Heutiges Thema

Ein- und Ausgabe, Einstieg in Monaden, Fehlerbehandlung...

- Speziell...
  - Ein- und Ausgabe in Haskell
- Allgemeiner...
  - Programmierung mit Monaden
- Zum Abschluss...
  - Fehlerbehandlung

⇝ Schnittstelle von *funktionaler* und *imperativer* Programmierung!

Hello World!

```
helloWorld :: IO ()
helloWorld = putStr "Hello World!"
```

Hello World...

- ...gewöhnlich eines der ersten Beispielprogramme in einer neuen Programmiersprache
- ...in dieser LVA erst im letzten Drittel!

Ungewöhnlich?

Zum Vergleich...

Ein-/Ausgabe-Behandlung in...

- S. Thompson [3]: ...in Kapitel 18 (von 20)
- P. Pepper [4]: ...in Kapitel 21&22 (von 23)
- R. Bird [2]: ...in Kapitel 10 (von 12)
- A. J. T. Davie. "*An Introduction to Functional Programming Systems Using Haskell*", Cambridge, 1992. ...in Kapitel 7 (von 11)
- M. M. T. Chakravarty, G. C. Keller. "*Einführung in die Programmierung mit Haskell*", Pearson Studium, 2004. ...in Kapitel 7 (von 13)

Zufall?

...oder ist Ein-/Ausgabe möglicherweise

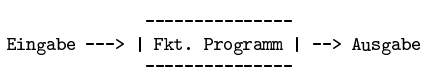
- ...weniger wichtig in funktionaler Programmierung?
- ...oder in besonderer Weise problembehaftet?

Tendenziell letzteres...

- Ein-/Ausgabe... führt uns an den Berührungspunkt von *funktionaler* und *imperativer* Programmierung!

Rückblick...

Unsere bisherige Sicht fkt. Programmierung...



In anderen Worten...

Unsere bisherige Sicht fkt. Programmierung ist...

- *stapelverarbeitungsfokussiert*, nicht
- *dialog-* und *interaktionsorientiert*

...wie es heutigen Anforderungen und heutiger Programmierrealität entspricht.

Erinnerung

Im Vergleich zu anderen Paradigmen...

- Das funktionale Paradigma betont das "was" (Ergebnisse) zugunsten des "wie" (Art der Berechnung der Ergebnisse)

Von zentrale Bedeutung dafür...

- Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab
  Stichwort(*e*): ...Kompositionalität (ref. Transparenz)
  ⇝ erleichtert Programmentwicklung und Korrektheitsüberlegungen
- Auswertungsabhängigkeiten, nicht aber Auswertungsreihenfolgen dezidiert festgelegt
  Stichwort(*e*): ...Flexibilität (Church-Rosser-Eigenschaft)
  ⇝ erleichtert Implementierung einschl. Parallelisierung
- ...

Angenommen...

...wir hätten Konstrukte der Art (Achtung: Kein Haskell!)

```
PRINT :: String -> a -> a
PRINT message value =
        << gib "message" am Bildschirm aus und liefere >>
        value

READFL :: Float
READFL = << lies Gleitkommazahl und liefere diese als Ergebnis >>
```

Mithin:

...Hinzunahme von Ein-/Ausgabe durch seiteneffektbehaftete Funktionen!

Knackpunkt 1: Kompositionalität (1)

Vergleiche...

```
val :: Float
val = 3.14

valDiff :: Float
valDiff = val - val
```

mit...

```
readDiff :: Float
readDiff = READFL - READFL
```

und der Anwendung in...

```
constFunOrNot :: Float
constFunOrNot = valDiff + readDiff
```

Knackpunkt 1: Kompositionalität (2)

Beachte: ...der Wert von Ausdrücken hinge nicht länger nur von seinen Teilausdrücken ab (sondern auch von der Position im Programm)

$\rightsquigarrow$ ...Verlust von *Kompositionalität*

(...und der damit einhergehenden positiven Eigenschaften).

Knackpunkt 2: Flexibilität (1)

...oder die Unabhängigkeit von der Auswertungsreihenfolge

Vom "Punkt"...

```
punkt r =
  let
  myPi = 3.14
  x    = r * myPi
  y    = r + 17.4
  z    = r * r
  in (x,y,z)
```


Knackpunkt 2: Flexibilität (2)

...zum "Knackpunkt" (Achtung: Kein Haskell!):

```
knackpunkt r =
  let
  myPi = PRINT "Constant Value" 3.14
  u    = PRINT "Erstgelesener Wert" dummy
  c    = READFL
  x    = r * c
  v    = PRINT "Zweitgelesener Wert" dummy
  d    = READFL
  y    = r + d
  z    = r * r
  in (x,y,z)
```

$\rightsquigarrow$ ...Verlust von *Auswertungsreihenfolgenunabhängigkeit*

Ergo...

Konzentration auf die Essenz der Programmierung wie im funktionalen Paradigma ("was" statt "wie") ist wichtig und richtig, aber...

- Kommunikation mit dem Benutzer (bzw. der Außenwelt) muss die zeitliche Abfolge von Aktivitäten auszudrücken gestatten.

Weiter in den Worten von P. Pepper [4]...

- ..." *der Benutzer lebt in der Zeit und kann nicht anders als zeitabhängig sein Programm beobachten*".

Konsequenz ...man (bzw. ein jedes Paradigma) darf von der Arbeitsweise des Rechners, nicht aber von der des Benutzers abstrahieren!

Haskells Ansatz zur Behandlung von Ein- und Ausgabe

Zentral...

- Elementare Ein-/Ausgabeoperationen (Kommandos) auf speziellen Typen (IO-Typen) sowie
- (Kompositions-) Operatoren, um Anweisungssequenzen (Kommandosequenzen) auszudrücken

Damit...

- Trennung von
  - funktionalem Kern und
  - imperativähnlicher Ein-/Ausgabe

...somit gelangen wir an die Schnittstelle von funktionaler und imperativer Welt!

(Ausgewählte) elementare Ein-/Ausgabeoperationen

```
-- Eingabe
getChar :: IO Char
getLine :: IO String

-- Ausgabe
putChar :: Char -> IO ()
putLine :: String -> IO ()
putStr  :: String -> IO ()
```

Bemerkung:

- `()`: ...spezieller einelementiger Haskell-Typ, dessen einziges Element (ebenfalls) mit `()` bezeichnet wird.
- `IO a`: ...spezieller Haskell-Typ "*I/O Aktion* (*Kommando*) *vom Typ* `a`". `IO`: ...Typkonstruktor (ähnlich wie `[a]` für Listen)

Kompositionsoperatoren

```
(>>)  :: IO a -> IO b -> IO b
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Intuitiv:

- `(>>)`: Wenn `p` und `q` Kommandos sind, dann ist `p >> q` das Kommando, das zunächst `p` ausführt, den Rückgabewert (`x` vom Typ `a`) ignoriert, und anschließend `q` ausführt.
- `(>>=)`: Wenn `p` und `q` Kommandos sind, dann ist `p >>= q` das Kommando, das zunächst `p` ausführt, dabei den Rückgabewert `x` vom Typ `a` liefert, und daran anschließend `q x` ausführt und dabei den Rückgabewert `y` vom Typ `b` liefert.

Einfache Anwendungsbeispiele

```
-- Schreiben mit Zeilenvorschub (Standardoperation in Haskell)
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")

-- Lesen einer Zeile und Ausgeben der gelesenen Zeile
echo :: IO ()
echo = getLine >>= putLine
```

Weitere Ein-/Ausgabeoperationen

```
-- Schreiben und Lesen von Werten unterschiedlicher Typen
print :: Show a => a -> IO ()
print = putStrLn . show

read :: Read a => String -> a

-- Rueckgabewerterzeugung ohne Ein-/Ausgabe(aktion)
return :: a -> IO a
```

Erinnerung:

```
show :: Show a => a -> String
```


Die do-Notation: Bequemere (Kommando-) Sequenzenbildung

Komfortabler als (>>) und (>>=), ist Haskells do-Notation...

```
putStrLn :: String -> IO ()
putStrLn str = do putStr str
                  putStr "\n"

putTwice :: String -> IO ()
putTwice str = do putStrLn str
                  putStrLn str

putNtimes :: Int -> String -> IO ()
putNtimes n str = if n <= 1
                    then putStrLn str
                    else do putStrLn str
                            putNtimes (n-1) str
```


Weitere Beispiele zur do-Notation

```
putTwice = putNtimes 2

read2lines :: IO ()
read2lines = do getLine
                getLine
                putStrLn "Two lines read."

echo2times :: IO ()
echo2times = do line <- getLine
                putLine line
                putLine line

getInt :: IO Int
getInt = do line <- getLine
            return (read line :: Int)
```


"Single vs. updatable Assignment" (1)

Durch das Konstrukt

```
var <- ...
```

wird stets eine *frische* Variable eingeführt.

Sprechweise:

...Unterstützung des Konzepts des

- "single assignment", nicht das des
- "updatable assignment" (wie aus traditionellen imperativen Programmiersprachen bekannt)

"Single vs. updatable Assignment" (2)

...zur Illustration des Unterschieds betrachte:

```
goUntilEmpty :: IO ()
goUntilEmpty = do line <- getLine
                  while (return (line /= []))
                        (do putStrLn line
                            line <- getLine
                            return () )
```

wobei `while :: IO Bool -> IO () -> IO ()`

Abhilfe: ...Rekursion statt Iteration!

"Single vs. updatable Assignment" (3)

...wie etwa in S. Thompson [3] auf S. 393 vorgeschlagen ist:

```
goUntilEmpty :: IO ()
goUntilEmpty =
    do line <- getLine
       if (line == [])
          then return ()
          else (do putStrLn line
                   goUntilEmpty)
```


Stichwort: Iteration

```
while :: IO Bool -> IO () -> IO ()

while test action
  = do res <- test
       if res then do action
                      while test action
              else return ()             -- "null I/O-action"
```

Erinnerung:

```
-- Rueckgabewerterzeugung ohne Ein-/Ausgabe(aktion)
return :: a -> IO a
```


Ein-/Ausgabe von und auf Dateien

...auch hierfür vordefinierte Standardoperatoren.

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

where

```
type FilePath = String     -- implementationsabhaengig

-- Anwendungsbeispiel: Bestimmung der Laenge einer Datei
size :: IO Int
size = do putLine "Dateiname = "
          name <- getLine
          text <- readFile name
          return(length(text))
```

Zusammenhang do-Konstrukt und (>>), (>>=)-Operatoren

...illustriert anhand eines Beispiels:

Mittels do...

```
incrementInt :: IO ()
incrementInt = do line <- getLine
                  putStrLn (show (1 + read line :: Int))
```

Äquivalent dazu...

```
incrementInt = getLine >>= \line ->
               putStrLn (show (1 + read line :: Int))
```

Intuitiv...

"do = (>>=) + anonyme lambda-Abstraktion"

"Programmierregel"

- Hauptdefinition (übersetzter) Haskell-Programme ist (per Konvention) eine Definition `main` vom Typ `IO a`.

Beispiel:

```
main :: IO ()
main = do c <- getChar
          putChar c
```

...`main` ist Startpunkt eines (übersetzten) Haskell-Programms. (intuitiv somit: "Programm = Ein-/Ausgabekommando")

Fazit über Ein- und Ausgabe

Es gilt...

- Ein-/Ausgabe grundsätzlich unterschiedlich in funktionaler und imperativer Programmierung

  Am augenfälligsten:

  - *Imperativ*: Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich
  - *Funktional*: Ein-/Ausgabe an bestimmten Programmstellen konzentriert

Häufige Beobachtung...

- ...die vermeintliche Einschränkung erweist sich oft als Stärke bei der Programmierung im Großen!

Zusammenhang "Ein-/Ausgabe und Monade in Haskell"

...nächstes Mal!

Fehlerbehandlung in Haskell

- Bislang von uns nur rudimentär behandelt.

  Typische Formulierung aus den Aufgabenstellungen:

  ...so hat die Funktion als Ergebnis das (-2)-adische Produkt dieser beiden Zahlen; ansonsten ist das Ergebnis die Zeichenreihe `Ungueltige Eingabe`.

- In der Folge Wege zu einem systematischeren Umgang mit unerwarteten Programmsituationen und Fehlern

Typische Fehlersituationen

- Division durch 0
- Zugriff auf das erste Element einer leeren Liste
- ...

In der Folge...

- 3 Varianten zum Umgang mit solchen Situationen

Variante 1 (1)

- *Berechnung anhalten und Fehlerursache melden.*

Hilfsmittel: Die Funktion `error`...

```
error :: String -> a
```

Aufruf von...

```
error "Unbehebbarer Fehler aufgetreten..."
```

liefert Ausgabe...

```
Program error: Unbehebbarer Fehler aufgetreten...
```

und Programmausführung stoppt.

Variante 1 (2)

Vor- und Nachteile von Variante 1:

- Schnell und einfach
- *Aber*: Die Berechung stoppt unwiderruflich. Jegliche (auch) sinnvolle Information über den Programmlauf ist verloren.

Ziel: Kein Panikmodus. Programmlauf nicht gänzlich abbrechen.

Variante 2 (1)

- *Verwendung von dummy-Werten im Fehlerfall.*

Statt...

```
tail :: [a] -> [a]
tail (_:xs) = xs
tail []     = error ``PreludeList.tail: empty list''
```

benutze zum Beispiel...

```
tl :: [a] -> [a]
tl (_:xs) = xs
tl []     = []
```


Variante 2 (2)

Vor- und Nachteile von Variante 2:

- Programmlauf wird nicht abgebrochen
- *Aber*: Ein geeigeter Default-Wert ist nicht immer offensichtlich.

  Betrachte z.B.:

```
hd :: [a] -> a
hd (x:_) = x
hd []    = ???????????
```

Möglicher Ausweg:

- ...jeweils gewünschten Wert als Parameter mitgeben.

  Im obigen Beispiel etwa:

```
hdy :: a -> [a] -> a
hdy y (x:_) = x
hdy y []    = y
```

Variante 2 (3)

Generelles Muster:

Ersetze übliche Implementierung einer (einstelligen) Funktion `f`...

```
f x = ...
```

durch...

```
fErr y x
  | cond      = y
  | otherwise = f x
```

mit `cond` Charakterisierung der Fehlersituation.

Vor- und Nachteile:

- Generell; stets anwendbar
- Auftreten des Fehlerfalls nicht beobachtbar: `y` mag auch als gewöhnliches Ergebnis auftreten

Variante 3 (1)

- *Fehlerwerte und -typen statt schlichter dummy-Werte*

```
data Maybe a = Nothing | Just a
               deriving (Eq, Ord, Read, Show)
```

...i.w. der Typ `a` mit dem Zusatzwert `Nothing`.

Damit...

```
fErr x
  | cond      = Nothing
  | otherwise = Just (f x)
```

...und anhand eines konkreten Beispiels:

```
errDiv :: Int -> Int -> Maybe Int
errDiv n m
  | (m /= 0)  = Just (n 'div' m)
  | otherwise = Nothing
```


Variante 3 (2)

Vor- und Nachteile von Variante 3:

- Geänderte Funktionalität: statt `a`, jetzt `Maybe a`
- *Aber*:
  - Fehlerursachen können durch einen Funktionsaufruf "hindurchgereicht" werden (der Effekt der Funktion `mapMaybe`...)
  - Fehler können "gefangen" werden (die Rolle von der Funktion `maybe`...)

Variante 3 (3)

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b

mapMaybe g Nothing  = Nothing
mapMaybe g (Just x) = Just (g x)


maybe :: b -> (a -> b) -> Maybe a -> b

maybe n f Nothing  = n
maybe n f (Just x) = f x
```


Variante 3 (4)

Anwendungsbeispiel(e):

Der Fehler wird "(auf-) gefangen":

```
maybe 42 (+1) (mapMaybe (*3) errDiv 9 0))
= maybe 42 (+1) (mapMaybe (*3) Nothing)
= maybe 42 (+1) Nothing
= 42
```

Kein Fehlerfall, "alles geht gut":

```
maybe 42 (+1) (mapMaybe (*3) errDiv 9 1))
= maybe 42 (+1) (mapMaybe (*3) (Just 9))
= maybe 42 (+1) (Just 27)
= 1 + 27
= 28
```


Variante 3 (5)

Wesentlicher Vorteil der letzten Variante:

- Systementwicklung ohne explizite Fehlerbehandlung möglich.
- Fehlerbehandlung kann am Ende mithilfe der Funktionen `mapMaybe` und `maybe` ergänzt werden.

...für weitere Details siehe S. Thompson [3], Kapitel 14.3.

Das FP-Team wünscht Ihnen...

Ein frohes Weihnachtsfest

und

einen guten Rutsch ins neue Jahr!

Aufwiedersehen im neuen Jahr am Di, 18.01.2005, 13:00 Uhr s.t. - 14:00 Uhr, im TreitI-Hörsaal!