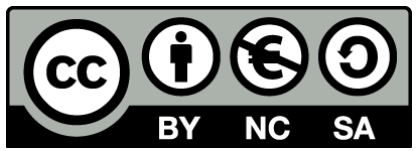
Vorlesung
Programmierung und Modellierung mit Haskell

2 Ein- und Ausgaben

François Bry

Sommersemester 2021



Zusammenfassung der letzten Vorlesung

1 Erste Programme

- Der Interpreter ghci
- Ausdrücke und Operatoren
- Definitionen, lokale Definitionen und Überschatten
- Funktionen
- Char, String und List
- Comprehensions

Quiz Lernfortschritte

Welche Aussagen sind korrekt?

A. `4^3^2` steht für `(4^3)^2`.

B. `4^3^2` steht für `4^(3^2)`.

C. `(&&) True False` ist kein korrekter Ausdruck.

D. `let a = let a = 1 in a + a` hat den Wert 2.

E. `head (tail (tail "abc"))` ist eine Liste.

F. `head tail tail "abc"` ist ein Buchstabe (Char).

G. `[x | x <- [0..], x `mod` 5 == 0]` ist eine Liste.

H. `[x | x <- [0..], x >= 0]` ist die Liste aller natürlichen Zahlen.

Antworten: https://lmu.onlineted.de

Inhalt der heutigen Vorlesung

2 Ein- und Ausgaben

- Daten ein- und ausgeben
- Dateien einlesen und Daten in Dateien ausgeben (Programme kompilieren und auswerten)
- Ein- und Ausgabe-Aktionen

Daten ein- und ausgeben

Das Folgenden von ghci ausführen lassen:

```
putStrLn("Hello World!")
:type putStrLn
putStrLn 1
show 12
putStrLn (show 12)
```

`putStrLn` ist eine I/O-Aktion, die einen String erhält, um das Zeichen New Line ergänzt und ausgibt.

`show` erhält einen Wert von einer show-Klasse und gibt eine String-Darstellung davon zurück.

Daten ein- und ausgeben

Das folgenden von ghci ausführen lassen:

```
print("Hello World!")
print 12
:type print

let myPrint1 x = putStrLn (show x)
myPrint1("Hello World!")
myPrint1 12

let myPrint2 = putStrLn . show
myPrint2("Hello World!")
myPrint2 12
:type (.)
```

Quiz print

```
myPrint1 x = putStrLn (show x)
myPrint2 = putStrLn.show
```

Sind `print`, `myPrint1` und `myPrint2` bis auf die Namen die selben Funktionen?

1. Ja
2. Nein

Antworten: https://lmu.onlineted.de

Daten ein- und ausgeben

In ghci aufrufen:

```
zeichenfolge <- getLine
:type getLine
```

`getLine`: eine I/O-Aktion, die ein String erhält.

`getChar`: eine I/O-Aktion, die ein Char erhält.

`z <- getLine` bringt das String, das die I/O-Aktion getLine enthält, in die Variable z.

`<-` dient dazu, Daten aus einer I/O-Aktion zu entnehmen.

`<-` verletzt die referentielle Transparenz: Nach verschiedenen Ausführungen von `z <- getLine` ist der Wert von `z` nicht immer gleich.

Daten ein- und ausgeben

Eine I/O-Aktion

- hat einen `Typ IO a`
  `getLine` hat den Typ `IO String`,
  `getChar` hat den Typ `IO Char`,
  `print "bcd"` hat den Typ `IO ()`
- wird nicht immer, und nicht immer sofort, ausgeführt wenn ausgewertet.
  In ghci aufrufen:

```
:{
let iotest = do z <- getLine
                print ("OK " ++ z )
:}
iotest
```

- hat einen Nebeneffekt wenn ausgeführt.

Daten ein- und ausgeben

Die Ausführung einer Aktion vom Typ `IO a`

- kann, muss aber nicht, zur einer Ein- oder Ausgabe führen,
- gibt irgendwann einen Wert vom `Typ a` zurück.

Aktionen werden in Monaden definiert.

Die I/O-Aktionen, d.h. die Ein- und Ausgaben, werden in der Monade `IO` definiert.

Daten ein- und ausgeben

Das folgende in ghci aufrufen:

```
let b = do {c <- getLine; print c}
:t b
```

Jeder Ausdruck (mit passendem Typ) kann als Wert eine Aktion haben aber eine solche Aktion wird erst innerhalb einer Aktion (wie z.B. `main` oder oben `b`) ausgeführt.

Dateien einlesen und Daten in Dateien ausgeben

Mit dem Programm-Editor die folgende Datei `waslernstdu1.hs` erstellen:

```
-- waslernstdu1.hs
main :: IO ()
main = do
    putStrLn "Was lernst du?"
    sprache <- getLine
    putStrLn ("Viel Erfolg beim " ++ sprache ++ " lernen!")
```

Im Terminal das Folgende ausführen:

```
ghc --make waslernstdu1.hs
./waslernstdu1
runghc waslernstdu1.hs
```

Dateien einlesen und Daten in Dateien ausgeben

Mit dem Programm-Editor eine Datei namens `antworthaskell.txt` mit dem folgenden Inhalt erstellen:

```
Haskell
```

Im Terminal das folgende ausführen:

```
./waslernstdu1 < antworthaskell.txt
echo Haskell | runghc waslernstdu1.hs
```

Dateien einlesen und Daten in Dateien ausgeben

```
-- waslernstdu2.hs
import Control.Monad

main :: IO ()
main = do
    putStrLn "Was lernst du?"
    sprache <- getLine
    when (sprache == "Haskell") $ do {putStrLn "Gute Wahl!"}
    putStrLn ("Viel Erfolg beim " ++ sprache ++ " lernen!")
```

Im Terminal ausführen:

```
ghc --make waslernstdu2.hs
./waslernstdu2
```

Dateien einlesen und Daten in Dateien ausgeben

```
-- waslernstdu3.hs
main :: IO ()
main = do
    putStrLn "Was lernst du?"
    sprache <- getLine
    if sprache == "Haskell"
        then putStrLn "Gute Wahl!"
        else putStrLn "Denk daran, auch Haskell zu lernen!"
    putStrLn ("Viel Erfolg beim " ++ sprache ++ " lernen!")
```

Im Terminal ausführen:

```
ghc --make waslernstdu3.hs
./waslernstdu3
```

Dateien einlesen und Daten in Dateien ausgeben

Statt:

```
putStrLn ("Viel Erfolg beim "++ sprache ++ "lernen!")
```

kann man auch schreiben:

```
putStrLn $ "Viel Erfolg beim "++ sprache ++ "lernen!"
```

`($)` ist ein Operator mit Typ `(a -> b) -> a -> b`

Ein- und Ausgabe-Aktionen

In ghci ausführen:

```
do {a <- print "Wer bist du?" ; print a}
```

Die Variable a ist nutzlos, weil ihr Wert () ist.

Ein- und Ausgabe-Aktionen

In ghci ausführen:

```
do {a <- print "Wer bist du?"}
```

Wird abgelehnt, weil der Wert des letzten Ausdrucks eine Aktion sein muss.

```
do {a <- print "Wer bist du?"; getLine }
```

Der Wert eines do-Blockes ist der Wert der letzten Aktion des do-Blockes.

Folglich benötigt Haskell kein return à la C, Java, JavaScript oder Python.

Ein- und Ausgabe-Aktionen

Haskells `return`

- dient nicht dazu, einem do-Block einen Wert zu geben;
- dient dazu, einen Wert in eine Aktion zu packen.

Ein- und Ausgabe-Aktionen

```
-- palindrom.hs
main :: IO ()
main = do
  z <- getLine
  if null z -- beendet die Schleife
      then return () --tue-nichts-Aktion
      else do
          if istPalindrom z
              then putStrLn "ist ein Palindrom"
              else putStrLn "ist kein Palindrom"
          main -- rekursive Aufruf fuer die Schleife

istPalindrom w = w == reverse w
```

Quiz `return`

Das `return` von Haskell ist anders aber erfüllt eine ähnliche Funktion wie das `return` von C, Java, JavaScript oder Python.

A. korrekt
B. falsch

Antworten: `https://lmu.onlineted.de`

Ein- und Ausgabe-Aktionen

Anders in anderen Sprachen `return` beendet nicht die Ausführung eines do-Blocks.

```
-- nurreturn.hs
main :: IO ()
main = do
    _ <- return "a"
    b <- return "Ende"
    print b
```

Der Übersetzer warnt vor ungenutzten Variablen nicht, die mit _ anfangen.

Im Terminal ausführen:

```
ghc nurreturn.hs
./nurreturn
```

Ein- und Ausgabe-Aktionen

`return` und `<-` sind zueinander symmetrisch:

- `return "bc"` bringt das String `"bc"` in eine Aktion hinein.
- `<- getLine` holt ein String aus der Aktion `getLine` heraus.

Quiz `return` und `<-` I

Was gibt das folgende Programm aus?

```
main = do
    vorname <- return "Anna"
    nachname <- return "Abel"
    putStrLn (vorname ++ " " ++ nachname)
```

A. Anna Abel

B. nichts

Antworten: https://lmu.onlineted.de

Quiz `return` und `<-` II

Wirken `vorname <- return "Bernd"` und `vorname = "Bernd"` gleich?

A. Ja
B. Nein

Antworten: https://lmu.onlineted.de

Ein- und Ausgabe-Aktionen

```
-- quelle.txt
abc
```

```
-- ziel.txt
def
```

```
-- vonquellezumziel1.hs
import System.IO

main :: IO()
main = do
    leseGriff <- openFile "quelle.txt" ReadMode
    inhalt <- hGetContents leseGriff
    schreibGriff <- openFile "ziel.txt" AppendMode
    hPutStr schreibGriff inhalt
    hClose leseGriff
    hClose schreibGriff
```

Ein- und Ausgabe-Aktionen

Testen:

```
ghc vonquellezumziel1.hs
./vonquellezumziel1
more quelle.txt
more ziel.txt
```

ApendMode durch WriteMode ersetzen und testen

```
data IOMode = ReadMode | WriteMode | AppendMode |
              ReadWriteMode
```

Ein- und Ausgabe-Aktionen

`hGetContents leseGriff` liefert die Chars der Datei nach und nach, aber erst wenn sie benötigt werden
(„verzögerte Auswertung“ oder „Lazy Evaluation“).

Das Laufzeitsystem löscht die gelesene Daten, so bald sie nicht länger benötigt werden
(„Speicherbereinigung“ oder „Garbage Collection“).

Ein- und Ausgabe-Aktionen

```
-- vonquellezumziel2.hs
import System.IO

main :: IO()
main = do
    inhalt <- readFile "quelle.txt"
    writeFile "ziel.txt" inhalt
```

Das Laufzeitsystem schließt selbständig die Dateien `quelle.txt` und `ziel.txt`.

Ein- und Ausgabe-Aktionen

```
-- myprogramm.hs
import System.Environment
import Data.List()

main :: IO()
main = do
    progName <- getProgName
    args <- getArgs
    putStrLn "Programm-Name:"
    putStrLn progName
    putStrLn "Programm-Argumente:"
    mapM_ putStrLn args
    -- irgendetwas berechnen
```

`mapM_` wendet `putStrLn` auf jedes Element der List `args` an, was eine Sequenz von Aktionen ergibt, und gibt den Wert `()` zurück.

Ein- und Ausgabe-Aktionen

Im Terminal das Programm `myprogramm.hs` testen:

```
ghc myprogramm.hs
./myprogramm a1 a2 a3
```

Zusammenfassung der heutigen Vorlesung

2 Ein- und Ausgaben

- Daten ein- und ausgeben
- Dateien einlesen und Daten in Dateien ausgeben (Programme kompilieren und auswerten)
- Ein- und Ausgabe-Aktionen