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

- [x] Ein paar Worte vorneweg
- [x] Haskell in einer Nußschale
- [x] Ausdrücke und Definitionen
- [x] Typen und Typklassen
- [x] Ein- und Ausgabe
- [x] Module und Bibliotheken
- [x] Programmiertechniken

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 (erschienen 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
- ✘ Programmiertechniken

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, werteorientiert) 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*.

$$date \quad = \quad (18, \texttt{"April"}, 1996)$$

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:

$$date \quad :: \quad (Int, String, Int)$$

. . . Struktur eines Haskell Programms

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

$$
\begin{array}{lcl}
succ & :: & Int \rightarrow Int \\
succ\ n & = & n + 1 \\
twice\ f\ a & = & f\ (f\ a)
\end{array}
$$

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*.

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

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

Aufgabe 3 Reduziere $twice\ succ\ 0$ und $twice\ twice\ succ\ 0$! **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.

$$\textbf{data}\ IntTree \quad = \quad Empty \mid Node\ IntTree\ Int\ IntTree$$

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

$$\begin{array}{lll} atree & :: & IntTree \\ atree & = & Node\ (Node\ Empty\ 2\ Empty)\ 5\ Empty \end{array}$$

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.

$$
\begin{array}{lcl}
insert & :: & Int \to IntTree \to IntTree \\
insert\ a\ Empty & = & Node\ Empty\ a\ Empty \\
insert\ a\ (Node\ l\ b\ r) & = & \textbf{if}\ a \leqslant b\ \textbf{then}\ Node\ (insert\ a\ l)\ b\ r \\
 & & \qquad\qquad \textbf{else}\ Node\ l\ b\ (insert\ a\ r)
\end{array}
$$

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

Aufgabe 4 Reduziere $insert\ 7\ atree$!

. . . Beispiel: Binärbäume . . .

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

$$
\begin{array}{lcl}
big & :: & Int \to IntTree \\
big\ n & = & Node\ (big\ (2 * n))\ n\ (big\ (2 * n + 1))
\end{array}
$$

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

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

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.

$$
\begin{array}{lcl}
delete & :: & Int \to IntTree \to IntTree \\
delete\ a\ Empty & = & Empty \\
delete\ a\ (Node\ l\ b\ r) & & \\
\quad \mid a < b & = & Node\ (delete\ a\ l)\ b\ r \\
\quad \mid a == b & = & join\ l\ r \\
\quad \mid otherwise & = & Node\ l\ b\ (delete\ a\ r)
\end{array}
$$

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.

$$
\begin{array}{lcl}
join & :: & IntTree \to IntTree \to IntTree \\
join\ Empty\ r & = & r \\
join\ (Node\ ll\ a\ lr)\ r & = & \textbf{let}\ (b, t) = split\ ll\ a\ lr\ \textbf{in}\ Node\ t\ b\ r \\
split & :: & IntTree \to Int \to IntTree \to (Int, IntTree) \\
split\ l\ a\ Empty & = & (a, l) \\
split\ l\ a\ (Node\ rl\ b\ rr) & = & \textbf{let}\ (c, t) = split\ rl\ b\ rr\ \textbf{in}\ (c, Node\ l\ a\ t)
\end{array}
$$

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*.

$$\textbf{data}\ Tree\ lab \quad = \quad Empty \mid Node\ (Tree\ lab)\ lab\ (Tree\ lab)$$

Der Typ $IntTree$ läßt sich als Abkürzung definieren.

$$\textbf{type}\ IntTree \quad = \quad Tree\ Int$$

. . . Beispiel: polymorphe Binärbäume . . .

Bezüglich dieser Typdefinition lauten die Typen der obigen Funktionen.

$$
\begin{array}{lcl}
big & :: & Int \to Tree\ Int \\
insert & :: & ?? \\
prune & :: & Int \to Tree\ lab \to Tree\ lab \\
delete & :: & ?? \\
join & :: & Tree\ lab \to Tree\ lab \to Tree\ lab \\
split & :: & Tree\ lab \to lab \to Tree\ lab \to (lab, Tree\ lab)
\end{array}
$$

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

Lies: für alle Typen lab ist $Int \to Tree\ lab \to Tree\ lab$ 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 $\leqslant$, $<$ 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.

$$
\begin{array}{lcl}
insert & :: & (Ord\ lab) \Rightarrow lab \rightarrow Tree\ lab \rightarrow Tree\ lab \\
delete & :: & (Ord\ lab) \Rightarrow lab \rightarrow Tree\ lab \rightarrow Tree\ lab
\end{array}
$$

Ord ist ein Beispiel für eine *Typklasse*. *Lies:* für alle Typen lab, auf denen eine Ordnung definiert ist, ist $lab \rightarrow Tree\ lab \rightarrow Tree\ lab$ 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.

$$
\begin{array}{lcl}
\textbf{type}\ Dict\ attr\ val & = & Tree\ (attr, val) \\
empty & :: & Dict\ a\ b \\
add & :: & (Ord\ a) \Rightarrow (a, b) \rightarrow Dict\ a\ b \rightarrow Dict\ a\ b \\
delete & :: & (Ord\ a) \Rightarrow a \rightarrow Dict\ a\ b \rightarrow Dict\ a\ b \\
lookupWithDefault & :: & (Ord\ a) \Rightarrow Dict\ a\ b \rightarrow b \rightarrow a \rightarrow b
\end{array}
$$

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.

$$\textbf{data}\ \mathit{List}\ a \quad = \quad \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

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

$$\begin{array}{lcl} [\,] & :: & [a] \\ (:) & :: & a \to [a] \to [a] \end{array}$$

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.:

$$
\begin{array}{lcl}
(+\!\!+) & :: & [a] \to [a] \to [a] \\
[\,] +\!\!+ y & = & y \\
(a : x) +\!\!+ y & = & a : (x +\!\!+ y)
\end{array}
$$

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.

$$
\begin{array}{lcl}
inorder & :: & Tree\ a \to [a] \\
build & :: & [a] \to Tree\ a
\end{array}
$$

„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.

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

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).

$$\begin{array}{lcl} putStrLn & :: & String \to IO\ () \\ putStrLn\ [\,] & = & putChar\ \texttt{'\textbackslash n'} \\ putStrLn\ (c : s) & = & putChar\ c \gg putStrLn\ s \end{array}$$

Der Operator $\gg$ korrespondiert zu dem Semikolon in Pascal.

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

. . . 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 >>= λc →
          if c == '\n' then
              return ""
          else
              getLine >>= λs →
              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.

$$
\begin{array}{lcl}
(\gg\!=) & :: & IO\ a \to (a \to IO\ b) \to IO\ b \\
return & :: & a \to IO\ a
\end{array}
$$

C-Programmierer. $act1 \gg\!= \lambda x \to act2$ 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 <- getChar
             if c == '\n' then return ""
                          else do s <- 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 compiliertem 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
- ✘ Programmiertechniken

Boolesche Ausdrücke

Die Wahrheitswerte sind vordefiniert.

$$\textbf{data}\ Bool \quad = \quad False \mid True$$

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.

$$\textbf{type}\ String \quad = \quad [Char]$$

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

Zahlen

Haskell verfügt über eine große Zahl numerischer Typen:

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

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

$$
\begin{array}{lll}
fac & :: & (Integral\ a) \Rightarrow a \rightarrow a \\
fac\ n \mid n == 0 & = & 1 \\
\quad \mid otherwise & = & n * fac\ (n-1)
\end{array}
$$

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. $(1, \texttt{'a'}, True)$) 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.

$$\textbf{data}\ [a] \quad = \quad [\,] \mid a : [a]$$

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:

$$\begin{array}{lcl}
head & :: & [a] \to a \\
tail & :: & [a] \to [a] \\
(+\!\!+) & :: & [a] \to [a] \to [a] \\
(\backslash\backslash) & :: & (Eq\ a) \Rightarrow [a] \to [a] \to [a] \\
length & :: & [a] \to Int \\
(!!) & :: & (Integral\ a) \Rightarrow [b] \to a \to b \\
take, drop & :: & (Integral\ a) \Rightarrow a \to [b] \to [a]
\end{array}$$

. . . Listen . . .

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

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*.

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

Naive Definition von Quicksort.

$$
\begin{array}{lcl}
qsort\ [\,] & = & [\,] \\
qsort\ (a : x) & = & qsort\ [b \mid b \leftarrow x, b \leqslant a] \\
 & +\!\!+ & [a] \\
 & +\!\!+ & qsort\ [b \mid b \leftarrow x, b > a]
\end{array}
$$

Mathematiker. Die Syntax von Listenbeschreibungen ist eng an die Notation von Mengen (z.B. $\{\, n \mid n \in \mathbb{N}, n \geqslant 7 \,\}$) 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.

$$
\begin{array}{lcl}
perms & :: & [a] \to [[a]] \\
perms\ [\,] & = & [[\,]] \\
perms\ x & = & [a : y' \mid (a, y) \leftarrow remove\ x, y' \leftarrow perms\ y] \\
remove & :: & [a] \to [(a, [a])] \\
remove\ [\,] & = & [\,] \\
remove\ (a : x) & = & (a, x) : [(b, a : y) \mid (b, y) \leftarrow remove\ x]
\end{array}
$$

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

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

mit

$\langle \text{qual} \rangle$	$\longrightarrow$	$\langle \text{pat} \rangle$	$\longleftarrow$ $\langle \text{exp} \rangle$	Generator
	$\mid$	$\langle \text{exp} \rangle$		Wächter

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.

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

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

Die Anwendung einer Funktion auf Argumente wird ohne Syntax notiert.

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

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

. . . Funktionen . . .

Haskell kennt *nur* einstellige Funktionen.

$$\begin{array}{lcl} addInt & :: & Int \to Int \to Int \\ addInt\ a\ b & = & a + b \end{array}$$

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

$$\begin{array}{lcl} addInt' & :: & (Int, Int) \to Int \\ addInt'\ (a, b) & = & a + b \end{array}$$

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

. . . Funktionen

Funktionen des Typs $\tau_1 \rightarrow \tau_2 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$ 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

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

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

$$\begin{array}{lcl} elem\ a\ [\,] & = & False \\ elem\ a\ (b : x) & = & a == b\ ||\ elem\ a\ x \end{array}$$

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

$$\begin{array}{lcl} elem & = & \lambda a\ y \rightarrow \textbf{case}\ y\ \textbf{of} \\ & & \quad [\,] \rightarrow False \\ & & \quad b : x \rightarrow a == b\ ||\ elem\ a\ x \end{array}$$

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.

$$
\begin{array}{llll}
f & \langle \text{pat} \rangle_{11} & \ldots \langle \text{pat} \rangle_{1k} & \langle \text{match} \rangle_1 \\
\ldots & & & \\
f & \langle \text{pat} \rangle_{n1} & \ldots \langle \text{pat} \rangle_{nk} & \langle \text{match} \rangle_n
\end{array}
$$

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

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

oder

$$
\begin{array}{lll}
\mid \langle \text{guard} \rangle_1 & = & \langle \text{exp} \rangle_1 \\
\ldots & & \\
\mid \langle \text{guard} \rangle_m & = & \langle \text{exp} \rangle_m \\
 & & \textbf{where} \ \{ \ \langle \text{decl} \rangle_1 ; \ldots ; \langle \text{decl} \rangle_n \ \}
\end{array}
$$

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.

$$
\begin{array}{lcl}
qsort & :: & (Ord\ a) \Rightarrow [a] \to [a] \\
qsort\ [\,] & = & [\,] \\
qsort\ [a] & = & [a] \\
qsort\ (a : x) & = & partition\ [\,]\ [\,]\ x \\
\quad \textbf{where} & & \\
\quad partition\ l\ r\ [\,] & = & qsort\ l +\!\!+ a : qsort\ r \\
\quad partition\ l\ r\ (b : y) & & \\
\qquad \mid b \leqslant a & = & partition\ (b : l)\ r\ y \\
\qquad \mid otherwise & = & partition\ l\ (b : r)\ y
\end{array}
$$

. . . Funktionsdefinitionen . . .

Laut Syntax werden einzelne Definitionen durch ‘;’ getrennt.

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

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.

$$head\ (a : x) \quad = \quad a$$

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 Reihenfole von Gleichungen eine Rolle.

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

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

. . . Funktionsdefinitionen

Muster haben die folgende Form (leicht vereinfacht).

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

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

Wertedefinitionen

Wertedefinitionen haben die folgende Form.

$$
\begin{array}{lllll}
\langle \mathsf{pat} \rangle & \mid & \langle \mathsf{guard} \rangle_1 & = & \langle \mathsf{exp} \rangle_1 \\
& \ldots & & & \\
& \mid & \langle \mathsf{guard} \rangle_m & = & \langle \mathsf{exp} \rangle_m \\
& & & & \textbf{where } \{\ \langle \mathsf{decl} \rangle_1 ; \ldots ; \langle \mathsf{decl} \rangle_n\ \}
\end{array}
$$

Im Gegensatz zu Funktionsdefinitionen bindet eine Wertedefinition unter Umständen mehrere Bezeichner.

$$
\begin{array}{lcl}
qsort & :: & (Ord\ a) \Rightarrow [a] \rightarrow [a] \\
qsort\ [\,] & = & [\,] \\
qsort\ (a : x) & = & qsort\ l +\!\!+ a : qsort\ r \\
\quad \textbf{where}\ (l, r) & = & partition\ (\leqslant a)\ x
\end{array}
$$

. . . Wertedefinitionen . . .

Wertedefinitionen werden häufig verwendet, um strukturierte Rückgabewerte zu verarbeiten.

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

$$
\begin{array}{lcl}
build & :: & [a] \to Tree\ a \\
build\ x & = & fst\ (buildn\ (length\ x)\ x) \\
buildn & :: & Int \to [a] \to (Tree\ a, [a]) \\
buildn\ 0\ x & = & (Empty, x) \\
buildn\ (n+1)\ x & = & (Node\ l\ a\ r, z) \\
\quad \textbf{where}\ m & = & n\ `div`\ 2 \\
\qquad\qquad (l, a : y) & = & buildn\ m\ x \\
\qquad\qquad (r, z) & = & buildn\ (n-m)\ y
\end{array}
$$

. . . Wertedefinitionen

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

$$
\begin{array}{lll}
\ldots & & \\
\textbf{where}\ m & = & n\ `div`\ 2 \\
\quad (l, a : y) & = & buildn\ m\ x \\
\quad (r, z) & = & buildn\ (n - m)\ y
\end{array}
$$

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 :: τ), 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

$$\begin{array}{l} \quad \mid \; g_1 \quad = \quad e_1 \\ \ldots \\ \quad \mid \; g_m \quad = \quad e_m \\ \qquad\qquad\quad \textbf{where} \; \{ \; \langle \text{decls} \rangle \; \} \end{array}$$

  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
- ✘ Programmiertechniken

Datentypen . . .

Neue Typen werden mittels einer Datentypdefinition eingeführt.

$$\begin{array}{lcl} \textbf{data}\ T\ \alpha_1 \ldots \alpha_k & = & C_1\ \tau_{11} \ldots \tau_{1m_1} \\ & | & \ldots \\ & | & C_n\ \tau_{n1} \ldots \tau_{nm_n} \end{array}$$

Die C_i heißen *Konstruktoren*: mit Ihrer Hilfe werden Elemente des Datentyps T konstruiert. Die α_i sind die *Typparameter* des Datentyps T.

. . . 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, $\diamondsuit$ 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.

$$
\begin{array}{lcl}
layTree & :: & Tree\ Int \to Text \\
layTree\ Empty & = & text\ \texttt{"Empty"} \\
layTree\ (Node\ l\ a\ r) & = & indent\ 4\ (text\ \texttt{"Node"} \diamondsuit nl \diamondsuit \\
 & & \quad layTree\ l \diamondsuit nl \diamondsuit \\
 & & \quad text\ (show\ a) \diamondsuit nl \diamondsuit \\
 & & \quad layTree\ r)
\end{array}
$$

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*.

```
render                  ::  Text → String
render (Text s)         =   s
render Newline          =   "\n"
render (Indent i d)     =   tab i (render d)
render (d1 :♢ d2)       =   render d1 ++ render d2

tab                     ::  Int → String → String
tab i ""                =   ""
tab i (c : s)
    | c == '\n'         =   c : replicate i ' ' ++ tab i s
    | otherwise         =   c : tab i s
```

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:

$$
\begin{array}{lcl}
text\ \texttt{""} \diamondsuit d & = & d \\
d \diamondsuit text\ \texttt{""} & = & d \\
d1 \diamondsuit (d2 \diamondsuit d3) & = & (d1 \diamondsuit d2) \diamondsuit d3 \\
indent\ 0\ d & = & d \\
indent\ i\ (text\ s) & = & text\ s \\
indent\ i\ nl & = & nl \diamondsuit text\ (replicate\ i\ \texttt{' '}) \\
indent\ i\ (indent\ j\ d) & = & indent\ (i + j)\ d \\
indent\ i\ (d1 \diamondsuit d2) & = & indent\ i\ d1 \diamondsuit indent\ i\ d2 \\
render\ (text\ s) & = & s \\
render\ nl & = & \texttt{"\textbackslash n"} \\
render\ (d1 \diamondsuit d2) & = & render\ d1 +\!\!+ render\ d2
\end{array}
$$

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

. . . Beispiel: Text mit Einrückung . . .

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

$$render'\ i\ doc\ x \quad = \quad render\ (indent\ i\ doc) +\!\!+ x$$

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 $doc = Text\ s$:

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

. . . Beispiel: Text mit Einrückung . . .

Fall $doc = Newline$:

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

. . . Beispiel: Text mit Einrückung . . .

Fall $doc = Indent\ j\ d$:

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

. . . Beispiel: Text mit Einrückung . . .

Fall $doc = d1 :\diamondsuit d2$:

$$
\begin{array}{ll}
 & render'\ i\ (d1 :\diamondsuit d2)\ x \\
= & \quad \{\ \text{Spezifikation von } render'\ \} \\
 & render\ (indent\ i\ (d1 :\diamondsuit d2)) +\!\!+ x \\
= & \quad \{\ indent\ i\ (d1 \diamondsuit d2) = indent\ i\ d1 \diamondsuit indent\ i\ d2\ \} \\
 & render\ (indent\ i\ d1 \diamondsuit indent\ i\ d2) +\!\!+ x \\
= & \quad \{\ render\ (d1 \diamondsuit d2) = render\ d1 +\!\!+ render\ d2\ \} \\
 & render\ (indent\ i\ d1) +\!\!+ render\ (indent\ i\ d2) +\!\!+ x \\
= & \quad \{\ \text{Spezifikation von } render'\ \} \\
 & render\ (indent\ i\ d1) +\!\!+ render'\ i\ d2\ x \\
= & \quad \{\ \text{Spezifikation von } render'\ \} \\
 & render'\ i\ d1\ (render'\ i\ d2\ x)
\end{array}
$$

. . . Beispiel: Text mit Einrückung . . .

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

$$
\begin{array}{lcl}
render' & :: & Int \to Text \to String \to String \\
render'\ i\ (Text\ s)\ x & = & s +\!\!+ x \\
render'\ i\ Newline\ x & = & \texttt{'\textbackslash n'} : replicate\ i\ \texttt{' '} +\!\!+ x \\
render'\ i\ (Indent\ j\ d)\ x & = & render'\ (i+j)\ d\ x \\
render'\ i\ (d1 :\diamondsuit\ d2)\ x & = & render'\ i\ d1\ (render'\ i\ d2\ x)
\end{array}
$$

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

. . . Beispiel: Text mit Einrückung

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

$$
\begin{array}{ll}
 & render\ d \\
= & \quad \{\ x +\!\!+ \text{""} = x\ \} \\
 & render\ d +\!\!+ \text{""} \\
= & \quad \{\ indent\ 0\ d = d\ \} \\
 & render\ (indent\ 0\ d) +\!\!+ \text{""} \\
= & \quad \{\ \text{Spezifikation von } render'\ \} \\
 & render'\ 0\ d\ \text{""}
\end{array}
$$

$$\boxed{render\ d \quad = \quad render'\ 0\ d\ \text{""}}$$

Typsynonyme

Mit Hilfe von Typsynonymen können Namen für Typausdrücke eingeführt werden.

$$\textbf{type}\ T\ \alpha_1 \ldots \alpha_k \quad = \quad \tau$$

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:

$(==)$**,** $(\leqslant)$ **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: $(\leqslant)$ 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.

$$\begin{array}{l}\textbf{class } Eq\ a \textbf{ where} \\ \quad (==) \quad :: \quad a \to a \to Bool \\ \quad (\not==) \quad :: \quad a \to a \to Bool\end{array}$$

Lies: der Typ a ist eine Instanz der Typklasse Ord, wenn auf a die Operationen $(==)$ und $(\not==)$ definiert sind. Die Funktionen einer Typklasse nennt man auch *Methoden*.

Typklassen schränken die Gültigkeit eines Typs ein.

$$elem \quad :: \quad (Eq\ a) \Rightarrow a \to a \to Bool$$

Lies: für alle Typen a, auf denen die Gleichheit definiert ist, ist $a \to a \to Bool$ 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 Gleichkeit 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.

$$
\begin{array}{l}
\textbf{class } Eq\ a \textbf{ where} \\
\quad (==), (\neq) \quad :: \quad a \to a \to Bool \\
\quad x \neq y \quad = \quad not\ (x == y)
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{instance } Eq\ Int \textbf{ where} \\
\quad m == n \quad = \quad intEq\ m\ n
\end{array}
$$

. . . Typklassen . . .

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

$$\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where}$$
$$(<), (\leqslant), (\geqslant), (>) \quad :: \quad a \to a \to Bool$$

Aufgabe 18 Vergleiche Haskells Typklassen mit dem Klassenkonzept objektorientierter Sprachen.

. . . Typklassen . . .

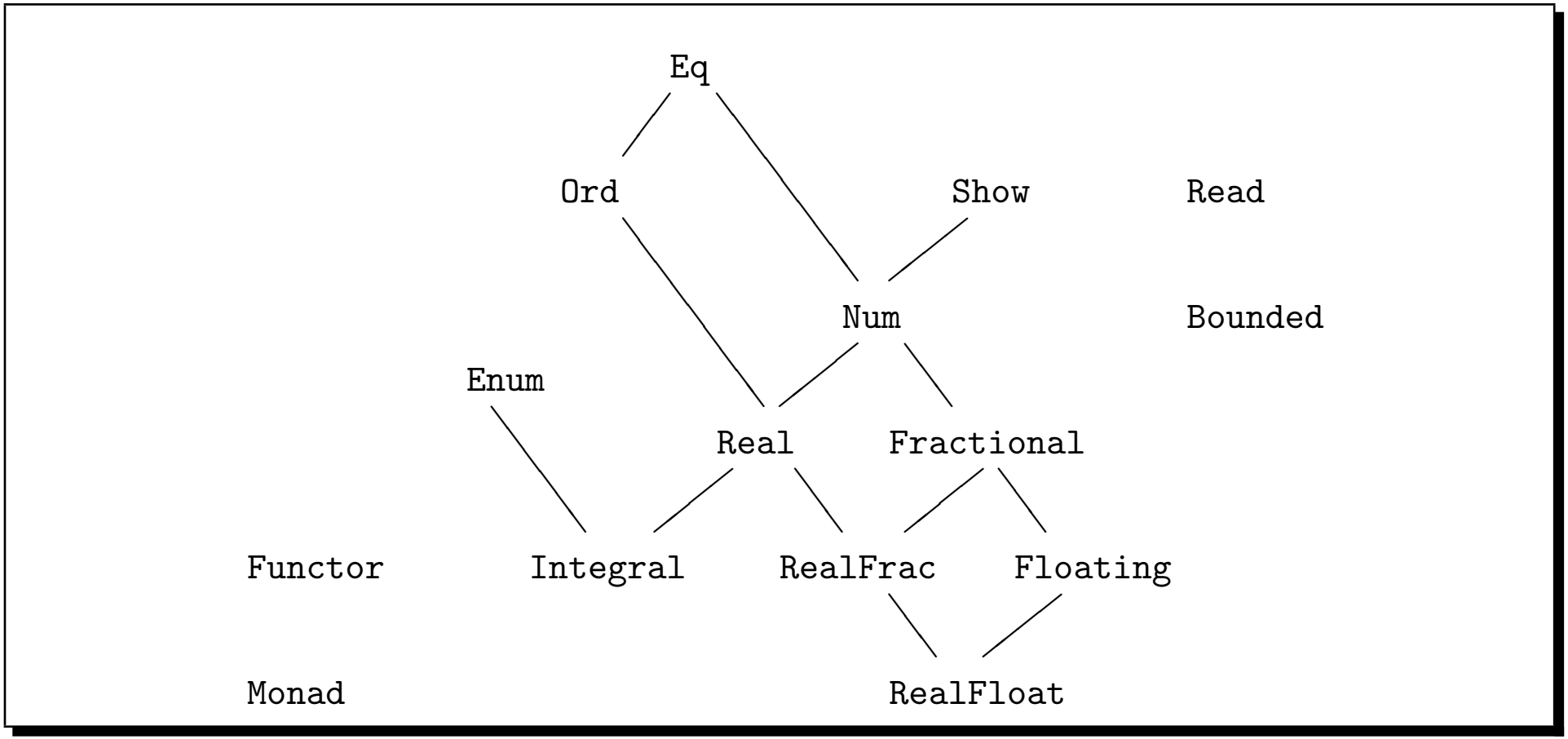
Die vollständige Deklaration der Typklasse Ord lautet.

$$
\begin{array}{lcl}
\textbf{data}\ Ordering & = & LT \mid EQ \mid GT \\
\textbf{class}\ (Eq\ a) \Rightarrow Ord\ a\ \textbf{where} & & \\
\quad compare & :: & a \to a \to Ordering \\
\quad (<), (\leqslant), (\geqslant), (>) & :: & a \to a \to Bool \\
\quad max, min & :: & a \to a \to a \\
\quad compare\ x\ y & & \\
\qquad \mid x == y & = & EQ \\
\qquad \mid x \leqslant y & = & LT \\
\qquad \mid otherwise & = & GT \\
\quad x \leqslant y & = & compare\ x\ y \neq GT \\
\quad x < y & = & compare\ x\ y == LT \\
\quad x \geqslant y & = & compare\ x\ y \neq LT \\
\quad x > y & = & compare\ x\ y == GT
\end{array}
$$

Beachte: Eine Instanz muß mindestens $\leqslant$ 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:

$$
\begin{array}{lcl}
[n\,..] & = & enumFrom\ n \qquad \text{-- Pseudocode} \\
[n, m\,..] & = & enumFromThen\ n\ m \\
[n\,..\,m] & = & enumFromTo\ n\ m \\
[n, n'\,..\,m] & = & enumFromThenTo\ n\ n'\ m
\end{array}
$$

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

. . . Typklassen . . .

$$
\begin{array}{lcl}
\textbf{class } \mathit{Enum}\ a \textbf{ where} & & \\
\quad \mathit{succ}, \mathit{pred} & :: & a \to a \\
\quad \mathit{toEnum} & :: & \mathit{Int} \to a \\
\quad \mathit{fromEnum} & :: & a \to \mathit{Int} \\
\quad \mathit{enumFrom} & :: & a \to [a] \qquad \text{-- } [n\,..] \\
\quad \mathit{enumFromThen} & :: & a \to a \to [a] \qquad \text{-- } [n, n'\,..] \\
\quad \mathit{enumFromTo} & :: & a \to a \to [a] \qquad \text{-- } [n\,..\,m] \\
\quad \mathit{enumFromThenTo} & :: & a \to a \to a \to [a] \qquad \text{-- } [n, n'\,..\,m] \\
\quad \mathit{succ} & = & \mathit{toEnum} \cdot (+1) \cdot \mathit{fromEnum} \\
\quad \mathit{pred} & = & \mathit{toEnum} \cdot (\mathit{subtract}\ 1) \cdot \mathit{fromEnum} \\
\quad \mathit{enumFrom}\ x & = & \mathit{map}\ \mathit{toEnum}\ [\mathit{fromEnum}\ x\,..] \\
\quad \mathit{enumFromTo}\ x\ y & = & \mathit{map}\ \mathit{toEnum}\ [\mathit{fromEnum}\ x\,..\,\mathit{fromEnum}\ y] \\
\quad \mathit{enumFromThen}\ x\ y & = & \mathit{map}\ \mathit{toEnum}\ [\mathit{fromEnum}\ x, \mathit{fromEnum}\ y\,..] \\
\quad \mathit{enumFromThenTo}\ x\ y\ z & & \\
\qquad & = & \mathit{map}\ \mathit{toEnum} \\
\qquad & & \quad [\mathit{fromEnum}\ x, \mathit{fromEnum}\ y\,..\,\mathit{fromEnum}\ z]
\end{array}
$$

. . . Typklassen . . .

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

$$
\begin{array}{lcl}
show & :: & (Show\ a) \Rightarrow a \rightarrow String \\
read & :: & (Read\ a) \Rightarrow String \rightarrow a
\end{array}
$$

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.

$$
\begin{array}{l}
\ldots\ readFile\ fname \gg\!= \lambda cnts \rightarrow \\
\qquad\qquad\qquad \ldots\ read\ cnts\ \ldots \\
\ldots\ writeFile\ fname\ (show\ dict)
\end{array}
$$

$show$ 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.

$$\textbf{data}\ Tree\ lab \quad = \quad Empty \mid Node\ (Tree\ lab)\ lab\ (Tree\ lab)\ \textbf{deriving}\ (Eq, Ord, Show, Read)$$

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?

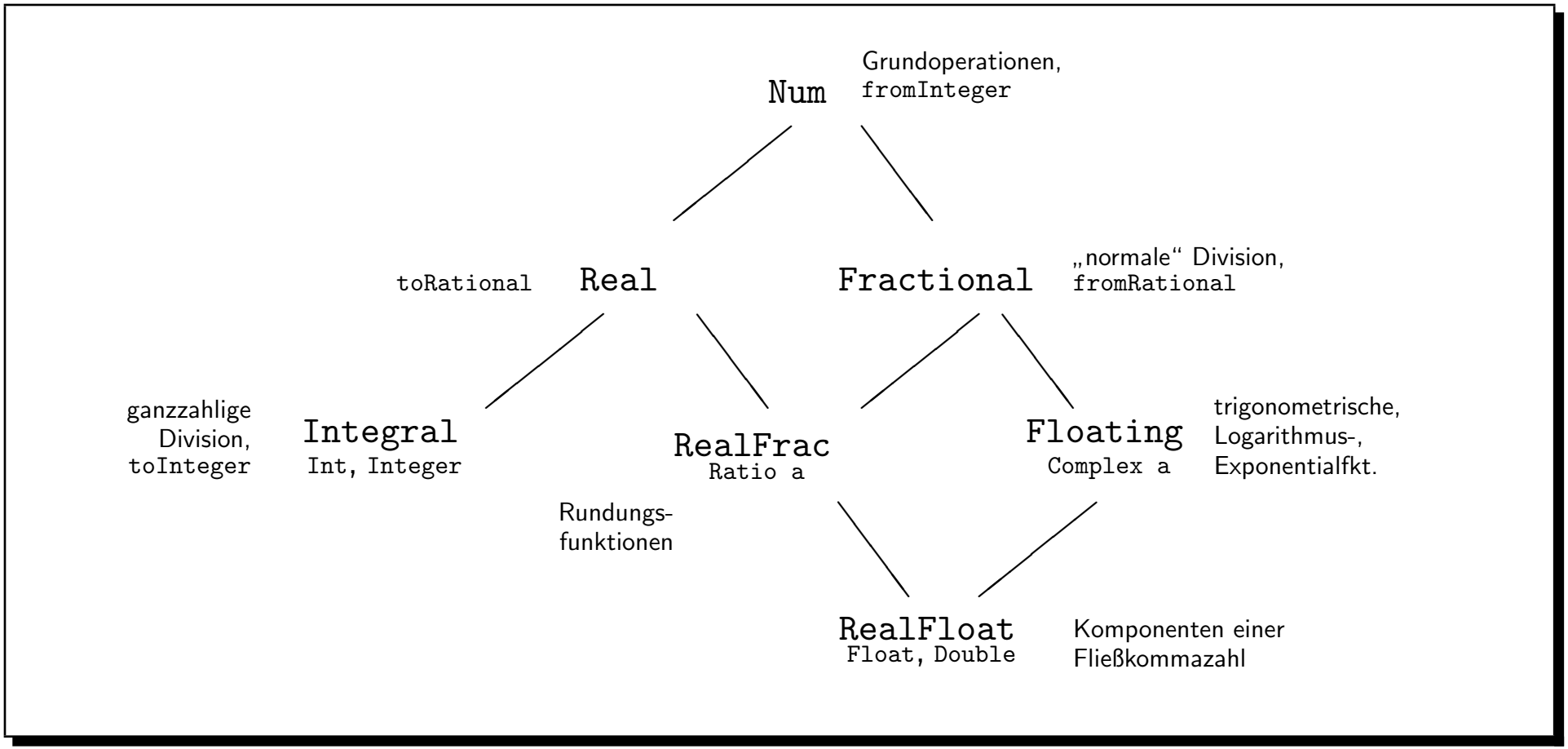
- Mit jedem der Typpaare $Int/Integer$, $Float/Double$, $Ratio\ Int/Ratio\ Integer$, $Complex\ Float/Complex\ Double$ korrespondiert eine Klasse: $Integral$, $RealFrac$, $RealFloat$ und $Floating$.

- 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 Klassenhierachie: Auf komplexen Zahlen läßt sich keine Ordnung definieren; alle Zahlen lassen sich auf Gleichkeit 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.

$$
\begin{array}{lcl}
\textbf{class}\ (Eq\ a, Show\ a) \Rightarrow Num\ a\ \textbf{where} & & \\
\quad (+), (-), (*) & :: & a \to a \to a \\
\quad negate & :: & a \to a \\
\quad abs, signum & :: & a \to a \\
\quad fromInteger & :: & Integer \to a \\
\quad x - y & = & x + negate\ y \\
\quad negate\ x & = & 0 - x
\end{array}
$$

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).

$$\begin{array}{l} \textbf{data}\ (Integral\ a) \Rightarrow Ratio\ a \\ \textbf{type}\ Rational\ \ =\ \ Ratio\ Integer \end{array}$$

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

$$\begin{array}{lcl} (\%) & :: & (Integral\ a) \Rightarrow a \rightarrow a \rightarrow Ratio\ a \\ numerator & :: & (Integral\ a) \Rightarrow Ratio\ a \rightarrow a \\ denominator & :: & (Integral\ a) \Rightarrow Ratio\ a \rightarrow a \end{array}$$

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 $fromRational\ (3.1415 :: Rational)$.

$$\begin{array}{lcl} fromInteger & :: & (Num\ a) \Rightarrow Integer \rightarrow a \\ fromRational & :: & (Fractional\ a) \Rightarrow Rational \rightarrow a \end{array}$$

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.

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

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

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

. . . Typklassen . . .

Überladene Funktionen lassen sich nicht immer eindeutig auflösen.

$$\begin{array}{lcl} pyth & :: & (Floating\ a) \Rightarrow a \to a \to a \\ pyth\ x\ y & = & sqrt\ (x\ \hat{}\ 2 + y\ \hat{}\ 2) \end{array}$$

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

$$\textbf{default}\ (Int, Double)$$

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

. . . Typklassen

Das Problem tritt nicht nur bei numerischen Funktionen auf.

$$\begin{array}{lcl} whoops & :: & String \to String \\ whoops\ x & = & show\ (read\ x) \end{array}$$

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

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.

$$\begin{array}{lcl} whoops & :: & String \to String \\ whoops\ x & = & show\ (read\ x :: [Int]) \end{array}$$

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) ⇒ Lay a where
    lay     ::  a → 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.

$$\textbf{data}\ IO\ val$$

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

Vordefinierte Ausgabefunktionen:

$$\begin{array}{lcl} putChar & :: & Char \to IO\ () \\ putStr & :: & String \to IO\ () \\ putStrLn & :: & String \to IO\ () \\ print & :: & (Show\ a) \Rightarrow a \to IO\ () \end{array}$$

. . . Vordefinierte EA-Aktionen . . .

Vordefinierte Eingabefunktionen:

$$
\begin{array}{lcl}
getChar & :: & IO\ Char \\
getLine & :: & IO\ String \\
readLine & :: & (Read\ a) \Rightarrow IO\ a
\end{array}
$$

Vordefinierte Operationen auf Dateien.

$$
\begin{array}{lcl}
\textbf{type}\ FilePath & = & String \\
readFile & :: & FilePath \rightarrow IO\ String \\
writeFile & :: & FilePath \rightarrow String \rightarrow IO\ () \\
appendFile & :: & FilePath \rightarrow String \rightarrow IO\ ()
\end{array}
$$

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

. . . Vordefinierte EA-Aktionen . . .

Verknüpfung von EA-Aktionen:

$$
\begin{array}{lcl}
return & :: & a \to IO\ a \\
(\gg) & :: & IO\ a \to IO\ b \to IO\ b \\
(\gg\!=) & :: & IO\ a \to (a \to IO\ b) \to IO\ b
\end{array}
$$

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

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

$$
main \quad = \quad print\ [(i, i \hat{}\ 2) \mid i \leftarrow [0\,..\,9]]
$$

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.

$$
\begin{array}{lcl}
askFor & :: & String \to IO\ String \\
askFor\ s & = & putStr\ s \gg getLine \\
main & :: & IO\ () \\
main & = & askFor\ \texttt{"filename: "} \gg\!= \lambda file \to \\
 & & readFile\ file \gg\!= \lambda cnts \to \\
 & & putStr\ cnts
\end{array}
$$

do-Notation . . .

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

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

mit

$$\begin{array}{lcll} \langle \text{stat} \rangle & \rightarrow & \langle \text{exp} \rangle & \\ & | & \langle \text{pat} \rangle \leftarrow \langle \text{exp} \rangle & \text{Generator} \\ & | & \textbf{let}\ \{\ \langle \text{decl} \rangle_1; \ldots; \langle \text{decl} \rangle_n\ \} & \text{lokale Definition} \end{array}$$

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 $\gg=$ übersetzt werden (leicht vereinfacht).

$$
\begin{array}{lcl}
\textbf{do}\,\{\,e\,\} & = & e \\
\textbf{do}\,\{\,e; \mathit{stmts}\,\} & = & e \gg \textbf{do}\,\{\,\mathit{stmts}\,\} \\
\textbf{do}\,\{\,p \leftarrow e; \mathit{stmts}\,\} & = & e \gg= \lambda p \rightarrow \textbf{do}\,\{\,\mathit{stmts}\,\} \\
\textbf{do}\,\{\,\textbf{let}\ \mathit{decls}; \mathit{stmts}\,\} & = & \textbf{let}\ \mathit{decls}\ \textbf{in do}\,\{\,\mathit{stmts}\,\}
\end{array}
$$

. . . 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.

$$
\begin{array}{lcl}
sequence\_ & :: & [IO\ a] \to IO\ () \\
sequence\_\ [\,] & = & return\ () \\
sequence\_\ (a : as) & = & a \gg sequence\_\ as \\
sequence & :: & [IO\ a] \to IO\ [a] \\
sequence\ [\,] & = & return\ [\,] \\
sequence\ (a : as) & = & a \gg\!= \lambda v \to \\
 & & sequence\ as \gg\!= \lambda vs \to \\
 & & return\ (v : vs)
\end{array}
$$

$sequence\_$ und $sequence$ werden gerne mit Listenbeschreibungen kombiniert.

$$
\begin{array}{lcl}
askForMany & :: & [String] \to IO\ [String] \\
askForMany\ xs & = & sequence\ [askFor\ s \mid s \leftarrow xs]
\end{array}
$$

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.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{IOError} & & \\
\mathit{userError} & :: & \mathit{String} \to \mathit{IOError} \\
\mathit{ioError} & :: & \mathit{IOError} \to \mathit{IO}\ a \\
\mathit{catch} & :: & \mathit{IO}\ a \to (\mathit{IOError} \to \mathit{IO}\ a) \to \mathit{IO}\ a \\
\mathit{fail} & :: & \mathit{String} \to \mathit{IO}\ a \\
\mathit{fail}\ s & = & \mathit{ioError}\ (\mathit{userError}\ s)
\end{array}
$$

. . . Fehlerbehandlung . . .

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

$$
\begin{array}{lcl}
isAlreadyExistsError & :: & IOError \to Bool \\
isDoesNotExistError & :: & IOError \to Bool \\
isAlreadyInUseError & :: & IOError \to Bool \\
isFullError & :: & IOError \to Bool \\
isEOFError & :: & IOError \to Bool \\
isIllegalOperation & :: & IOError \to Bool \\
isPermissionError & :: & IOError \to Bool \\
isUserError & :: & IOError \to Bool
\end{array}
$$

. . . Fehlerbehandlung . . .

Abgeleitete Operationen:

$$
\begin{array}{lcl}
try & :: & IO\ a \to IO\ (Either\ IOError\ a) \\
try\ p & = & (p \gg\!= (return \cdot Right)) \\
 & & \quad `catch`(return \cdot Left) \\
either & :: & (a \to c) \to (b \to c) \to (Either\ a\ b) \to c \\
either\ f\ g\ (Left\ x) & = & f\ x \\
either\ f\ g\ (Right\ x) & = & g\ x
\end{array}
$$

Typische Anwendung von try und $either$:

$$try\ a \gg\!= either\ (\lambda err \to \ldots)\ (\lambda ok \to \ldots)$$

. . . 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:

$$
\begin{array}{lcl}
main & = & getArgs \gg\!= echo \\
echo\ [\,] & = & return\ () \\
echo\ (\texttt{"-n"} : x) & = & putStr\ (unwords\ x) \\
echo\ x & = & putStrLn\ (unwords\ x)
\end{array}
$$

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
- ✘ Programmiertechniken

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⟩ ($\langle \text{export} \rangle_1, \ldots, \langle \text{export} \rangle_k$)
where {
    $\langle \text{impdecl} \rangle_1; \ldots; \langle \text{impdecl} \rangle_m;$
    $\langle \text{topdecl} \rangle_1; \ldots; \langle \text{topdecl} \rangle_n$
}

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 $T(C_1, \ldots, C_n)$: die aufgeführten Konstruktoren werden exportiert,
oder $T(\ldots)$: Abkürzung für alle Konstruktoren,
oder T: die Konstruktoren werden *nicht* exportiert,

Typsynonyme in der Form T

Typklassen in der Form $C(f_1, \ldots, f_n)$: einige Methoden werden aufgeführt,
oder $C(\ldots)$: 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** $(Eq\ a) \Rightarrow Eq\ (Tree\ a)$ etc. werden automatisch exportiert.

Import . . .

Importdeklarationen können zwei verschiedene Formen haben.

⟨impdecl⟩	→	**import** ⟨modid⟩ [⟨impspec⟩]	
⟨impspec⟩	→	(⟨import⟩$_1$, . . . , ⟨import⟩$_n$)	positiv
	|	*hiding* (⟨import⟩$_1$, . . . , ⟨import⟩$_n$)	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.

```
r           = 5
volume_of_r = 3 / 4 * pi * r ^ 3
```

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

```
volume r    = 3 / 4 * pi * r ^ 3
volume_of_r = volume r
```

. . . Funktionen höherer Ordnung . . .

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

$$
\begin{array}{lcl}
qsortBy & :: & (a \to a \to Bool) \to [a] \to [a] \\
qsortBy\ (\leqslant)\ [\,] & = & [\,] \\
qsortBy\ (\leqslant)\ (a : x) & = & qsortBy\ (\leqslant)\ l +\!\!+ a : qsortBy\ (\leqslant)\ r \\
\quad \textbf{where}\ (l, r) & = & partition\ (\leqslant a)\ x \\
qsort & :: & (Ord\ a) \Rightarrow [a] \to [a] \\
qsort & = & qsortBy\ (\leqslant)
\end{array}
$$

$qsortBy$ ist eine sogenannte Funktion höherer Ordnung. [$f :: \tau_1 \to \tau_2$ 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.

$$
\begin{array}{lcl}
inclist\ [\,] & = & [\,] \\
inclist\ (a : x) & = & a + 1 : inclist\ x \\
strupr\ [\,] & = & [\,] \\
strupr\ (a : x) & = & toUpper\ a : strupr\ x
\end{array}
$$

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

$$[x_1, x_2, x_3, \ldots, x_{n-2}, x_{n-1}, x_n]$$

$$\downarrow \quad map\ f$$

$$[f\ x_1, f\ x_2, f\ x_3, \ldots, f\ x_{n-2}, f\ x_{n-1}, f\ x_n]$$

. . . Funktionen höherer Ordnung . . .

Realisierung der Funktion map:

$$
\begin{array}{lcl}
map & :: & (a \to b) \to [a] \to [b] \\
map\ f\ [\,] & = & [\,] \\
map\ f\ (a : x) & = & f\ a : map\ f\ x
\end{array}
$$

Spezialisierung der map-Funktion:

$$
\begin{array}{lcl}
inclist & = & map\ (+1) \\
strupr & = & map\ toUpper
\end{array}
$$

Beachte: map läßt sich einfacher mit Hilfe von Listenbeschreibungen definieren.

$$
map\ f\ x \quad = \quad [f\ a \mid a \leftarrow x]
$$

. . . Funktionen höherer Ordnung . . .

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

$$
\begin{array}{lcl}
mapMaybe & :: & (a \to b) \to Maybe\ a \to Maybe\ b \\
mapMaybe\ f\ Nothing & = & Nothing \\
mapMaybe\ f\ (Just\ a) & = & Just\ (f\ a) \\
\textbf{data}\ Tree\ lab & = & Node\ lab\ [Tree\ lab] \\
mapTree & :: & (a \to b) \to Tree\ a \to Tree\ b \\
mapTree\ f\ (Node\ a\ ts) & = & Node\ (f\ a)\ (map\ (mapTree\ f)\ ts)
\end{array}
$$

Mathematiker: Sei T ein einstelliger Typkonstruktor, dann hat die zugehörige „map“-Funktion den Typ $(\alpha \to \beta) \to (T\ \alpha \to T\ \beta)$. 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.

$$
\begin{array}{lcl}
sum\ [\,] & = & 0 \\
sum\ (a : x) & = & a + sum\ x \\
sequence\_\ [\,] & = & return\ () \\
sequence\_\ (a : as) & = & a \gg sequence\_\ as
\end{array}
$$

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

$$x_1 : x_2 : x_3 : \cdots : x_{n-2} : x_{n-1} : x_n : [\,]$$

$$\Big\downarrow\ foldr\ (\otimes)\ e$$

$$x_1 \otimes (x_2 \otimes (x_3 \otimes \cdots \otimes (x_{n-2} \otimes (x_{n-1} \otimes (x_n \otimes e)))))$$

. . . Funktionen höherer Ordnung . . .

Realisierung der Funktion $foldr$:

$$
\begin{array}{lcl}
foldr & :: & (a \to b \to b) \to b \to [a] \to b \\
foldr\ (*)\ e\ [\,] & = & e \\
foldr\ (*)\ e\ (a : x) & = & a * foldr\ (*)\ e\ x
\end{array}
$$

Spezialisierung der $foldr$-Funktion:

$$
\begin{array}{lcl}
sum & = & foldr\ (+)\ 0 \\
sequence\_ & = & foldr\ (\gg)\ (return\ ())
\end{array}
$$

Aufgabe 21 Definiere $product$, or, $concat$, $isort$ (Sortieren durch Einfügen) und $sequence$ mit Hilfe von $foldr$!

. . . Funktionen höherer Ordnung

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

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Tree}\ \mathit{lab} & = & \mathit{Node}\ \mathit{lab}\ [\mathit{Tree}\ \mathit{lab}] \\
\mathit{fold} & :: & (a \to [b] \to b) \to \mathit{Tree}\ a \to b \\
\mathit{fold}\ f\ (\mathit{Node}\ a\ ts) & = & f\ a\ (\mathit{map}\ (\mathit{fold}\ f)\ ts)
\end{array}
$$

Spezialisierung von *fold*:

$$
\begin{array}{lcl}
\mathit{sumup} & = & \mathit{fold}\ (\lambda a\ ns \to a + \mathit{sum}\ ns) \\
\mathit{size} & = & \mathit{fold}\ (\lambda \_\ ns \to 1 + \mathit{sum}\ ns) \\
\mathit{height} & = & \mathit{fold}\ (\lambda \_\ ns \to 1 + \mathit{maximum}\ ns) \\
\mathit{preorder} & = & \mathit{fold}\ (\lambda a\ xs \to a : \mathit{concat}\ xs)
\end{array}
$$

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.

$$
\begin{array}{lcl}
iterate & :: & (a \to a) \to a \to [a] \\
takeWhile & :: & (a \to Bool) \to [a] \to [a]
\end{array}
$$

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

$$
\begin{array}{lcl}
group\ n & = & iterate\ (drop\ n) \\
 & \# & takeWhile\ (not \cdot null) \\
 & \# & map\ (take\ n) \\
f \# g & = & g \cdot f
\end{array}
$$

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.

$$
\begin{array}{lcl}
\textbf{data}\ Line & = & Blank\ String \\
 & \mid & Line\ String \\
classify & :: & String \to Line \\
classify\ s \mid all\ isSpace\ s & = & Blank\ s \\
\quad \mid otherwise & = & Line\ s \\
unclassify & :: & Line \to String \\
unclassify\ (Blank\ s) & = & s \\
unclassify\ (Line\ s) & = & s \\
squeeze & :: & [Line] \to [Line] \\
squeeze\ [\,] & = & [\,] \\
squeeze\ (Blank\ \_ : x@(Blank\ \_ : \_)) & = & squeeze\ x \\
squeeze\ (a : x) & = & a : squeeze\ x
\end{array}
$$

. . . Lazy evaluation . . .

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

$$
\begin{array}{lcl}
number & :: & Int \to [String] \to [String] \\
number\ n & = & zip\ [n\,..]\ \#\ map\ lineno \\
bnumber & :: & Int \to [Line] \to [Line] \\
bnumber\ n\ [\,] & = & [\,] \\
bnumber\ n\ (Blank\ s : x) & = & Blank\ (indent\ s) : bnumber\ n\ x \\
bnumber\ n\ (Line\ s : x) & = & Line\ (lineno\ (n, s)) : bnumber\ (n+1)\ x \\
indent & :: & String \to String \\
indent\ s & = & replicate\ 8\ \text{'}\ \text{'} +\!\!+ s \\
lineno & :: & (Int, String) \to String \\
lineno\ (n, s) & = & rjustify\ 6\ (show\ n) +\!\!+ \text{"}\quad\ \text{"} +\!\!+ s
\end{array}
$$

. . . 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.

$$
\begin{array}{lcl}
lefty & :: & Int \to Tree\ Int \\
lefty\ 0 & = & Empty \\
lefty\ (n+1) & = & Node\ (lefty\ n)\ n\ Empty
\end{array}
$$

. . . 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:*

$$showsTree\ t\ x \quad = \quad showTree\ t +\!\!+ x \qquad \text{-- Spezifikation}$$

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

$$\begin{array}{lcl}
showsTree & :: & (Show\ a) \Rightarrow Tree\ a \rightarrow String \rightarrow String \\
showsTree\ Empty\ x & = & \texttt{"Empty"} +\!\!+ x \\
showsTree\ (Node\ l\ a\ r)\ x & = & \texttt{"(Node "} \\
 & +\!\!+ & showsTree\ l\ (\texttt{" "} \\
 & +\!\!+ & shows\ a\ (\texttt{" "} \\
 & +\!\!+ & showsTree\ r\ (\texttt{")"} +\!\!+ x))) \\
showTree & :: & (Show\ a) \Rightarrow Tree\ a \rightarrow String \\
showTree\ t & = & showsTree\ t\ \texttt{""}
\end{array}$$

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

showsTree läßt sich noch eleganter notieren.

$$
\begin{array}{lcl}
\textbf{type}\ ShowS & = & String \to String \\
showsTree & :: & (Show\ a) \Rightarrow Tree\ a \to ShowS \\
showsTree\ Empty & = & showString\ \texttt{"Empty"} \\
showsTree\ (Node\ l\ a\ r) & = & showString\ \texttt{"(Node "} \\
 & \cdot & showsTree\ l \cdot showChar\ \texttt{' '} \\
 & \cdot & shows\ a \cdot showChar\ \texttt{' '} \\
 & \cdot & showsTree\ r \cdot showChar\ \texttt{')'} \\
showChar & = & (:) \\
showString & = & (+\!\!+)
\end{array}
$$

C-Programmierer. Lies *showfoo* als *printf* und '$\cdot$' als ';'.

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

Einige Klammern in der Ausgabe können eingespart werden . . .

$$
\begin{array}{lcl}
showTree & :: & (Show\ a) \Rightarrow Tree\ a \to String \\
showTree\ t & = & showsTreePrec\ 0\ t\ \texttt{""} \\
showsTreePrec & :: & (Show\ a) \Rightarrow Int \to Tree\ a \to ShowS \\
showsTreePrec\ d\ Empty & = & showString\ \texttt{"Empty"} \\
showsTreePrec\ d\ (Node\ l\ a\ r) & = & showParen\ (d \geqslant 10)\ (\\
 & & showString\ \texttt{"Node "} \\
 & \cdot & showsTreePrec\ 10\ l \cdot showChar\ \texttt{' '} \\
 & \cdot & showsPrec\ 10\ a \cdot showChar\ \texttt{' '} \\
 & \cdot & showsTreePrec\ 10\ r) \\
showParen & :: & Bool \to ShowS \to ShowS \\
showParen\ False\ p & = & p \\
showParen\ True\ p & = & showChar\ \texttt{'('} \cdot p \cdot showChar\ \texttt{')'}
\end{array}
$$

. . . 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.

$$\textbf{type}\ ReadS\ a \quad = \quad String \to [(a, String)]$$

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.

$$(reads :: ReadS\ Int)\ \texttt{"5 hamburger"} \quad \Rightarrow \quad [(5, \texttt{" hamburger"})]$$

Ein Parser für die lexikalische Analyse ist vordefiniert.

$$lex \quad :: \quad ReadS\ String$$

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

. . . Parsing—Typklasse $Read$ . . .

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

$$
\begin{array}{lcl}
readsTree & :: & (Read\ a) \Rightarrow ReadS\ (Tree\ a) \\
readsTree\ s & = & [(Empty, x) \mid (\texttt{"Empty"}, x) \leftarrow lex\ s] \\
 & +\!\!+ & [(Node\ l\ a\ r, y) \\
 & & \mid (\texttt{"("}, t) \leftarrow lex\ s, \\
 & & \ (\texttt{"Node"}, u) \leftarrow lex\ t, \\
 & & \ (l, v) \leftarrow readsTree\ u, \\
 & & \ (a, w) \leftarrow reads\ v, \\
 & & \ (r, x) \leftarrow readsTree\ w, \\
 & & \ (\texttt{")"}, y) \leftarrow lex\ x]
\end{array}
$$

. . . 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 (λr →
                      [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):

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Term} & = & \mathit{Con}\ \mathit{Integer} \\
& | & \mathit{Bin}\ \mathit{Term}\ \mathit{Op}\ \mathit{Term} \\
& & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\
\textbf{data}\ \mathit{Op} & = & \mathit{Add} \mid \mathit{Sub} \mid \mathit{Mul} \mid \mathit{Div} \\
& & \textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{array}
$$

Ein einfacher Auswerter für arithmetische Ausdrücke:

$$
\begin{array}{lcl}
\mathit{sys}\ \mathit{Add} & = & (+) \\
\mathit{sys}\ \mathit{Sub} & = & (-) \\
\mathit{sys}\ \mathit{Mul} & = & (*) \\
\mathit{sys}\ \mathit{Div} & = & \mathit{div} \\
\mathit{eval} & :: & \mathit{Term} \to \mathit{Integer} \\
\mathit{eval}\ (\mathit{Con}\ n) & = & n \\
\mathit{eval}\ (\mathit{Bin}\ t\ \mathit{op}\ u) & = & \mathit{sys}\ \mathit{op}\ (\mathit{eval}\ t)\ (\mathit{eval}\ u)
\end{array}
$$

. . . 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).

$$
\begin{array}{lcl}
\textbf{type}\ Count\ a & = & Int \to (a, Int) \\
eval & :: & Term \to Count\ Integer \\
eval\ (Con\ n) & = & \lambda i \to (n, i) \\
eval\ (Bin\ t\ op\ u) & = & \lambda i \to \textbf{let}\ (v, j) = eval\ t\ i \\
 & & \qquad\qquad (w, k) = eval\ u\ j \\
 & & \quad \textbf{in}\ (sys\ op\ v\ w, k + 1)
\end{array}
$$

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

. . . Monaden . . .

Erweiterung 3: wir protokollieren die durchgeführten Operationen.

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Trace}\ a & = & (a, \mathit{String}) \\
\mathit{eval} & :: & \mathit{Term} \to \mathit{Trace}\ \mathit{Integer} \\
\mathit{eval}\ e@(\mathit{Con}\ n) & = & (n, \mathit{trace}\ e\ n) \\
\mathit{eval}\ e@(\mathit{Bin}\ t\ \mathit{op}\ u) & = & \textbf{let}\ (v, x) = \mathit{eval}\ t \\
 & & \quad (w, y) = \mathit{eval}\ u \\
 & & \quad r = \mathit{sys}\ \mathit{op}\ v\ w \\
 & & \textbf{in}\ (r, x +\!\!+ y +\!\!+ \mathit{trace}\ e\ r) \\
\mathit{trace}\ t\ n & = & \texttt{"eval ("} +\!\!+ \mathit{show}\ t +\!\!+ \texttt{") = "} \\
 & +\!\!+ & \mathit{show}\ n +\!\!+ \texttt{"\textbackslash n"}
\end{array}
$$

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

. . . Monaden . . .

Erweiterung 4: die Reduktionen werden ausgegeben.

$$
\begin{array}{lcl}
eval & :: & Term \to IO\ Integer \\
eval\ e@(Con\ n) & = & putStr\ (trace\ e\ n) \gg \\
 & & return\ n \\
eval\ e@(Bin\ t\ op\ u) & = & eval\ t \gg\!= \lambda v \to \\
 & & eval\ u \gg\!= \lambda w \to \\
 & & \textbf{let}\ r = sys\ op\ v\ w\ \textbf{in} \\
 & & putStr\ (trace\ e\ r) \gg \\
 & & return\ r \\
trace\ t\ n & = & \texttt{"eval ("} +\!\!+ show\ t +\!\!+ \texttt{") = "} \\
 & +\!\!+ & show\ n +\!\!+ \texttt{"\textbackslash n"}
\end{array}
$$

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

$$\begin{array}{lcl} return & :: & a \to M\ a \\ (\gg\!=) & :: & M\ a \to (a \to M\ b) \to M\ b \end{array}$$

Die Operation $(\gg)$ ist abgeleitet:

$$\begin{array}{lcl} (\gg) & :: & M\ a \to M\ b \to M\ b \\ m \gg n & = & m \gg\!= \lambda\_ \to n \end{array}$$

. . . Monaden . . .

Monaden sind als Typklassen definiert.

$$
\begin{array}{lcl}
\textbf{class } \mathit{Monad}\ m \textbf{ where} & & \\
\quad (\gg\!=) & :: & m\ a \to (a \to m\ b) \to m\ b \\
\quad (\gg) & :: & m\ a \to m\ b \to m\ b \\
\quad \mathit{return} & :: & a \to m\ a \\
\quad \mathit{fail} & :: & \mathit{String} \to m\ a \\
\quad m \gg k & = & m \gg\!= \lambda\_ \to k \\
\quad \mathit{fail}\ s & = & \mathit{error}\ s
\end{array}
$$

. . . Monaden . . .

Der einfache Auswerter in monadischer Notation:

$$
\begin{array}{lcl}
eval & :: & Term \to Id\ Integer \\
eval\ (Con\ n) & = & return\ n \\
eval\ (Bin\ t\ op\ u) & = & eval\ t \gg\!= \lambda v \to \\
 & & eval\ u \gg\!= \lambda w \to \\
 & & return\ (sys\ op\ v\ w)
\end{array}
$$

. . . Monaden . . .

Die Identitätsmonade führt *keine* Berechnung im Hintergrund aus.

$$
\begin{array}{lcl}
\textbf{newtype}\ Id\ a & = & Return\ a \\
\textbf{instance}\ Monad\ Id\ \textbf{where} & & \\
\quad return\ a & = & Return\ a \\
\quad Return\ a \gg\!= f & = & f\ a
\end{array}
$$

. . . Monaden . . .

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

$$
\begin{array}{lcl}
eval & :: & Term \to Exception\ Integer \\
eval\ (Con\ n) & = & return\ n \\
eval\ (Bin\ t\ op\ u) & = & eval\ t \gg\!= \lambda v \to \\
 & & eval\ u \gg\!= \lambda w \to \\
 & & \textbf{if}\ (op == Div\ \&\&\ w == 0)\ \textbf{then} \\
 & & \quad raise\ \texttt{"div by zero"} \\
 & & \textbf{else} \\
 & & \quad return\ (sys\ op\ v\ w)
\end{array}
$$

Anmerkung: der Auswerter ist wirklich *erweitert* worden.

. . . Monaden . . .

Die Fehlermonade unterscheidet zwischen „normalen“ Werten und Ausnahmen.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Exception}\ a & = & \mathit{Raise}\ \mathit{String} \mid \mathit{Return}\ a \\
\textbf{instance}\ \mathit{Monad}\ \mathit{Exception}\ \textbf{where} & & \\
\quad \mathit{return}\ a & = & \mathit{Return}\ a \\
\quad m \gg\!= f & = & \textbf{case}\ m\ \textbf{of}\ \mathit{Raise}\ s \rightarrow \mathit{Raise}\ s \\
 & & \qquad\qquad\quad \mathit{Return}\ v \rightarrow f\ v
\end{array}
$$

. . . 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).

$$
\begin{array}{lcl}
eval & :: & Term \to Count\ Integer \\
eval\ (Con\ n) & = & return\ n \\
eval\ (Bin\ t\ op\ u) & = & eval\ t \gg\!= \lambda v \to \\
 & & eval\ u \gg\!= \lambda w \to \\
 & & incr \gg\! \\
 & & return\ (sys\ op\ v\ w)
\end{array}
$$

Anmerkung: der Auswerter ist wirklich *erweitert* worden.

. . . Monaden . . .

Count ist ein Beispiel für eine Zustandsmonade.

$$
\begin{array}{lcl}
\textbf{newtype}\ Count\ a & = & Count\ (Int \to (a, Int)) \\
apply\ (Count\ f)\ i & = & f\ i \\
\textbf{instance}\ Monad\ Count\ \textbf{where} & & \\
\quad return\ a & = & Count\ (\lambda i \to (a, i)) \\
\quad m \gg\!= f & = & Count\ (\lambda i \to \textbf{let}\ (a, j) = apply\ m\ i \\
 & & \qquad\qquad \textbf{in}\ apply\ (f\ a)\ j) \\
incr & :: & Count\ () \\
incr & = & Count\ (\lambda i \to ((), i + 1))
\end{array}
$$

. . . Monaden . . .

Erweiterung 3: wir protokollieren die durchgeführten Operationen.

$$
\begin{array}{lcl}
eval & :: & Term \to Trace\ Integer \\
eval\ e@(Con\ n) & = & output\ (trace\ e\ n) \gg \\
 & & return\ n \\
eval\ e@(Bin\ t\ op\ u) & = & eval\ t \gg\!= \lambda v \to \\
 & & eval\ u \gg\!= \lambda w \to \\
 & & \textbf{let}\ r = sys\ op\ v\ w\ \textbf{in} \\
 & & output\ (trace\ e\ r) \gg \\
 & & return\ r
\end{array}
$$

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.

$$
\begin{array}{lcl}
return\ a \gg\!= k & = & k\ a \\
m \gg\!= return & = & m \\
m \gg\!= (\lambda a \to k\ a \gg\!= h) & = & (m \gg\!= k) \gg\!= h
\end{array}
$$

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

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