

Funktionale Programmierung

LVA 185.A03, VU 2.0, ECTS 3.0
WS 2020/2021

Vortrag VI
Orientierung, Einordnung
02.12.2020

Jens Knoop



Technische Universität Wien
Information Systems Engineering
Compilers and Languages



Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Vortrag VI

Orientierung, Einordnung

...zum selbstgeleiteten, eigenständigen Weiterlernen.

Teil VI: Weiterführende Konzepte

- Kapitel 15: Interaktive Programme: Ein-/Ausgabe
- Kapitel 16: Robuste Programme: Fehlerbehandlung
- Kapitel 17: Programmierung im Großen: Module
- Kapitel 18: Programmierprinzipien – ...am 16.12.2020

Teil VI

Weiterführende Konzepte

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Kapitel 15

Interaktive Programme: Ein-/Ausgabe

Kapitel 15.1

Motivation

Votr. VI

Teil VI

Kap. 15

15.1

15.1.1

15.1.2

15.1.3

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Kapitel 15.1.1

Problem und Ziel

Votr. VI

Teil VI

Kap. 15

15.1

15.1.1

15.1.2

15.1.3

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehr

Klassen-

zim-

mer V

Modusänd

Test 1

Hinweis

Aufgabe

Das Problem: Dialog findet nicht statt!

...unsere Programme sind bislang **stapelverarbeitungsorientiert**:

- ▶ **Eingabedaten** müssen **zu Programmbeginn vollständig** zur Verfügung gestellt werden.
- ▶ **Einmal gestartet**, besteht **keine Möglichkeit** mehr, mit weiteren Eingaben auf das Verhalten oder Ergebnisse des Programms **zu reagieren** und es **zu beeinflussen**.

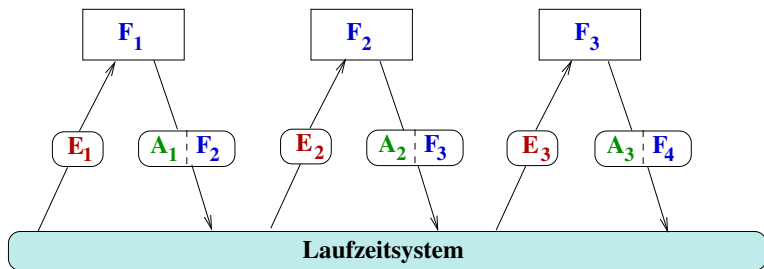


Peter Pepper. *Funktionale Programmierung*. Springer-Verlag, 2003, S. 245.

...**Dialog, Interaktion** zwischen Benutzer und Programm findet (bisher) nicht statt.

Das Ziel: Wir hätten gerne auch

...dialog- und interaktionsorientierte Haskell-Programme:



Peter Pepper. *Funktionale Programmierung*.
Springer-Verlag, 2003, S. 253.

Kapitel 15.1.2

Herausforderung

Votr. VI

Teil VI

Kap. 15

15.1

15.1.1

15.1.2

15.1.3

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehr

Klassen-

zim-

mer V

Modusänd

Test 1

Hinweis

Aufgabe

Herausforderung

...Auflösung eines scheinbar unauflöslchen **Widerspruchs**: Der

- ▶ Umgang m. Seiteneffekten in einer seiteneffektlosen Welt!

Konstituierendes Kennzeichen

- ▶ **rein** funktionaler Programmierung:
 - **Vollkommene Abwesenheit** von Seiteneffekten!

- ▶ **Ein-/Ausgabe**:
 - **Unvermeidbare Anwesenheit** von Seiteneffekten!
Ein- und Ausgabe, lesen und schreiben verändern den Zustand der äußeren Welt **notwendig** und **irreversibel**.

Wichtig: Das gilt **paradigmenunabhängig!** Ein- und Ausgabe erzeugen **Seiteneffekte** **notwendig**, **unvermeidbar**, sind **ohne Seiteneffekte** nicht vorstellbar!

Verzicht auf Ein-/Ausgabe ist keine Option!

“Der Benutzer lebt in der Zeit und kann nicht anders als zeitabhängig sein Programm beobachten.”

Peter Pepper. **Funktionale Programmierung.**
Springer-V., 2. Auflage, 2003.

...wir können **abstrahieren** von der Arbeitsweise

- ▶ des **Rechners**
- ▶ nicht aber von der des **Benutzers**.

Die Ermöglichung **dialog-, interaktionsorientierter Ein-/Ausgabe** ist deshalb unverzichtbar, bringt uns an die Nahtstelle

▶ von **reiner funktionaler** und **imperativer** Programmierung und erfordert sie zu **überschreiten**.

Kapitel 15.1.3

Warum (naive) Einfachheit versagt

Votr. VI

Teil VI

Kap. 15

15.1

15.1.1

15.1.2

15.1.3

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehr

Klassen-

zim-

mer V

Modusänd

Test 1

Hinweis

Aufgabe

Ohne Ein-/Ausgabe gilt:

Ist

$x = f \text{ <komplexer_ausdruck>}$

$y = f \text{ <komplexer_ausdruck>}$

$z = (g \ x) + (g \ y)$

Fragment eines rein fkt. Programms (z.B. in Haskell), gilt:

1. Die beiden Aufrufe von f liefern denselben Wert, unabhängig von der Wahl von $\text{<komplexer_ausdruck>}$.
2. x und y haben denselben Wert.
3. Die beiden Aufrufe von g liefern denselben Wert.
4. Der Wert von z ist unabhängig von der Reihenfolge, in der die beiden Aufrufe von g ausgewertet werden.
5. Der Wert von z ist gleich der Summe der Werte von $g \ x$ und $g \ y$, aber auch gleich dem Doppelten des Werts von $g \ x$ und gleich dem Doppelten des Werts von $g \ y$:

$z = (g \ x) + (g \ y) = 2 * (g \ x) = 2 * (g \ y)$

Was ändert sich nur durch Eingabe?

...in einem um Eingabe erweiterten fkt. Programm:

$x = \text{READ_INT}$

$y = \text{READ_INT}$

$z = (g\ x) + (g\ y)$

gilt hingegen:

1. Die beiden Aufrufe von `READ_INT` liefern i.a. **nicht denselben Wert**.
2. x und y haben i.a. **nicht denselben Wert**.
3. Die beiden Aufrufe von `g` liefern i.a. **nicht denselben Wert**.
4. Der Wert von z ist **nicht unabhängig von der Reihenfolge**, in der die beiden Aufrufe von `g` ausgewertet werden.
5. Der Wert von z ist **gleich der Summe** der Werte von `g x` und `g y`, aber i.a. **nicht gleich dem Doppelten** des Werts von `g x` u. **nicht gleich dem Doppelten** d. Werts von `g y`:

$$z = (g\ x) + (g\ y) \stackrel{i.a.}{\neq} 2 * (g\ x) \stackrel{i.a.}{\neq} 2 * (g\ y)$$

Woran liegt das?

Die Hinzunahme seiteneffektbehafteter Operationen in eine fkt. Sprache wie Ein-/Ausgabe führt zum Verlust von

- ▶ referentieller Transparenz

...und damit zum Verlust einer Reihe von Gewissheiten:

- ▶ Die Unveränderbarkeit des Zustands der äußeren Welt (Seiteneffektfreiheit).
- ▶ Der Wert eines Ausdrucks hängt nur vom Wert seiner Teilausdrücke ab (Kompositionalität), nicht von der Reihenfolge ihrer Auswertung (Reihenfolgenunabhängigkeit).
- ▶ Der Wert eines Ausdrucks ist unveränderlich über die Zeit (Zeitunabhängigkeit); er verändert sich nicht durch die Anzahl seiner Auswertungen (Auswertungshäufigkeitsunabhängigkeit).
- ▶ Ein Ausdruck darf stets durch seinen Wert ersetzt werden und umgekehrt (Austauschbarkeit).

Die Hinzunahme von Ein-/Ausgabeoperationen

...in fkt. Sprachen stellt damit auch ein weiteres leitendes **Prinzip reiner funktionaler** (und allgemeiner **deklarativer**) Programmierung infrage:

- ▶ Die Betonung des **'was'** (Ergebnisse) statt des **'wie'** (die Art ihrer Berechnung)

und rüttelt damit an den **Grundfesten**, auf die sich

- ▶ **reine funktionale** Programmierung

gründet und von denen sich ihre **Stärke** und **Eleganz** ableitet.

Kapitel 15.2

Haskells Lösung

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.2.1

15.2.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Kapitel 15.2.1

Konzeption und Umsetzung

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.2.1

15.2.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

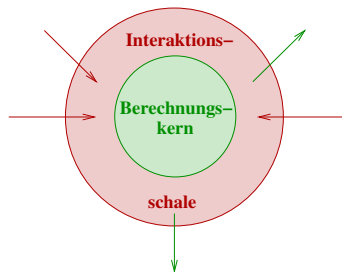
Aufgabe

Konzeptuelle E/A-Lösung Haskell's

Konzeptuell wird in **Haskell** ein Programm geteilt in

- einen **rein funktionalen Berechnungskern**
- eine **imperativähnliche Dialog- und Interaktionsschale**.

zwischen denen mittels vordefinierter besonderer Ein-/Ausgabefunktionen Daten geschützt ausgetauscht werden können:



Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson, 2004, S. 89.

Haskells Umsetzung d. E/A-Lsg. im Überblick

- A)** Ein (vordef.) polymorpher Datentyp für Ein-/Ausgabe:
- `data IO a = ...` (Details implementierungsintern versteckt)
- B)** Vordefinierte primitive E/A-Operationen:
- `getChar :: IO Char`
`getInt :: IO Int`
...
 - `putChar :: Char -> IO ()`
`putInt :: Int -> IO ()`
...
- C)** Ein Operator zur Komposition von E/A-Operationen:
- `(>>=) :: IO a -> (a -> IO b) -> IO b`
 - 'Syntaktischer Zucker' für `(>>=)`: `do`-Notation.
- D)** Zwei Vermittlungsoperatoren zwischen Schale und Kern:
- `return :: a -> IO a`
 - `'<-' :: IO a -> a'` (\rightsquigarrow informell!)

Lösungsbeiträge d. Umsetzungsbestandteile (1)

A) Trennung in rein funktionalen Berechnungskern und imperativartige Dialog- und Interaktionsschale:

Der Datentyp `(IO a)` erlaubt die Unterscheidung von Typen

- ▶ des rein funktionalen Berechnungskerns (`Char`, `Int`, `Bool`, etc.)
- ▶ der imperativartigen Dialog- und Interaktionsschale (`(IO Char)`), `(IO Int)`, `(IO Bool)`, etc.)

...`IO`-Werte bleiben in der Schale und können nicht den rein funktionalen Kern 'kontaminieren'.

Lösungsbeiträge d. Umsetzungsbestandteile (2)

B) Vordefinierte primitive E/A-Operationen

...als Bausteine, aus denen komplexe(re) Ein-/Ausgabeoperationen gebaut werden können.

C) Festlegung der zeitlichen Abfolge von E/A-Operationen (“Der Benutzer lebt in der Zeit...und kann nicht anders...”):

Der Kompositionsoperator ($\gg=$) (oder gleichwertig die do-Notation, s. Kap. 15.4) erlauben die präzise Festlegung der

- ▶ zeitlichen Abfolge von E/A-Operationen.

Lösungsbeiträge d. Umsetzungsbestandteile (3)

D) Verbindung von funktionalem Kern und E/A-Schale

- ▶ **return**: Von **Kern** in **Schale** (in **äußere Welt**).

`return :: a` → `IO a`
 Kern nach Schale

`return` erlaubt **rein funktionale** Werte (engl. **pure values**) aus dem funktionalen Kern über die Schale als **seiten-effektverursachende** Werte (engl. **impure values**) in die äußere Welt zu transferieren.

...**'kontaminieren'** reiner Werte.

- ▶ **<-**: Von **Schale** (von **äußerer Welt**) in **Kern**.

`<- :: IO a` → `a`
 Schale nach Kern

`<-` erlaubt den **'reinen' Anteil** (`a`-Wert) **seiteneffektverursachender** Werte (`(IO a)`-Wert) aus der äußeren Welt in den funktionalen Kern zu transferieren.

...**'dekontaminieren'** E/A-verschmutzter Werte.

Kapitel 15.2.2

Aktionen

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.2.1

15.2.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Ein-/Ausgabe in Aktion

Lesen einer (Zeichenreihen-) Zeile vom Bildschirm (`getLine`), Schreiben einer Zeichenreihe auf den Bildschirm (`putStr`), zwei vordefinierte E/A-Aktionen in Haskell:

```
getLine :: IO String
putStr  :: String -> IO ()
```

E/A in Aktion: Lies eine Zeile vom Bildschirm, schreib die gelesene Zeile wieder auf den Bildschirm und mache einen Zeilenvorschub:

```
getLine >>= (\zeile -> putStr zeile) >>= (\_ -> putStr "\n")
:: IO String :: String :: IO () :: String :: IO ()
:: String
:: ()
```

...für das zweite Vorkommen von (`>>=`) können wir gleichbedeutend (`>>`) zur Komposition nehmen und erhalten etwas kürzer:

```
getLine >>= (\zeile -> putStr zeile) >> putStr "\n"
```

Beobachtung

...die **Komposition von E/A-Aktionen** mittels der

- Kompositionsoperatoren ($\gg=$) und (\gg)

ist notationell umständlich und sieht nicht 'schön' aus.

Praktischer ist **Haskell's do-Notation!**

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.2.1

15.2.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Haskells do-Notation: Syntaktischer Zucker

...zum Ersatz der Kompositionsoperatoren ($\gg=$) und (\gg) zur Komposition von Ein-/Ausgabeaktionen:

```
do zeile <- getLine
   _ <- putStr zeile
   _ <- putStr "\n"
```

was abgekürzt werden kann zu:

```
do zeile <- getLine
   putStr zeile
   putStr "\n"
```

statt:

```
getLine >>= (\zeile -> putStr zeile) >>= (\_ -> putStr "\n")
getLine >>= (\zeile -> putStr zeile) >> putStr "\n"
```

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.2.1

15.2.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Zur Deutlichkeit

...auch für die `do`-Notation mit ergänzter Typinformation:

```
do zeile <- getLine
  :: String :: IO String
  _ <- putStr zeile
      :: String
  :: ()      :: IO ()
  _ <- putStr "\n"
      :: String
  :: ()      :: IO ()
  :: IO ()
```

...der Typ der letzten Aktion bestimmt den Typ des (gesamten) `do`-Ausdrucks, d.h. ein `do`-Ausdruck hat als Typ den Typ seiner letzten Aktion.

Was steckt hinter den Aktionen nun genau?

... ???

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.2.1

15.2.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Aktionen sind Ausdrücke vom Typ (IO a)

Ausdrücke vom Typ (IO a)

- ▶ sind **wertliefernde** ('funktionaler' Anteil) **E/A-Operationen** ('prozeduraler' Anteil).
- ▶ bewirken einen **Lese-** oder **Schreibseiteneffekt** (prozedurales Verhalten) **und** liefern einen **a-Wert** als Ergebnis (**funktionales** Verhalten), der eingepackt als (IO a)-Wert zur Verfügung gestellt wird.
- ▶ heißen **Aktionen** (oder **Kommandos**) (engl. **actions** oder **commands**).

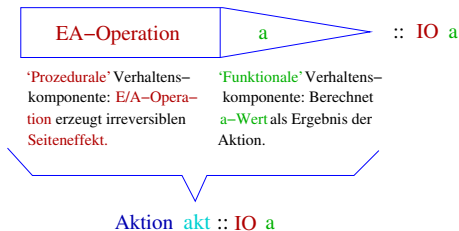
Informell:

$$\begin{aligned} \text{Aktion} &= (1) \text{ E/A-Operation ('prozedural')} \\ &\quad + (2) \text{ Wertlieferung ('funktional')} \\ &= \text{wertliefernde E/A-Operation} \end{aligned}$$

Veranschaulichung des Effekts von Aktionen

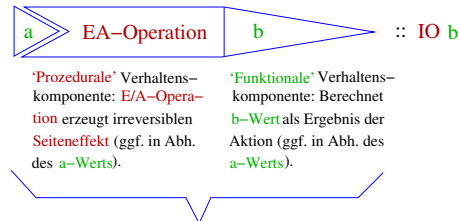
Aktion $\text{akt} :: \text{IO } a$

(z.B. getLine)



Aktion liefernde Fkt. $\text{f_akt} :: a \rightarrow \text{IO } b$

(z.B. putStr)



Typ

...aller **Leseaktionen** ist

- ▶ **(IO a)** (für 'lesegeeignete' Typinstanzen von **a**).

Der in einen **a**-Wert transformierte gelesene Wert wird als (formal erforderliches und inhaltlich gewolltes) Ergebnis von Leseoperationen verwendet.

...aller **Schreibaktionen** ist

- ▶ **(IO ())** mit **()** der einelementige **Nulltupeltyp** mit gleichbenanntem einzigen Datenwert **()**.

() als (einziger) Wert des Nulltupeltyps **()** wird als **formal erforderliches** Ergebnis von Schreiboperationen verwendet.

Auswertung, Ausführung von Aktionen

Wegen des kombinierten

1. **prozeduralen** (seiteneffekterzeugende Lese-/Schreiboperation) und
2. **funktionalen** (Wert als Ergebnis liefernden)

Effekts der Auswertung von **Aktionen** (oder **E/A-Ausdrücken**), spricht man statt von **Auswertung** oft von **Ausführung** von **Aktionen** (oder **E/A-Ausdrücken**).

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.2.1

15.2.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Interpretation der Signatur von ($\gg=$)

...des Kompositionsoperators ($\gg=$):

▶ ($\gg=$) :: $\text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b$

Die Signatur liefert:

- ▶ ($\gg=$) ist eine Abbildung, die eine (Argument-) Aktion mit einem a -Wert als Ergebnis (d.h. einen $(\text{IO } a)$ -Wert) auf eine (Bild-) Aktion mit einem b -Wert als Ergebnis abbildet (d.h. auf einen $(\text{IO } b)$ -Wert) mithilfe einer Funktion, deren Ergebnis angewendet auf den a -Ergebniswert der Argumentaktion die gesuchte Bildaktion ist.

Interpretation der Signatur von (\gg)

...des Kompositionsoperators (\gg):

▶ (\gg) :: IO a -> IO b -> IO b

Die Signatur liefert:

- ▶ (\gg) ist eine Abbildung, die eine (Argument-) Aktion mit einem a-Wert als Ergebnis (d.h. einen (IO a)-Wert) und eine zweite (Argument-) Aktion mit einem b-Wert als Ergebnis (d.h. einen (IO b)-Wert) auf diese zweite Aktion als Bildaktion abbildet.

(Scheinbar hat das erste Argument keine Bedeutung und verschwindet; dies gilt für sein funktionales Ergebnis, den a-Wert, nicht aber für seinen prozeduralen Lese-/Schreibseiteneffekt!)

Interpretation der Signatur von `return`

...der aktionsliefernden Funktion `return`:

▶ `return :: a -> IO a`

Die Signatur liefert:

▶ `return` ist eine Abbildung, die einen `a`-Wert auf eine Aktion mit einem `a`-Wert als Ergebnis abbildet (d.h. auf einen `(IO a)`-Wert).

Operationelle Bedeutung

...des Kompositionsoperators ($\gg=$):

► $(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$

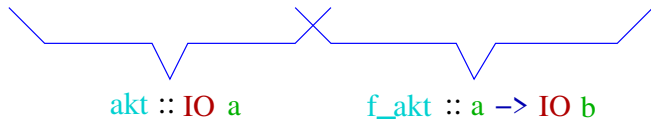
Sei $(akt :: IO\ a)$ eine Aktion, $(f_akt :: a \rightarrow IO\ b)$ eine Aktion liefernde Abbildung.

Operationelle Bedeutung der Komposition $(akt \gg= f_akt)$:

- akt wird ausgeführt, bewirkt dabei einen Lese- oder Schreibseiteneffekt und liefert als Ergebnis einen a -Wert; dieser a -Wert wird zum Argument von f_akt , deren Bildwert vom Typ $(IO\ b)$ eine Aktion ist, die ausgeführt wird, dabei einen weiteren Lese- oder Schreibseiteneffekt bewirkt und als Ergebnis einen b -Wert liefert; dieser ist zugleich das (funktionale) Ergebnis der Komposition $(akt \gg= f_akt)$.

Veranschaulichung der operat. Bedeutung

...der komponierten Aktion ($\text{akt} \gg= \text{f_akt}$):



$$\text{akt} \gg= \text{f_akt} \hat{=} \text{akt} \gg= \backslash x \rightarrow \text{f_akt } x$$

Operationelle Bedeutung

...des Kompositionsoperators (\gg):

► $(\gg) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$

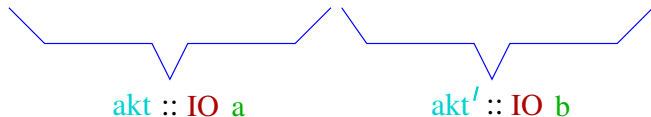
Seien $(akt :: IO\ a)$, $(akt' :: IO\ b)$ zwei Aktionen.

Operationelle Bedeutung der Komposition: $(akt \gg akt')$:

- akt wird ausgeführt, bewirkt dabei einen Lese- oder Schreibseiteneffekt und liefert als Ergebnis einen a -Wert. Dieser a -Wert wird ignoriert und unmittelbar die Aktion akt' ausgeführt, die dabei einen weiteren Lese- oder Schreibseiteneffekt bewirkt und als Ergebnis einen b -Wert liefert; dieser ist zugleich das (funktionale) Ergebnis der Komposition $(akt \gg akt')$.

Veranschaulichung der operat. Bedeutung

...der komponierten Aktion ($\text{akt} \gg \text{akt}'$):



$$\text{akt} \gg \text{akt}' \hat{=} \text{akt} \gg= _ \rightarrow \text{akt}'$$

Komposition: 'binde-dann'-, 'dann'-Operator

Die Kompositionsoperatoren

- ▶ $(>>=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$
- ▶ $(>>) :: IO\ a \rightarrow IO\ b \rightarrow IO\ b$
 $akt \gg akt' = akt \gg= \backslash_ \rightarrow akt'$ (vordefiniert)

...gelesen als

- ▶ binde-dann-Operator (engl. `bind` oder `then`)
- ▶ dann-Operator (engl. `sequence`).

Bem.: Die Definition von $(>>)$ macht deutlich, dass $(>>)$ kein eigenständiger Operator, sondern von $(>>=)$ abgeleitet und eine spezielle Anwendung von $(>>=)$ ist, die das Ergebnis von `akt` (`a`-Wert) als Argument für `akt'` (`_ \rightarrow akt'`) ignoriert: Der `a`-Wert von `akt` wird anders als bei $(>>=)$ nicht für weitere Verwendung an einen Namen gebunden, er wird 'vergessen'.

Operationelle Bedeutung

...der Funktion `return`:

► `return` :: `a` -> `IO a`

Sei (`w` :: `a`) ein `a`-Wert.

Operationelle Bedeutung des aktionsliefernden Ausdrucks
(`return w`):

► `return` bildet den `a`-Wert `w` in 'offensichtlicher' Weise auf den 'entsprechenden' (`IO a`)-Wert ab, ohne einen Lese- oder Schreibseiteneffekt zu bewirken.

(Das **prozedurale** Verhalten von `return` entspricht der leeren Anweisung '*skip*'; `return` hat (deshalb) abweichend von anderen Aktionen nur ein **funktionales** beobachtbares Verhalten, kein prozedurales).

Veranschaulichung

...der operationellen Bedeutung von `return`:



'Prozedurale' Verhaltenskomponente: 'Leer'; keine E/A-Operation, kein Seiteneffekt.

'Funktionale' Verhaltenskomponente: Reicht den `a-Wert` als Ergebnis der Aktion durch.

Aktion `return` :: `a` \rightarrow IO `a`

Wichtig zu beachten

Die E/A-Aktion `return` in Haskell

- ▶ hat eine gänzlich andere Aufgabe und Bedeutung als das aus imperativen oder objektorientierten Sprachen bekannte `return`; außer der Namensgleichheit besteht weder konzeptuell noch funktionell eine Ähnlichkeit.
- ▶ Haskell's `return` kann in einer Aktionssequenz auftreten und ausgewertet werden, ohne dass dadurch die Auswertung der restlichen Aktionssequenz beendet würde; `return` kann deshalb auch mehrfach in sinnvoller Weise in einer Aktionssequenz auftreten.
- ▶ Zum Verständnis von Haskell's `return` ist eine Orientierung am imperativen, objektorientierten `return` deshalb nicht sinnvoll und allenfalls irreführend.

Kapitel 15.3

E/A-Primitive für Bildschirm- und Datei- Ein-/Ausgabe

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Bildschirmein-/ausgabe: Vordef. E/A-Primitive

...Lesen und Schreiben vom bzw. auf den Bildschirm.

Leseoperationen:

```
getChar  :: IO Char
getInt   :: IO Int
getline  :: IO String
readIO   :: Read a => String -> IO a
readLn   :: Read a => IO a
...
```

Schreiboperationen:

```
putChar  :: Char -> IO ()
putStr   :: String -> IO ()
putStrLn :: String -> IO ()
print    :: Show a => a -> IO ()
...
```

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Dateiein-/ausgabe: Vordef. E/A-Primitive

...Lesen und Schreiben aus bzw. in Dateien.

Leseoperationen:

```
readFile :: FilePath -> IO String
```

Schreiboperationen:

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

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

Dateiende-Prädikat:

```
isEOF :: FilePath -> Bool
```

Pfad-/Dateinamen:

```
type FilePath = String
```

...mit betriebssystemabhängigen Werten für `FilePath`.

E/A-Operationen und die Fkt. show, read

Mithilfe der Fkt. `show` der Typklasse `Show` und der globalen Fkt. `read` (Achtung: `read` keine Fkt. der Typklasse `Read`!):

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

...lassen sich Werte von Instanztypen der Typklasse `Show` ausgeben und von Instanztypen der Typklasse `Read` einlesen:

```
putLine :: Show a => a -> IO ()
```

```
putLine = putStrLn . show
```

```
print :: Show a => a -> IO ()
```

```
print = putLine
```

```
read :: Read a => String -> a
```

```
read s = ... -- definiert im Präludium
```

Bem.: Vordefinierte Instanzen von `Show` und `Read`: Alle im Präludium definierten Typen mit Ausnahme v. Funktions- u. `IO`-Typen.

Konstruktion von E/A-Sequenzen

...mit

1. Funktionskomposition (.)

Schreiben mit Zeilenvorschub (vordefinierte Sequenz):

```
putStrLn :: String -> IO ()  
putStrLn = putStr . (++ "\n")
```

2. IO-Komposition (>>=)

Lesen einer Zeile und anschließendes Schreiben der
gelesenen Zeile (selbstdefinierte Sequenz):

```
echo :: IO ()  
echo = getLine >>= (\zeile -> putLine zeile)
```

Beispiel: Hallo, Welt!

```
halloWelt :: IO ()  
halloWelt = putStrLn "Hallo, Welt!"
```

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Kapitel 15.4

Syntaktischer Zucker: Die do-Notation

Allgemeines Muster von do-Ausdrücken

```
do w1 <- akt1      -- Sprechweise: akti Generator
   w2 <- akt2      -- für Wert wi vom Typ ai
   ...
   wn <- aktn
   return (f w1 w2 ... wn)
```

mit Verknüpfungsfunktion vom Typ:

```
f :: a1 -> a2 -> ... -> an -> b
```

und Aktionen der Typen:

```
akt1 :: IO a1
akt2 :: IO a2
...
aktn :: IO an
```

Bedeutungsgleich auch in einer Zeile mittels ';' möglich:

```
do w1 <- akt1; ... ; wn <- aktn; return (f w1 w2 ... wn)
```

Die Bedeutung

...des `do`-Ausdrucks:

```
do w1 <- akt1
   w2 <- akt2
   ...
   wn <- aktn
return (f w1 w2 ... wn)
```

ist definiert durch den `(>>=)`-Ausdruck:

```
akt1 >>= \p1 ->
akt2 >>= \p2 ->
...
aktn >>= \pn ->
return (f p1 p2 ... pn)
```

Der Typ eines do-Ausdrucks

...ist durch den Typ seiner letzten Aktion bestimmt:

```
( do w1 <- akt1
  w2 <- akt2
  ...
  wn <- aktn
  return (f w1 w2 ... wn) )
```

$:: IO\ b$

für Verknüpfungsfunktion vom Typ:

```
f :: a1 -> a2 -> ... -> an -> b
```

Nicht verwendete Aktionsergebnisse

...in `do`-Ausdrücken.

Aktionen liefern stets ein **Ergebnis**. Bleibt es unverwendet (entspricht Aktionskomposition mit `(>>)` statt mit `(>>=)`), kann die Nichtverwendung syntaktisch dadurch ausgedrückt werden, dass ein Aktionsergebnis nicht an einen Wertnamen `wi`, sondern an `_` 'gebunden' wird, quasi 'unbenannt' gebunden wird:

```
do w1 <- akt1
   _  <- akt2
   _  <- akt3
   w4 <- akt4
   ...
   wn <- aktn
return (f w1 w4 ... wn)
```

Vortr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehr

Klassen-

zim-

mer V

Modusänd

Test 1

Hinweis

Aufgabe

Unbenannte Bindungen

...können ganz einfach auch völlig entfallen:

```
do w1 <- akt1
   akt2
   akt3
   w4 <- akt4
   ...
   wn <- aktn
return (f w1 w4 ... wn)
```


Bsp.: Ausgabe-Sequenzen mittels do-Notation

Einmaliges Schreiben einer Zeichenreihe mit Zeilenvorschub:

```
putStrLn :: String -> IO ()           -- Definition aus
putStrLn str = do putStrLn str        -- Präludium
                putStrLn "\n"
```

Zweimaliges Schreiben einer Zeichenreihe (mit Z-Vorschüben):

```
putStrLn_2mal :: String -> IO ()
putStrLn_2mal str = do putStrLn str
                    putStrLn str
```

Viermaliges Schreiben einer Zeichenreihe (mit Z-Vorschüben):

```
putStrLn_4mal :: String -> IO ()
putStrLn_4mal str = do putStrLn str
                    putStrLn str
                    putStrLn str
                    putStrLn str
```

Vortr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmer V

Modusänderung
Test 1

Hinweis

Aufgabe

Bsp.: Ein-/Ausg.-Sequenzen mittels do-Notat.

Zwei Lese-, eine Schreibaktion:

```
read2lines_and_report :: IO ()
read2lines_and_report
= do getLine      -- Z. wird gelesen u. vergessen
     getLine      -- Z. wird gelesen u. vergessen
     putStrLn "Zwei Zeilen gelesen."
```

Eine Lese-, zwei Schreibaktionen:

```
read1line_and_echo2times :: IO ()
read1line_and_echo2times
= do line <- getLine -- Z. w. gelesen u. gemerkt
     putStrLn line   -- Gemerkte Z. w. geschrieben
     putStrLn line   -- Gemerkte Z. w. geschrieben
```

Bsp.: Ausg.-Sequenzen parametrisierter Länge

n-maliges Schreiben einer Zeichenreihe (mit Z-Vorschüben):

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

Das erlaubt auch folgende (alternative) Definitionen:

```
putStrLn_2mal :: String -> IO ()
putStrLn_2mal = putStrLn_nmal 2

putStrLn_4mal :: String -> IO ()
putStrLn_4mal = putStrLn_nmal 4
```

Bsp.: do-Ausdrücke mit return (1)

Lesen einer Zeichenreihe vom Bildschirm und Konversion in eine ganze Zahl:

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

Im Detail:

```
getInt :: IO Int
getInt = do line <- getLine
           return (read line :: Int)
           Konvertierung 'String' (der
           Typ von line) zu 'Int' (der
           Argumenttyp von return)
           :: IO Int
           :: IO Int
```

Bsp.: do-Ausdrücke mit return (2)

Bestimmung der Länge, der Zeichenzahl einer Datei:

```
groesse :: IO Int
groesse = do putStrLn "Dateiname = "
             name <- getLine
             text <- readFile name
             return (length text)
```

Bsp.: do-Ausdrücke mit return (3)

Mit detaillierter Typinformation:

```
groesse :: IO Int
groesse = do putStrLn "Dateiname = "
            name    <-    getLine
            :: String      :: IO String
            text    <-    readFile name
            :: String      :: IO String
            return (length text)
                    :: String
                    :: Int
                    :: IO Int
                    :: IO Int
```

Bsp.: Bedeutungsgleiche E/A-Sequenzen

...mit (`>>`) und `do`.

Die **Ausgabe-Sequenz** mit (`>>`):

```
writeFile "meineDatei.txt" "Hallo, Dateisystem!"  
>> putStr "Hallo, Welt!"
```

...entspricht der **Ausgabe-Sequenz** mit `do`:

```
do writeFile "meineDatei.txt" "Hallo, Dateisystem!"  
  putStr "Hallo, Welt!"
```

und definiert deren Bedeutung.

Bsp.: Bedeutungsgleiche E/A-Sequenzen

...mit ($\gg=$) und `do`.

Die E/A-Sequenz mit ($\gg=$):

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

...entspricht der E/A-Sequenz mit `do`:

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

und definiert deren Bedeutung.

Informell: 'do' entspricht ' $\gg=$ ' plus anonyme λ -Abstraktion'.

Bsp.: Bedeutungsgleiche E/A-Sequenzen

...mit (`>>=`) und `do`, `return`.

Die Eingabe-Sequenz mittels (`>>=`) und `return`:

```
readStringPair :: IO (String,String)
readStringPair
  = getLine >>=
    (\zeile -> (getLine >>=
                (\zeile' -> (return (zeile,zeile')))))
```

...entspricht der Eingabe-Sequenz mit `do` und `return`:

```
readStringPair' :: IO (String,String)
readStringPair'
  = do zeile <- getLine
      zeile' <- getLine
      return (zeile,zeile')
```

und definiert deren Bedeutung.

Kapitel 15.5

E/A-Programmbeispiele

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweise

Aufgabe

Kapitel 15.5.1

Dialog- und Interaktionsprogramme

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweis

Aufgabe

Dialog- und Interaktionsprogramme

Zwei Frage/Antwort-Interaktionen mit dem Benutzer:

```
ask :: String -> IO String
ask frage = do putStrLn frage
            getLine
```

```
interAct :: IO ()           -- Bildschirm-Interaktion
interAct
= do name <- ask "Wie heißen Sie?"
     putStrLn ("Willkommen " ++ name ++ "!!")
```

```
interAct' :: IO ()         -- Datei-Interaktion
interAct'
= do putStrLn "Bitte Dateinamen angeben: "
     dateiname <- getLine
     inhalt    <- readFile dateiname
     putStrLn inhalt
```

Vortr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrt
Klassen-
zim-
mer V

Modusäne
Test 1

Hinweis

Aufgabe

Lokale Deklarationen in do-Ausdrücken

Die **E/A-Sequenz** (ohne lokale Deklarationen):

```
reverse2lines :: IO ()
reverse2lines
  = do line1 <- getLine
       line2 <- getLine
       putStrLn (reverse line2)
       putStrLn (reverse line1)
```

...ist **bedeutungsgleich** zur **Sequenz** mit **lokalen Deklarationen**:

```
reverse2lines :: IO ()
reverse2lines
  = do line1 <- getLine
       line2 <- getLine
       let rev1 = reverse line1
           rev2 = reverse line2
       putStrLn rev2
       putStrLn rev1
```

Unterschiedliche Bindung von `<-` und `let`

Benannte Wertvereinbarungen mittels

- ▶ `<-`: für den `a`-Wert von **Aktionen** vom Typ `(IO a)` (für 'unreine' Werte aus der **äußeren Welt!**).
- ▶ `let`: für den Wert **rein funktionaler Ausdrücke** (für 'reine' Werte aus dem **rein funktionalen Programmkernel**).

Kapitel 15.5.2

Rekursive E/A-Programme

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweis

Aufgabe

Rekursive E/A-Programme (1)

...lesen und schreiben gelesener Eingaben: **Kopieren**.

Nichtterminierendes Kopieren (Notabbruch mit **Ctrl-c**):

```
kopiere :: IO ()
kopiere
  = do zeile <- getLine
      putStrLn zeile
      kopiere                -- Rekursion!
```

n-maliges Kopieren:

```
kopiere_n_mal :: Int -> IO ()
kopiere_n_mal n
  = if n <= 0
      then return ()
      else do zeile <- getLine
              putStrLn zeile
              kopiere_n_mal (n-1)    -- Rekursion!
```


Rekursive E/A-Programme (2)

Kopieren bis zur Eingabe der leeren Zeile:

```
kopiere_bis_leer :: IO ()
kopiere_bis_leer
= do zeile <- getLine
     if zeile == ""
     then return ()
     else do putStrLn zeile
             kopiere_bis_leer           -- Rekursion!
```

Kopieren bis zur Eingabe der leeren Zeile unter Mitzählen:

```
kopiere_bis_leer_und_zaehle_mit :: Int -> IO ()
kopiere_bis_leer_und_zaehle_mit n
= do zeile <- getLine
     if zeile == ""
     then putStrLn
          (show n ++ " Zeilen gelesen u. kopiert.")
     else do putStrLn zeile
             kopiere_bis_leer_und_zaehle_mit (n+1)
```

Vortr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrt
Klassen-
zimmerV

Modusänderung
Test 1

Hinweis

Aufgabe

73/194

Rekursive E/A-Programme (3)

Summieren einer Folge ganzer Zahlen bis 0 eingegeben wird:

```
summiere :: IO Int
summiere
= do n <- getInt
    if n == 0
    then return 0
    else (do m <- summiere
          return (n + m))
```

Vergleiche d. strukturelle Ähnlichkeit von `summiere`, `sum`, `sum'`:

```
sum :: [Int] -> Int
sum [] = 0
sum (n:ns)
= let m = sum ns
  in (n + m)

sum' :: [Int] -> Int
sum' [] = 0
sum' (n:ns)
= n + sum' ns
```

Rekursive E/A-Programme (4)

Interaktives Summieren einer Folge ganzer Zahlen bis 0 eingegeben wird, abgestützt auf `summiere`:

```
summiere_interaktiv :: IO ()
summiere_interaktiv
= do putStrLn "Gib ganze Zahl ein, je eine pro"
     putStrLn "Zeile. Diese werden summiert bis"
     putStrLn "Null eingegeben wird."
     summe <- summiere
     putStr "Der Summenwert ist "
     putLine summe
```

Vortrag VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmerV

Modusänderung
Test 1

Hinweis

Aufgabe

Kapitel 15.5.3

Iterativartige E/A-Programme

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweis

Aufgabe

Iterativartige E/A-Programme

Iterativartiger Ausdruck/Programm, genauer die iterativartige Funktion `while`:

```
while :: IO Bool -> IO () -> IO ()
while bedingung aktion
  = do b <- bedingung
      if b
      then
        do aktion
            while bedingung aktion -- Rekursion!
      else
        return ()
```

Vortr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweis

Aufgabe

Zur operationellen Bedeutung der Fkt. while

Intuitiv:

- ▶ Ist die Bedingung (`bedingung :: IO Bool`) erfüllt (und hat `b` somit den Wert `True`), so wird die Aktion (`aktion :: IO ()`) ausgeführt (do-Ausdruck im then-Ausdruck); anderenfalls endet die Ausführung/-wertung von `while` ohne weiteren E/A-Seiteneffekt mit dem Resultatwert `() :: ()`.
- ▶ Nach abgeschlossener Ausführung/-wertung von `aktion` (im Fall der erfüllten Bedingung) wird `while` rekursiv aufgerufen, wodurch insgesamt die 'iterativartige' Anmutung entsteht, dass eine `Aktion` so lange ausgeführt wird, wie eine `Bedingung` erfüllt ist.
- ▶ Mögliches Argument für die Bedingung: Der Ausdruck `isEOF :: IO Bool` zum Test auf das Eingabeende.

Vortr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrt
Klassen-
zim-
mer V

Modusän-
Test 1

Hinweis

Aufgabe

Anwendung der Funktion `while`

...um eine Datei zeilenweise zu lesen und gelesene Zeilen wieder auszugeben, bis das Dateiende erreicht ist.

```
kopiere_eingabe_nach_ausgabe :: IO ()
kopiere_eingabe_nach_ausgabe
  = while (do wert <- isEOF           -- Arg. f. Param.
           return (not wert))       -- bedingung
         (do zeile <- getLine        -- Arg. f. Param.
           putStrLn zeile)         -- aktion
```

Bem.: Die Klammerung der Argumente von `while` ist nötig.

Kapitel 15.5.4

'Iteration' vs. Rekursion

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweis

Aufgabe

Wertvereinbarung vs. Wertzuweisung

...funktionale Wertvereinbarung vs. imperative Wertzuweisung.

Zur Natur des

- ▶ Wertvereinbarungsoperators ' \leftarrow ' in **do**-Ausdrücken

im Vergleich zum

- ▶ destruktiven Wertzuweisungsoperator ' $:=$ ' in destruktiven Zuweisung(sanweisung)en (engl. destructive assignments) imperativer Sprachen.

Tatsächlich besitzt

- ▶ ' \leftarrow ' Ähnlichkeit mit einer Wertzuweisung, ist aber **gänzlich verschieden** der destruktiven Wertzuweisung imperativer Sprachen.

Einmal-Wertvereinbarungsoperator '`<-`'

'`<-`' leistet eine **Einmal-Wertvereinbarung** für einen **Namen**:

- ▶ `zeile <- getLine` bindet das Resultat von `getLine` (allgemeiner: einer Eingabeoperation), an einen Namen, hier `zeile`.
- ▶ Diese **Verbindung** zwischen dem **Namen**, hier `zeile`, und dem von einer Eingabeoperation gelieferten **Wert**, hier `getLine`, bleibt für den gesamten Programmablauf erhalten und ist **nicht** mehr **veränderbar**.

Mehrfach-Wertzuweisungsoperator ‘:=’

‘:=’ leistet eine temporäre Wertzuweisung an eine durch einen Namen bezeichnete Speicherzelle:

- ▶ `x := READ_STRING`: Der von `READ_STRING` gelesene Wert wird in die von `x` bezeichnete Speicherzelle geschrieben; der vorher dort gespeicherte Wert wird dabei überschrieben und zerstört (**destruktiv!**).
- ▶ Die durch die Zuweisung geschaffene Verbindung zwischen Name (d.h. der mit ihm bezeichneten Speicherzelle) und Wert (d.h. dem Inhalt der Speicherzelle) bleibt so lange erhalten (**temporär!**), bis sie durch eine erneute Zuweisung an diese Zelle überschrieben und zerstört wird (**destruktive Zuweisung!**).
- ▶ Der Inhalt einer Speicherzelle kann jederzeit und beliebig oft überschrieben werden und so die Verbindung von Name und Wert geändert werden (s. a. Anhang A.6).

Zur Wirkung von Einmal-Wertvereinbarungen

...anhand eines **Beispiels**:

Aufgabe: Schreibe ein Programm, das so lange eine Zeile vom Bildschirm einliest und wieder ausgibt, bis schließlich die leere Zeile eingelesen wird und die Ausführung abgebrochen wird.

Vortr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.5.1

15.5.2

15.5.3

15.5.4

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Der Effekt von Einmal-Wertvereinbarungen

'Iterativer' Lösungsversuch mittels `while`-Funktion/Ausdrucks:

```
goUntilEmpty :: IO ()
goUntilEmpty
  = do zeile <- getLine
      while           -- while mit Argumenten:
        (return (zeile /= [])) -- Bedingungsarg.
        (do putStrLn zeile      -- Aktionsarg.
            zeile <- getLine
            return ())
```

- ▶ Die Auswertung von `goUntilEmpty` terminiert nicht (es sei denn, `[]` wird als erste Eingabe gewählt).
- ▶ `zeile` und `zeile` sind unterschiedliche Einmal-Wertvereinbarungen gleichen Namens.
- ▶ Test und Ausgabe erfolgen bei jedem Aufruf von `while` (in jeder 'Schleife') für den Wert von `zeile`, nie v. `zeile`.

Lösung: Direkte Rekursion statt 'Iteration'

Direkt-rekursive Lösung (ohne den iterativartigen Ausdruck `while`):

```
goUntilEmpty' :: IO ()
goUntilEmpty'
  = do zeile <- getLine
      if (zeile == [])
      then return ()
      else (do putStrLn zeile
              goUntilEmpty')      -- Rekursion!
```

(siehe Simon Thompson. [The Craft of Functional Programming](#). Addison-Wesley/Pearson, 2. Auflage, 1999, S. 393.)

Kapitel 15.6

Zusammenfassung

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Haskell-Programme als E/A-Aktionen

Einstiegspunkt für die Auswertung (übersetzter) interaktiver Haskell-Programme ist (per Konvention) eine eindeutig bestimmte

- Definition mit Namen `main` vom Typ `(IO T)`, `T` Typ.
- Intuitiv: 'Haskell-Programm = E/A-Aktion'.

Beispiel:

```
main :: IO ()           -- E/A-Schale
main
  = do n <- getInt      -- E/A-Schale
      let ergebnis = meine_funktion n -- Fkt. Kern
          putStr ergebnis           -- E/A-Schale
meine_funktion :: Int -> String    -- Fkt. Kern
meine_funktion n = ... meine_funktion' ...
meine_funktion' :: ...           -- Fkt. Kern
meine_funktion' ... = ...
...
```

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Ein-/Ausgabebehandlung

...in **funktionaler** und **imperativer** Programmierung grundsätzlich unterschiedlich. Am augenfälligsten:

- ▶ **Imperativ**: Ein-/Ausgabe prinzipiell an jeder Programmstelle möglich.
- ▶ **Funktional**, hier in **Haskell**: Ein-/Ausgabe an bestimmten Programmstellen konzentriert (in meist wenigen global definierten Funktionen der **'E/A-Schale'**).

Häufige Beobachtung: Die vermeintliche Einschränkung erweist sich

- ▶ als **Stärke** bei der **Programmierung im Großen!**

Kapitel 15.7

Leseempfehlungen

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Basiselesempfehlungen für Kapitel 15

-  Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015. (Kapitel 10.1, The IO monad)
-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 17.5, Ein- und Ausgaben)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 7, Eingabe und Ausgabe)
-  Ernst-Erich Doberkat. *Haskell: Eine Einführung für Objektorientierte*. Oldenbourg Verlag, 2012. (Kapitel 5, Ein-/Ausgabe; Kapitel 5.1, IO-Aktionen)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 8, Input and output; Kapitel 9, More input and more output)

Votr. VI

Teil VI

Kap. 15

15.1

15.2

15.3

15.4

15.5

15.6

15.7

Kap. 16

Kap. 17

Umgekehrt

Klassen-
zim-
mer V

Modusän

Test 1





Hinweis

Aufgabe

Weiterführ. Leseempfehlungen für Kap. 15 (1)

-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 21, Ein-/Ausgabe: Konzeptuelle Sicht; Kapitel 22, Ein-/Ausgabe: Die Programmierung)
-  Peter Pepper, Petra Hofstedt. *Funktionale Programmierung: Sprachdesign und Programmieretechnik*. Springer-V., 2006. (Kapitel 18, Objekte und Ein-/Ausgabe)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 7, I/O; Kapitel 9, I/O Case Study: A Library for Searching the Filesystem)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 8, Playing the game: I/O in Haskell; Kapitel 18, Programming with monads)

Weiterführ. Leseempfehlungen für Kap. 15 (2)

-  Antonie J. T. Davie. *An Introduction to Functional Programming Systems using Haskell*. Cambridge University Press, 1992. (Kapitel 7.5, Input/Output in Functional Programming)
-  Andrew J. Gordon. *Functional Programming and Input/Output*. British Computer Society Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.
-  Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000. (Kapitel 16, Communicating with the Outside World)
-  Graham Hutton. *Programming in Haskell*. Cambridge University Press, 2. Auflage, 2016. (Kapitel 10, Interactive programming)

Ein Genie macht keine Fehler. Seine Irrtümer
sind Tore zu neuen Entdeckungen.

James Joyce (1882-1941)
irisch. Schriftsteller

...für alle anderen:

Kapitel 16

Robuste Programme: Fehlerbehandlung

Votr. VI

Teil VI

Kap. 15

Kap. 16

16.1

16.2

16.3

16.4

16.5

Kap. 17

Umgekehrt

Klassen-

zim-

mer V

Modusände

Test 1

Hinweis

Aufgabe

Ein Mensch würde nie dazu kommen,
etwas zu tun, wenn er stets warten würde,
bis er es so gut kann, dass niemand mehr
einen Fehler entdecken könnte.

John Henry Newman (1801-1890)
engl. Kardinal

Kapitel 16.1

Überblick, Orientierung

Kleine Fehler in einem großen Werk sind die
Brosamen, die man dem Neid hinwirft.

Claude Adrien Helvétius (1715-1771)
franz. Philosoph

Typische Fehlersituationen und Sonderfälle

Typische Fehlersituationen:

- ▶ Division durch null: `div 1 0`.
- ▶ Zugriff auf das erste Element einer leeren Liste: `head []`.
- ▶ ...

Typische Sonderfälle:

- ▶ Auseinanderfallen von **intendiertem** und **implementiertem Definitionsbereich** einer Funktion, z.B.
 - `! : IN -> IN`: **Intendierter Definitionsbereich** ist **IN**.
 - `fac :: Integer -> Integer`: **Implementierter Definitionsbereich** ist **\mathbb{Z}** (abgesehen von Ressourcenbeschränkungen der Maschine).
- ▶ Umgang mit Argumentwerten außerhalb des **intendierten** Definitionsbereichs.
- ▶ ...

Um anderer Leute Fehler zu sehen, verwandeln manche Menschen ihre Augen in Mikroskope.

Georg Christoph Lichtenberg (1742-1799)
dt. Physiker und Naturforscher

Fehlersituationen und Sonderfälle

...bislang von uns *unsystematisch*, *naiv* behandelt:

Typische Formulierungen aus den Aufgabenstellungen:

...liefert die Funktion den vorher beschriebenen Wert als Resultat; anderenfalls...

- ▶ *ist das Ergebnis*
 - die Zeichenreihe *"Ungültige Eingabe"*.
 - die leere Liste `[]`.
 - der Wert `0`.
 - ...
- ▶ *endet die Berechnung mit dem Aufruf*
error "Ungültige Eingabe".
- ▶ ...

Jeder Fehler erscheint unheimlich dumm,
wenn andre ihn begehen.

Georg Christoph Lichtenberg (1742-1799)
dt. Physiker und Naturforscher

In diesem Kapitel

...beschreiben wir drei Möglichkeiten eines sukzessive **systematisch(er)en Umgangs** mit unerwarteten Programmsituationen und Fehlern:

1. **Panikmodus** (Kap. 16.2)
2. **Auffangwerte** (engl. `default values`) (Kap. 16.3)
 - 2.1 Funktionsspezifisch
 - 2.2 Aufrufspezifisch
3. **Fehlertypen, Fehlerwerte, Fehlerfunktionen** (Kap. 16.4)

Fremde Fehler haben wir vor Augen,
unsere liegen uns im Rücken.

Seneca der Jüngere (um 4 v.Chr. - 65 n.Chr.)
röm. Politiker, Philosoph und Schriftsteller

Kapitel 16.2

Panikmodus

Votr. VI

Teil VI

Kap. 15

Kap. 16

16.1

16.2

16.3

16.4

16.5

Kap. 17

Umgekehr

Klassen-

zim-

mer V

Modusände

Test 1

Hinweis

Aufgabe

Panikmodus

Ziel:

1. Fehler und Fehlerursache melden
2. Fehlerhafte Programmauswertung stoppen.

Werkzeug:

- ▶ Die polymorphe Funktion `error :: String -> a`.

Wirkung:

Der Aufruf

- `error "Funktion f: Ungültige Eingabe."`

liefert die Meldung

- `Programmfehler: Funktion f: Ungültige Eingabe.`

und stoppt danach die Programmauswertung unwiderruflich.

Beispiel

```
fac :: Integer -> Integer
fac n
  | n == 0    = 1
  | n > 0     = n * fac (n-1)
  | otherwise = error "Ungültige Eingabe."
```

Verhalten in Aufrufsituationen:

```
fac 5    ->> 120
fac 0    ->> 1
fac (-5) ->> Programmfehler: Ungültige Eingabe.
fac (-7) ->> Programmfehler: Ungültige Eingabe.
```

Bewertung des Panikmodus

Positiv:

- + Schnell und einfach umzusetzen.

Negativ:

- Die Berechnung stoppt unwiderruflich.
- Jegliche Information über den Programmablauf ist verloren, auch sinnvolle.
- Für sicherheitskritische Systeme können die Folgen eines unbedingten Programmabbruchs fatal sein.

Kapitel 16.3

Auffangwerte

Votr. VI

Teil VI

Kap. 15

Kap. 16

16.1

16.2

16.3

16.4

16.5

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusände-
rung
Test 1

Hinweis

Aufgabe

Auffangwerte

Ziel:

1. Panikmodus vermeiden.
2. Programmablauf nicht zur Gänze abbrechen, sondern Berechnung möglichst sinnvoll fortführen.

Werkzeug:

1. Funktionsspezifische (Variante 1)
2. Aufrufspezifische (Variante 2)

Auffangwerte (engl. **default values**) zur Weiterrechnung im Fehlerfall.

Variante 1: Funktionsspezifischer Auffangwert

...im Fehlerfall wird ein

► **funktionsspezifischer Wert**

als Resultat geliefert.

Beispiel:

```
fac :: Integer -> Integer
fac n
  | n == 0    = 1
  | n > 0    = n * fac (n-1)
  | otherwise = -1
```

Verhalten in Aufrufsituationen:

fac 5 ->> 120

fac 0 ->> 1

fac (-5) ->> -1

fac (-7) ->> -1

(**Verschiedene Argumente**,
gleicher Fehlerwert)

Analyse des Beispiels

Im **Beispiel** von `fac` gilt:

- ▶ Negative Werte treten **nie als reguläres Resultat** einer Berechnung auf.
- ▶ Der **funktionspezifische Auffangwert `-1`** erlaubt deshalb, negative Eingaben als fehlerhaft zu erkennen und zu melden, ohne den Programmablauf unwiderruflich abubrechen.

Insgesamt:

- ▶ Die Fehlersituation ist für den Programmierer **transparent**.

Bewertung von Auffangwertvariante 1

Positiv

- + Panikmodus vermieden, Programmablauf nicht abgebrochen.

Negativ

- Oft gibt es einen zwar naheliegenden und plausiblen funktionsspezifischen Auffangwert; jedoch kann dieser das Eintreten der Fehlersituation **verschleiern** und **intransparent** machen, wenn der Auffangwert auch als Resultat einer regulären Berechnung auftreten kann.
- Oft fehlt ein naheliegender und plausibler Wert als Auffangwert; die Wahl eines Auffangwerts ist in diesen Fällen willkürlich und unintuitiv.
- Oft **fehlt** ein funktionsspezifischer Auffangwert **gänzlich**; **Auffangwertvariante 1** ist in diesen Fällen **nicht anwendbar**.

...dazu zwei **Beispiele**.

1) Auffangwert vorhanden, aber verschleiernd

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

Die Verwendung von `[]` als **funktionsspezifischem Auffangwert**

- ▶ liegt nahe und ist plausibel.

Allerdings:

- ▶ Das Auftreten der Fehlersituation wird **verschleiert** und bleibt für den Programmierer **intransparent**, da `[]` auch als reguläres Resultat einer Berechnung auftreten kann:

```
rest [42] ->> []      ([] als reguläres Resultat:
                       Keine Fehlersituation!)
rest []   ->> []      ([] als irreguläres Resultat:
                       Fehlersituation eingetreten!)
```

2) (Naheliegender) Auffangwert fehlt

```
kopf :: [a] -> a
kopf (u:_) = u
kopf []    = ???
```

Ohne Kenntnis der Instanz von `a` ist

- ▶ ein `a`-Wert überhaupt nicht angebar: Völliges Fehlen eines Auffangwerts.

Mit Kenntnis der Instanz von `a`, z.B.:

```
kopf :: [Int] -> Int, bietet sich
```

- ▶ kein `Int`-Wert als Auffangwert an: Fehlen eines naheliegenden, plausiblen Auffangwerts.

...in solchen Fällen Übergang zu [Auffangwertvariante 2](#) mit [aufrufspezifischen Auffangwerten](#).

Variante 2: Aufrufspezifische Auffangwerte

...Im Fehlerfall wird ein

► **aufrufspezifischer Auffangwert**

als Resultat geliefert.

Beispiel:

```
fac :: Integer -> Integer
fac n
  | n == 0    = 1
  | n > 0     = n * fac (n-1)
  | otherwise = n
```

Verhalten in Aufrufsituationen:

```
fac 5    ->> 120
fac 0    ->> 1
fac (-5) ->> -5      (Verschiedene Argumente,
fac (-7) ->> -7      verschiedene Fehlerwerte)
```

Analyse des Beispiels

Im **Beispiel** von `fac` gilt:

- ▶ Das Argument als **aufrufspezifischer Auffangwert** erlaubt wieder, negative Eingaben als fehlerhaft zu erkennen und zu melden, ohne den Programmablauf unwiderruflich abbrechen.
- ▶ Zusätzlich liefert das Argument als **Auffangwert aufrufspezifisch** die Rückmeldung, welcher Argumentwert zum Fehler geführt hat, was die Fehlersuche begünstigt.

Insgesamt:

- ▶ Die Fehlersituation ist für den Programmierer **transparent**.

Allerdings:

- ▶ Nicht immer taugt das Argument selbst als Auffangwert.

In solchen Fällen ist folgender allgemeinere Ansatz nötig.

Variante 2: Allgemeiner Ansatz

Grundlegende Idee: Erweitere die Signatur und übergib bei jedem Aufruf den gewünschten Anfangswert als Argument.

Beispiel: Ersetze `kopf` durch `kopf'` mit Signatur:

```
kopf' :: a -> [a] -> a
```

```
kopf' _ (u:_) = u
```

```
kopf' x []     = x
```

...und **aufrufspezifischem Anfangargument `x`**.

Variante 2: Allgemeiner Ansatz (schematisch)

...anhand einer hier einstellig angenommenen fehlerbehandlungs-freien Implementierung einer Funktion f :

► Ergänze f :

```
f :: a -> b
```

```
f u = ...
```

um die fehlerbehandelnde Hüllfunktion f' :

```
f' :: b -> a -> b
```

```
f' x u
```

```
  | fehlerFall = x
```

```
  | otherwise  = f u
```

wobei `fehlerFall` die Fehlersituation charakterisiert.

Bemerkung: Im sehr einfachen Beispiel von `kopf'` war die Abstützung auf `kopf` nach obigem Schema nicht nötig; der Effekt konnte implementierungstechnisch einfacher erreicht werden.

Bewertung von Auffangwertvariante 2

Positiv:

- + Panikmodus vermieden, Programmablauf nicht abgebrochen.
- + Generalität, stets anwendbar.
- + Flexibilität, aufrufspezifische Auffangwerte ermöglichen variierende Fehlerwerte und Fehlerbehandlung.

Bewertung von Auffangwertvariante 2 (figs.)

Negativ:

- Transparente Fehlerbehandlung ist nicht gewährleistet, wenn aufrufspezifische Auffangwerte auch reguläres Resultat einer Berechnung sein können, z.B.:
 - kopf' 'F' "Fehler" ->> 'F' ('F' als reg. Ergebnis)
 - kopf' 'F' "" ->> 'F' ('F' als irreg. Ergebnis)
- In diesen Fällen Gefahr ausbleibender Fehlerwahrnehmung mit (möglicherweise fatalen) Folgen durch
 - Vortäuschen eines regulären und korrekten Berechnungsablaufs und eines regulären und korrekten Ergebnisses!
(Typischer Fall eines "sich ein 'x' für ein 'u' vormachen zu lassen!")

Der schlimmste aller Fehler ist, sich keines solchen bewusst zu sein.

Thomas Carlyle (1795-1881)
schott. Essayist und Historiker

Kapitel 16.4

Fehlertypen, Fehlerwerte, Fehlerfunktionen

Votr. VI

Teil VI

Kap. 15

Kap. 16

16.1

16.2

16.3

16.4

16.5

Kap. 17

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweis

Aufgabe

Fehlertypen, Fehlerwerte, Fehlerfunktionen

Ziel: Systematisches

1. Erkennen
2. Anzeigen
3. Behandeln

von Fehlersituationen.

Werkzeug: Dezidierte

1. Fehlertypen
2. Fehlerwerte
3. Fehlerfunktionen

statt schlichter Auffangwerte.

Zentral: Anzeigbarkeit von Fehlern

...wird erreicht durch Übergang von Typ `a` zum (Fehler-) Datentyp `Maybe a`:

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

...umfasst die Werte des Typs `a` in der Form `Just a` mit dem Zusatzwert `Nothing` als explizitem Fehlerwert.

Beispiel:

```
div' :: Int -> Int -> Maybe Int
```

```
div' n m
```

```
| m /= 0 = Just (div n m)
```

```
| m == 0 = Nothing
```

```
div' 13 5 ->> Just 2           (Division geklappt)
```

```
div' 13 0 ->> Nothing        (Division gescheitert)
```

Systematisierung d. Beispielsidee (schematisch)

...anhand einer hier einstellig angenommenen fehlerbehandlungs-freien Implementierung einer Funktion f :

► Ergänze f :

```
f :: a -> b
```

```
f u = ...
```

um die fehlererkennende und -anzeigende Hüllfunktion f' :

```
f' :: a -> Maybe b
```

```
f' u
```

```
  | fehlerFall = Nothing
```

```
  | otherwise  = Just (f u)
```

wobei `fehlerFall` die Fehlersituation charakterisiert.

Angewendet auf das Beispiel

...ergänze die (vordef.) nichtfehlerbehandelnde Funktion `div`:

```
div :: Int -> Int -> Int
```

```
div n m = ... (Details Haskell-intern)
```

```
div 13 5 ->> 2
```

```
div 13 0 ->> Programmabbruch mit Laufzeitfehler
```

um die fehlererkennende und -meldende Hüllfunktion `div'`:

```
div' :: Int -> Int -> Maybe Int
```

```
div' n m
```

```
  | m == 0    = Nothing
```

```
  | otherwise = Just (div n m)
```

```
div' 13 5 ->> Just 2 (Division geklappt)
```

```
div' 13 0 ->> Nothing (Division gescheitert)
```


Analyse, Diskussion des Beispiels

...anders als `div`, deren Auswertung im Fehlerfall (d.h. Division durch null) gemäß des

► Panikmodus

vom `Laufzeitsystem` abgebrochen wird, kann `div'` einen Fehler ohne Auswertungsabbruch

1. erkennen: `m == 0`
2. anzeigen: `Nothing`

Noch offen:

- Was machen wir im Fehlerfall mit dem Resultat `Nothing`?

Generalisierung der bisherigen Idee

Ziel:

Erkennen, weiterreichen, fangen und behandeln von Fehlern mithilfe der Funktionen:

- `map_Maybe`: Erkennen und weiterreichen von Fehlern.
- `maybe`: Fangen und behandeln von Fehlern.

...die im Zusammenspiel das Erkennen, Weiterreichen, Fangen u. schließliche Behandeln von Fehlern zu organisieren erlauben.

Beachte: `map_Maybe` ist verschieden von der im Standard-Präludium definierten namensähnlichen Funktion `mapMaybe` mit Signatur:
`mapMaybe :: (a -> Maybe b) -> [a] -> [b]`.

Die Funktion `map_Maybe`

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

`map_Maybe f Nothing = Nothing` (Durchreichen
von Fehlern)

`map_Maybe f (Just u) = Just (f u)` (Rechnen mit `f`
im Normalfall)

Curryfizierte und uncurryfizierte Lesart von `map_Maybe`:

- ▶ **Curryfiziert:** `map_Maybe` bildet eine (nicht fehlerbehandelnde) Funktion vom Typ (a -> b) auf eine Funktion vom Typ (Maybe a -> Maybe b) ab (entspricht einem 'Typ-Lifting').
- ▶ **Uncurryfiziert:** `map_Maybe` bildet einen (Maybe a)-Wert auf einen (Maybe b)-Wert ab mithilfe einer (nicht fehlerbehandelnden) Funktion vom Typ (a -> b).

Die Funktion maybe

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

`maybe x f Nothing = x` (Aufrufspez. Fehlerwert)

`maybe x f (Just u) = f u` (Rechnen mit `f` im Normalfall)

Curryfizierte und uncurryfizierte Lesart von `map_Maybe`:

- ▶ **Curryfiziert:** Gegeben einen `b`-Wert bildet `maybe` eine Funktion vom Typ `(a -> b)` auf eine Funktion vom Typ `(Maybe a -> b)` ab (entspricht einem 'Typ-Lifting').
- ▶ **Uncurryfiziert:** `maybe` bildet einen `(Maybe a)`-Wert auf einen `b`-Wert ab mithilfe einer (nicht fehlerbehandelnden) Funktion vom Typ `(a -> b)` und eines aufrufspezifischen Fehlerarguments vom Typ `b` (entspricht **Auffangwertvariante 2**).

Im Zusammenspiel

...erlauben `map_Maybe` und `maybe` Fehlerwerte

- ▶ `weiterzureichen`, die Fähigkeit von `map_Maybe`:

```
map_Maybe f Nothing = Nothing
```

...der Fehlerwert `Nothing` wird von `map_Maybe` durchgereicht.

- ▶ zu `fangen` und (im Sinn von `Auffangwertvariante 2`) zu `behandeln`, die Fähigkeit von `maybe`:

```
maybe x f Nothing = x
```

...der aufrufspezifische Auffangwert `x` wird als Resultat geliefert (`Auffangwertvariante 2`).

Beispiel

...zum Zusammenspiel von `map_Maybe` und `maybe`:

- ▶ **Fehlerfall:** Der Fehler wird von `div'` erkannt und angezeigt, von `map_Maybe` weitergereicht und schließlich von `maybe` gefangen und behandelt.

```
maybe 9999 (+1) (map_Maybe (*3) (div' 9 0))
->> maybe 9999 (+1) (map_Maybe (*3) Nothing)
->> maybe 9999 (+1) Nothing
->> 9999
```

- ▶ **Fehlerfreier Fall:** Alles läuft 'normal' ab.

```
maybe 9999 (+15) (map_Maybe (*3) (div' 9 1))
->> maybe 9999 (+15) (map_Maybe (*3) (Just 9))
->> maybe 9999 (+15) (Just 27)
->> (+15) 27
->> 27 + 15
->> 42
```

Bewertung d. Fehlerbehandlung mittels Maybe

Positiv:

- + Fehler können erkannt, angezeigt, weitergereicht und schließlich gefangen und (im Sinn von Auffangwertvariante 2) behandelt werden.

Negativ:

- Geänderte Funktionalität: `Maybe b` statt `b`.

Pragmatische Zusatzvorteile:

- + Systementwicklung ist ohne explizite Fehlerbehandlung möglich (z.B. mit nichtfehlerbehandelnden Funktionen wie `div`).
- + Fehlerbehandlung kann nach Abschluss durch Ergänzung der fehlerbehandelnden Funktionsvarianten (wie z.B. der Funktion `div'`) zusammen mit den Funktionen `map_Maybe` und `maybe` umgesetzt werden.

Der schöpferische Irrtum

Irrtümer haben ihren Wert;
jedoch nur hie und da.
Nicht jeder, der nach Indien fährt,
entdeckt Amerika.

Erich Kästner (1899-1974)
dt. Schriftsteller

Vortr. VI

Teil VI

Kap. 15

Kap. 16

16.1

16.2

16.3

16.4

16.5

Kap. 17

Umgekehrt

Klassen-

zim-

mer V

Modusände

Test 1

Hinweis

Aufgabe

Kapitel 16.5

Leseempfehlungen

Votr. VI

Teil VI

Kap. 15

Kap. 16

16.1

16.2

16.3

16.4

16.5

Kap. 17




Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Basisleseempfehlungen für Kapitel 16

-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 19, Error Handling)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 2. Auflage, 1999. (Kapitel 14.4, Case study: program errors)
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011. (Kapitel 14.4, Modelling program errors)

Die schlimmsten Fehler werden gemacht
in der Absicht, einen begangenen Fehler
wieder gut zu machen.

Jean Paul (1763-1825)
dt. Schriftsteller

Kapitel 17

Programmierung im Großen: Module

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Kapitel 17.1

Überblick, Orientierung

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Module, Modularisierung von Programmen

...Zerlegung von Programmen in überschaubare, (oft) getrennt übersetzbare Programmeinheiten als wichtige programmiersprachliche Unterstützung der

- ▶ Programmierung im Großen.

Ich denke gern in großen Dimensionen.
Wenn man schon denkt,
kann man es ja auch gleich ordentlich tun.

Donald Trump (* 1946)
amerik. Unternehmer
45. Präsident der USA

...ein programmiersprachen- und programmierstilübergreifend anzutreffendes und umgesetztes Konzept.

Zwei wichtige Eigenschaften

...zur Charakterisierung guter Modularisierung:

1. Kohäsion (modullokal, intramodular)

- beschäftigt sich mit dem **inneren Zusammenhang** von Modulen, mit Art und Typ der in einem Modul zusammengefassten Funktionen.

2. Koppelung (modulübergreifend, intermodular)

- beschäftigt sich mit dem **äußeren Zusammenhang** von Modulen, dem Import-/Export- und Datenaustauschverhalten.

Kapitel 17.2

Ziele guter Modularisierung

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zimmer
V

Modusänderung
Test 1

Hinweis

Aufgabe

Ziele guter Modularisierung

...von einer **technischen Programmperspektive** aus:

Modullokal (intramodular): Module sollen

- ▶ einen klar umrissenen, unabhängig von anderen Modulen verständlichen Zweck besitzen.
- ▶ nur einer Abstraktion entsprechen.
- ▶ einfach zu testen sein.

Modulübergreifend (intermodular): Modular entworfene Programme sollen

- ▶ **Auswirkungen** von **Designentscheidungen** (z.B. Einfachheit vs. Effizienz einer Implementierung)
- ▶ **Abhängigkeiten** von anderen Programmen oder Hardware

...auf (möglichst) wenige Module beschränken.

Ziele guter Modularisierung

...von einer **semantischen, inhaltl. Programmperspektive** aus:

Modullokal (intramodular):

- ▶ **Funktionale Kohäsion:** Fasse Funktionen gleicher Funktionalität zusammen, z.B. Sortierverfahren, Ein-/Ausgabe,...
- ▶ **Datenkohäsion:** Fasse Funktionen zusammen, die auf den gleichen Datenstrukturen arbeiten, z.B. Funktionen auf trigonometrischen Daten,...

Modulübergreifend (intermodular):

- ▶ **Schwache funktionale Koppelung:** Strebe nach wenigen, wohlbegründeten funktionalen Beziehungen und Abhängigkeiten zwischen Modulen.
- ▶ **Feste Datenkoppelung:** Strebe nach Kommunikation modulverschiedener Funktionen durch Wertübergabe: Ergebnisse einer Funktion werden Argumente einer anderen.

Zu vermeiden

Modullokal (intramodular):

- ▶ **Logische Kohäsion:** Vermeide Funktionen vergleichbarer Funktionalität, aber unterschiedlicher Implementierung zusammenzufassen, z.B. verschiedene Benutzerschnittstellen eines Systems.
- ▶ **Zufällige Kohäsion:** Vermeide Funktionen ohne sachlichen Grund zusammenzufassen.

Modulübergreifend (intermodular):

- ▶ **Starke funktionale Koppelung:** Vermeide eine Vielzahl fkt. Beziehungen und Abhängigkeiten zwischen Modulen.
- ▶ **Lose Datenkoppelung:** Vermeide andere Mechanismen als Wertübergabe zur Kommunikation von modulverschiedenen Funktionen, z.B. über Dateien.

Gut zu wissen: In fkt. Sprachen ist Datenkoppelung durch Wertübergabe *per se* die Standardform.

Kennzeichen gelungener Modularisierung

Starke funktionale und Datenkohäsion

- ▶ enger inhaltlicher Zusammenhang der Definitionen eines Moduls.

Schwache funktionale und lose Datenkoppelung

- ▶ wenige Abhängigkeiten zwischen verschiedenen Modulen, insbesondere keine direkten oder indirekten zirkulären Abhängigkeiten.

Für eine vertiefende Diskussion siehe:



Manuel Chakravarty, Gabriele Keller. Einführung in die Programmierung mit Haskell, Pearson Studium, 2004, Kapitel 10.

Anforderungen an Modularisierungskonzepte

...zur Erreichung vorgenannter Ziele.

Unterstützung des Geheimnisprinzips durch Trennung von

- ▶ **Schnittstelle (Import/Export)**
 - Wie interagiert das Modul mit seiner Umgebung?
 - Welche Funktionalität stellt es zur Verfügung (**Export**)?
 - Welche Funktionalität benötigt es (**Import**)?
- ▶ **Implementierung (Daten/Funktionen)**
 - Wie sind die Datenstrukturen implementiert?
 - Wie ist die Funktionalität auf den Datenstrukturen realisiert?

Kapitel 17.3

Haskells Modulkonzept

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.3.1

17.3.2

17.3.3

17.3.4

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Schematischer Aufbau von Haskellmodulen

Moduldateien werden eingeleitet von der Zeile:

```
module M where
```

gefolgt von Deklarationen/Definitionen von:

1. Typen (algebraische Typen, Neue Typen, Typsynonyme)
2. Typklassen
3. Funktionen

Schematischer Modulaufbau: Illustration

```
module M where                -- Moduldefinition
data D_1 ... = ...           -- Algebraische Typen
...
data D_n ... = ...
newtype N_1 ... = ...       -- Neue Typen
...
newtype N_m ... = ...
type T_1 ... = ...          -- Typsynonyme
...
type T_p ... = ...
class C_1 ...                -- Typklassen
...
class C_q ...
f_1 :: ...                   -- Funktionen
f_1 ... = ...
...
f_r :: ...
f_r ... = ...
```

Vortr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.3.1

17.3.2

17.3.3

17.3.4

17.4

17.5

17.6

Umgekehr

Klassen-

zim-

mer V

Modusänd

Test 1

Hinweis

Aufgabe

Haskells Modulkonzept

...unterstützt den Import und Export von Datentypen, Typsynonymen, Typklassen und Funktionen.

Im einzelnen:

▶ Import

- Selektiv/nicht selektiv
- Qualifiziert
- Mit Umbenennung

▶ Export

- Selektiv/nicht selektiv
- Händischer Reexport
- **Nicht unterstützt:** Automatischer Reexport

Kapitel 17.3.1

Import

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.3.1

17.3.2

17.3.3

17.3.4

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Nicht selektiver Import (schematisch)

```
module M1 where
```

```
...
```

```
module M2 where
```

```
import M1
```

```
...
```

- ▶ Modul **M2** importiert aus Modul **M1** alle (global sichtbaren) Bezeichner und Definitionen, die danach in **M2** verwendet werden können.

Selektiver Import (schematisch)

```
module M1 where
...
module M2 where                                -- Variante 1
import M1 (D_1 (..), D_2, T_1, C_1 (..), C_2, f_5)
...
module M3 where                                -- Variante 2
import M1 hiding (D_1, T_2, f_1)
...
```

- ▶ M2 importiert aus M1 ausschließlich die explizit genannten Bezeichner und Definitionen; das sind: D_1 (einschließlich von M1 exportierter Konstruktoren), D_2 (ohne Konstruktoren), T_1, C_1 (..) (einschließlich von M1 exportierter Funktionen), C_2 (ohne Funktionen), f_5.
- ▶ M3 importiert aus M1 alle in M1 (sichtbaren) Bezeichner und Definitionen mit Ausnahme der explizit genannten.

Kapitel 17.3.2

Export

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.3.1

17.3.2

17.3.3

17.3.4

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Nicht selektiver Export (schematisch)

```
module M1 where
data D_1 ... = ...
...
newtype N_1 ... = ...
...
type T_1 = ...
...
class C_1 ...
...
f_1 :: ...
f_1 ... = ...
...
```

- ▶ Alle in **M1** eingeführten global sichtbaren Bezeichner und Definitionen sind exportbereit und können von anderen Modulen importiert werden.

Beachte: Die Zeile `module M1 where...` ist bedeutungsgleich zu `module M1 (module M1) where...`

Selektiver Export (schematisch)

```
module M1 (D_1 (...), D_2, D_3 (Dc_1,...,Dc_k), C_1 (...),  
          C_2, C_3 (cf_1,...,cf_l), T_1, f_2, f_5) where  
data D_1 ... = ...  
...  
newtype N_1 ... = ...  
...  
type T_1 = ...  
...  
class C_1 ...  
...  
f_1 :: ...  
f_1 ... = ...  
...
```

- ▶ Nur die explizit genannten Bezeichner, Definitionen aus **M1** sind exportbereit und können von anderen Modulen importiert werden. Dabei ist **D_1** einschließl. seiner Konstruktoren exportbereit, **D_2** ohne, **D_3** mit den explizit genannten. Analog für die Klassen **C_i**.
- ▶ **Beachte:** Selektiver Export unterstützt das **Geheimnisprinzip!**

Kapitel 17.3.3

Reexport

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.3.1

17.3.2

17.3.3

17.3.4

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Reexport (schematisch): Nicht automatisch!

```
module M1 where...
```

```
module M2 where
```

```
import M1
```

```
...
```

```
f_M2j
```

```
...
```

```
module M3 where
```

```
import M2
```

```
...
```

- ▶ M2 importiert nicht selektiv aus M1, d.h. alle in M1 (global sichtbaren) Bezeichner, Definitionen werden von M2 importiert und können in M2 benutzt werden.
- ▶ M3 importiert nicht selektiv aus M2, d.h. alle in M2 (global sichtbaren) Bezeichner, Definitionen werden von M3 importiert und können in M3 benutzt werden, nicht jedoch die von M2 aus M1 importierten Namen, d.h. **kein automatischer Reexport!**

Abhilfe: Händischer Reexport!

...in den zwei Varianten **nicht selektiv** und **selektiv**:

```
module M2 (module M1,f_M2_j) where           (nicht selektiv)
import M1
```

```
...
```

```
f_M2_j
```

```
...
```

(selektiv)

```
module M3 (D_1 (..), D_2, D_3 (Dc_1,Dc_2), C_1 (..), C_2,
           C_3 (cf_1,cf_2,cf_3), f_1,f_M3_k) where
```

```
import M1
```

```
...
```

```
f_M3_k
```

```
...
```

- ▶ **Nicht selektiver Reexport von M1 aus M2**: M2 reexportiert jeden aus M1 importierten Namen, sowie das M2-lokale f_M2_j aus M2.
- ▶ **Selektiver Reexport von M1 aus M3**: M3 reexportiert von den aus M1 importierten Namen ausschließlich D_1 (einschließl. Konstruktoren), D_2 (ohne Konstruktoren), D_3 (mit angegebenen Konstruktoren); analog f. d. Klassen C_1, C_2, C_3, f_1 und das M3-lokale f_M3_k.

Kapitel 17.3.4

Namenskonflikte, Umbenennungen, Konventionen

Vortr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.3.1

17.3.2

17.3.3

17.3.4

17.4

17.5

17.6

Umgekehr

Klassen-

zim-

mer V

Modusänd

Test 1

Hinweis

Aufgabe

Namenskonflikte, Umbenennungen

Namenskonflikte

- ▶ können durch **qualifizierten Import** aufgelöst werden:

```
import qualified M1
```

Verwendung: `M1.f` zur Bezeichnung der aus `M1` importierten Funktion `f`; `f` zur Bezeichnung der im importierenden Modul lokal definierten Funktion `f`.

Umbenennen importierter Module und Bezeichner

- ▶ durch Einführen lokaler Namen im importierenden Modul

- für **Modulnamen**:

```
import qualified M1 as MyLocalNameForM1
```

`...MyLocalNameForM1` wird im importierenden Modul anstelle von `M1` verwendet.

- für **ausgewählte Bezeichner**:

```
import M1 (f1,f2)
```

```
renaming (f1 to fac, f2 to fib)
```

Haskell-Programme

...sind **Modulsysteme**.

Soll ein Haskell-Programm **übersetzt** (statt **interpretiert**) werden, muss dessen Modulsystem ein **Hauptmodul** namens

- `Main`

mit einer Funktion namens

- `main :: IO τ` für τ konkreter Typ

enthalten, mit deren Auswertung die Ausführung des übersetzten Programms beginnt (wobei das Ergebnis vom Typ τ unbeachtet bleibt).

Beachte: Die `module`-Deklaration darf in einem Haskell-Skript fehlen; implizit wird in diesem Fall die `module`-Deklaration

```
module Main (main) where
```

ergänzt.

Konventionen, gute Praxis

Konventionen

- Pro Datei **ein** Modul.
- Modul- und Dateiname stimmen überein (abgesehen von der Endung **.hs** bzw. **.lhs** im Dateinamen).
- Alle Deklarationen beginnen in derselben Spalte wie das Schlüsselwort **module**.

Gute Praxis

- Module unterstützen **eine (!)** klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium).
- Module sind **'kurz'** (d.h. so kurz wie möglich, so lang wie nötig).

Kapitel 17.4

Modul-Anwendung: Abstrakte Datentypen

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Konkrete vs. abstrakte Datentypen

Konkrete Datentypen (KDT) (in Haskell: Algebr. Datentypen)

- ▶ werden durch die exakte Angabe und Darstellung ihrer Werte spezifiziert, aus denen sie bestehen.
- ▶ auf ihnen gegebene Funktionen/Operationen werden zum Definitionszeitpunkt nicht angegeben und bleiben offen.

Abstrakte Datentypen (ADT)

- ▶ werden durch ihr Verhalten spezifiziert, d.h. durch die auf ihren Werten definierten Funktionen/Operationen und deren Zusammenspiel.
- ▶ die tatsächliche Darstellung der Werte des Datentyps wird zum Definitionszeitpunkt nicht angegeben u. bleibt offen.
- ▶ Dem Anwender eines abstrakten Datentyps wird die Darstellung der Werte und die Implementierung der Funktionen darauf nie bekanntgegeben: Geheimnisprinzip!

Grundlegende Idee von ADT-Definitionen

...Festlegung u. Implementierung eines Datentyps in 3 Teilen:

- A) **Schnittstellenfestlegung**: Angabe der auf den Werten des Datentyps zur Verfügung stehenden Operationen in Form ihrer syntaktischen Signaturen (**öffentlich**).
- B) **Verhaltensfestlegung**: Festlegung der Bedeutung der Operationen durch Angabe ihres Zusammenspiels in Form von **Axiomen** (sog. **Gesetzen**), die von jeder (!) Implementierung dieser Operationen einzuhalten sind (**öffentlich**).
- C) **Implementierung**: Implementierung des ADT durch einen KDT, der A) und B) erfüllt (**nicht öffentlich**).

Wichtig: In A) und B) wird die Darstellung der Werte des abstrakten Datentyps ausdrücklich nicht festgelegt; sie bleibt verborgen und deshalb für die Implementierung in C) als Freiheitsgrad offen!

Herausforderung für ADT-Definitionen

...in

- ▶ **Teil B)**: Die **Gesetze** so zu wählen, dass das Verhalten der Operationen exakt und eindeutig festgelegt ist; also so, dass weder eine **Überspezifikation** (keine widerspruchsfreie Implementierung möglich) noch eine **Unterspezifikation** (mehrere in sich widerspruchsfreie, aber sich widersprechende Implementierungen möglich) vorliegt.
- ▶ **Teil C)**: Die Implementierung durch Funktionen auf einem KDT so vorzunehmen, dass die **Gesetze** aus **Teil B)** erfüllt sind.

Vortr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Vorteil von ADT-Definitionen

...die Trennung von **öffentlicher A), B) Schnittstellen- und Verhaltensfestlegung** und **nicht öffentlicher C) Implementierung** erlaubt die

- ▶ **Implementierung** zu verstecken (**Geheimnisprinzip!**)

und nach

- ▶ **Zweckmäßigkeit** und **Anforderungen** (z.B. Einfachheit, Performanz) **auszuwählen** und in der Einsatzphase bei Bedarf auch **auszutauschen**.

Beispiel: Der ADT Warteschlange (FIFO)

...in Pseudo-Code (kein Haskell):

A) Schnittstellenfestlegung durch Signaturangabe:

```
NEW:                               -> Queue
ENQUEUE: Queue × Item -> Queue
FRONT:  Queue           -> Item
DEQUEUE: Queue         -> Queue
IS_EMPTY: Queue        -> Boolean
```

B) Verhaltensfestlegung in Form von Axiomen/Gesetzen:

```
b1) IS_EMPTY(NEW)                = true
b2) IS_EMPTY(ENQUEUE(q,i))      = false
b3) FRONT(NEW)                   = error
b4) FRONT(ENQUEUE(q,i))         = if IS_EMPTY(q) then i
                                   else FRONT(q)
b5) DEQUEUE(NEW)                 = error
b6) DEQUEUE(ENQUEUE(q,i)) =
    if IS_EMPTY(q) then NEW
    else ENQUEUE(DEQUEUE(q),i)
```

Implementierung des ADT Warteschlange

...in Haskell.

Implementierungstechnischer Schlüssel:

- ▶ Haskell's Modulkonzept, speziell der **selektive Export**, bei dem Konstruktoren algebraischer Datentypen verborgen bleiben

wodurch das mit **ADT-Definitionen** verfolgte Ziel:

- ▶ **Kapselung** von Daten, Realisierung des **Geheimnisprinzips** auf Datenebene (engl. **information hiding**)

erreicht werden kann.

A)&B): Schnittstellen-, Verhaltensfestlegung

```
module Queue
  {- Kommunikation von Teil A: Schnittstellenspezifikation -}
  (Queue,      -- Name des Datentyps (Geheimnisprinzip, kein
               -- Konstruktorexport!)

  new,        -- new :: Queue a
  enqueue,    -- enqueue :: Queue a -> a -> Queue a
  front,      -- front :: Queue a -> a
  dequeue,    -- dequeue :: Queue a -> Queue a
  is_empty,   -- is_empty :: Queue a -> Bool

  {- Kommunikation von Teil B: Axiome/Gesetze
   b1) is_empty(new)           = True
   b2) is_empty(enqueue(q,i)) = False
   b3) front(new)              = error "Niemand wartet!"
   b4) front(enqueue(q,i))     = if is_empty(q) then i
                               else front(q)
   b5) dequeue(new)            = error "Niemand wartet!"
   b6) dequeue(enqueue(q,i))   = if is_empty(q) then new
                               else enqueue(dequeue(q),i) -}
) where...
```

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehr

Klassen-

zim-

mer V

Modusän

Test 1

Hinweis

Aufgabe

C) Implementierung 1 über KDT mit data

{- Implementierung von A), B) über algebraischem Datentyp -}

```
data Queue a = Qu [a]
```

```
new :: Queue a
```

```
new = Qu []
```

```
enqueue :: Queue a -> a -> Queue a
```

```
enqueue (Qu xs) x = Qu (xs ++ [x])
```

```
front :: Queue a -> a
```

```
front q@(Qu xs)           -- Hier praktisch: Das als-Muster
```

```
  | not (is_empty q) = head xs
```

```
  | otherwise       = error "Schlange leer; niemand wartet!"
```

```
dequeue :: Queue a -> Queue a
```

```
dequeue q@(Qu xs)       -- Hier praktisch: Das als-Muster
```

```
  | not (is_empty q) = Qu (tail xs)
```

```
  | otherwise       = error "Schlange leer; niemand wartet!"
```

```
is_empty :: Queue a -> Bool
```

```
is_empty (Qu []) = True
```

```
is_empty _      = False
```

Vortr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

C) Implementierung 2 über KDT m. `newtype`

{- Implementierung von A), B) über Neuem Typ -}

```
newtype Queue a = Qu [a]
```

```
new :: Queue a
```

```
new = Qu []
```

```
enqueue :: Queue a -> a -> Queue a
```

```
enqueue (Qu xs) x = Qu (xs ++ [x])
```

```
front :: Queue a -> a
```

```
front q@(Qu xs)          -- Hier praktisch: Das als-Muster
```

```
| not (is_empty q) = head xs
```

```
| otherwise        = error "Schlange leer; niemand wartet!"
```

```
dequeue :: Queue a -> Queue a
```

```
dequeue q@(Qu xs)      -- Hier praktisch: Das als-Muster
```

```
| not (is_empty q) = Qu (tail xs)
```

```
| otherwise        = error "Schlange leer; niemand wartet!"
```

```
is_empty :: Queue a -> Bool
```

```
is_empty (Qu []) = True
```

```
is_empty _      = False
```

Vortr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

C) Implementierung 3 über KDT mit `type`

```
{- Implementierung von A), B) über Typsynonym -}
```

```
type Queue a = [a]

new :: Queue a
new = []

enqueue :: Queue a -> a -> Queue a
enqueue q x = q ++ [x]

front :: Queue a -> a
front q
  | not (is_empty q) = head q
  | otherwise        = error "Schlange leer; niemand wartet!"

dequeue :: Queue a -> Queue a
dequeue q
  | not (is_empty q) = tail q
  | otherwise        = error "Schlange leer; niemand wartet!"

is_empty :: Queue a -> Bool
is_empty q = (q == [])
```

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Konzeptueller Vorteil abstrakter Datentypen

...das **Geheimnisprinzip**: Nur die **ADT-Schnittstelle** ist bekannt, die **KDT-Implementierung** bleibt verborgen.

Das gewährleistet folgende Vorteile der **ADT-Konzepts**:

1. **Schutz** der Datenstruktur vor unkontrolliertem oder nicht beabsichtigtem/zugelassenem Zugriff.

Beispiel: Ein eigendefinierter Leerheitstest wie:

```
emptyQ == Qu []
```

fürhte in **Queue** importierenden Modulen zu einem Laufzeitfehler, da die Implementierung und somit der Konstruktor **Qu** dort nicht sichtbar sind.

2. Einfache **Austauschbarkeit** der zugrundeliegenden Implementierung.
3. **Unterstützung** arbeitsteiliger Programmierung.

Zur ADT-Realisierbarkeit in Haskell

...das ADT-Konzept ist kein erstrangiges Sprachelement (engl. *first class citizens*) in Haskell:

- ▶ Haskell bietet kein dezidiertes Sprachkonstrukt zur Spezifikation von ADTs, das eine externe Offenlegung von Signaturen und Gesetzen bei intern bleibender Implementierung erlaubt.

Allerdings können ADTs in Haskell (behelfsmäßig) mithilfe des

- ▶ **Modulkonzepts** realisiert werden.

Das erlaubt, die **KDT-Implementierung** eines ADT

- + intern und damit im Sinn des Geheimnisprinzips versteckt zu halten.
- jedoch können die Funktionssignaturen und Gesetze dem ADT-Anwender nur umständlich und in unsicherer Weise in Form von Kommentaren kommuniziert werden.

Wegweisende Arbeiten

...zu abstrakten Datentypen:

- ▶ John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
- ▶ John V. Guttag, James Jay Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.
- ▶ John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.

Kapitel 17.5

Zusammenfassung

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe

Modularisierungsvorteile und -gewinne

- ▶ **Arbeitsphysiologisch:** Unterstützung arbeitsteiliger Programmierung.
- ▶ **Softwaretechnisch:** Unterstützung der Wiederbenutzung von Programmen und Programmteilen.
- ▶ **Implementierungstechnisch:** Unterstützung getrennter Übersetzung (engl. separate compilation).

Insgesamt:

- ▶ Höhere Effizienz der Softwareerstellung bei gleichzeitiger Qualitätssteigerung (Verlässlichkeit) und Kostenreduktion.

Kapitel 17.6

Leseempfehlungen

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

17.1

17.2

17.3

17.4

17.5

17.6




Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1





Hinweis

Aufgabe





Basisleseempfehlungen für Kapitel 17

-  Marco Block-Berlitz, Adrian Neumann. *Haskell Intensivkurs*. Springer-V., 2011. (Kapitel 8, Modularisierung und Schnittstellen)
-  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson Studium, 2004. (Kapitel 10, Modularisierung und Programmdekomposition)
-  Miran Lipovača. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, 2011. (Kapitel 6, Modules)

Weiterführ. Leseempfehlungen für Kap. 17 (1)

-  David L. Parnas. *On the Criteria to be used on Decomposing Systems into Modules*. Communications of the ACM 15(12):1053-1058, 1972.
-  David L. Parnas, Paul C. Clements, David M. Weiss. *The Modular Structure of Complex Systems*. IEEE Transactions on Software Engineering 11(3):259-266, 1985.
-  Peter Pepper. *Funktionale Programmierung in OPAL, ML, Haskell und Gofer*. Springer-V., 2. Auflage, 2003. (Kapitel 14, Datenstrukturen und Modularisierung)
-  Bryan O'Sullivan, John Goerzen, Don Stewart. *Real World Haskell*. O'Reilly, 2008. (Kapitel 5, Writing a Library: Working with JSON Data – The Anatomy of a Haskell Module, Generating a Haskell Program and Importing Modules)

Weiterführ. Leseempfehlungen für Kap. 17 (2)

-  John V. Guttag. *Abstract Data Types and the Development of Data Structures*. Communications of the ACM 20(6):396-404, 1977.
-  John V. Guttag, James Jay Horning. *The Algebra Specification of Abstract Data Types*. Acta Informatica 10(1):27-52, 1978.
-  John V. Guttag, Ellis Horowitz, David R. Musser. *Abstract Data Types and Software Validation*. Communications of the ACM 21(12):1048-1064, 1978.
-  Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley/Pearson, 3. Auflage, 2011.
(Kapitel 15.1, Modules in Haskell; Kapitel 15.2, Modular design; Kapitel 16, Abstract data types)

Umgekehrtes Klassenzimmer V

...zur Übung, Vertiefung

...nach Eigenstudium von Teil V 'Fundierung fkt. Prog.':

- Zwar weiß ich viel...

Als Bonusthema, so weit die Zeit erlaubt:

- Testvorbereitung: Zwei beispielhafte Tests

Zwar weiß ich viel...

doch möchte ich alles wissen.

Wagner, Assistent von Faust
Johann Wolfgang von Goethe (1749-1832)
dt. Dichter und Naturforscher

Vortrag VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

Umgekehrte

Klassen-

zim-

mer V

Zwar weiß
ich viel...

Bonusthema:

Testvor-
bereitung –

Zwei
beispiel-

hafte
Tests

Modusänderung

Test 1

Hinweis

Aufgabe

Zeit für Ihren Zweifel, Ihre Fragen!

Der Zweifel ist der Beginn der Wissenschaft.

Wer nichts anzweifelt, prüft nichts.

Wer nichts prüft, entdeckt nichts.

Wer nichts entdeckt, ist blind und bleibt blind.

Pierre Teilhard de Chardin (1881-1955)

franz. Jesuit, Theologe, Geologe und Paläontologe

Die großen Fortschritte in der Wissenschaft

beruhen oft, vielleicht stets, darauf, dass man

eine zuvor nicht gestellte Frage doch,

und zwar mit Erfolg, stellt.

Carl Friedrich von Weizsäcker (1912-2007)

dt. Physiker und Philosoph

...entdecken Sie den **Wagner** in sich!

Vortr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

Umgekehrte

Klassen-

zim-

mer V

Zwar weiß
ich viel...

Bonusthema:

Testvor-

bereitung –

Zwei

beispiel-

hafte

Tests

Modusänderung

Test 1

Hinweis

Aufgabe

Bonusthema

Testvorbereitung: Zwei beispielhafte Tests

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

Umgekehrt

Klassen-
zim-
mer V

Zwar weiß
ich viel...

Bonusthema:
Testvor-
bereitung –
Zwei
beispiel-
hafte
Tests

Modusänderung

Test 1

Hinweis

Aufgabe

Test 1 (i)

Aufgabe 1 Zwei Zeichenreihen haben *Abstand* n , $n \in \mathbb{N}_0$, gdw. die beiden Zeichenreihen sind von gleicher Länge und unterscheiden sich an genau n Positionen voneinander.

1. Gibt es Fälle für die Berechnung des Abstandes von Zeichenreihen, die von der obigen Beschreibung nicht erfasst sind und eine besondere Behandlung erfordern? Wenn ja, welche?
2. Wie können etwaige besondere Fälle sinnvoll behandelt werden? Nennen Sie eine Methode dafür und wie genau Sie damit etwaige besondere Fälle hier behandeln.
3. Schreiben Sie eine curryfizierte Haskell-Rechenvorschrift `abs` einschließlich ihrer syntaktischen Signatur, die angewendet auf zwei Zeichenreihen ihren Abstand liefert. Etwaige besondere Fälle sollen von `abs` so behandelt werden, wie in der vorigen Teilaufgabe beschrieben.
4. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.

Test 1 (ii)

Aufgabe 2

1. Wie sind Typklassen in Haskell aufgebaut?
2. Wozu dienen sie?
3. Mit welcher Art von Polymorphie sind Typklassen eng verbunden?
4. Was ermöglicht diese Art von Polymorphie wiederzuverwenden?
5. Welche synonymen Bezeichnungen gibt es für diese Polymorphieart?

Aufgabe 3 Wir betrachten Bäume, deren Blätter eine Benennung und deren Nichtblätter zwei Benennungen tragen und keinen oder beliebig viele Teilbäume besitzen. Ein Wald von Bäumen enthält beliebig viele oder auch gar keinen Baum.

1. Geben Sie möglichst typallgemeine Definitionen für Bäume und Wälder in Haskell an. Verwenden Sie algebraische Datentypen nur, wenn nötig.
2. Machen Sie den Baumtyp zu einer Instanz der Typklasse Eq, ohne dafür eine `deriving`-Klausel zu verwenden. Zwei Bäume sind gleich gdw. die Bäume stimmen in Struktur und Benennungen überein.

Test 1 (iii)

Aufgabe 4

```
f :: Integer -> Integer
```

```
f n = if n == 0 then 0 else f (n-1) + n * n
```

1. Was berechnet f ?
2. Von welchem Rekursionstyp ist f ?
3. Schreiben Sie die Funktion f bedeutungsgleich
 - 3.1 mithilfe bewachter Ausdrücke.
 - 3.2 argumentfrei mithilfe einer anonymen λ -Abstraktion.
 - 3.3 unter (Mit-) Verwendung einer Listenkomprehension.
4. Geben Sie die ersten 5 Schritte d. Auswertung des Aufrufs $f (2+3)$ entsprechend der Standard-Auswertungsordnung von Haskell an:
 $f (2+3) \rightarrow \dots$

Aufgabe 5 Was bedeutet die Signatur der Funktion h :

```
h :: (a -> b) -> [a] -> [b]
```

1. in curryfizzierter Lesart?
2. in nicht-curryfizzierter Lesart?

Test 1 (iv)

Aufgabe 6 Eine Menge von Zeichenreihen hat *Abstand* n , $n \in \mathbb{N}_0$, gdw. die Menge ist nicht leer, alle Zeichenreihen der Menge sind von gleicher Länge, die Menge enthält zwei voneinander verschiedene Zeichenreihen mit Abstand n , die Menge enthält keine zwei verschiedenen Zeichenreihen mit Abstand m und $m < n$.

1. Gibt es Fälle für die Berechnung des Abstandes einer Menge von Zeichenreihen, die von der obigen Beschreibung nicht erfasst sind und eine besondere Behandlung erfordern? Wenn ja, welche?
2. Wie können etwaige besondere Fälle sinnvoll behandelt werden? Nennen Sie eine andere Methode als in Aufgabe 1 dafür und wie genau Sie damit etwaige besondere Fälle hier behandeln.
3. Schreiben Sie eine Haskell-Rechenvorschrift `mabs` einschließlich ihrer syntaktischen Signatur, die angewendet auf eine Menge von Zeichenreihen ihren Abstand liefert. Die Menge ist dabei in Form einer Liste von Zeichenreihen gegeben. Etwaige besondere Fälle sollen von `mabs` so behandelt werden, wie in der vorigen Teilaufgabe beschrieben.
4. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.

Test 2 (i)

Aufgabe 1 Zwei Zeichenreihen z und z' heißen *anfangsgleich vom Grad n* , $n \geq 0$, wenn die ersten n Zeichen von z u. z' positionsweise ident sind.

1. Schreiben Sie eine Haskell-Rechenvorschrift ag einschließlich ihrer syntaktischen Signatur, die angewendet auf zwei Zeichenreihen den Grad ihrer Anfangsgleichheit berechnet. Führen Sie für alle vorkommenden Typen treffende Typsynonyme ein.
2. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.

Aufgabe 2 Gegeben ist die Funktion f :

$$f\ x\ y = (x + y) * (x - y)$$

1. Welches ist der allgemeinstmögliche Typ von f ?
2. Ist dieser Typ monomorph oder polymorph?
3. Im Fall von Polymorphie:
 - 3.1 Wie heißt diese Art von Polymorphie möglichst genau?
 - 3.2 Gibt es Synonyme für diesen Polymorphiebegriff? Wenn ja, welches oder welche?

Test 2 (ii)

Aufgabe 3

- Schreiben Sie eine Funktion `sum` für die Berechnung der Summe der ersten n natürlichen Zahlen mit eins als kleinster natürlicher Zahl. Dabei soll gelten: `sum`
 - hat den Typ `Int -> Int`.
 - stützt sich auf eine repetitiv rekursive Funktion `sum'` ab.
 - sieht eine Panikmodusfehlerbehandlung vor.
- Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.
- Welchen Wert liefert der Aufruf `sum (2+3) + 5`?

Aufgabe 4 Was sind die

- Vorteile
- Nachteile

einer Panikmodusfehlerbehandlung?

Test 2 (iii)

Aufgabe 5 Gegeben ist die Funktion h :

$h [] \quad ys = ys$

$h (x:xs) \quad ys = h \ xs \ (x:ys)$

1. Welches ist der allgemeinstmögliche Typ von h ?
2. Ist h eine curryfizierte oder uncurryfizierte Funktion?
3. Von welchem Rekursionstyp ist h und woran erkennt man diesen Rekursionstyp?
4. Was berechnet h ? Was ist seine Bedeutung?
5. Was ist die Bedeutung von h mit der leeren Liste als zweitem Argument?
6. Von welcher Berechnungskomplexität ist h in der Zahl rekursiver Aufrufe?

Test 2 (iv)

Aufgabe 6 Werten Sie den Aufruf:

$f\ 16\ ((5-3)*7)$

der Funktion f aus Aufgabe 5 mit $f\ x\ y = (x + y) * (x - y)$ Schritt für Schritt

1. linksapplikativ
2. linksnormal

aus. Ein Schritt ist dabei *ein* Expansionsschritt oder *eine* Operatoranwendung.

Test 2 (v)

Aufgabe 7 Das junge Online-Warenhaus ALADIN verwendet für seinen Warenkatalog folgende Haskell-Typen:

```
type Nat1           = Int
type Artikelnummer = Int
type PreisInEURcent = Nat1
type Warenkatalog  = (Artikelnummer -> PreisInEURcent)
type NeuerPreis     = PreisInEURcent
```

1. Schreiben Sie eine Haskell-Rechenvorschrift:

```
preisaenderung :: Warenkatalog -> Artikelnummer
                -> NeuerPreis -> Warenkatalog
```

Angewendet auf einen Warenkatalog, eine Artikelnummer und einen neuen Preis wird der Preis des entsprechenden Artikels auf den neuen Preis gesetzt. Die Preise aller anderen Artikel bleiben unverändert. Eine Ausnahmebehandlung für 'unplausible' neue Preise ist nicht gefordert.

2. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.
3. Fkt. wie `preisaenderung` gehören zu einer wichtigen Teilmenge von Funktionen. Welcher? Woran erkennt man ihre Elemente?

Test 2 (vi)

Aufgabe 8 ALADIN bereitet sich auf eine Rabattschlacht vor: Die Preise bestimmter Artikel aus dem Normalkatalog sollen für den Aktionskatalog um einen bestimmten gleichen Prozentsatz gesenkt werden.

```
type Nat0           = Int
type Aktionsartikel = [Artikelnummer]
type Aktionsrabatt  = Nat0 -- in Prozent, Werte 0 bis 100.
type Normalpreis    = PreisInEURcent
type Aktionspreis   = PreisInEURcent
type Normalkatalog  = Warenkatalog
type Aktionskatalog = Warenkatalog
```

1. Schreiben Sie eine Haskell-Rechenvorschrift `aktion`, die den Aktionskatalog berechnet, in dem die Aktionsartikel den gegenüber dem Normalpreis aus dem Normalkatalog reduzierten Aktionspreis haben, Nichtaktionsartikel ihren Normalpreis aus dem Normalkatalog.

```
aktion :: Normalkatalog -> Aktionsartikel
        -> Aktionsrabatt -> Aktionskatalog
```

(Keine Ausnahmebehandlung für 'unplausible' Normalpreise/Aktionsrabatte; kein spezielles Auf-/Abrunden bei Rabattberechnungen)

2. Erklären Sie knapp, aber gut nachvollziehbar, wie ihre Rechenvorschrift vorgeht.

Modusänderung für Test 1 (2. Ankündigung)

Information über Modusänderung

- ▶ Der für **Donnerstag, 14.01.2021, 16-18 Uhr**, angekündigte **Test 1**, findet statt in **schriftlicher Form als online-Test via Zoom** (nicht in schriftlicher Form in Präsenz).
- ▶ Testdatum und -uhrzeit bleiben nach Möglichkeit und nach heutigem Stand erhalten; nötige, auch kurzfristig mögliche Änderungen werden frühestmöglich bekanntgegeben.

Informationen zu Testumgebung und -ablauf

Endgültige Testumgebung und -ablauf stehen noch nicht fest. Informationen hierzu werden frühestmöglich bekanntgegeben. Beispielhaft finden Sie zur Orientierung auf der Webseite der Lehrveranstaltung Informationen zu Testumgebungen und -abläufen

- ▶ der *Fakultät für Maschinenwesen und Betriebswissenschaften* vom 16.11.2020.
- ▶ des *Instituts für Geotechnik, Forschungsbereich für Grundbau, Boden- und Felsmechanik* vom 19.11.2020.

Hinweis

...für das Verständnis von **Vorlesungsteil VI** ist eine über den unmittelbaren Inhalt von **Vortrag VI** hinausgehende weitergehende und vertiefende Beschäftigung mit dem Stoff nötig; siehe:

- ▶ **vollständige Lehrveranstaltungsunterlagen**

...verfügbar auf der Webseite der Lehrveranstaltung:

http://www.complang.tuwien.ac.at/knoop/fp185A05_ws2021.html

Aufgabe bis Mittwoch, 16.12.2020

...selbstständiges Durcharbeiten von **Teil VI 'Weiterführende Konzepte'**, Kap. 15, 16 und 17 (ggf. auch schon von Kap. 18) und von **Leit- und Kontrollfragenteil VI** zur Selbsteinschätzung und als Grundlage für die **umgekehrte Klassenzimmersitzung** am **16.12.2020**:

Vortrag, umgek. Klassenz.	Thema Vortrag	Thema umgek. Klassenz.
Di, 06.10.2020, 08:15-09:45	Teil I	n.a. / Vorbesprechung
Di, 13.10.2020, 08:15-09:45	Teil II	Teil I
Di, 27.10.2020, 08:15-09:45	Teil III	Teil II
Mi, 04.11.2020, 08:15-09:45	Teil IV	Teil III
Mi, 18.11.2020, 08:15-09:45	Teil V	Teil IV
Mi, 02.12.2020, 08:15-09:45	Teil VI	Teil V
Mi, 16.12.2020, 08:15-09:45	Teil VII	Teil VI

Votr. VI

Teil VI

Kap. 15

Kap. 16

Kap. 17

Umgekehrte
Klassen-
zim-
mer V

Modusänderung
Test 1

Hinweis

Aufgabe