Programmierung und Modellierung mit Haskell

Teil 9: Rein Funktionale Ein-/Ausgabe, Applikative Funktoren und Monaden

Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

13. Juni 2018

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 9: Rein Funktionale Ein-/Ausgabe, Applikative Funktoren und Monaden

Dinge die wir jetzt wieder brauchen werden:

- **Referenzielle Transparenz**: keine Seiteneffekte, da Variablen nie ihren Wert ändern Folie 1.17
- **Unit-Typ** `()` ist ein Typ mit nur einem Element, welches ebenfalls `()` geschrieben wird – das leere Tupel Folie 1.38
  ```
  data () = ()
  ```
- Anonyme Funktionsdefinition `\x -> x+1` Folie 1.44
- Begriffe **Typkonstruktors** und **Kind** Folien 4.23ff./7.11
  z.B.: `Maybe Int` ist ein Typ, `Maybe` ist ein Typkonstruktor; `Either` ist ein Typkonstruktor mit Kind `* -> * -> *`
- Typklasse `Functor` mit `fmap` und `(<$>)` 7.13

Hello World

Simples “Hello World” in Haskell:

```
main = putStrLn "Hello World!"
```

Dieses Haskell Programm gibt bei Ausführung den String `"Hello World!"` auf dem Bildschirm aus und beendet sich dann.

```
> ghc helloworld.hs
[1 of 1] Compiling Main (helloworld.hs, helloworld.o)
Linking helloworld ...
> ./helloworld
Hello World
```

Hello World V2.0

Ein "merkwürdiges" Programm:

```
main = do
  putStrLn "Wie heisst Du? "
  name <- getLine                          -- Benutzerabfrage
  putStrLn ("Hallo " ++ name ++ "!")
```

Warum ist dieses Programm merkwürdig?

Hello World V2.0

Ein "merkwürdiges" Programm:

```
main = do
  putStrLn "Wie heisst Du? "
  name <- getLine
  putStrLn ("Hallo " ++ name ++ "!")
```

Seiteneffekt:
Alle Auswirkungen der Programmausführung, abgesehen vom Ergebnis.

Warum ist dieses Programm merkwürdig?

- Bildschirmausgabe ist ein Seiteneffekt!
  Eingabe ist ein Seiteneffekt!
  $\Rightarrow$ Wie passt das zur Philosophie *"keine Seiteneffekte"*?
- Wie kann `getLine` immer andere Werte zurückliefern?
  Referenzielle Transparenz (siehe Folie 1.17) erzwingt doch, dass eine Funktion für gleiche Argumente immer das gleiche Ergebnis liefert?!

I/O vs. Funktional Programmierung

Rein Funktionale Welt:

Haskell Funktionen sind rein funktional – also keine Seiteneffekte.
⇒ Aufruf rein funktionalen Codes löscht niemals die Festplatte.
⇒ Funktionen liefern das gleiche Ergebnis für gleiche Argumente.
⇒ Haskell Code ist leichter zu verstehen und sehr modular.

versus

Input/Output:

Input/Output besteht nur aus Seiteneffekten!

- Eine Funktion, welche einen String auf den Bildschirm ausgibt hat nur den Seiteneffekt, den Bildschirm zu verändern.
- Schlimmer: Funktionen, welche von Tastatur einlesen, haben keine Argumente und liefert oft andere Ergebnisse!
- Noch schlimmer:
  I/O-Funktionen sollen die Festplatte löschen können!

Haskell's I/O

Haskell versucht nicht das Rad neu zu erfinden

Input/Output ist am einfachsten im imperativen Stil durchzuführen, auch in Haskell.

Haskell verrät seine Grundprinzipien dennoch nicht

Bei allem schmutzigen ein/aus bleibt Haskell trotzdem rein!

Haskell macht es mit Monade

Die Lösung liegt im Konzept der **Monade**. $\Rightarrow$ Typklasse `Monad`

Dieses simple aber abstrakte Konzept finden manche schwierig zu verstehen, weshalb wir uns zuerst an die Verwendung von Monaden im Spezialfall der **IO**-Monade gewöhnen, mithilfe der einfachen Sichtweise "Zustandsmonade".

Nur Analogie, IO ist keine echte Zustandsmonade!

Monaden kapseln Seiteneffekte

Viele funktionale Sprachen sind nicht rein funktional und kennen I/O Befehle mit echten Seiteneffekten. Haskell ging mit Monaden einen Sonderweg, um rein funktional zu bleiben.

Zur Vereinfachung des Umgangs mit Monaden bietet Haskell eine spezielle Syntax (DO-Notation). Diese Syntax ist bequem, aber nicht notwendig! Man kann daher problemlos das Konzept der **Monade in anderen Programmiersprachen einsetzen**, auch wenn diese keine besondere Syntax dafür kennen.

Viele Sprachen haben inzwischen Monaden-Bibliotheken

Grundidee: **Monaden kapseln Seiteneffekte**

- Mögliche Seiteneffekte sind an der Typsignatur ablesbar
- Verschiedene Arten von Seiteneffekten werden unterschieden
- Reihenfolge der Seiteneffekte kontrollierbar $\Rightarrow$ Parallelität

Haskell verändert die Welt

Analogie

Damit rein funktionale Funktionen die Welt verändern können, muss die Welt als Argument übergeben werden! Die veränderte Welt wird als Ergebnis mit zurückgegeben.

```
type IO a = Welt -> (a,Welt)     -- Nur Analogie zur Vorstellung.
                                 -- Typsynonym expandiert:
putStrLn :: String -> IO ()      -- String -> Welt -> ((),Welt)
getLine  ::           IO String  --       Welt -> (String,Welt)
```

Bemerke:

- Die Ergebnis-Welt der einen Funktion muss als Argument an die nächste Welt-verändernde Funktion weitergereicht werden!
- "Staffellauf" der Welt erzwingt Reihenfolge der I/O-Aufrufe.

Hinweis:

Der Typkonstruktor `IO` ist nicht wirklich so definiert, aber wir können es uns so vorstellen!

Je ein Typ pro Seiteneffekt-Art

So einen "Staffellauf" haben wir schon in H5-2 (Data.Map) und H5-3 (Warteschlange) gesehen:

```
data Warteschlange a = -- Definition heute unwichtig
abholen :: Warteschlange a -> (Maybe a, Warteschlange a)
einstellen :: a -> Warteschlange a -> Warteschlange a
```

Auch hier erstellt uns ein Typsynonym einen *Typ für Funktionen mit Seiteneffekten auf Warteschlangen*:

```
type WarteOp a b = Warteschlange a -> (b,Warteschlange a)

abholen     ::      WarteOp a (Maybe a)
einstellen' :: a -> WarteOp a ()
einstellen  =  snd . einstellen'
```

Merke:

- Funktionstyp erklärt bereits alle möglichen Seiteneffekte!
- Aufruf von Funktion mit Seiteneffekten nennen wir **Aktion**.

Komposition von IO-Aktionen

Analogie

Um zwei Funktionen mit Seiteneffekten nacheinander auszuführen, muss aus dem Ergebnis der ersten Aktion die veränderte Welt ausgepackt und als Argument an die zweite Aktion übergeben werden. Man muss die Welt also **durchfädeln** (engl. *threading*):

```
type IO a = Welt -> (a,Welt)
nacheinander :: IO a ->       IO b  -> IO b
komposition  :: IO a -> (a -> IO b) -> IO b
```

- `nacheinander` führt zwei IO-Aktionen hintereinander aus.
- Bei `komposition` kann die zweite IO-Aktion auf das rein funktionale Ergebnis der ersten IO-Aktion noch reagieren, da dieses als Argument übergeben wird.

Hinweis:

In der Standardbibliothek tragen diese beiden Funktionen die Infix-Namen `(>>)` und `(>>=)` worauf wir später zurückkommen.

Komposition von IO-Aktionen

Analogie

Die Implementation von `nacheinander` und `komposition` ist eine einfache Übung in unserer simplen I/O-Analogie:

```
import Prelude hiding (IO)    -- Echtes IO ausblenden,
                              -- vermeidet Namenskonflikt
type Welt = Integer           -- Dummy ohne Bedeutung
type IO a = Welt -> (a,Welt) -- Nur hier als Analogie
```

Live Demonstration in der Vorlesung:

Komposition von IO-Aktionen

Analogie

Die Implementation von `nacheinander` und `komposition` ist eine einfache Übung in unserer simplen I/O-Analogie:

```
import Prelude hiding (IO)    -- Echtes IO ausblenden,
                              -- vermeidet Namenskonflikt
type Welt = Integer           -- Dummy ohne Bedeutung
type IO a = Welt -> (a,Welt) -- Nur hier als Analogie
```

Live Demonstration in der Vorlesung:

```
komposition  :: IO a -> (a -> IO b) -> IO b
komposition  = undefined

nacheinander :: IO a ->       IO b  -> IO b
nacheinander = undefined
```

Komposition von IO-Aktionen

Analogie

Die Implementation von `nacheinander` und `komposition` ist eine einfache Übung in unserer simplen I/O-Analogie:

```
import Prelude hiding (IO)    -- Echtes IO ausblenden,
                              -- vermeidet Namenskonflikt
type Welt = Integer           -- Dummy ohne Bedeutung
type IO a = Welt -> (a,Welt) -- Nur hier als Analogie
```

Live Demonstration in der Vorlesung:

```
komposition  :: IO a -> (a -> IO b) -> IO b
komposition  :: (Welt->(a,Welt)) ->
                (a ->Welt->(b,Welt)) ->
                Welt->(b,Welt)
nacheinander :: IO a ->       IO b  -> IO b
nacheinander :: (Welt->(a,Welt)) ->
                (Welt->(b,Welt)) -> Welt -> (b,Welt)
```

Komposition von IO-Aktionen

Analogie

Die Implementation von `nacheinander` und `komposition` ist eine einfache Übung in unserer simplen I/O-Analogie:

```
import Prelude hiding (IO)    -- Echtes IO ausblenden,
                              -- vermeidet Namenskonflikt
type Welt = Integer           -- Dummy ohne Bedeutung
type IO a = Welt -> (a,Welt) -- Nur hier als Analogie
```

Live Demonstration in der Vorlesung:

```
komposition  :: IO a -> (a -> IO b) -> IO b
komposition f g welt1 = let (x,welt2) = f   welt1 in
                        let (y,welt3) = g x welt2 in
                        (y,welt3)
```

Komposition von IO-Aktionen

Analogie

Die Implementation von `nacheinander` und `komposition` ist eine einfache Übung in unserer simplen I/O-Analogie:

```
import Prelude hiding (IO)    -- Echtes IO ausblenden,
                              -- vermeidet Namenskonflikt
type Welt = Integer           -- Dummy ohne Bedeutung
type IO a = Welt -> (a,Welt) -- Nur hier als Analogie
```

Live Demonstration in der Vorlesung:

```
komposition  :: IO a -> (a -> IO b) -> IO b
komposition f g welt1 = let (x,welt2) = f   welt1 in
                        let (y,welt3) = g x welt2 in
                        (y,welt3)

nacheinander :: IO a ->       IO b  -> IO b
nacheinander f g = komposition f (\_ -> g)
```

Komposition von IO-Aktionen

Analogie

Die Implementation von `nacheinander` und `komposition` ist eine einfache Übung in unserer simplen I/O-Analogie:

```
import Prelude hiding (IO)    -- Echtes IO ausblenden,
                              -- vermeidet Namenskonflikt
type Welt = Integer           -- Dummy ohne Bedeutung
type IO a = Welt -> (a,Welt) -- Nur hier als Analogie
```

Live Demonstration in der Vorlesung:

```
komposition  :: IO a -> (a -> IO b) -> IO b
komposition f g welt1 = let (x,welt2) = f   welt1 in
                        let (y,welt3) = g x welt2 in
                        (y,welt3)

nacheinander :: IO a ->       IO b  -> IO b
nacheinander f g = komposition f (const g)
```

DO-Notation

Um mehrere Aktionen hintereinander auszuführen, kennt Haskell die **DO-Notation**, welche sich implizit ums Fädeln kümmert:

```
hello1 :: IO ()
hello1 = do
           putStr   "Hallo "
           putStrLn "Welt!"
```

- In jeder Zeile darf eine **Aktion** stehen.
  Eine Aktion ist ein Ausdruck des Typs `IO a`
- Typparameter für `IO` darf in jeder Zeile anders sein, z.B. `IO String`, `IO Int`, `IO Bool`,...
- Es gilt wieder die *Layout-Regel*: Erstes Zeichen nach `do` legt die Spalte fest, in der alle IO-Aktionen beginnen müssen. Steht etwas weiter links, ist der DO-Block beendet.
- Gesamter DO-Ausdruck hat den Typ des letzten Ausdrucks.

DO-Notation (2)

Alle IO-Aktionen haben ein Ergebnis (neben der veränderten Welt):
IO-Aktion des Typs `IO a` liefert Wert des Typs `a` als Ergebnis.

Das Ergebnis können wir mit dem Rückpfeil `<-` matchen:

wie bei Pattern-Guards oder List-Comprehensions

```
hello2 :: IO String
hello2 = do
  ()   <- putStrLn "Ihr Name bitte: "
  name <- getLine
  _    <- putStrLn ("Sie heissen " ++ name ++ "?")
  getLine
```

Wenn uns das Ergebnis der IO-Aktion egal ist, dann können wir `_ <-` auch einfach komplett weglassen.

Bei der letzten Aktion dürfen wir niemals einen Rückpfeil verwenden, denn die letzte Aktion ist das Ergebnis des `do`-Blocks!

DO-Notation (3)

Falls eine IO-Aktion Argumente verlangt, z.B.

```
putStrLn :: String -> IO ()
```

so können wir diese normal funktional bearbeiten:

```
hello3 :: IO String
hello3 =
  do  putStrLn "Ihr Name bitte: "
      name <- getLine
      putStrLn $ "Sie heissen "++name++" ?"++sicher
      getLine
  where
    sicher = '\n':"Sind Sie sich sicher?"
```

- Ein DO-Block ist wirklich nur *ein Ausdruck* des Typs `IO a`
- `'\n' :: Char` ist das Zeichen für Zeilchenvorschub

DO-Notation (3)

Falls eine IO-Aktion Argumente verlangt, z.B.

```
putStrLn :: String -> IO ()
```

so können wir diese normal funktional bearbeiten:

```
hello4 :: IO String
hello4 = let sicher = "\nSind Sie sich sicher?" in
  do  putStrLn "Ihr Name bitte: "
      name <- getLine
      putStrLn ( "Sie heissen "++name++" ?"++sicher )
      getLine
```

- Ein DO-Block ist wirklich nur *ein Ausdruck* des Typs `IO a`
- `'\n' :: Char` ist das Zeichen für Zeilchenvorschub

DO-Notation (4)

Mit `let` können wir rein funktionale Abkürzungen definieren, wie wir das von List-Comprehensions schon kennen:

```
hello5 :: IO String
hello5 = do let begrüssung = "Ihr Name bitte: "
            putStrLn begrüssung
            name <- getLine
            let sicher = "\nSind Sie sich sicher?"
            let frage  = "Sie heissen also "
                         ++ (map Data.Char.toUpper name)
                         ++ " ?" ++ sicher
            putStrLn frage
            getLine
```

- `let` ist weiterhin durch Layout (Einrückung) begrenzt
- Sowohl `let` als auch `<-` können Pattern-Matching verwenden
- Bezeichner eines `let`-Block sind wieder wechselseitig rekursiv

Also alles so, wie wir es bereits sonst schon kennengelernt haben!

DO-Notation (4)

Mit `let` können wir rein funktionale Abkürzungen definieren, wie wir das von List-Comprehensions schon kennen:

```
hello5 :: IO String
hello5 = do let begrüssung = "Ihr Name bitte: "
            putStrLn begrüssung
            name <- getLine
            let frage  = "Sie also heissen "
                         ++ (map Data.Char.toUpper name)
                         ++ " ?" ++ sicher
                sicher = "\nSind Sie sich sicher?"
            putStrLn frage
            getLine
```

- `let` ist weiterhin durch Layout (Einrückung) begrenzt
- Sowohl `let` als auch `<-` können Pattern-Matching verwenden
- Bezeichner eines `let`-Block sind wieder wechselseitig rekursiv

Also alles so, wie wir es bereits sonst schon kennengelernt haben!

DO-Notation komponiert lediglich

Die DO-Notation fasst mehrere IO-Aktionen zu einer einzigen IO-Aktionen zusammen.

Erreicht wird dies durch Einfügen von `nacheinander` und `komposition` aus Folie 9.11. In der Haskell Standardbibliothek sind diese als Infix-Funktionen mit folgendem Namen definiert:

```
(>>)  :: IO a ->        IO b  -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Die DO-Notation kümmert sich lediglich ums Fädeln der Welt!

Hinweis:
DO-Notation für *alle Instanzen* der Typklasse `Monad` verwendbar, nicht nur für `IO`

DO-Notation ist "Syntaktischer Zucker"

Mithilfe dieser beiden Infix-Funktionen zur Komposition von Aktionen wird ein DO-Block zu normalen Haskell Code

```
(>>)  :: IO a ->        IO b  -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Beispiel

```
main = do
    putStrLn "Wie heisst Du? "
    name <- getLine
    putStrLn ("Hallo " ++ name ++ "!")
```

wird automatisch übersetzt zu

```
main =
    putStrLn "Wie heisst Du? "  >>
    getLine   >>=  \name ->
    putStrLn ("Hallo " ++ name ++ "!")
```

DO-Notation ist "Syntaktischer Zucker"

Mithilfe dieser beiden Infix-Funktionen zur Komposition von Aktionen wird ein DO-Block zu normalen Haskell Code

```
(>>)  :: IO a ->        IO b  -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Beispiel

```
main = do
    putStrLn "Wie heisst Du? "
    name <- getLine
    putStrLn ("Hallo " ++ name ++ "!")
```

wird automatisch übersetzt zu

```
main =
  ((putStrLn "Wie heisst Du? ") >>
   (getLine)) >>= (\name ->
   (putStrLn ("Hallo " ++ name ++ "!")) )
```

Main Methode

Bisher haben wir Haskell nur im Interpreter laufen lassen. Wenn wir eine ausführbare Datei erstellen wollen, dann braucht unser Haskell Programm eine Funktion mit dem Namen `main`:
Datei `helloworld.hs` enthält nur die eine Zeile:

```
main = putStrLn "Hello World!"
```

```
> ghc helloworld.hs
[1 of 1] Compiling Main    ( helloworld.hs, helloworld.o )
Linking helloworld ...

> ./helloworld
Hello World!
```

Hinweis: Besteht ein Programm aus mehreren Modulen, so werden normalerweise auch diese in der passenden Reihenfolge kompiliert. Standard GHC Option `--make`

Main Methode

Die `main` Funktion hat meist den Typ `IO ()`

Natürlich können wir auch in der Funktion `main` mehrere IO-Aktionen durchführen:

```
main = do
  putStrLn "Psst! Wie heisst Du?"
  name <- getLine
  putStrLn $ "Hey " ++ (map Data.Char.toUpper) name

fgut :: String -> IO ()
fgut name = putStrLn $ name ++ " gefällt mir gut!"
```

`main` beginnt den "Staffellauf" der `IO`-Monade. Die IO-Aktionen in `fgut` sind ohne Wirkung, da `main` die Funktion `fgut` nicht aufruft!

Main Methode

Wenn `main` aber `fgut` aufruft, dann werden dessen Aktionen an der entsprechenden Stelle ausgeführt. Natürlich kann `fgut` weitere Funktionen mit Typ `IO a` aufrufen.

```
main = do
  putStrLn "Psst! Wie heisst Du?"
  name <- getLine
  fgut name
  putStrLn $ "Hey " ++ (map Data.Char.toUpper) name

fgut :: String -> IO ()
fgut name = putStrLn $ name ++ " gefällt mir gut!"
```

In jedem Falle gilt:

Man kann am Typ einer Funktion erkennen ob diese Seiteneffekte haben kann – und wie wir später sehen werden kann der Typ auch Einschränken welche Art Seiteneffekte.

Kapselung von Effekten

Es empfiehlt sich jedoch sehr, funktionalen Code von I/O-Code so weit wie möglich zu trennen:

```
main = do
  line <- getLine
  if null line
    then putStrLn "(no input)"
    else do
      putStrLn $ reverseWords line
      main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

- DO-Ausdrücke können wie alle anderen Ausdrücke überall auftauchen, wo Ihr Typ gefragt ist.
- Auch IO-Aktionen können Rekursion nutzen

Ausgabe

- `putChar  :: Char -> IO ()`
  Gibt ein einzelnes Zeichen aus.

- `putStr   :: String -> IO ()`
  `putStrLn :: String -> IO () -- fügt '\n' ans Ende an`
  Gegeben einen String aus.
  Aufgrund der verzögerten Auswertung kann es sein, dass nur komplette Zeilen ausgegeben werden, siehe Folien 9.32 und 11.13ff.

- `print    :: Show a => a -> IO ()`
  Gibt einen Wert der Typklasse `Show` aus.
  Identisch zu `putStrLn . show`

GHCi und I/O

Der Interpreter GHCi erlaubt an der Eingabeaufforderung übrigens auch beliebige IO-Aktionen.

In der Tat wird auf das Ergebnis einer jeden Eingabe ohnehin die Funktion `print` angewendet:

```
> [1..5]
[1,2,3,4,5]
it :: [Integer]
> print [1..5]
[1,2,3,4,5]
it :: ()
```

Lediglich der Ergebnistyp wird beibehalten. `print` hat den Typ `Show a => a -> IO ()`, d.h. liefert immer nur `()` zurück.

Eingabe

- `getChar :: IO Char` Liest ein einzelnes Zeichen ein.
- `getLine :: IO String`
  Liest so lange ein, bis ein Zeilchenvorschub durch Drücken der Return-Taste erkannt wird.
- `getContents :: IO String`
  Liest alles ein, was der Benutzer jemals eingeben wird (oder bis ein Dateiende-Zeichen (Ctrl-D) erkannt wird)
- `interact :: (String -> String) -> IO ()`
  Verarbeitet den gesamten Input mit der gegebenen Funktion und gibt das Ergebnis zeilenweise aus.

getContents

Die Funktion `getContents :: IO String` liest die gesamte Benutzereingabe auf einmal ein. GHC fragt den Benutzer aber erst, wenn die nächste Zeile zur Bearbeitung wirklich benötigt wird:

```
import Data.Char

main = do
  input <- getContents
  let shorti   = shortLinesOnly input
      bigshort = map toUpper shorti
  putStr bigshort

shortLinesOnly :: String -> String
shortLinesOnly = unlines . shortfilter . lines
  where
    shortfilter = filter (\line -> length line < 11)
```

INTERACT

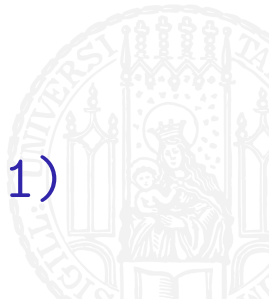
Die Funktion `interact :: (String -> String) -> IO ()` erlaubt uns, dies noch knapper auszudrücken:

```
import Data.Char

main = interact mangleinput
  where
    mangleinput = (map toUpper) . shortLinesOnly



shortLinesOnly :: String -> String
shortLinesOnly = unlines . shortfilter . lines
  where
    shortfilter = filter (\line -> length line < 11)
```



Dateizugriff

Das Modul `System.IO` bietet Varianten der IO-Funktionen für den Zugriff auf verschiedene Ein-/Ausgabegeräte an.
Die Varianten erwarten ein zusätzliche Argument des Typs `Handle`:

```
hPutStr      ::             Handle -> String -> IO ()
hPutStrLn    ::             Handle -> String -> IO ()
hPrint       :: Show a =>   Handle -> a -> IO ()
hGetLine     ::             Handle -> IO String
hGetContents ::             Handle -> IO String
```

Das Modul exportiert auch `stdin,stdout,stderr :: Handle`.
Es gilt:

```
putStr  = hPutStr stdout
getLine = hGetLine stdin
```

Datei Handles

```
readFile   :: FilePath -> IO String.
writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
```

Mit Datei Handle:

```
openFile :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()
withFile :: FilePath -> IOMode -> (Handle->IO a) -> IO a
```

- `type FilePath = String` $\Rightarrow$ Betriebssystem abhängig
- `writeFile` löscht Datei beim Öffnen
- `data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode deriving (Enum,Ord,Eq,Show,`
- `withFile` schließt die Datei in jedem Falle

Beispiele Dateizugriff:

Beispiel 1

```
import System.IO
import Data.Char (toUpper)
main = do
  contents <- readFile "whisper.txt"
  writeFile "shout.txt" (map toUpper contents)
```

Beispiel 2

```
main = do
  hIn   <- openFile "whisper.txt" ReadMode -- Öffnen
  hOut  <- openFile "shout.txt"  WriteMode
  input <- hGetContents hIn                -- Arbeiten
  let biginput = map toUpper input
  hPutStrLn hOut biginput
  hClose hIn                               -- Schliessen
  hClose hOut
```

Beispiele Dateizugriff:

Beispiel 1

```
import System.IO
import Data.Char (toUpper)
main = do
  contents <- readFile "whisper.txt"
  writeFile "shout.txt" (map toUpper contents)
```

Beispiel 2

```
main = do
  hIn   <- openFile "whisper.txt" ReadMode -- Öffnen
  hOut  <- openFile "shout.txt"  WriteMode
  input <- hGetContents hIn                -- Arbeiten
  let biginput = map toUpper input
  hPutStrLn hOut biginput
  hClose hIn                               -- Schliessen
  hClose hOut
```

Beispiele Dateizugriff:

Beispiel 3

```
main = withFile "file.txt" ReadWriteMode shout
  where
    shout file = do
      line1 <- hGetLine file
      let line2 = map toUpper line1
      hPutStrLn file line2
```

Dabei könnte `withFile` ungefähr so implementieren:

```
withFile :: FilePath -> IOMode -> (Handle->IO a) -> IO a
withFile path mode hact = do
    handle <- openFile path mode
    result <- hact handle
    hClose handle
    return result
```

Beispiele Dateizugriff:

Beispiel 3

```
main = withFile "file.txt" ReadWriteMode (\file ->
    do
      line1 <- hGetLine file
      let line2 = map toUpper line1
      hPutStrLn file line2
    )
```

Dabei könnte `withFile` ungefähr so implementieren:

```
withFile :: FilePath -> IOMode -> (Handle->IO a) -> IO a
withFile path mode hact = do
    handle <- openFile path mode
    result <- hact handle
    hClose handle
    return result
```

Verfeinerter Zugriff

Positionieren

```
hGetPosn :: Handle -> IO HandlePosn
hSetPosn :: HandlePosn -> IO ()
hSeek    :: Handle -> SeekMode -> Integer -> IO ()
```

- Nicht alle Handle-Arten unterstützen Positionierung
- Datentyp `HandlePosn` nur in Typklassen `Eq` und `Show`. Kann man sich also nur merken und wiederverwenden.

Pufferung Man kann üblicherweise auch die Pufferung beeinflussen. `hFlush` erzwingt das leeren des Puffers.

```
hSetBuffering :: Handle -> BufferMode -> IO ()
hGetBuffering :: Handle -> IO BufferMode
hFlush        :: Handle -> IO ()
```

Pufferung für Text-Konsole besser abschalten mit:

```
hSetBuffering stdout NoBuffering
```

Verfeinerter Zugriff

Positionieren

```
hGetPosn :: Handle -> IO HandlePosn
hSetPosn :: HandlePosn -> IO ()
hSeek    :: Handle -> SeekMode -> Integer -> IO ()
```

- Vorgestellte Funktionen nur für einfache I/O-Aufgaben verwenden. Bei großen Dateien bzw. Bechmarks (!) sollte Modul `Data.Bytestring` oder `Data.Text` verwendet werden. Beide Module bieten nahezu identische Funktionen wie `Data.String`

Warnung

Puffe

beeinflussen. `hFlush` erzwingt das leeren des Puffers.

```
hSetBuffering :: Handle -> BufferMode -> IO ()
hGetBuffering :: Handle -> IO BufferMode
hFlush        :: Handle -> IO ()
```

Pufferung für Text-Konsole besser abschalten mit:

```
hSetBuffering stdout NoBuffering
```

Zusammenfassung I/O

- Haskell erlaubt I/O im imperativen Stil;
  unter der Haube bleibt alles rein funktional dank “Monaden”
- IO-Aktionen verändern den Zustand der Welt, welche wie bei einem Staffellauf zwischen allen ausgeführten IO-Aktionen herumgereicht wird
- Die DO-Notation nimmt uns das explizite Fädeln der Welt ab; wird mit Aktions-Komposition zu normalen Haskell übersetzt
- IO-Aktionen sind monadisch
- DO-Notation kann für alle Monaden verwendet werden

Seiteneffekte mit Monaden

Warum Monaden?

- Haskell erlaubt prinzipiell keine Seiteneffekte.
- Monaden erlauben die Simulation von Seiteneffekten.
- Monaden kann man in jeder Sprache verwenden! Haskell's DO-Notation und Bibliotheken vereinfachen Verwendung.

Konsequenz

Jeder Seiteneffekt muss in der Typsignatur deklariert werden!

- Die `IO`-Monade erlaubt beliebige Seiteneffekte.
- Man kann leicht Monaden erstellen, welche nur sehr begrenzte Seiteneffekte erlauben
  z.B. nur Zugriff auf eine handvoll festgelegter Variablen.
- Seiteneffekte werden umso beherrschbarer, je weniger Programmcode innerhalb der `IO`-Monade ist.

Typklasse Functor


```
class Functor f where
  fmap  :: (a -> b) -> f a -> f b

  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap        -- Infix-Synonym
```

Folgende Gesetze sollten gelten:

Identität `fmap id == id`

Komposition `fmap f . fmap g == fmap (f . g)`

Der Parameter `f` der Typklasse `Functor` muss Kind `* -> *` haben, d.h. `f` ist kein Typ sondern ein Typkonstruktor mit einem „Loch“.

Beispiel: `Maybe :: * -> *` *dagegen* `Maybe Int :: *`

Die Typklasse `Functor` ist die Klasse aller Typen-in-Kontext, welche es erlauben Ihre Inhalte auf andere im gleichen Kontext abzubilden.

Beispiel: Functor Maybe


```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Eine Funktion auf ein Maybe anwenden:

```
data Maybe a = Nothing | Just a
instance Functor Maybe where
  fmap g (Just x) = Just (g x)
  fmap _ Nothing  = Nothing
```

`Nothing` bleibt `Nothing`; aber auf Inhalte von `Just` wird die gegebene Funktion angewendet und das Ergebnis wieder verpackt:

```
> fmap even (Just 42)
Just True
> fmap even Nothing
Nothing
```


Kontext: `Nothing` interpretiert als „Fehler bei Berechnung“

Mehrstellige Funktionen und Funktoren

```
> fmap (*2) (Just 21)
Just 42
> :t fmap (*) (Just 2)
fmap (*) (Just 2) :: Num a => Maybe (a -> a)
```

- Wie können wir z.B. die binäre Operation `(*)` auf zwei Werte des Typs `Maybe Int` anwenden?
- Was bedeutet ein Funktionstyp-mit-Kontext, wie hier im Beispiel `Maybe (a -> a)`?
- Wie wenden wir Funktionstypen-mit-Kontext an?

Applikative Funktoren

Modul `Control.Applicative` bietet speziellere Funktoren:

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Ein **Applikativer Funktor** erlaubt die Anwendung einer Funktion-mit-Kontext auf einen Wert-mit-Kontext!

Folgende Gesetze sollten gelten:

Identität `v == pure id <*> v`

Komposition `u <*> (v<*>w) == pure (.) <*> u <*> v <*> w`

Homomorphie `pure f <*> pure x == pure (f x)`

Interchange `u <*> pure y == pure (\f -> f y) <*> u`

Gesetze für Funktoren folgen automatisch aus diesen, falls wir definieren: `f <$> x = pure f <*> x`

Pure

Die Funktion `pure :: a -> f a` platziert einen funktionalen Wert in einen "leeren" Kontext/Seiteneffekt.

Dies ist nützlich, wenn wir eine gewöhnliche Funktion `foo :: a -> b -> c` auf Werte im Kontext `x,y :: f a` anwenden wollen:

```
pure foo <*> x <*> y :: f c
```

Wir hätten aber auch einfach `foo <$> x <*> y` schreiben können, denn jeder applikative Funktor ist dank `pure` auch ein normaler Funktor: `(<*>) . pure` entspricht genau `fmap` bzw. `(<$>)`

Zur Erinnerung:

```
pure :: a -> f a
<*>  :: f (a -> b) -> f a -> f b
<$>  ::   (a -> b) -> f a -> f b
```

Beispiel: Applicative-Instanz für Maybe

```
instance Applicative Maybe where
  pure  :: a -> Maybe a
  pure = Just
  (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  (Just f) <*> (Just x)  = Just (f x)
  Nothing  <*> _         = Nothing
```

Wenn wir keine Funktion bekommen, dann können wir auch keine Funktionsanwendung durchführen.

```
> Just (+3) <*> Just 8
Just 11
> pure (+) <*> Just 3 <*> Just 8
Just 11
> (+) <$> Just 3 <*> Just 8
Just 11
```

Ist einer der Werte `Nothing`, dann kommt `Nothing` heraus.

Beispiel: Applicative-Instanz für Maybe

```
instance Applicative Maybe where
  pure  :: a -> Maybe a
  pure = Just
  (<*>) :: Maybe (a -> b) -> Maybe a -> Maybe b
  (Just f) <*> something = f <$> something
  Nothing  <*> _         = Nothing
```

Wenn wir keine Funktion bekommen, dann können wir auch keine Funktionsanwendung durchführen.

```
> Just (+3) <*> Just 8
Just 11
> pure (+) <*> Just 3 <*> Just 8
Just 11
> (+) <$> Just 3 <*> Just 8
Just 11
```

Ist einer der Werte `Nothing`, dann kommt `Nothing` heraus.

Beispiel: Applicative-Instanz für Listen

```
instance Applicative [] where
  pure  :: a -> [a]
  pure x = [x]
  (<*>) :: [a -> b] -> [a] -> [b]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

Ist der Kontext eine Vielzahl von Möglichkeiten, dann werden einfach alle Möglichkeiten kombiniert.

```
> (++) <$> ["H","P"] <*> ["i","a"]
["Hi","Ha","Pi","Pa"]

> [(*0),(+10),(*7)] <*> [1,2,3]
[0,0,0,11,12,13,7,14,21]

> [(+),(*)] <*> [1,2] <*> [3,5]
[4,6,5,7,3,5,6,10]
```

Beispiel: Applicative-Instanz für Listen

```
instance Applicative [] where
  pure  :: a -> [a]
  pure x = [x]
  (<*>) :: [a -> b] -> [a] -> [b]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

Ist der Kontext eine Vielzahl von Möglichkeiten, dann werden einfach alle Möglichkeiten kombiniert.

```
> (++) <$> ["H","P"] <*> ["i","a"]
["Hi","Ha","Pi","Pa"]

> [(*0),(+10),(*7)] <*> [1,2,3]
[0,0,0,11,12,13,7,14,21]

> [(+),(*)] <*> [1,2] <*> [3,5]
[4,6,5,7,3,5,6,10]
```

Interpretation

Typ `[a]` ist eine nicht-deterministische Berechnung eines Wertes des Typs `a`, welche „zufällig“ einen der gelisteten Ergebnisse haben kann.

Liften

Wir können also Funktionen mit beliebiger Stelligkeit in den Kontext hieven. Das Muster ist immer die gleiche mehrfache Anwendung der links-assoziativen Infix-Funktion `<*>`:

$$\texttt{f <\$> } x_1 \texttt{ <*> } \cdots \texttt{ <*> } x_n$$

Dies bezeichnet man als **liften** einer Funktion.

Modul `Control.Applicative` definiert entsprechend:

```
liftA  :: Applicative f => (a -> b) -> f a -> f b
liftA  f a     = pure f <*> a

liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b   = f <$> a <*> b

liftA3 :: Applicative f =>
            (a -> b -> c -> d) -> f a -> f b -> f c -> f d
liftA3 f a b c = f <$> a <*> b <*> c
```

Zusammenfassung Applicative

- Applikative Funktoren erlauben die Behandlung beliebig stelliger Funktionen-im-Kontext
- Behandlung eines Kontexts wird bequem versteckt:
  Funktionsanwendung ohne Kontext `f $ x y`
  Funktionsanwendung mit Kontext `f <$> x <*> y`
- `f <$> x <*> y` ist äquivalent zu `pure f <*> x <*> y`
- Der Infix-Operator `<*>` ist links-assoziativ
- Applikative Funktoren wurden erst spät identifiziert:
  Functional Pearl: "Applicative Programming with Effects"
  von Conor McBride und Ross Paterson
  im Journal of Functional Programming 18:1 (2008)

Effekte abschütteln?

Beobachtung

Es gibt keinen allgemeinen Weg, einen einmal erhaltenen Kontext (oder auch Effekt) wieder abzuschütteln:

```
unpure :: Applicative f => f a -> a

unpure :: Maybe a -> a
unpure (Just x) = x
unpure Nothing  = undefined -- ??? GEHT NICHT !!!

unpure :: [a] -> a
unpure (x:_)    = x
unpure []       = undefined -- ??? GEHT NICHT !!!
```

Kontext muss immer berücksichtigt werden

```
iffy :: Applicative f => f Bool -> f a -> f a -> f a
iffy fb ft ff = cond <$> fb <*> ft <*> ff
  where cond b t f = if b then t else f
```

Dies hat als Konsequenz, dass applikative Funktoren immer alle Kontexte/Effekte berücksichtigen müssen:

```
> iffy (Just True) (Just 42) (Just 0)
Just 42
> iffy (Just True) (Just 42) Nothing
Nothing
```

Der Rückgabewerte einer Berechnung-im-Kontext kann die nachfolgenden Berechnungen-im-Kontext nicht beeinflussen!

Dies *kann* wünschenswert sein, aber nicht immer.

Monaden

Einen Ausweg bieten **Monaden**:
Auch hier können Kontexte/Effekte nicht abgeschüttelt werden, aber es gibt eine Operation, welche Werte kurzzeitig aus dem Kontext/Effekt herausnimmt, so dass darauf reagiert werden kann:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

```
miffy :: Monad m => m Bool -> m a -> m a -> m a
miffy mb mt mf = mb >>= condm
  where condm b = if b then mt else mf

 > miffy (Just True) (Just 42) (Nothing)
Just 42
```

Funktion `(>>=)` kennen wir schon: `komposition` von Folie 9.11

Monaden

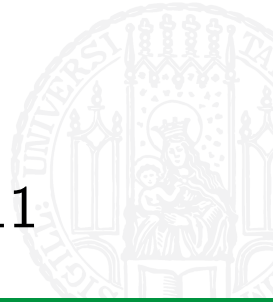
Einen Ausweg bieten **Monaden**:
Auch hier können Kontexte/Effekte nicht abgeschüttelt werden, aber es gibt eine Operation, welche Werte kurzzeitig aus dem Kontext/Effekt herausnimmt, so dass darauf reagiert werden kann:

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

```
miffy :: Monad m => m Bool -> m a -> m a -> m a
miffy mb mt mf = mb >>= condm
  where condm b = if b then mt else mf
```

```
 > miffy (Just True) (Just 42) (Nothing)
Just 42
```

Funktion `(>>=)` kennen wir schon: `komposition` von Folie 9.11



Grundidee Monaden

Monaden kann man sich grob als Container vorstellen, welche funktionale Werte mit Seiteneffekt/Zustand verpacken.

Zwei grundlegende Operationen in einer Monade:

- Komposition zweier monadischer Aktionen, d.h. zwei Aktionen werden zu einer monadischen Aktion verschmolzen; Seiteneffekt/Zustand wird zwischen Aktionen hindurchgefädelt.
- Einbettung funktionaler Werte in die Monade mit leerem Seiteneffekt bzw. ohne Zustandsänderung.

Die Typklasse `Monad` enthält wie `Functor` und `Applicative` keine Typen, sondern Typkonstruktoren mit Kind `* -> *`, d.h. wie wir sehen werden ist `Maybe` ist in der Typklasse, nicht `Maybe Int`.

Typklasse Monad aus Modul Control.Monad

```
class Applicative m => Monad m where
  return :: a -> m a                    -- pure
  (>>=)  :: m a -> (a -> m b) -> m b    -- komposition

  (>>)   :: m a ->       m b  -> m b    -- nacheinander
  x >> y = x >>= \_ -> y  -- Default aus (>>=) generiert

  fail :: String -> m a   -- Default
  fail msg = error msg
```

- `return` entspricht `pure` Umbenennung historisch
- `(>>=)` gesprochen "**bind**", ist Komposition von Aktionen
- `(>>)` ist Nacheinanderausführung von Aktionen, entspricht Komposition mit Wegwerfen des Zwischenergebnis
- `fail` erlaubt spezielle Fehlerbehandlung
- *Gesetze:* folgen auf Folie 9.51

DO-Notation für alle Monaden

DO-Notation wird für jede Instanz der Typklasse `Monad` unterstützt:

```
foo = do
  x <- action1 w
  y <- action2
  action3 x
  z <- action4 v y
  return $ bar x y z
```

wird von Haskell automatisch behandelt wie

```
foo =
  action1 w    >>= (\x ->
  action2      >>= (\y ->
  action3 x    >>
  action4 v y >>= (\z ->
  return $ bar x y z)))
```

DO-Notation vs. Applicative

Programme mit DO-Notation sehen sehr "imperativ" aus — und sind es oft auch *unnötigerweise*:

```
foo  :: Monad m => m (a -> b) -> m a -> m b
foo mf mx = do
  f <- mf
  x <- mx
  return  (f x)
```

Dieses häufig auftretende Muster sollte man besser so schreiben:

```
foo  :: Applicative m => m (a -> b) -> m a -> m b
foo mf mx = mf <*> mx
```

Kürzerer, lesbarer Code; und
beide Seiteneffekte können sich nicht gegenseitig beeinflussen!

$\Rightarrow$ Wenn Applicative ausreicht, nicht mit Monaden draufhauen!

DO-Notation vs. Applicative

Programme mit DO-Notation sehen sehr "imperativ" aus — und sind es oft auch *unnötigerweise*:

```
foo2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c
foo2 f mx my = do
  x <- mx
  y <- my
  return $ f x y
```

Dieses häufig auftretende Muster sollte man besser so schreiben:

```
foo2 :: Applicative f => (a->b->c) -> f a -> f b -> f c
foo2 f x y = f <$> x <*> y
```

Kürzerer, lesbarer Code; und
beide Seiteneffekte können sich nicht gegenseitig beeinflussen!

$\Rightarrow$ Wenn Applicative ausreicht, nicht mit Monaden draufhauen!

Monaden Gesetze

Instanzen der Typklasse `Monad` *sollten* folgenden Gesetze einhalten. Wie bei den Gesetzen der Typklassen `Functor` und `Applicative` ist der Programmierer der Instanz für die Einhaltung dieser Gesetze zuständig!

1. **Links-Identität**
   `return x >>= act` macht das Gleiche wie `act x`
2. **Rechts-Identität**
   `mval >>= return` macht das Gleiche wie `mval`
3. **Assoziativität**
   `mavl >>= (\x-> act1 x >>= act2)` macht das gleiche wie `(mval >>= act1) >>= act2`

Alles was diesen Gesetzen genügt, ist eine **Monade**.

Monaden Gesetze

Monaden Gesetze ausgedrückt unter Verwendung de DO-Notation:

1. **Links-Identität**

```
do  y <- return x        |  do  act x
    act y                |
```

2. **Rechts-Identität**

```
do  y <- mval            |  do  mval
    return y             |
```

3. **Assoziativität**

```
do  x <- mval            |  do  y <- do
    y <- act1 x          |              x <- mval
    act2 y               |              act1 x
                         |      act2 y
```

Alles was diesen Gesetzen genügt, ist eine **Monade**.

Monaden Gesetze

Monaden Gesetze ausgedrückt unter Verwendung de DO-Notation:

1. **Links-Identität**

```
do  y <- return x      |  do  act x
    act y              |
```

2. **Rechts-Identität**

```
do  y <- act x         |  do  act x
    return y           |
```

3. **Assoziativität**

```
do  x <- act0 z        |  do  y <- do
    y <- act1 x        |              x <- act0 z
    act2 y             |              act1 x
                       |          act2 y
```

Alles was diesen Gesetzen genügt, ist eine **Monade**.

Alle Monaden sind Applikative Funktoren

Alle Monaden definieren auch schon (applikative) Funktoren:

```
fmap f mx = mx >>= (\x -> return $ f x)

pure = return
mf <*> mx = mf >>= (\f ->
                mx >>= (\x -> return $ f x) )
```

- Instanzdeklaration für `Monad` plus obigen Code für Instanzdeklarationen von `Applicative` und `Functor` reicht.
- `Applicative` ist erst seit GHC 7.10 Oberklasse von `Monad`. Deshalb gibt es noch folgende redundante Definitionen:

```
fmap == liftM :: Monad m =>    (a -> b) -> m a -> m b
<*>  == ap    :: Monad m => m (a -> b) -> m a -> m b
```

- *Aber:* Nicht alle applikative Funktoren sind Monaden

Fazit: Monaden sind stärker; reicht aber ein applikativer Funktor, so ist dies die bessere Wahl (wg. Verständlichkeit & Effizenz)!

Maybe als Monade

```
data Maybe a = Nothing | Just a
```

Diesen Datentyp können wir auch zur Monade machen:

```
instance Monad Maybe where
  return x       = Just x
  Nothing >>= _  = Nothing
  Just x  >>= f  = f x

  fail _         = Nothing
```

Modelliert Berechnungen mit Seiteneffekt "Fehler":

1. "Aktion" ist Berechnung, welche Wert liefert oder fehlschlägt.
2. Wenn eine Berechnung einen Wert liefert, kann damit weiter gerechnet werden.
3. Wenn eine einzelne Berechnung fehlschlägt, so schlägt auch die gesamte Berechnung fehl.

Maybe als Monade

```
data Maybe a = Nothing | Just a
```

Diesen Datentyp können wir auch zur Monade machen:

```
instance Monad Maybe where
  return x   = Just x
  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (>>=) mx f = case mx of
                 Nothing -> Nothing
                 (Just x) -> f x
  fail _     = Nothing
```

Modelliert Berechnungen mit Seiteneffekt "Fehler":

1. "Aktion" ist Berechnung, welche Wert liefert oder fehlschlägt.
2. Wenn eine Berechnung einen Wert liefert, kann damit weiter gerechnet werden.
3. Wenn eine einzelne Berechnung fehlschlägt, so schlägt auch die gesamte Berechnung fehl.

Maybe erfüllt Monaden-Gesetze

1. Links-Identität

```
return x >>= f
 = case (return x) of Nothing -> Nothing; (Just y) -> f y
 = case (Just x) of Nothing -> Nothing; (Just y) -> f y
 = f x
```

2. Rechts-Identität

```
m >>= return
 = case m of Nothing -> Nothing; (Just y) -> return y
 = case m of Nothing -> Nothing; (Just y) -> (Just y)
 = m
```

3. Assoziativität

```
m >>= (\x-> f x >>= g)
 = ...Übung...
 = (m >>= f) >>= g
```

Maybe-Monade: Beispiel

```
maybeMult mx my =
  do  x <- mx
      y <- my
      return $ x * y

> maybeMult (Just 4) (Just 5)
Just 20
> maybeMult Nothing  (Just 5)
Nothing
> maybeMult (Just 4) Nothing
Nothing
```

Das Beispiel ist vielleicht etwas unsinnig, aber in der Praxis erspart die DO-Notation für Maybe-Monaden wiederholte pattern-matches mit `(Just x)` — wenn man denn nur das Ergebnis haben will, falls *alle* Zwischenergebnisse nicht `Nothing` waren.

Maybe-Monade: Beispiel

```
maybeMult mx my =
  do  x <- mx
      y <- my
      return $ x * y
```

```
maybeMult mx my = (*) <$> mx <*> my
```

. . . applikativer Funktor reicht hier auch!

```
> maybeMult (Just 4) (Just 5)
Just 20
> maybeMult Nothing  (Just 5)
Nothing
> maybeMult (Just 4) Nothing
Nothing
```

Das Beispiel ist vielleicht etwas unsinnig, aber in der Praxis erspart die DO-Notation für Maybe-Monaden wiederholte pattern-matches mit `(Just x)` — wenn man denn nur das Ergebnis haben will, falls *alle* Zwischenergebnisse nicht `Nothing` waren.

Listen als Monade

Auch Listen können wir als Monaden auffassen:

```
instance Monad [] where
  return x = [x]
  -- (>>=) :: [a] -> (a->[b]) -> [b]
  xs >>= f = concat (map f xs)
  fail   _ = []
```

Listen-Monade *simuliert* nichtdeterministische Berechnungen:

- Anstatt eines einzelnen Wertes wird mit einer Liste von Werten simuliert gleichzeitig gerechnet.
- Schlägt eine Berechnung fehl, so wird die Menge der möglichen Ergebniswerte für diese Berechnung leer.

Simulation: Die Berechnung bleibt wie gehabt deterministisch; es werden lediglich alle auftretenden Möglichkeiten nacheinander durchprobiert!

Listen-Monade: Beispiele (1)

Unser Input ist entweder $3, 4$ oder 5. Was kommt heraus, wenn wir darauf eine Funktion anwenden, welche entweder die Zahl negiert oder unverändert zurück gibt?

```
> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Wenn wir keinen Input bekommen, so kann auch nichts berechnet werden:

```
> [] >>= \x -> [1..5]
[]
```

Listen-Monade: Beispiele (2)

Mehrere Alternativen müssen alle miteinander kombiniert werden:

```
> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

In DO-Notation geschrieben sieht das so aus:

```
do
  n  <- [1,2]
  ch <- ['a','b']
  return (n,ch)
```

Kommt uns diese Notation nicht bekannt vor?

Listen-Monade: Beispiele (2)

List-Comprehensions sind in der Tat gleich zur DO-Notation:

```
foo1 :: [Int] -> [Int] -> [(Int,Int)]
foo1 xs ys = [(z,y)|x<-xs, x/=5, let z=x*10, y<-ys]

foo2 :: [Int] -> [Int] -> [(Int,Int)]
foo2 xs ys = do x <- xs
                unless (x/=5) (fail "Oops!")
                let z = x*10
                y <- ys
                return (z,y)

> foo1 [4..6] [7..9]
[(40,7),(40,8),(40,9),(60,7),(60,8),(60,9)]
> foo2 [4..6] [7..9]
[(40,7),(40,8),(40,9),(60,7),(60,8),(60,9)]
```

WHEN & UNLESS

Bedingte Ausführung von IO-Aktionen erlauben uns

```
when   :: Monad m => Bool -> m () -> m ()
unless :: Monad m => Bool -> m () -> m ()
```

Die Funktion `when` führt die übergebene Aktion nur dann aus, wenn das erste (funktionale) Argument zu `True` auswertet.

Bei der Funktion `unless` ist das umgekehrt. Sie führt die übergebene Aktion nur dann aus, wenn das erste (funktionale) Argument zu `False` auswertet.

FOREVER

Wir können eine IO-Aktion mit der Funktion `forever` bis zum Programmabbruch wiederholen lassen:

```
forever :: Monad m => m a -> m b
--                    IO a -> IO b
```

BEISPIEL

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

`forever` nutzt einfach nur Rekursion und DO-Notation.

SEQUENCE

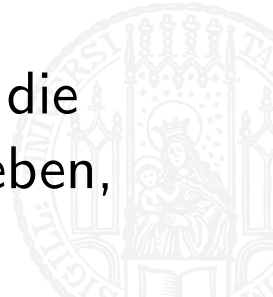
Mehrere IO-Aktionen können wir hintereinander ausführen lassen:

```
sequence  :: [IO a] -> IO [a]    -- IO nur Beispiel
sequence_ :: [IO a] -> IO ()
```

Beispiel

```
> sequence $ map print [1..5]
1
2
3
4
5
[(),(),(),(),()]
```

Wenn das aufgesammelte Ergebnis nicht interessiert, z.B. weil die verwendeten IO-Aktion wie im Beispiel immer nur `()` zurückgeben, dann können wir auch die Variante `sequence_` verwenden.



SEQUENCE

Mehrere IO-Aktionen können wir hintereinander ausführen lassen:

```
sequence  :: [IO a] -> IO [a]    -- IO nur Beispiel
sequence_ :: [IO a] -> IO ()
```

Beispiel

```
> sequence $ map print [1..5]
1
2
3
4
5
[(),(),(),(),()]
```

Hinweis: `sequence` kann man schon auf alle applikativen Funktoren anwenden.

Anstatt `IO` kann man also alle Typen der Klasse `Applicative` einsetzen.

Wenn das aufgesammelte Ergebnis nicht interessiert, z.B. weil die verwendeten IO-Aktion wie im Beispiel immer nur `()` zurückgeben, dann können wir auch die Variante `sequence_` verwenden.

MAPM

Wir können eine Sequenz von IO-Aktionen vorher auch noch transformieren lassen:

```
mapM  :: (a -> IO b) -> [a] -> IO [b]
mapM_ :: (a -> IO b) -> [a] -> IO ()
```

Dabei ist `mapM f` ist tatsächlich äquivalent zu `sequence . map f`

BEISPIEL

```
> mapM_ print [1..5]
1
2
3
4
5
()
```

Die Variante `mapM_` verwirft lediglich das Endergebnis, d.h. nur die Seiteneffekt interessiert uns.

MAPM

Wir können eine Sequenz von IO-Aktionen vorher auch noch transformieren lassen:

```
mapM  :: (a -> IO b) -> [a] -> IO [b]
mapM_ :: (a -> IO b) -> [a] -> IO ()
```

Dabei ist `mapM f` ist tatsächlich äquivalent zu `sequence . map f`

BEISPIEL

```
> mapM_ print [1..5]
1
2
3
4
5
()
```

HINWEIS: Auch `mapM` kann man schon auf alle applikativen Funktoren anwenden.
Verallgemeinerung `traverse` für alle Typen der Klasse `Traversable`.

Die Variante `mapM_` verwirft lediglich das Endergebnis, d.h. nur die Seiteneffekt interessiert uns.

Implementierung mapM

Wir implementieren in der Vorlesung als Übung auf drei Arten

```
mapM  :: Monad m => (a -> m b) -> [a] -> m [b]
```

forM

Zum Abschluß noch ein (vielleicht) alter Bekannter:

```
forM  :: [a] -> (a -> IO b) ->  IO [b]
forM_ :: [a] -> (a -> IO b) ->  IO ()
```

Beispiel

Man kann mit `forM == flip mapM` und anonymen Funktionen fremd-aussehende Programme schreiben:

```
import Control.Monad
main = do
    colors <- forM [1,2,3,4] (\a -> do
        putStrLn $ "Welche Farbe assoziierst Du mit "
                   ++ show a ++ "?"
        color <- getLine
        return color)
    putStrLn "Farbe der Zahlen 1, 2, 3 und 4 sind: "
    mapM putStrLn colors
```

Übersicht Monaden

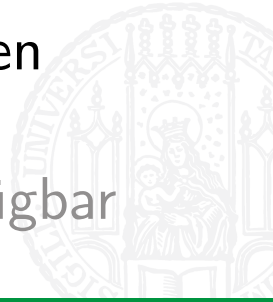
Häufige Anwendungen für Monaden:

- **I/O Monade** ermöglicht funktionale User-Interaktion.
- **Fehlermonade** für Berechnungen, welche Fehler werfen können z.B. Maybe-Monade, Either-Monade, MonadError
- **Nichtdeterminismus** für Berechnungen, welche mit mehreren Alternativen gleichzeitig rechnen z.B. Listen-Monade
- **Zustandsmonade** Berechnungen mit veränderlichen Zustand `Control.Monad.Trans.State`

  Spezialfälle:
  - **Lesemonade** liest Zustand nur aus `Control.Monad.Trans.Reader`
  - **Schreibmonade** beschreibt Zustand nur, z.B. logging `Control.Monad.Trans.Writer`

Zusammenfassung Monaden

- Monaden sind ein *Programmierschema* für zusammengesetzte abhängige Berechnungen mit Kontext oder Seiteneffekten
- Kontext/Seiteneffekte werden durch die Monade explizit ausgedrückt und deklariert, also greifbar und sichtbar gemacht
- Monaden separieren das Fädeln des Kontexts von der eigentlich Berechnung; d.h. Hintereinanderausführung mehrerer Berechnungen mit Kontext/Seiteneffekt wird vereinfacht.
- Monadenoperation müssen drei Gesetze erfüllen: *Links-/Rechtsidentität* und *Assoziativität*.
- Alle Monaden sind immer auch (applikative) Funktoren
- Monaden sind keine eingebaute Spracherweiterung; man kann Monaden auch in anderen Programmiersprachen verwenden
  GHC bietet syntaktische Unterstützung (DO-Notation)
  Monaden-Bibliotheken auch für andere Sprachen verfügbar



Zusammenfassung Do-Notation

Do-Notation erlaubt es, mehrere monadische Aktionen hintereinander auszuführen. Es werden automatisch Anwendungen von bind eingefügt, um den Kontext hindurchzufädeln.

- Do-Block gilt als eine einzelne monadische Aktion
- Gesamter Do-Block hat immer den Typ der letzten Aktion
- Die Monade aller Aktionen muss die gleiche sein
- Es gilt Layout-Regel: Pro Zeile eine monadische Aktion
  Einrückung *weiter rechts*: Zeile geht weiter
  Einrückung *weiter links*: Do-Block beendet
- `let x = expression` für rein funktionale Berechnungen im Do-Block erlaubt.
  Achtung: kein `in` im Do-Block
- `let` und `<-` erlauben Pattern-Match auf linker Seite;
  Schlägt dieser Pattern-Match fehl, so wird `fail` aufgerufen.

let versus <-

Unterschied zwischen `let` und `<-` innerhalb eines Do-Blocks:

```
01 bar :: (Monad m) => m a -> m a
02 bar ma = do
03             x <- ma      --  x ::   a
04             let y = ma   --  y :: m a
05             z <- y       --  z ::   a
06             return z
```

Zeile 04 rein funktionale Berechnung ohne Seiteneffekt

Z. 03 & 05 Seiteneffekte der monadischen Operation geschehen

Beispiel

Wenn wir `bar` mit einem Argument aufrufen, welches eine Bildschirmausgabe veranlasst, so findet diese nur in Zeilen 03 und 05 statt.

z.B. `bar (print 42)`

Häufige Muster

Applikative Funktoren nicht vergessen:

```
do x <- mx
   y <- my
   return (f x y)
```

```
f <$> mx <*> my
```

Falls zwischen den monadischen Aktionen keine Abhängigkeit besteht, d.h. hier ist `x` kein Argument für `my`

Rechts-Identität nicht vergessen:

```
do x <- mx
   return x
```

```
mx
```

Unnötiges auspacken und wieder einpacken.

DO-Notation vor allem nützlich bei unnatürlicher Reihenfolge, z.B. Ergebnis Aktion 1 wird nicht gleich für Aktion 2 gebraucht.

Häufige Muster

Applikative Funktoren nicht vergessen:

```
do x <- mx
   y <- my
   return (f x y)
```

```
f <$> mx <*> my
```

Falls zwischen den monadischen Aktionen keine Abhängigkeit besteht, d.h. hier ist `x` kein Argument für `my`

Rechts-Identität nicht vergessen:

```
do x <- mx
   y <- my x
   return y
```

```
do x <- mx
   my x
```

```
mx >>= my
```

Unnötiges auspacken und wieder einpacken. Do mit einem bind

DO-Notation vor allem nützlich bei unnatürlicher Reihenfolge, z.B. Ergebnis Aktion 1 wird nicht gleich für Aktion 2 gebraucht.