Monadische I/O: Einfache Beispiele

```
-- Funktionen aus der Standardbücherei:
getLine  :: IO String       -- eine Zeile lesen
putStr   :: String-> IO ()  -- Zeile ausgeben
putStrLn :: String-> IO ()  -- Zeile mit LF ausgeben

echo :: IO ()
echo = getLine >>= putStrLn >>= \_ -> echo

interactOnce :: (String-> String)-> IO ()
interactOnce f = getLine >>= (putStr . f)

revecho :: IO ()
revecho = getLine >>= putStrLn . reverse >> revecho

(>>) :: IO t-> IO u-> IO u
```


Monadische I/O: Die do Notation

- Syntaktischer Zucker für Monaden:

```
echo = do s<- getLine                echo = getLine
          putStrLn s        <-->          >>= \s -> putStrLn s
          echo                            >> echo
```

- Oder auch (Abseitsregel):

```
echo = do { s<- getLine; putStrLn s; echo }
```


Eine „Benutzeroberfläche“ für Nim

- Am Anfang Begrüßung
- Auswürfeln der Anzahl der Hölzchen:

  `randomRIO:: (a, a)-> IO a` aus dem Modul `Random`
- Für jeden Zug:
  - Ausgabe der Anzahl der Hölzchen im Haufen
  - Abfrage der Eingabe des Spielers
  - Falls kein optimaler Gewinnzug möglich, aufgeben; ansonsten nächster Zug (mit entsprechend weniger Hölzchen).

Eine „Benutzeroberfläche“ für Nim

```
import Random (randomRIO)
play :: Int-> IO ()
play n =
  do putStrLn ("Es sind "++ show n ++" Hölzchen im Haufen.")
     if n == 1 then putStrLn "Ich habe gewonnen!"
        else do m<- getInput -- auf der nächsten Folie
                case wins (n-m) of
                    Nothing -> putStrLn "Ich gebe auf."
                    Just l  -> do putStrLn ("Ich nehme " ++ show l)
                                  play (n-(m+l))
main :: IO ()
main = do putStrLn "\nWillkommen bei Nim!\n"
          n <- randomRIO(5,49)
          play n
```

Eine „Benutzeroberfläche“ für Nim

```
getInput :: IO Int
getInput =
  do putStr "Wieviele nehmen Sie? "
     n <- getLine >>= return . read
     if n<=(0::Int) || n>3 then putStrLn "Ungültige Eingabe!" >> getInput
       else return n
```

Beispiellauf:

- nur bei erstem Zug Gewinnchance ...
- Fehlerbehandlung?

Fehler!

Fehler werden durch den abstrakten Datentyp `IOError` repräsentiert.

Ausnahmebehandlung ähnlich Java:

```
ioError  :: IOError -> IO a                    -- "throw"
catch    :: IO a -> (IOError -> IO a)-> IO a

getInput :: IO Int
getInput =
  do putStr "Wieviele nehmen Sie? "
     n <- catch (getLine >>= readIO)
                (\_ -> (putStrLn "Eingabefehler." >> getInput))
     if n<=(0::Int) || n>3 then putStrLn "Ungültige Eingabe!" >> getInput
       else return n
```

`readIO:: String-> IO a` „wirft“ Ausnahme, die gefangen werden kann.

Haskell — *world's best imperative programming language*?

Aktionen sind Werte — dadurch Definition *eigener* Kontrolloperatoren:

```
forever :: IO a -> IO a
forever a = a >> forever a

forN :: Int-> IO a-> IO ()
forN n a = if n== 0 then return () else a >> forN (n-1) a

for :: (a, a-> Bool, a-> a) -> (a-> IO ())-> IO ()
for (s, p, next) cmd = if p s then return ()
                          else cmd s >> for(next s, p, next) cmd
-- Beispiel: for(3,(<) 10, (+) 1) (\x-> putStrLn (show x))
```


Vordefinierte Kontrolloperatoren aus dem `Prelude`

- Liste abarbeiten:

```
sequence  :: [IO a] -> IO [a]
sequence (c:cs) = do x  <- c
                     xs <- sequence cs
                     return (x:xs)
sequence_ :: [IO a] -> IO ()
```

- Variation der bekannten Listenoperationen für Monaden:

```
mapM    :: (a-> IO b)-> [a]-> IO [b]
mapM f  = sequence . map f
mapM_   :: (a-> IO b)-> [a]-> IO ()
mapM_ f = sequence_ . map f
```

Andere (wie `filterM :: (a-> IO Bool)-> [a]-> IO [a]`) in Modul `Monad`

File I/O

Abstrakter Zugriff durch verzögerte Auswertung (*lazy evaluation*):

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

Beispiel:

```
cntWords :: FilePath -> IO ()
cntWords file = do c <- readFile file
                   let s = (length . words) c
                   putStrLn (file ++ " contains " ++ show s ++ " words")
```


Kommandozeilenargumente

Interaktion mit der Umgebung: Modul `System`

```
data ExitCode = ExitSuccess | ExitFailure Int
getArgs       :: IO [String]
getProgName   :: IO String
getEnv        :: String -> IO String
system        :: String -> IO ExitCode
exitWith      :: ExitCode -> IO a
```

Beispiel:

```
main = do r<- getProgName; a<- getArgs
          catch (mapM_ cntWords a)
                (\e-> putStrLn (r++ ": "++ (show e)))
```


Das Modul `Directory`

Abstrakte Modellierung des Dateisystems.

```
createDirectory             :: FilePath -> IO ()
removeDirectory, removeFile :: FilePath -> IO ()
renameDirectory, renameFile :: FilePath -> FilePath -> IO ()

getDirectoryContents :: FilePath -> IO [FilePath]
getCurrentDirectory  :: IO FilePath
setCurrentDirectory  :: FilePath -> IO ()

data Permissions = ...
readable, writeable, executable, searchable :: Permissions-> Bool
getPermissions       :: FilePath -> IO Permissions
setPermissions       :: FilePath -> Permissions -> IO ()
getModificationTime  :: FilePath -> IO ClockTime
```


Das Modul *Directory*, Beispiel

```
import Directory
import Time
import System(getArgs)

cleanup dir =
 do now<- getClockTime
    c<- getDirectoryContents dir
    setCurrentDirectory dir
    mapM_ (\f-> do mt<- getModificationTime f
                   if ((last f == '~') &&
                      tdDay (diffClockTimes mt now) >= 1)
                   then removeFile f else return ()}) c
main = do { d<- getArgs; cleanup (head d) }
```

Systemfunktionen nach POSIX

- Abstrakte Modellierung in *Haskell98 Standard Library:*
  `IO`, `Directory` `System`, `Time`
  Siehe *Library Report*
- Konkrete Modellierung in Modul *Posix* (nur für `GHC`) nach IEEE Standard 1003.1, e.g.:

```
executeFile :: FilePath                      -- Command
            -> Bool                          -- Search PATH?
            -> [String]                      -- Arguments
            -> Maybe [(String, String)]      -- Environment
            -> IO ()
```


Fehleranalyse

Aus dem Modul `IO`:

```
isAlreadyExistsError  :: IOError -> Bool
isDoesNotExistError   :: IOError -> Bool
isAlreadyInUseError   :: IOError -> Bool
isFullError           :: IOError -> Bool
isEOFError            :: IOError -> Bool
isIllegalOperation    :: IOError -> Bool
isPermissionError     :: IOError -> Bool
isUserError           :: IOError -> Bool

ioeGetErrorString     :: IOError -> String
ioeGetHandle          :: IOError -> Maybe Handle
ioeGetFileName        :: IOError -> Maybe FilePath
```

`error :: String-> a` erzeugt einen `user error`

Typklassen

- Typklassen erlauben das *Überladen* von Operationen.
- Beispiel Gleichheit:

```
class Eq a where
  (==) :: a-> a-> Bool
  (|=) :: a-> a-> Bool
  a |= b = not (a == b)
```

- Bei *Instanziierung* müssen Klassenfunktionen definiert werden.
- Durch Vorbedingungen kann *Klassenhierarchie* definiert werden:

```
class Eq a=> Ord a where
  (<) :: a-> a-> Bool
```

- Weitere Standardklassen: `Show`, `Read`, `Num`, ...

Ausführbare Haskell-Programme

- Der Glasgow Haskell Compiler `ghc` übersetzt Haskell in ausführbare Programme.

  Das zu übersetzende Programm muß die Aktion `main:: IO()` definieren.

  Benutzung: `ghc -o <name-of-exectable> Sourcefile.hs`

  Verzeichnis `/usr/local/pub-bkb/bin` muß im Suchpfad (`PATH`) sein.
- Vorteile:

  Erzeugt schnellen Code

  Getrennte Übersetzung, `Make`-Funktionalität

  Viele Büchereien

`hugs` interpretiert Haskell-Programme (interaktive Eingabe)

`runhugs` kann Skripte verarbeiten (wie Perl oder die Shell)

Voraussetzung: Funktion `main :: IO ()`

Erste Zeile des Programmes muß auf `runhugs` zeigen:
`#!<Pfad-zu-runhugs>`

Beispiel:

```
#!/usr/local/lang/runhugs
main :: IO ()   -- Wichtig!
main = getLine >>= putStrLn >> main
```