) -> (Int -> [a] -> [a])
  phi _ 0 s = s
  phi f n (_:s) = f (n-1) s
  phi _ _ [] = []
```

Aufgabe 10.4 (8 Punkte) *Graphen*
Definieren Sie folgende Haskell-Funktionen.

Hineweis: Die Typnamen und Funktionsnamen im Painter-Paket unterscheiden sich von den Namen auf den Folien.

a) `reverseGraph :: Ord a => Graph' a -> Graph' a` – Dreht alle Kanten in einem Graph um.
Beispiel: `reverseGraph graph1' ↪`

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

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

c) `undirected :: Ord a => Graph' a -> Graph' a` – macht aus einem Graphen einen ungerichteten Graphen. Ein ungerichteter Graph lässt sich als gerichteter Graph darstellen, indem für jede Kante eine Kante in die Gegenrichtung eingefügt wird.
Beispiel: `undirected graph1'` ⇝

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

d) `breadthFirst :: Ord 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 Breitensuche aus.
Beispiel: `breadthFirst 6 graph1'` ⇝ `[6,2,4,5,1,3]`

Lösungsvorschlag

```
-- Aufgabe 10.4 a)
reverseGraph :: Ord a => Graph' a -> Graph' a
reverseGraph = rel2Graph' . map swap . graph2Rel'

-- Aufgabe 10.4 b)
isCyclic :: Ord a => Graph' a -> Bool
isCyclic graph = Map.foldlWithKey hasLoop False closure where
  closure = warshall' graph
  hasLoop b node nodes = b || node `elem` nodes

-- Aufgabe 10.4 c)
undirected :: Ord a => Graph' a -> Graph' a
undirected graph = Map.mapWithKey upd graph where
  nodes = Map.keys graph
  upd n sucs = sucs `union` [n' | n' <- nodes , n `elem` apply graph n']

-- Aufgabe 10.4 d)
breadthFirst :: Ord a => a -> Graph' a -> [a]
breadthFirst start graph = bf [start] [] where
  bf (a:as) visited
    | a `elem` visited = bf as visited
    | otherwise = bf (as ++ apply graph a) (visited ++ [a])
  bf [] visited = visited
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 11

Das Übungsblatt behandelt Themen bis einschließlich Folie 205.

Aufgabe 11.1 (4 Punkte) *Kinds*
Bestimmen Sie den Kind der angegebenen Typkonstruktoren.

a) `data T f g a = T (f a -> g a f -> a)`

Bestimmen Sie den Kind von `g`.

b)
```
class C f where
  type F f :: * -> *
  fun :: f a -> F f a
```

Bestimmen Sie den Kind von `F`.

Lösungsvorschlag

```
-- Aufgabe 11.1 a)
kind_g = "* -> (* -> *) -> *"

-- Aufgabe 11.1 b)
kind_F = "(* -> *) -> * -> *"
```

Aufgabe 11.2 (12 Punkte) *Typfamilien*
Gegeben sei folgende Typklasse:

```
class Trav t where
  type Index t :: *
  trav :: t a -> Index t -> Maybe a
```

Die Funktion `trav` durchläuft einen Container und gibt den Inhalt des gegebenen Indexes aus. Existiert der Index nicht, wird `Nothing` ausgegeben. Zum Beispiel könnte `Tree` als absturzsichere Variante von `nodes` implementiert werden.

```
instance Trav Tree where
  type Index Tree = Node
  trav t [] = Just $ root t
  trav (F _ ts) (i:node) | i `elem` indices ts = trav (ts !! i) node
  trav _ _ = Nothing
```

Instanziieren Sie `Trav` für folgende Datentypen.

a) `[a]` – Gibt das Element am Index aus.
   Beispiel: `trav [3,2,4] 1` $\rightsquigarrow$ `Just 2`

b) `Map k a` – Gibt das Element zu angegebenem Schlüssel aus.
   Beispiel: `trav (Map.fromList [("fst", 3), ("snd", 5)]) "snd"` $\rightsquigarrow$ `Just 5`

c) `data TreeL label a = TreeL a (Map label (TreeL label a))`

   Ein Datentyp für Bäume mit Kantenmarkierungen. Knoten können durch Listen von Kantenmarkierungen (`[String]`) beschrieben werden, anstelle von Listen von Ganzzahlen (`Node`).
   Beispiel: `trav treeL1 ["right","deep"]` $\rightsquigarrow$ `Just 6`

Lösungsvorschlag

```
-- Aufgabe 11.2 a)
instance Trav [] where
  type Index [] = Int
  trav (a:_) 0 = Just a
  trav (_:s) n | n > 0 = trav s (n-1)
  trav _ _ = Nothing

-- Aufgabe 11.2 b)
instance Ord k => Trav (Map k) where
  type Index (Map k) = k
  trav = (Map.!?)

-- Aufgabe 11.2 c)
instance Ord label => Trav (TreeL label) where
  type Index (TreeL label) = [label]
  trav (TreeL a _) [] = Just a
  trav (TreeL _ ts) (l:ls) = case Map.lookup l ts of
    Just t -> trav t ls
    Nothing -> Nothing
```

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

a) Für Ausdrücke von natürlichen Zahlen erweitert um Variablen.

```
data NatE x = NVar x | Z | S (NatE x)
  | NatE x :+: NatE x | NatE x :*: NatE x
```

Beispiel: `fmap store $ NVar "x" :*: NVar "y" :+: NVar "z"` ⇝ `3*10+5`

b) Für Bäume mit Kantenmarkierungen.

```
data TreeL label a = TreeL a (Map label (TreeL label a))
```

Beispiel: `fmap (fromEnum . even) treeL1`
⇝ `0{"left"->1{},"middle"->1{},"right"->0{"deep"->1{}}}`

Lösungsvorschlag

```
-- Aufgabe 11.3 a)
instance Functor NatE where
  fmap f (NVar x) = NVar (f x)
  fmap _ Z = Z
  fmap f (S n) = S (fmap f n)
  fmap f (n :+: m) = fmap f n :+: fmap f m
  fmap f (n :*: m) = fmap f n :*: fmap f m

-- Aufgabe 11.3 b)
instance Functor (TreeL label) where
  fmap f (TreeL a ts) = TreeL (f a) ((fmap . fmap) f ts)
```

technische universität dortmund

FLDIT

Prof. Dr. Peter Padawitz

Lehrstuhl 1
Logik in der Informatik

Jos Kusiek (*jos.kusiek@tu-dortmund.de*) Wintersemester 2019/2020

Übungen zu Funktionaler Programmierung
Beispiellösung 12

Das Übungsblatt behandelt Themen bis einschließlich Folie 249.

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

```
solutions :: [(Int, Int, Int)]
solutions = [ (x,y,z) | y <- [0..], x <- [0..y]
            , z <- [0..y] , 3*x^2 + 10*z + 7 == y ]
```

a) Überführen Sie die Listenkomprehension in die do-Notation (Folie 219).

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

Lösungsvorschlag

```
-- Aufgabe 12.1 a)
solutionsDo :: [(Int, Int, Int)]
solutionsDo = do
  y <- [0..]
  x <- [0..y]
  z <- [0..y]
  guard (3*x^2 + 10*z + 7 == y)
  return (x,y,z)

-- Aufgabe 12.1 b)
solutionsBind :: [(Int, Int, Int)]
solutionsBind =
  [0..] >>= \y ->
  [0..y] >>= \x ->
  [0..y] >>= \z ->
  guard (3*x^2 + 10*z + 7 == y) >>
  return (x,y,z)
```

Aufgabe 12.2 (4 Punkte) *Monaden*
Verallgemeinern Sie folgende Funktionen zu monadischen Funktionen und geben Sie einen möglichst allgemeinen Typ an.

a) Schreiben Sie eine Funktion `madd`, welche zwei Werte mit `(+)` addiert. Die Funktion soll sowohl total, partiell als auch nicht-deterministisch genutzt werden können (`Id`, `Maybe` und `[]`).
Beispiele:
```
Id 5  madd  Id 7   Id 12
Just 5  madd  Just 7   Just 12
Nothing  madd  Just 7   Nothing
[5,10]  madd  [7]   [12,17]
```

b) Schreiben Sie eine absturzsichere Funktion `mdiv`, welche zwei Werte mit `div` dividiert. Sie soll sowohl deterministisch als auch nicht-deterministisch genutzt werden können (`Maybe` und `[]`) und für eine Divisionen durch Null keine Werte zurückgeben.
Beispiele:
```
Just 12  mdiv  Just 2   Just 6
Just 12  mdiv  Just 0   Nothing
[12,6]  mdiv  [3,2]   [4,6,2,3]
[12,6]  mdiv  [3,0]   [4,2]
```

Lösungsvorschlag

```
-- Aufgabe 12.2 a)
madd :: (Monad m, Num a) => m a -> m a -> m a
madd = liftM2 (+)

-- Aufgabe 12.2 b)
mdiv :: (MonadPlus m, Integral a) => m a -> m a -> m a
mdiv m1 m2 = do
  x <- m1
  y <- m2
  guard $ y /= 0
  return $ x  div  y
```

Aufgabe 12.3 (12 Punkte) *Zustandsmonade*
Gegeben sei folgender Datentyp zur Repräsentation von arithmetischen und booleschen Variablenbelegungen.

```
type MStore x = (Map x Int, Map x Bool)
```

Implementieren Sie folgende Funktionen, um eine zustandsbasierte While-Sprache zu simulieren.

a) `returnE :: Ord x => Exp x -> State (MStore x) Int`
`returnB :: Ord x => BExp x -> State (MStore x) Bool`

Interpretieren arithmetische bzw. boolesche Ausdrücke und greifen dabei auf die Belegung im Zustande `MStore` zu. (2 Punkte)
Hinweis: Nutzen Sie die bereits definierten Interpretationsfunktionen.
Beispielprogramm: `returnE $ Var "x" :- Con 1`

b) `(?=) :: Ord x => x -> Exp x -> State (MStore x) ()` – Speichert die Auswertung eines arithmetischen Ausdruckes in einer Variablen. (2 Punkte)
Beispielprogramm: `"x" ?= 10 :* Var "y"`

c) `ite :: Ord x => BExp x -> State (MStore x) ()`
`-> State (MStore x) () -> State (MStore x) ()`

Simuliert einen `if_then_else` Ausdruck. Wird der boolesche Ausdruck zu `True` ausgewertet, dann wird die erste Zustandsmonade ausgeführt. Andernfalls die Zweite. (2 Punkte)
Beispielprogramm:

```
ite (Var "x" := Con 10)("x" ?= Con 100)("x" ?= Con 5)
```

d) `while :: Ord x => BExp x -> State (MStore x) ()`
`-> State (MStore x) ()`

Simuliert eine `while`-Schleife. So lange der boolesche Ausdruck zu `True` ausgewertet wird, wird die Zustandsmonade wiederholt ausgeführt. (2 Punkte)
Beispielprogramm:

```
while (Not (Var "x" := Con 10)) $ "x" ?= Var "x" :- Con 1
```

e) Simulieren Sie folgendes Pseudo-Java-Programm.

```
int f(boolean b, int i) {
  if b {
    int countdown = i;
    while (0 <= countdown) {
      i = i + i;
      countdown = countdown - 1;
    }
  } else {
    i = i * 10;
  }
  return i;
}
```

Implementieren Sie zuerste den Rumpf als `fBody :: State (MStore String) Int`. Danach kann die Funktion `f :: (Bool,Int) -> Int` implementiert werden, welche die Zustandsmonade `fBody` mit einer entsprechenden Variablenbelegung für `b` und `i` als Startwert aufruft. (4 Punkte)
Beispiele: `f(False,3)  30`
`f(True,3)  48`

Lösungsvorschlag

```
-- Aufgabe 12.3 a)
returnE :: Ord x => Exp x -> State (MStore x) Int
returnE e = State $ \(store,bstore)
  -> (eval e ((Map.!) store), (store,bstore))

returnB :: Ord x => BExp x -> State (MStore x) Bool
returnB e = State $ \(store,bstore)
  -> (evalB e ((Map.!) bstore ) ((Map.!) store), (store,bstore))

-- Aufgabe 12.3 b)
(?=) :: Ord x => x -> Exp x -> State (MStore x) ()
x ?= e = do
  i <- returnE e
  State $ \(store, bstore)
    -> ((),(Map.insert x i store, bstore))

-- Aufgabe 12.3 d)
ite :: Ord x => BExp x -> State (MStore x) ()
  -> State (MStore x) () -> State (MStore x) ()
ite be thn els = do
  b <- returnB be
  if b then thn else els

-- Aufgabe 12.3 e)
while :: Ord x => BExp x -> State (MStore x) ()
  -> State (MStore x) ()
while be body = do
  b <- returnB be
  when b $ do
    body
    while be body

-- Aufgabe 12.3 f)
fBody :: State (MStore String) Int
fBody = do
  ite (BVar "b")
    ( do
      "countdown" ?= Var "i"
      while (Con 0 :<= Var "countdown") $ do
        "i" ?= Sum [Var "i", Var "i"]
        "countdown" ?= Var "countdown" :- Con 1
    )
    ( "i" ?= Prod [Var "i", Con 10] )
  returnE $ Var "i"

f :: (Bool,Int) -> Int
f(b,i)=fst $ runS fBody (Map.singleton "i" i,Map.singleton "b" b)
```

Aufgabe 12.4 (4 Punkte) *IO-Monade*
Legen Sie eine Datei mit dem Namen `palindrom.hs` an. Die Datei soll nur die Funktion `main :: IO ()` enthalten, welches Konsoleneingaben ausliest und prüft, ob es sich um Palindrome handelt. Das Programm soll interaktiv sein und endlos laufen. Es kann durch die Eingabe `:exit` abgebrochen werden. Ein Ablauf soll wie folgt aussehen:

```
Eingabe:
hallo
hallo ist kein Palindrom
Eingabe:
rentner
rentner ist ein Palindrom
Eingabe:
:exit
```

Übersetzen Sie das Programm mit dem Befehl

```
ghc palindrom.hs
```

in eine ausführbare Datei und führen Sie diese aus.

Lösungsvorschlag

```
-- Aufgabe 12.4
import Control.Monad
import Data.Char

main :: IO ()
main = do
  putStrLn "Eingabe:"
  input <- getLine
  when (input /= ":exit") $ do
    putStrLn $ input ++ if isPalindrom (map toLower input)
      then " ist ein Palindrom"
      else " ist kein Palindrom"
    main

isPalindrom :: String -> Bool
isPalindrom str = str == reverse str
```