



Funktionale Programmierung

Woche 10 – Eingabe und Ausgabe, Vier Gewinnt

René Thiemann

Philipp Dablander

Joshua Ocker

Michael Schaper

Lilly Schönherr

Adam Pescoller

Letzte Vorlesung

- **Sichtbarkeitsregeln** für Funktions- und Variablennamen
- größere Programme sollten in **Modulen** strukturiert werden
 - explizite **Export-Listen** differenzieren zwischen internen und externen Teilen eines Moduls
 - Module werden durch Importe verfügbar, nicht durch Kopieren des Codes
 - qualifizierte Importe und Qualifier vermeiden Namenskonflikte oder lösen diese auf
 - Standard-Module und -Importe
 - wenn Programm keine Modul-Deklaration enthält, wird `module Main where` hinzugefügt
 - wenn Programm `Prelude` nicht selber importiert, wird `import Prelude` hinzugefügt
- Beispiel

```
module Rat(Rat,createRat) where ...
```

```
module Application where
```

```
import Prelude hiding (pi)      -- hide import of pi
```

```
import Rat
```

```
pi :: Rat      -- so that here there won't be a conflict
```

```
pi = createRat -- pi with precision of 70 digits
```

31415926535897932384626433832795028841971693993751058209749445923078164

[illegible]

Eingabe und Ausgabe in Haskell

I/O: Eingabe und Ausgabe (Input und Output)

- Ziel: Kommunikation mit der Welt (AnwenderIn, Dateisystem, Netzwerk, ...)
 - lese Eingaben von AnwenderIn
 - gebe Antworten aus
 - **außerhalb** des read-eval-print-loops von ghci
 - erstelle Programme, deren Ausführung weder ghc-Installation noch Haskell-Kenntnis erfordern
- I/O kann vielseitig verstanden werden
 - Dateizugriff
(z.B. transformieren Compiler .hs zu .exe, oder .tex zu .pdf)
 - Speicherzugriff
(veränderbare Variablen, Arrays)
 - Netzwerkzugriff
(z.B. um einen Web-Server oder Internet-Browser zu entwickeln)
 - starte externe Programme und kommuniziere mit diesen
 - Audioaufnahme und -wiedergabe
 - Kommunikation mit GUI
 - ...

Ein erstes I/O-Beispiel

- ```
main = do -- file: WelcomeIO.hs
 putStrLn "Greetings! Please tell me your name."
 name <- getLine
 putStrLn $ "Welcome to Haskell's IO, " ++ name ++ "!"
```
- Compilierung mit GHC (nicht mit GHCi) mittels:  

```
$ ghc --make WelcomeIO.hs
```
- Ausführung  

```
$./WelcomeIO # WelcomeIO.exe on Windows
Greetings! Please tell me your name.
Homer # this was typed in
Welcome to Haskell's IO, Homer!
```
- Anmerkungen
  - `putStrLn` – gibt einen String aus und springt in nächste Zeile
  - `getLine` – liest eine Zeile ein
  - neue Syntax: `do` und `<-`

# I/O und das Typsystem

- `ghci> :l WelcomeIO.hs`  
`ghci> :t putStrLn`  
`putStrLn :: String -> IO ()`  
`ghci> :t getLine`  
`getLine :: IO String`  
`ghci> :t main`  
`main :: IO ()`
- `IO a` ist der Typ von I/O-Aktionen, die ein Resultat vom Typ `a` liefern (und zusätzlich I/O-Operationen durchführen können)
- Beispiele
  - `String -> IO ()` – berechne eine Aktion basierend auf einem String (im Fall von `putStrLn` ist die Aktion, den String auszugeben)
  - `IO ()` – führe nur eine Aktion durch (das Ergebnis `()` hat keine Bedeutung) (im Fall von `main`, führe das Programm aus)
  - `IO String` – führe eine Aktion aus, die einen String liefert (im Fall von `getLine`, lese eine Zeile ein)

## Kombination von I/O-Aktionen

- I/O-Aktionen können sequentiell kombiniert werden
- Sequenz-Operator: **Bind** (Syntax `>>=`)  
 $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
- betrachte `act1 >>= \ x -> act2`
  - bei der Auswertung des Ausdrucks wird zuerst Aktion `act1` durchgeführt
  - das Resultat von Aktion `act1` wird in `x` gespeichert
  - anschließend wird `act2` ausgeführt (und `act2` darf von `x` abhängen)
  - insgesamt werden beide Aktionen ausgeführt, und das finale Resultat ist das von `act2`
- **schwaches Bind**:  $(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b, a1 \gg a2 = a1 \gg= \_ \rightarrow a2$
- Beispiel

```
putStrLn "Hi. What's your name?" >> -- ignore result, which is ()
getLine >>= \ name -> -- store result in variable name
let answer = "Hello " ++ name in -- no I/O in this line
putStrLn answer -- final result from putStrLn: ()
```

  - Typ des Ausdrucks ist `IO ()`, also genau der Typ der letzten I/O-Aktion `putStrLn answer`
  - die Ausführung von mehreren Aktionen ist sequentiell, wie in der imperativen Programming

## Do-Notation

- es gibt eine Spezial-Syntax für Kombinationen von Bind,  $\lambda$ -Abstraktionen und `let`

```
do x <- act = act >>= \ x -> do block
 block
```

```
do act = act >> do block
 block
```

```
do let x = e = let x = e in do block
 block
```

- `putStrLn "Hi. What's your name?" >>`

```
 getLine >>= \ name ->
```

```
 let answer = "Hello " ++ name in
```

```
 putStrLn answer
```

kann mittels do-Notation wie folgt geschrieben werden

```
do putStrLn "Hi. What's your name?"
```

```
 name <- getLine
```

```
 let answer = "Hello " ++ name -- no "in"!
```

```
 putStrLn answer
```

- wie bei `let` können `do`-Blöcke auch geklammert werden: `do {..; ..; ..}`



## Weitere Anmerkungen

- innerhalb eines `do`-Blocks ist die Reihenfolge wichtig; I/O-Aktionen werden sequentiell ausgeführt; Resultat eines Blocks ist das der **letzten** Aktion
- Auslesen des Results einer Aktion mittels `x <- a` ist nur innerhalb I/O-Aktionen möglich; es gibt keine Funktion vom Typ `IO a -> a`, die das Ergebnis einer I/O-Aktion extrahiert, ohne dabei selber eine I/O-Aktion zu sein
  - sobald wir innerhalb einer I/O-Aktion sind, können wir I/O nicht verlassen
  - **strikte Trennung zwischen rein-funktionalem Code und I/O**
  - wenn `IO a` nicht im Typ vorkommt, können wir sicher sein, dass keine I/O-Aktionen (also auch keine **Seiteneffekte**) durchgeführt werden
- `main :: IO ()` ist die I/O-Aktion, die ausgeführt wird, wenn man eine Datei mittels `ghc --make Prog.hs` compiliert und anschließend mittels `./Prog` startet (Prog.hs muss Modul `Main` beinhalten und `main` exportieren)
- gibt man in `ghci` einen Ausdruck `act :: IO a` ein, dann wird erst `act` ausgeführt, und danach der Wert des Resultats ausgegeben, falls `a` nicht `()` ist

## Nutzung von Rein-Funktionalem Code innerhalb von I/O-Aktionen

```
-- reply is purely functional: no IO in type
reply :: String -> String
reply name =
 "Pleased to meet you, " ++ name ++ ".\n" ++
 "Your name contains " ++ n ++ " characters."
 where n = show $ length name

-- pure code can be invoked from I/O-part
main :: IO ()
main = do
 putStrLn "Greetings again. What's your name?"
 name <- getLine
 let niceReply = reply name
 putStrLn niceReply
```

- der Aufruf von rein funktionalem Code von I/O ist einfach
- die andere Richtung ist nicht möglich!

## Vordefinierte Funktionen mit I/O-Aktionen

- `return :: a -> IO a` – verwandle Wert in I/O-Aktion, die den Wert als Resultat liefert
- `System.Environment.getArgs :: IO [String]` – liefert Kommandozeilen Argumente
- `putChar :: Char -> IO ()` – gebe einzelnes Zeichen aus
- `putStr :: String -> IO ()` – gebe String aus
- `putStrLn :: String -> IO ()` – gebe String aus mit Zeilenumbruch
- `getChar :: IO Char` – lese einzelnes Zeichen von stdin
- `getLine :: IO String` – lese Zeile (Zeilenumbruch ist nicht im Resultat)
- `interact :: (String -> String) -> IO ()` – verwandle gesamte Eingabe in Ausgabe mittels einer Funktion
- `type FilePath = String`
- `readFile :: FilePath -> IO String` – lese gesamte Datei
- `writeFile :: FilePath -> String -> IO ()`
- `appendFile :: FilePath -> String -> IO ()`

## Rekursive I/O-Aktionen

- Verzweigung und Rekursion sind auch mit I/O-Aktionen möglich
- Beispiel: implementiere `getLine` mittels `getChar`

```
import Prelude hiding (getLine)

getLine = do
 c <- getChar
 if c == '\n' -- branching
 then return ""
 else do
 l <- getLine -- recursion
 return $ c : l
```

## Beispiele – Imitation einiger GNU Programme

- `cat.hs` – gebe Dateiinhalt aus

```
import System.Environment (getArgs)
main = do
 [file] <- getArgs -- assume there is exactly one file
 s <- readFile file
 putStr s
```
- `wc.hs` – zähle Anzahl von Zeilen/Wörtern/Zeichen in Eingabe

```
count s = nl ++ " " ++ nw ++ " " ++ nc ++ "\n"
 where nl = show $ length $ lines s
 nw = show $ length $ words s
 nc = show $ length s
main = interact count
```
- `sort.hs` – sortiere Eingabe zeilenweise

```
import Data.List (sort)
main = interact (unlines . sort . lines)
```

# Lazy-Evaluation und I/O-Aktionen

- betrachten Sie ein einfaches Programm, um Dateien zu kopieren

```
main = do -- imports omitted
 [src, dest] <- getArgs
 s <- readFile src
 writeFile dest s
```

- `readFile` und `writeFile` haben eine verzögerte Auswertung (**lazy-Evaluation**), d.h., `readFile` liest Zeichen nur bei Bedarf
  - positiver Effekt: kopiere große Dateien, ohne diese vollständig in den Speicher zu laden
- Lazy-Evaluation kann auch Probleme verursachen

```
main = do -- imports omitted
 [file] <- getArgs
 s <- readFile file
 writeFile file (map toUpper s)
```

- weil `readFile` lazy ist, wird bei `s <- readFile file` nichts direkt gelesen
  - danach wird versucht, **dieselbe** Datei schreibend zu öffnen; führt zu Programm-Abbruch
  - Lösung: feinere Kontrolle, in der Dateien explizit geöffnet und geschlossen werden können; siehe Vorlesungen Betriebssysteme und Fortgeschrittene Funktionale Programming

## Higher-Order und I/O-Aktionen

- `foreach :: [a] -> (a -> IO b) -> IO ()`  
`foreach [] io = return ()`  
`foreach (a:as) io = do { io a; foreach as io }`
- bessere Variante von `cat.hs`

```
main = do
 files <- getArgs
 if null files then interact id else do
 foreach files readAndPrint
 where readAndPrint f = readFile f >>= putStr
```

## Beispiel Anwendung: Vier Gewinnt



# Vier Gewinn

- Ziel: Implementierung von **Vier Gewinn**, MB Spiele



- mit einer textuellen Benutzeroberfläche

0123456

.....

.X0.X..

.X000X0

X0X0X0X

0XX0X00

XX0X00X

Player X to go

Choose one of [0,1,2,3,4,5,6]

## Vier Gewinnt: Implementierung

- klare Trennung von zwei Teilen
  - Benutzeroberfläche (I/O)
    - Einlesen eines Zuges
    - Ausgabe des aktuellen Spielstands
    - ...
  - Spiellogik (rein funktionaler Code)
    - Typ, um Spielstand zu repräsentieren (Brett + nächster Spieler)
    - Durchführung eines Zugs
    - Prüfung, ob jemand gewonnen hat
    - Darstellung eines Spielstands als String
    - ...
- jeder Teil ist in eigenem Modul implementiert
  - `Logic` beinhaltet die Spiellogik
  - `Main` beinhaltet Benutzeroberfläche und die `main` Funktion

## Spiellogik: Bereitgestellte Funktionalität

- Typen: `State`, `Move` und `Player`
- Konstante `initState :: State`
- Funktion `showPlayer :: Player -> String`
- Funktion `showState :: State -> String`
- Funktion `winningPlayer :: State -> Maybe Player`
- Funktion `validMoves :: State -> [Move]`
- Funktion `dropTile :: Move -> State -> State`
- in Summe

```
module Logic(State, Move, Player,
 initState, showPlayer, showState,
 winningPlayer, validMoves, dropTile) where
 ... -- details, which the user interface doesn't have to know
```

## Die Read-Klasse

- Klasse `Read` bietet Methoden an, um `Strings` in andere Typen zu verwandeln
  - `read :: Read a => String -> a`
  - `readMaybe :: Read a => String -> Maybe a`  
Import von Modul `Text.Read` wird benötigt
  - bei Benutzung von `read` wird oft der Typ `a` explizit angegeben
  - Beispiele
    - `(read "(41, True)" :: (Integer, Bool)) = (41, True)`
    - `(read "(41, True)" :: (Integer, Integer)) = error ...`
    - `(readMaybe "1" :: Maybe Integer) = Just 1`
    - `(readMaybe "one" :: Maybe Integer) = Nothing`
- für das `Logic` Modul nehmen wir an, dass Typ `Move` eine Instanz von `Show` und `Read` ist

# Benutzeroberfläche

```
module Main(main) where -- module name must be "Main" for compilation
import Logic
main = do
 putStrLn "Welcome to Connect Four"
 game initState
game state = do
 putStrLn $ showState state
 case winningPlayer state of
 Just player -> putStrLn $ showPlayer player ++ " wins!"
 Nothing -> let moves = validMoves state in
 if null moves then putStrLn "Game ends in draw."
 else do
 putStrLn $ "Choose one of " ++ show moves ++ ": "
 hFlush stdout -- flush output buffer
 moveStr <- getLine
 let move = (read moveStr :: Move)
 game (dropTile move state)
```

## Spiellogik: Repräsentation eines Zustands; Startzustand

```
type Tile = Int -- 0, 1, or 2
type Player = Int -- 1 and 2
type Move = Int -- column number
data State = State Player [[Tile]] -- list of rows

empty :: Tile
empty = 0

numRows, numCols :: Int
numRows = 6
numCols = 7

startPlayer :: Player
startPlayer = 1

initState :: State
initState = State startPlayer (replicate numRows (replicate numCols empty))
```

## Spiellogik: Gültige Züge und Darstellung eines Spielzustands

```
validMoves :: State -> [Move]
validMoves (State _ rows) =
 map fst . filter ((== empty) . snd) . zip [0 .. numCols - 1] $ head rows

showPlayer :: Player -> String
showPlayer 1 = "X"
showPlayer 2 = "O"

showTile :: Tile -> Char
showTile t = if t == empty then '.' else head $ showPlayer t

showState :: State -> String
showState (State player rows) = unlines $
 concatMap show [0 .. numCols - 1] :
 map (map showTile) rows
 ++ ["\nPlayer " ++ showPlayer player ++ " to go"]
```

## Spiellogik: Durchführung eines Zugs

```
otherPlayer :: Player -> Player
```

```
otherPlayer = (3 -)
```

```
dropTile :: Move -> State -> State
```

```
dropTile col (State player rows) = State
```

```
 (otherPlayer player)
```

```
 (reverse $ dropAux $ reverse rows)
```

```
 where
```

```
 dropAux (row : rows) =
```

```
 case splitAt col row of
```

```
 (first, t : last) ->
```

```
 if t == empty
```

```
 then (first ++ player : last) : rows
```

```
 else row : dropAux rows
```



## Spiellogik: Gewinn-Bedingung

```
winningRow :: Player -> [Tile] -> Bool
```

```
winningRow player [] = False
```

```
winningRow player row = take 4 row == replicate 4 player
 || winningRow player (tail row)
```

```
transpose ([] : _) = []
```

```
transpose xs = map head xs : transpose (map tail xs)
```

```
winningPlayer :: State -> Maybe Player
```

```
winningPlayer (State player rows) =
```

```
 let prevPlayer = otherPlayer player
```

```
 longRows = rows ++ transpose rows -- ++ diags rows
```

```
 in if any (winningRow prevPlayer) longRows
```

```
 then Just prevPlayer
```

```
 else Nothing
```

## Vier Gewinnt: Abschließende Bemerkungen

- Implementierung ist rudimentär
  - Diagonalen werden bei der Gewinn-Bedingung nicht berücksichtigt
  - Programm-Abbruch bei Eingabe ungültiger Züge
  - ...
- Übung: Ausbau der Implementierung

## Zusammenfassung

- in Haskell ist Ein- und Ausgabe möglich;  
`IO a` ist der Typ von I/O-Aktionen mit Resultat vom Typ `a`
- Typsystem bietet klare Trennung von rein-funktionalem und I/O-Code
- mehrere Aktionen können mit `(>>=)` oder in `do`-Blöcken kombiniert werden
- es gibt viele vordefinierte Funktionen für die Ein- und Ausgabe
- weitere Informationen bzgl. I/O in Haskell:  
<http://book.realworldhaskell.org/read/io.html>
- `Read` Klasse bietet Funktion `read :: String -> a`, die duale Funktion zu  
`show :: a -> String`
- Vier Gewinnt: getrennte Implementierung der Spiellogik (rein funktional) und Benutzeroberfläche (I/O)