

HASKELL KURS

RALF HINZE

Institut für Informatik III
Universität Bonn
Römerstraße 164
53117 Bonn
Germany

Email: `ralf@informatik.uni-bonn.de`

Homepage: `http://www.informatik.uni-bonn.de/~ralf/`

April 2002

Haskell 98 Kurs

- ✘ Ein paar Worte vorneweg
- ✘ Haskell in einer Nußschale
- ✘ Ausdrücke und Definitionen
- ✘ Typen und Typklassen
- ✘ Ein- und Ausgabe
- ✘ Module und Bibliotheken
- ✘ Programmieretechniken

Zur Sprache „Haskell“

Zum Namen:

Haskell ist nach dem amerikanischen Logiker Haskell Brooks Curry benannt (weil Haskell [die Sprache] curryfizierte Funktionen liebt, wie Haskell [der Logiker] ;-)).

Zur Geschichte:

- 1987** Gründung des Haskell-Komitees (auf der FPCA), dem u.a. Paul Hudak, Simon Peyton Jones und Philip Wadler angehören.
- 1990** Haskell 1.0 Sprachdefinition
- 1992** Haskell 1.2 Sprachdefinition (erschiene in den Sigplan Notices)
- 1996** Haskell 1.3 Sprachdefinition (Mai)
- 1997** Haskell 1.4 Sprachdefinition (April)
- 1999** Haskell 98 Sprachdefinition (Februar)

Beeinflußt von: Lisp, APL, Standard ML, Miranda, u.v.a.

Zum Haskell Kurs

Was der Kurs leistet:

- ✘ hmm (er vermittelt hoffentlich einen Eindruck von Haskell und funktionaler Programmierung).

Was der Kurs *nicht* leistet: er ist

- ✘ weder als Einführung in die funktionale Programmierung geeignet,
- ✘ noch als Sprachdefinition von Haskell.

Für eine Einführung sei auf die Literatur verwiesen (siehe nächste Folie), für eine Sprachdefinition auf den Haskell 98 Report (§ 3.4 verweist im folgenden auf Kapitel 3, Abschnitt 4 des Reports).

C-Programmierer. So werden Hinweise für C-Programmierer (bzw. Kenner und Könner imperativer Sprachen) gekennzeichnet.

Literatur

Es gibt drei sehr gute Bücher über Haskell:

- ✘ Richard Bird. *Introduction to Functional Programming using Haskell*, 2. überarbeitete Ausgabe, Prentice Hall Europe, 1998.
- ✘ Simon Thompson. *Haskell: The Craft of Functional Programming*, 2. überarbeitete Ausgabe, Addison-Wesley, 1999.
- ✘ Paul Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.

Weiteres Material (Tutorials, Papers etc.) sowie Haskell Compiler findet man unter:

`www.haskell.org`

Haskell 98 Kurs

- ✓ Ein paar Worte vorneweg
- ✗ Haskell in einer Nußschale
- ✗ Ausdrücke und Definitionen
- ✗ Typen und Typklassen
- ✗ Ein- und Ausgabe
- ✗ Module und Bibliotheken
- ✗ Programmieretechniken

Ein Rat vorneweg

C-Programmierer. Vergeßt! Vergeßt am besten alles, was Ihr über Programmierung und Programmiersprachen wißt.

Einordnung von Haskell

Haskell ist eine *rein funktionale* Sprache.

„Funktional“ (auch: applikativ, wertorientiert) im Gegensatz zu

- ✘ imperativ,
- ✘ objektorientiert oder
- ✘ logisch.

„Rein“ funktionale Sprache im Gegensatz zu hybriden Sprachen wie

- ✘ LISP, Scheme oder
- ✘ Standard ML.

Struktur eines Haskell Programms . . .

Ein *Haskell Programm* besteht aus einem oder mehreren *Modulen*.

Ein *Modul* besteht aus *Definitionen* und *Deklarationen*.

```
date = (18, "April", 1996)
```

Definitionen werden im allgemeinen in Form von Gleichungen angegeben: der Bezeichner auf der linken Seite wird durch den *Ausdruck* auf der rechten Seite definiert.

Jeder Ausdruck besitzt einen *Typ*. Der Typ von *date* wird wie folgt deklariert:

```
date :: (Int, String, Int)
```

. . . Struktur eines Haskell Programms

Funktionen werden ebenfalls durch Gleichungen definiert. Der oder die *formalen Parameter* werden nach dem Namen der Funktion aufgeführt.

```
succ      :: Int → Int
succ n    = n + 1
twice f a = f (f a)
```

Aufgabe 1 Rate den Typ von *twice*!

Beachte: Haskell kommt mit wenig Syntax aus; $f\ a$ bezeichnet die Anwendung der Funktion f auf das Argument a .

Aufgabe 2 Was ist der Unterschied zwischen $f\ (f\ a)$ und $f\ f\ a$?

Berechnungsmodell

Das Berechnungsmodell von Haskell ist denkbar einfach: ein gegebener Ausdruck wird zu einem Wert (auch: Normalform) *reduziert*.

$$\begin{aligned} \text{succ} (\text{succ } 5) &\Rightarrow \text{succ } 5 + 1 && (\text{succ}.1) \\ &\Rightarrow (5 + 1) + 1 && (\text{succ}.1) \\ &\Rightarrow 7 \end{aligned}$$

Wir sagen: $\text{succ} (\text{succ } 5)$ reduziert zu 7.

Aufgabe 3 Reduziere $\text{twice succ } 0$ und $\text{twice twice succ } 0$! **Beachte:** der Ausdruck $f a b$ kürzt $(f a) b$ ab, da die Funktionsanwendung von links assoziiert.

Beispiel: Binärbäume

Eine (Programmier-) Sprache lernt man am besten durch Beispiele. Unser erstes: Binärbäume.

Ein neuer Typ wird mittels einer *Datentypdefinition* eingeführt.

```
data IntTree = Empty | Node IntTree Int IntTree
```

Der Bezeichner *IntTree* ist der Typname; *Empty* und *Node* sind *Konstruktoren*: mit ihrer Hilfe werden Element des Datentyps konstruiert.

```
atree :: IntTree  
atree = Node (Node Empty 2 Empty) 5 Empty
```

Beachte: *nur* Bezeichner von Typen und Konstruktoren werden groß geschrieben.

. . . Beispiel: Binärbäume . . .

Konstruktoren werden wie „normale Funktionen“ verwendet. Zusätzlich — und das unterscheidet sie von Funktionen — dienen sie zur Programmierung von Fallunterscheidungen.

```
insert :: Int → IntTree → IntTree  
insert a Empty = Node Empty a Empty  
insert a (Node l b r) = if  $a \leq b$  then Node (insert a l) b r  
                        else Node l b (insert a r)
```

Empty und *Node l b r* sind Beispiele für *Muster*. Die Variablen *l*, *b* und *r* benennen die entsprechenden Komponenten eines Baumknotens.

Aufgabe 4 Reduziere *insert 7 atree!*

. . . Beispiel: Binärbäume . . .

Für Testzwecke ist es nützlich, einen „größeren“ Binärbaum zur Hand zu haben.

$$\begin{aligned} \mathit{big} &:: \mathit{Int} \rightarrow \mathit{IntTree} \\ \mathit{big} \ n &= \mathit{Node} (\mathit{big} (2 * n)) \ n (\mathit{big} (2 * n + 1)) \end{aligned}$$

Etwas stutzen kann nicht schaden: *prune* verhindert, daß *big 1* in den Himmel wächst.

$$\begin{aligned} \mathit{prune} &:: \mathit{Int} \rightarrow \mathit{IntTree} \rightarrow \mathit{IntTree} \\ \mathit{prune} \ 0 \ t &= \mathit{Empty} \\ \mathit{prune} \ (n + 1) \ \mathit{Empty} &= \mathit{Empty} \\ \mathit{prune} \ (n + 1) \ (\mathit{Node} \ l \ a \ r) &= \mathit{Node} (\mathit{prune} \ n \ l) \ a (\mathit{prune} \ n \ r) \end{aligned}$$

Mit *prune n (big 1)* wird ein vollständig ausgeglichener Binärbaum der Tiefe *n* konstruiert.

Aufgabe 5 Reduziere *prune 2 (big 1)*! Worauf muß man achten?

. . . Beispiel: Binärbäume . . .

Löschen ist schwieriger als Einfügen.

$$\begin{aligned} \textit{delete} & \quad :: \quad \textit{Int} \rightarrow \textit{IntTree} \rightarrow \textit{IntTree} \\ \textit{delete} \ a \ \textit{Empty} & \quad = \quad \textit{Empty} \\ \textit{delete} \ a \ (\textit{Node} \ l \ b \ r) & \\ \quad | \ a < b & \quad = \quad \textit{Node} \ (\textit{delete} \ a \ l) \ b \ r \\ \quad | \ a == b & \quad = \quad \textit{join} \ l \ r \\ \quad | \ \textit{otherwise} & \quad = \quad \textit{Node} \ l \ b \ (\textit{delete} \ a \ r) \end{aligned}$$

Der senkrechte Strich | kennzeichnet einen *Wächter*: ein boolescher Ausdruck, der über die Anwendung einer Gleichung wacht.

Aufgabe 6 Formuliere *delete* mit Hilfe von **if**!

. . . Beispiel: Binärbäume . . .

Die Funktion *join* konkateniert zwei Suchbäume: der am weitesten rechts stehende Knoten des linken Teilbaums wird zur Wurzel.

<i>join</i>	::	$IntTree \rightarrow IntTree \rightarrow IntTree$
<i>join Empty r</i>	=	<i>r</i>
<i>join (Node ll a lr) r</i>	=	let (<i>b, t</i>) = <i>split ll a lr</i> in <i>Node t b r</i>
<i>split</i>	::	$IntTree \rightarrow Int \rightarrow IntTree \rightarrow (Int, IntTree)$
<i>split l a Empty</i>	=	(<i>a, l</i>)
<i>split l a (Node rl b rr)</i>	=	let (<i>c, t</i>) = <i>split rl b rr</i> in (<i>c, Node l a t</i>)

Beispiel: polymorphe Binärbäume

Wir haben uns bisher auf den Typ *Int* für Knotenmarkierungen festgelegt. Die folgende Datentypdefinition ist bezüglich des „Knotentyps“ parametrisiert. Der Bezeichner *lab* ist ein *Typparameter*.

```
data Tree lab = Empty | Node (Tree lab) lab (Tree lab)
```

Der Typ *IntTree* läßt sich als Abkürzung definieren.

```
type IntTree = Tree Int
```

. . . Beispiel: polymorphe Binärbäume . . .

Bezüglich dieser Typdefinition lauten die Typen der obigen Funktionen.

<i>big</i>	::	$Int \rightarrow Tree\ Int$
<i>insert</i>	::	??
<i>prune</i>	::	$Int \rightarrow Tree\ lab \rightarrow Tree\ lab$
<i>delete</i>	::	??
<i>join</i>	::	$Tree\ lab \rightarrow Tree\ lab \rightarrow Tree\ lab$
<i>split</i>	::	$Tree\ lab \rightarrow lab \rightarrow Tree\ lab \rightarrow (lab, Tree\ lab)$

Die Funktionen *prune*, *join* und *split* sind *polymorph*; sie arbeiten auf Binärbäumen über beliebigen Grundtypen.

Lies: für alle Typen *lab* ist $Int \rightarrow Tree\ lab \rightarrow Tree\ lab$ ein gültiger Typ von *prune*.

Aufgabe 7 Rate den Typ von *insert* und *delete*!

. . . Beispiel: polymorphe Binärbäume . . .

Die Funktionen *insert* und *delete* verwenden die Vergleichsfunktionen \leq , $<$ und $=$. Diese sind *überladen*; sie arbeiten je nach Typ unterschiedlich und sind nicht für alle Typen definiert: es ist z.B. nicht möglich, Funktionen zu vergleichen.

$$\begin{array}{l} \textit{insert} \quad :: \quad (\textit{Ord } \textit{lab}) \Rightarrow \textit{lab} \rightarrow \textit{Tree } \textit{lab} \rightarrow \textit{Tree } \textit{lab} \\ \textit{delete} \quad :: \quad (\textit{Ord } \textit{lab}) \Rightarrow \textit{lab} \rightarrow \textit{Tree } \textit{lab} \rightarrow \textit{Tree } \textit{lab} \end{array}$$

Ord ist ein Beispiel für eine *Typklasse*. *Lies*: für alle Typen *lab*, auf denen eine Ordnung definiert ist, ist $\textit{lab} \rightarrow \textit{Tree } \textit{lab} \rightarrow \textit{Tree } \textit{lab}$ ein gültiger Typ von *insert*.

. . . Beispiel: polymorphe Binärbäume

Aufgabe 8 „Mengen“ können durch binäre Suchbäume *ohne* Duplikate repräsentiert werden. Modifiziere *insert* und *delete* entsprechend!

Aufgabe 9 Implementiere den Datentyp „Wörterbuch“ unter Verwendung binärer Suchbäume.

```
type Dict attr val = Tree (attr, val)
empty                :: Dict a b
add                  :: (Ord a) => (a, b) -> Dict a b -> Dict a b
delete               :: (Ord a) => a -> Dict a b -> Dict a b
lookupWithDefault   :: (Ord a) => Dict a b -> b -> a -> b
```

Funktionale Programmierer sprechen auch von *endlichen Abbildungen*.

Exkurs: polymorphe Listen . . .

Zwei von drei Funktionen, die in Haskell geschrieben werden, arbeiten wahrscheinlich mit Listen ;-). Für die folgenden Beispiele kommen auch wir nicht ohne aus. Listen *könnten* wie folgt definiert werden.

```
data List a = Nil | Cons a (List a)
```

Haskell verwendet die folgende Notation: $[a]$ statt *List a*, $[]$ statt *Nil* und $:$ (Infixoperator) statt *Cons*.

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

Die Liste der ersten 5 Primzahlen ist $2:(3:(5:(7:(11:[])))$ oder kurz $2:3:5:7:11:[]$.

Aufgabe 10 Welche der Ausdrücke $1 : []$, $1 : 2 : []$, $(1 : []) : []$, $(1 : []) : (2 : [])$, $((1 : []) : []) : []$ sind zulässig?

. . . Exkurs: polymorphe Listen

Es gibt Myriaden vordefinierter Listenfunktionen. Z.B.:

$$\begin{aligned} (++) & \quad :: \quad [a] \rightarrow [a] \rightarrow [a] \\ [] ++ y & \quad = \quad y \\ (a : x) ++ y & \quad = \quad a : (x ++ y) \end{aligned}$$

Beachte: ‘++’ durchläuft die gesamte erste Liste.

Aufgabe 11 Definiere Funktionen, die einen Binärbaum in eine Liste und umgekehrt überführen. **Beachte:** *build* sollte möglichst einen ausgeglichenen Baum erzeugen.

$$\begin{aligned} inorder & \quad :: \quad Tree\ a \rightarrow [a] \\ build & \quad :: \quad [a] \rightarrow Tree\ a \end{aligned}$$

„Literate Programming“

Die Idee des „Literate Programming“ geht wie so vieles in der Informatik auf Donald E. Knuth zurück. Der übliche Kommentierstil wird auf den Kopf gestellt: alles ist Kommentar, Programmzeilen werden speziell gekennzeichnet.

Die Funktion ‘insert’ fuegt ein Element zu einen binaeren Suchbaum hinzu.

```
> insert                :: Int -> IntTree -> IntTree
> insert a Empty       = Node Empty a Empty
> insert a (Node l b r) = if a <= b then Node (insert a l) b r
>                       else Node l b (insert a r)
```

Auf diese Weise läßt sich z.B. ein Haskellprogramm in \LaTeX dokumentieren (mit etwas Programmunterstützung).

Der Haskell-Interpreter Hugs

Hugs ist ein Haskell-Interpreter und somit gut geeignet, kleinere Programme oder Programmfragmente zu erstellen und zu testen. Typische Vorgehensweise:

1. Das Programm (z.B. `BinTree.lhs`) wird mit einem Texteditor erstellt (die Endung `lhs` zeigt an, daß es sich um ein Literate Haskell Programm handelt).
2. Hugs wird gestartet.

```
basilisk> hugs BinTree.lhs
```

3. Jetzt können Ausdrücke eingegeben werden.

```
Prelude> insert 7 atree  
Node (Node Empty 2 Empty) 5 (Node Empty 7 Empty)
```

Nützliche Kommandos:

- `:browse` Alle definierten Bezeichner anzeigen.
- `:reload` Das Programm neu laden.
- `:quit` Hugs verlassen.
- `:?` Kommandos anzeigen.
- `:type <exp>` Den Typ von <exp> anzeigen.

Ein- und Ausgabe . . .

Beispiel: Ein- und Ausgabe von Zeichenketten. Die Ausgabe eines Zeichens ist vordefiniert.

$$putChar \quad :: \quad Char \rightarrow IO \ ()$$

Elemente vom Typ *IO val* beschreiben EA-Aktionen, die ein Element vom Typ *val* zurückgeben (*()* ist der „Dummytyp“). **Beachte:** die Aktionen werden nicht ausgeführt. Mit *putStrLn* wird ein String ausgegeben (vordefiniert).

$$\begin{aligned} putStrLn & \quad :: \quad String \rightarrow IO \ () \\ putStrLn [] & \quad = \quad putStrLn \ ' \backslash n ' \\ putStrLn (c : s) & \quad = \quad putStrLn \ c \gg putStrLn \ s \end{aligned}$$

Der Operator \gg korrespondiert zu dem Semikolon in Pascal.

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

. . . Ein- und Ausgabe . . .

Jedes Haskell Programm muß den Bezeichner $main :: IO ()$ definieren.

```
main = putStrLn "hello world"
```

Beim Programmstart wird die an $main$ gebundene Aktion (respektive die Beschreibung der Aktion) ausgeführt.

Warum ist es wichtig, die *Beschreibung* einer Aktion von deren *Ausführung* zu trennen? Anderenfalls wären die Ausdrücke

```
let a = putStrLn "ha" in a >> a  
putStrLn "ha" >> putStrLn "ha"
```

nicht äquivalent. Das Prinzip der Ersetzung „von Gleichem durch Gleiches“ wäre verletzt.

. . . Ein- und Ausgabe . . .

Das Pendant von *putChar* ist *getChar* (vordefiniert).

```
getChar  ::  IO Char
```

Mit *getLine* wird entsprechend eine Zeile eingelesen (ebenfalls vordefiniert).

```
getLine  ::  IO String  
getLine  =  getChar >>=  $\lambda c \rightarrow$   
           if  $c == '\n'$  then  
             return ""  
           else  
             getLine >>=  $\lambda s \rightarrow$   
             return ( $c : s$ )
```

Der Ausdruck $\lambda x \rightarrow e$ bezeichnet eine Funktion mit dem formalen Parameter x und dem Rumpf e .

. . . Ein- und Ausgabe

Der Operator $\gg=$ erlaubt, auf das Ergebnis einer Aktion zuzugreifen; *return a* bezeichnet die „leere“ EA-Aktion, die *a* zum Ergebnis hat.

```
( $\gg=$ )  :: IO a  $\rightarrow$  (a  $\rightarrow$  IO b)  $\rightarrow$  IO b
return  :: a  $\rightarrow$  IO a
```

C-Programmierer. $act1 \gg= \lambda x \rightarrow act2$ läßt sich als $x = act1; act2$ lesen. *Vorsicht:* x wird neu eingeführt; das λ bindet x !

Anstelle der Operatoren \gg und $\gg=$ kann alternativ die sogenannte **do**-Syntax verwendet werden.

```
getLine  :: IO String
getLine  = do c  $\leftarrow$  getChar
           if c == '\n' then return ""
                    else do s  $\leftarrow$  getLine
                           return (c : s)
```

Beispiel: Wörterbuch

Wir verwenden die bisher eingeführten Funktionen (insbesondere aus Aufgabe 6), um ein „interaktives Wörterbuch“ zu programmieren. Beispielsitzung:

```
basilisk> db asterix
db> lookup
key: alea iacta est
der Wuerfel ist gefallen
db> enter
key: acta est fabula
value: vorbei ist vorbei
db> save
db> end
```

Beispiel für ein Wörterbuch:

```
[("ab imo pectore", "von ganzem Herzen"),
 ("acta est fabula", "vorbei ist vorbei"), ...
```

. . . Beispiel: Wörterbuch . . .

```
import IO
import System

type State = (FilePath, Dict String String)

main      :: IO ()
main      = do hSetBuffering stdout NoBuffering
           args ← getArgs
           let fname = args !! 0
               cnts ← readFile fname
               loop (fname, read cnts)

loop      :: State → IO ()
loop state = do putStr "db> "
           line ← getLine
           state' ← exec state (words line)
           loop state'

askFor    :: String → IO String
askFor s  = putStr s >> getLine
```

. . . Beispiel: Wörterbuch

```
exec           :: State → [String] → IO State
exec state []   = return state
exec state@(fname, dict) (cmd : _)
  = case cmd of
    "end"        → exitWith ExitSuccess
    "enter"      → do key ← askFor "key: "
                    val ← askFor "value: "
                    return (fname, add (key, val) dict)
    "lookup"     → do key ← askFor "key: "
                    putStrLn (lookupWithDefault dict
                                "not found" key)
                    return state
    "save"       → do writeFile fname (show dict)
                    return state
    _           → do putStrLn "unknown command"
                    return state
```

Der Glasgow Haskell Compiler

Der Glasgow Haskell Compiler (GHC) ist ein Batchcompiler ähnlich wie cc. Typische Vorgehensweise:

1. Das Programm (z.B. Database.lhs) wird mit einem Texteditor erstellt. **Beachte:** der Bezeichner *main :: IO ()* muß definiert werden.
2. Das Programm wird übersetzt (-o gibt den Namen des ausführbaren Programms an).

```
basilisk> ghc -o db Database.lhs
```

3. Das ausführbare Programm wird gestartet.

```
basilisk> db asterix  
db> ...
```

GHC bietet eine Unzahl von Optionen an (siehe Handbuch).

GHC interaktiv

Der Glasgow Haskell Compiler umfaßt auch einen Interpreter, der mit `ghci` (GHC interactive) gestartet wird.

Die 'Umgebung' ist der von Hugs sehr ähnlich.

Highlights:

- ✘ eine flexible Bedienungsschnittstelle,
- ✘ (sehr) gute Fehlermeldungen,
- ✘ gemischte Ausführung von kompiliertem und interpretiertem Code.

Haskell 98 Kurs

- ✓ Ein paar Worte vorneweg
- ✓ Haskell in einer Nußschale
- ✗ Ausdrücke und Definitionen
 - Boolesche Ausdrücke
 - Zeichen und Zeichenketten
 - Zahlen
 - Tupel
 - Listen
 - Funktionen
 - **case**-Ausdrücke
- ✗ Typen und Typklassen
- ✗ Ein- und Ausgabe
- ✗ Module und Bibliotheken
- ✗ Programmieretechniken

Boolesche Ausdrücke

Die Wahrheitswerte sind vordefiniert.

```
data Bool = False | True
```

Fallunterscheidungen können vielfältig programmiert werden: mit

```
if <exp> then <exp> else <exp> ,
```

mit Wächtern . . . (später mehr dazu).

Aufgabe 12 Definiere die logische Konjunktion $\&\&$ und die Disjunktion $\|\!$. Ist der Ausdruck $n \neq 0 \&\& 1 / n > eps$ sinnvoll?

Aufgabe 13 Definiere die Funktion $elem :: (Eq\ a) \Rightarrow a \rightarrow [a] \rightarrow Bool$? Der Aufruf $elem\ a\ x$ ergibt $True$, falls das Element a in der Liste x enthalten ist. Sonst $False$.

Zeichen und Zeichenketten

Der Typ *Char* umfaßt die ASCII-Zeichen. Zeichen werden „wie gewohnt“ notiert: 'a', '\n', '\BEL' etc. (siehe §2.6).

Zeichenketten sind Listen von Zeichen.

```
type String = [Char]
```

Für Zeichenketten gibt es die „übliche“ abkürzende Notation: "ha" statt 'h':'a':[].

Zahlen

Haskell verfügt über eine große Zahl numerischer Typen:

<i>Int</i>	ganze Zahlen mit beschränkter Genauigkeit (32 bit)
<i>Integer</i>	ganze Zahlen mit <i>unbeschränkter</i> Genauigkeit
<i>Float</i>	Fließkommazahlen (einfache Genauigkeit)
<i>Double</i>	Fließkommazahlen (doppelte Genauigkeit)
<i>Ratio a</i>	Rationale Zahlen ($(Integral\ a) \Rightarrow Ratio\ a$)
<i>Complex a</i>	Komplexe Zahlen ($(RealFloat\ a) \Rightarrow Complex\ a$)

Die arithmetischen (+, −, * etc.) und zahlentheoretischen Funktionen (*log*, *exp* etc.) sind *überladen* (später mehr zu diesem Thema).

<i>fac</i>		::	$(Integral\ a) \Rightarrow a \rightarrow a$
<i>fac n</i>	<i>n == 0</i>	=	1
	<i>otherwise</i>	=	$n * fac\ (n - 1)$

Tupel

Tupel beliebiger Stelligkeit sind vordefiniert.

```
data ()           = ()      -- das leere Tupel
data (a, b)       = (a, b)   -- Paare
data (a, b, c)    = (a, b, c) -- Tripel
...
```

Beachte: für Tupel (z.B. $(1, 'a', True)$) und Tupeltypen (z.B. $(Int, Char, Bool)$) wird die gleiche Syntax verwendet.

Tupel werden benutzt, um eine *feste* Zahl von Werten *unterschiedlichen* Typs zusammenzufassen.

Listen

Listen stellen — wie gesagt — die wichtigste Datenstruktur in Haskell dar.

```
data [a] = [] | a : [a]
```

Listen werden benutzt, um eine *beliebige* Zahl von Werten *gleichen* Typs zusammenzufassen.

Es gibt — wie gesagt — Myriaden vordefinierter Funktionen (siehe *Prelude* und Standardbibliothek *List*). Eine Auswahl:

```
head      :: [a] → a  
tail     :: [a] → [a]  
(++)     :: [a] → [a] → [a]  
(\\)     :: (Eq a) ⇒ [a] → [a] → [a]  
length  :: [a] → Int  
(!!)     :: (Integral a) ⇒ [b] → a → b  
take, drop :: (Integral a) ⇒ a → [b] → [a]
```

. . . Listen . . .

<i>lines, words</i>	::	$String \rightarrow [String]$
<i>unlines, unwords</i>	::	$[String] \rightarrow String$
<i>and, or</i>	::	$[Bool] \rightarrow Bool$
<i>sum, product</i>	::	$(Num\ a) \Rightarrow [a] \rightarrow a$
<i>elem, notElem</i>	::	$(Eq\ a) \Rightarrow a \rightarrow [a] \rightarrow Bool$
<i>concat</i>	::	$[[a]] \rightarrow [a]$

Um Listen zu konstruieren, gibt es vielfältige Möglichkeiten. Arithmetische Folgen können z.B. wie folgt notiert werden.

$[1..]$	Liste aller positiven Zahlen
$[1..99]$	dito bis einschließlich 99
$[1,3..]$	Liste aller positiven, ungeraden Zahlen
$[1,3..99]$	dito bis einschließlich 99
$['a'..'z']$	Liste aller Buchstaben

Listenbeschreibungen . . .

Um Funktionen auf Listen zu definieren, haben wir bisher rekursive Gleichungen verwendet. Einfacher und lesbarer sind in vielen Fällen *Listenbeschreibungen*.

$$\begin{aligned} \text{elem } a \ x &= \text{or } [a == b \mid b \leftarrow x] \\ \text{concat } xs &= [a \mid x \leftarrow xs, a \leftarrow x] \end{aligned}$$

Naive Definition von Quicksort.

$$\begin{aligned} \text{qsort } [] &= [] \\ \text{qsort } (a : x) &= \text{qsort } [b \mid b \leftarrow x, b \leq a] \\ &\ ++ \ [a] \\ &\ ++ \ \text{qsort } [b \mid b \leftarrow x, b > a] \end{aligned}$$

Mathematiker. Die Syntax von Listenbeschreibungen ist eng an die Notation von Mengen (z.B. $\{n \mid n \in \mathbb{N}, n \geq 7\}$) angelehnt. Im Unterschied zu Mengen spielt die Reihenfolge eine Rolle und es können Duplikate auftreten.

. . . Listenbeschreibungen . . .

Beispiel: $perms\ x$ berechnet die Liste aller Permutationen von x .

$perms$	$::$	$[a] \rightarrow [[a]]$
$perms []$	$=$	$ [[]]$
$perms\ x$	$=$	$[a : y' \mid (a, y) \leftarrow remove\ x, y' \leftarrow perms\ y]$
$remove$	$::$	$[a] \rightarrow [(a, [a])]$
$remove []$	$=$	$ []$
$remove\ (a : x)$	$=$	$(a, x) : [(b, a : y) \mid (b, y) \leftarrow remove\ x]$

Aufgabe 14 Zur Abschreckung: Formuliere $perms$ und $remove$ ohne Listenbeschreibungen zu verwenden.

. . . Listenbeschreibungen . . .

Das bekannte n -Damen-Problem läßt sich wie folgt lösen: *queens n* berechnet die Liste *aller* Lösungen.

```
type Board    = [Int]
queens         :: Int → [Board]
queens n      = place n [] [] [1..n]
place c d1 d2 rs
  | c == 0    = [[]]
  | otherwise = [q : qs
                 | (q, rs') ← remove rs,
                   (q - c) 'notElem' d1,
                   (q + c) 'notElem' d2,
                   qs ← place (c - 1) ((q - c) : d1) ((q + c) : d2) rs']
```

Beachte: wird mit *queens 8 !! 0* nur auf die erste Lösung zugegriffen, werden die übrigen *nicht* generiert (lazy evaluation).

. . . Listenbeschreibungen

Listenbeschreibungen haben die Form

$$[\langle \text{exp} \rangle \mid \langle \text{qual} \rangle_1, \dots, \langle \text{qual} \rangle_n]$$

mit

$$\begin{array}{l} \langle \text{qual} \rangle \quad \rightarrow \quad \langle \text{pat} \rangle \leftarrow \langle \text{exp} \rangle \quad \text{Generator} \\ \quad \quad \quad \mid \quad \langle \text{exp} \rangle \quad \quad \quad \text{Wächter} \end{array}$$

Für den Generator $p \leftarrow e$ muß gelten: Wenn p vom Typ τ ist, muß e vom Typ $[\tau]$ sein. Generatoren führen Variablen ein, die „weiter rechts“ oder im Kopf verwendet werden können. Wächter, das sind Boolesche Ausdrücke, schränken die Belegungen von Variablen ein.

C-Programmierer. Der Ausdruck $[i * j \mid i \leftarrow [0..9], j \leftarrow [i..9]]$ entspricht zwei ineinander geschachtelten Schleifen.

```
for (int i=0; i<=9; i++)
    for (int j=i; j<=9; j++)
        printf("%d\n", i*j);
```

Funktionen . . .

Als *funktionale* Sprache bietet Haskell die Möglichkeit, Funktionen durch Ausdrücke zu bezeichnen, d.h. ohne eine Definition vorzunehmen.

$$\lambda \langle \text{pat} \rangle_1 \dots \langle \text{pat} \rangle_n \rightarrow \langle \text{exp} \rangle$$

Ist p_i vom Typ τ_i und e vom Typ τ , so besitzt der Ausdruck $\lambda p_1 \dots p_n \rightarrow e$ den Typ $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$.

Die Anwendung einer Funktion auf Argumente wird ohne Syntax notiert.

$$\langle \text{exp} \rangle \langle \text{exp} \rangle_1 \dots \langle \text{exp} \rangle_n$$

Hat e den Typ $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ und e_i den Typ τ_i , so besitzt $e e_1 \dots e_n$ den Typ τ .

. . . Funktionen . . .

Haskell kennt *nur* einstellige Funktionen.

$$\begin{aligned} \mathit{addInt} & \quad :: \quad \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\ \mathit{addInt} \ a \ b & \quad = \quad a + b \end{aligned}$$

Der Typausdruck $\mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}$ kürzt $\mathit{Int} \rightarrow (\mathit{Int} \rightarrow \mathit{Int})$, d.h., \rightarrow assoziiert von rechts. Die Funktion addInt erwartet demnach als Argument eine ganze Zahl und gibt eine Funktion zurück!

$$\begin{aligned} \mathit{addInt}' & \quad :: \quad (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{Int} \\ \mathit{addInt}' \ (a, b) & \quad = \quad a + b \end{aligned}$$

Die Funktion addInt' erwartet als Argument ein Paar. Sei $\mathit{dup} \ a = (a, a)$ gegeben, dann ist $\mathit{addInt}' \ (\mathit{dup} \ 5)$ ein gültiger Ausdruck.

. . . Funktionen

Funktionen des Typs $\tau_1 \rightarrow \tau_2 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ heißen *gestaffelt* (oder curryfiziert: aha, daher der Namen Haskell). Gestaffelte Funktionen sind im allgemeinen flexibler (und effizienter!) als Funktionen, die auf Tupeln arbeiten. Betrachten wir

$$\text{lookupWithDefault} \quad :: \quad (\text{Ord } a) \Rightarrow \text{Dict } a \ b \rightarrow b \rightarrow a \rightarrow b$$

aus Aufgabe 6. Versorgt man diese Funktion „peu à peu“ mit Argumenten, so erhält man folgende Ableger:

lookupWithDefault Nachschlagen eines Wortes in einem beliebigen Wörterbuch mit beliebigem Default-Wert

lookupWithDefault germengl
Nachschlagen eines Wortes im Deutsch-Englisch Wörterbuch mit beliebigem Default-Wert

*lookupWithDefault germengl "*not found*"*
Nachschlagen eines Wortes im Deutsch-Englisch Wörterbuch mit dem Default-Wert "*not found*"

*lookupWithDefault germengl "*not found*" "Rechner"*
ergibt "computer"

case-Ausdrücke

Elemente eines Datentyps können mit Hilfe von **case** analysiert werden. Statt

$$\begin{aligned} \text{elem } a \ [] &= \text{False} \\ \text{elem } a \ (b : x) &= a == b \ || \ \text{elem } a \ x \end{aligned}$$

kann *elem* auch (weniger elegant) mit Hilfe von **case** definiert werden.

$$\begin{aligned} \text{elem} &= \lambda a \ y \rightarrow \text{case } y \ \text{of} \\ &\quad [] \rightarrow \text{False} \\ &\quad b : x \rightarrow a == b \ || \ \text{elem } a \ x \end{aligned}$$

Ein **case**-Ausdruck bietet sich an, wenn das *Ergebnis* eines Funktionsaufrufs weiterverarbeitet werden soll.

Aufgabe 15 Definiere `||` mit **case**.

Funktionsdefinitionen . . .

Funktionen können mit Hilfe mehrerer Gleichungen definiert werden. Eine Funktionsdefinition hat die folgende syntaktische Form.

$$\begin{array}{l} f \quad \langle \text{pat} \rangle_{11} \dots \langle \text{pat} \rangle_{1k} \quad \langle \text{match} \rangle_1 \\ \dots \\ f \quad \langle \text{pat} \rangle_{n1} \dots \langle \text{pat} \rangle_{nk} \quad \langle \text{match} \rangle_n \end{array}$$

wobei $\langle \text{match} \rangle$ eine der beiden folgenden Formen hat.

$$= \langle \text{exp} \rangle \text{ where } \{ \langle \text{decl} \rangle_1; \dots; \langle \text{decl} \rangle_n \}$$

oder

$$\begin{array}{l} | \langle \text{guard} \rangle_1 = \langle \text{exp} \rangle_1 \\ \dots \\ | \langle \text{guard} \rangle_m = \langle \text{exp} \rangle_m \\ \text{where } \{ \langle \text{decl} \rangle_1; \dots; \langle \text{decl} \rangle_n \} \end{array}$$

Mit **where** werden lokale Definitionen eingeführt (später mehr dazu). Die Wächter sind Boolesche Ausdrücke, die über die Anwendung einer Gleichung wachen.

. . . Funktionsdefinitionen . . .

Beispiel: eine endrekursive Variante von Quicksort.

```
qsort                :: (Ord a) => [a] -> [a]
qsort []             = []
qsort [a]            = [a]
qsort (a : x)        = partition [] [] x
  where
    partition l r [] = qsort l ++ a : qsort r
    partition l r (b : y)
      | b <= a       = partition (b : l) r y
      | otherwise    = partition l (b : r) y
```

. . . Funktionsdefinitionen . . .

Laut Syntax werden einzelne Definitionen durch ‘;’ getrennt.

... **where** { $\langle \text{decl} \rangle_1; \dots; \langle \text{decl} \rangle_n$ }

Wir haben bisher weder geschweifte Klammern noch ein Semikolon verwendet. Die sogenannte *Abseitsregel* erlaubt es, Definitionen „ohne syntaktischen“ Ballast zu notieren.

Folgt dem **where** (bzw. **let** oder **of**) *keine* offene Klammer ‘{’, so tritt die Abseitsregel in Kraft: Die Einrückung der nächsten lexikalischen Einheit wird vermerkt. Mit den folgenden Zeilen wird wie folgt verfahren (leicht vereinfacht):

- ✘ kleinere Einrückung: es wird eine geschlossene Klammer ‘}’ eingefügt
- ✘ gleiche Einrückung: es wird ein Semikolon ‘;’ eingefügt
- ✘ größere Einrückung: es wird nichts eingefügt

. . . Funktionsdefinitionen . . .

Muster dienen sowohl zur Unterscheidung verschiedener Fälle als auch zur Benennung von Komponenten eines Wertes.

$$\mathit{head} (a : x) = a$$

Die Gleichung ist nur anwendbar, wenn der aktuelle Parameter die Form $e_1 : e_2$ hat (ggf. muß der Parameter reduziert werden). In diesem Fall wird a an den Listenkopf e_1 und x an den Listenrest e_2 gebunden.

Muster werden von *links nach rechts* und Gleichungen von *oben nach unten* abgearbeitet. Somit spielt unter Umständen die Reihenfolge von Gleichungen eine Rolle.

$$\begin{aligned} \mathit{take} (n + 1) (a : x) &= a : \mathit{take} n x \\ \mathit{take} _ _ &= [] \end{aligned}$$

Ein guter Programmierer ;-)) versucht, dies so weit wie möglich zu vermeiden.

. . . Funktionsdefinitionen

Muster haben die folgende Form (leicht vereinfacht).

$\langle \text{pat} \rangle$	\rightarrow	$\langle \text{var} \rangle [@ \langle \text{pat} \rangle]$	geschachteltes Muster
		$\langle \text{con} \rangle \langle \text{pat} \rangle_1 \dots \langle \text{pat} \rangle_n$	Konstruktoranwendung
		$[-] \langle \text{integer} \rangle$	Konstante
		$\langle \text{var} \rangle + \langle \text{integer} \rangle$	Nachfolgermuster
		$-$	anonyme Variable
		$(\langle \text{pat} \rangle_1, \dots, \langle \text{pat} \rangle_n)$	Tupelmuster
		$[\langle \text{pat} \rangle_1, \dots, \langle \text{pat} \rangle_n]$	Listenmuster

Ein Muster der Form $x + k$ paßt auf den Wert n , falls $n \geq k$ und bindet x an $n - k$.

. . . Wertedefinitionen . . .

Wertedefinitionen werden häufig verwendet, um strukturierte Rückgabewerte zu verarbeiten.

Beispiel: Die Funktion *build* überführt eine Liste in einen vollständigen balancierten Binärbaum.

<i>build</i>	::	$[a] \rightarrow Tree\ a$
<i>build</i> <i>x</i>	=	$fst\ (buildn\ (length\ x)\ x)$
<i>buildn</i>	::	$Int \rightarrow [a] \rightarrow (Tree\ a, [a])$
<i>buildn</i> 0 <i>x</i>	=	$(Empty, x)$
<i>buildn</i> (<i>n</i> + 1) <i>x</i>	=	$(Node\ l\ a\ r, z)$
where <i>m</i>	=	$n\ 'div'\ 2$
(<i>l</i> , <i>a</i> : <i>y</i>)	=	$buildn\ m\ x$
(<i>r</i> , <i>z</i>)	=	$buildn\ (n - m)\ y$

. . . Wertedefinitionen

Beachte: die Reihenfolge der lokalen Bindungen spielt *keine* Rolle. Die Auswertungsreihenfolge wird allein durch die Datenabhängigkeiten bestimmt.

```
...  
where m           = n 'div' 2  
        (l, a : y)  = buildn m x  
        (r, z)      = buildn (n - m) y
```

Beachte: Wertebindungen können auch fehlschlagen: Bei der Auswertung von

```
buildn 4 [1]
```

tritt ein Fehlschlag auf: $(l, a : y)$ paßt nicht auf $(Node\ Empty\ 1\ Empty, [])$. Der Zugriff auf eine der Variablen l , a oder y führt zum *Programmabbruch*.

Lokale Definitionen

Mit **let** und **where** werden lokale Definitionen eingeführt. Nur Typdeklarationen ($f :: \tau$), Funktions- und Wertedefinitionen dürfen lokal verwendet werden, *keine* Typdefinitionen (**data**, **type**).

Unterschiede zwischen **let** und **where**:

- ✘ **let** {⟨decls⟩} **in** e ist ein Ausdruck;
= e **where** {⟨decls⟩} ist syntaktischer Bestandteil einer Gleichung.
- ✘ Die in **let** {⟨decls⟩} **in** e definierten Bezeichner sind in e sichtbar;
die Sichtbarkeit der Definitionen in
 - | $g_1 = e_1$
 - ...
 - | $g_m = e_m$**where** { ⟨decls⟩ }
erstreckt sich auf die e_i und die g_i .

C-Programmierer. Die Sichtbarkeitsregeln entsprechen denen in blockstrukturierten Sprachen.

Haskell 98 Kurs

- ✓ Ein paar Worte vorneweg
- ✓ Haskell in einer Nußschale
- ✓ Ausdrücke und Definitionen
- ✗ Typen und Typklassen
 - Datentypen
 - Beispiel: Text mit Einrückung
 - Typsynonyme
 - Typklassen
 - Beispiel: Text mit Einrückung
- ✗ Ein- und Ausgabe
- ✗ Module und Bibliotheken
- ✗ Programmieretechniken

. . . Datentypen . . .

Mit Ausnahme von Funktionen können alle vordefinierte Typen als Datentypen definiert werden.

```
data Bool      = False | True
data Char      = '\NUL' | ... | '\255'
data Int       = -65532 | ... | -1 | 0 | ... | 65532
data ()        = ()      -- das leere Tupel
data (a, b)    = (a, b)   -- Paare
data (a, b, c) = (a, b, c) -- Tripel
data [a]       = [] | a : [a]
```

Die Typen *Integer*, *Float* und *Double* sind schwieriger zu realisieren.

Aufgabe 16 Überlege, wie der Typ *Integer*, ganze Zahlen mit *beliebiger* Genauigkeit, dargestellt werden kann!

. . . Datentypen . . .

Datentypen subsumieren *Aufzählungstypen*.

```
data Color = Red | Green | Blue
```

Sie umfassen Tupeltypen: Jeder Datentyp mit einem Konstruktor heißt *Tupeltyp*.

```
data Point = Point Int Int
```

Sie verallgemeinern *Vereinigungstypen*.

```
data Maybe a = Nothing | Just a  
data Either a b = Left a | Right b
```

Datentypen können parametrisiert werden und sie dürfen rekursiv definiert sein.

Aufgabe 17 Wie können Bäume mit einem *beliebigen* Verzweigungsgrad definiert werden?

. . . Datentypen . . .

Konstruktoren werden in Mustern verwendet, um Funktionen auf den neu eingeführten Datentypen zu definieren:

Die Fallunterscheidung wird durch die Datentypdefinition diktiert. [In den meisten Fällen jedenfalls.]

Also: 3 Konstruktoren bedingen 3 Gleichungen bzw. 3 Fälle; rekursive Datentypdefinitionen führen zu rekursiven Funktionsdefinitionen.

Auch Korrektheitsbeweise orientieren sich an der Datentypdefinition.

Der Nachweis der Korrektheit erfolgt induktiv über die Struktur des Datentyps. [In vielen Fällen.]

Also: nicht rekursive Konstruktoren ergeben Basisfälle, rekursive Konstruktoren werden induktiv behandelt.

. . . Datentypen

Mit Hilfe der vordefinierten Datentypen *Maybe* und *Either* lassen sich „Ausnahmen“ programmieren (später mehr dazu). Zum *Beispiel*: die Suche nach einem Element ist erfolgreich oder schlägt fehl.

```
lookup :: (Ord a) => Tree (a, b) -> a -> Maybe b
lookup Empty b = Nothing
lookup (Node l (a, v) r) b
  | b < a      = lookup l b
  | b == a     = Just v
  | b > a      = lookup r b
lookupWithDefault :: (Ord a) => Tree (a, b) -> b -> a -> b
lookupWithDefault t def b = case lookup t b of
  Nothing -> def
  Just v -> v
```

Beispiel: Text mit Einrückung. . .

Aufgabe: Implementiere folgende Funktionen für die Erstellung 'hierarchischer Dokumente' (vulgo: Text mit Einrückung).

```
data Text
text    :: String → Text    -- ohne '\n'
nl      :: Text
indent  :: Int → Text → Text
(◇)     :: Text → Text → Text
render  :: Text → String
```

Semantik: *text* überführt einen String in Text, *nl* ist ein Zeilenvorschub, *render i t* fügt *i* Leerzeichen *nach* jedem Zeilenvorschub in *t* ein, *◇* konkateniert zwei Texte, und *render* schließlich überführt Text in einen String.

. . . Beispiel: Text mit Einrückung. . .

Die Funktionen lassen sich zum Beispiel verwenden, um Binärbäume darzustellen: die Knoten werden entsprechend ihrer Tiefe (Entfernung zur Wurzel) eingerückt.

```
layTree           :: Tree Int → Text
layTree Empty    = text "Empty"
layTree (Node l a r) = indent 4 (text "Node" ◇ nl ◇
                          layTree l ◇ nl ◇
                          text (show a) ◇ nl ◇
                          layTree r)
```

Der Aufruf *render (layTree (Node (Node Empty 4711 Empty) 12 Empty))* ergibt:

```
Node
  Node
    Empty
    4711
    Empty
  12
Empty
```

. . . Beispiel: Text mit Einrückung. . .

Eine einfache Referenzimplementierung: alle Funktionen (mit Ausnahme von *render*) werden durch Konstruktoren implementiert.

```
data Text = Text String -- ohne '\n'  
          | Newline  
          | Indent Int Text  
          | Text :◇ Text  
  
text      = Text  
nl        = Newline  
indent    = Indent  
(◇)      = (:◇)
```

. . . Beispiel: Text mit Einrückung. . .

Die Funktion *render* überführt einen *Text* in einen *String*.

<i>render</i>	::	<i>Text</i> → <i>String</i>
<i>render</i> (<i>Text</i> <i>s</i>)	=	<i>s</i>
<i>render</i> <i>Newline</i>	=	"\n"
<i>render</i> (<i>Indent</i> <i>i</i> <i>d</i>)	=	<i>tab</i> <i>i</i> (<i>render</i> <i>d</i>)
<i>render</i> (<i>d1</i> :◇ <i>d2</i>)	=	<i>render</i> <i>d1</i> † <i>render</i> <i>d2</i>
<i>tab</i>	::	<i>Int</i> → <i>String</i> → <i>String</i>
<i>tab</i> <i>i</i> ""	=	""
<i>tab</i> <i>i</i> (<i>c</i> : <i>s</i>)		
<i>c</i> == '\n'	=	<i>c</i> : <i>replicate</i> <i>i</i> ' ' † <i>tab</i> <i>i</i> <i>s</i>
<i>otherwise</i>	=	<i>c</i> : <i>tab</i> <i>i</i> <i>s</i>

Die Hilfsfunktion *tab* *i* *s* fügt *i* Leerzeichen *nach* jedem Zeilenvorschub in *s* ein.

. . . Beispiel: Text mit Einrückung . . .

Wir wünschen uns eine ganze Reihe von Eigenschaften:

$$\begin{aligned} \text{text } "" \diamond d &= d \\ d \diamond \text{text } "" &= d \\ d1 \diamond (d2 \diamond d3) &= (d1 \diamond d2) \diamond d3 \\ \text{indent } 0 \ d &= d \\ \text{indent } i \ (\text{text } s) &= \text{text } s \\ \text{indent } i \ nl &= nl \diamond \text{text } (\text{replicate } i \ ' \ ') \\ \text{indent } i \ (\text{indent } j \ d) &= \text{indent } (i + j) \ d \\ \text{indent } i \ (d1 \diamond d2) &= \text{indent } i \ d1 \diamond \text{indent } i \ d2 \\ \text{render } (\text{text } s) &= s \\ \text{render } nl &= "\n" \\ \text{render } (d1 \diamond d2) &= \text{render } d1 \ ++ \ \text{render } d2 \end{aligned}$$

Beachte: In der Referenzimplementierung sind einige Gesetze nur *unter Beobachtung* gültig: statt $d1 = d2$ gilt nur $\text{render } d1 = \text{render } d2$.

. . . Beispiel: Text mit Einrückung . . .

Wir können die Effizienz von *render* verbessern, indem wir ‘mehrere Dinge’ auf einmal machen. *Spezifikation*:

$$\mathit{render}'\ i\ \mathit{doc}\ x = \mathit{render}\ (\mathit{indent}\ i\ \mathit{doc})\ \mathit{++}\ x$$

Die spezifizierte Funktion *render'* rückt einen Text ein, überführt das Ergebnis in einen String und hängt an diesen String einen anderen String an. Puuuh! *Herleitung*:
Fall $\mathit{doc} = \mathit{Text}\ s$:

$$\begin{aligned} & \mathit{render}'\ i\ (\mathit{Text}\ s)\ x \\ = & \quad \{ \text{Spezifikation von } \mathit{render}' \} \\ & \mathit{render}\ (\mathit{indent}\ i\ (\mathit{Text}\ s))\ \mathit{++}\ x \\ = & \quad \{ \mathit{indent}\ i\ (\mathit{text}\ s) = \mathit{text}\ s \} \\ & \mathit{render}\ (\mathit{Text}\ s)\ \mathit{++}\ x \\ = & \quad \{ \mathit{render}\ (\mathit{text}\ s) = s \} \\ & s\ \mathit{++}\ x \end{aligned}$$

. . . Beispiel: Text mit Einrückung . . .

Fall $doc = Newline$:

$$\begin{aligned} & \text{render}' i \text{ Newline } x \\ = & \quad \{ \text{Spezifikation von } \text{render}' \} \\ & \text{render} (\text{indent } i \text{ Newline}) \text{ ++ } x \\ = & \quad \{ \text{indent } i \text{ nl} = \text{nl} \diamond \text{text} (\text{replicate } i \text{ ' '}) \} \\ & \text{render} (\text{nl} \diamond \text{text} (\text{replicate } i \text{ ' '})) \text{ ++ } x \\ = & \quad \{ \text{render} (d1 \diamond d2) = \text{render } d1 \text{ ++ render } d2 \} \\ & \text{render } \text{nl} \text{ ++ render} (\text{text} (\text{replicate } i \text{ ' '})) \text{ ++ } x \\ = & \quad \{ \text{render } \text{nl} = "\backslash\mathbf{n}" \} \\ & "\backslash\mathbf{n}" \text{ ++ render} (\text{text} (\text{replicate } i \text{ ' '})) \text{ ++ } x \\ = & \quad \{ \text{Definition von ++ und } \text{render} (\text{text } s) = s \} \\ & \text{'\backslash n'} : \text{replicate } i \text{ ' ' ++ } x \end{aligned}$$

. . . Beispiel: Text mit Einrückung . . .

Fall $doc = Indent\ j\ d$:

$$\begin{aligned} & render' i (Indent\ j\ d)\ x \\ = & \{ \text{Spezifikation von } render' \} \\ & render (indent\ i (Indent\ j\ d)) ++ x \\ = & \{ indent\ i (indent\ j\ d) = indent\ (i + j)\ d \} \\ & render (indent\ (i + j)\ d) ++ x \\ = & \{ \text{Spezifikation von } render' \} \\ & render' (i + j)\ d\ x \end{aligned}$$

. . . Beispiel: Text mit Einrückung . . .

Fall $doc = d1 : \diamond d2$:

$$\begin{aligned} & \text{render}' i (d1 : \diamond d2) x \\ = & \quad \{ \text{Spezifikation von } \text{render}' \} \\ & \text{render} (\text{indent } i (d1 : \diamond d2)) \text{++ } x \\ = & \quad \{ \text{indent } i (d1 \diamond d2) = \text{indent } i d1 \diamond \text{indent } i d2 \} \\ & \text{render} (\text{indent } i d1 \diamond \text{indent } i d2) \text{++ } x \\ = & \quad \{ \text{render} (d1 \diamond d2) = \text{render } d1 \text{++ } \text{render } d2 \} \\ & \text{render} (\text{indent } i d1) \text{++ } \text{render} (\text{indent } i d2) \text{++ } x \\ = & \quad \{ \text{Spezifikation von } \text{render}' \} \\ & \text{render} (\text{indent } i d1) \text{++ } \text{render}' i d2 x \\ = & \quad \{ \text{Spezifikation von } \text{render}' \} \\ & \text{render}' i d1 (\text{render}' i d2 x) \end{aligned}$$

. . . Beispiel: Text mit Einrückung . . .

Wir haben die folgende Definition von $render'$ hergeleitet:

$$\begin{aligned} render' &:: Int \rightarrow Text \rightarrow String \rightarrow String \\ render' i (Text s) x &= s ++ x \\ render' i Newline x &= '\n' : replicate i ' ' ++ x \\ render' i (Indent j d) x &= render' (i + j) d x \\ render' i (d1 :◇ d2) x &= render' i d1 (render' i d2 x) \end{aligned}$$

Die Laufzeit ist linear zur Größe des erzeugten Strings (betrachte zum Beispiel $render' (factorial\ 1000)\ nl\ ""$).

. . . Beispiel: Text mit Einrückung

Wir müssen noch *render* auf *render'* zurückführen.

$$\begin{aligned} & \textit{render } d \\ = & \{ x ++ "" = x \} \\ & \textit{render } d ++ "" \\ = & \{ \textit{indent } 0 \ d = d \} \\ & \textit{render } (\textit{indent } 0 \ d) ++ "" \\ = & \{ \text{Spezifikation von } \textit{render}' \} \\ & \textit{render}' 0 \ d ++ "" \end{aligned}$$

$$\textit{render } d = \textit{render}' 0 \ d ++ ""$$

Typsynonyme

Mit Hilfe von Typsynonymen können Namen für Typausdrücke eingeführt werden.

```
type  $T \alpha_1 \dots \alpha_k = \tau$ 
```

Typsynonyme werden verwendet, um „große“ Typausdrücke abzukürzen und um die Lesbarkeit des Programms durch Verwendung aussagekräftiger Typnamen zu erhöhen.

```
type String    = [Char]    -- vordefiniert  
type FilePath = String    -- vordefiniert  
type Set a    = [a]  
type Dict a b = Tree (a, b)
```

Beachte: Im Gegensatz zu Datentypdefinitionen dürfen Typsynonyme *nicht* rekursiv definiert werden.

Typklassen . . .

Namen zu erfinden ist schwer!

Typklassen erlauben es, den gleichen Namen für syntaktisch verschiedene, aber semantisch ähnliche Funktionen zu verwenden. Kandidaten für überladene Funktionen:

$(==)$, (\leq) **etc.** Die Vergleichsoperationen lassen sich auf (fast) allen Typen sinnvoll definieren.

$(+)$, $(*)$ **etc.** Fast jede Sprache definiert die arithmetischen Verknüpfungen auf ganzen Zahlen *und* Fließkommazahlen.

show, *read* Die Überführung zwischen interner und externer Repräsentation (als Zeichenkette) ist für (fast) alle Typen nützlich.

Beachte: (\leq) ist für verschiedene Typen, z.B. *Int* und [*Char*], unterschiedlich definiert.

. . . Typklassen . . .

Überladene Funktionen kommen meist in Gruppen. Eine Typklasse deklariert eine solche Gruppe als überladen.

```
class Eq a where
  (==) :: a → a → Bool
  (≠) :: a → a → Bool
```

Lies: der Typ a ist eine Instanz der Typklasse Eq , wenn auf a die Operationen $(=)$ und (\neq) definiert sind. Die Funktionen einer Typklasse nennt man auch *Methoden*.

Typklassen schränken die Gültigkeit eines Typs ein.

```
elem :: (Eq a) ⇒ a → a → Bool
```

Lies: für alle Typen a , auf denen die Gleichheit definiert ist, ist $a \rightarrow a \rightarrow Bool$ ein gültiger Typ von $elem$.

Typklassen können als *Prädikate auf* oder *Eigenschaften von* Typen aufgefaßt werden, daher die Schreibweise.

. . . Typklassen . . .

Die Zugehörigkeit eines Typs zu einer Typklasse wird *explizit* durch eine Instanzdeklaration angegeben.

```
instance Eq Int where  
  m == n    = intEq m n  
  m /= n    = not (m == n)
```

intEq ist die primitive Funktion, die zwei ganze Zahlen auf Gleichheit testet.

```
instance (Eq a) => Eq (Tree a) where  
  Empty == Empty           = True  
  Node l1 a1 r1 == Node l2 a2 r2 = l1 == l2 && a1 == a2 && r1 == r2  
  _ == _                   = False  
  t /= u                    = not (t == u)
```

D.h., *Tree a* ist eine Instanz von *Eq*, wenn *a* eine Instanz von *Eq* ist.

. . . Typklassen . . .

Methoden können in einer Klassendeklaration für alle Instanzen definiert werden.

```
class Eq a where
```

```
  (==), (≠)  ::  a → a → Bool
```

```
  x ≠ y      =  not (x == y)
```

Falls in einer Instanzdeklaration \neq nicht spezifiziert wird, verwendet der Übersetzer die Definition aus der Typklasse.

```
instance Eq Int where
```

```
  m == n    =  intEq m n
```

. . . Typklassen . . .

Typklassen können sich auf andere Klassen, sogenannte *Oberklassen*, abstützen.
Beispiel: Die Vergleichsoperationen $<$, \leq etc. sind nur sinnvoll auf Typen, die auch $=$ zur Verfügung stellen (Auszug aus der Klassendeklaration).

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

Aufgabe 18 Vergleiche Haskells Typklassen mit dem Klassenkonzept objektorientierter Sprachen.

. . . Typklassen . . .

Die vollständige Deklaration der Typklasse *Ord* lautet.

```
data Ordering      = LT | EQ | GT
class (Eq a) => Ord a where
  compare          :: a -> a -> Ordering
  (<), (≤), (≥), (>) :: a -> a -> Bool
  max, min        :: a -> a -> a

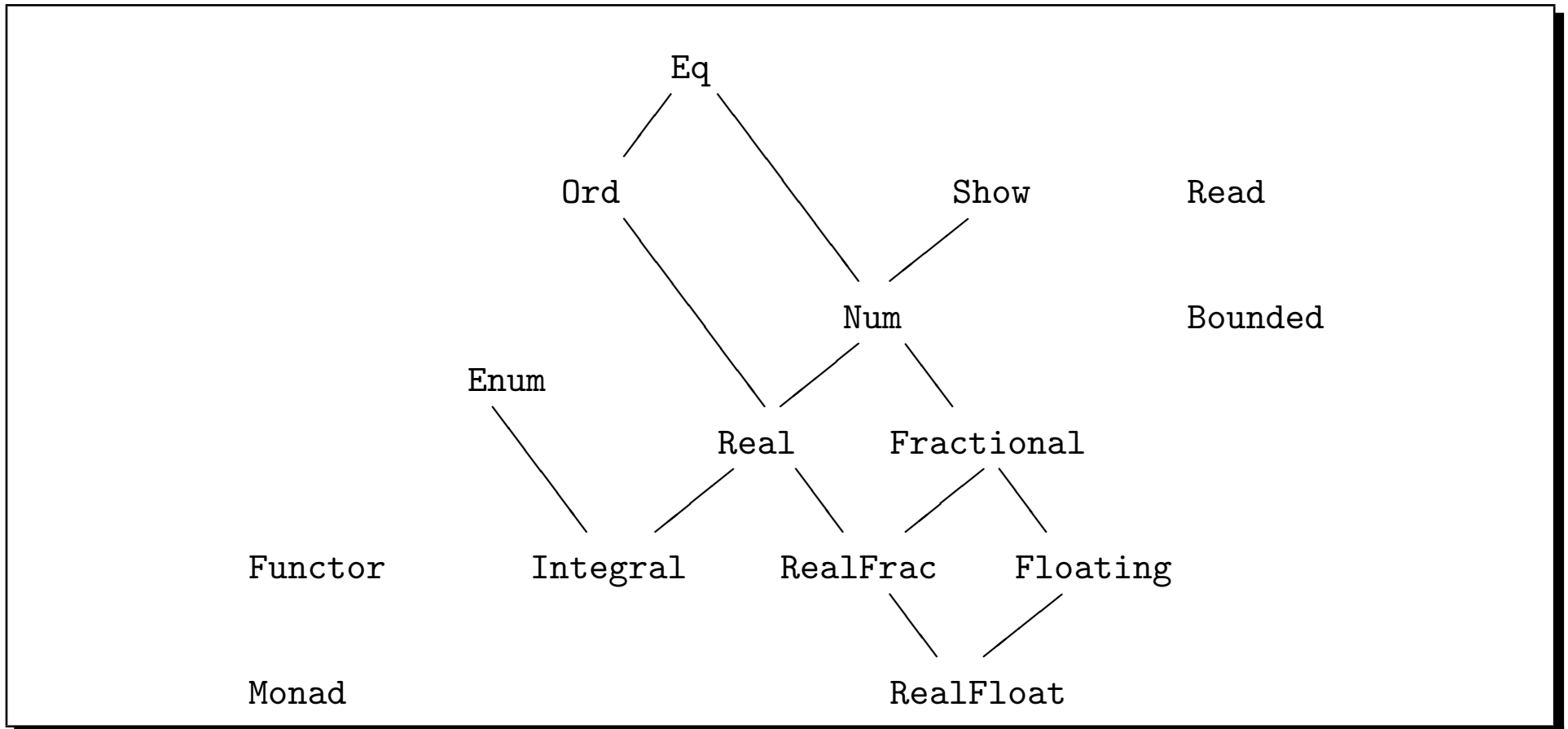
  compare x y
    | x == y      = EQ
    | x ≤ y       = LT
    | otherwise   = GT

  x ≤ y          = compare x y ≠ GT
  x < y          = compare x y == LT
  x ≥ y          = compare x y ≠ LT
  x > y          = compare x y == GT
```

Beachte: Eine Instanz muß mindestens \leq oder *compare* definieren.

. . . Typklassen . . .

Die Standardklassen von Haskell sind wie folgt organisiert.



Eq und *Ord* haben wir schon kennengelernt; mit *Enum*, *Show* etc. und den numerischen Klassen *Num*, *Real* etc. beschäftigen wir uns im folgenden.

. . . Typklassen . . .

Die arithmetischen Folgen sind Abkürzungen für:

$[n \dots]$	=	$enumFrom\ n$	-- Pseudocode
$[n, m \dots]$	=	$enumFromThen\ n\ m$	
$[n \dots m]$	=	$enumFromTo\ n\ m$	
$[n, n' \dots m]$	=	$enumFromThenTo\ n\ n'\ m$	

Die Typklasse *Enum* ermöglicht es, diese Funktionen zu überladen.

. . . Typklassen . . .

class *Enum* *a* **where**

<i>succ, pred</i>	::	$a \rightarrow a$
<i>toEnum</i>	::	$Int \rightarrow a$
<i>fromEnum</i>	::	$a \rightarrow Int$
<i>enumFrom</i>	::	$a \rightarrow [a] \quad \text{-- } [n \dots]$
<i>enumFromThen</i>	::	$a \rightarrow a \rightarrow [a] \quad \text{-- } [n, n' \dots]$
<i>enumFromTo</i>	::	$a \rightarrow a \rightarrow [a] \quad \text{-- } [n \dots m]$
<i>enumFromThenTo</i>	::	$a \rightarrow a \rightarrow a \rightarrow [a] \quad \text{-- } [n, n' \dots m]$
<i>succ</i>	=	$toEnum \cdot (+1) \cdot fromEnum$
<i>pred</i>	=	$toEnum \cdot (subtract\ 1) \cdot fromEnum$
<i>enumFrom x</i>	=	$map\ toEnum\ [fromEnum\ x \dots]$
<i>enumFromTo x y</i>	=	$map\ toEnum\ [fromEnum\ x \dots fromEnum\ y]$
<i>enumFromThen x y</i>	=	$map\ toEnum\ [fromEnum\ x, fromEnum\ y \dots]$
<i>enumFromThenTo x y z</i>	=	$map\ toEnum$ $[fromEnum\ x, fromEnum\ y \dots fromEnum\ z]$

. . . Typklassen . . .

Die Klassen *Show* und *Read* stellen Funktionen zur Verfügung, um Werte „aus-“ und „einzugeben“ (die Klassendeklarationen sehen wir uns später an).

```
show  :: (Show a) => a -> String
read  :: (Read a) => String -> a
```

Fast alle vordefinierten Typen sind Instanzen der Typklassen *Show* und *Read*. Wir haben bei der Programmierung des „interaktiven Wörterbuchs“ Gebrauch von *show* und *read* gemacht.

```
... readFile fname >>= \cnts ->
      ... read cnts ...
... writeFile fname (show dict)
```

show realisiert einen „Pretty Printer“, *read* einen Parser (später mehr dazu).

. . . Typklassen . . .

Instanzen der Typklassen *Eq*, *Ord*, *Enum*, *Bounded*, *Show* und *Read* können vom Übersetzer automatisch erzeugt werden.

```
data Tree lab = Empty | Node (Tree lab) lab (Tree lab)
                deriving (Eq, Ord, Show, Read)
```

Mit dem Zusatz **deriving** lassen sich folgende kanonische Instanzen ableiten.

Eq Identitätsrelation

Ord lexikographische Ordnung (die Konstruktoren werden entsprechend der Reihenfolge ihres Auftretens angeordnet)

Enum, *Bounded* dito (*nur* für Aufzählungstypen)

Show, *Read* „Aus- und Eingabe“ entsprechend der textuellen Repräsentation im Programmtext

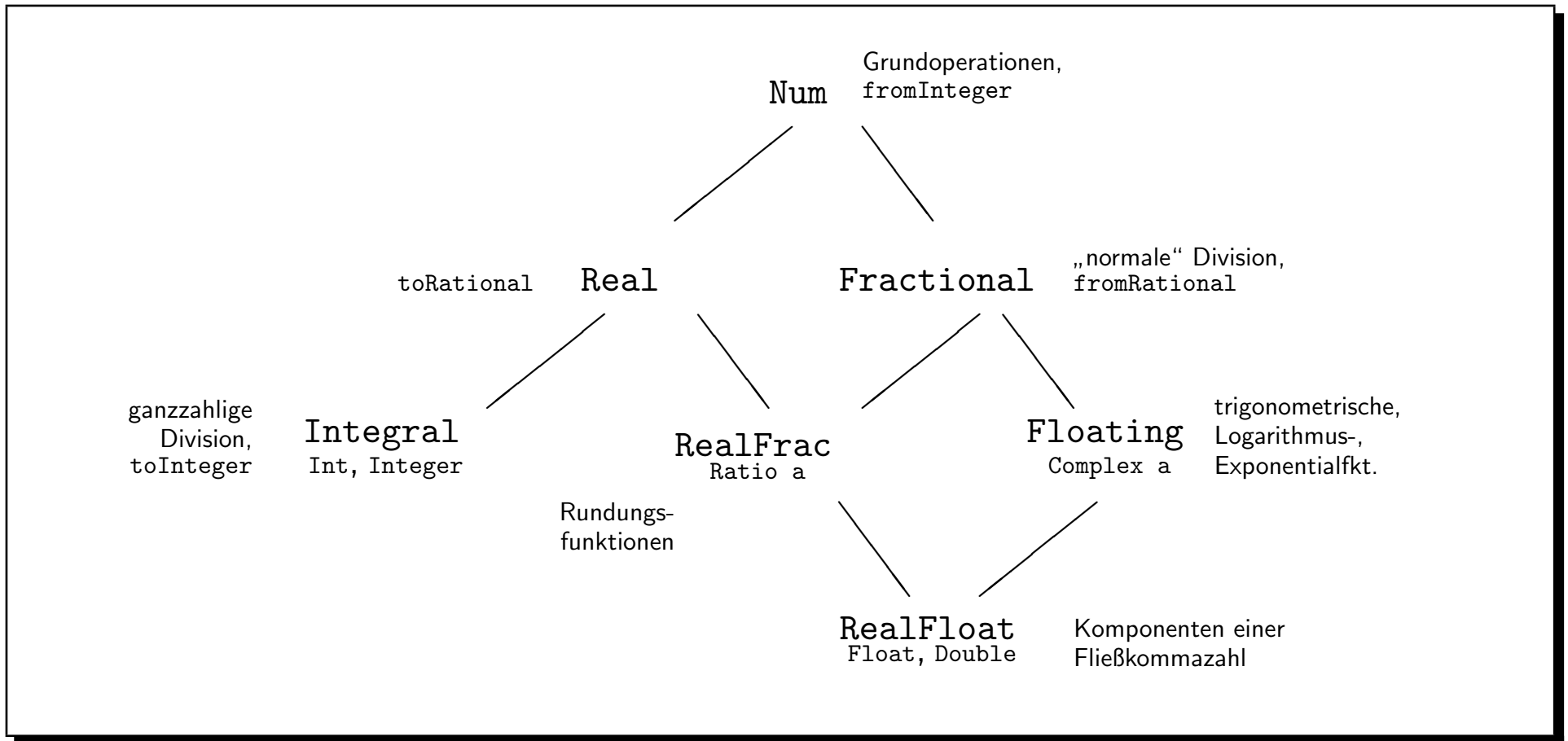
. . . Typklassen . . .

Den größten Teil der Klassenhierarchie machen numerische Klassen aus. Wie erklären sich die Klassen und ihre Anordnung?

- ✘ Mit jedem der Typpaare *Int/Integer*, *Float/Double*, *Ratio Int/Ratio Integer*, *Complex Float/Complex Double* korrespondiert eine Klasse: *Integral*, *RealFrac*, *RealFloat* und *Floating*.
- ✘ Auf den numerischen Typen sind circa 50 primitive Funktionen definiert: (+), (*), *div*, *mod*, (/), *exp*, *log*, *sin*, *cos*, *ceiling*, *floor*, *exponent*, *significand* und viele andere mehr.
Gemeinsame Funktionen verschiedener Typklassen werden in Oberklassen „hochgezogen“: *Real*, *Fractional* und *Num*.
- ✘ Einbettung in die Klassenhierarchie: Auf komplexen Zahlen läßt sich keine Ordnung definieren; alle Zahlen lassen sich auf Gleichheit testen und ein- und ausgeben.

. . . Typklassen . . .

Die numerischen Standardklassen von Haskell sind wie folgt organisiert.



. . . Typklassen . . .

In der Typklasse *Num* sind die arithmetischen Grundoperationen zusammengefaßt, die allen numerischen Typen gemeinsam sind.

```
class (Eq a, Show a)  $\Rightarrow$  Num a where  
  (+), (-), (*)  :: a  $\rightarrow$  a  $\rightarrow$  a  
  negate        :: a  $\rightarrow$  a  
  abs, signum   :: a  $\rightarrow$  a  
  fromInteger   :: Integer  $\rightarrow$  a  
  
  x - y        = x + negate y  
  negate x     = 0 - x
```

Beachte: $-x * y$ ist äquivalent zu *negate* ($x * y$).

Für die Definition der übrigen Klassen sei auf den Haskell Report verwiesen (§6.3).

. . . Typklassen . . .

Der Typ der rationalen Zahlen ist abstrakt (später mehr zu abstrakten Typen).

```
data (Integral a)  $\Rightarrow$  Ratio a  
type Rational = Ratio Integer
```

Rationale Zahlen werden mit `%` konstruiert. Zähler und Nenner einer rationalen Zahl erhält man mit *numerator* und *denominator*.

```
(%)      :: (Integral a)  $\Rightarrow$  a  $\rightarrow$  a  $\rightarrow$  Ratio a  
numerator  :: (Integral a)  $\Rightarrow$  Ratio a  $\rightarrow$  a  
denominator :: (Integral a)  $\Rightarrow$  Ratio a  $\rightarrow$  a
```

Beachte: rationale Zahlen werden automatisch gekürzt. Aus diesem Grund ist *numerator* (*x % y*) im allgemeinen nicht gleich *x*.

. . . Typklassen . . .

Wie notiere ich Elemente der numerischen Typen? Anders gefragt: Welchen Typ haben die Literale 27 oder 3.1415?

Lösung: das Literal 27 ist eine Abkürzung für *fromInteger* ($27 :: Integer$) und das Literal 3.1415 für *fromRational* ($3.1415 :: Rational$).

$fromInteger \quad :: \quad (Num \ a) \Rightarrow Integer \rightarrow a$
$fromRational \quad :: \quad (Fractional \ a) \Rightarrow Rational \rightarrow a$

Das heißt, die numerischen Literale sind implizit überladen. *Vorteil:* Sie können für alle numerischen Typen verwendet werden.

Aufgabe 19 Warum wird 3.1415 als *rationale* Zahl interpretiert und nicht als *Fließkommazahl*?

. . . Typklassen . . .

Komplexe Zahlen sind wie folgt definiert.

$$\mathbf{data} \ (RealFloat\ a) \Rightarrow Complex\ a \ = \ a\ :+ \ a$$

Die komplexe Zahl $a + bi$ wird als $a\ :+ \ b$ notiert. Der Konstruktor $:+$ kann wie üblich als Muster verwendet werden.

$$\begin{aligned} conjugate & \quad \quad \quad :: \ (RealFloat\ a) \Rightarrow Complex\ a \rightarrow Complex\ a \\ conjugate\ (x\ :+ \ y) & \ = \ x\ :+ \ (-y) \end{aligned}$$

. . . Typklassen . . .

Überladene Funktionen lassen sich nicht immer eindeutig auflösen.

```
pyth      :: (Floating a) => a -> a -> a
pyth x y  =  sqrt (x ^ 2 + y ^ 2)
```

Die Funktion $(^)$ hat den Typ $(Num\ a, Integral\ b) \Rightarrow a \rightarrow b \rightarrow a$, 2 hat den Typ $(Num\ b) \Rightarrow b$; somit erhält $x ^ 2$ den Typ $(Num\ a, Integral\ b) \Rightarrow a$.

Problem: Welcher Typ soll für b gewählt werden? *Int* oder *Integer*?

Lösung: für *numerische* Typen kann das mit Hilfe einer **default** Deklaration festgelegt werden.

```
default (Int, Double)
```

Der erste Typ, der „paßt“, wird genommen.

. . . Typklassen

Das Problem tritt nicht nur bei numerischen Funktionen auf.

```
whoops    :: String → String
whoops x  = show (read x)
```

Die Funktion *read* hat den Typ $(Read\ a) \Rightarrow String \rightarrow a$, die Funktion *show* hat den Typ $(Show\ a) \Rightarrow a \rightarrow String$; somit erhält *show (read x)* den Typ $(Read\ a, Show\ a) \Rightarrow String \rightarrow String$.

Problem: Welcher Typ soll für *a* gewählt werden? Alle Typen, die Instanzen von *Read* und *Show* sind, kommen in Frage!

Lösung: In diesen Fällen muß der Typ explizit angegeben werden.

```
whoops    :: String → String
whoops x  = show (read x :: [Int])
```

Beispiel: Text mit Einrückung . . .

Die Funktion $layTree :: Tree Int \rightarrow Text$ läßt sich sinnvoll zu einer Methode $lay :: (Lay a) \Rightarrow a \rightarrow Text$ verallgemeinern.

```
class (Show a)  $\Rightarrow$  Lay a where  
    lay    :: a  $\rightarrow$  Text  
    lay a  = text (show a)  
instance Lay Int  
instance Lay Integer
```

Lay ist eine Unterklasse von *Show* (jede Instanz von *Lay* muß auch Instanz von *Show* sein).

. . . Beispiel: Text mit Einrückung

```
instance (Lay a) ⇒ Lay (Tree a) where  
  lay Empty          = text "Empty"  
  lay (Node l a r)  = indent 4 (text "Node" ◇ nl ◇  
                                lay l ◇ nl ◇  
                                lay a ◇ nl ◇  
                                lay r)
```

Aufgabe 20 Mache *Text* selbst zu einer Instanz von *Lay*.

Haskell 98 Kurs

- ✓ Ein paar Worte vorneweg
- ✓ Haskell in einer Nußschale
- ✓ Ausdrücke und Definitionen
- ✓ Typen und Typklassen
- ✗ Ein- und Ausgabe
 - Vordefinierte EA-Aktionen
 - **do**-Notation
 - *sequence_* und *sequence*
 - Standardbibliothek *System*
 - Fehlerbehandlung
 - Beispiel: *echo*
 - Beispiel: *cat*
- ✗ Module und Bibliotheken
- ✗ Programmiertechniken

Vordefinierte EA-Aktionen . . .

Zur Erinnerung: mit Hilfe des abstrakten Typs *IO val* werden Aktionen *beschrieben*, die Ein- oder Ausgaben vornehmen. Allgemeiner: . . . , die mit dem Betriebssystem oder der Umgebung kommunizieren.

```
data IO val
```

Ein Element vom Typ *IO val* *beschreibt* eine EA-Aktion, die einen Wert vom Typ *val* zurückgibt.

Vordefinierte Ausgabefunktionen:

```
putChar   :: Char → IO ()  
putStr    :: String → IO ()  
putStrLn :: String → IO ()  
print     :: (Show a) ⇒ a → IO ()
```

. . . Vordefinierte EA-Aktionen . . .

Vordefinierte Eingabefunktionen:

```
getChar   :: IO Char  
getLine  :: IO String  
readLine :: (Read a) ⇒ IO a
```

Vordefinierte Operationen auf Dateien.

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

Beachte: *readFile* arbeitet „lazy“: Der Einleseprozeß wird erst durch den Zugriff auf die Zeichenkette angestoßen.

. . . Vordefinierte EA-Aktionen . . .

Verknüpfung von EA-Aktionen:

```
return  :: a → IO a  
(>>)    :: IO a → IO b → IO b  
(>>=)   :: IO a → (a → IO b) → IO b
```

C-Programmierer. *return* bleibt *return*; *act1 >> act2* läßt sich als *act1; act2* lesen und *act1 >>= λx → act2* als *x = act1; act2*. *Vorsicht: x* wird neu eingeführt; das *λ* bindet *x*!

Jedes Haskell Programm muß den Bezeichner *main :: IO ()* definieren.

```
main = print [(i, i ^ 2) | i ← [0..9]]
```

Beim Programmstart wird die an *main* gebundene Aktion (respektive die Beschreibung der Aktion) ausgeführt.

. . . Vordefinierte EA-Aktionen

Beispiel: Das Programm fragt den Benutzer nach einem Dateinamen und gibt den Inhalt der Datei auf dem Bildschirm aus.

```
askFor    :: String → IO String  
askFor s  = putStr s >> getLine  
main     :: IO ()  
main     = askFor "filename: " >>= λfile →  
           readFile file >>= λcnts →  
           putStr cnts
```

do-Notation . . .

Um die Erstellung interaktiver Programme zu erleichtern, gibt es die sogenannte **do**-Notation.

do { $\langle \text{stat} \rangle_1; \dots; \langle \text{stat} \rangle_n; \langle \text{exp} \rangle$ }

mit

$\langle \text{stat} \rangle$	\rightarrow	$\langle \text{exp} \rangle$	
		$\langle \text{pat} \rangle \leftarrow \langle \text{exp} \rangle$	Generator
		let { $\langle \text{decl} \rangle_1; \dots; \langle \text{decl} \rangle_n$ }	lokale Definition

Die Ausdrücke müssen jeweils vom Typ $IO \tau$ sein. Für den Generator $p \leftarrow e$ muß gelten: Wenn p vom Typ τ ist, muß e vom Typ $IO \tau$ sein. Generatoren führen Variablen ein, die „weiter rechts“ verwendet werden können. Mit **let** werden lokale Definitionen eingeführt, die ebenfalls „weiter rechts“ sichtbar sind.

. . . **do**-Notation . . .

Die **do**-Notation ist lediglich „syntaktischer Zucker“ und kann wie folgt in \gg und \ggg übersetzt werden (leicht vereinfacht).

$$\begin{aligned}\mathbf{do} \{ e \} &= e \\ \mathbf{do} \{ e; stmts \} &= e \gg \mathbf{do} \{ stmts \} \\ \mathbf{do} \{ p \leftarrow e; stmts \} &= e \ggg \lambda p \rightarrow \mathbf{do} \{ stmts \} \\ \mathbf{do} \{ \mathbf{let} \text{ decls}; stmts \} &= \mathbf{let} \text{ decls} \mathbf{in} \mathbf{do} \{ stmts \}\end{aligned}$$

. . . do-Notation

Beispiel: Das obige Programm in **do**-Notation.

```
askFor    :: String → IO String  
askFor s  = do putStr s  
           getLine  
  
main     :: IO ()  
main     = do file ← askFor "filename: "  
           cnts ← readFile file  
           putStr cnts
```

Beachte: Auch für die **do**-Notation gilt die Abseitsregel.

sequence_ und sequence

Die Funktionen *sequence_* und *sequence* führen Listen von Aktionen aus.

```
sequence_      :: [IO a] → IO ()  
sequence_ []   = return ()  
sequence_ (a : as) = a >> sequence_ as  
  
sequence      :: [IO a] → IO [a]  
sequence []    = return []  
sequence (a : as) = a >>= λv →  
                    sequence as >>= λvs →  
                    return (v : vs)
```

sequence_ und *sequence* werden gerne mit Listenbeschreibungen kombiniert.

```
askForMany    :: [String] → IO [String]  
askForMany xs = sequence [askFor s | s ← xs]
```

Standardbibliothek *System*

Die Standardbibliothek *System* definiert Operationen, um mit dem Betriebssystem zu kommunizieren.

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

Fehlerbehandlung . . .

EA-Aktionen können *fehlschlagen*: eine Datei ist z.B. nicht vorhanden oder nicht lesbar. Derartige Fehler können abgefangen oder auch selbst ausgelöst werden.

data *IOError*

userError :: *String* → *IOError*

ioError :: *IOError* → *IO a*

catch :: *IO a* → (*IOError* → *IO a*) → *IO a*

fail :: *String* → *IO a*

fail s = *ioError (userError s)*

. . . Fehlerbehandlung . . .

Die Standardbibliothek *IO* definiert verschiedene Funktionen, mit denen die Fehlerart bestimmt werden kann.

<i>isAlreadyExistsError</i>	::	<i>IOError</i>	→	<i>Bool</i>
<i>isDoesNotExistError</i>	::	<i>IOError</i>	→	<i>Bool</i>
<i>isAlreadyInUseError</i>	::	<i>IOError</i>	→	<i>Bool</i>
<i>isFullError</i>	::	<i>IOError</i>	→	<i>Bool</i>
<i>isEOFError</i>	::	<i>IOError</i>	→	<i>Bool</i>
<i>isIllegalOperation</i>	::	<i>IOError</i>	→	<i>Bool</i>
<i>isPermissionError</i>	::	<i>IOError</i>	→	<i>Bool</i>
<i>isUserError</i>	::	<i>IOError</i>	→	<i>Bool</i>

. . . Fehlerbehandlung . . .

Abgeleitete Operationen:

$$\begin{aligned} \text{try} & \quad :: \quad IO\ a \rightarrow IO\ (Either\ IOError\ a) \\ \text{try } p & \quad = \quad (p \gg= (\text{return} \cdot \text{Right})) \\ & \quad \quad \quad \text{'catch'}(\text{return} \cdot \text{Left}) \\ \text{either} & \quad :: \quad (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow (Either\ a\ b) \rightarrow c \\ \text{either } f\ g\ (\text{Left } x) & \quad = \quad f\ x \\ \text{either } f\ g\ (\text{Right } x) & \quad = \quad g\ x \end{aligned}$$

Typische Anwendung von *try* und *either*:

$$\text{try } a \gg= \text{either } (\lambda \text{err} \rightarrow \dots) (\lambda \text{ok} \rightarrow \dots)$$

. . . Fehlerbehandlung

Beispiel: Ausgabe einer Datei mit Fehlerbehandlung.

```
askFor    :: String → IO String
askFor s = putStr s >>
            getLine 'catch' λerr →
            if IO.isEOFError err then exitWith ExitSuccess
              else ioError err

main     = askFor "filename: " >>= λfile →
            try (readFile file) >>=
            either
            (λerr → if IO.isDoesNotExistError err
                  then putStrLn "file does not exist"
                  else ioError err)
            (λcnts → putStr cnts)
```

Beispiel: *echo*

Beispiel: Realisierung des UNIX-Kommandos *echo*.

NAME

echo - display a line of text

SYNOPSIS

echo [-n] [string ...]

Haskell Programm:

```
main           = getArgs >>= echo
echo []        = return ()
echo ("-n" : x) = putStr (unwords x)
echo x        = putStrLn (unwords x)
```

Beispiel: *cat* . . .

Beispiel: Realisierung des UNIX-Kommandos *cat* (vereinfacht).

NAME

cat - concatenate files and print on the standard output

SYNOPSIS

cat [-n] [file...]

Haskell Programm:

```
import IO
import System

main      =  getArgs >>= cat
cat       ::  [String] → IO ()
cat []    =  return ()
cat ("-n" : x) = sequence_ [copy True f | f ← x]
cat x     =  sequence_ [copy False f | f ← x]
```

. . . Beispiel: *cat*

```
copy           :: Bool → FilePath → IO ()  
copy b f      = do cnts ← readFile f  
                sequence_ [  
                    do when b (putLineNo n)  
                        putStrLn l  
                    | (n, l) ← zip [1..] (lines cnts)]  
  
putLineNo     :: Int → IO ()  
putLineNo n  = putStr (rjustify 6 (show n) ++ " ")  
  
when         :: Bool → IO () → IO ()  
when b a     = if b then a else return ()  
  
rjustify     :: Int → String → String  
rjustify n s = replicate (n - length s) ' ' ++ s
```

Haskell 98 Kurs

- ✓ Ein paar Worte vorneweg
- ✓ Haskell in einer Nußschale
- ✓ Ausdrücke und Definitionen
- ✓ Typen und Typklassen
- ✓ Ein- und Ausgabe
- ✗ Module und Bibliotheken
 - Aufbau eines Haskell Programms
 - Export
 - Import
 - Qualifizierter Import
 - Abstrakte Datentypen
 - Programmentwicklung
- ✗ Programmieretechniken

Aufbau eines Haskell Programms

Zur Erinnerung: Ein Haskell Programm besteht aus einem oder mehreren *Modulen*.

Ein Modul definiert Werte, Datentypen, Typsynonyme und Typklassen. Diese Einheiten können *exportiert* oder von anderen Modulen *importiert* werden.

Ein Modul hat die folgende syntaktische Form.

```
module <modid> ( <export>1, ..., <export>k )  
where {  
    <impdecl>1; ...; <impdecl>m;  
    <topdecl>1; ...; <topdecl>n  
}
```

Wird der Modulkopf nicht angegeben (so wie bisher), wird implizit als Modulkopf **module** *Main* **where** eingesetzt.

Export . . .

In der Exportliste können Einheiten aufgeführt werden, die in dem Modul definiert oder aus einem anderen Modul importiert werden.

Werte durch Angabe des Namens,

Datentypen in der Form $T(C_1, \dots, C_n)$: die aufgeführten Konstruktoren werden exportiert,
oder $T(\dots)$: Abkürzung für alle Konstruktoren,
oder T : die Konstruktoren werden *nicht* exportiert,

Typsynonyme in der Form T

Typklassen in der Form $C(f_1, \dots, f_n)$: einige Methoden werden aufgeführt,
oder $C(\dots)$: Abkürzung für alle Methoden,
oder C : die Methoden werden *nicht* exportiert,

Module in der Form **module** M : alle importierten Einheiten werden reexportiert.

Beachte: auch das aktuelle Modul kann angegeben werden.

. . . Export

Beispiel: polymorphe Binärbäume als Modul.

```
module Tree (Tree (Empty, Node), insert, delete, ...)
where
data Tree lab = Empty | Node (Tree lab) lab (Tree lab)
           deriving (Eq, Ord, Show, Read)
insert      :: (Ord lab) => lab -> Tree lab -> Tree lab
delete     :: (Ord lab) => lab -> Tree lab -> Tree lab
join       :: IntTree -> IntTree -> IntTree
split     :: IntTree -> Int -> IntTree -> (Int, IntTree)
...
```

Beachte: *join* und *split* sind nur lokal sichtbar.

Beachte: die Instanzdeklarationen **instance** (*Eq a*) => *Eq (Tree a)* etc. werden automatisch exportiert.

Import . . .

Importdeklarationen können zwei verschiedene Formen haben.

$\langle \text{impdecl} \rangle$	\rightarrow	import	$\langle \text{modid} \rangle$	$[\langle \text{impspec} \rangle]$
$\langle \text{impspec} \rangle$	\rightarrow	$(\langle \text{import} \rangle_1, \dots, \langle \text{import} \rangle_n)$		positiv
		<i>hiding</i>	$(\langle \text{import} \rangle_1, \dots, \langle \text{import} \rangle_n)$	negativ

Die zu importierenden Einheiten werden wie in der Exportliste angegeben. Klar: nur das kann importiert werden, was auch exportiert wird.

- ✘ keine Importspezifikation: alle Einheiten werden importiert,
- ✘ positive Importspezifikation: nur die aufgeführten Einheiten werden importiert,
- ✘ negative Importspezifikation (*hiding*): alle Einheiten *bis* auf die angegeben werden importiert.

. . . Import

Beispiel: Wir verwenden Binärbäume um „Splay Trees“ zu implementieren.

```
module Splay (splay, join)  
where  
import Tree (Tree (Empty, Node))  
splay  :: (a → Ordering) → Tree a → (Ordering, Tree a)  
join   :: Tree a → Tree a → Tree a  
...
```

Qualifizierter Import . . .

Was tun, wenn es knallt? Werden zwei *verschiedene* Einheiten *gleichen* Namens importiert, gibt es eine *Namenskollision*. [Wie gesagt, Namen zu erfinden ist schwer.]

```
import List (insert)
import Tree (insert)
... insert ... insert ...
```

Mit Hilfe 'qualifizierter' Imports werden Namenskollisionen vermieden.

```
import qualified List (insert)
import qualified Tree (insert)
... Tree.insert ... List.insert ...
```

. . . Qualifizierter Import

Module können beim Import umbenannt werden.

```
import qualified BinarySearchTree as Set  
... Set.insert ...
```

Vorteil: Möchte man anstatt binärer Suchbäume geordnete Listen verwenden, muß nur die erste Zeile geändert werden.

Mehrere Module können auch den gleichen Namen erhalten!

```
import qualified HTMLCore as HTML  
import qualified HTMLExtensions as HTML
```

Abstrakte Datentypen

Mit Hilfe des Modulsystems lassen sich *abstrakte Datentypen* definieren. Im Unterschied zu normalen Datentypen ist die Repräsentation eines abstrakten Datentyps unbekannt: ein abstrakter Typ definiert sich über sein Verhalten, über die Menge der auf dem Typ definierten Operationen.

```
module Set (Set, empty, add, ...)
where
import Tree
data Set a      = Set (Tree a)
empty           :: Set a
empty           = Set Empty
add             :: a → Set a → Set a
add a (Set x)   = Set (insert a x)
...
```

Daß Mengen durch Binärbäume repräsentiert werden, ist außerhalb des Moduls nicht sichtbar. *Vorteil:* Die Repräsentation läßt sich problemlos ändern.

Programmentwicklung

Klar: Eine größere Programmieraufgabe sollte man als Sammlung von Modulen realisieren, die einzeln entwickelt und getestet werden können.

Pro Modul wird eine Datei des *gleichen* Namens (und der Endung `.hs` oder `.lhs`) angelegt.

Mit `ghc --make Main.lhs` läßt sich ein ausführbares Programm erzeugen (alle benötigten Module werden automatisch übersetzt). *Tip:* mit der Option `-Wall` erhält man Warnungen, die möglicherweise Programmfehler aufzeigen.

Für die Entwicklung einzelner Module bieten sich Hugs (Aufruf: *hugs*) oder GHCi (Aufruf: *ghci*) an.

Haskell 98 Kurs

- ✓ Ein paar Worte vorneweg
- ✓ Haskell in einer Nußschale
- ✓ Ausdrücke und Definitionen
- ✓ Typen und Typklassen
- ✓ Ein- und Ausgabe
- ✓ Module und Bibliotheken
- ✗ Programmiertechniken
 - Funktionen höherer Ordnung
 - „Lazy evaluation“
 - Pretty printing—Typklasse *Show*
 - Parsing—Typklasse *Read*
 - Monaden

Funktionen höherer Ordnung . . .

Ein zentrales Prinzip der Programmierung ist das *Abstraktionsprinzip* — das man auch verwendet, ohne es zu kennen.

Abstraktion, 1. Verallgemeinerung. 2. auf zufällige Einzelheiten verzichtende, begrifflich zusammengefaßte Darstellung.

Beispiel: Eine „Funktion“, die das Kugelvolumen zu einem speziellen Radius berechnet, ist fast nutzlos.

$$\begin{aligned} r &= 5 \\ \text{volume_of_}r &= 3 / 4 * pi * r ^ 3 \end{aligned}$$

Wird von dem speziellen Radius abstrahiert — indem der Radius zum Parameter gemacht wird — so erhält man eine Definition, die sehr viel nützlicher ist.

$$\begin{aligned} \text{volume } r &= 3 / 4 * pi * r ^ 3 \\ \text{volume_of_}r &= \text{volume } r \end{aligned}$$

. . . Funktionen höherer Ordnung . . .

Diesem Prinzip folgend kann z.B. eine Sortierfunktion mit der zugrundeliegenden Vergleichsfunktion parametrisiert werden.

<i>qsortBy</i>	::	$(a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow [a]$
<i>qsortBy</i> (\leq) []	=	[]
<i>qsortBy</i> (\leq) ($a : x$)	=	<i>qsortBy</i> (\leq) $l \uplus a : qsortBy$ (\leq) r
where (l, r)	=	<i>partition</i> ($\leq a$) x
<i>qsort</i>	::	$(Ord\ a) \Rightarrow [a] \rightarrow [a]$
<i>qsort</i>	=	<i>qsortBy</i> (\leq)

qsortBy ist eine sogenannte Funktion höherer Ordnung. [$f :: \tau_1 \rightarrow \tau_2$ heißt Funktion höherer Ordnung, wenn τ_1 oder τ_2 ein funktionaler Typ ist.]

. . . Funktionen höherer Ordnung . . .

Funktionen höherer Ordnung werden unter anderem (und gerne) verwendet, um *Rekursionsmuster* „einzufangen“. *Beispiel*: Den Funktionen *inclist* und *strupr* liegt ein gemeinsames Rekursionsmuster zugrunde.

$inclist []$	$=$	$[]$
$inclist (a : x)$	$=$	$a + 1 : inclist x$
$strupr []$	$=$	$[]$
$strupr (a : x)$	$=$	$toUpper a : strupr x$

Der Mathematiker würde sagen: Hier wird eine Funktion auf eine Liste fortgesetzt. Die Funktion *map* implementiert dieses Muster. Graphische Darstellung:

$$\begin{array}{c} [x_1, x_2, x_3, \dots, x_{n-2}, x_{n-1}, x_n] \\ \downarrow \text{map } f \\ [f\ x_1, f\ x_2, f\ x_3, \dots, f\ x_{n-2}, f\ x_{n-1}, f\ x_n] \end{array}$$

. . . Funktionen höherer Ordnung . . .

Auf jedem (polymorphen) Datentyp läßt sich sinnvoll eine „*map*“-Funktion definieren.

```
mapMaybe :: (a → b) → Maybe a → Maybe b
mapMaybe f Nothing = Nothing
mapMaybe f (Just a) = Just (f a)
data Tree lab = Node lab [Tree lab]
mapTree :: (a → b) → Tree a → Tree b
mapTree f (Node a ts) = Node (f a) (map (mapTree f) ts)
```

Mathematiker: Sei T ein einstelliger Typkonstruktor, dann hat die zugehörige „*map*“-Funktion den Typ $(\alpha \rightarrow \beta) \rightarrow (T \alpha \rightarrow T \beta)$. Mit einigen zusätzlichen Bedingungen bildet (T, map) einen *Funktor*.

. . . Funktionen höherer Ordnung . . .

Da sich auf jedem (polymorphen) Datentyp eine „*map*“-Funktion sinnvoll definieren läßt, ist die *map*-Funktion unter dem Namen *fmap* überladen.

```
class Functor f where
```

```
  fmap                :: (a → b) → f a → f b
```

```
instance Functor Maybe where
```

```
  fmap f Nothing    = Nothing
```

```
  fmap f (Just a)  = Just (f a)
```

```
instance Functor Tree where
```

```
  fmap f (Node a ts) = Node (f a) (fmap (fmap f) ts)
```

. . . Funktionen höherer Ordnung . . .

Ein anderes Rekursionsmuster liegt den folgenden Funktionen zugrunde.

$sum []$	$=$	0
$sum (a : x)$	$=$	$a + sum x$
$sequence_ []$	$=$	$return ()$
$sequence_ (a : as)$	$=$	$a \gg sequence_ as$

Die Funktion *foldr* (rechtsassoziative Auffaltung) implementiert dieses Muster. Graphische Darstellung:

$$\begin{array}{c} x_1 : x_2 : x_3 : \cdots : x_{n-2} : x_{n-1} : x_n : [] \\ \downarrow \text{foldr } (\otimes) e \\ x_1 \otimes (x_2 \otimes (x_3 \otimes \cdots \otimes (x_{n-2} \otimes (x_{n-1} \otimes (x_n \otimes e)))))) \end{array}$$

. . . Funktionen höherer Ordnung

Auf vielen (polymorphen) Typen läßt sich sinnvoll eine *fold*-Funktion definieren.

```
data Tree lab      = Node lab [Tree lab]
fold                :: (a → [b] → b) → Tree a → b
fold f (Node a ts) = f a (map (fold f) ts)
```

Spezialisierung von *fold*:

```
sumup    = fold (λa ns → a + sum ns)
size     = fold (λ_ ns → 1 + sum ns)
height   = fold (λ_ ns → 1 + maximum ns)
preorder = fold (λa xs → a : concat xs)
```

Aufgabe 22 Bestimme die Typen der obigen Funktionen!

Lazy evaluation . . .

Zur Erinnerung: Das Motto von Lazy Evaluation lautet:

Berechne einen Ausdruck bzw. einen Teilausdruck nur, wenn es unbedingt nötig ist und dann auch nur einmal.

„Lazy evaluation“ erlaubt einen sehr *modularen* Programmaufbau. Was wir damit meinen, läßt sich am besten anhand von Beispielen verdeutlichen . . .

. . . Lazy evaluation . . .

Beispiel: Wir programmieren eine Funktion, die eine Liste in Teillisten der Länge n unterteilt. Folgende Funktionen aus der Standardumgebung sind nützlich.

```
iterate      :: (a → a) → a → [a]
takeWhile   :: (a → Bool) → [a] → [a]
```

Die Funktion $group :: Int \rightarrow [a] \rightarrow [[a]]$ läßt sich als Komposition von drei Standardfunktionen definieren.

```
group n     = iterate (drop n)
              # takeWhile (not · null)
              # map (take n)
f # g      = g · f
```

Fazit: keine Scheu vor großen Zwischenlisten.

. . . Lazy evaluation . . .

Beispiel: Realisierung des UNIX-Kommandos *cat*.

NAME

cat - concatenate files and print on the standard output

SYNOPSIS

cat [-benstv] [file...]

DESCRIPTION

This manual page documents the GNU version of cat. cat writes the contents of each given file, or the standard input if none are given or when a file named '-' is given, to the standard output.

OPTIONS

- b Number all nonblank output lines, starting with 1.
- e Display a '\$' after the end of each line.
- n Number all output lines, starting with 1.
- s Replace multiple adjacent blank lines with a single blank line.
- t Display TAB characters as '^I'.
- v Display control characters except for LFD and TAB using '^' notation and precede characters that have the high bit set with 'M-'.

. . . Lazy evaluation . . .

Der Vollständigkeit halber das Hauptprogramm.

```
import IO
import System

main      =  getArgs >>= λargs →
             case args of
             ('-' : opts) : args' →
                 sequence_ [copy opts f | f ← args']
             _ → sequence_ [copy "" f | f ← args]

type Options = [Char]

copy      :: Options → FilePath → IO ()
copy opts f = readFile f >>= λcnts →
             let str = process opts cnts in
             putStr str
```

. . . Lazy evaluation . . .

Für das „squeezen“ von Leerzeilen führen wir einen neuen Datentyp ein, der es erlaubt zwischen normalen und Leerzeilen zu unterscheiden.

data <i>Line</i>	=	<i>Blank String</i>
		<i>Line String</i>
<i>classify</i>	::	<i>String</i> → <i>Line</i>
<i>classify s</i> <i>all isSpace s</i>	=	<i>Blank s</i>
<i>otherwise</i>	=	<i>Line s</i>
<i>unclassify</i>	::	<i>Line</i> → <i>String</i>
<i>unclassify (Blank s)</i>	=	<i>s</i>
<i>unclassify (Line s)</i>	=	<i>s</i>
<i>squeeze</i>	::	[<i>Line</i>] → [<i>Line</i>]
<i>squeeze []</i>	=	[]
<i>squeeze (Blank _ : x@(Blank _ : _))</i>	=	<i>squeeze x</i>
<i>squeeze (a : x)</i>	=	<i>a : squeeze x</i>

. . . Lazy evaluation . . .

Die Numerierung von Zeilen bzw. von nicht-leeren Zeilen übernehmen die Funktionen *number* bzw. *bnumber*.

<i>number</i>	::	$Int \rightarrow [String] \rightarrow [String]$
<i>number n</i>	=	$zip [n..] \# map\ lineno$
<i>bnumber</i>	::	$Int \rightarrow [Line] \rightarrow [Line]$
<i>bnumber n []</i>	=	$[]$
<i>bnumber n (Blank s : x)</i>	=	$Blank (indent\ s) : bnumber\ n\ x$
<i>bnumber n (Line s : x)</i>	=	$Line (lineno\ (n, s)) : bnumber\ (n + 1)\ x$
<i>indent</i>	::	$String \rightarrow String$
<i>indent s</i>	=	$replicate\ 8\ ' '\ ++\ s$
<i>lineno</i>	::	$(Int, String) \rightarrow String$
<i>lineno (n, s)</i>	=	$rjustify\ 6\ (show\ n) \++\ " \ " \++\ s$

. . . Lazy evaluation

Die Hauptarbeit erledigt *process*.

```
process                :: Options → String → String
process opts         =          lines
                       #          map classify
                       # when 's' squeeze
                       # when 'b' (bnumber 1)
                       #          map unclassify
                       # when 'n' (number 1)
                       # when 'e' (map (+ "$"))
                       #          unlines
                       # when 't' (concatMap showTab)
                       # when 'v' (concatMap showNonPrinting)
    where when c f     = if c 'elem' opts then f else id
showTab                :: Char → String
showTab c | c == '\t'  = "^I"
            | otherwise = [c]
```

Pretty printing—Typklasse *Show* . . .

Wie definiere ich eine Instanz von *Show*? *ODER*: Wie programmiere ich einen „Pretty Printer“?

Haskell verwendet leider keinen *Text* Datentyp, sondern . . .

Beispiel: Ausgabe von Bäumen.

```
data Tree lab = Empty | Node (Tree lab) lab (Tree lab)
```

showTree führt im Prinzip einen Inorder-Durchlauf durch.

```
showTree           :: (Show a) => Tree a -> String  
showTree Empty    = "Empty"  
showTree (Node l a r) = "(Node "  
                    ++ showTree l ++ " "  
                    ++ show a ++ " "  
                    ++ showTree r ++ ")"
```

. . . Pretty printing—Typklasse *Show* . . .

Problem: Die Laufzeit von *showTree* wächst quadratisch zur Baumgröße.

$lefty$	$::$	$Int \rightarrow Tree Int$
$lefty\ 0$	$=$	$Empty$
$lefty\ (n + 1)$	$=$	$Node\ (lefty\ n)\ n\ Empty$

. . . Pretty printing—Typklasse *Show* . . .

Die Verwendung von $\#$ verursacht die Laufzeitprobleme. *Lösung*: Wir programmieren eine Funktion, die einen Baum ausgibt *und* an das Ergebnis eine Zeichenkette anhängt. *Spezifikation*:

```
showsTree t x = showTree t  $\#$  x    -- Spezifikation
```

Aus der Spezifikation läßt sich die Implementierung einfach ableiten ;-).

```
showsTree                :: (Show a)  $\Rightarrow$  Tree a  $\rightarrow$  String  $\rightarrow$  String  
showsTree Empty x     = "Empty"  $\#$  x  
showsTree (Node l a r) x = "(Node "  
                           $\#$  showsTree l (" "  
                           $\#$  shows a (" "  
                           $\#$  showsTree r (")"  $\#$  x))  
showTree                :: (Show a)  $\Rightarrow$  Tree a  $\rightarrow$  String  
showTree t             = showsTree t ""
```

. . . Pretty printing—Typklasse *Show* . . .

showsTree läßt sich noch eleganter notieren.

```
type ShowS           = String → String
showsTree             :: (Show a) ⇒ Tree a → ShowS
showsTree Empty      = showString "Empty"
showsTree (Node l a r) = showString "(Node "
                        · showsTree l · showChar ' '
                        · shows a · showChar ' '
                        · showsTree r · showChar ')'
showChar              = (:)
showString            = (++)
```

C-Programmierer. Lies *showfoo* als *printf* und *'·'* als *','*.

. . . Pretty printing—Typklasse *Show* . . .

Einige Klammern in der Ausgabe können eingespart werden . . .

<i>showTree</i>	::	$(Show\ a) \Rightarrow Tree\ a \rightarrow String$
<i>showTree t</i>	=	<i>showsTreePrec</i> 0 <i>t</i> ""
<i>showsTreePrec</i>	::	$(Show\ a) \Rightarrow Int \rightarrow Tree\ a \rightarrow ShowS$
<i>showsTreePrec d Empty</i>	=	<i>showString</i> "Empty"
<i>showsTreePrec d (Node l a r)</i>	=	<i>showParen</i> ($d \geq 10$) (<i>showString</i> "Node " · <i>showsTreePrec</i> 10 <i>l</i> · <i>showChar</i> ' ' · <i>showsPrec</i> 10 <i>a</i> · <i>showChar</i> ' ' · <i>showsTreePrec</i> 10 <i>r</i>)
<i>showParen</i>	::	$Bool \rightarrow ShowS \rightarrow ShowS$
<i>showParen False p</i>	=	<i>p</i>
<i>showParen True p</i>	=	<i>showChar</i> ' (' · <i>p</i> · <i>showChar</i> ')'

. . . Pretty printing—Typklasse *Show*

Die Typklasse *Show* ist wie folgt definiert:

```
class Show a where
  show                :: a → String
  showsPrec          :: Int → a → ShowS
  showList           :: [a] → ShowS
  showsPrec _ x s    = show x ++ s
  show x             = showsPrec 0 x ""
  showList []         = showString "[]"
  showList (x : xs) = showChar '[' · shows x · showl xs
    where showl []    = showChar ']'
           showl (x : xs) = showChar ',' · shows x · showl xs
```

Aufgabe 23 Warum enthält die Typklasse *Show* zusätzlich die Funktion *showList*?

Parsing—Typklasse *Read* . . .

Wie programmiere ich einen Parser? *Beispiel*: Eingabe von Bäumen.

```
type ReadS a = String → [(a, String)]
```

Ein Parser überführt ein Anfangsstück seiner Eingabe in einen „semantischen“ Wert, der zusammen mit der restlichen Eingabe zurückgegeben wird. *ReadS* beschreibt *nicht-deterministische* Parser: Die Liste *aller Möglichkeiten*, Teile der Eingabe zu parsen, wird ermittelt.

```
(reads :: ReadS Int) "5 hamburger" ⇒ [(5, " hamburger")]
```

Ein Parser für die lexikalische Analyse ist vordefiniert.

```
lex :: ReadS String
```

lex ermittelt die nächste lexikalische Einheit („token“).

. . . Parsing—Typklasse *Read* . . .

Ein Parser für Binärbäume läßt sich einfach mit Hilfe von Listenbeschreibungen notieren.

```
readsTree      :: (Read a) ⇒ ReadS (Tree a)
readsTree s   = [(Empty, x) | ("Empty", x) ← lex s]
                ++ [(Node l a r, y)
                   | ("(", t) ← lex s,
                     ("Node", u) ← lex t,
                     (l, v) ← readsTree u,
                     (a, w) ← reads v,
                     (r, x) ← readsTree w,
                     (")", y) ← lex x]
```

. . . Parsing—Typklasse *Read* . . .

Mit optionalen Klammern . . .

```
readsTreePrec          :: (Read a) => Int -> ReadS (Tree a)
readsTreePrec d s      = readParen False readEmpty s
                       +- readParen (d >= 10) readNode s

  where
    readEmpty s         = [(Empty, x) | ("Empty", x) <- lex s]
    readNode s          = [(Node l a r, x)
                          | ("Node", u) <- lex s,
                            (l, v) <- readsTreePrec 10 u,
                            (a, w) <- reads v,
                            (r, x) <- readsTreePrec 10 w]

readParen              :: Bool -> ReadS a -> ReadS a
readParen b g          = if b then mandatory else optional
  where optional r     = g r +- mandatory r
        mandatory r   = [(x, u) | ("(", s) <- lex r,
                                   (x, t) <- optional s,
                                   (")", u) <- lex t]
```

. . . Parsing—Typklasse *Read*

Die Typklasse *Read* ist wie folgt definiert:

class *Read* *a* **where**

readsPrec :: *Int* → *ReadS* *a*

readList :: *ReadS* [*a*]

readList = *readParen* *False* ($\lambda r \rightarrow$
[*pr* | ("*[*", *s*) ← *lex* *r*,
pr ← *readl* *s*])

where

readl *s* = [(*[]*, *t*) | ("*]*", *t*) ← *lex* *s*]
++ [(*x* : *xs*, *u*) | (*x*, *t*) ← *reads* *s*,
(*xs*, *u*) ← *readl'* *t*]

readl' *s* = [(*[]*, *t*) | ("*]*", *t*) ← *lex* *s*]
++ [(*x* : *xs*, *v*) | ("*,*", *t*) ← *lex* *s*,
(*x*, *u*) ← *reads* *t*,
(*xs*, *v*) ← *readl'* *u*]

Monaden . . .

Zu den zentralen Eigenschaften „rein“ funktionaler Sprachen gehören:

- ✘ alle Datenabhängigkeiten sind explizit: benötigt eine Funktion Informationen, muß sie diese über die Parameter erhalten,
- ✘ funktionale Sprachen arbeiten seiteneffektfrei: möchte eine Funktion Informationen „nach außen“ bekanntgeben, muß dies über den Rückgabewert erfolgen.

Beachte: Die Schnittstelle einer Funktion ist *manifest*.

Vor -und Nachteile:

- + Eine Funktion kann „lokal“ gelesen, verstanden, auf Fehler getestet und verifiziert werden.
- Erweiterungen der Funktionalität bedingen eine Änderung der Schnittstelle: mehr Parameter, komplexere Rückgabewerte.

. . . Monaden . . .

Im Vergleich dazu haben Prozeduren imperativer Sprachen sowohl explizite als auch implizite Parameter (globale Variablen). Die oben genannten Vor- und Nachteile verkehren sich.

Betrachten wir das Problem der *Erweiterbarkeit*, mit dem funktionale Programmierer kämpfen, an einem Beispiel. Ausgangspunkt: gegeben ist ein einfacher Auswerter für arithmetische Ausdrücke (2 Zeilen). Dieser soll um folgende Funktionalitäten erweitert werden:

- ✘ Fehlerbehandlung,
- ✘ einen Zähler, der die Anzahl der Reduktionen festhält,
- ✘ ein Protokoll der durchgeführten Reduktionsschritte bzw.
- ✘ eine Ausgabe der Reduktionsschritte.

. . . Monaden . . .

Repräsentation von arithmetischen Ausdrücken (im Prinzip: Binärbäume):

```
data Term    = Con Integer
              | Bin Term Op Term
              deriving (Eq, Show)
data Op      = Add | Sub | Mul | Div
              deriving (Eq, Show)
```

Ein einfacher Auswerter für arithmetische Ausdrücke:

```
sys Add      = (+)
sys Sub      = (-)
sys Mul      = (*)
sys Div      = div
eval         :: Term → Integer
eval (Con n) = n
eval (Bin t op u) = sys op (eval t) (eval u)
```

. . . Monaden . . .

Erweiterung 1: wir fügen eine Fehlerbehandlung (Division durch 0) hinzu.

```
data Exception a = Raise String
                  | Return a

eval :: Term → Exception Integer
eval (Con n) = Return n
eval (Bin t op u) = case eval t of
    Raise s → Raise s
    Return v → case eval u of
        Raise s → Raise s
        Return w →
            if (op == Div && w == 0) then
                Raise "div by zero"
            else
                Return (sys op v w)
```

Beobachtung: der Auswerter ist nicht erweitert, sondern umgeschrieben worden.

. . . Monaden . . .

Erweiterung 2: wir zählen die durchgeführten Reduktionen (vielleicht um eine Tiefenschranke zu realisieren).

```
type Count a      = Int → (a, Int)
eval              :: Term → Count Integer
eval (Con n)      = λi → (n, i)
eval (Bin t op u) = λi → let (v, j) = eval t i
                        (w, k) = eval u j
                        in (sys op v w, k + 1)
```

Beobachtung: der Auswerter ist nicht erweitert, sondern umgeschrieben worden.

. . . Monaden . . .

Erweiterung 3: wir protokollieren die durchgeführten Operationen.

```
type Trace a      = (a, String)
eval              :: Term → Trace Integer
eval e@(Con n)   = (n, trace e n)
eval e@(Bin t op u) = let (v, x) = eval t
                        (w, y) = eval u
                        r = sys op v w
                        in (r, x ++ y ++ trace e r)
trace t n        = "eval (" ++ show t ++ ") = "
                ++ show n ++ "\n"
```

Beobachtung: der Auswerter ist nicht erweitert, sondern umgeschrieben worden.

. . . Monaden . . .

Erweiterung 4: die Reduktionen werden ausgegeben.

```
eval                :: Term → IO Integer
eval e@(Con n)      = putStr (trace e n) >>
                    return n
eval e@(Bin t op u) = eval t >>= λv →
                    eval u >>= λw →
                    let r = sys op v w in
                    putStr (trace e r) >>
                    return r
trace t n           = "eval (" ++ show t ++ ") = "
                    ++ show n ++ "\n"
```

Beobachtung: der Auswerter ist nicht erweitert, sondern umgeschrieben worden.

. . . Monaden . . .

Beobachtung: Es gibt jeweils zwei getrennte „Berechnungsebenen“:

1. Vordergrund: Auswertung eines Ausdrucks,
2. Hintergrund: Fehlerbehandlung, Verwaltung eines Zustandes, Ein- und Ausgabe.

Die Ebenen haben einen unterschiedlichen Stellenwert: Die Berechnung im Vordergrund ist abhängig von der Anwendung, die Berechnung im Hintergrund ist weitgehend unabhängig davon.

Diese Trennung wird bei den EA-Aktionen deutlich: die Berechnung im Hintergrund (Kommunikation mit dem Betriebssystem) ist für den Programmierer nicht sichtbar. Er arbeitet nur mit *return*, (\gg) und den primitiven EA-Aktionen *putChar* etc.

. . . Monaden . . .

Was wir bisher verschwiegen haben: *IO* ist ein Beispiel für eine Monade ;-).

Eine Monade besteht aus

1. einem einstelligen Typkonstruktor M (wie z.B. *IO* oder *Maybe*) und
2. den Operationen

$$\begin{aligned} \text{return} &:: a \rightarrow M a \\ (\gg=) &:: M a \rightarrow (a \rightarrow M b) \rightarrow M b \end{aligned}$$

Die Operation (\gg) ist abgeleitet:

$$\begin{aligned} (\gg) &:: M a \rightarrow M b \rightarrow M b \\ m \gg n &= m \gg= \lambda_ \rightarrow n \end{aligned}$$

. . . Monaden . . .

Monaden sind als Typklassen definiert.

```
class Monad m where  
  ( $\gg=$ )    :: m a  $\rightarrow$  (a  $\rightarrow$  m b)  $\rightarrow$  m b  
  ( $\gg$ )     :: m a  $\rightarrow$  m b  $\rightarrow$  m b  
  return  :: a  $\rightarrow$  m a  
  fail    :: String  $\rightarrow$  m a  
  m  $\gg$  k  = m  $\gg=$   $\lambda$ _  $\rightarrow$  k  
  fail s  = error s
```

. . . Monaden . . .

Der einfache Auswerter in monadischer Notation:

$$\begin{aligned} eval & \quad :: \quad Term \rightarrow Id \ Integer \\ eval (Con \ n) & \quad = \quad return \ n \\ eval (Bin \ t \ op \ u) & \quad = \quad eval \ t \gg= \lambda v \rightarrow \\ & \quad \quad \quad eval \ u \gg= \lambda w \rightarrow \\ & \quad \quad \quad return (sys \ op \ v \ w) \end{aligned}$$

. . . Monaden . . .

Die Identitätsmonade führt *keine* Berechnung im Hintergrund aus.

```
newtype Id a      = Return a  
instance Monad Id where  
  return a        = Return a  
  Return a  $\gg=$  f = f a
```

. . . Monaden . . .

Erweiterung 1: wir fügen eine Fehlerbehandlung (Division durch 0) hinzu.

```
eval                :: Term → Exception Integer  
eval (Con n)      = return n  
eval (Bin t op u) = eval t >>= λv →  
                    eval u >>= λw →  
                    if (op == Div && w == 0) then  
                      raise "div by zero"  
                    else  
                      return (sys op v w)
```

Anmerkung: der Auswerter ist wirklich *erweitert* worden.

. . . Monaden . . .

Die Fehlermonade unterscheidet zwischen „normalen“ Werten und Ausnahmen.

```
data Exception a = Raise String | Return a
instance Monad Exception where
  return a      = Return a
  m >>= f      = case m of Raise s → Raise s
                  Return v → f v
```

. . . Monaden . . .

Alternativ zu den überladenen Operationen *return* und ($\gg=$) kann die **do**-Syntax verwendet werden.

```
eval                :: Term → Exception Integer  
eval (Con n)       = return n  
eval (Bin t op u) = do { v ← eval t;  
                          w ← eval u;  
                          if (op == Div && w == 0) then  
                            raise "div by zero"  
                          else  
                            return (sys op v w) }
```

Aufgabe 24 Welcher Zusammenhang besteht zwischen Listenbeschreibungen und der **do**-Syntax?

. . . Monaden . . .

Erweiterung 2: wir zählen die durchgeführten Reduktionen (vielleicht um eine Tiefenschranke zu realisieren).

$eval$	$::$	$Term \rightarrow Count\ Integer$
$eval\ (Con\ n)$	$=$	$return\ n$
$eval\ (Bin\ t\ op\ u)$	$=$	$eval\ t \gg= \lambda v \rightarrow$ $eval\ u \gg= \lambda w \rightarrow$ $incr \gg$ $return\ (sys\ op\ v\ w)$

Anmerkung: der Auswerter ist wirklich *erweitert* worden.

. . . Monaden . . .

Count ist ein Beispiel für eine Zustandsmonade.

```
newtype Count a = Count (Int → (a, Int))
apply (Count f) i = f i
instance Monad Count where
    return a = Count (λi → (a, i))
    m >>= f = Count (λi → let (a, j) = apply m i
                        in apply (f a) j)
incr      :: Count ()
incr      = Count (λi → ((), i + 1))
```

. . . Monaden . . .

Erweiterung 3: wir protokollieren die durchgeführten Operationen.

```
eval :: Term → Trace Integer
eval e@(Con n) = output (trace e n) >>
                return n
eval e@(Bin t op u) = eval t >>= λv →
                       eval u >>= λw →
                       let r = sys op v w in
                       output (trace e r) >>
                       return r
```

Anmerkung: der Auswerter ist wirklich *erweitert* worden.

. . . Monaden

Trace ist ein Beispiel für eine „Schreibermonade“.

```
newtype Trace a = Trace (a, String)
pair (Trace p)   = p
instance Monad Trace where
    return a      = Trace (a, "")
    m >>= f      = let (a, x) = pair m
                    (b, y) = pair (f a)
                    in Trace (b, x ++ y)
output           :: String → Trace ()
output s         = Trace ((), s)
```

. . . Monaden

Also: Monaden erlauben es, „imperative Effekte“ in einem rein funktionalen Programm zu erzielen. Ein „monadisches“ Programm kann leicht durch Änderung der Monade erweitert werden.

Mathematiker: Damit M , *return* und ($\gg=$) im mathematischen Sinn eine Monade bilden, müssen folgende Eigenschaften erfüllt sein.

$$\begin{aligned} \text{return } a \gg= k &= k \ a \\ m \gg= \text{return} &= m \\ m \gg= (\lambda a \rightarrow k \ a \gg= h) &= (m \gg= k) \gg= h \end{aligned}$$

Aufgabe 25 Definiere Monaden, die verschiedene imperative Effekte kombinieren: Ausgabe mit Fehlerbehandlung etc.

Aufgabe 26 Definiere Parser als eine Monade (siehe *ReadS*).