

PROGRAMMIERUNG UND MODELLIERUNG MIT HASKELL

REIN FUNKTIONALE EIN-/AUSGABE, (APPLIKATIVE) FUNKTOREN UND MONADEN

Martin Hofmann Steffen Jost

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

15. Juni 2015

DINGE DIE WIR JETZT WIEDER BRAUCHEN WERDEN:

- **Referenzielle Transparenz**: keine Seiteneffekte, da Variablen nie ihren Wert ändern Folie 01-15
- Der Unit-Typ `()` ist ein Typ mit nur einem Element, welches ebenfalls `()` geschrieben wird – das leere Tupel Folie 01-35
`data () = ()`
- Anonyme Funktionsdefinition `\x -> x+1` Folie 01-41
- Typen mit Parametern: Listen, Bäume, **Maybe**, **Either** 04-21ff.
- Begriff des **Typkonstruktors** Folie 04-18
z.B.: **Maybe Int** ist ein Typ, **Maybe** ist ein Typkonstruktor
- Definition von Typklassen und Instanzen, insbesondere **Show** und **Read** Folie 05-15ff.
- Funktion `map :: (a -> b) -> [a] -> [b]`, welche eine gegebene Funktion punktweise auf eine Liste anwendet 06-14



HELLO WORLD

Simple "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
[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           -- Benutzerabfrage
  putStrLn ("Hallo " ++ name ++ "!")
```

WARUM IST DIESES PROGRAMM MERKWÜRDIG?

- Bildschirmausgabe ist ein Seiteneffekt!
Eingabe ist ein Seiteneffekt!
⇒ Wie passt das zur Philosophie “keine Seiteneffekte”?
- Wie kann `getLine` immer andere Werte zurückliefern?
Referenzielle Transparenz (siehe Folie 01-15) erzwingt doch, dass eine Funktion für gleiche Argumente immer das gleiche Ergebnis liefert?!



I/O vs. FUNKTIONALE 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**.

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 Monaden-Sichtweise "Zustandsmonade".

Nur Analogie, IO ist keine echte Zustandsmonade!



HASKELL'S I/O

HASKELL VERSUCHT NICHT DAS RAD NEU ZU ERFINDEN:

HINWEIS

Die meisten anderen funktionalen Sprachen sind nicht rein funktional und kennen I/O Befehle mit echten Seiteneffekten.

Haskell geht mit Monaden einen Sonderweg, um rein funktional zu bleiben. Zur Vereinfachung bietet Haskell für den Umgang mit Monaden eine spezielle DO-Notation an.

Man kann aber auch in anderen Programmiersprachen Monaden zur Kapselung von Seiteneffekten einsetzen, auch wenn die Sprache dazu gar keine besondere Syntax kennt.

Monaden-Sichtweise "Zustandsmonade".

Nur Analogie, IO ist keine echte Zustandsmonade!



HASKELL VERÄNDERT DIE WELT:

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)
```

Bemerge:

- 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 A5-3 und H5-2 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 alle 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

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

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)
```

LIVE DEMONSTRATION IN DER VORLESUNG:



KOMPOSITION VON IO-AKTIONEN

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)
```

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

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)
```

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 f g = komposition f (const g)
```



DO-NOTATION

Um mehrere Aktionen hintereinander auszuführen, kennt Haskell die **DO-Notation**, welche sich implizit für uns 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):
Eine IO-Aktion des Typs `IO a` liefert einen Wert des Typs `a`

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 den Rückpfeil auch einfach weglassen.

Nur bei der letzten Aktion dürfen wir keinen Rückpfeil verwenden
— die letzte Aktion ist ja 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 Zeilenvorschub



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 Zeilenvorschub



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 also heißen "  
                    ++ (map Data.Char.toUpper name)  
                    ++ " ?" ++ sicher  
                putStrLn frage  
            getLine
```

- `let` ist wieder 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 wieder 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 09-9. 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 Monaden verwendbar, nicht nur 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 ++ "!")
```



PROGRAMMIERUNG UND MODELLIERUNG MIT HASKELL

REIN FUNKTIONALE EIN-/AUSGABE, (APPLIKATIVE) FUNKTOREN UND MONADEN

Martin Hofmann Steffen Jost

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

22. Juni 2015



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!
```

TIPP: Besteht ein Programm aus mehreren Modulen, dann sollte die Option `--make` angegeben werden, damit alle notwendigen Module ebenfalls gleich kompiliert werden.



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 dringend, 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 spätere Vorlesung).

- `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 Zeilenvorschub 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

Handles kann man auf verschiedene Arten bekommen:

```
readFile    :: FilePath -> IO String.  
writeFile   :: FilePath -> String -> IO ()  
appendFile :: FilePath -> String -> IO ()  
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
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
  -- Arbeiten
  input <- hGetContents hIn
  let biginput = map toUpper input
  hPutStrLn hOut biginput
  hClose hIn -- Schliessen
  hClose hOut
```

BEISPIELE DATEIZUGRIFF:

BEISPIEL 1

```
import System.IO
import Data.Char
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
  -- Arbeiten
  input <- hGetContents hIn
  let biginput = map toUpper input
  hPutStrLn hOut biginput
  hClose hIn -- Schliessen
  hClose hOut
```

BEISPIELE DATEIZUGRIFF:

BEISPIEL 3

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

Dabei kann man sich `withFile` vorstellen als:

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

withFile' path mode f = do
  handle <- openFile path mode
  result <- f handle
  hClose handle
  return result
```



BEISPIELE DATEIZUGRIFF:

BEISPIEL 3

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

Dabei kann man sich `withFile` vorstellen als:

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

```
withFile' path mode f = do
  handle <- openFile path mode
  result <- f 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 ()
```

WARNUNG

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

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 Monaden
- 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.



Datentypen können parametrisiert sein:

Beispiel: `data Tree a = Empty | Node a (Tree a) (Tree a)`

Achtung: `Tree` ist kein Typ, aber `Tree Int` ist ein Typ!

Typkonstruktor `Tree` bildet Typen auf Typen ab, z.B. Typ `Int` auf den Typ der Binärbäume mit ganzzahligen Knotenbeschriftungen.

Weitere Beispiele: `Maybe`, `Either`, `Map`, `[]` (Listen), ...

Tipp: Mit dem GHCi-Befehl `:kind` kann man sich anzeigen lassen, wie viele Typparameter benötigt werden:

```
> :kind Maybe
Maybe :: * -> *           -- 1 Typparameter
> :kind Data.Map.Map
Data.Map.Map :: * -> * -> *   -- 2 Typparameter
```

“Kinds” sind quasi “Typen” von Typen



KIND

Der **Kind** eines Typen beschreibt die Art des Typen, also die Anzahl und Art der Argumente eines Typkonstruktors. Ein Kind ist entweder `*` oder aus zwei Kinds per Pfeil zusammengesetzt:

$$\kappa ::= * \mid \kappa \rightarrow \kappa$$

`(*)` steht für alle konkreten Datentypen, z.B. `Int`, `Bool`, `Double` und auch `[Int]`, `Maybe Bool`, `Either String Double` und auch vollständigen Funktionstypen wie `Int -> Int`

`(* -> *)` steht für alle Typkonstruktoren mit genau einem Argument, z.B. `[]`, `Maybe` und auch `Either String`.

`(* -> (* -> *))` steht für alle Typkonstruktoren mit genau zwei Argumenten, z.B. `Either`.

Wie bei Funktionstypen ist die Rechtsklammerung implizit, d.h.

$$* \rightarrow (* \rightarrow *) = * \rightarrow * \rightarrow * \neq (* \rightarrow *) \rightarrow *$$



TYPKLASSE FUNCTOR

Im Module `Data.Functor` findet sich folgende Definition:

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

Der Parameter `f` der Typklasse `Functor` steht also nicht für einen konkreten Typ wie z.B. `Tree Int`, sondern für einen Typkonstruktor wie z.B. `Tree`.

Genauer: `f` hat Kind `* -> *`

Die Typklasse `Functor` ist also die Klasse aller "Container"-Typen, welche es erlauben Ihre Inhalte auf andere abzubilden.

Die Deklaration ist dabei unabhängig von dem beinhalteten Typ!



BEISPIEL: FUNCTOR TREE

In Übung A6-3 haben wir Bäume bereits zu einer Instanz von `Functor` gemacht: siehe auch H5-2: `fanwenden`

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)
```

```
instance Functor Tree where
```

```
  fmap f (Leaf a)      = Leaf (f a)
```

```
  fmap f (Node a l r) = Node (f a) (fmap f l) (fmap f r)
```

```
> fmap even (Node 2 (Leaf 1) (Leaf 4))
Node True (Leaf False) (Leaf True)
```

```
> fmap (*2) (Node 2 (Leaf 1) (Leaf 4))
Node 4 (Leaf 2) (Leaf 8)
```

```
> :type fmap
```

```
fmap :: Functor f => (a -> b) -> f a -> f b
```



BEISPIEL: FUNCTOR LISTE

Typklasse `Functor` definiert Infix `<$>` als Synonym für `fmap`:

```
class Functor f where
  fmap  :: (a -> b) -> f a -> f b
  (<$>) :: (a -> b) -> f a -> f b
  (<$>) = fmap      -- nur ein Synonym
```

BEISPIEL:

`fmap` für Listen kennen wir auch längst unter dem Namen `map`:

```
instance Functor [] where
  fmap = map
```

Der Typ `map :: (a -> b) -> [a] -> [b]` passt genau!

```
> even <$> [1..5]
[False,True,False,True,False]
```

BEISPIEL: FUNCTOR MAYBE

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

Eine Funktion auf ein Maybe-Wert 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
```


BEISPIEL: FUNCTOR MAYBE

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

Eine Funktion auf ein Maybe-Wert 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:

```
> even <$> (Just 42)
Just True
> even <$> Nothing
Nothing
```

GENERISCHE FUNKTOR INSTANZEN

Für GHC beinhaltet eine Erweiterung, welche **Functor**-Instanzen für Container-Datentypen automatisch generieren kann:

```
{-# LANGUAGE DeriveFunctor #-}
```

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)  
    deriving (Functor )
```

- Mit `{-# ... #-}` am Anfang der Datei kann der Kompiler beeinflusst werden. Solche speziellen Kommentar nennt man auch **Pragma**.
- Mit dem Pragma **LANGUAGE** werden Spracherweiterung gegenüber dem Haskell Standard aktiviert
- Mehrere Spracherweiterung werden mit Komma getrennt.

FUNKTOR

$$\text{fmap} :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b$$

`fmap f` wendet Funktion `f` punktweise auf eine Datenstruktur an.

Dabei sollten folgende Gesetze erfüllt werden:

- 1 Identität: `fmap id == id`
- 2 Komposition: `fmap f . fmap g == fmap (f . g)`

GHC prüft dies nicht, Programmierer muss dies alleine sicherstellen!

Hinweis: Der Begriff “Funktork” kommt aus der Mathematik: ein Funktor ist strukturerehaltende Abbildung zwischen zwei Kategorien



ZUSAMMENFASSUNG FUNKTOREN

Funktoren sind ein Programmierschema für die punktweise Anwendung einer Funktion auf eine Datenstruktur

- *Identität*: Funktoren verändern nie die Form einer Datenstruktur, sondern nur deren Inhalt
- *Komposition*: Es ist egal, ob wir mehrfach über die Datenstruktur gehen oder nur einmal und dabei gleich mehrere Funktionen hintereinander punktweise anwenden.

⇒ Änderungen durch Funktoren innerhalb einer Datenstruktur sind immer lokal und voneinander unabhängig!



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 etwa `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 $v == \text{pure id } \langle * \rangle v$

KOMPOSITION $u \langle * \rangle (v \langle * \rangle w) == \text{pure } (.) \langle * \rangle u \langle * \rangle v \langle * \rangle w$

HOMOMORPHIE $\text{pure } f \langle * \rangle \text{pure } x == \text{pure } (f x)$

INTERCHANGE $u \langle * \rangle \text{pure } y == \text{pure } (\backslash f \rightarrow f y) \langle * \rangle u$

Für die Definition $f \langle \$ \rangle x = \text{pure } f \langle * \rangle x$ folgen damit alle Gesetze für Funktoren.



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:

$$\text{pure } \text{foo } \langle * \rangle \text{ x } \langle * \rangle \text{ y } :: \text{f c}$$

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
  Nothing <*> _           = Nothing
  (Just f) <*> something = f <f$> something
```

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
> (+) <f$> 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]
```



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 `<*>`:

$$f \langle \$ \rangle x_1 \langle * \rangle \dots \langle * \rangle x_n$$

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 $f \$ x \quad y$
 Funktionsanwendung mit Kontext $f \langle \$ \rangle x \langle * \rangle y$
- $f \langle \$ \rangle x \langle * \rangle y$ ist äquivalent zu **pure** $f \langle * \rangle x \langle * \rangle y$
- Der Infix-Operator $\langle * \rangle$ 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?

BEOBSACHTUNG

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

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

```
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
```

```
> 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**: Kontexte/Effekte können hier zwar auch nicht abgeschüttelt werden, aber es gibt eine Operation, welche Werte kurzzeitig aus ihrem Kontext/Effekt herausnehmen kann, so dass darauf reagiert werden kann:

$$(>>=) :: \text{Monad } m \Rightarrow m \ a \rightarrow (a \rightarrow m \ b) \rightarrow m \ b$$

```
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 09-9!



TYPKLASSE **MONAD**

Aus dem 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 hat historische Gründe
- `(>>=)` gesprochen "**bind**", ist Komposition von Aktionen
- `(>>)` ist Nacheinanderausführung von Aktionen, entspricht Komposition mit Wegwerfen des Zwischenergebnis
- `fail` erlaubt spezielle Fehlerbehandlung



DO-NOTATION FÜR ALLE MONADEN

WDH. 09-16

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)))
```



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

`mav1 >>= (\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
  act y
```

```
do act x
```

② Rechts-Identität

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

```
do mval
```

③ Assoziativität

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

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

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



MONADEN GESETZE

Monaden Gesetze ausgedrückt unter Verwendung der DO-Notation:

① Links-Identität

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

```
do act x
```

② Rechts-Identität

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

```
do act x
```

③ Assoziativität

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

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

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



ALLE MONADEN SIND APPLIKATIVE FUNKTOREN

```

pure = return
mf <*> mx = do  f <- mf
                x <- mx
                return $ f x
f <$> mx = do   x <- mx
                return $ f x

```

- Die Gesetze für (applikative) Funktoren folgen automatisch aus den Gesetzen für Monaden
- *Aber*: Nicht alle (applikative) Funktoren sind Monaden
- Aus historischen Gründen beinhaltet Modul `Control.Monad` alternative Namen für die Funktoren-Operationen:

```

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

```

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

ALLE MONADEN SIND APPLIKATIVE FUNKTOREN

```

pure = return
mf <*> mx = do  f <- mf
                x <- mx
                return $ f x
f <$> mx = do   x <- mx
                return $ f x

```

- Die Gesetze für (applikative) Funktoren folgen automatisch aus den Gesetzen für Monaden
- *Aber*: Nicht alle (applikative) Funktoren sind Monaden
- Aus historischen Gründen beinhaltet Modul `Control.Monad` alternative Namen für die Funktoren-Operationen:

```

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

```

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

ALLE MONADEN SIND APPLIKATIVE FUNKTOREN

```
pure = return
```

Für GHC < 7.10 reichte eine einzelne Instanzdeklaration für `Monad` aus. Die Instanzen für `Applicative` und `Functor` werden automatisch abgeleitet.

Ab GHC 7.10 ist dies nicht mehr der Fall, dann müssen die Instanzdeklaration für `Functor` und `Applicative` explizit hingeschrieben werden. Dabei kann man aber `liftM` und `ap` verwenden.

- *Aber:* Nicht alle (applikative) Funktoren sind Monaden
- Aus historischen Gründen beinhaltet Modul `Control.Monad` alternative Namen für die Funktoren-Operationen:

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

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

PROGRAMMIERUNG UND MODELLIERUNG MIT HASKELL

REIN FUNKTIONALE EIN-/AUSGABE, (APPLIKATIVE) FUNKTOREN UND MONADEN

Martin Hofmann Steffen Jost

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

25. Juni 2015



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
  (>>=) 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]
  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` 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` 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.



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 assoziiertst 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, MonadError`
- **Nichtdeterminismus** für Berechnungen, welche mit mehreren Alternativen gleichzeitig rechnen `z.B. Listen-Monade`
- **Zustandsmonade** für Berechnungen mit veränderlichen Kontext `Control.Monad.State`

SPEZIALFÄLLE:

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



ZUSAMMENFASSUNG MONADEN

- Monaden kann man sich 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.
- Alle Monaden sind immer auch (applikative) Funktoren
- `IO` ist eine Monade
- Monaden sind im Modul `Control.Monad` implementiert



ZUSAMMENFASSUNG MONADEN

- Monaden sind ein *Programmierschema* für zusammengesetzte Berechnungen mit Kontext oder Seiteneffekten; grundlegende Operation ist die Komposition von Aktionen
- 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**.
- 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

