

Grundlagen der Programmierung 2 (1.D)

Prof. Dr. Manfred Schmidt-Schauß

Künstliche Intelligenz und Softwaretechnologie

3. Mai 2007

Ein-Ausgabe mittels **IO-Aktionen**

Typ: $a1 \rightarrow a2 \dots \rightarrow IO\ a,$

Spezialfall: $IO\ a$

Ein/Ausgabe durch Auswerten eines Ausdrucks vom Typ $IO\ a$

a : Typ der Eingabe.

Falls keine Eingabe, dann $IO\ ()$

```
putStr :: String -> IO ()
putStrLn :: String -> IO ()
getLine :: IO String
print :: (Show a) => a -> IO ()
readLn :: (Read a) => IO a
```

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

putStr, putStrLn und print geben ihr Argument aus

print und readLn sind polymorph.

Verwendete Methode ist abhängig vom statisch ermittelten Typ

Das Hello-World-Programm:

```
*Main> putStr "Hello World"  
Hello World*Main>
```

do-Notation analog zu Listen-Komprehensionen
bewirkt Kombination von Aktionen
Ermöglicht Definition von neuen IO-Aktionen
mittels vordefinierter IO-Aktionen
Kontrollierte Selektion der Eingabe aus IO-Objekten
do ... arg <- ... wirkt wie ein Selektor,

Beispiel

```
do {input <- getLine; putStr input}
```

äquivalent dazu ist:

```
do input <- getLine  
    putStr input
```

```
*Main> do {input <- getLine; putStrLn input}
```

```
Hello -- eingetippt
```

```
Hello -- Ausgabe
```

```
*Main>
```

```
Prelude> do {inp <- readLn; putStr ((show (inp*2)) ++ "\n")}
```

```
3
```

```
6
```

```
Prelude>
```

- Kombination von
- Typisierung
 - eingeschränktem Satz von vordefinierten IO-Aktionen
 - nur do-Notation darf selektieren

bewirkt

Programmiersbeschränkungen
die zum reinen Programmieren passen:

Trennung von Ein/Ausgabe und Auswertung

Sequentialisierung der Ein-Ausgaben

```
echoLine :: IO String
echoLine = do
    input <- getLine
    putStr input
```

Einlesen einer Int-Zahl

```
getNumber :: IO Int
getNumber = do
    putStr "Bitte eine Zahl eingeben:"
    readLn
```

Parametrisierte Listen-Ausgabe:

```
main = do a <- getNumber
          b <- getNumber
          print (take a (repeat b))
```

```
*Main> main
```

```
Bitte eine Zahl eingeben:4
```

```
Bitte eine Zahl eingeben:6
```

```
[6,6,6,6]
```

File lesen und ausgeben:

```
fileLesen = do
    putStr "File-Name:?"
    fname <-getLine
    contents<-readFile fname
    putStr contents
```

Wiederholte Nachfrage, ob
(E)rgebnis gewünscht oder (W)eiter?

```
weiter = do
  putStrLn "(E)rgebnis oder (W)eiter?"
  w <- getLine
  case (head w) of
    'E' -> return False
    'W' -> return True
    other -> weiter
```

Rekursive Definition einer Aktion.

Addieren beliebig vieler Zahlen

```
addiere = addiere_ 0
```

```
addiere_ x = do
```

```
  a <- getNumber
```

```
  w <- weiter
```

```
  if w
```

```
    then addiere_ (a+x)
```

```
    else putStrLn ("Das Ergebnis ist " ++ (show (a + x)))
```

Addieren beliebig vieler Zahlen mit Speicherung der Eingabe

```
addiereS = addiereS_ []

addiereS_ xs = do
  a <- getNumber
  w <- weiter
  if w
  then addiereS_ (a:xs)
  else putStrLn $ (concat $ intersperse (" + ")
                  ((reverse .map show) (a:xs)))
                ++ " = "
                ++ (show (sum (a:xs)))
```

Module dienen zur

- Strukturierung / Hierarchisierung
- Kapselung:
- Wiederverwendbarkeit:

Zum Modul gehören: Funktionen, Datentypen, Typsynonyme
können **exportiert** werden,
von anderen Modulen **importiert** werden

Syntax

```
module Modulname(Exportliste) where  
    Modulimporte,  
    Datentypdefinitionen,  
    Funktionsdefinitionen, ... } Modulrumpf
```

Für jedes Modul muss eine separate Datei angelegt werden, wobei der Modulname dem Dateinamen ohne Dateiendung entsprechen muss.

Haskell-Programm besteht aus einer Menge von Modulen

Das Modul `Main` muss existieren

und eine Funktion namens `main :: IO ()` definieren und exportieren.

Grundgerüst eines Haskell-Programms:

```
module Main(main) where
  ...
  main = ...
  ...
```

```
module Spiel where
  data Ergebnis = Sieg | Niederlage | Unentschieden
  berechneErgebnis a b = if a > b then Sieg
                        else if a < b then Niederlage
                        else Unentschieden

  istSieg Sieg = True
  istSieg _    = False
  istNiederlage Niederlage = True
  istNiederlage _          = False
```

Exportliste:

Funktionen oder Typen mit oder ohne Konstruktoren
auch importierte Namen können wieder exportierten werden

Mittels `import` am Anfang des Modulrumpfs.

Varianten:

`import Modulname`

importiert alle Einträge der Exportliste von *Modulname*

`import Modulname(N1,N2,...)`

importiert spezifizierte Namen von *Modulname*

`import Spiel hiding(...)`

importiert alles bis auf ...

Aufruf von importierten Funktionen:
mit unqualifizierten Namen
mit *Modulname.unqualifizierter Name*

Beispiel zur Notwendigkeit qualifizierter Namen:

```
module A(f) where
  f a b = a + b
```

```
module B(f) where
  f a b = a * b
```

```
module C where
  import A
  import B
  g1 = f 1 2 + f 3 4      -- funktioniert nicht
  g2 = A.f 1 2 + B.f 3 4 -- funktioniert
```

Modul M exportiert f und g,

Import-Deklaration	definierte Namen
<code>import M</code>	f, g, M.f, M.g
<code>import M()</code>	keine
<code>import M(f)</code>	f, M.f
<code>import qualified M</code>	M.f, M.g
<code>import qualified M()</code>	keine
<code>import qualified M(f)</code>	M.f
<code>import M hiding ()</code>	f, g, M.f, M.g
<code>import M hiding (f)</code>	g, M.g
<code>import qualified M hiding ()</code>	M.f, M.g
<code>import qualified M hiding (f)</code>	M.g
<code>import M as N</code>	f, g, N.f, N.g
<code>import M as N(f)</code>	f, N.f
<code>import qualified M as N</code>	N.f, N.g