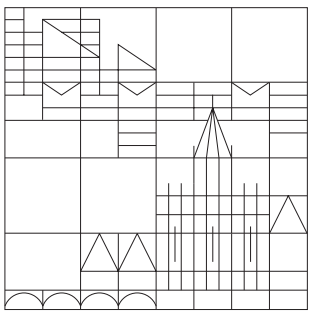
# Universität Konstanz
# FB Informatik und Informationswissenschaft

# Deklarative Programmierung
# (Declarative Programming)

http://www.inf.uni-konstanz.de/dbis


**Prof. Dr. Marc H. Scholl**


Sommersemester 2006

Folien basierend auf früheren Lehrveranstaltungen von M. Scholl und T. Grust

# 0 Funktionale Programmierung

**Aunt Agatha:** Well, James, what programming language are you studying in this term at university?
**James:** Haskell.
**Agatha:** Is that a procedural language, like Pascal?
**James:** No.
**Agatha:** Is it object-oriented, like Java or C++?
**James:** No.
**Agatha:** What then?
**James:** Haskell is a fully higher-order purely functional language with a non-strict semantics and polymorphic static typing.
**Agatha:** Oh. ⟨*Pause*⟩ More tea, James?

*Originally due to Richard Bird.*

# Programmieren mit Termersetzung

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

$$
\begin{array}{ll}
 & \texttt{fac 3} \\
\rightarrow & \texttt{3 * fac (3-1)} \\
\rightarrow & \texttt{3 * fac 2} \\
\rightarrow & \texttt{3 * 2 * fac (2-1)} \\
\rightarrow & \texttt{3 * 2 * fac 1} \\
\rightarrow & \texttt{3 * 2 * 1 * fac (1-1)} \\
\rightarrow & \texttt{3 * 2 * 1 * fac 0} \\
\rightarrow & \texttt{3 * 2 * 1 * 1} \\
\rightarrow^* & \texttt{6}
\end{array}
$$

**Fragen:**

- Wie wird der nächste zu ersetzende Term bestimmt?
- Spiel diese Auswahl überhaupt eine Rolle?
- Gibt es unendliche Ersetzungsfolgen?
- ...

$\rightsquigarrow$ $\lambda$-Kalkül

# The Taste of Functional Programming (FP)

- A programming language is a medium for **expressing ideas** (not to get a computer perform operations). Thus programs must be written for people to read, and only incidentally for machines to execute.
- Using FP, we restrict or **limit not what** we program, but only the notation for our program descriptions.
- Large programs grow from small ones – **idioms**. Develop an arsenal of idioms of whose correctness we are convinced (or whose correctness we have proven). Combining idioms is crucial.
- It is better to have **100 functions** operate on one data structure than 10 functions on 10 data structures.

  *Alan J. Perlis*

- FP in the **real world**: see Ericsson & `http://www.erlang.org/`

# Materialien zur Vorlesung und Haskell

- **Homepage** der Vorlesung:

  `http://www.inf.uni-konstanz.de/dbis/teaching/ss06/fp/`

  Dort: Termine, Räume, Vorlesungsskript, Übungsblätter, Quelltexte, ...

- **Mailing-Liste** der Vorlesung:

  `fp_S06@inf.uni-konstanz.de`

  Erreicht alle Teilnehmer der Vorlesung. **Bitte regelmäßig e-mail lesen!**

- Dokumente zur **Definition von Haskell 98**:

  `http://www.haskell.org/onlinereport/` (Sprachdefinition)
  `http://www.haskell.org/onlinelibrary/` (Bibliotheken)

  Dort sind auch PostScript- und PDF-Versionen der Dokumente zu finden.

- Ideales **Begleitbuch** zur Vorlesung:
  *Richard Bird (, Philip Wadler[1] ),* Introduction to Functional Programming using Haskell*, 2. Ausgabe, 1999, Prentice Hall.*
  U KN Bibliothek `lbs 843/b47` (10 Exemplare, einige Exemplare der 1. Ausgabe finden sich unter `kid 240.20/b47`).

[1] nur in der ersten Auflage

# Haskell im Netz

- **Haskell im Web**

  `http://haskell.org/`

  - Dokumente (*Haskell 98 Report, Haskell 98 Library Report*)
  - Tutorials
  - Haskell-Compiler (`ghc`, `hbc`, `nhc`) und -Interpreter (`hugs`)
  - ...

- **Haskell in den Usenet News**

  `comp.lang.functional`

  Diskussionen über funktionale Programmiersprachen allgemein, teilweise fortgeschrittenere Themen, teilweise Fragen von Anfängern. **Keine Übungsaufgaben posten!**

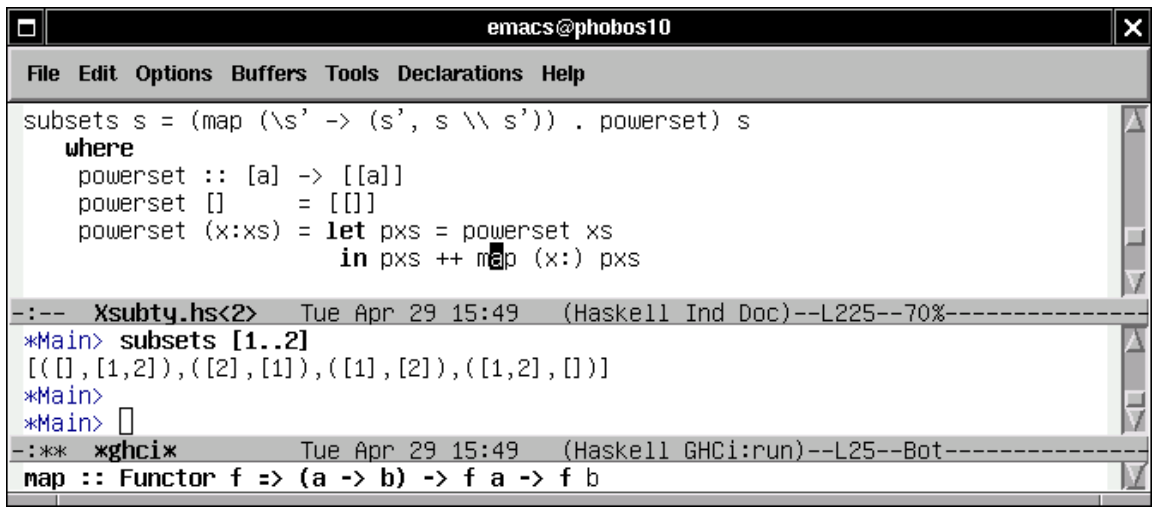
- **Haskell Mailing-Liste**

  `http://haskell.org/mailinglist.html`

  Alle Aspekte von Haskell, Anwendungen, Definition und Entwicklung der Sprache selbst, meist fortgeschrittenere Themen. Aber auch als *read-only* Liste sehr lesenswert!

# Haskell im Linux-Pool

- Editor `emacs`
  - kommt mit `haskell-mode`,
  - beherrscht *syntax coloring*,
  - rückt automatisch ein ($\rightarrow$ 2-dimensionale Syntax, *layout*),
  - interagiert mit `ghci`.
- Haskell-Interpreter `ghci`
  - macht interaktives Arbeiten mit Haskell möglich,
  - bietet *browsing* von Haskell-Quelltexten,
  - liefert Informationen über Typen, Operatoren.



# 1 Funktionale vs. Imperative Programmierung

## 1.1 Einführung

Funktionale Programmiersprachen sind *eine, (wichtige)* Klasse von **deklarativen** Programmiersprachen. Eine andere sind bspw. die **logischen Programmiersprachen**.

Gemeinsam ist allen deklarativen Sprachen, daß die Lösung eines Problems

- auf einer hohen Abstraktionsebene *spezifiziert*, und
- nicht auf einer niedrigen Maschinenebene „ausprogrammiert“ wird.

Offensichtlich sind gemäß dieser Unterscheidung die Grenzen *fließend*!

Entscheidend ist hierbei, was wir als „high-level“ vs. „low-level“ bezeichnen. Gemeint ist in diesem Zusammenhang etwa *nicht* der Unterschied zwischen maschinen-nahen (Assembler-) und problemorientierten („höheren“) Programmiersprachen.

Wir werden uns i.w. auf **funktionale Sprachen** konzentrieren, mit dem Ziel, anhand dieser Klasse die oben „unscharf“ beschriebene Unterscheidung klarer herauszuarbeiten.

# Funktionale Programmierung

Programme einer funktionalen Programmiersprache (*functional programming language, FPL*) bestehen ausschließlich aus **Funktionsdefinitionen** und **Funktionsaufrufen**.

Die Bausteine der Funktionsdefinitionen sind dabei

- der Aufruf weiterer vom Programmierer definierter Funktionen und
- der Aufruf elementarer Funktionen (und Operatoren), die durch die FPL selbst definiert werden.

Anwendung (*application*) einer Funktion $f$ auf ein Argument $e$ ist *das* zentrale Konzept in FPLs und wird daher standardmäßig einfach durch Nebeneinanderschreiben (Juxtaposition) notiert:

$$f\ e$$

Funktionale Programme werden ausschließlich durch Kompositionen von Funktionsanwendungen konstruiert.

| **Funktionale PL** | **Imperative PL** |
|---|---|
| **Programmkonstruktion**: | **Programmkonstruktion**: |
| Applikation und Komposition | Sequenzen von Anweisungen |
| **Operational**: | **Operational**: |
| Funktionsaufruf, Ersetzung von Ausdrücken | Zustandsänderungen (Seiteneffekte) |
| **Formale Semantik**: | **Formale Semantik**: |
| $\lambda$-Kalkül | schwierig (z.B. denotationale Semantiken) |

FPLs bieten konsequenterweise folgende Konzepte *nicht*:

- **Sequenzoperatoren für Anweisungen** (‘;’ in Pascal oder C)
  (Programme werden durch Funktionskomposition zusammengesetzt, eine explizite Reihung von Anweisungen existiert nicht)
- **Zustand**
  (FPLs sind zustandslos und bieten daher keine änderbaren Variablen)
- **Zuweisungen** (‘:=’ in Pascal)
  (Berechnungen in FPLs geschehen allein durch Auswertung von Funktionen, nicht durch Manipulation des Maschinenzustandes bzw. -speichers)

**Beispiel 1.1**

Entwerfe eine Funktion, die testet, ob eine Zahl `n` eine Primzahl ist oder nicht.

① Wenn `n` prim ist, ist die Menge der Teiler (*factors*) von `n` leer.

② Die Menge der Teiler von `n` sind alle Zahlen `x` zwischen $2 \ldots \texttt{n} - 1$, die `n` ohne Rest teilen.

Diese Beschreibung der Eigenschaften einer Primzahl läßt sich direkt in ein funktionales Programm (hier: Haskell) übersetzen:

```
-- Ist n eine Primzahl?
isPrime   :: Integer -> Bool
isPrime n = (factors n == [])                                    -- ①
            where
                 factors n = [ x | x <- [2..n-1], n `mod` x == 0 ]    -- ②
```

Das Programm liest sich mehr wie die deklarative Spezifikation der Eigenschaften einer Primzahl als eine explizite Vorschrift, den Primzahltest auszuführen. Bspw. ist eine parallele Ausführung von `factors` nicht ausgeschlossen.

Imperative Programmiersprachen sind dagegen eng mit dem zugrundeliegenden **von Neumann'schen Maschinenmodell** verknüpft, indem sie die Maschinenarchitektur sehr direkt abstrahieren:

- der **Programmzähler** (PC) der CPU arbeitet Anweisung nach Anweisung sequentiell ab
  (der Programmierer hat seine Anweisungen also explizit aufzureihen und Wiederholungen/Sprünge zu codieren)
- der **Speicher der Maschine** dient zur Zustandsprotokollierung
  (der Zustand eines Algorithmus muß durch Variablenzuweisung bzw. -auslesen explizit kontrolliert werden)

Zusätzlich zur Lösung seines Problemes hat der Programmierer einer imperativen PL die Aufgabe, obige Punkte korrekt zu spezifizieren.

Imperative Programme

- sind oft **länger als ihre FPL-Äquivalente**,
  (Zustandsupdates und -kontrollen sind explizit zu codieren)
- sind oft **schwieriger zu verstehen**,
  (eigentliche Problemlösung und Kontrolle der von Neumann-Maschine werden vermischt)
- sind nur **mittels komplexer Methoden auf Korrektheit zu überprüfen**.
  (Bedeutung jedes Programmteils immer von Zustand des gesamten Speichers und Änderungen auf diesem abhängig)

**Beispiel 1.2**

Primzahltest. Formulierung in PASCAL:

```
program isPrime (output);

function isPrime (n : integer) : boolean;
var
   m            : integer;
   found_factor : boolean;
begin
   m := 2;
   found_factor := false;

   while (m <= n-1) and (not found_factor) do
      if (n mod m) = 0 then found_factor := true
                       else m := m + 1;

   isPrime := not found_factor
end; { isPrime }

begin
   writeln (isPrime(112))
end.
```

Das Programm kontrolliert die Maschine durch explizite Schleifenanweisungen (`while`, `for`), bedingte Anweisungen (`if` · `then` · `else`) und Sequenzierung (';') von Anweisungen. Die Anweisungsfolge ist explizit festgelegt. Das ist das Hauptmerkmal des **imperativen** Stils.

Die Berechnung erfolgt als Seiteneffekt auf den Zustandsvariablen (`m`, `found_factor`). Die Variablen dienen gleichzeitig zur Kontrolle der Maschine (`m`, `found_factor`) und zur Protokollierung des Ergebnisses des eigentlichen Problems (`found_factor`).

Andere Konzepte imperativer PLs bieten noch weitergehenden direkten Zugriff auf die Maschine:

- **Arrays** und **Indexzugriff** (`A[i]`)
  (modelliert direkt den linearen Speicher der Maschine sowie indizierende Adressierungsmodi der CPU)
- **Pointer** und **Dereferenzierung**
  (modellieren 1:1 die indirekten Adressierungsmodi der CPU)
- **explizite (De-)Allokation von Speicher** (`malloc`, `free`, `new`)
  (der Speicher wird eigenverantwortlich als Resource verwaltet)
- **Sprunganweisungen** (`goto`)
  (direkte Manipulation des PC)
- ...

# 2 Programmieren mit Funktionen

Funktionale Programmiersprachen sind vollständig auf dem **mathematischen Funktionsbegriff** aufgebaut. Programme berechnen ihre Ergebnisse allein durch die **Ersetzung** von Funktionsaufrufen durch Funktionsergebnisse.

Dieser Ersetzungsvorgang ist so zentral, daß wir dafür das Zeichen „$\rightarrow\!\triangleright$“ (*reduces to*) reservieren.

**Beispiel 2.1**

```
length [0, 2*2, fac 100]   →▷   3
```

① Die Reihenfolge der Ersetzungen wird durch die Programme *nicht* spezifiziert,
(insbesondere können mehrere Ersetzungen parallel erfolgen)

② und ein Funktionsaufruf kann jederzeit durch sein Ergebnis ersetzt werden, ohne die Bedeutung des Programmes zu ändern (**referentielle Transparenz**, ausführlich in Kapitel 4).

**Frage:** Gilt ② nicht auch für imperative PLs?

**Beispiel 2.2**
Die Mathematik definiert eine Funktion $f$ als eine Menge von geordneten Paaren $(x, y)$. Ob diese Menge explizit (durch eine Maschine) konstruierbar ist, ist hierbei *nicht* relevant.
Die Funktion $f$ mit

$$f\ x = \begin{cases} 1 & \text{wenn } x \text{ irrational ist,} \\ 0 & \text{sonst.} \end{cases}$$

ist auf einem Rechner aufgrund der endlichen und daher ungenauen Repräsentation von irrationalen Zahlen nicht implementierbar.

Im Gegensatz zur Mathematik besitzen FPLs einen ausschließlich **konstruktiven** Funktionsbegriff.

**Der $\lambda$-Kalkül** (eingeführt durch amerikanischen Logiker Alonso Church, ca. 1940) ist ein simpler Formalismus, in dem *alle berechenbaren* Funktionen dargestellt werden können.

Im Gegensatz zur ausdrucksäquivalenten Turing-Maschine war der $\lambda$-Kalkül jedoch gleichzeitig eine geeignete Basis zur Definition von Programmiersprachen. Prinzipiell ist der Kern der modernen FPLs ein um syntaktischen Zucker stark angereicheter $\lambda$-Kalkül (mehr dazu in Kapitel 3).

**Funktionsdefinitionen** werden durch **$\lambda$-Abstraktionen** dargestellt. Beispiel:

$$\underbrace{\lambda x}_{\text{formaler Parameter } x} . \underbrace{(+\ x\ x)}_{\text{Funktionsrumpf}}$$

**Applikation** wird dann über Termersetzung ($\rightarrowtriangle$) formalisiert. Beispiel:

$$(\lambda\ x.(+\ x\ x))\ 3 \quad \rightarrowtriangle \quad (+\ 3\ 3) \quad \rightarrowtriangle \quad 6$$

(die durch das $\lambda$ gebundenen Vorkommen von $x$ im Rumpf werden durch den aktuellen Parameter 3 ersetzt)

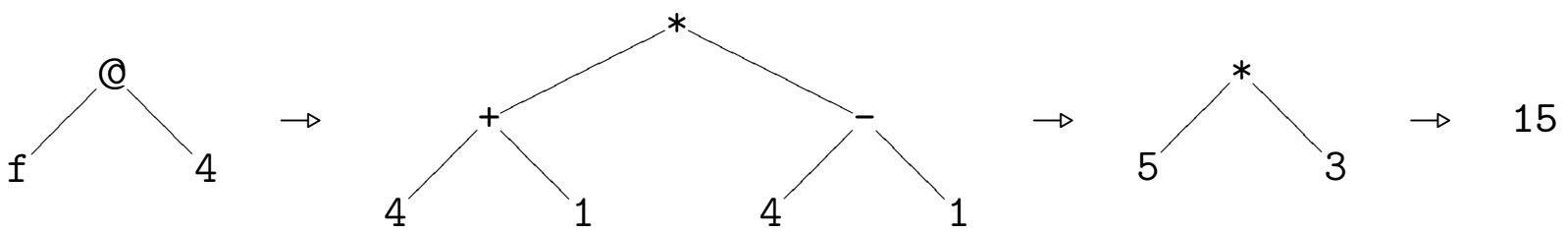
Ersetzungsregeln dieser Art bilden *allein* das operationale Modell aller FPLs.

Die FPL-Gemeinde hat bis heute ausgefeilte Implementierungstechniken entwickelt, um die Operation $\rightarrow\!\triangleright$ effizient zu unterstützen.

Moderne FPL-Compiler erzeugen fast ausschließlich eine interne Graph-Repräsentation des Programmes, die mittels (paralleler) **Graph-Reduktion** die Termersetzung via $\rightarrow\!\triangleright$ nachbildet (sog. „**G-Machines**“).

**Beispiel 2.3**

Sei `f` $= \lambda\, x.$(`*` (`+` $x$ 1) (`-` $x$ 1)). Dann wird der Ausdruck `f 4` wie folgt reduziert:




(@ symbolisiert die Applikation)

**Beobachtungen:**

- **Ausführung eines funktionalen Programmes = Auswertung eines Ausdrucks**.
- Die Auswertung geschieht durch simple **Reduktionsschritte** (Graph-Reduktion).
- Reduktionen können in **beliebiger Reihenfolge**, auch parallel, ausgeführt werden.
  (Reduktionen sind unabhängig voneinander, Seiteneffekte existieren nicht)
- Die Auswertung ist komplett, wenn keine Reduktion mehr möglich ist (**Normalform** erreicht).

Exotische Architekturen, die die Reduktion $\twoheadrightarrow$ auf Maschinen-Ebene unterstützten (z.B. die LISP-Maschinen von Symbolics) konnten sich nicht auf Dauer durchsetzen.

Compiler für imperative PLs erzeugen derzeit jedoch immer noch effizienteren Maschinen-Code: die maschinennahen Konzepte der imperativen PLs sind direkter auf die klassischen von Neumann-Maschinen abbildbar.

Für parallele Maschinen-Architekturen ist dies jedoch nicht unbedingt wahr.

**Beispiel 2.4**

Imperative Vektortransformation (hier in PASCAL notiert) **überspezifiziert die Lösung** durch explizite Iteration (bzgl. `i`):

```
for i := 1 to n do
    A[i] := transform(A[i]);
```

Ein vektorisierender PASCAL-Compiler hat nun die schwierige (oft unmögliche) Aufgabe, zu beweisen, daß die Funktion `transform` keine Seiteneffekte besitzt, um die Transformation parallelisieren zu können. Der inherente Parallelismus wird durch die Iteration verdeckt und muß nachträglich kostspielig wiederentdeckt werden.

Das äquivalente funktionale Programm

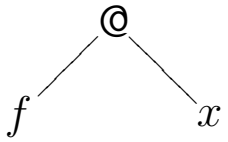
```
map transform A
```

spezifiziert keine explizite Iteration und läßt dem Compiler alle Optimierungsmöglichkeiten.

„Parallele imperative Sprachen“ mit entsprechenden Spracherweiterungen erlauben dem Programmierer die explizite Angabe von Parallelisierungsmöglichkeiten, etwa:

```
for i := 1 to n do in parallel
    A[i] := transform(A[i]);
```

# Strikte vs. nicht-strikte Auswertung

Wird im Graphen 

erst der ① rechte oder der ② linke Zweig reduziert?

① **Strikte Auswertung.** Die weitaus meisten PLs reduzieren das Argument $x$ *bevor* die Definition von $f$ eingesetzt und weiter reduziert wird.

② **Nicht-strikte Auswertung.** Die Expansion der Definition von $f$ *bevor* das Argument $x$ ausgewertet wird, kann unnötige Berechnungen ersparen. Ein Argument wird erst dann ausgewertet, wenn eine Funktion tatsächlich auf das Argument zugreifen muß (Beispiel: Operator + benötigt immer beide Argumente zur Auswertung).

**Beispiel 2.5**

Nicht-strikte Funktionen:

$$\begin{aligned} \texttt{k}\ x\ y &= x \\ \texttt{pos}\ f\ x &= \texttt{if}\ x\ \texttt{<}\ \texttt{0}\ \texttt{then 0 else}\ f\ x \end{aligned}$$

Nicht-strikte Auswertung eröffnet neue Wege zur Strukturierung von Programmen.

- Programme können auf potentiell **unendlich großen Datenstrukturen** (etwa Listen) operieren. Nicht-strikte Auswertung inspiziert die unendliche Struktur nur soweit, wie dies zur Berechnung des Ergebnisses notwendig ist (also nur einen endlichen Teil der Struktur).

**Beispiel 2.6**

Das "Sieb des Eratosthenes" kann einfach als Filter auf einem potentiell unendlichen Strom von natürlichen Zahlen (`[2 .. ]`) definiert werden:

```
primes      :: [Integer]

primes       = sieve [2 .. ]
sieve (x:xs) = x : sieve [ y | y <- xs, y `mod` x > 0 ]
```

Solange nur jeweils eine endliche Anzahl von Primzahlen inspiziert wird (etwa durch den Aufruf `take` $n$ `primes`) terminiert das Programm:

```
> take 10 primes
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29] :: [Integer]
```

- Programme können durch Funktionskomposition **klar strukturiert** und aus einfachen Funktionselementen zusammengesetzt werden. Bei nicht-strikter Auswertung werden in der Komposition

$$g\ (f\ x)$$

$g$ und $f$ synchronisiert ausgeführt: $f$ wird nur dann aufgerufen, wenn $g$ dies verlangt; $f$ konsumiert sein Argument $x$ nur soweit, wie dies zur Beantwortung von $g$s Anfrage notwendig ist.

**Beispiel 2.7**

Ein Programm zur iterativen Wurzelberechnung kann aus einfachen Bauteilen zusammengesetzt werden, die dann durch Komposition zu einer Problemlösung werden.

Erinnerung: Iterative Berechnung von $\sqrt{x}$:

$$\begin{aligned} r_0 &= \frac{x}{2} \\ r_{n+1} &= r_n + \frac{x - r_n^2}{2 \cdot r_n} \end{aligned}$$

Die Iterationvorschrift wird durch die Funktion `rn1` implementiert:

```
rn1 rn = rn + (x - rn*rn) / (2*rn)
```

Die eigentliche Iteration realisieren wir durch die Standardfunktion `iterate f x0`, die die unendliche Liste `[x0, f x0, f(f x0), f(f(f x0)), ... ]` generiert. Der Startwert `x0` ist $r_0 =$ `x/2`. Beispiel (`x = 17`):

```
> iterate rn1 r0 where { r0 = x/2; rn1 rn = rn + (x - rn*rn) / (2*rn) }
[8.5, 5.25, 4.24405, 4.12483, 4.12311, 4.12311, 4.12311, 4.123^C{Interrupted!}
```

Es fehlt nur noch eine Funktion, die entscheidet, ob die Iteration weit genug fortgeschritten ist (die Differenz zweier aufeinanderfolgender Listenelemente ist $< \varepsilon$):

```
within eps (x1:x2:xs) | abs (x1-x2) < eps = x2
                      | otherwise         = within eps (x2:xs)
```

Komposition fügt die Teile zur einer iterativen Wurzelberechnungsfunktion `isqrt` zusammen:

```
isqrt eps x = within eps (iterate rn1 r0)
              where r0     = x/2
                    rn1 rn = rn + (x - rn*rn) / (2*rn)
```

**Beispiel 2.7**

# 3 Exkurs: Der $\lambda$-Kalkül

Alonso Churchs $\lambda$-**Kalkül** (ca. 1940) ist der formale Kern jeder funktionalen Programmiersprache.

Der $\lambda$-Kalkül

- ist eine *einfache* Sprache, mit nur wenigen syntaktischen Konstrukten und simpler Semantik.
  ($\Rightarrow$ Eine Implementation des $\lambda$-Kalküls ist leicht zu erhalten und damit eine gute Basis für die Realisierung von FPLs.)
- ist eine *mächtige* Sprache. Jede berechenbare Funktion kann im $\lambda$-Kalkül dargestellt werden.
  (Alle Konzepte, auch die moderner FPLs, lassen sich auf den $\lambda$-Kalkül abbilden. Im Prinzip könnten wir einen Compiler für eine FPL erhalten, indem wir sie auf den $\lambda$-Kalkül abbilden und dessen Implementation nutzen.)

## FPL $\longleftrightarrow$ $\lambda$-Kalkül.

| | | |
|---|---|---|
| Funktionales Programm | $\equiv$ | Ausdruck des $\lambda$-Kalküls |
| Ausführung des Programms | $\equiv$ | Auswertung des Ausdrucks durch **Reduktion** |

### Primitiver Algorithmus zur Auswertung:

① Wähle nächsten zu reduzierenden Teilausdruck $e$ (**Redex**, *reducible expression*).

② Ersetze Redex $e$ durch Reduktionsergebnis $e'$.

③ Stop, wenn **Normalform** erreicht. Sonst zurück zu ①.

Schreibe für Schritt ② ab jetzt (sprich: $e$ wird zu $e'$ reduziert, $e$ *reduces to* $e'$):

$$e \twoheadrightarrow e'$$

Auswertungsverfahren (insb. die Schritte ① und ③) sind Thema des Kapitels 13.

**Beispiel 3.1**

Gegeben sei der $\lambda$-Ausdruck:

$$(+\ (\times\ 5\ 6)\ (\times\ 8\ 3))$$

**Auswertung durch Reduktion:**

| **Schritt** | **Aktion** | $\lambda$**-Ausdruck** |
|---|---|---|
| ① | Zwei Redexe: $(\times\ 5\ 6)$ und $(\times\ 8\ 3)$. Wähle ersten (beliebig). | $(+\ (\times\ 5\ 6)\ (\times\ 8\ 3))$ |
| ② | Reduktion: $(\times\ 5\ 6) \rightarrow 30$ | $(+\ 30\ (\times\ 8\ 3))$ |
| ③ | Normalform? Nein, $\geq 1$ Redex verbleibt. | $(+\ 30\ (\times\ 8\ 3))$ |
| ① | Wähle einzigen Redex $(\times\ 8\ 3)$ | $(+\ 30\ (\times\ 8\ 3))$ |
| ② | Reduktion: $(\times\ 8\ 3) \rightarrow 24$ | $(+\ 30\ 24)$ |
| ③ | Normalform? Nein, $\geq 1$ Redex verbleibt. | $(+\ 30\ 24)$ |
| ① | Wähle einzigen Redex $(+\ 30\ 24)$ | $(+\ 30\ 24)$ |
| ② | Reduktion: $(+\ 30\ 24) \rightarrow 54$ | 54 |
| ③ | Normalform? Ja, kein Redex verbleibt. **Stop**. | 54 |

## 3.1 Syntax des $\lambda$-Kalküls

**Funktionsanwendung.** Juxtaposition von Funktion und Argument, notiert in **Prefix-Form**:

$$(\mathtt{f}\ \mathtt{x})$$

**Currying.** Anwendung von `f` auf mehr als 2 Argumente? Beispiel: Summe von `x` und `y`:

$$((+\ \mathtt{x})\ \mathtt{y})$$

- Ausdruck $(+\ \mathtt{x})$ ist die Funktion, die `x` zu ihrem Argument addiert,
  (Ein funktionswertiger Ausdruck!)
- die dann auf `y` angewandt wird.

Mittels Currying kann jede Funktion als Funktion eines einzigen Arguments verstanden werden.

**Vereinbarung.** Funktionsanwendung ist *links-assoziativ*. Schreibe daher auch kürzer $(+\ \mathtt{x}\ \mathtt{y})$.

### 3.1.1 Konstante und vordefinierte Funktionen

Der Kern des $\lambda$-Kalküls bietet keine Konstanten (wie `1`, `"foo"`, `True`) oder vordefinierte Funktionen (wie $+$, $\times$, `if`).

In dieser Vorlesung: um Konstante und entsprechende Reduktionsregeln erweiterter $\lambda$-Kalkül.

**Beispiel 3.2**

Reduktion einiger vordefinierter Funktionen (später: $\delta$-Reduktion $\underset{\delta}{\rightarrow\!\triangleright}$, siehe Abschnitt 3.3):

$$
\begin{array}{rcl}
(+\ \mathtt{x}\ \mathtt{y}) & \rightarrow\!\triangleright & \mathtt{x} \boxplus \mathtt{y} \\
(\times\ \mathtt{x}\ \mathtt{y}) & \rightarrow\!\triangleright & \mathtt{x} \boxtimes \mathtt{y} \\
(\mathtt{if\ True}\ e\ f) & \rightarrow\!\triangleright & e \\
(\mathtt{if\ False}\ e\ f) & \rightarrow\!\triangleright & f \\
(\mathtt{and\ False}\ e) & \rightarrow\!\triangleright & \mathtt{False} \\
(\mathtt{and\ True}\ e) & \rightarrow\!\triangleright & e
\end{array}
$$

(Operationen in $\boxplus$ seien direkt auf der Zielmaschine ausführbar, wenn die Argumente `x` und `y` zuvor bis zur Normalform reduziert wurden)

### 3.1.2 $\lambda$-Abstraktionen

Mit den sog. $\lambda$-**Abstraktionen** werden im $\lambda$-Kalkül neue Funktionen definiert:

$$\lambda x.e$$

In dieser $\lambda$-Abstraktion ist $x$ der **formale Parameter**, der im **Funktionskörper** $e$ zur Definition der Funktion benutzt werden kann (sprich: $x$ ist in $e$ gebunden, siehe Abschnitt 3.2.1). Der Punkt '.' trennt $x$ und $e$.

**Beispiel 3.3**
Funktion, die ihr Argument inkrementiert:

$$\lambda x.(+\ x\ 1)$$

Maximumsfunktion:

$$\lambda x.(\lambda y.(\texttt{if}\ (<\ x\ y)\ y\ x))$$

### 3.1.3 Syntax des $\lambda$-Kalküls (Grammatik)

Kontextfreie Grammatik, die syntaktisch korrekte $\lambda$-Ausdrücke beschreibt:

| | | | |
|---|---|---|---|
| $e$ | $\rightarrow$ | $c$ | Konstanten, vordef. Funktionen |
| | $\mid$ | $v$ | Variablen |
| | $\mid$ | $(e\ e)$ | Applikation (Juxtaposition) |
| | $\mid$ | $(\lambda v.e)$ | $\lambda$-Abstraktion |

Treten keine Mehrdeutigkeiten auf, können Klammern $(\cdot)$ weggelassen werden.

**Beispiel 3.4**

$$((\lambda x.(+\ x\ 1))\ 4) \quad \equiv \quad (\lambda x.(+\ x\ 1))\ 4 \quad \equiv \quad (\lambda x.+\ x\ 1)\ 4$$

## 3.2 Operationale Semantik des $\lambda$-Kalküls

Als zweiten Schritt definieren wir die Bedeutung (Semantik) eines $\lambda$-Ausdrucks und legen damit fest, wie man mit $\lambda$-Termen „rechnen“ kann.

### 3.2.1 Freie und gebundene Variablen

Um den folgenden $\lambda$-Ausdruck auszuwerten

$$(\lambda x.+\ x\ y)\ 4$$

- wird der hier nicht bekannte „globale“ Wert der Variablen $y$ benötigt, während
- der Wert des Parameters $x$ im Funktionskörper $(+\ x\ y)$ durch das Argument 4 festgelegt ist.

Innerhalb der $\lambda$-Abstraktion

- ist der Parameter $x$ (durch das $\lambda$) **gebunden**, während
- die Variable $y$ **frei** ist.

**Definition 3.1**

Eine Variable $x$ ist in einem Ausdruck **frei** bzw. **gebunden**:

| | | |
|---|---|---|
| $x$ ist frei in $v$ | $\Longleftrightarrow$ | $x = v$ |
| | | (nicht wenn $x \neq v$ oder $v = c$) |
| $x$ ist frei in $(e\ e')$ | $\Longleftrightarrow$ | $x$ ist frei in $e$ **oder** |
| | | $x$ ist frei in $e'$ |
| $x$ ist frei in $(\lambda y.e)$ | $\Longleftrightarrow$ | $x \neq y$ **und** |
| | | $x$ ist frei in $e$ |

| | | |
|---|---|---|
| $x$ ist gebunden in $(e\ e')$ | $\Longleftrightarrow$ | $x$ ist gebunden in $e$ **oder** |
| | | $x$ ist gebunden in $e'$ |
| $x$ ist gebunden in $(\lambda y.e)$ | $\Longleftrightarrow$ | ($x = y$ und $x$ ist frei in $e$) **oder** |
| | | $x$ ist gebunden in $e$ |

(in einem Ausdruck, der aus einer einzigen Variablen $v$ bzw. Konstanten $c$ besteht, ist $x$ nie gebunden)

Generell hängt der Wert eines Ausdrucks nur von seinen freien Variablen ab.

**Beispiel 3.5**
In der $\lambda$-Abstraktion

$$\lambda x.\ +\ ((\lambda y.\ \times\ y\ z)\ 7)\ x$$

sind $x$ und $y$ gebunden, $z$ jedoch frei.

**Vorsicht!** Bindung/Freiheit muß für *jedes Auftreten einer Variable* entschieden werden. Ein Variablenname kann innerhalb eines Ausdrucks gebunden *und* frei auftreten.

**Beispiel 3.6**

$$+\ \underset{①}{x}\ ((\lambda x.\ +\ \underset{②}{x}\ 1)\ 4)$$

Bei ① tritt $x$ frei auf, während $x$ an Position ② gebunden vorliegt.

## 3.2.2 $\beta$-Reduktion

Eine $\lambda$-Abstraktion beschreibt eine Funktion. Wir definieren hier, wie eine $\lambda$-Abstraktion auf ein Argument angewandt wird.

**Definition 3.2**
Die Funktionsanwendung

$$(\lambda x.e)\ M$$

wird reduziert zu

- einer **Kopie des Funktionsrumpfes** $e$, in der
- die **freien Vorkommen von** $x$ **durch** $M$ **ersetzt** wurden.

**Beispiel 3.7**
Damit gilt

$$(\lambda x.+\ x\ 1)\ 4 \quad \underset{\beta}{\rightarrow} \quad +\ 4\ 1 \quad \rightarrow \quad 4 \boxplus 1 \quad \rightarrow \quad 5$$

Die Reduktion $\underset{\beta}{\rightarrow}$ ist die sog. $\beta$**-Reduktion** des $\lambda$-Kalküls.

**Beispiel 3.8**

Der formale Parameter kann mehrfach oder gar nicht im Funktionsrumpf auftreten:

$$(\lambda x.+\ x\ x)\ 5 \quad \underset{\beta}{\rightarrow} \quad +\ 5\ 5 \quad \rightarrow \quad 10$$

$$(\lambda x.3)\ 5 \quad \underset{\beta}{\rightarrow} \quad 3$$

In einem Funktionsrumpf kann eine weitere $\lambda$-Abstraktion enthalten sein (Currying in Aktion, vgl. Abschnitt 3.1):

$$\begin{aligned}(\lambda x.(\lambda y.\times\ y\ x))\ 4\ 5 \quad &\underset{\beta}{\rightarrow} \quad (\lambda y.\times\ y\ 4)\ 5 \\ &\underset{\beta}{\rightarrow} \quad \times\ 5\ 4 \\ &\rightarrow \quad 20\end{aligned}$$

Wir werden im folgenden $\lambda x.(\lambda y.e)$ oft durch $\lambda x.\lambda y.e$ oder gar $\lambda x\ y.e$ abkürzen.

**Beispiel 3.9**

Funktionen können problemlos als Argumente übergeben werden:

$$
\begin{aligned}
(\lambda f.f\ 3)\ (\lambda x.+\ x\ 1) \quad &\underset{\beta}{\rightarrow} \quad (\lambda x.+\ x\ 1)\ 3 \\
&\underset{\beta}{\rightarrow} \quad +\ 3\ 1 \\
&\rightarrow \quad 4
\end{aligned}
$$

Bei $\beta$-Reduktion werden nur die *freien* Vorkommen des formalen Parameters im Funktionsrumpf ersetzt:

$$
\begin{aligned}
(\lambda x.(\lambda x.+\ (-\ \underset{①}{x}\ 1))\ x\ 3)\ 9 \quad &\underset{\beta}{\rightarrow} \quad (\lambda x.+\ (-\ x\ 1))\ 9\ 3 \\
&\underset{\beta}{\rightarrow} \quad +\ (-\ 9\ 1)\ 3 \\
&\rightarrow \quad 11
\end{aligned}
$$

Das Vorkommen von $x$ bei ① ist durch die innere $\lambda$-Abstraktion gebunden und wird daher bei der ersten $\beta$-Reduktion nicht ersetzt.

### 3.2.3 $\alpha$-Konversion

Die Bedeutung einer $\lambda$-Abstraktion ändert sich nicht, wenn wir ihre formalen Parameter *konsistent* umbenennen, d.h. alle gebundenen Vorkommen des Parameters ebenfalls umbenennen:

$$(\lambda x.\times\ x\ 2) \quad \underset{\alpha}{\leftrightarrow} \quad (\lambda y.\times\ y\ 2)$$

Manchmal ist diese $\alpha$-**Konversion** unerläßlich, um Namenskollisionen und damit fehlerhafte Reduktionen (sog. „*name capture*“) zu vermeiden.

**Beispiel 3.10**
Unter dem Namen *TWICE* sei folgende Funktion definiert (die ihr erstes Argument zweimal auf ihr zweites anwendet):

$$\mathit{TWICE} \quad \equiv \quad (\lambda f.\lambda x.f\ (f\ x))$$

Wir verfolgen jetzt die Reduktion des Ausdrucks ($\mathit{TWICE}\ \mathit{TWICE}$) mittles $\beta$-Reduktion.

$$\begin{aligned}(\mathit{TWICE}\ \mathit{TWICE}) &\underset{\equiv}{\rightarrow} (\lambda f.\lambda x.f\ (f\ x))\ \mathit{TWICE} \\ &\underset{\beta}{\rightarrow} (\lambda x.\ \underbrace{\mathit{TWICE}\ (\underbrace{\mathit{TWICE}\ x}_{②})}_{①})\end{aligned}$$

Es entstehen die Redexe ① und ②. Wir wählen ② beliebig:

$$\begin{array}{ll} \underset{\equiv}{\rightarrow} & (\lambda x.\mathit{TWICE}\ ((\lambda f.\lambda x.f\ (f\ x))\ x)) \\ \underset{\beta}{\rightarrow} & (\lambda x.\mathit{TWICE}\ (\lambda x.\underline{x}\ (\underline{x}\ x))) \end{array}$$

**falsch**

Die $\underline{x}$ sind nun fälschlicherweise durch die innere $\lambda$-Abstraktion gebunden (*captured)*. Umbennunng mittels $\alpha$-Konversion hilft hier. Nach Reduktion von ②:

$$\begin{array}{ll} \underset{\equiv}{\rightarrow} & (\lambda x.\mathit{TWICE}\ ((\lambda f.\lambda x.f\ (f\ x))\ x)) \\ \underset{\alpha}{\leftrightarrow} & (\lambda x.\mathit{TWICE}\ ((\lambda f.\lambda y.f\ (f\ y))\ x)) \\ \underset{\beta}{\rightarrow} & (\lambda x.\mathit{TWICE}\ (\lambda y.x\ (x\ y))) \end{array}$$

**Beispiel 3.10**

## 3.3 Zusammenfassung

Probleme wie *name capture* können die Anwendung der ansonsten simplen $\beta$-Reduktionsregel verkomplizieren. Wir geben hier daher eine endgültige formale Definition der $\beta$-Reduktion, die derartige Probleme berücksichtigt.

**Definition 3.3**
$\beta$-Reduktion:

$$(\lambda x.e)\ M \quad \underset{\beta}{\rightarrow} \quad e[^M/_x]$$

$e[^M/_x]$ (sprich: $M$ für $x$ in $e$) ist definiert durch

| | | | |
|---|---|---|---|
| $v[^M/_x]$ | = | $v$ | wenn $v \neq x$ oder $v =$ c |
| $x[^M/_x]$ | = | $M$ | |
| $(e\ e')[^M/_x]$ | = | $(e[^M/_x]\ e'[^M/_x])$ | |
| $(\lambda x.e)[^M/_x]$ | = | $\lambda x.e$ | |
| $(\lambda v.e)[^M/_x]$ | = | $\lambda v.e[^M/_x]$ | wenn $x$ nicht frei in $e$ ist **oder** $v$ nicht frei in $M$ ist |
| | = | $\lambda z.(e[^z/_v])[^M/_x]$ | sonst ($z$ neuer Variablenname) |

Damit sind alle Werkzeuge zum Umgang mit dem $\lambda$-Kalkül zusammengestellt.

**Operationale Semantik des $\lambda$-Kalküls:**

| | | | | |
|---|---|---|---|---|
| $\alpha$-Konversion | $\lambda x.e$ | $\underset{\alpha}{\leftrightarrow\!\!\triangleright}$ | $\lambda y.e[y/x]$ | wenn $y$ nicht frei in $e$ ist |
| $\beta$-Reduktion | $(\lambda x.e)\ M$ | $\underset{\beta}{\rightarrow\!\!\triangleright}$ | $e[M/x]$ | |
| $\delta$-Reduktion | $(*\ e)$ | $\underset{\delta}{\rightarrow\!\!\triangleright}$ | $\circledast\ e$ | $\circledast$ implementiert $*$ auf der Zielmaschine |

Dieses kompakte formale System ist ausreichend, um als Zielsprache für alle funktionalen Programmiersprachen zu dienen.

- Tatsächlich ist Haskell ein syntaktisch stark angereicherter $\lambda$-Kalkül.
- Sprachelemente von Haskell werden wir teilweise auf den $\lambda$-Kalkül zurückführen.

# 4 Referentielle Transparenz

Die folgenden 4 Zeilen sind Anweisungen einer imperativen Programmiersprache (etwa Pascal):

```
R := f(M)*N + f(N)*M;                    (* ① *)

R := f(M) + f(M);                        (* ② *)

R := 2 * f(M);                           (* ③ *)

if f and f and f then ...                (* ④ *)
```

- Hat die Reihenfolge der beiden Aufrufe von `f` in ① einen Einfluß auf den Wert von `R`? Können beide Aufrufe parallel erfolgen?
- Sind + und $*$ kommutativ?
- Sind die Anweisungen ② und ③ austauschbar?
- Ist in ④ eine Vereinfachung mittels der Äquivalenz `f and f` $=$ `f` möglich?

Generell muß die Antwort hier *nein* lauten! Wo liegt das Problem?

- `f` kann während der Ausführung die (Speicherplätze der) Variablen `M` und `N` ändern, also **Seiteneffekte** erzielen.
- Um sein Resultat zu errechnen, kann `f` auf globale Variablen zugreifen, die sich von Aufruf zu Aufruf geändert haben können.

Der Wert eines Ausdrucks (hier bspw. `f`)

- hängt nicht nur von diesem Ausdruck selbst (hier also der Definition von `f` und `fs` Parametern), sondern vom **Speicher- und Registerzustand** der gesamten Maschine ab und
- wird beeinflußt von der **Reihenfolge der Auswertung** seiner Teilausdrücke.

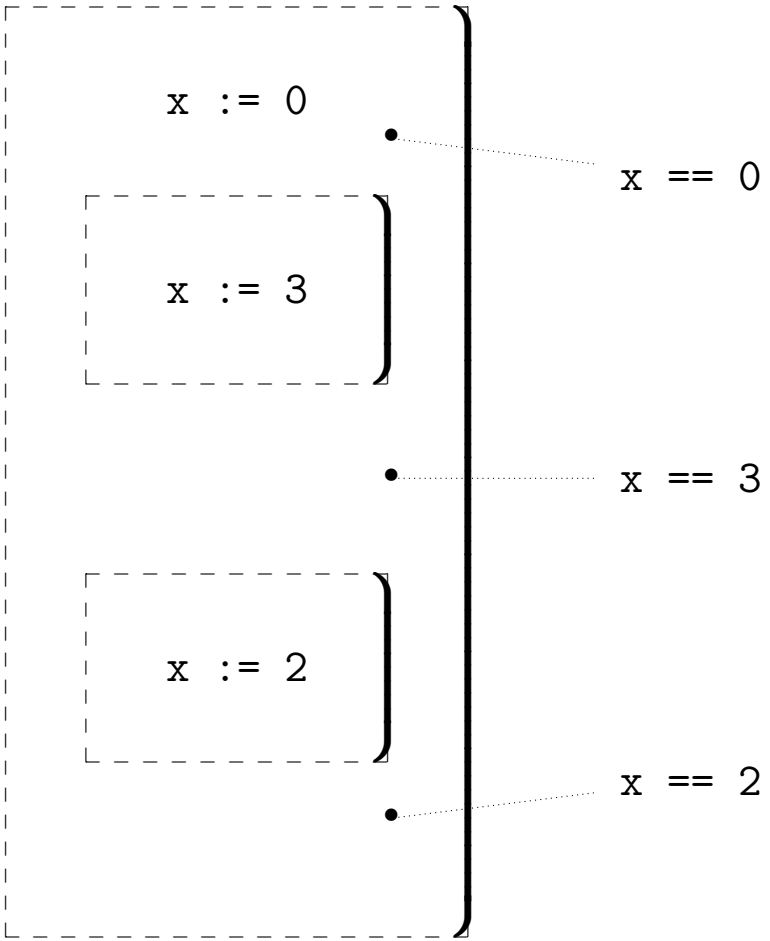
Im mathematischen Sinne ist eine imperative Prozedur `f` eben keine Funktion, d.h.

$$x \texttt{ == } y \quad \not\Rightarrow \quad \texttt{f } x \texttt{ == f } y$$

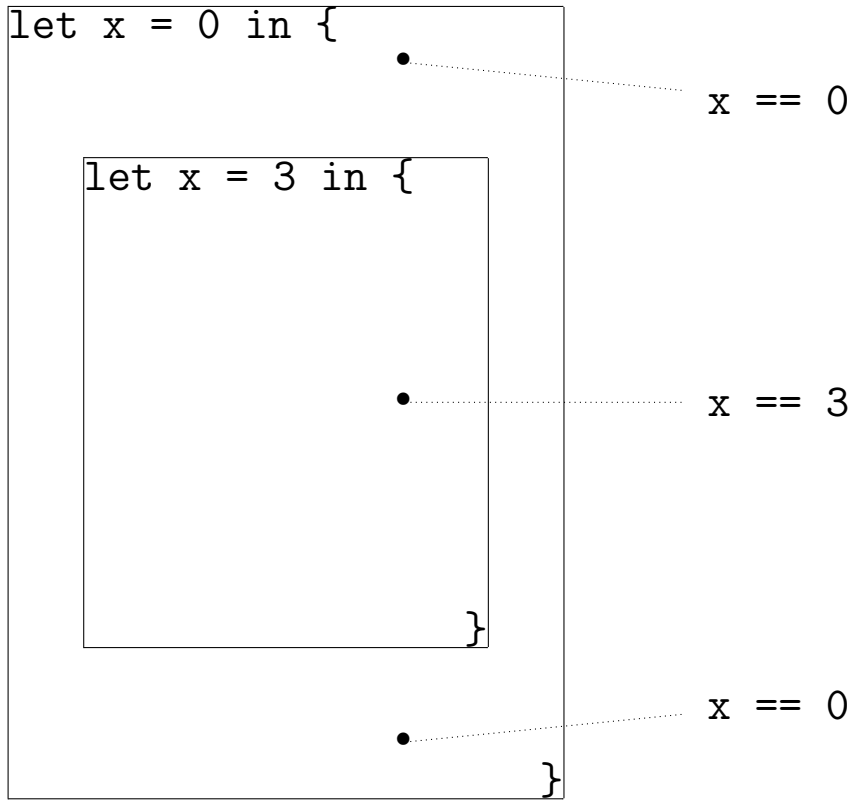
(vgl. Abschnitt 6.2).

In funktionalen Programmen stehen **Namen für Werte** und *nicht* für änderbare Variable.

## Imperative Sprachen




## Funktionale Sprachen




let x = 0 in $e$ $\equiv$ $e[0/x]$

**Konsequenzen:**

① Ein Name darf jeweils durch den von ihm definierten Wert ersetzt werden.
(Das eröffnet unter Umständen Möglichkeiten für Optimierungen durch *unfolding*.)

② Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.
(Es können keine Seiteneffekte durch Variablen-Updates entstehen.)

③ Die Reihenfolge der Auswertung der Teilausdrücke ist irrelevant.
Insb. ist parallele Auswertung möglich.
(Ohne Seiteneffekte gibt es in der Ausdrucksauswertung keine Abhängigkeiten oder Beeinflussungen von außen.)

Sprachen mit diesen Eigenschaften nennen sich

**referentiell transparent**

**Ausdrucksauswertung** in funktionalen Sprachen ist referentiell transparent:

- Ersetze Namen durch ihre Definitionen.
- Wende Funktionen auf Argumente an ($\beta$-Reduktion des $\lambda$-Kalküls).

Ersetzung und Reduktion kann in beliebiger Reihenfolge (auch parallel) erfolgen.

**Beispiel 4.1**

Sei `square` $x$ `=` $x$ `*` $x$.

```
square (3+4)   ⟶ (Def. +)      square 7
               ⟶ (Def. square)  7 * 7
               ⟶ (Def. *)       49
```

```
square (3+4)   ⟶ (Def. square)  (3+4) * (3+4)
               ⟶ (Def. +)       7 * (3+4)
               ⟶ (Def. +)       7 * 7
               ⟶ (Def. *)       49
```

# Referentielle Transparenz...

- ermöglicht einen einfacheren Zugang zur **Programmverifikation**.
  (Einzelne Funktionen können unabhängig vom Rest eines größeren Programmes analysiert werden)
- eröffnet die Möglichkeit, **Programmtransformationen** (etwa zur automatischen Optimierung durch einen Compiler) auszuführen, die nachweislich die Bedeutung des Programms nicht ändern.
  (Äquivalenz von Ausdrücken ist im mathematischen Sinne beweisbar)
- erlaubt den Aufbau ganzer **Programmalgebren**.
  (Elemente dieser Algebren sind Programme, Operationen sind Programmtransformationen)

# Interaktion mit der Welt?

Aber wie verträgt sich die referentielle Transparenz der funktionalen Sprachen mit *real world* Programmieraufgaben und -techniken?

- Input/Output?

```
(\x -> (x,x)) (putStr "foo")
(putStr "foo", putStr "bar")
```

↯ ③ Die Reihenfolge der Auswertung der Teilausdrücke ist irrelevant.

```
getChar == getChar
```

↯ ② Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.

- Zufall?

```
random == random
```

↯ ② Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.

- Zeitsensitive Ausdrücke?

```
getClockTime == getClockTime
```

↯ ② Der Wert eines Ausdrucks hängt nur von den Werten seiner Teilausdrücke ab.

# 5 Haskell – Vorbemerkungen zu Typen

## 5.1 Typen

Intuitiv unterteilt man die Objekte, die man mit einer Programmiersprache manipulieren will, in disjunkte Mengen, etwa: Zeichen, ganze Zahlen, Listen, Bäume und Funktionen:

- Objekte verschiedener Mengen haben **unterschiedliche Eigenschaften**,
  (Zeichen und auch ganze Zahlen sind bspw. anzuordnen, Funktionen sind dies nicht)
- für die Objekte verschiedener Mengen sind **unterschiedliche Operationen** sinnvoll.
  (eine Funktion kann angewandt werden, eine ganze Zahl kann mit `0` verglichen werden, aber auf einen Wahrheitswert kann man nicht addieren, etc.)

Programmiersprachen (wie auch Haskell) formalisieren diese Intuition mittels eines **Typsystems**.

Ein **Typ** definiert

① eine Menge von gleichartigen Objekten (Wertevorrat, „Domain“) und

② Operationen, die auf diese Objekte anwendbar sind (Interface).

Einige **Basis-Typen:**

| Objektmenge | Typname | Operationen (Bsp.) |
|---:|:---:|---|
| Ganze Zahlen | `Integer` | `+`, `max`, `<` |
| Zeichen | `Char` | `ord`, `toUpper`, `<` |
| Wahrheitswerte | `Bool` | `&&`, `==`, `not` |
| Fließkommazahlen | `Double` | `*`, `/`, `round` |

**Typkonstruktoren** konstruieren aus beliebigen Typen $\alpha, \beta$ neue Typen:

| Objektmenge | Typkonstruktor | Operationen (Bsp.) |
|---:|:---:|---|
| Funktionen von $\alpha$ nach $\beta$ | $\alpha$ `->` $\beta$ | `$` (apply) |
| Listen von $\alpha$-Objekten | `[`$\alpha$`]` | `head`, `reverse`, `length` |
| Paare von $\alpha, \beta$-Objekten | `(`$\alpha$`,`$\beta$`)` | `fst`, `snd` |

Die Notation `x` :: $\alpha$ (`x` *has type* $\alpha$) wird vom Haskell-Compiler eingesetzt, um anzuzeigen, daß das Objekt `x` den Typ $\alpha$ besitzt. Umgekehrt können wir so dem Compiler anzeigen, daß `x` eine **Instanz** des Typs $\alpha$ sein soll.

**Beispiel 5.1**

```
          2  ::  Integer
        'X'  ::  Char
       0.05  ::  Double
        ord  ::  Char -> Integer
      round  ::  Double -> Integer
      [2,3]  ::  [Integer]
       head  ::  [α] -> α
('a',(2,True))  ::  (Char,(Integer,Bool))
        snd  ::  (α,β) -> β
```

Die Typen der Funktionen `head` und `snd` enthalten **Typvariablen** $\alpha$ und $\beta$. Dies entspricht der Beobachtung, daß `snd` das zweite Element eines Paares bestimmen kann, ohne Details der gepaarten Objekte zu kennen oder Operationen auf diese anzuwenden. Analog für `head` ($\rightarrow$ **Polymorphie**, siehe Kapitel 12).

# Interpretation komplexer Typen

**Beispiel 5.2**

"Decodierung" des Typs der Prelude-Funktion

```
unzip :: [(α,β)] -> ([α],[β])
```

| | | | | |
|---|---|---|---|---|
| `unzip` | :: | `...` | `->` `...` | `unzip` ist eine Funktion ... |
| `unzip` | :: | `[ ... ]` | `->` `...` | ... die eine Liste |
| `unzip` | :: | `[(α,β)]` | `->` `...` | ... von Paaren als Argument hat |
| `unzip` | :: | `[(α,β)]` | `->` `(...,...)` | ... und ein Paar |
| `unzip` | :: | `[(α,β)]` | `->` `([α],[β])` | ... von Listen als Ergebnis liefert. |

$$\texttt{unzip [(}x_1\texttt{,}y_1\texttt{),...,(}x_n\texttt{,}y_n\texttt{)]} \quad \twoheadrightarrow \quad \texttt{([}x_1\texttt{,...,}x_n\texttt{], [}y_1\texttt{,...,}y_n\texttt{])}$$

# Currying und der Typkonstruktor ->

**Erinnerung.** Mittels Currying kann eine Funktion mehrerer Argumente *sukzessive* auf ihre Argumente angewandt werden (s. Abschnitt 3.1).

**Frage:** Welchen Typ hat die Funktion (der Operator) $+$ daher konsequenterweise (mit $x$ :: `Integer`, $y$ :: `Integer`)?

$$x + y \qquad \equiv \qquad ((+\ x)\ y)$$

**Antwort:**

① Der Teilausdruck $(+\ x)$ besitzt den Typ `Integer -> Integer`,

② damit hat $+$ also den Typ `Integer -> (Integer -> Integer)`.

**Vereinbarung:** `->` ist *rechts-assoziativ*. Schreibe daher kürzer

```
(+) :: Integer -> Integer -> Integer
```

Haskell besitzt einen Mechanismus zur **Typinferenz**, der für (fast) jedes Objekt `x` den zugehörigen Typ $\alpha$ automatisch bestimmt. Darauf kommt Kapitel 12 zurück.

Haskell ist

**streng typisiert,** d.h. eine Operation kann niemals auf Objekte angewandt werden, für die sie nicht definiert wurde.

**statisch typisiert,** d.h. schon zur Übersetzungszeit und nicht während des Programmlaufs wird sichergestellt, daß Programme keine Typfehler enthalten.

(im Haskell-Interpreter werden inkorrekt typisierte Ausdrücke sofort zurückgewiesen)

**Beispiel 5.3**

Typische Typfehlermeldung:

```
Prelude> fst [2,3]

<interactive>:1:
    Couldn't match `(α, β)' against `[γ]'
        Expected type: (α, β)
        Inferred type: [γ]
    In the first argument of `fst', namely `[2, 3]'
Prelude>
```

# 6 Werte und einfache Definitionen

Haskell-Programm (**Skript**) = Liste von **Deklarationen** + **Definitionen**

**Beispiel 6.1**
Fakultätsfunktion `fact`.

```
fact  :: Integer -> Integer
fact n = if n == 0 then
            1
         else
            n * fact (n-1)
```

- **Deklaration** `fact :: Integer -> Integer`
  `fact` ist eine Funktion, die einen Wert des Typs `Integer` (ganze Zahl) auf einen Wert des Typs `Integer` abbildet.
  (Haskell kann den Typ eines Wertes fast immer automatisch aus seiner Definition ableiten, **Typinferenz** siehe Kapitel 12)
- **Definition** `fact n = ...`
  (Rekursive) Regeln für die Berechnung der Fakultätsfunktion.

## 6.1 Basis-Typen

Haskell stellt diverse Basis-Typen zur Verfügung. Die Notation für Konstanten dieser Typen ähnelt anderen Programmiersprachen.

### 6.1.1 Konstanten des Typs `Integer` (ganze Zahlen)

Eine nichtleere Sequenz von Ziffern `0...9` stellt eine Konstante des Typs `Integer` dar. Der Wertebereich ist unbeschränkt, der Typ `Integer` wird in Haskell durch sog. *bignums* realisiert.[2]

Allgemein werden negative Zahlen werden durch die Anwendung der Funktion `negate` oder des Prefix-Operators - gebildet.

**Achtung:** Operator - wird auch zur Subtraktion benutzt, wie etwa in

```
f -123   ≠   f (-123)
```

**Beispiele:** `0`, `42`, `14050061177528798985431426062445115699363840000000`

[2] Haskell kennt außerdem den Typ `Int`, der *fixed precision integers* mit Wertebereich $[-2^{29}, 2^{29}-1]$ realisiert.

## 6.1.2 Konstanten des Typs `Char` (Zeichen)

Zeichenkonstanten werden durch Apostrophe `'·'` eingefaßt. Nichtdruckbare und Sonderzeichen werden mit Hilfe des bspw. auch in C verwendeten \ (*escape, backslash*) eingegeben. Nach \ kann ein ASCII-Mnemonic (etwa `NUL`, `BEL`, `FF`, ...) oder ein dezimaler (oder hexadezimaler nach `\x` bzw. oktaler nach `\o`) Wert stehen, der den ASCII-Code des Zeichens festlegt.

Zusätzlich werden die folgenden Abkürzungen erkannt:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| `\a` | (*alarm*) | `\b` | (*backspace*) | `\f` | (*formfeed*) | `\n` | (*newline*) |
| `\r` | (*carriage return*) | `\t` | (Tab) | `\v` | (*vertical feed*) | `\\` | (*backslash*) |
| `\"` | (*dbl quote*) | `\'` | (*apostroph*) | `\&` | (*NULL*) | | |

**Beispiele:** `'P'`, `'s'`, `'\n'`, `'\BEL'`, `'\x7F'`, `'\''`, `'\\'`

### 6.1.3 Konstanten des Typs `Float` (Fließkommazahlen)

Fließkommakonstanten enthalten stets einen Dezimalpunkt. Vor und hinter diesem steht mindestens eine Ziffer `0...9`. Die Konstante kann optional von `e` bzw. `E` und einem optionalen ganzzahligen Exponenten gefolgt werden.

**Beispiele:** `3.14159`, `10.0e-4`, `0.001`, `123.45E6`

### 6.1.4 Konstanten des Typs `Bool` (Wahrheitswerte)

`Bool` ist ein sog. **Summentyp** (Aufzählungstyp, *enumerated type*) und besitzt lediglich die beiden Konstanten[3] `True` und `False`.

[3] Später: **Konstruktoren**.

## 6.2 Funktionen

Funktionen in funktionalen Programmiersprachen sind tatsächlich als Funktionen im mathematischen Sinne zu verstehen. Ein Wert `f` mit

$$\texttt{f :: } \alpha \texttt{ -> } \beta$$

bildet bei Anwendung Objekte des Typs $\alpha$ auf Objekte des Typs $\beta$ ab und es gilt[4]

$$x \texttt{ == } y \quad \Rightarrow \quad \texttt{f } x \texttt{ == f } y$$

Diese einfache aber fundamentale mathematische Eigenschaft von Funktionen zu bewahren, ist *die* Charakteristik funktionaler Programmiersprachen.

Kapitel 4 beleuchtet die Hintergründe und Konsequenzen.

[4] Dies gilt für eine Prozedur oder Subroutine `f` einer imperativen Programmiersprache im allgemeinen nicht (siehe vorn)!

Sowohl die in der *standard prelude* (Bibliothek vordefinierter Funktionen) als auch in Skripten definierte Funktionsnamen beginnen jeweils mit Kleinbuchstaben `a` ... `z` gefolgt von `a` ... `z`, `A` ... `Z`, `0` ... `9`, `_` und `'`. Haskell ist *case-sensitive*.

**Beispiele:** `foo`, `c_3_p_o`, `f'`

Die **Funktionsapplikation** ist der einzige Weg in Haskell, komplexere Ausdrücke zu bilden. Applikation wird syntaktisch einfach durch **Juxtaposition** (Nebeneinanderschreiben) ausgedrückt:

Anwendung von Funktion `f` auf die Argumente `x` und `y`:

`f␣x␣y`

In Haskell hat die Juxtaposition via ␣ höhere Priorität als Infix-Operatoranwendungen:

`f x + y` bedeutet `(f x) + y`

Klammern `(·)` können zur Gruppierung von Ausdrücken eingesetzt werden.

### 6.2.1 Operatoren

Haskell erlaubt die Deklaration und Definition neuer Infix-Operatoren. Operatornamen werden aus den Symbolen `!#$%&*+/<=>?@\^|~:.` zusammengesetzt. Viele übliche Infix-Operatoren (`+`, `*`, `==`, `<`, ...) sind bereits in der *prelude* vordefiniert.

Operatoren, die mit `:` beginnen, spielen eine Sonderrolle (→ 9.2).

**Beispiel 6.2**
Definition von "fast gleich" `~=~`:

```
epsilon :: Float
epsilon  = 1.0e-4

(~=~)   :: Float -> Float -> Bool
x ~=~ y  = abs (x-y) < epsilon
```

**Infix-Operator → Prefix-Applikation.** Jeder Infix-Operator `op` kann in der Notation `(op)` auch als Prefix-Operator geschrieben werden (siehe auch „Sections“ in Abschnitt 6.2.3):

```
1 + 3          ≡  (+) 1 3
True && False  ≡  (&&) True False
```

**Prefix-Applikation → Infix-Operator.** Umgekehrt kann man jede binäre Funktion `f` (Funktion zweier Argumente) mittels der Schreibweise `‘f‘` als Infix-Operator verwenden:

```
max 2 5  ≡  2 ‘max‘ 5
```

Die so notierten Infix-Operatoren werden durch die Sprache als links-assoziativ und höchster Operatorpriorität (Level 9) interpretiert:

```
5 ‘max‘ 3 + 4  ⇾  9
```

**Bemerkung:** Information über die Assoziativität und Priorität eines Operators durch den Interpreter:

```
> :i +
-- + is a method in class Num
infixl 6 +
(+) :: forall a. (Num a) => a -> a -> a
```

## 6.2.2 Currying

Prinzipiell hat jede in Haskell definierte Funktion *nur einen* Parameter. Funktionen mehrerer Parameter werden durch eine Technik namens **Currying**[5] realisiert:

Der Typ einer Funktion mehrerer Parameter, etwa

```
max :: Integer -> Integer -> Integer
```

ist zu lesen, indem `->` *rechts-assoziativ* interpretiert wird, also:

```
Integer -> Integer -> Integer   ≡   Integer -> (Integer -> Integer)
```

Damit `max` eine Funktion *eines* `Integer`-Parameters, die bei Anwendung einen Wert (hier: wieder eine Funktion) des Typs `Integer -> Integer` liefert. Dieser kann dann auf ein weiteres `Integer`-Argument angewandt werden, um letzlich das Ergebnis des Typs `Integer` zu bestimmen.

[5] Die Bezeichnung erinnert an den Logiker Haskell B. Curry, dessen Vorname der Sprache ihren Namen verleiht. Die Technik wurde tatsächlich vorher von Schönfinkel eingesetzt, *Schönfinkeling* konnte sich jedoch nicht als Bezeichnung durchsetzen :-)

## Currying im $\lambda$-Kalkül:

Haskell-Funktionsdefinitionen sind tatsächlich lediglich syntaktischer Zucker für die schon bekannten $\lambda$-Abstraktionen:

$$\texttt{f x} = e \quad \equiv \quad \texttt{f} = \lambda\texttt{x}.e$$
$$\texttt{g x y} = e \quad \equiv \quad \texttt{g} = \lambda\texttt{x}.\lambda\texttt{y}.e$$

Damit läßt sich Currying einfach durch mehrfache $\beta$-Reduktion erklären.

**Beispiel 6.3**
Maximumsfunktion `max`:

```
max    :: Integer -> Integer -> Integer
max x y = if x<y then y else x     -- ≡ max = λx.λy.if x<y then y else x
```

Auswertung von `max 2 5`:

$$
\begin{aligned}
\texttt{max 2 5} &\underset{\text{Def. max}}{\longrightarrow} \texttt{(}\lambda\texttt{x.}\lambda\texttt{y.if x<y then y else x) 2 5} \\
&\underset{\beta}{\longrightarrow} \texttt{(}\lambda\texttt{y.if 2<y then y else 2) 5} \\
&\underset{\beta}{\longrightarrow} \texttt{if 2<5 then 5 else 2} \\
&\underset{\delta}{\longrightarrow} \texttt{5}
\end{aligned}
$$

Currying erlaubt die **partielle Anwendung von Funktionen**. Der Wert des Ausdrucks `(+) 1` hat den Typ `Integer -> Integer` und ist die Funktion, die `1` zu ihrem Argument addiert.

**Beispiel 6.4**

Nachfolgerfunktion `inc`:

```
inc :: Integer -> Integer
inc  = (+) 1
```

### 6.2.3 Sections

Currying ist auch auf binäre Operatoren anwendbar, da Operatoren ja lediglich binäre Funktionen in Infix-Schreibweise (mit festgelegter Assoziativität) sind. Man erhält dann die sog. **Sections**.

Für jeden Infix-Operator `op` gilt (die Klammern $(\cdot)$ gehören zur Syntax!):

$$\begin{aligned} \texttt{(x op)} &\equiv \lambda\texttt{y.x op y} \\ \texttt{(op y)} &\equiv \lambda\texttt{x.x op y} \\ \texttt{(op)} &\equiv \lambda\texttt{x.}\lambda\texttt{y.x op y} \end{aligned}$$

**Beispiel 6.5**
Mittels Sections können viele Definitionen und Ausdrücke elegant notiert werden:

```
inc      = (1+)
halve    = (/2)
add      = (+)
positive = (`max` 0)
```

## 6.2.4 $\lambda$-Abstraktionen (anonyme Funktionen)

Prinzipiell sind $\lambda$-Abstraktionen der Form $\lambda\mathtt{x}.e$ namenslose (anonyme) Funktionsdefinitionen. Haskell erlaubt $\lambda$-Abstraktionen als Ausdrücke und somit anonyme Funktionen. Haskells Notation für $\lambda$-Abstraktionen ähnelt dem $\lambda$-Kalkül:

$$\begin{array}{rcl} \lambda\mathtt{x}.e & \equiv & \texttt{\textbackslash x -> } e \\ \lambda\mathtt{x}.\lambda\mathtt{y}.e & \equiv & \texttt{\textbackslash x -> \textbackslash y -> } e \quad \equiv \quad \texttt{\textbackslash x y -> } e \end{array}$$

Damit wird der Ausdruck `(\x -> 2*x) 3` zu 6 ausgewertet und die vorige Definition von `max` kann alternativ wie folgt geschrieben werden:

```
max :: Integer -> Integer -> Integer
max  = \x -> \y -> if x<y then y else x
```

## 6.3 Listen

Listen sind *die* primäre Datenstruktur in funktionalen Programmiersprachen. Haskell unterstützt die Konstruktion und Verarbeitung homogener Listen beliebigen Typs: Listen von `Integer`, Listen von Listen, Listen von Funktionen, ...

Die Struktur von Listen ist **rekursiv**:

- Eine Liste ist entweder leer
  (notiert als `[]`, gesprochen *nil*)
- oder ein konstruierter Wert aus Listenkopf `x` (*head*) und Restliste `xs` (*tail*)
  (notiert als `x:xs`, der Operator `(:)` heißt *cons*[6])

Der Typ von Listen, die Elemente des Typs $\alpha$ enthalten, wird mit $[\alpha]$ bezeichnet (gesprochen *list of* $\alpha$).

[6] Für *list construction*.

`[]` und `(:)` sind Beispiele für sog. "*data constructors*", Funktionen, die Werte eines bestimmten Typs konstruieren. Wir werden dafür noch zahlreiche Beispiele kennenlernen. Jede Liste kann mittels `[]` und `(:)` konstruiert werden:

- Liste, die 1 bis 3 enthält: `1:(2:(3:[]))`
  (`(:)` assoziiert nach rechts, daher bezeichnet `1:2:3:[]` den gleichen Wert)

**Syntaktische Abkürzung:**

$$[e_1,e_2,\ldots,e_n] \quad \equiv \quad e_1:e_2:\ldots:e_n:[]$$

**Beispiel 6.6**

```
                    []  ::  [α]
                 'z':[]  ::  [Char]
           [[1],[2,3],[]]  ::  [[Integer]]
            (False:[]):[]  ::  [[Bool]]
[(<),(<=),(>),(>=)]  ::  [α -> α -> Bool]
                  [[]]  ::  [[α]]
```

**Arithmetische Sequenzen:**

$$\begin{array}{rcl} [x..y] & \equiv & \text{wenn } x\text{<=}y \text{ dann } [x,x\text{+}1,x\text{+}2,\ldots,y] \text{ sonst } [] \\ [x,s..y] & \equiv & \text{Liste der Werte } x \text{ bis } y \text{ mit Schrittweite } s\text{-}x \end{array}$$

**Beispiel 6.7**
Der Ausdruck `[2 .. 6]` wird zu `[2,3,4,5,6]` ausgewertet, `[7,6 .. 3]` ergibt `[7,6,5,4,3]` und `[0.0,0.3 .. 1.0]` $\rightarrowtriangle$ `[0.0,0.3,0.6,0.9]`.

## 6.3.1 Listen-Dekomposition

Mittels der vordef. Funktionen `head` und `tail` kann eine *nicht-leere* Liste `x:xs` wieder in ihren Kopf und Restliste zerlegt werden:

```
head (x:xs)  ⊳  x
tail (x:xs)  ⊳  xs
head []      ⊳  *** Exception: Prelude.head: empty list
tail []      ⊳  *** Exception: Prelude.tail: empty list
```

## 6.3.2 Konstanten des Typs `String` (Zeichenketten)

Zeichenketten werden in Haskell durch den Typ `[Char]` repräsentiert, eine Zeichenkette ist also eine Liste von Zeichen. Funktionen auf Listen können damit auch auf Strings operieren. Haskell kennt `String` als Synonym für den Typ `[Char]` (realisiert durch die Deklaration `type String = [Char]` ($\rightarrow$ später)).

Strings werden in doppelten Anführungszeichen `"` $\cdot$ `"` notiert.

**Beispiel 6.8**

```
                    ""
                 "AbC"
               'z':[]   ≡   "z"
['C','u','r','r','y']   ≡   "Curry"
        head "Curry"    →▷   'C'
        tail "Curry"    →▷   "urry"
  tail (tail "OK\n")    →▷   "\n"
```

## 6.4 Tupel

Tupel erlauben die Gruppierung von Daten unterschiedlicher Typen (Listen erlauben dies nicht).

Ein **Tupel** $(c_1, c_2, \ldots, c_n)$ besteht aus einer fixen Anzahl von **Komponenten** $c_i :: \alpha_i$. Der Typ dieses Tupels wird notiert als $(\alpha_1, \alpha_2, \ldots, \alpha_n)$.

**Beispiel 6.9**

```
                (1, 'a')  ::  (Integer, Char)
       ("foo", True, 2)  ::  ([Char], Bool, Integer)
  ([(*1), (+1)], [1..10])  ::  ([Integer -> Integer], [Integer])
((1, 'a'), ((3, 4), 'b'))  ::  ((Integer, Char), ((Integer, Integer), Char))
```

Die Position einer Komponente in einem Tupel ist signifikant. Es gilt

$$(c_1, c_2, \ldots, c_n) \texttt{ == } (d_1, d_2, \ldots, d_m)$$

nur, wenn $n = m$ und $c_i$ == $d_i$ $(1 \leq i \leq n)$.

Der Zugriff auf die einzelnen Komponenten eines Tupels geschieht ausschließlich durch **Pattern Matching** (siehe auch Abschnitt 7.1).

**Beispiel 6.10**

Zugriffsfunktionen für die Komponenten eines 2-Tupels und Tupel als Funktionsergebnis:

```
fst      :: (α, β) -> α
fst (x,y) = x

snd      :: (α, β) -> β
snd (x,y) = y

mult     :: Integer -> (Integer, Integer -> Integer)
mult x    = (x, (*x))

> (snd (mult 3)) 5
15
it :: Integer
```

# 7 Funktionsdefinitionen

Typischerweise analysieren Funktionen ihre Argumente, um **Fallunterscheidungen** für die Berechnung des Funktionsergebnisses zu treffen. Je nach Beschaffenheit des Argumentes wird ein bestimmter Berechnungszweig gewählt.

**Beispiel 7.1**
Summiere die Elemente einer Liste `l`:

```
sum  :: [Integer] -> Integer
sum l = if l == [] then 0                          -- ①
                   else (head l) + sum(tail l)     -- ②
```

`sum` hat den Berechnungszweig mittels `if·then·else` im Fall der leeren Liste ① bzw. nichtleeren Liste ② auszuwählen und im letzeren Fall explizit auf Kopf und Restliste von `l` zuzugreifen.

## 7.1 Pattern Matching und `case`

Mit **Pattern Matching** bietet Haskell die Möglichkeit, für jeden der einzelnen Berechnungszweige einer Funktion `f` Muster $p_{i1} \ldots p_{ik}$ anzugeben, die die erwartete Struktur der $k$ Argumente spezifieren. Nur wenn die Argumente von `f` den Mustern $p_{i1} \ldots p_{ik}$ entsprechen, wird der entsprechende Berechnungszweig $e_i$ ausgewählt:

$$
\begin{array}{lcl}
\texttt{f} & \texttt{::} & \alpha_1 \texttt{ -> } \ldots \texttt{ -> } \alpha_k \texttt{ -> } \beta \\
\texttt{f } p_{11} \ldots p_{1k} & \texttt{=} & e_1 \\
\texttt{f } p_{21} \ldots p_{2k} & \texttt{=} & e_2 \\
 & \vdots & \\
\texttt{f } p_{n1} \ldots p_{nk} & \texttt{=} & e_n
\end{array}
$$

Die $e_i$ müssen dabei einen gemeinsamen (allgemeinsten) Typ $\beta$ besitzen.

**Semantik:** Beim Aufruf `f` $x_1 \ldots x_k$ werden die $x_j$ gegen die Muster $p_{ij}$ ($i = 1 \ldots n$, von oben nach unten) „gematcht“ und der erste Berechnungszweig gewählt, bei dem ein vollständiger *match* vorliegt. Wenn kein *match* erzielt wird, wird das Programm mit einem `Program error` angehalten.

**Definition 7.1**

Erlaubte Formen für ein Pattern $p_{ij}$:

- **Variable** $v$

  (der *match* gelingt immer; $v$ wird an das aktuelle Argument $x_j$ gebunden und ist in $e_i$ verfügbar; Pattern müssen **linear** sein, d.h. eine Variable $v$ darf nur genau *einmal* in einem Pattern auftauchen)

- **Konstante** $c$

  (der *match* gelingt nur mit einem Argument $x_j$ mit $x_j$ == $c$)

- **Successor-Pattern** $n$+$k$ ($n$ Variable, $k$ Konstante des Typs `Integer`)

  (der *match* gelingt, wenn $x_j \geq k$; $n$ wird dann an den Wert $x_j$ - $k$ gebunden)

- **Wildcard** '_' (*don't care*)

  (der *match* gelingt immer, es wird aber keine Bindung hergestellt)

- **Tupel-Pattern** $(p_1, p_2, \ldots, p_m)$

  (der *match* gelingt mit einem $m$-Tupel, dessen Komponenten mit den Pattern $p_1 \ldots p_m$ *matchen*)

- **List-Pattern** [] und $p{:}ps$

  (während [] nur auf die leere Liste *matcht*, gelingt der *match* mit $p{:}ps$ für jede nichtleere Liste, deren Kopf $p$ und deren Rest $ps$ *matcht*)

Diese Definition von Patterns ist aufgrund der beiden letzten Fälle rekursiv.

**Beispiel 7.2**

Die Funktion `sum` läßt sich mittels Pattern Matching wie folgt reformulieren:

```
sum       :: [Integer] -> Integer
sum []     = 0
sum (x:xs) = x + sum xs
```

Sowohl die Fallunterscheidung als auch der Zugriff auf *head* und *tail* des Arguments geschehen nun elegant durch Pattern Matching.

**Beispiel 7.3**

Funktion zur Bestimmung der ersten `n` Elemente einer Liste:

```
take          :: Integer -> [α] -> [α]
take 0  _      = []
take _  []     = []
take n  (x:xs) = x : take (n-1) xs
```

**Beispiel 7.4**

Berechne $\mathtt{x}^{\mathtt{n}}$ (kombiniert Wildcard, Tupel- und $n$+$k$-Pattern):

```
power         :: (Float, Integer) -> Float
power (_, 0)   = 1.0
power (x, n+1) = x * power (x,n)
```

**Beachte:** `power` ist trotz der Übergabe von `x` und `n` eine Funktion *eines* (tupelwertigen) Parameters.

**Beispiel 7.5**

**Achtung:** Eine potentielle Fehlerquelle sind Definitionen wie etwa `sum2`:

```
sum2        :: [Integer] -> Integer
sum2 [x, y]  = x + y                    -- ①
sum2 [0, _]  = 0                        -- ②
```

Der Zweig ② wird auch bei einem Aufruf `sum2 [0,5]` nicht ausgewertet (das Pattern in Zweig ① überdeckt das Pattern in Zweig ②). Allgemein gilt: Spezialfälle *vor* den allgemeineren Fällen anordnen.

### 7.1.1 Layered Patterns

Auf die Komponenten eines Wertes $e$ kann mittels Pattern Matching ($e = (p_1, \ldots, p_n), e = p{:}ps, \ldots$) zugegriffen werden. Oft ist in Funktionsdefinitionen aber gleichzeitig auch der Wert von $e$ selbst interessant.

$v$ sei eine Variable, $p$ ein Pattern. Das **Layered Pattern** (**as-Pattern**)

$$v@p$$

*matcht* gegen $e$, wenn $p$ gegen $e$ *matcht*. Zusätzlich wird $v$ an den Wert $e$ gebunden.

**Beispiel 7.6**
Variante der Funktion `within` aus Beispiel 2.7. Schneide Liste von Näherungswerten ab, sobald sich die Werte weniger als `eps` unterscheiden.

```
within'                    :: Float -> [Float] -> [Float]
within' _   []              = []
within' _   [y]             = [y]
within' eps (y:rest@(x:xs)) = if abs(x-y) < eps then [y,x]
                                                 else y : within' eps rest
```

**Beispiel 7.7**

Mische zwei geordnete Listen bzgl. der Relation `lt` (bspw. benötigt in der *merge*-Phase von Mergesort). Beispiel: `merge (<) [1,3 .. 10] [2,4 .. 10]` $\rightarrowtriangle$ `[1,2,3, ..., 10]`. Formulierung ohne as-Patterns:

```
merge                  :: (α -> α -> Bool) -> [α] -> [α] -> [α]
merge lt [] ys          = ys
merge lt xs []          = xs
merge lt (x:xs) (y:ys) = if x `lt` y then x : merge lt xs (y:ys)
                                     else y : merge lt (x:xs) ys
```

Die Listenargumente werden erst mittels Pattern Matching analysiert, um danach evtl. wieder via `(:)` identisch zusammengesetzt zu werden. Jetzt Formulierung via as-Patterns:

```
merge lt [] ys                = ys
merge lt xs []                = xs
merge lt l1@(x:xs) l2@(y:ys) = if x `lt` y then x : merge lt xs l2
                                            else y : merge lt l1 ys
```

Hier ist die Listenrekonstruktion nicht notwendig.

### 7.1.2 `case`

Pattern Matching ist so zentral, daß es nicht nur in den einzelnen Zweigen einer Funktionsdefinition zur Verfügung steht. Tatsächlich stellen `case`-Ausdrücke die eigentliche Maschinerie zum Pattern Matching zur Verfügung (s. unten).

Im `case`-Ausdruck

$$\begin{array}{lllll} \texttt{case}\ e\ \texttt{of} & p_1 & \texttt{->} & e_1 \\ & p_2 & \texttt{->} & e_2 \\ & & \vdots & \\ & p_n & \texttt{->} & e_n \end{array}$$

wird der Wert des Ausdrucks $e$ nacheinander gegen die Pattern $p_i$ $(i = 1 \ldots n)$ *gematcht*. Falls $e$ auf $p_k$ *matcht* ist der Wert des Gesamtausdrucks $e_k$. Trifft kein Muster zu, resultiert ein `Program error` (der Wert des Ausdrucks ist dann $\bot$ (*bottom*), siehe Kapitel 13).

**Beispiel 7.8**

"*Zipper*" für zwei Listen:

```
zip        :: [α] -> [β] -> [(α,β)]
zip xs ys = case (xs, ys) of
                 ([], _)      -> []
                 (_, [])      -> []
                 (x:xs, y:ys) -> (x, y) : zip xs ys
```

In Haskell gehören `case`-Ausdrücke zum innersten Sprachkern. Viele andere Sprachkonstrukte (insb. *alle*, die auf Pattern Matching bauen) werden intern auf `case` zurückgeführt.

**Beispiel 7.9**

Bedingte Ausdrücke werden mittels `case` implementiert:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \quad \equiv \quad \begin{array}{l}\texttt{case } e_1 \texttt{ of True } \texttt{->} \; e_2 \\ \qquad\qquad\quad \texttt{False } \texttt{->} \; e_3\end{array}$$

Damit wird die Forderung nach einem gemeinsamen allgemeinsten Typ $\alpha$ von $e_2$ und $e_3$ deutlich.

Funktionsdefinitionen mittels Pattern Matching werden intern in `case`-Ausdrücke über Tupeln übersetzt:

$$\texttt{f } p_1 \ldots p_k \texttt{ = } e \quad \equiv \quad \begin{array}{l}\texttt{f } v_1 \ldots v_k \texttt{ = case } (v_1\texttt{,}\ldots\texttt{,}v_k) \texttt{ of} \\ \qquad\qquad\qquad\qquad\quad (p_1\texttt{,}\ldots\texttt{,}p_k) \texttt{ -> } e\end{array}$$

($v_1, \ldots, v_k$ neue Variablen). Damit hat der Compiler lediglich die etwas einfachere Aufgabe, Funktionsdefinitionen ohne Pattern Matching übersetzen zu können.

## 7.2 Guards

Oft ist die Analyse der Struktur der Argumente einer Funktion nicht ausreichend, um den korrekten Berechnungszweig auszuwählen. **Guards** („Wächter“) bieten neben Pattern Matching die Möglichkeit, beliebige Tests auszuführen, bevor ein Zweig tatsächlich gewählt wird:

$$
\begin{array}{lll}
\texttt{f} & :: & \alpha_1 \texttt{ -> } \ldots \texttt{ -> } \alpha_k \texttt{ -> } \beta \\
\texttt{f } p_{11} \ldots p_{1k} \texttt{ | } g_{11} & = & e_{11} \\
\texttt{ | } g_{12} & = & e_{12} \\
\texttt{ | } \ldots & = & \ldots \\
 & \vdots & \\
\texttt{f } p_{n1} \ldots p_{nk} \texttt{ | } g_{n1} & = & e_{n1} \\
\texttt{ | } g_{n2} & = & e_{n2} \\
\texttt{ | } \ldots & = & \ldots
\end{array}
$$

Die Guards $g_{ij}$ sind Ausdrücke des Ergebnistyps `Bool`. In den Guards $g_{ij}$ $(j \geq 1)$ sind die durch die Patterns $p_{i1} \ldots p_{ik}$ gebundenen Variablen nutzbar.

**Semantik:** Falls die Pattern $p_{i1} \ldots p_{ik}$ *matchen*, werden die Guards $g_{ij}$ von oben nach unten getestet. Wird dabei $g_{ij}$ als erster Guard zu `True` ausgewertet, ist $e_{ij}$ der Wert von `f`. Der spezielle Guard `otherwise` evaluiert immer zu `True`, dienlich für „*catchall*“-Alternativen.

Guards können oft explizite Abfragen mittels `if` · `then` · `else` ersetzen.

**Beispiel 7.10**

Selektiere die Elemente einer Liste, die die Bedingung `p` erfüllen:

```
filter                    :: (α -> Bool) -> [α] -> [α]
filter p []                = []
filter p (x:xs) | p x      = x : filter p xs
                | otherwise = filter p xs
```

**Beispiel 7.11**

Erneute Neuformulierung der Hilfsfunktion zur Iteration `within'`:

```
within'                                    :: Float -> [Float] -> [Float]
within' _   []                              = []
within' _   [y]                             = [y]
within' eps (y:rest@(x:xs)) | abs(x-y) < eps = [y,x]
                            | otherwise      = y : within' eps rest
```

**Beispiel 7.12**

Lösche adjazente Duplikate aus einer Liste:

```
remdups                          :: [α] -> [α]
remdups (x:xs@(y:_)) | x == y    = remdups xs
                     | otherwise = x : remdups xs
remdups xs                       = xs
```

**Frage:** Könnte der letzte Zweig auch `remdups [] = []` geschrieben werden?

**Beispiel 7.13**

Ist ein Element `e` in einer *absteigend geordneten* Liste vorhanden?

```
elem'                  :: α -> [α] -> Bool
elem' _ []              = False
elem' e (x:xs) | e > x  = False
               | e == x = True
               | e < x  = elem' e xs
```

Guards können auch in `case`-Ausdrücken verwandt werden. Die Syntax wird analog zu der von Funktionsdefinitionen erweitert:

$$\begin{array}{rrcl}
\texttt{case}\ e\ \texttt{of}\ p_1 & \texttt{|}\ g_{11} & \texttt{->} & e_{11} \\
& \texttt{|}\ g_{12} & \texttt{->} & e_{12} \\
& & \vdots & \\
p_n & \texttt{|}\ g_{n1} & \texttt{->} & e_{n1} \\
& \texttt{|}\ g_{n2} & \texttt{->} & e_{n2}
\end{array}$$

**Beispiel 7.14**

Entferne die ersten `n` Elemente der Liste `l`:

```
drop    :: Integer -> [α] -> [α]
drop n l = case l of
               []              -> []
               (x:xs) | n > 0  -> drop (n-1) xs
                      | n == 0 -> x:xs
```

## 7.3 Lokale Definitionen (`let` und `where`)

Es kann oft nützlich sein, lediglich **lokal sichtbare Namen** in Ausdrücken nutzen zu können. Damit können

- **Sichtbarkeiten von Namen** beschränkt und
  (um so Implementationdetails zu verbergen)
- **öfter auftretende identische Teilausdrücke** aus einem Ausdruck „herausfaktorisiert“ werden.
  (kann die Effizienz der Ausdrucksauswertung steigern)

Wenn $e$ ein Haskell-Ausdruck ist, ist der `let`-Ausdruck

$$\begin{array}{ll} \texttt{let} & d_1 \\ & d_2 \\ & \vdots \\ & d_n \\ \texttt{in} & e \end{array}$$

ein gültiger Ausdruck. Die $d_i$ ($i \geq 1$) sind dabei Definitionen der Form $p_i$ = $e_i$ ($p_i$ Pattern). Reihenfolge der $d_i$ ist unerheblich, die $d_i$ dürfen verschränkt rekursiv sein. In $e$ erscheinen die durch den Pattern-Match von $e_i$ gegen $p_i$ definierten Namen an ihre Werte **lokal gebunden** und sind **außerhalb des Scope von $e$ unbekannt**.

**Semantik:** Der obige `let`-Ausdruck hat den Wert

$$(\lambda\ p_1\ p_2 \ldots p_n.e)\ e_1\ e_2 \ldots e_n \qquad \underset{\beta}{\rightarrow\!\!\triangleright^*} \qquad e\,[e_1/p_1][e_2/p_2]\ldots[e_n/p_n]$$

**Beachte:** Die linke Seite macht deutlich, daß die Auswertung der $e_i$ *lazy* geschieht ($e_i$ wird nur dann tatsächlich ausgewertet, wenn dies zur Auswertung von $e$ erforderlich ist).

**Beispiel 7.15**

Wir vervollständigen die Implementation von Mergesort (siehe `merge` in Beispiel 7.7) durch Definition von `mergesort`:

```
mergesort       :: (α -> α -> Bool) -> [α] -> [α]
mergesort lt []  = []
mergesort lt [x] = [x]
mergesort lt xs  = let (l1,l2) = split xs
                   in
                       merge lt (mergesort lt l1) (mergesort lt l2)
```

Es verbleibt die Definition der für Mergesort typischen *Divide-Phase* mittels `split`, die eine Liste `xs` in zwei Listen ungefähr gleicher Länge teilt (das Listenpaar wird in einem 2-Tupel zurückgegeben).

Wie ist eine Liste `xs` unbekannter Länge in zwei ca. gleich lange Teillisten $\texttt{l}_1, \texttt{l}_2$ zu teilen?

**Implementation von `split`, Idee ①:** Stelle Listenlänge `n` von `xs` fest. Dann sind die ersten `(n 'div' 2)` Elemente in $\texttt{l}_1$, alle anderen in $\texttt{l}_2$.

```
split   :: [α] -> ([α],[α])
split xs = nsplit ((length xs) 'div' 2) xs ([],[])

nsplit                      :: Integer -> [α] -> ([α],[α]) -> ([α],[α])
nsplit 0     xs     (l1,_)  = (l1,xs)
nsplit _     []     (l1,l2) = (l1,l2)
nsplit (n+1) (x:xs) (l1,l2) = nsplit n xs (x:l1,l2)
```

**Besser:** Verstecke Implementationsdetail `nsplit` der Version ① mittels eines `let`-Ausdrucks. Funktion `nsplit` ist nur lokal in `split` sichtbar und anwendbar:

```
split   :: [α] -> ([α],[α])
split xs = let nsplit 0     xs     (l1,_)  = (l1,xs)
               nsplit _     []     (l1,l2) = (l1,l2)
               nsplit (n+1) (x:xs) (l1,l2) = nsplit n xs (x:l1,l2)
           in
               nsplit ((length xs) 'div' 2) xs ([],[])
```

**Beispiel 7.15**

**Implementation von `split`, Idee ②:** Offensichtlicher Nachteil der Version ①: `xs` ist zweimal zu durchlaufen. Daher: Durchlaufe `xs` nur einmal, füge dabei abwechselnd ein Element in $\texttt{l}_1$ oder $\texttt{l}_2$ ein.

```
split2          :: [α] -> ([α],[α])
split2 []        = ([],[])
split2 [x]       = ([x],[])
split2 (x:x':xs) = let (l1,l2) = split2 xs
                   in
                       (x:l1,x':l2)
```

Hier wird u.a. Gebrauch davon gemacht, daß die lokalen Definitionen gegen Patterns $p_i$ (hier: Tupel-Pattern `(l1,l2)`) *gematcht* werden, um das Zwischenergebnis des rekursiven `split2`-Aufrufes zu analysieren.

**Beispiellauf:**

```
> split [1..1000]                                          -- Version ①
([500, 499, ... 1], [501, 502, ... 1000]) :: ([Integer],[Integer])
(31031 reductions, 54943 cells)

> split2 [1..1000]                                         -- Version ②
([1, 3, ... 999], [2, 4, ... 1000]) :: ([Integer],[Integer])
(21524 reductions, 42929 cells)
```

**Beispiel 7.15**

## 7.3.1 `where`

Nur lokal sichtbare Definitionen lassen sich alternativ mit Haskells `where`-Syntax einführen. `where` erweitert die Syntax von **Funktionsdefinitionen und** `case`**-Ausdrücken** ein weiteres Mal (Syntax für `case` entsprechend):

$$
\begin{array}{lll}
\texttt{f} & :: & \alpha_1 \texttt{ -> } \ldots \texttt{ -> } \alpha_k \texttt{ -> } \beta \\
\texttt{f } p_{11} \ldots p_{1k} \texttt{ | } g_{11} & = & e_{11} \\
\quad \texttt{| } g_{12} & = & e_{12} \\
\quad \texttt{| } \ldots & = & \ldots \\
\quad \texttt{where} & d_{11} & \\
& d_{12} & \\
& \vdots & \\
\texttt{f } p_{n1} \ldots p_{nk} \texttt{ | } g_{n1} & = & e_{n1} \\
\quad \texttt{| } g_{n2} & = & e_{n2} \\
\quad \texttt{| } \ldots & = & \ldots \\
\quad \texttt{where} & d_{n1} & \\
& d_{n2} & \\
& \vdots &
\end{array}
$$

Die $d_{ij}$ sind jeweils in den Guards $g_{ik}$ und Bodies $e_{ik}$ $(k = 1 \ldots)$ **des gesamten Definitionszweiges** $i$ sichtbar. Dies läßt sich mit `let`-Ausdrücken nicht formulieren. Für die lokalen Definitionen $d_{ij}$ gelten die zuvor bei `let` erklärten Vereinbarungen.

Die lokalen Definitionen sind in allen Guards und rechten Seiten eines Definitionszweiges von `f` sichtbar:

```
f x y | y > z          = ... z ...
      | y == z         = ... z ...
      | otherwise      = ... z ...
      where z = x*x
```

**Beispiel 7.16**

Eulers Algorithmus zur Bestimmung des größten gemeinsamen Teilers (ggT):

```
ggT    :: Integer -> Integer -> Integer
ggT x y = ggT' (abs x) (abs y)
        where
          ggT' x 0 = x
          ggT' x y = ggT y (x `rem` y)

          x `rem` y = x - y * (x `div` y)
```

 Mit `let` werden **Ausdrücke** konstruiert, dagegen gehört `where` zur **Syntax der Funktionsdefinition bzw.** `case`.

## Beispiel 7.17

Mittels `where` und `let` formulieren wir unsere Mergesort-Implementierung nun endgültig:

```
mergesort       :: (α -> α -> Bool) -> [α] -> [α]
                -- Divide-and-Conquer Sortierung einer Liste (easy split)
mergesort lt []  = []
mergesort lt [x] = [x]
mergesort lt xs  = let (l1,l2) = split xs
                   in
                       merge lt (mergesort lt l1) (mergesort lt l2)
                 where
                       -- splitte eine Liste in zwei gleich lange Teile
                       split []        = ([],[])
                       split [x]       = ([x],[])
                       split (x:x':xs) = let (l1,l2) = split xs
                                         in
                                             (x:l1,x':l2)

                       -- mische zwei sortierte Listen
                       merge lt []        ys                    = ys
                       merge lt xs        []                    = xs
                       merge lt l1@(x:xs) l2@(y:ys) | x `lt` y  = x : merge lt xs l2
                                                    | otherwise = y : merge lt l1 ys
```

## 7.4 Layout (2-dimensionale Syntax)

Haskells Syntax verzichtet auf Separator- oder Terminator-Symbole wie ';', um bspw. einzelne Deklarationen voneinander abzugrenzen. Trotzdem analysiert Haskells Parser etwa den Ausdruck

```
let y   = a * b
    f x = (x+y)/y
in
    f c + f d
```

eindeutig wie erwartet (der `let`-Ausdruck definiert den Wert `y` und die Funktion `f`) und *nicht* als

```
let y   = a * b f
    x   = (x+y)/y
in
    f c + f d
```

Haskell erreicht dies durch eine **2-dimensionale Syntax** (*layout*): die Einrückungen (Spalte im Quelltext) der einzelnen Deklarationen hinter dem Schlüsselwort `let` wird zur Auflösung von Mehrdeutigkeiten herangezogen. Das *layout* des Quelltextes ist relevant jeweils in Funktionsdefinitionen und nach den Schlüsselworten

| `let` | `where` | `of` | `do` |
|---|---|---|---|
| (lokale Dekl.) | (lokale Dekl.) | (`case`-Alternativen) | (monadische Seq.) |

**Layout.** Haskells *layout* wird durch zwei einfache Vereinbarungen ("Abseits-Regel", *off-side rule*) definiert: Das nächste Token *nach* einem `let`, `where`, `of` oder `do` definiert die obere linke Ecke einer Box.

```
ggT x y = ggT' (abs x) (abs y)
        where ┌──────────────────────────────
              │ggT' x 0 = x
              │ggT' x y =
              │         ggT (x `rem` y)
              │x `rem` y = x - y * (x `div` y)
```

Das erste Token, das *links* von der Box im Abseits steht, schließt die Box (hier: `kgV`):

```
ggT x y = ggT' (abs x) (abs y)
        where ┌──────────────────────────────
              │ggT' x 0 = x
              │ggT' x y =
              │         ggT (x `rem` y)
              │x `rem` y = x - y * (x `div` y)
              └──────────────────────────────
kgV x y = ...
```

Bevor der Parser den eigentlichen Quelltext sieht, formuliert Haskell das *layout* in eine alternative Syntax für `let`, `where`, `case`...`of` und `do` um, die Deklarationen/Alternativen explizit gruppiert (mittels $\{\cdot\}$) und voneinander trennt (mittels ';'). `let` besitzt bspw. die alternative Syntax:

$$\texttt{let \{ } d_1 \texttt{ ; } d_2 \texttt{ ; ... ; } d_n \texttt{ [;] \} in } e$$

## Layout $\rightarrow$ explizite Syntax

① Vor der Box wird eine öffnende Klammer { eingefügt,

② hinter der Box wird eine schließende Klammer } eingefügt,

③ vor einer Zeile, die direkt an der linken Box-Grenze startet, wird ein Semikolon ; eingefügt.
(eine neue Deklaration/Alternative beginnt)

Die explizite Syntax für das ggT-Beispiel lautet daher:

```
ggT x y = ggT' (abs x) (abs y)
        where {ggT' x 0 = x
              ;ggT' x y =
                        ggT (x `rem` y)
              ;x `rem` y = x - y * (x `div` y)
}kgV x y = ...
```

Die Mehrdeutigkeit des ersten Beispiels wird wie folgt aufgelöst (Box und explizite Syntax):

```
 let {y = a * b
     ;f x = (x+y)/y
 }in
     f c + f d
```

- Die vorhergenden Vereinbarungen zum *layout* erlauben die explizite Nutzung von Semikola ';' um Deklarationen innerhalb einer Zeile zu trennen. Erlaubt wäre beispielsweise

$$\begin{array}{ll} \texttt{let} & d_1 \ \texttt{;} \ d_2 \ \texttt{;} \ d_3 \\ & d_4 \ \texttt{;} \ d_5 \\ \texttt{in} & e \end{array}$$

- Sobald der Programmierer von sich aus Klammern { · } setzt, ist *layout* außer Kraft gesetzt.
- Der Parser fügt automatisch ein schließende Klammer } ein, wenn so ein Syntaxfehler vermieden werden kann. Beispiel:

`let x = 3 in x+x` wird expandiert zu `let {x = 3 }in x+x`

- Kommentare (`--...` und `{-...-}`) werden bei der Bestimmung des Abseits-Tokens *nicht* beachtet.
- Die **Reichweite einer Funktionsdefinition** wird ebenfalls mittels einer *layout*-Box erklärt:

```
split xs = nsplit ...
⋮                    (alle Zeilen, die rechts von der Boxgrenze beginnen, gehören zur Definition von split)
```

```
nsplit 0 xs (l1,_) = ...
⋮                    (ein Token direkt an der linken Grenze startet eine neue Box)
```

# Referenzen

Paul Hudak, John Peterson und Joseph H. Fasel (1997). *A Gentle Introduction to Haskell*. Yale University, University of California. http://haskell.org/tutorial/.

John Hughes und Simon L. Peyton Jones (editors) (1999). *Haskell 98: A Non-strict, Purely Functional Language*. http://haskell.org/definition/.

Simon Thompson (1997). *Haskell: the Craft of Functional Programming*. Addison-Wesley.

# 8 Listenverarbeitung

Funktionen, die Listen (Werte des Typs $[\alpha]$ für einen Typ $\alpha$) als Argumente besitzen, orientieren sich oft an der rekursiven Struktur der Listen (s. Abschnitt 6.3). Pattern Matching hilft, folgende Fälle zu unterscheiden:

① **(Rekursionsabbruch)** das Argument ist die leere Liste `[]` :: $[\alpha]$, oder
② **(Rekursion)** das Argument ist nicht leer, sondern der Form $x$`:`*xs* mit $x$ :: $\alpha$ und *xs* :: $[\alpha]$

Die allgemeine Struktur einer listenverarbeitenden Funktion $f$ hat daher oft folgende Form ($z$ :: $\beta, c$ :: $\gamma$ `->` $\delta$ `->` $\beta, h$ :: $\alpha$ `->` $\gamma, t$ :: $[\alpha]$ `->` $\delta$ seien Haskell-Ausdrücke):

$$\begin{array}{rcl} f & :: & [\alpha] \texttt{ -> } \beta \\ f\ [] & = & z \\ f\ (x\texttt{:}xs) & = & c\ (h\ x)\ (t\ xs) \end{array}$$

wobei $z$ das Ergebnis bei Rekursionsabbruch darstellt, $h$ bzw. $t$ auf Kopf bzw. Rest des Arguments im Rekursionsfall ② angewandt werden, während $c$ das Ergebnis beider Aufrufe kombiniert. $t$ ruft dabei typischerweise $f$ selbst rekursiv auf.

**Bemerkung:** Wir werden gleich die höherwertige Funktion `fold` einführen, die das obige Rekursionsmuster allgemein implementiert (Abschnitt 8.1).

**Beispiel 8.1**

Die Summationsfunktion `sum :: [Integer] -> Integer`

```
sum []     = 0
sum (x:xs) = x + sum xs
```

erhält man also mit $z = \texttt{0}, h = \texttt{id}, t = \texttt{sum}$ und $c = \texttt{(+)}$. Die Funktion `id :: α -> α` ist Teil der *standard prelude* und hat die Definition `\x -> x` (Identitätsfunktion).

**Beispiel 8.2**

Mit $z = \texttt{[]}, h = \texttt{f}, t = \texttt{map f}$ und $c = \texttt{(:)}$ gewinnt man die Standardfunktion `map :: (α -> β) -> [α] -> [β]`, die ihr funktionswertiges Argument `f` auf jedes Element der Argumentliste anwendet:

```
map f []     = []
map f (x:xs) = f x : map f xs

> map (+1) [1..10]
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

**Frage:** Welche Funktion $f$ erhält man mittels $z = \texttt{False}, h = \texttt{(e ==)}, t = f\ \texttt{e}$ und $c = \texttt{(||)}$?

Ein großer Teil der Haskell *standard prelude* definiert immer wieder benötige Funktionen über Listen.

```
head (x:_)          = x
tail (_:xs)         = xs

init [x]            = []
init (x:xs)         = x : init xs

last [x]            = x
last (_:xs)         = last xs

length []           = 0
length (_:xs)       = 1 + length xs

(x:_)  !! 0         = x
(_:xs) !! n | n>0   = xs !! (n-1)

take 0 _            = []
take _ []           = []
take n (x:xs) | n>0 = x : take (n-1) xs

reverse []          = []
reverse (x:xs)      = reverse xs ++ [x]
```

```
drop 0 xs                = xs
drop _ []                = []
drop n (_:xs) | n>0      = drop (n-1) xs

zipWith f (x:xs) (y:ys)  = f x y : zipWith f xs ys
zipWith _ _      _       = []

filter _ []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs

[]     ++ ys             = ys
(x:xs) ++ ys             = x : (xs ++ ys)

concat []                = []
concat (xs:xss)          = xs ++ concat xss

dropWhile _ []           = []
dropWhile p xs@(x:xs')
            | p x        = dropWhile p xs'
            | otherwise = xs
```

(Typen dieser Funktionen? Funktionsweise?)

## 8.1 `fold`

Das im vorigen Abschnitt besprochene Rekursionsschema ist so zentral in der Listenverabeitung, daß die *standard prelude* zwei höherwertige Funktionen, `foldr` (*fold right*) und `foldl` (*fold left*), bereitstellt, die Varianten dieses Schemas implementieren.

### 8.1.1 `foldr`

Informell gilt ($\oplus$ sei ein Infix-Operator des Typs $\alpha$ -> $\beta$ -> $\beta$):

$$\texttt{foldr (}\oplus\texttt{)}\ z\ [x_1,x_2,\ldots,x_n] \quad = \quad x_1 \oplus (x_2 \oplus (\cdots (x_n \oplus z)\cdots))$$

(Die Klammerung ist rechts-assoziativ, was den Namen *fold right* erklärt. Falls $\oplus$ assoziativ ist, können die Klammern weggelassen werden.)

Damit hat `foldr` den Typ $(\alpha$ -> $\beta$ -> $\beta)$ -> $\beta$ -> $[\alpha]$ -> $\beta$. `foldr` „reduziert“ die Argumentliste des Typs $[\alpha]$ zu einem Wert des Typs $\beta$ (in der Literatur ist `fold` auch als *reduce* bekannt).

In der *standard prelude* ist `foldr` definiert durch

```
foldr              :: (α -> β -> β) -> β -> [α] -> β
foldr (⊕) z []     = z
foldr (⊕) z (x:xs) = x ⊕ (foldr (⊕) z xs)
```

Alternativ läßt sich der Effekt von `foldr` damit auch wie folgt illustrieren:

$$\begin{array}{rl} \texttt{foldr}\ \oplus\ z & (x_1\ \texttt{:}\ (x_2\ \texttt{:}\ (\ldots(\ x_n\ \texttt{:}\ \texttt{[]}\ )\ldots))) \\ & \downarrow\ \downarrow\ \downarrow\ \downarrow \qquad \downarrow\ \downarrow\ \downarrow \\ = & (x_1 \oplus (x_2 \oplus (\ldots(\ x_n \oplus\ z\ )\ldots))) \end{array}$$

`foldr` ersetzt also alle Vorkommen der Listen-Konstruktoren `(:)` und `[]` durch $\oplus$ bzw. $z$.

Die `foldr`-Operationen ist so generell, daß sich unzählige Funktionen darauf zurückführen lassen.

**Beispiel 8.3**

```
sum     = foldr (+) 0
product = foldr (*) 1
concat  = foldr (++) []
and     = foldr (&&) True
or      = foldr (||) False
```

All diesen Beispielen ist gemeinsam, daß $\oplus$ assoziativ ist und die Identität $z$ besitzt, d.h.

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$
$$x \oplus z = x = z \oplus x$$

Damit formen $\oplus$ und $z$ einen **Monoid** (Halbgruppe mit Einselement).

Weitere Beispiele, in denen $\oplus$ und $z$ *keinen* Monoid formen, sind:

**Beispiel 8.4**

```
filter p     =  foldr (\x xs -> if p x then x:xs else xs) []
length       =  foldr (\_ n -> 1+n) 0
reverse      =  foldr (\x xs -> xs ++ [x]) []
takeWhile p  =  foldr (#) []
                where
                    x # xs | p x       = x:xs
                           | otherwise = []
```

Damit wird `takeWhile (<3) [1..4]` etwa wie folgt reduziert:

```
takeWhile (<3) [1..4] = foldr (#) [] (1 : (2 : (3 : (4 : []))))
                      =  1 # (2 # (3 # (4 # [])))
                      =  1 # (2 # (3 # []))
                      =  1 # (2 # [])
                      =  1 # (2 : [])
                      =  1 : (2 : [])
                      =  [1,2]
```

## 8.1.2 `foldl`

Die `fold`-Variante *fold left* klammert die Listenelemente während der Reduktion nach links und ist informell definiert als

$$\texttt{foldl (}\otimes\texttt{) }z\texttt{ [}x_1\texttt{,}x_2\texttt{,...,}x_n\texttt{]} \quad = \quad (\cdots((z \otimes x_1) \otimes x_2)\cdots) \otimes x_n$$

Die Haskell *standard prelude* definiert `foldl` via

```
foldl             :: (α -> β -> α) -> α -> [β] -> α
foldl (⊗) z []     = z
foldl (⊗) z (x:xs) = foldl (⊗) (z ⊗ x) xs
```

Falls $\otimes$ assoziativ sein sollte, gilt $\alpha = \beta$. Damit haben `foldl` und `foldr` in diesem Fall denselben Typ. Es gilt sogar das

**Satz 8.1 (1. Dualitätstheorem)**
Falls $\otimes$ und $z$ einen Monoid bilden, dann gilt

$$\texttt{foldr (}\otimes\texttt{) }z \quad = \quad \texttt{foldl (}\otimes\texttt{) }z$$

Die nahe Verwandtschaft zwischen `foldr` und `foldl` wird zusätzlich durch das 2. Dualitätstheorem beleuchtet.

**Satz 8.2 (2. Dualitätstheorem)**
Falls

$$\begin{aligned} x \oplus (y \otimes z) &= (x \oplus y) \otimes z \quad \text{und} \\ x \oplus z &= z \otimes x \end{aligned}$$

dann gilt

$$\texttt{foldr (}\oplus\texttt{)}\ z = \texttt{foldl (}\otimes\texttt{)}\ z$$

**Beispiel 8.5**

```
length  = foldl (\n _ -> 1+n) 0
reverse = foldl (\xs x -> [x] ++ xs) []
```

Nach dem 2. Dualitätstheorem sind diese und die vorher gezeigten `foldr`-basierten Definitionen äquivalent (nachrechnen!) und sogar effizienter (siehe Übung).

Schließlich gilt das

**Satz 8.3 (3. Dualitätstheorem)**
Sei `flip` $f\ x\ y = f\ y\ x$, dann gilt

$$\texttt{foldr (}\oplus\texttt{)}\ z\ \mathit{xs} = \texttt{foldl (flip (}\oplus\texttt{))}\ z\ \texttt{(reverse}\ \mathit{xs}\texttt{)}$$

**Beispiel 8.6**

Ein praktisches Beispiel für den Einsatz von `foldl` ist die Funktion `pack`, die eine Liste von Ziffern $[x_{n-1}, x_{n-2}, \ldots, x_0]$ $(x_i \in \{0 \ldots 9\})$ in den durch sie „dargestellten“ Wert transformiert:

$$\sum_{k=0}^{n-1} x_k \cdot 10^k$$

```
pack   :: [Integer] -> Integer
pack xs = foldl (|*|) 0 xs
          where
               n |*| x = 10 * n + x
```

**Bemerkung ①.** Die Ersetzung der Konstruktoren eines Datentyps (hier *cons* `(:)` und *nil* `[]`) durch Operatoren bzw. Werte ist ein Prinzip, das sich **auch für andere konstruierte Datentypen sinnvoll** anwenden läßt. Darauf kommt Kapitel 9 zurück.

**Bemerkung ②.** Auf der Basis von `fold` lassen sich ausdrucksstarke funktionale Subsprachen definieren. Am Datenbanklehrstuhl untersuchen wir **Anfragesprachen** für DBMS, die komplett auf `fold` basieren.

## 8.2 Effizienz

Die Anzahl der Reduktionen, die benötigt werden, um ein Programm (einen Ausdruck) in seine Normalform zu überführen und dadurch auszuwerten, ist in FPLs ein sinnvolles Maß für die Komplexität einer Berechnung.

**Beispiel 8.7**

Die Länge der ersten Argumentliste bestimmt die Anzahl der Reduktionen der Listen-Konkatenation ++ (++.$n$ steht für die Zeile $n$ in der Definition von ++):

$$\begin{array}{rcll} \texttt{[3,2] ++ [1]} & = & \texttt{3:([2] ++ [1])} & (\texttt{++}.2) \\ & = & \texttt{3:(2:([] ++ [1]))} & (\texttt{++}.2) \\ & = & \texttt{3:(2:[1])} & (\texttt{++}.1) \\ & = & \texttt{[3,2,1]} & \end{array}$$

(Die letzte Zeile ist *keine* Reduktion – derselbe Wert wurde lediglich in die Klammernotation für Listen übersetzt.)

**Generell**: für die Auswertung von *xs* ++ $ys$ mit `length(`*xs*`)` $\twoheadrightarrow n$ werden $n$ Reduktionen via (++.2), gefolgt von einer Reduktion via (++.1) benötigt.

Ganz ähnliche Überlegungen kann man für `reverse` anstellen, das in seiner Definition auf ++ zurückgreift:

**Beispiel 8.8**

Reduziere `reverse [1,2,3]`:

$$
\begin{array}{lcll}
\texttt{reverse [1,2,3]} & = & \texttt{reverse [2,3] ++ [1]} & (\texttt{reverse}.2) \\
& = & \texttt{(reverse [3] ++ [2]) ++ [1]} & (\texttt{reverse}.2) \\
& = & \texttt{((reverse [] ++ [3]) ++ [2]) ++ [1]} & (\texttt{reverse}.2) \\
& = & \texttt{(([] ++ [3]) ++ [2]) ++ [1]} & (\texttt{reverse}.1) \\
& = & \texttt{([3] ++ [2]) ++ [1]} & (\texttt{++}.1) \\
& = & \texttt{[3,2] ++ [1]} & (\texttt{++}.2, \texttt{++}.1) \\
& = & \texttt{[3,2,1]} & (\texttt{++}.2, \texttt{++}.2, \texttt{++}.1)
\end{array}
$$

Gilt `length(`*xs*`)` $\twoheadrightarrow n$, dann benötigt (`reverse` *xs*) $n$ Reduktionen via (`reverse`.2), gefolgt von einer Reduktion via (`reverse`.1), gefolgt von

$$1 + 2 + \cdots + n = \frac{n(n-1)}{2}$$

Reduktionen, um mittels ++ die Konkatenationen auszuführen. Damit ist die Anzahl der Reduktionen grob proportional zu $n^2$.

Das Reversieren einer Liste läßt sich aber durchaus in linearer Zeit (proportional zur Listenlänge $n$) durchführen. Die Funktion `rev` realisiert dies mit Hilfe der Funktion `shunt`:

```
rev   :: [α] -> [α]
rev xs = shunt [] xs
         where
              shunt          :: [α] -> [α] -> [α]
              shunt ys []     = ys
              shunt ys (x:xs) = shunt (x:ys) xs
```

Tatsächlich reversiert `shunt` $ys$ *xs* nicht nur die Liste *xs*, sondern konkateniert diese zusätzlich auch noch mit $ys$. Dies ist der Schlüssel zur Performance von `shunt`, denn in der „naiven“ Definition von `reverse` gilt `reverse` $(x$`:`*xs*$)$ $\rightarrow\!\triangleright$ `reverse` *xs* `++` $[x]$.

**Beispiel 8.9**

`rev` *xs* benötigt proportional zu (`length` *xs*) Reduktionen:

```
rev [1,2,3]  =  shunt [] [1,2,3]    (rev.1)
             =  shunt [1] [2,3]     (shunt.2)
             =  shunt [2,1] [3]     (shunt.2)
             =  shunt [3,2,1] []    (shunt.2)
             =  [3,2,1]             (shunt.1)
```

## 8.3 Induktion über Listen

Es gilt für alle *xs* :: $[\alpha]$

$$\texttt{rev}\ xs \quad = \quad \texttt{reverse}\ xs$$

Dank referentieller Transparenz kann man Behauptungen wie die obige relativ einfach *beweisen*. Beweise über listenverarbeitende Funktionen können häufig mittels **Induktion über Listen** (analog zur Induktionsbeweisen für Behauptungen über Elemente aus $\mathbb{N}$) geführt werden:

① **Induktionsverankerung**: leere Liste `[]`,

② **Induktionsschritt**: von *xs* zu $x$ : *xs*.

**Beispiel 8.10**

Es gilt `rev` *xs* = `reverse` *xs* für alle Listen *xs* :: $[\alpha]$. Da `rev` *xs* $\rightarrowtriangle$ `shunt []` *xs* und aufgrund der obigen Bemerkungen zu `shunt` beweisen wir hier sogar die allgemeinere Behauptung:

$$\texttt{shunt}\ ys\ xs \quad = \quad (\texttt{reverse}\ xs)\ \texttt{++}\ ys \quad \text{, für alle } ys :: [\alpha]$$

Wir verwenden Listeninduktion über *xs*.

**Induktionsverankerung** `[]`:

$$\begin{array}{lcll} \texttt{shunt}\ ys\ \texttt{[]} & = & ys & (\texttt{shunt}.1) \\ & = & \texttt{[]}\ \texttt{++}\ ys & (\texttt{++}.1) \\ & = & \texttt{reverse}\ \texttt{[]}\ \texttt{++}\ ys & (\texttt{reverse}.1) \end{array}$$

**Induktionsschritt** $(x\texttt{:}xs)$:

$$\begin{array}{lcll} \texttt{shunt}\ ys\ (x\texttt{:}xs) & = & \texttt{shunt}\ (x\texttt{:}ys)\ xs & (\texttt{shunt}.2) \\ & = & \texttt{reverse}\ xs\ \texttt{++}\ (x\texttt{:}ys) & (\text{Hypothese}) \\ & = & \texttt{reverse}\ xs\ \texttt{++}\ (\texttt{[}x\texttt{]}\ \texttt{++}\ ys) & (\texttt{++}.2, \texttt{++}.1) \\ & = & (\texttt{reverse}\ xs\ \texttt{++}\ \texttt{[}x\texttt{]})\ \texttt{++}\ ys & (\texttt{++}\ \text{assoziativ}) \\ & = & \texttt{reverse}\ (x\texttt{:}xs)\ \texttt{++}\ ys & (\texttt{reverse}.2) \end{array}$$

Die hier verwendete Annahme über die Assoziativität von ++ müßten wir prinzipiell ebenfalls noch beweisen:

$$xs\ \texttt{++}\ (ys\ \texttt{++}\ zs) = (xs\ \texttt{++}\ ys)\ \texttt{++}\ zs$$

Auch hier führt Listeninduktion über den Parameter zum Ziel, über den die Rekursion der betrachteten Funktion (hier: ++) formuliert ist, hier also *xs*. **Übung.**

**Beispiel 8.10**

## 8.4 Programm-Synthese

Bei der **Beweisführung über Programme** werden Eigenschaften eines gegebenen Programms Schritt für Schritt nachvollzogen und dadurch bewiesen (s. den Beweis zur Äquivalenz von `rev` und `reverse` aus dem vorigen Abschnitt).

**Programm-Synthese** kehrt dieses Prinzip um:

- Gegeben ist eine **formale Spezifikation** eines Problems,
- gesucht ist ein problemlösendes Programm, das durch schrittweise **Umformung der Spezifikation** gewonnen (**synthetisiert**) wird.

Wenn die Transformationen diszipliniert vorgenommen werden, kann die Synthese als Beweis dafür gelesen werden, daß das Programm die Spezifikation erfüllt (*der Traum aller Software-Ingenieure*).

**Beispiel 8.11**
Die Funktion `init` der *standard prelude* bestimmt das initiale Segment ihres Listenargumentes, also etwa `init [1..10]` $\rightarrowtriangle$ `[1, 2, 3, 4, 5, 6, 7, 8, 9]`. Damit wäre eine naheliegende **Spezifikation** für `init` die folgende:

$$\texttt{init}\ xs \quad = \quad \texttt{take}\ (\texttt{length}\ xs\ \texttt{-}\ 1)\ xs \quad ,\ xs\ \texttt{/=}\ \texttt{[]}$$

*„Nimm alle Elemente von xs, aber nicht das letzte Element“*

Die Synthese versucht eine effizientere Variante von `init` abzuleiten (die Spezifikation wäre ja prinzipiell schon ausführbar, traversiert *xs* zur Berechnung des Ergebnisses aber zweimal).
Für die Synthese instantiieren wir *xs* ① mit $[x]$ und ② mit $(x : x' : xs)$. Jede nichtleere Liste besitzt die eine oder die andere Form.

**Fall** ① $[x]$:

$$\begin{array}{rcll} \texttt{init}\ [x] & = & \texttt{take}\ (\texttt{length}\ [x]\ \texttt{-}\ 1)\ [x] & \text{(Instantiierung)} \\ & = & \texttt{take}\ 0\ [x] & (\texttt{length}, \text{Arithmetik}) \\ & = & [] & (\texttt{take}.1) \end{array}$$

**Fall** ② $(x : x' : xs)$:

$$\begin{array}{rcll} \texttt{init}\ (x : x' : xs) & = & \texttt{take}\ (\texttt{length}\ (x : x' : xs)\ \texttt{-}\ 1)\ (x : x' : xs) & \text{(Instantiierung)} \\ & = & \texttt{take}\ (\texttt{length}\ xs\ \texttt{+}\ 1)\ (x : x' : xs) & (\texttt{length}, \text{Arithmetik}) \\ & = & x\ \texttt{:}\ \texttt{take}\ (\texttt{length}\ xs)\ (x' : xs) & (\texttt{take}.3) \\ & = & x\ \texttt{:}\ \texttt{take}\ (\texttt{length}\ (x' : xs)\ \texttt{-}\ 1)\ (x' : xs) & (\texttt{length}.2, \text{Arithmetik}) \\ & = & x\ \texttt{:}\ \texttt{init}\ (x' : xs) & \end{array}$$

**Beispiel 8.11**

Zusammenfassen der beiden so erhaltenenen Gleichungen ergibt

```
init [x]       = []
init (x:x':xs) = x : init (x':xs)
```

Dies ist die effiziente rekursive Definition von `init`, die so ebenfalls in der *standard prelude* zu finden ist.

**Beispiel 8.11**

## 8.5 List Comprehensions

**List comprehensions** sind vor allem in modernen FPLs als eine alternative Notation für Operationen auf Listen verbreitet[7].

Die Notation von Objekten mittels *comprehensions* ist vor allem in der Mathematik (Mengenlehre) üblich.

*List comprehensions* erweitern die Ausdruckskraft der Sprache *nicht* (wir werden in Abschnitt 8.5.2 eine Abbildung auf den Haskell-Kern besprechen), aber erlauben oft eine kompakte, leicht lesbare und elegante Notation von Listenoperationen.

[7] **Miranda**™ (Dave Turner, 1976) sah als erste FPL *list comprehensions* syntaktisch vor.

**Beispiel 8.12**

Die Menge aller natürlichen geraden Zahlen kann durch eine *set comprehension* kompakt notiert werden:

$$\{n \mid n \in \mathbb{N}, n \bmod 2 = 0\}$$

Eine entsprechende *list comprehension* (die unendliche Liste aller geraden Zahlen $\geq 1$) wird syntaktisch ganz ähnlich notiert:

```
[ n | n <- [1 .. ], n `mod` 2 == 0 ]
```

**Beispiel 8.13**

Die Standardfunktionen `map` und `filter` sind mittels *list comprehensions* ohne die sonst notwendige Rekursion zu formulieren:

```
map     :: (α -> β) -> [α] -> [β]
map f xs = [ f x | x <- xs ]

filter     :: (α -> Bool) -> [α] -> [α]
filter p xs = [ x | x <- xs, p x ]
```

Die allgemeine Form einer *list comprehension* ist

$$\texttt{[}\ e\ \texttt{|}\ q_1\texttt{,}q_2\texttt{,}\ldots\texttt{,}q_n\ \texttt{]} \quad n \geq 1$$

wobei $e$, der **Kopf**, ein beliebiger Ausdruck ist und die $q_i$, die **Qualifier**, eine von drei Formen besitzen:

① **Generatoren** der Form $p_i$ `<-` $e_i$, wobei $p_i$ ein Pattern und $e_i$ ein Ausdrucks des Typs `[`$\alpha_i$`]` ist,

② **Prädikate** (Ausdrücke des Ergebnistyps `Bool`) und

③ **lokale Bindungen** der Form `let {` $p_{i1}$ `=` $e_{i1}$ `;` $p_{i2}$ `=` $e_{i2}$ `...` `}` ($p_{ij}$ Patterns, $e_{ij}$ beliebige Ausdrücke).

**Semantik:**

- Ein **Generator** $q_i = p_i$ `<-` $e_i$ *matched* das Pattern $p_i$ sequentiell gegen die Listenelemente von $e_i$. Die durch einen erfolgreichen *match* gebundenen Variablen sind in den Qualifiern $q_{i+1} \ldots q_n$ sichtbar, die unter all diesen Bindungen ausgewertet werden.
- Eine Bindung wird solange propagiert, bis ein **Prädikat** unter ihr zu `False` evaluiert wird. Der Kopf $e$ wird unter allen Bindungen ausgewertet, die alle Prädikate passieren konnten.
- Eine **lokale Bindung** $q_i$ kann Variablen an Werte binden, die in $q_{i+1} \ldots q_n$ sichtbar sind.

Nach der Semantik wird in [ $e$ | $p_1$ <- $e_1$ , $p_2$ <- $e_2$] über die **Domain** $e_1$ des ersten Generators $p_1$ <- $e_1$ zuerst iteriert, dann über $e_2$. Dies trifft die Intuition der aus der Mengenlehre bekannten *set comprehension*:

$$\texttt{[ (x,y) | x <- [}x_1\texttt{,}x_2\texttt{], y <- [}y_1\texttt{,}y_2\texttt{] ]} \quad \rightarrowtriangle \quad \texttt{[(}x_1\texttt{,}y_1\texttt{),(}x_1\texttt{,}y_2\texttt{),(}x_2\texttt{,}y_1\texttt{),(}x_2\texttt{,}y_2\texttt{)]}$$

**Beispiel 8.14**

So läßt sich bspw. ein „Join“ zwischen zwei Listen bzgl. eines Prädikates `p` sehr einfach definieren:

```
join          :: (α -> β -> γ) -> (α -> β -> Bool) -> [α] -> [β] -> [γ]
join f p xs ys = [ f x y | x <- xs, y <- ys, p x y ]
```

Den „klassischen relationalen Join“ $R_1 \bowtie_{\#1=\#1} R_2$ ($R_i$ binäre Relationen) erhält man dann durch

```
rjoin = join (\(_,x2) (_,y2) -> (x2,y2)) (\(x1,_) (y1,_) -> x1==y1)
```

```
rjoin [(1,'a'),(2,'b'),(1,'c')] [(1,0),(2,10),(3,100)]  ⇀
```

| [ #1 | #2 |
|---|---|
| ('a' | 0) |
| ('b' | 10) |
| ('c' | 0) ] |

**Beispiel 8.15**

Eine elegante (aber ineffiziente) Variante des Algorithmus Quicksort läßt sich mittels *list comprehensions* als 2-Zeiler codieren:

```
qsort             :: (α -> α -> Bool) -> [α] -> [α]
qsort _    []      = []
qsort (<<) (x:xs) = qsort (<<) [ y | y <- xs, y << x ]
                    ++ [x] ++
                    qsort (<<) [ y | y <- xs, not (y << x) ]
```

**Beachte:** In der *split*-Phase dieser Implementation wird die Liste `xs` jeweils (unnötigerweise) zweimal durchlaufen.

**Beispiel 8.16**

Matrizen über einem Typ $\alpha$ können durch Listen von Zeilenvektoren (Listen) des Typs $\alpha$ repräsentiert werden. Folgende Funktion bestimmt den ersten Spaltenvektor einer Matrix:

```
firstcol  :: [[α]] -> [α]
firstcol m = [ e | (e:_) <- m ]
```

`firstcol` nutzt die Möglichkeit, in Generatoren Pattern zu spezifizieren.

**Beispiel 8.17**

Alle Permutationen der Elemente einer Liste `xs` kann man wie folgt bestimmen:

① Die leere Liste `[]` hat sich selbst als einzige Permutation.

② Wenn `xs` nicht leer ist, wählen wir ein Element `a` aus `xs` und stellen `a` den Permutationen der Liste `(xs\\[a])` vor.

Der Operator `(\\)` implementiert Listendifferenz: `xs\\ys` ist die Liste `xs`, in der die Vorkommen der Elemente von `ys` entfernt wurden, etwa: `[1,2,1,2,3] \\ [2,3]` ↠ `[1,1,2]`.

```
perms   :: [α] -> [[α]]
perms [] = [[]]
perms xs = [ a:p | a <- xs, p <- perms (xs \\ [a]) ]

> perms [2, 3]
[[2, 3], [3, 2]]
```

**Beispiel 8.18**

Berechne alle Pythagoräischen Dreiecke mit Seitenlänge $\leq$ `n`:

```
pyth n = [ (a,b,c) | a <- [1..n], b <- [1..n-a], c <- [1..n-a-b],
                     a^2 + b^2 == c^2 ]
```

**Beispiel 8.19**

**Frage:** Was berechnet die folgende Funktion `bar`? Wie lautet ihr Typ?

```
bar xs = [ x | [x] <- xs ]
```

## 8.5.1 Operationale Semantik für list comprehensions

Die Semantik der *list comprehensions*, welche wir bisher eher durch „*hand-waving*“ erklärt haben, läßt sich – ganz ähnlich wie bei der $\beta$-Reduktion des $\lambda$-Kalküls – durch **Reduktionsregeln** auch formal erklären.

Sei $e$ ein Haskell-Ausdruck, $qs$ eine Sequenz von Qualifiern. Die folgenden Regeln reduzieren die Liste der Qualifier jeweils um den ersten Qualifier:

$$
\begin{array}{rcll}
\texttt{[}\ e\ \texttt{|}\ v\ \texttt{<-}\ \texttt{[]}\texttt{,}\ qs\ \texttt{]} & \rightarrow & \texttt{[]} & ① \\
\texttt{[}\ e\ \texttt{|}\ v\ \texttt{<-}\ \texttt{(}x\texttt{:}xs\texttt{)}\texttt{,}\ qs\ \texttt{]} & \rightarrow & \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]}[x/v]\ \texttt{++}\ \texttt{[}\ e\ \texttt{|}\ v\ \texttt{<-}\ xs\texttt{,}\ qs\ \texttt{]} & ② \\
\texttt{[}\ e\ \texttt{|}\ \texttt{False}\texttt{,}\ qs\ \texttt{]} & \rightarrow & \texttt{[]} & ③ \\
\texttt{[}\ e\ \texttt{|}\ \texttt{True}\texttt{,}\ qs\ \texttt{]} & \rightarrow & \texttt{[}\ e\ \texttt{|}\ qs\ \texttt{]} & ④ \\
\texttt{[}\ e\ \texttt{|}\ \texttt{]} & \rightarrow & \texttt{[}\ e\ \texttt{]} & ⑤
\end{array}
$$

Die ersten beiden Reduktionsregeln reduzieren einen Generator über einer leeren ① bzw. nichtleeren ② Liste. Regeln ③ und ④ testen Prädikate. Regel ⑤ ist anwendbar, sobald die Liste der Qualifier vollständig reduziert wurde.

**Beispiel 8.20**

Mit diesen Regeln reduziert sich die *list comprehension* `[ x^2 | x <- [1,2,3], odd x ]` wie folgt:

```
                [ x^2 | x <- [1,2,3], odd x ]
①+3×②  →       [ 1^2 | odd 1 ] ++ [ 2^2 | odd 2 ] ++ [ 3^2 | odd 3 ] ++ []
odd     →       [ 1^2 | True ] ++ [ 2^2 | False ] ++ [ 3^2 | True ]
③+④     →       [ 1^2 | ] ++ [] ++ [ 3^2 | ]
⑤       →       [ 1^2 ] ++ [] ++ [ 3^2 ]
        →       [ 1, 9 ]
```

## 8.5.2 Abbildung von list comprehensions auf den Haskell-Kern

Prinzipiell erlaubt das System der eben besprochenen Reduktionsregeln, *list comprehensions* auf in Haskell vordef. Funktionen zurückzuführen.

Hier schlagen wir ein alternatives Schema $\mathbb{T}$ vor, das *list comprehensions* direkt auf den Sprachkern – also einen erweiterten $\lambda$-Kalkül – abbildet. Es basiert auf der Funktion `flatmap`:

```
flatmap    ::  (α -> [β]) -> [α] -> [β]
flatmap f  =   foldr (\x xs -> f x ++ xs) []
```

`flatmap f xs` wendet die listenwertige Funktion `f` auf jedes Element von `xs` an und konkateniert die Ergebnisse.

Damit ist $\mathbb{T}$ einfach spezifiert. Wieder wird die Liste der Qualifier (Regel ①: Generatoren bzw. Regel ②: Prädikate) reduziert, bis Regel ③ den Fall der leeren Qualifierliste behandeln kann.

Sei $e$ ein Ausdruck, $v$ eine Variable, $p$ ein Prädikat und $qs$ wieder eine Liste von Qualifiern.

Übersetzungsschema $\mathbb{T}$:

$$
\begin{array}{rcll}
\mathbb{T}([\ e \mid v \texttt{<-}\ xs, qs\ ]) & = & \texttt{flatmap (\}v \texttt{ -> } \mathbb{T}([\ e \mid qs\ ])) \ \mathbb{T}(xs) & ① \\
\mathbb{T}([\ e \mid p, qs\ ]) & = & \texttt{if } \mathbb{T}(p) \texttt{ then } \mathbb{T}([\ e \mid qs\ ]) \texttt{ else []} & ② \\
\mathbb{T}([\ e \mid\ ]) & = & \mathbb{T}(e)\texttt{:[]} & ③ \\
\mathbb{T}(e) & = & e & ④
\end{array}
$$

**Beispiel 8.21**

Wiederum soll die *list comprehension* `[ x^2 | x <- [1,2,3], odd x ]` als Beispiel dienen. $\mathbb{T}$ übersetzt wie folgt:

$$
\begin{array}{rl}
 & \mathbb{T}(\texttt{[ x\^{}2 | x <- [1,2,3], odd x ]}) \\
\underset{(1)+(4)}{=} & \texttt{flatmap (\textbackslash x -> } \mathbb{T}\texttt{([ x\^{}2 | odd x ])) [1,2,3]} \\
\underset{(2)+(4)}{=} & \texttt{flatmap (\textbackslash x -> if odd x then } \mathbb{T}\texttt{([ x\^{}2 | ]) else []) [1,2,3]} \\
\underset{(3)+(4)}{=} & \texttt{flatmap (\textbackslash x -> if odd x then (x\^{}2):[] else []) [1,2,3]} \\
 & \\
\underset{\texttt{flatmap}}{=} & \texttt{[1\^{}2] ++ [] ++ [3\^{}2] ++ []} \\
\rightarrowtriangle & \texttt{[ 1, 9 ]}
\end{array}
$$

# Referenzen

Richard Bird und Philip Wadler (1998). *Introduction to Functional Programming using Haskell*. Series in Computer Science. Prentice Hall International.

Jeroen Fokker (1995). *Functional Programming*. Department of Computer Science, Utrecht University. http://www.cs.ruu.nl/~jeroen/courses/fp-eng.ps.gz.

Torsten Grust und Marc H. Scholl (1999). *How to Comprehend Queries Functionally*. *Journal of Intelligent Information Systems*, 12(2/3):191–218. Special Issue on Functional Approach to Intelligent Information Systems.

# 9 Algebraische Datentypen

Dieses Kapitel erweitert Haskells Typsystem, das neben Basistypen (`Integer`, `Float`, `Char`, `Bool`, ...) und Typkonstruktoren (`[·]` und `(·)`) auch **algebraische Datentypen** kennt.

- Ganz analog zum Typkonstruktor `[·]`, der die beiden **Konstruktorfunktionen** `(:)` und `[]` einführte, um Werte des Typs $[\alpha]$ zu konstruieren, kann der Programmierer **neue Konstruktoren** definieren, um Werte eines neuen algebraischen Datentyps zu erzeugen.
- Wie bei Listen und Tupeln möglich, können Werte dieser neuen Typen dann mittels **Pattern Matching** wieder analysiert (dekonstruiert) werden.

In der Tat ist der eingebaute Typkonstruktor $[\alpha]$ selbst ein algebraischer Datentyp (s. unten).

## 9.1 Deklaration eines algebraischen Datentyps

Mittels einer `data`-Deklaration wird ein neuer algebraischer Datentyp spezifizert durch:

① den **Namen $T$ des Typkonstruktors** (Identifier beginnend mit Zeichen $\in \{\texttt{A} \ldots \texttt{Z}\}$) und seine Typparameter $\alpha_j$,

② die **Namen $K_i$ der Konstrukturfunktionen** (Identifier beginnend mit Zeichen $\in \{\texttt{A} \ldots \texttt{Z}\}$) und der Typen $\beta_{ik}$, die diese als Parameter erwarten.

Syntax einer `data`-Deklaration ($n \geq 0, m \geq 1, n_i \geq 0$, die $\beta_{ik}$ sind entweder Typbezeichner oder $\beta_{ik} = \alpha_j$):

$$\begin{array}{lcll} \texttt{data}\ T\ \alpha_1\ \alpha_2 \ldots \alpha_n & = & K_1 & \beta_{11} \ldots \beta_{1n_1} \\ & | & K_2 & \beta_{21} \ldots \beta_{2n_2} \\ & \vdots & & \\ & | & K_m & \beta_{m1} \ldots \beta_{mn_m} \end{array}$$

Dieses `data`-Statement deklariert einen **Typkonstruktor** $T$ und $m$ **Konstruktorfunktionen**:

$$K_i :: \beta_{i1} \texttt{ -> } \ldots \texttt{ -> } \beta_{in_i} \texttt{ -> } T\ \alpha_1\ \alpha_2 \ldots \alpha_n$$

**Sonderfälle:**

- $n = 0, n_i = 0$: $\texttt{data}\ T = K_1 \mid K_2 \mid \cdots \mid K_m$
  $T$ ist damit ein reiner **Aufzählungstyp** wie aus vielen Programmiersprachen bekannt (etwa in C: `enum`).
- $m = 1$: $\texttt{data}\ T\ \alpha_1 \ldots \alpha_n = K_1\ \beta_{11} \ldots \beta_{1n_1}$
  $T$ verhält sich damit ähnlich wie der Tupelkonstruktor und wird auch **Produkttyp** genannt. In der Typtheorie oft als $\beta_{11} \times \beta_{12} \times \cdots \times \beta_{1n_1}$ notiert.

In seiner allgemeinsten Form führt die `data`-Deklaration also Alternativen (Typtheorie: Summe) von Produkttypen ein, bezeichnet als **sum-of-product types**.

**Beispiel 9.1**

Der benutzerdefinierte Aufzählungstyp

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

definiert den Typkonstruktor `Weekday` und die Konstruktorfunktionen `Mon` ... `Sun` mit `Mon :: Weekday` .... Funktionen über diesem Typ werden mittels Pattern Matching realisiert:

```
weekend :: Weekday -> Bool
weekend Sat = True
weekend Sun = True
weekend _   = False
```

Der vordefinierte Typ `Bool` ist prinzipiell ein Aufzählungstyp:

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

Dies gilt theoretisch ebenso für die anderen Basisdatentypen in Haskells Typsystem:

```
data Int  = -2^29 | ... | -1 | 0 | 1 | ... | 2^29 - 1    -- Pseudo-Code!
data Char = 'a' | 'b' | ... | 'A' | ... | '1' | ...
```

Bei der Arbeit mit diesen neuen Typen reagiert der Haskell-Interpreter/Compiler merkwürdig:

```
> Mon
ERROR: Cannot find "show" function for:                    ①
*** expression : Mon
*** of type    : Weekday

> Tue == Fri
ERROR: Weekday is not an instance of class "Eq"            ②
```

① Das Haskell-System hat keine Methode `show` für die Ausgabe von Werten des Typs `Weekday` mitgeteilt bekommen. Intuition: Name des Konstruktors $K_i$ benutzen.

② Gleichheit auf den Elementen des Typs ist nicht definiert worden. Intuition: nur Werte die durch **denselben Konstruktor $K_i$ mit identischen Parametern** erzeugt wurden, sind gleich.

Haskell kann diese Intuitionen automatisch zur Verfügung stellen, wenn die `data`-Deklaration durch den Zusatz

```
deriving (Show, Eq)
```

gefolgt wird. Der neue Typ $T$ wird damit automatisch Instanz der Typklasse `Show` aller druckbaren Typen und Instanz der Typklasse `Eq` aller Typen mit Gleichheit `(==)`.

(Der `deriving`-Mechanismus ist genereller und wird in Kapitel 10 näher besprochen.)

Algebraische Datentypen erlauben die Erweiterung eines Typs um einen speziellen Wert, der eingesetzt werden kann, wenn Berechnungen kein sinnvolles oder ein unbekanntes Ergebnis besitzen.

**Beispiel 9.2**

Erweitere den Typ `Integer` um einen "Fehlerwert" `Nothing`:

```
data MaybeInt = I Integer
              | Nothing
                deriving (Show, Eq)

safediv    :: Integer -> Integer -> MaybeInt
safediv _ 0 = Nothing
safediv x y = I (x `div` y)
```

Der folgende neue Typkonstruktor `Maybe` $\alpha$ kann jeden Typ $\alpha$ um das Element `Nothing` erweitern. Der Typkonstruktor ist **polymorph** (wie etwa auch [$\alpha$]):

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

safediv    :: Integer -> Integer -> Maybe Integer
safediv _ 0 = Nothing
safediv x y = Just (x `div` y)
```

**Unions** sind ebenfalls durch algebraische Datentypen darstellbar (vgl. mit C's `union` oder PASCALs varianten Records).

**Beispiel 9.3**

Der Typkonstruktor `Either` $\alpha$ $\beta$ konstruiert einen Union-Typ mit den zwei Diskrimanten `Left` und `Right`. `getLeft` filtert die mit `Left` markierten Elemente aus einer Liste des Union-Typs:

```
data Either a b = Left a
                | Right b
                  deriving (Show, Eq)

getLeft :: [Either a b] -> [a]
getLeft  = foldr (\x xs -> case x of Left e -> e:xs
                                     _   ->   xs) []

> :t [Left 'x', Right True, Right False, Left 'y']
[Left 'x',Right True,Right False,Left 'y'] :: [Either Char Bool]

> getLeft [Left 'x', Right True, Right False, Left 'y']
"xy" :: [Char]
```

**Frage:** Welches Ergebnis liefert der Aufruf

```
getLeft [Left 'x', Right True, Right 'z', Left 'y']?
```

## 9.2 Rekursive algebraische Typen

Die interessantesten Konstruktionen lassen sich durch rekursive Typ-Deklarationen erzielen. Damit lassen sich vor allem diverse Arten von **Bäumen** als neue Typen realisieren.

Der rekursive Typ `BinTree` $\alpha$ definiert den Typ der binären Bäume über einem beliebigen Typ $\alpha$:

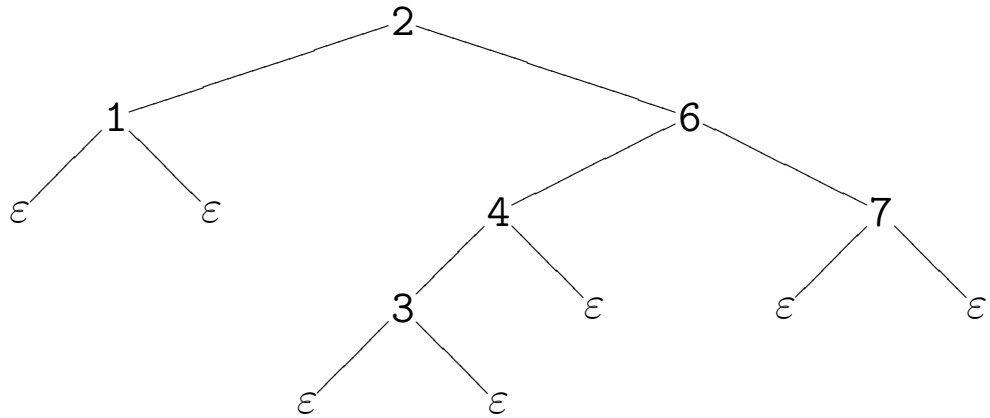
```
data BinTree a  =  Empty
                |  Node (BinTree a) a (BinTree a)
                   deriving (Eq, Show)
```

Der Konstruktor `Empty` steht damit für den leeren (Unter-)Baum, während `Node` einen Knoten mit linkem Nachfolger, Knotenbeschriftung des Typs $\alpha$ und rechtem Nachfolger repräsentiert.

Die Konstruktion eines Binärbaums mit `Integer`-Knotenlabels ist dann einfach:

```
atree :: BinTree Integer
atree  = Node (Node Empty 1 Empty)
              2
              (Node (Node (Node Empty 3 Empty) 4 Empty)
                    6
                    (Node Empty 7 Empty))
```

`atree` repräsentiert den folgenden binären Baum ($\varepsilon$ bezeichnet leere Unterbäume):




Um die Notation weiter zu vereinfachen, setzen wir eine Funktion zur Konstruktion von Blättern, `leaf`, ein:

```
leaf  :: a -> BinTree a
leaf x = Node Empty x Empty
```

Damit notieren wir `atree'` mit `atree' == atree` (Gleichheit sinnvoll aufgrund `deriving (Eq, ...)`) kürzer als

```
atree' :: BinTree Integer
atree'  = Node (leaf 1) 2 (Node (Node (leaf 3) 4 Empty) 6 (leaf 7))
```

Der eingebaute Typkonstruktor für Listen `[·]` ist, ganz ähnlich wie `BinTree`, ein rekursiver algebraischer Datentyp. Seine Definition lautet

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

Entgegen der bisherigen Vereinbarungen wird hier der Konstruktor $K_2 =$ `(:)` in Infix-Notation gebraucht. Auch für nutzerdefinierte Konstruktorfunktionen steht dieses Feature zur Verfügung:

Ein Konstruktorname der Form `:[!#$&*+/<=>?@\^|~:.]*` (ein `':'` gefolgt von einer beliebigen Folge von Symbolen, vgl. Abschnitt 6.2.1) kann auch infix angewandt werden.

**Beispiel 9.4**
Mittels Infix-Konstruktoren läßt sich bspw. der hier neu definierte Typ rationaler Zahlen sehr natürlich im Programmtext darstellen:

```
data Frac = Integer :/ Integer
          deriving Show
> 2 :/ 3
2 :/ 3 :: Frac
```

**Frage:** Wieso wird hier nicht auch die Gleichheit mittels `deriving (Show, Eq)` abgeleitet?

## 9.3 Bäume

Wir werden im folgenden einige Algorithmen auf Bäumen näher betrachten und dabei die Anwendung algebraischer Datentypen weiter vertiefen.

### 9.3.1 Größe und Höhe eines Baumes

Bei der Analyse von Algorithmen auf Bäumen hängt die Laufzeit oft von der Größe (Anzahl der Knoten) und Höhe (Länge des längsten Pfades von der Wurzel zu einem Blatt) eines Baumes ab.

Wir definieren hier die entsprechenden Funktionen `size` und `depth` für unsere vorher deklarierten `BinTrees`.

```
size, depth :: BinTree a -> Integer

size Empty        = 0
size (Node l a r) = size l + 1 + size r

depth Empty        = 0
depth (Node l a r) = 1 + depth l `max` depth r
```

Beide **Funktionen orientieren sich an der rekursiven Struktur des Typs** `BinTree` und sehen **je einen Fall für seine Konstruktoren** vor (vgl. Kapitel 8 zur Listenverarbeitung).

Beweise über Algorithmen auf algebraischen Datentypen verlaufen ganz analog zu Beweisen von Aussagen über Listen (vgl. Abschnitt 8.3).

Für den Typ `BinTree` $\alpha$ lautet das Schema für Induktionsbeweise (jeder `BinTree` hat entweder Form ① oder ②):

① **Induktionsverankerung**: leerer Baum `Empty`,
② **Induktionsschritt**: von $\ell$ und $r$ zu `Node` $\ell$ $a$ $r$

**Beispiel 9.5**
Zwischen der Größe und Tiefe eines Binärbaums $t$ besteht der folgende Zusammenhang („ein Baum der Tiefe $n$ enthält mindestens $n$ und höchstens $2^n - 1$ Knoten“):

$$\texttt{depth}\ t \quad \leq \quad \texttt{size}\ t \quad \leq \quad 2 \uparrow \texttt{depth}\ t - 1$$

Wir verwenden Induktion über die Struktur von $t$ zum Beweis des zweiten ‘$\leq$’:
**Induktionsverankerung** `Empty`:

$$\begin{array}{lcll} \texttt{size Empty} & = & 0 & (\texttt{size.1}) \\ & = & 2 \uparrow \texttt{depth Empty} - 1 & (\texttt{depth.1}, \text{Arithmetik}) \end{array}$$

Falls wir uns an dieser Stelle nicht auf die Begründung *Arithmetik* verlassen wollen, besteht die Möglichkeit, arithmetische Operationen durch Haskell-Äquivalente zu ersetzen (bspw. $\uparrow$ durch das durch uns definierte `power`).

**Induktionsschritt** `Node` $\ell\ a\ r$:

$$
\begin{array}{lll}
 & \texttt{size (Node } \ell\ a\ r\texttt{)} & \\
= & \texttt{size } \ell + 1 + \texttt{size } r & (\texttt{size}.2) \\
\leq & (2 \uparrow \texttt{depth } \ell - 1) + 1 + (2 \uparrow \texttt{depth } r - 1) & (\text{Hypothese}) \\
\leq & 2 \times ((2 \uparrow \texttt{depth } \ell)\ \texttt{`max`}\ (2 \uparrow \texttt{depth } r)) - 1 & (a \leq a\ \texttt{`max`}\ b) \\
= & 2 \times 2 \uparrow (\texttt{depth } \ell\ \texttt{`max`}\ \texttt{depth } r) - 1 & (a \leq b \Rightarrow 2 \uparrow a \leq 2 \uparrow b) \\
= & 2 \uparrow (1 + \texttt{depth } \ell\ \texttt{`max`}\ \texttt{depth } r) - 1 & (\text{Arithmetik}) \\
= & 2 \uparrow \texttt{depth (Node } \ell\ a\ r\texttt{)} - 1 & (\texttt{depth}.2)
\end{array}
$$

**Beispiel 9.5**

## 9.3.2 Linkester Knoten eines Binärbaumes

Ein weiteres kleines Problem auf Binärbäumen besteht in der Ermittlung der Knotenmarkierung des Knotens „links außen“. Wir schreiben dazu die Funktion `leftmost`.

`leftmost` kann nicht immer ein sinnvolles Ergbenis liefern: ein leerer Baum (`Empty`) hat keinen linkesten Knoten. Unsere Implementation von `leftmost` setzt daher daher den algebraischen Typkonstruktor `Maybe` ein, um dieses Problem evtl. signalisieren zu können.

Damit haben wir also

leftmost :: Bintree $\alpha$ -> Maybe $\alpha$

```
leftmost                 :: BinTree α -> Maybe α
leftmost Empty            = Nothing
leftmost (Node Empty a r) = Just a
leftmost (Node l a r)     = leftmost l
```

Eine alternative Formulierung wäre `leftmost'`, die zuerst rekursiv in den Baum absteigt, einen evtl. linkesten Knoten (`Just b`) nach oben propagiert bzw. den aktuell linkesten Knoten zurückgibt (`Just a`), wenn der linke Teilbaum leer sein sollte:

```
leftmost'             :: BinTree α -> Maybe α
leftmost' Empty        = Nothing
leftmost' (Node l a r) = case leftmost' l of
                              Nothing -> Just a
                              Just b  -> Just b
```

**Frage:** Welcher Variante würdet ihr den Vorzug geben?

Die folgende Variante des Problems ermittelt uns das Element links außen, gibt aber gleichzeitig auch den Baum zurück, der bei Entfernung des „Linksaußen“ entsteht:

```
splitleftmost'              :: BinTree α -> Maybe (α, BinTree α)
splitleftmost' Empty         = Nothing
splitleftmost' (Node l a r) = case splitleftmost' l of
                                   Nothing      -> Just (a, r)
                                   Just (a',l') -> Just (a', Node l' a r)
```

**Übung:** `splitleftmost'` orientiert sich an dem Rekursionsschema für `leftmost'` und nicht an dem für `leftmost`. Die ganze Arbeit beim rekursiven Abstieg in den Baum zu leisten ist schwieriger. Wie könnte eine endrekursive Variante `splitleftmost` implementiert werden?

 Nicht ganz einfach.

## 9.3.3 Linearisierung von Bäumen

Dieser Abschnitt befaßt sich mit der Überführung von Bäumen in Listen von Knotenmarkierungen. Wir werden sowohl **Tiefendurchläufe** als auch **Breitendurchläufe** ansprechen.

① **Tiefendurchlauf** Tiefendurchläufe folgen der rekursiven Struktur unserer Binärbäume und sind vergleichsweise simpel zu implementieren.

Je nachdem, ob man die Markierung eines Knotens (a) vor, (b) zwischen oder (c) nach der Linearisierung seiner Teilbäume ausgibt, erhält man verschiedene Tiefendurchläufe:

(b) *Inorder*:

$$\begin{array}{lcl} \texttt{inorder} & :: & \texttt{BinTree}\ \alpha\ \texttt{->}\ [\alpha] \\ \texttt{inorder Empty} & = & [] \\ \texttt{inorder (Node}\ \ell\ a\ r\texttt{)} & = & \texttt{inorder}\ \ell\ \texttt{++}\ [a]\ \texttt{++}\ \texttt{inorder}\ r \end{array}$$

Die entscheidenden Gleichungen von (a) *Preorder* und (c) *Postorder* lauten

$$\begin{array}{lcl} \texttt{preorder (Node}\ \ell\ a\ r\texttt{)} & = & [a]\ \texttt{++}\ \texttt{preorder}\ \ell\ \texttt{++}\ \texttt{preorder}\ r \\ \texttt{postorder (Node}\ \ell\ a\ r\texttt{)} & = & \texttt{postorder}\ \ell\ \texttt{++}\ \texttt{postorder}\ r\ \texttt{++}\ [a] \end{array}$$

**Beispiel:** `inorder atree` $\twoheadrightarrow$ `[1,2,3,4,6,7]`.

Die Effizienz von `inorder` wird durch die Laufzeit der Listenkonkatenation `++` bestimmt, die linear im ersten Argument ist (siehe Abschnitt 8.2). Der *worst-case* für `inorder` ist somit ein linksentarteter Baum.

Die Funktion `leftist` erzeugt einen linksentarteten Baum aus einer Liste von vorgegebenen Knotenmarkierungen:

```
leftist       :: [α] -> BinTree α
leftist []     = Empty
leftist (x:xs) = Node (leftist xs) x Empty
```

Aufgrund der Laufzeit von ++ benötigt `inorder (leftist [1..`$n$`])` eine Laufzeit proportional zu $n^2$.

**Übung:** Für `inorder` läßt sich eine Implementation finden, die linear in $n$ ist. Die Lösung orientiert sich an der Idee zur Beschleunigung von `reverse` aus Abschnitt 8.2.

② **Breitendurchlauf** Ein Breitendurchlauf eines Baumes zählt die Knoten ebenenweise von der Wurzel ausgehend auf. Wir setzen dazu eine Hilfsfunktion `traverse` ein, die eine Liste $ts$ von Teilbäumen (einer Ebene) erhält, und deren Knoten entsprechend aufzählt:

$$\begin{array}{lcl} \texttt{traverse} & :: & \texttt{[BinTree}\ \alpha\texttt{]} \texttt{ -> [}\alpha\texttt{]} \\ \texttt{traverse []} & = & \texttt{[]} \\ \texttt{traverse}\ ts & = & \texttt{roots}\ ts\ \texttt{++ traverse (sons}\ ts\texttt{)} \end{array}$$

Unseren Breitendurchlauf erhalten wir dann einfach mittels

$$\texttt{levelorder}\ t = \texttt{traverse}\ [t]$$

Es fehlen lediglich noch die Funktionen `roots` zur Bestimmung aller Wurzeln bzw. `sons` zur Bestimmung aller Teilbäume einer Liste von Bäumen:

```
roots                  :: [BinTree α] -> [α]
roots []                = []
roots (Empty       :ts) = roots ts
roots ((Node _ a _):ts) = a : roots ts

sons                   :: [BinTree α] -> [BinTree α]
sons []                = []
sons (Empty       :ts) = sons ts
sons ((Node l _ r):ts) = l : r : sons ts
```

Mit der Hilfe von *list comprehensions* (siehe Abschnitt 8.5), lassen sich beide Funktionen elegant als Einzeiler realisieren:

```
roots ts = [ a | Node _ a _ <- ts ]
sons ts  = [ t | Node l _ r <- ts, t <- [l, r] ]
```

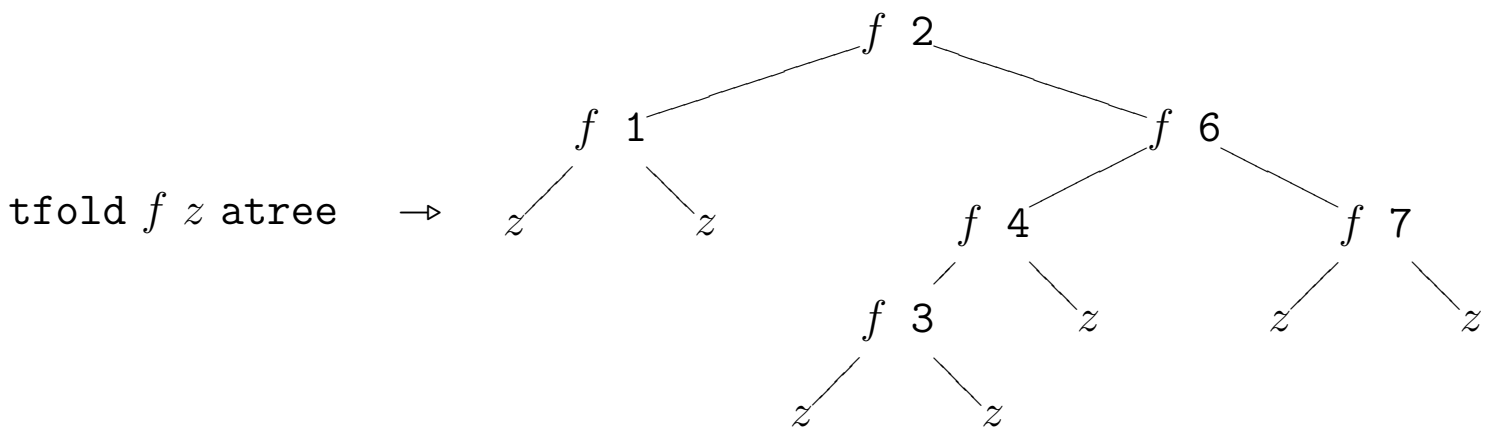
## 9.3.4 `fold` über Bäumen

Das in Abschnitt 8.1 besprochene allgemeine Rekursionsschema über Listen (implementiert durch `foldr`/`foldl`) läßt sich auf andere konstruierte algebraische Datentypen übertragen.

`foldr` $(\oplus)$ $z$ $xs$ ersetzt in der Liste $xs$ die Vorkommen der Listenkonstruktoren `(:)` durch Aufrufe von $\oplus$ bzw. `[]` durch $z$. Ganz analog läßt sich eine Funktion `tfold` (*tree fold*) über `BinTrees` definieren:

$$
\begin{array}{lcl}
\texttt{tfold} & :: & (\beta \texttt{ -> } \alpha \texttt{ -> } \beta \texttt{ -> } \beta) \texttt{ -> } \beta \texttt{ -> BinTree } \alpha \texttt{ -> } \beta \\
\texttt{tfold } f\ z\ \texttt{Empty} & = & z \\
\texttt{tfold } f\ z\ \texttt{(Node } \ell\ a\ r\texttt{)} & = & f\ \texttt{(tfold } f\ z\ \ell\texttt{)}\ a\ \texttt{(tfold } f\ z\ r\texttt{)}
\end{array}
$$

Der Effekt von `tfold` auf den Binärbaum `atree` aus den vorigen Abschnitten ist damit:



Die rekursiven Funktionen `size` und `depth` aus Abschnitt 9.3 können alternativ durch `tfold` implementiert werden:

```
size, depth :: BinTree α -> Integer

size  = tfold (\l _ r -> l + 1 + r) 0
depth = tfold (\l _ r -> 1 + l `max` r) 0
```

Die Tiefendurchläufe können ebenfalls als Instanzen von `tfold` verstanden werden:

```
inorder, preorder, postorder :: BinTree α -> [α]

inorder   = tfold (\l a r -> l ++ [a] ++ r) []
preorder  = tfold (\l a r -> [a] ++ l ++ r) []
postorder = tfold (\l a r -> l ++ r ++ [a]) []
```

Schließlich ist auch `leftmost'` mittels `tfold` ausdrückbar:

```
leftmost' :: BinTree α -> Maybe α
leftmost'  = tfold (\l a r -> case l of
                                Nothing -> Just a
                                Just b  -> Just b) Nothing
```

# 10 Typklassen und Overloading

Viele der bisher betrachteten Funktionen waren **polymorph**, d.h. der Typ dieser Funktionen enthielt mindestens eine Typvariable ($\alpha, \beta, \ldots$):

```
id          :: α -> α
snd         :: (α,β) -> β
length      :: [α] -> Int
take        :: Int -> [α] -> [α]
foldr       :: (α -> β -> α) -> α -> [β] -> α
levelorder  :: BinTree α -> [α]
```

Diesen Funktionen ist gemeinsam, daß für sie jeweils nur eine *einzige Funktionsdefinition* existiert, vollkommen unabhängig von den konkreten Typen der Argumente, auf die sie angewendet werden. Jeder Wert jedes Typs ist zulässig, es werden keinerlei Voraussetzungen an die Typen gestellt.

Man sagt daher auch, die Typvariablen polymorpher Funktionen sind **allquantifiziert**. Dann liest sich der Typ von `foldr` wie

$$\texttt{foldr} :: \forall\alpha.\forall\beta.(\alpha \texttt{ -> } \beta \texttt{ -> } \alpha) \texttt{ -> } \alpha \texttt{ -> } [\beta] \texttt{ -> } \alpha$$

Funktionen dieser Art werden **parametrisch polymorph** genannt.

**Beispiel 10.1**
In ① `take 3 "foo"` und ② `take 10 [0..]` wird jeweils die gleiche Funktionsdefinition von `take` angewandt:

```
take :: Int -> [α] -> [α]
take 0 _              = []
take _ []             = []
take n (x:xs) | n > 0 = x : take (n-1) xs
```

① $\alpha =$ `Char` und damit etwa `[] :: [Char]` und `(:) :: Char -> [Char] -> [Char]`,

② $\alpha =$ `Int` und daher `[] :: [Int]` und `(:) :: Int -> [Int] -> [Int]`.

`take` ist parametrisch polymorph, `take` :: $\forall\alpha$ . `[Int] -> [`$\alpha$`] -> [`$\alpha$`]`.

## 10.1 Ad-Hoc Polymorphie (Overloading)

Eine *andere* Art von Polymorphie versteckt sich hinter dem Konzept des **Overloading**:

> *Ein einzelnes syntaktisches Objekt (Operator, Funktionsname, . . . ) steht für* **verschiedene** *– aber semantisch ähnliche – Definitionen.*

**Beispiel 10.2**

Typische überladene Funktionen und Operatoren:

| | |
|---|---|
| `(==)`, `(<=)` | Gleichheit und Ordnung lassen sich für eine ganze Klasse von Typen sinnvoll definieren. |
| `(*)`, `(+)` | Arithmetische Ausdrücke über ganzen Zahlen und Fließkommazahlen werden mit identischen Operatorsymbolen notiert. |
| `show`, `read` | Für eine ganze Klasse von Typen lassen sich sinnvolle externe Repräsentationen (als `String`) angeben. |

**Beobachtungen:**

- Ohne Overloading wären Funktionsnamen wie `equalInt` und `equalBool`, oder `showDouble` und `showBinTree` notwending.
- Für den Typ `BinTree` $\alpha$ muß `show` offensichtlich vollkommen anders implementiert sein, als für `Double`.

**Offensichtliche Frage:**

Welchen Typ besitzt `(==)`? Gilt hier wirklich `(==)` :: $\forall\alpha.\alpha$ `->` $\alpha$ `->` `Bool`?

(Wie steht's mit $\alpha = \beta$ `->` $\gamma$ ?)

Haskell unterstützt eine kontrollierte Form des Overloading (auch: **Ad-Hoc Polymorphie**) mittels

## 10.2 Typklassen (`class`)

Eine **Typklasse** (*type class*, `class`) deklariert eine Familie von Funktionen, die für verschiedene konkrete Typen (die sog. `instances`) jeweils verschieden implementiert werden können.

Eine `class` schreibt lediglich den polymorphen Typ dieser Funktionen vor. Die Implementation liegt in den Händen der jeweiligen `instance`[8].

**Syntax:**

$$\begin{array}{l} \texttt{class}\ \ C\ \alpha\ \ \texttt{where} \\ \quad f_1 :: \tau_1 \\ \quad\ \vdots \\ \quad f_n :: \tau_n \end{array}$$

**Bemerkungen:**

- Der Klassenname $C$ muß mit einem Zeichen $\in$ `[A..Z]` beginnen.
- Die Typvariable $\alpha$ dient als Platzhalter für die später zu definierenden `instances`.
- Die Typen $\tau_i$ *müssen* die Typvariable $\alpha$ enthalten.

[8] Siehe aber die *default class methods* in Abschnitt 10.2.2.

**Beispiel 10.3**

Typen, für deren Werte Gleichheitstests sinnvoll sind, können die Operatoren der Klasse `Eq` (*equality*) überladen:

```
class Eq a where
   (==) :: a -> a -> Bool
   (/=) :: a -> a -> Bool
```

Typen, deren Werte aufzählbar sind, können die Funktionen der Klasse `Enum` überladen ...

```
class Enum a where
   succ           :: a -> a
   pred           :: a -> a
   toEnum         :: Int -> a
   fromEnum       :: a -> Int
   enumFrom       :: a -> [a]
   enumFromThen   :: a -> a -> [a]
   enumFromTo     :: a -> a -> [a]
   enumFromThenTo :: a -> a -> a -> [a]
```

... und dann von synaktischem Zucker wie `[`$x$`..]` ($\equiv$ `enumFrom` $x$) oder `[`$x$`..`$y$`]` ($\equiv$ `enumFromTo` $x$ $y$) Gebrauch machen.

Jetzt ist klarer, welchen Typ bspw. `(==)` wirklich besitzt:

> Für alle Typen $\alpha$, *die alle Operationen der Typklasse* `Eq` *implementieren*, gilt
> `(==)` :: $\alpha$ `->` $\alpha$ `->` `Bool`.

Eine Typklasse $C$ kann also als **Prädikat auf Typen** verstanden werden. Das drückt die zugehörige Haskell-Syntax, die sog. *type constraints* oder *contexts*, klar aus:

$$f_i \texttt{ :: } (C\ \alpha) \texttt{ => } \tau_i$$

Hier also:

```
(==) :: (Eq a) => a -> a -> Bool
```

Und auch:

```
elem :: (Eq a) => a -> [a] -> Bool
```

**NB.** Kein Allquantor $\forall$ im polymorphen Typ von $f_i$. Die Funktionen `(==)`, bzw. `elem`. Typklassen beschränken (kontrollieren) Polymorphie.

## 10.2.1 `instance`

Die Zugehörigkeit eines Typs $\alpha$ zu einer Typklasse $C$ wird Haskell mittels einer `instance`-Erklärung angezeigt. Danach gilt der *type constraint* $(C\ \alpha)$ dann als erfüllt.

**Syntax:**

$$\begin{aligned}\texttt{instance}\ \ C\ \alpha\ \ \texttt{where}\\ f_1 &= \langle \textit{Definition } f_1\rangle\\ &\vdots\\ f_n &= \langle \textit{Definition } f_n\rangle\end{aligned}$$

**Bemerkung:**

- Die Typen der $f_i$ dürfen hier nicht wiederholt angegeben werden (sondern werden der zugehörigen `class` $C$-Deklaration entnommen).

**Beispiel 10.4**

Werte des Typs `Bool` erlauben Test auf Gleichheit:

```
instance Eq Bool where
   x == y = (x && y) || (not x && not y)
   x /= y = not (x == y)
```

**Beispiel 10.5**

Die Gleichheit von Werten des Typs `Integer` wird aus Effizienzgründen auf den primitiven Gleichheitstest `primEqInteger` der unterliegenden Maschine zurückgeführt:

```
instance Eq Integer where
   x == y = primEqInteger x y
   x /= y = not (x == y)
```

**Nächster Schritt:** Gleichheit von Listen des Typs $[\alpha]$. Wohldefiniert offensichtlich nur dann, wenn auch $\alpha$ Gleichheitstests zuläßt. Daher:

```
instance (Eq a) => Eq [a] where
   []     == []     = True
   (x:xs) == (y:ys) = x == y && xs == ys
   _      == _      = False

   xs /= ys         = not (xs == ys)
```

**NB.** Diese (==)-Operatoren enstammen verschiedenen Instantiierungen.

**Frage:** Wie können Paare des Typs $(\alpha, \beta)$ für den Gleichheitstest zugelassen werden?

## 10.2.2 `class` Defaults

Oft ist es sinnvoll, *default*-Definitionen für die Funktionen einer Typklasse vorzunehmen (s. `Eq` und `(/=)`).

Eine *default*-Definition wird innerhalb des `class`-Konstruktes vorgenommen. Jede Typinstanz dieser Typklasse "erbt" diese Definition, wenn diese nicht lokal überladen wird.

**Beispiel 10.6**
Gleichheit und Ungleichheit stehen als Relationen in einem offensichtlichen Verhältnis zueinander:

```
class Eq a where
   -- Overload either (==) or (/=)!
   (==) :: a -> a -> Bool
   x == y = not (x /= y)

   (/=) :: a -> a -> Bool
   x /= y = not (x == y)
```

**Bemerkungen:**

- In den `instance`-Deklarationen muß jetzt nur noch eine (diese aber mindestents!) der Definitionen überladen werden.
- *Defaults* erlauben eine eingeschränkte Form der Konsistenzkontrolle für Klassenoperationen.

**Beispiel 10.7**

Alle Typen, deren Werte anzuordnen sind, können die Operationen der Typklasse `Ord` überladen und dann mit Operatoren wie `(<)` und `max` verglichen werden (`data Ordering = EQ | LT | GT`):

```
class Ord a where
   compare              :: a -> a -> Ordering
   (<), (<=), (>=), (>) :: a -> a -> Bool
   max, min             :: a -> a -> a

   -- Minimal complete definition: (<=) or compare
   compare x y | x == y    = EQ
               | x <= y    = LT
               | otherwise = GT

   x <= y                  = compare x y /= GT
   x <  y                  = compare x y == LT
   x >= y                  = compare x y /= LT
   x >  y                  = compare x y == GT

   max x y   | x >= y      = x
             | otherwise   = y
   min x y   | x <= y      = x
             | otherwise   = y
```

**Frage:** Steht hier die “ganze Wahrheit”?

# 10.3 `instance`-Deklarationen für algebraische Datentypen

**Beispiel 10.8**

Erinnerung: Typ `Weekday`:

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Wie macht man `Weekday` zu einem aufzählbaren Datentyp (also Instanz von `Enum`)?
Die Prelude definiert *defaults* für alle Funktionen in `Enum` bis auf `toEnum` und `fromEnum`:

```
class Enum a where
   toEnum     :: Int -> a
   fromEnum   :: a -> Int
         ⋮
   succ :: a -> a
   succ = toEnum . (+1) . fromEnum
         ⋮
   enumFromTo :: a -> a -> [a]
   enumFromTo x y = map toEnum [ fromEnum x .. fromEnum y ]
         ⋮
```

Verbleibt also die Definition von `toEnum` und `fromEnum` mit der offensichtlich gewünschten Eigenschaft `(fromEnum . toEnum) == id`:

```
instance Enum Weekday where
   toEnum 0 = Mon          |          fromEnum Mon = 0
   toEnum 1 = Tue          |          fromEnum Tue = 1
   toEnum 2 = Wed          |          fromEnum Wed = 2
   toEnum 3 = Thu          |          fromEnum Thu = 3
   toEnum 4 = Fri          |          fromEnum Fri = 4
   toEnum 5 = Sat          |          fromEnum Sat = 5
   toEnum 6 = Sun          |          fromEnum Sun = 6
```

**NB.** `fromEnum`, `toEnum` etablieren einen Isomorphismus von `Weekday` und `[0..6]`.
Jetzt z.B. möglich: `[Wed .. Sat]` $\rightarrowtriangle$ `[Wed, Thu, Fri, Sat]`.

**Beispiel 10.8**

Die Etablierung derartiger Datentypen als Instanz von `Enum` (aber auch `Eq`, `Ord`, `Show`, `Read`[9]) ist kanonisch, aufwendig, und fehleranfällig.

[9] Einige dieser Typklassen tauchen später noch auf.

**Beispiel 10.9**

Wir geben an, wie ein Gleichheitstest auf Werten des `BinTree` auszuführen ist und machen `BinTree` damit zur Instanz von `Eq`:

```
instance (Eq a) => Eq (BinTree a) where
   Empty          == Empty          = True
   Node l1 x1 r1 == Node l2 x2 r2 = x1 == x2 && l1 == l2 && r1 == r2
   _              == _              = False
```

**Beobachtungen:**

- Die Struktur der Definition von (==) folgt der rekursiven Struktur des Datentypes `BinTree`.
- Nur jeweils durch gleiche Konstruktoren erzeugte Werte können potentiell gleich sein.

## 10.3.1 `deriving`

Diese reguläre typ-orientierte Struktur der Typklassen-Operationen der in der Prelude definierten Typklassen `Eq`, `Enum`, `Ord`, `Show` und `Read` macht es Haskell möglich, diese `instance`-Deklarationen automatisch abzuleiten (*to derive*).

**Syntax:**

$$\begin{array}{rcl} \texttt{data}\ T\ \alpha_1\ \alpha_2 \ldots \alpha_n & = & \ldots \\ & | & \ldots \\ & \vdots & \\ \multicolumn{3}{c}{\texttt{deriving}\ (C_1, \ldots, C_m)} \end{array}$$

Obige `data`-Deklaration macht den Typkonstruktor $T$ Instanz der Typklassen $C_1, \ldots C_m$. Die Definitionen der Typklassen-Operationen können **automatisch** erzeugt werden:

| | |
|---|---|
| `Eq` | Ableitbar für alle Datentypen, Gleichheit wird über die Identität von Konstruktoren und (rekursiv) der Gleichheit der Komponenten des Datentyps entschieden. |
| `Ord` | Ableitbar für alle Datentypen, Reihenfolge der Konstruktoren in `data`-Deklaration entscheidend, Ordnung wird lexikographisch (rekursiv) definiert. |
| `Enum` | Datentyp muß ein reiner Summentyp sein (s. auch Kapitel 9), also `data` $T$ = $K_0$ \| $\cdots$ \| $K_{(n-1)}$ mit `fromEnum` $K_i = i$. |
| `Show` | Textuelle Repräsentation der Konstruktoren, s. später. |
| `Read` | dito |

**Beispiel 10.10**
Mit

```
data Weekday = Mon | Tue | Wed | Thu | Fri | Sat | Sun
   deriving (Eq, Ord, Enum)
```

ist dann folgendes möglich:

```
> Mon < Tue
True :: Bool
> Mon == Sat
False :: Bool
> [Mon,Wed .. Sun]
[Mon,Wed,Fri,Sun] :: [Weekday]
> fromEnum Sat
5 :: Int
```

**Frage:** Was ergibt der Aufruf `map toEnum [6,5..0]`?
(Achtung: Über die involvierten Typen nachdenken!)

**Beispiel 10.11**

Die Deklaration des wohlbekannten `BinTree a` via

```
data BinTree a = Empty
               | Node (BinTree a) a (BinTree a)
   deriving (Eq, Ord)
```

führt automatisch zu:

```
instance (Eq a) => Eq (BinTree a) where
   Empty          == Empty          = True
   Node l1 x1 r1 == Node l2 x2 r2 = x1 == x2 && l1 == l2 && r1 == r2
   _              == _              = False

instance (Ord a) => Ord (BinTree a) where
   Empty          <= Empty          = True
   Empty          <= Node l2 x2 r2 = True
   Node l1 x1 r1 <= Empty          = False
   Node l1 x1 r1 <= Node l2 x2 r2 =    l1 < l2
                                    || l1 == l2 && x1 < x2
                                    || x1 == x2 && r1 <= r2
```

Erinnerung zu den `==`: *type constraint* `Ord a` impliziert `Eq a` (s. auch Abschnitt 10.5),

Nicht immer ist `deriving` die Antwort.

**Beispiel 10.12**

Erinnerung: `Frac` repräsentiert rationale Zahlen:

```
data Frac = Integer :/ Integer
   deriving Eq
```

Hier führt `deriving` nicht zum Ziel:

```
> 2 :/ 5 == 4 :/ 10
False :: Bool
```

Gemeint ist vielmehr:

```
equalFrac :: Frac -> Frac -> Bool
equalFrac (x1 :/ y1) (x2 :/ y2) = x1 * y2 == x2 * y1

instance Eq Frac where
  (==) = equalFrac

> 2 :/ 5 == 4 :/ 10
True :: Bool
```

## 10.4 Die Typklasse `Show`

Die Überführung ihrer Werte in eine externe Repräsentation (`String`) ist eine der Kernaufgaben jeder Programmiersprache:

- Ein-/Ausgabe von Werten, interaktive Programme.
- Speichern/Laden/Übertragung von Daten.
- *Pretty printing* in Interpreter-Umgebungen und Debuggern.

In Haskell wird die benötige Funktionalität von allen Instanzen der Typklasse `Show` bereitgestellt[10] (s. die Prelude für die vollständige Definition):

```
class Show a where
    show      :: a -> String
    showList  :: [a] -> (String -> String)

    -- Minimal complete definition: show
    showList []     = ...
    showList (x:xs) = ...
```

[10] Typklasse `Read` leistet die Umkehrabbildung, das sog. *parsing*.

**Beispiel 10.13**

Ausgabe von Werten des Typs `Frac` als Brüche der Form $\frac{x}{y}$:

```
showFrac :: Frac -> String
showFrac (x :/ y) = replicate (div (l-lx) 2) ' ' ++ show x ++ "\n" ++
                    replicate l '-'                       ++ "\n" ++
                    replicate (div (l-ly) 2) ' ' ++ show y
   where
        lx = length (show x)
        ly = length (show y)
        l  = max lx ly

instance Show Frac where
   show = showFrac
```

Haskell-Interpreter `Hugs` benutzt `show :: (Show a) => a -> String`, um in interaktiven Sessions Werte darzustellen:

```
> 42 :/ 1000
 42
----
1000 :: Frac
```

## Beispiel 10.14

Neue `show`-Funktion zur "Visualisierung" von Werten des Typs `BinTree a`:

```
showTree :: Show a => BinTree a -> String
showTree = concat . ppTree
   where
        ppTree :: Show a => BinTree a -> [String]
        ppTree Empty        = ["@\n"]
        ppTree (Node l x r) = [show x ++ "\n"]
                            ++ ("|--" ++ ls) : map ("|  " ++) lss
                            ++ ("`--" ++ rs) : map ("   " ++) rss
           where
                ls:lss = ppTree l
                rs:rss = ppTree r

instance Show a => Show (BinTree a) where
   show = showTree
```

```
1
|--2
|  |--@
|  `--3
|     |--@
|     `--@
`--@
```

## 10.5 Oberklassen

In größeren Software-Systemen sind hierarchische Abhängigkeiten zwischen Typklassen typisch:

> *Typ $T$ ist Instanz der Typklasse $C$, was aber nur Sinn macht, wenn $T$ auch schon Instanz der Typklassen $C_1, \ldots, C_n$ ist.*

**Beispiel:** In Haskell stützt sich Anordnung eines Typs (`Ord`) auf die Vergleichbarkeit seiner Werte (`Eq`), s. Implementation von `compare`.

**Konsequenzen:**

- Jeder Typ in Typklasse `Ord` muß auch Instanz von `Eq` sein.
- Ist ein Typ in `Ord`, sind auf seine Werte die Operationen aus `Ord` *und* `Eq` anwendbar.

**NB.** Vergleiche diese Situation mit der in *OO-Sprachen*, speziell *Vererbung*.

**Syntax:**

$$\texttt{class (}C_1\texttt{,...,}C_n\texttt{) => }C\ \alpha\ \texttt{where ...}$$

- Die Typklassenhierarchie muß *azyklisch* sein.
- Die einzige Typvariable, die in den $C_i$ auftreten darf, ist $\alpha$.

## 10.5.1 Haskells numerische Klassen

Haskell stellt eine ganze Familie von numerischen Datentypen zur Verfügung. Alle numerischen Typen sind Instanz der Oberklasse `Num`:

```
class (Eq a, Show a) => Num a where
   (+), (-), (*)  :: a -> a -> a
   negate         :: a -> a
   abs, signum    :: a -> a
   fromInteger    :: Integer -> a
   fromInt        :: Int -> a

   -- Minimal complete definition: All, except negate or (-)
   x - y           = x + negate y
   negate x        = 0 - x
```

**Bemerkungen:**

- Alle numerischen Datentypen sind vergleichbar und haben eine textuellte Repräsentation.
- `Num` enthält nur die Operationen auf Zahlen, die auf *alle* numerischen Datentypen anwendbar sind. Beispielsweise läßt sich jede ganze Zahl als reelle, rationale, komplexe, ... Zahl interpretieren (`fromInt`, `fromInteger`).
- `-x` ist syntaktischer Zucker für `negate x`.

Ein speziellerer Zahldatentyp ist `Fractional`, der Typ aller gebrochenen Zahlen:

```
class (Num a) => Fractional a where
   (/)          :: a -> a -> a
   recip        :: a -> a
   fromRational :: Rational -> a
   fromDouble   :: Double -> a

   -- Minimal complete definition: fromRational and ((/) or recip)
   recip x       = 1 / x
   fromDouble    = fromRational . toRational
   x / y         = x * recip y
```

## 10.5.2 Numerische Konstanten

Eine numerische Konstante wie 42 ist in Haskell selbst überladen, da sie – je nach Kontext – als Wert des Typs `Integer` aber auch z.B. als `Double` angesehen werden kann.

Die Konstante 42 ist tatsächlich nur syntaktischer Zucker für `fromInteger 42`. Damit gilt `42 :: Num a => a` (lies: 42 *hat einen beliebigen numerischen Typ*).

```
> :t 42
42 :: Num a => a
```

# Referenzen

Philip Wadler und Stephen Blott (1989). *How to make Ad-hoc Polymorphism less Ad Hoc*. In *16th Symposium on Principles of Programming Languages*. Austin, Texas.
http://www.cs.bell-labs.com/~wadler/papers/class/class.ps.gz.

# 11 Fallstudie: Reguläre Ausdrücke

Mit den bisher eingeführten Sprachelementen von Haskell lassen sich bereits substantielle Projekte realisieren. Dieses Kapitel

① beschreibt einen algebraischen Datentyp zur Repräsentation von **regulären Ausdrücken** (*regular expressions*) und

② entwickelt einen interessanten Algorithmus zum *regular expression matching*, der *ohne* DFAs operiert.

## 11.1 Regular Expressions

**Reguläre Ausdrücke** (*regular expressions*) beschreiben Mengen von *Strings*.

- *Strings* bezeichnet dabei nicht unbedingt Zeichenketten (in Haskell also den Typ `String`), sondern allgemeiner *Listen von Symbolen*.
- Symbole können Zeichen (`Char`) sein, aber auch Zahlen, Bits, IP-Pakete, Events einer SAX-Parsers, ...

  Wenn $\alpha$ den Typ der Zeichen repräsentiert, dann steht *Strings* also generell für $[\alpha]$.

Reguläre Ausdrücke werden mittels sog. **Konstruktoren** aufgebaut:

- **Konstante** (0-stellige Konstruktoren; sei $c$ ein Symbol):
  - $\varnothing$ "*none*"
  - $\varepsilon$ "*epsilon*"
  - $c$
- **Unäre Konstruktoren** (sei $r$ ein regulärer Ausdruck):
  - $r$`*` "*Kleensche Hülle*"
  - $r$`?` "*Option*"
  - $r$`+` "*positive Kleensche Hülle*"
- **Binäre Konstruktoren** ($r, s$ reguläre Ausdrücke):
  - $r$`,` $s$ "*Konkatenation*"
  - $r$ `|` $s$ "*Alternation*"

Konstruktoren können auf Konstante oder weitere Konstruktoren angewandt werden, jegliche Kombination führt zu gültigen regulären Ausdrücken.

**Beispiel 11.1 (Symbole sind die Zeichen `0`, `1`)**

- `1, 1, 0*`
- `1 |` $\varepsilon$
- `1, 1 | 0, 0`

Um Mehrdeutigkeiten wie in `1, 1 | 0, 0` aufzulösen, läßt man Klammern `(, )` zur Gruppierung zu. Um wiederum Klammern zu sparen, erhalten die Konstruktoren Prioritäten (je höher die Priorität, desto stärker bindet der Konstruktor):

| Konstruktor | Priorität |
|---|---|
| * | 4 |
| + | 3 |
| ? | 2 |
| , | 1 |
| \| | 0 |

**Beispiel 11.2**

- `1, 1 | 0, 0` $\equiv$ `(1, 1) | (0, 0)`
- `1, 1?, (0, 0)+`

### 11.1.1 Minimalität

Die Konstruktoren ? und + führen lediglich Abkürzungen ein und sind streng genommen überflüssig (d.h. fünf Konstruktoren sind ausreichend):

$$r\texttt{?} \quad \equiv \quad r \texttt{ | } \varepsilon$$
$$r\texttt{+} \quad \equiv \quad r\texttt{, } r\texttt{*}$$

**Beispiel 11.3**

▶ `1, 1?, (0, 0)+` $\equiv$ `1, (1 |` $\varepsilon$`), 0, 0, (0, 0)*`

Reguläre Ausdrücke lassen sich in Haskell offensichtlich direkt auf einen algebraischen Datentyp zurückführen. Den Typ der Symbole spezifizieren wir allgemein als $\alpha$:

```
data RegExp a = RxNone
              | RxEpsilon
              | RxSymbol a
              | RxStar (RegExp a)                  -- r*
              | RxPlus (RegExp a)                  -- r+
              | RxOpt (RegExp a)                   -- r?
              | RxSeq (RegExp a) (RegExp a)        -- r, s
              | RxAlt (RegExp a) (RegExp a)        -- r | s
```

Der reguläre Ausdruck aus dem letzten Beispiel (Typ `RegExp Integer`) wird beispielsweise mittels des folgenden Haskell-Ausdrucks konstruiert:

```
RxSeq (RxSymbol 1)
      (RxSeq (RxOpt (RxSymbol 1))
             (RxPlus (RxSeq (RxSymbol 0)
                            (RxSymbol 0))))
```

**Beachte:** Auf Haskell-Ebene klärt die explizite Schachtelung der Konstruktoren mögliche Mehrdeutigkeiten. Klammern sind überflüssig.

## 11.2 `RegExp` als Instanz von `Show`

Da wir während der Entwicklung unseres Skriptes immer wieder reguläre Ausdrücke (Werte des Typs `RegExp` $\alpha$) auf dem Terminal ausgeben/kontrollieren müssen, lohnt sich sicher der Aufwand, `RegExp` $\alpha$ zur Instanz von `Show` zu machen.

Die Ausgabe soll dabei der Notation folgen, die wie am Anfang des Kapitels genutzt haben. Wenn möglich, sollen Klammern fortgelassen werden.

```
1  instance (Show a) => Show (RegExp a) where
2      showsPrec _ RxNone       = showString ""
3      showsPrec _ RxEpsilon    = showString "eps"
4      showsPrec _ (RxSymbol c) = shows c
5      showsPrec _ (RxStar r)   = showsPrec 4 r . showChar '*'
6      showsPrec p (RxPlus r)   =
7          showParen (p > 3) (showsPrec 3 r . showChar '+')
8      showsPrec p (RxOpt r)    =
9          showParen (p > 2) (showsPrec 2 r . showChar '?')
10     showsPrec p (RxSeq  r s) =
11         showParen (p > 1) (showsPrec 1 r . showString ", " . showsPrec 1 s)
12     showsPrec p (RxAlt  r s) =
13         showParen (p > 0) (showsPrec 0 r . showString " | " . showsPrec 0 s)
```

Einige Dinge sollten klar sein:

- Funktion `showsPrec` behandelt jeden Konstruktor von `RegExp a` mittels *pattern matching*.
- Der Constraint `(Show a)` erklärt sich aus Zeile 4.

Aber . . .

① Wird hier Funktionskomposition statt ++ zur Konkatenation von `Strings` benutzt?
② Wieso wird hier `showsPrec` definiert und nicht `show`?

**Zu ①:**

++ hat lineare Komplexität in Länge des linken Argumentes. (Wiederholtes) Konkatenieren von Teilergebnissen (`Strings`) in einer `show` Funktion, wie sie für `RegExp` $\alpha$ notwendig wäre, führt zu einer Laufzeit, die quadratisch in Größe des regulären Ausdrucks ist (s. Kapitel 8).

**Idee:** Wandle Funktionen der Form

$$f :: \alpha \texttt{ -> String}$$

um in Funktionen der Form

$$f' :: \alpha \texttt{ -> (String -> String)} \ .$$

$f'\ x\ s$ transformiert $x :: \alpha$ in einen `String` und konkateniert den *Akkumulator* $s$ mit diesem, d.h.

$$f\ x \quad \equiv \quad f'\ x\ \texttt{""}$$

```
-- ‘‘Fast strings’’ with constant time concatenation
type ShowS = String -> String

mk :: String -> ShowS                -- convert String into a ‘‘fast string’’
mk = (++)

(+++) :: ShowS -> ShowS -> ShowS     -- concatentate two ‘‘fast strings’’
(+++) = (.)

string :: ShowS -> String
string f’ = f’ ""                    -- convert ‘‘fast string’’ to String
```

**Beispiel 11.4 (Rechnen mit `ShowS`)**

```
        string (mk "foo" +++ mk "bar" +++ mk "baz")
  ⟶mk   string (("foo" ++) +++ ("bar" ++) +++ ("baz" ++))
  ⟶+++  string (("foo" ++) . ("bar" ++) . ("baz" ++))
  ⟶string (("foo" ++) . ("bar" ++) . ("baz" ++)) ""
  ⟶(.)  (("foo" ++) . ("bar" ++)) ("baz" ++ "")
  ⟶++,(.) ("foo" ++) ("bar" ++ "baz")
  ⟶++   "foo" ++ "barbaz"
```

Haskells Typklasse `Show` ist für den Einsatz Technik entsprechend vorbereitet:

```
class Show a where
   show       :: a -> String
   showsPrec  :: a -> ShowS           -- type ShowS = String -> String

    -- Minimal complete definition: show or showsPrec
   show x          = showsPrec 0 x ""
   showsPrec _ x s = show x ++ s
   ⋮

showString :: String -> ShowS
showString = (++)

showChar :: Char -> ShowS
showChar = (:)
```

**Zu ②:**

**Einsparen von Klammern bei der Ausgabe:** Setze um einen Konstruktor Klammern genau dann, wenn der *umgebende* Konstruktor eine höhere Priorität $p$ aufweist.

**Beispiel 11.5 ()**
Dank der Schachtelung der Konstruktoren ist die Priorität des umgebenden Konstruktors schon jeweils bekannt. Der äußerste Konstruktor ist umgeben von einem "virtuellen Konstruktor" der Priorität 0 (s. `class Show`).

- $\overbrace{\underbrace{\texttt{0, 0}}_{p=1} \texttt{ | } \underbrace{\texttt{1, 1}}_{p=1}}^{p=0}$
- $\overbrace{\texttt{(}\underbrace{\texttt{0, 0}}_{p=1}\texttt{)*}}^{p=4}$

- `showsPrec :: Integer -> a -> ShowS`
  Erstes Argument repräsentiert die Priorität des *umgebenden* Konstruktors
- `showParen :: Bool -> ShowS -> ShowS`
  Erstes Argument bestimmt, ob zweites Argument in Klammern gesetzt werden soll oder nicht.

**Frage:** Implementation von `showParen`?

## 11.3 Regular Expression Matching

Jedem regulärem Ausdruck $r$ ist die Menge der Listen von Symbolen—seine **Sprache** $L(r)$—zugeordnet, die $r$ **akzeptiert**.

| $r$ | $L(r)$ |
|---|---|
| $\varnothing$ | $\emptyset$ |
| $\varepsilon$ | $\{\texttt{[]}\}$ |
| $c$ | $\{\texttt{[}c\texttt{]}\}$ |
| $r\texttt{*}$ | $\{\texttt{[]}\} \cup L(r\texttt{,}r\texttt{*})$ |
| $r\texttt{,}s$ | $\{\ x \texttt{ ++ } y \mid x \in L(r), y \in L(s)\ \}$ |
| $r\texttt{|}s$ | $L(r) \cup L(s)$ |

Dies können wir nutzen, um bspw. die Sprachen der nicht-wesentlichen Konstruktoren zu berechnen.

**Beispiel 11.6**

$$\begin{aligned} L(\texttt{'x'?}) &= L(\texttt{'x'|}\varepsilon) \\ &= L(\texttt{'x'}) \cup L(\varepsilon) \\ &= \{\texttt{['x']}\} \cup \{\texttt{[]}\} \\ &= \{\texttt{['x'], []}\} \\ &= \{\texttt{"x", ""}\} \end{aligned}$$

## Beispiel 11.7

$$
\begin{aligned}
L(\texttt{1+}) &= L(\texttt{1, 1*}) \\
&= \{\ x \texttt{ ++ } y \mid x \in L(\texttt{1}), y \in L(\texttt{1*})\ \} \\
&= \{\ x \texttt{ ++ } y \mid x \in \{\texttt{[1]}\}, y \in L(\texttt{1*})\ \} \\
&= \{\ \texttt{[1]} \texttt{ ++ } y \mid y \in L(\texttt{1*})\ \} \\
&= \{\ \texttt{[1]} \texttt{ ++ } y \mid y \in \{\texttt{[]}\} \cup L(\texttt{1, 1*})\ \} \\
&= \{\ \texttt{[1]} \texttt{ ++ } y \mid y \in \{\texttt{[]}\}\} \cup \{\ x \texttt{ ++ } z \mid x \in L(\texttt{1}), z \in L(\texttt{1*})\ \} \\
&= \{\ \texttt{[1]}\ \} \cup \{\texttt{[1]} \texttt{ ++ } y \mid y \in \{\ x \texttt{ ++ } z \mid x \in L(\texttt{1}), z \in L(\texttt{1*})\ \} \\
&= \{\ \texttt{[1]}\ \} \cup \{\texttt{[1]} \texttt{ ++ } y \mid y \in \{\ \texttt{[1]} \texttt{ ++ } z \mid z \in L(\texttt{1*})\ \}\ \} \\
&= \{\ \texttt{[1]}\ \} \cup \{\texttt{[1]} \texttt{ ++ [1] ++ } y \mid y \in \{\ z \mid z \in L(\texttt{1*})\ \}\ \} \\
&= \{\ \texttt{[1]}\ \} \cup \{\texttt{[1, 1]} \texttt{ ++ } y \mid y \in L(\texttt{1*})\ \} \\
&\ \ \vdots \\
&= \{\ \texttt{[1]}, \texttt{[1, 1]}, \texttt{[1, 1, 1]}, \ldots\}
\end{aligned}
$$

Zwei reguläre Ausdrücke $r, s$ sind äquivalent ($r \equiv s$), falls $L(r) = L(s)$. Dies können wir nutzen, um bereits bei der Konstruktion Vereinfachungen an regulären Ausdrücken vorzunehmen, ohne die akzeptierte Sprache zu verändern.

**Beispiel 11.8**

- $r\,\texttt{,}\,\varepsilon \quad \equiv \quad \varepsilon\,\texttt{,}\,r \quad \equiv \quad r$
- $r\ \texttt{|}\ \varnothing \quad \equiv \quad \varnothing\ \texttt{|}\ r \quad \equiv \quad r$
- $\varnothing\texttt{?} \quad \equiv \quad \varepsilon$
- $\varnothing\texttt{*} \quad \equiv \quad \varepsilon$

**Fragen:**

- $\varnothing\,\texttt{,}\,r \quad \equiv$
- $r\,\texttt{,}\,\varnothing \quad \equiv$
- $\varnothing\texttt{+} \quad \equiv$
- $\varepsilon\texttt{+} \quad \equiv$
- $r\texttt{**} \quad \equiv$
- $r\texttt{*?} \quad \equiv$

Um diese Vereinfachungen gleich bei der Konstruktion von Werten des Typs `RegExp` $\alpha$ ausnutzen zu können, codieren wir das Wissen über die Äquivalenzen mittels der folgenden **Konstruktorfunktionen**:

```
rxNone, rxEpsilon     :: RegExp a
rxSymbol              :: a -> RegExp a
rxStar, rxPlus, rxOpt :: RegExp a -> RegExp a

rxNone     = RxNone
rxEpsilon  = RxEpsilon

rxSymbol c = RxSymbol c

rxStar RxNone    = RxEpsilon
rxStar RxEpsilon = RxEpsilon
rxStar r         = RxStar r

rxPlus RxNone    = RxNone
rxPlus RxEpsilon = RxEpsilon
rxPlus r         = RxPlus r

rxOpt RxNone     = RxEpsilon
rxOpt RxEpsilon  = RxEpsilon
rxOpt r          = RxOpt r
```

```
rxSeq, rxAlt          :: RegExp a -> RegExp a -> RegExp a

rxSeq RxNone    _          = RxNone
rxSeq RxEpsilon r'         = r'
rxSeq _         RxNone     = RxNone
rxSeq r         RxEpsilon  = r
rxSeq r         r'         = RxSeq r r'

rxAlt RxNone    r'         = r'
rxAlt r         RxNone     = r
rxAlt r         r'         = RxAlt r r'
```

**Aufgabe:** Einige Dialekte zur Konstruktion von regulären Ausdrücken führen einen $n$-stelligen ($n \geq 1$) Konstruktor & mit folgender Semantik ein:

$$\begin{aligned}
\texttt{\&}\ r &\equiv r \\
\texttt{\&}\ r_1\ r_2 &\equiv r_1\texttt{,}\ r_2 \mid r_2\texttt{,}\ r_1 \\
\texttt{\&}\ r_1\ r_2\ r_3 &\equiv r_1\texttt{,}\ r_2\texttt{,}\ r_3 \mid r_1\texttt{,}\ r_3\texttt{,}\ r_2 \mid \cdots \mid r_3\texttt{,}\ r_2\texttt{,}\ r_1 \\
&\vdots
\end{aligned}$$

Die Anwendung von & auf $n$ Argumente führt also zu einer Kette von $n!$ Alternationen. Implementiert eine Funktion `rxAll :: [RegExp a] -> RegExp a`, die & auf die wesentlichen Konstruktoren zurückführt.

## 11.4 Die Ableitung regulärer Ausdrücke

Aber wie bekommen wir schließlich das *regular expression matching* Problem in den Griff? Wir suchen eine Funktion, die den Input $i$ gegen den regulären Ausdruck $r$ *matched*:

```
match :: RegExp a -> [a] -> Bool
match r i = i ∈? L(r)                    -- Pseudo-Code!
```

Da $L(r)$ potentiell unendlich ist, scheidet explizites Aufzählen aus.

**Idee:** Input $i$ hat die Form $[c_0, c_1, c_2, \ldots, c_n]$. Beginnend mit $k = 0$, immer wenn wir Symbol $c_k$ gesehen haben, transformiere $r$ in $r'$ so, daß $r'$ noch die restliche Eingabe $[c_{k+1}, c_{k+2}, \ldots, c_n]$ akzeptiert.

Diese Transformation von $r$ nennen wir die **Ableitung** von $r$ nach $c_k$:

$$r' = \partial_{c_k}(r) \ .$$

Nach unserer Idee muß für die von der Ableitung akzeptierte Sprache gelten

$$L(\partial_{c_k}(r)) \quad = \quad \{\ cs \mid c_k{:}cs \ \in L(r)\ \} \tag{1}$$

Wenn wir $\partial_c(r)$ für jedes Symbol $c$ und regulären Ausdruck $r$ berechnen können, ist unser *regular expression matching* Problem weitgehend gelöst, denn bspw.

$$\begin{aligned}
[c_1, c_2, c_3] \in L(r) \quad &\Leftrightarrow \quad [c_2, c_3] \in L\big(\partial_{c_1}(r)\big) \\
&\Leftrightarrow \quad [c_3] \in L\big(\partial_{c_2}(\partial_{c_1}(r))\big) \\
&\Leftrightarrow \quad [] \in L\big(\partial_{c_3}(\partial_{c_2}(\partial_{c_1}(r)))\big)
\end{aligned}$$

Damit haben wir unser Problem gelöst, wenn wir entscheiden können, ob ein regulärer Ausdruck die leere Eingabe `[]` akzeptiert.

Diesen sog. *nullable* Test kann man tatsächlich sehr einfach ausführen:

```
-- nullable r: akzeptiert r die Eingabe []?
nullable :: RegExp a -> Bool
nullable RxNone       = False
nullable RxEpsilon    = True
nullable (RxSymbol _) = False
nullable (RxStar r)   =
nullable (RxPlus r)   =
nullable (RxOpt r)    =
nullable (RxSeq r s)  =
nullable (RxAlt r s)  =
```

```
-- nullable r: akzeptiert r die Eingabe []?
nullable :: RegExp a -> Bool
nullable RxNone       = False
nullable RxEpsilon    = True
nullable (RxSymbol _) = False
nullable (RxStar _)   = True
nullable (RxPlus r)   = nullable r
nullable (RxOpt _)    = True
nullable (RxSeq r s)  = nullable r && nullable s
nullable (RxAlt r s)  = nullable r || nullable s
```

Sei

```
derive :: RegExp a -> a -> RegExp a
```

die Haskell-Implementation von $\partial$, also $\partial_x(r) =$ `derive` $r\ x$, dann ist die Spezifikation von `match` ein Einzeiler:

```
-- match r i: regular expression match: akzeptiert r die Eingabe i?
match :: RegExp a -> [a] -> Bool
match r i =
```

```
-- match r i: regular expression match: akzeptiert r die Eingabe i?
match :: RegExp a -> [a] -> Bool
match r i = nullable (foldl derive r i)
```

**Beispiel 11.9**

`foldl` garantiert die korrekte Ableitung von `r`:

$$
\begin{aligned}
& \texttt{foldl derive r [}c_0\texttt{, }c_1\texttt{, }c_2\texttt{]} \\
\xrightarrow[\texttt{foldl}]{} \; & \texttt{foldl derive (derive r }c_0\texttt{) [}c_1\texttt{, }c_2\texttt{]} \\
\xrightarrow[\texttt{foldl}]{} \; & \texttt{foldl derive (derive (derive r }c_0\texttt{) }c_1\texttt{) [}c_2\texttt{]} \\
\xrightarrow[\texttt{foldl}]{} \; & \texttt{foldl derive (derive (derive (derive r }c_0\texttt{) }c_1\texttt{) }c_2\texttt{) []} \\
\xrightarrow[\texttt{foldl}]{} \; & \texttt{derive (derive (derive r }c_0\texttt{) }c_1\texttt{) }c_2
\end{aligned}
$$

**Frage:** Verhält sich `match r []` korrekt, d.h. wird auch die leere Eingabe `[]` korrekt *gematched*?

Es verbleibt die Implementation von `derive`:

```
-- derive r x: Ableitung des regulaeren Ausdrucks r nach Symbol x
derive :: Eq a => RegExp a -> a -> RegExp a
derive RxNone       _              = rxNone

derive RxEpsilon    _              = rxNone

derive (RxSymbol c) x | c == x     = rxEpsilon
                      | otherwise  = rxNone

derive (RxStar r)   x              = rxSeq (derive r x) (rxStar r)

derive (RxPlus r)   x              = rxSeq (derive r x) (rxStar r)

derive (RxOpt r)    x              = derive r x

derive (RxSeq r r') x | nullable r = rxAlt (rxSeq (derive r x) r') (derive r' x)
                      | otherwise  = rxSeq (derive r x) r'

derive (RxAlt r r') x              = rxAlt (derive r x) (derive r' x)
```

Man kann sich überzeugen, daß diese Definition von `derive` tatsächlich unsere Idee implementiert (s. Gleichung (1)):

**Beispiel 11.10**

$$
\begin{aligned}
L(\partial_x(\varepsilon)) &= \{\ cs \mid x{:}cs \in L(\varepsilon)\ \} \\
&= \{\ cs \mid x{:}cs \in \{\,[]\,\}\ \} \\
&= \emptyset \\
&= L(\varnothing) \\
L(\partial_x(x)) &= \{\ cs \mid x{:}cs \in L(x)\ \} \\
&= \{\ cs \mid x{:}cs \in \{\,[x]\,\}\ \} \\
&= \{\,[]\,\} \\
&= L(\varepsilon) \\
L(\partial_x(r \mid s)) &= \{\ cs \mid x{:}cs \in L(r \mid s)\ \} \\
&= \{\ cs \mid x{:}cs \in L(r) \cup L(s)\ \} \\
&= \{\ cs \mid x{:}cs \in L(r)\ \} \cup \{\ cs \mid x{:}cs \in L(s)\ \} \\
&= L(\partial_x(r)) \cup L(\partial_x(s)) \\
&= L(\partial_x(r) \mid \partial_x(s))
\end{aligned}
$$

```
> let r = rxAlt (rxSymbol 1) (rxStar (rxSeq (rxSymbol 0) (rxSymbol 1))
r :: RegExp Integer
> r
1 | (0, 1)*
it :: RegExp Integer
> derive r 1
eps
it :: RegExp Integer
> derive r 0
1, (0, 1)*
it :: RegExp Integer
> derive r 42
{}
it :: RegExp Integer
> let r' = derive r 0
r' :: RegExp Integer
> derive r' 1
(0, 1)*
it :: RegExp Integer
> nullable it
True
it :: Bool
```

# 12 Typinferenz

In Haskell ist es bis auf wenige Ausnahmefälle nicht notwendig, Werte und Ausdrücke mit ihren jeweiligen **Typen** zu annotieren. Stattdessen **leitet der Compiler die Typisierung automatisch ab** (*type inference*).

**Programmierer** profitieren von der Typprüfung ihrer Programme:

- *„Well-typed programs do not 'go wrong'."* (Robin Milner): ein typkorrektes Programm wendet Funktionen nur auf Werte an, für die sie auch definiert wurden.
- Viele ansonsten schwer zu entdeckende Tipp- oder auch einfache logische Fehler in Programmen werden durch den **Type-Checker** zur Compile-Zeit erkannt.
  **Beispiel:**

```
foldr (\x xs -> (x, 0):xs) []  vs.  foldr (\x xs -> [x, 0]:xs) []
```

Aber auch der **Compiler und die Laufzeitumgebung** ziehen Vorteile aus der Typisierung:

- Für ein typkorrektes Programm muß das Laufzeitsystem der Sprache keine Tests auf die Typkorrektheit – bspw. vor Applikation einer Closure ($\approx$ Funktion) ein *type tag* prüfen – ausführen ($\Rightarrow$ kürzere Laufzeiten).
- Der Compiler muß entsprechend keinen Code für solche Tests generieren ($\Rightarrow$ kompakter Objekt-Code).

## 12.1 Polymorphie

Immer dann, wenn eine Funktion $f$ keinerlei Annahmen (wie die Anwendbarkeit eines Gleichheitstest `(==)`, arithmetischer Operationen `(+)`, `(/)`, etc.) über eines oder mehrere (alle) ihrer Argumente macht, erwarten wir, daß ein *beliebiger Typ* $\alpha$ als aktuelles Argument auftreten darf: $f$ ist dann **parametrisch polymorph**.

**Beispiel 12.1**
Die Identitätsfunktion `id` $= \lambda\, x.x$ ist auf Argumente beliebiger Typen $\alpha$ anwendbar. In

```
   id 3  =  3
 id 'a'  =  'a'
id (3,'a')  =  (3,'a')
```

wurde `id` als Funktion des Typs `Integer -> Integer`, `Char -> Char` und `(Integer, Char) -> (Integer, Char)` benutzt. Damit lautet die vollständige Typisierung von `id`

`id` :: $\forall\alpha.\alpha$ `->` $\alpha$

Die Allquantifizierung von Typvariablen werden wir in Zukunft nicht mehr explizit notieren. Alle **Typvariablen sind implizit universell quantifiziert**. Typvariablennamen entnehmen wir dem griechischen Alphabet.

Eine konkrete **Instantiierung des polymorphen Typs** von `id` (bspw. bei Anwendung auf das Argument `(3,'a')` bzw. `length`) erhält man durch die **konsistente Subsitution von Typvariablen durch einen Typausdruck** (hier: `(Integer, Char)` bzw. `[`$\beta$`] -> Integer`).

**Beispiel 12.2**
Innerhalb eines Ausdrucks können durchaus verschiedene Instantiierungen desselben polymorphen Objektes auftreten. In

```
id (length (id [1,2,3]))
```

tritt `id` in den Instantiierungen `[Integer] -> [Integer]` und `Integer -> Integer` auf.

### 12.1.1 Ad-hoc vs. parametrische Polymorphie

Ein polymorphes Typsystem erlaubt die Anwendung eines **Namens** $f$ in verschiedenen Typinstantiierungen. FPLs sind **parametrisch polymorph:**

① **Overloading** oder **Ad-hoc Polymorphie**. Der Compiler bzw. das Laufzeitsystem wählt aufgrund des Typkontextes eine adäquate Implementation für den Namen $f$ aus (s. Overloading etwa in C++ oder Haskells Typ-Klassen, Kapitel 10).

② **Parametrische Polymorphie**. Da keinerlei Annahmen über die Anwendbarkeit spezifischer Operationen gemacht werden, kann unabhängig von der konkreten Typinstantiierung immer genau dasselbe Programm für $f$ ausgeführt werden.

## 12.2 Automatische Typinferenz

Die Typinferenz-Komponente eines Compilers

① prüft, ob ein Programm nach den Typregeln der Sprache korrekt typisiert ist (*type check*), und
② (sollte ① erfüllt sein) leitet automatisch den Typ jedes Ausdrucks innerhalb dieses Programms ab (*type inference*).

Mittels Intuition und „scharfem Hinsehen“ lassen sich die Punkte ① und ② für viele Ausdrücke und Funktionsdefinitionen auch intuitiv ableiten. Der später vorgestellte Inferenzalgorithmus orientiert sich an dieser Intuition.

**Beispiel 12.3**
Typinferenz für die Funktion `foldr`, definiert durch

$$\texttt{foldr}\ f\ z = g\ \texttt{where}\ g\ [] = z$$
$$g\ (x\texttt{:}xs) = f\ x\ (g\ xs)$$

$g$ operiert offensichtlich auf Listen und hat daher den Typ `[`$\alpha$`]` `->` $\beta$. Sowohl $z$ als auch $f\ x\ (g\ xs)$ haben dann den Typ $\beta$. Da $g\ xs$ `::` $\beta$, muß $f$ vom Typ $\alpha$ `->` $\beta$ `->` $\beta$ sein. Also:

$$\texttt{foldr :: (}\alpha\ \texttt{->}\ \beta\ \texttt{->}\ \beta\texttt{)}\ \texttt{->}\ \beta\ \texttt{->}\ \texttt{[}\alpha\texttt{]}\ \texttt{->}\ \beta$$

Diese intuitive *type inference* umfäßt dabei die

① Bestimmung eines Typs für einen Ausdruck an sich, wie etwa im letzten Beispiel für $g$. Im allgemeinen wird dieser Typausdruck Typvariablen beinhalten.

② **Bestimmung eines Typs für einen Ausdruck als Teilausdruck bereits typisierter Ausdrücke**, wie eben für $z$ und $f$ geschehen.

Soll der Gesamtausdruck typkorrekt sein, dürfen sich die dabei abgeleiteten Typausdrücke nicht widersprechen, d.h. sie müssen durch geeignete Substiution von Typvariablen in dieselbe Form gebracht werden können.

### 12.2.1 Kernsprache für Typinferenz

Um den Rahmen der Betrachtungen nicht zu sprengen, entwickeln wir hier einen Typinferenz-Algorithmus für den erweiterten $\lambda$-Kalkül, wie wir ihn Kapitel 3 schon ausführlich besprochen haben.

Da FPLs wie Haskell durch Abbildung auf eine derartige Sprache definiert sind, verlieren wir an dieser Stelle nichts. Die Typinferenz wird angestoßen, nachdem der Quelltext in die Kernsprache überführt wurde.

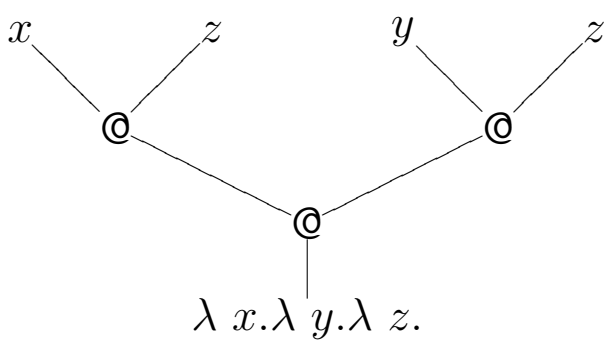
Wir erweitern den $\lambda$-Kalkül aus Kapitel 3 um lokale Definitionen via `let ... in` und erhalten

| | | | |
|---|---|---|---|
| $e$ | $\rightarrow$ | `c` | Konstanten, vordef. Funktionen |
| | $\mid$ | $v$ | Variablen |
| | $\mid$ | $(e\ e)$ | Applikation (Juxtaposition) |
| | $\mid$ | $(\lambda v.e)$ | $\lambda$-Abstraktion |
| | $\mid$ | `let` $v$ `=` $e$ | lokale Definitionen |
| | | $\ldots$ | |
| | | $v$ `=` $e$ | |
| | | `in` $e$ | |

Um die einzelnen Teilausdrücke eines Ausdrucks mit ihren jeweiligen Typen annotieren zu können, werden wir diese in Baumform notieren. Applikationsknoten werden durch @, Abstraktionen durch $\lambda$ gekennzeichnet.

**Beispiel 12.4**

Der Ausdruck $(\lambda\ x.\lambda\ y.\lambda\ z.x\ z\ (y\ z))$ hat in der Baumnotation die Form



Jeder Knoten eines derartigen Baumes korrespondiert mit einem Teilausdruck des Originalausdrucks und sollte daher mit einer Typisierung versehen werden können. Dazu annotieren wir die Ausdrucksbäume mit den entsprechenden Typausdrücken.

**Beispiel 12.5**

Seien $T_0, \ldots, T_7$ Typen, dann annotieren wir den Baum aus Beispiel 12.4 wie folgt:

$$\dfrac{\dfrac{\dfrac{x :: T_0 \qquad z :: T_1}{T_4}\,@ \qquad \dfrac{y :: T_2 \qquad z :: T_3}{T_5}\,@}{\dfrac{T_6}{T_7}\,\lambda xyz}\,@}{}$$

Aus der **Inferenzregel für die Funktionsapplikation** $f\ e$, nämlich

$$\dfrac{f :: T_0 \qquad e :: T_1}{T_2}\,@$$

können wir ableiten, daß $T_0$ ein Funktionstyp der Form $\alpha$ -> $\beta$ sein muß. Damit liegen dann auch $T_1$ mit $\alpha$ und $T_2$ mit $\beta$ fest, also

$$\dfrac{f :: \alpha \text{ -> } \beta \qquad e :: \alpha}{f\ e :: \beta}\,@$$

Für unser Beispiel erhalten wir mittels der @-Regel die folgenden drei **Typgleichungen**, die erfüllt sein müssen, wenn die $T_0 \ldots T_7$ eine korrekte Typisierung darstellen sollen:

$$\begin{aligned} T_0 &= T_1 \text{ -> } T_4 \\ T_2 &= T_3 \text{ -> } T_5 \\ T_4 &= T_5 \text{ -> } T_6 \end{aligned}$$

Wir substituieren $T_0, T_2$ und $T_4$ im annotieren Baum, um dieses abgeleitete Wissen darzustellen:

$$\dfrac{\dfrac{\dfrac{x :: T_1 \text{ -> } T_5 \text{ -> } T_6 \qquad z :: T_1}{T_5 \text{ -> } T_6}\,@ \qquad \dfrac{y :: T_3 \text{ -> } T_5 \qquad z :: T_3}{T_5}\,@}{\dfrac{T_6}{T_7}\,\lambda xyz}\,@$$

Für die Typinferenz des $\lambda$-Knotens ist es offensichtlich, daß $T_7$ die Form

$$(T_1 \text{ -> } T_5 \text{ -> } T_6) \text{ -> } (T_3 \text{ -> } T_5) \text{ -> } \ldots$$

besitzen muß (denn $x :: T_1$ -> $T_5$ -> $T_6$ und $y :: T_3$ -> $T_5$).

Wie ist aber mit den beiden Auftreten der Variable $z$ und den jeweiligen Typisierungen $T_1$ und $T_3$ zu verfahren?

Unter der Annahme[11], daß die beiden (durch das $\lambda$ gebundenen) Auftreten von $z$ den gleichen Typ besitzen, könnten wir die Gleichung

$$T_1 \quad = \quad T_3$$

zu unserem Gleichungssystem hinzunehmen und letztendlich die folgende Typisierung ableiten:

$$\dfrac{\dfrac{\dfrac{x :: T_1 \texttt{ -> } T_5 \texttt{ -> } T_6 \qquad z :: T_1}{T_5 \texttt{ -> } T_6}\,@ \qquad \dfrac{y :: T_1 \texttt{ -> } T_5 \qquad z :: T_1}{T_5}\,@}{T_6}\,@}{(T_1 \texttt{ -> } T_5 \texttt{ -> } T_6) \texttt{ -> } (T_1 \texttt{ -> } T_5) \texttt{ -> } T_1 \texttt{ -> } T_6}\,\lambda xyz$$

Ersetzen wir die $T_i$ wie gewohnt durch griechische Buchstaben, lautet das Ergebnis unserer Typinferenz also schließlich $(\alpha \texttt{ -> } \beta \texttt{ -> } \gamma) \texttt{ -> } (\alpha \texttt{ -> } \beta) \texttt{ -> } \alpha \texttt{ -> } \gamma$.

Das Haskell-System leitet denselben Typ ab:

```
> :t \x y z -> x z (y z)
(a -> b -> c) -> (a -> b) -> a -> c
```

[11] Diese Annahme wird sich später als wichtig erweisen.

Die Annahme, daß in der Abstraktion $(\lambda z.e)$ die Vorkommen von $z$ in $e$ jeweils den gleichen Typ besitzen müssen, mag für ein polymorphes Typsystem sehr einschränkend erscheinen. Das nächste Beispiel erläutert die Annahme weiter.

**Beispiel 12.6**

Sei $F = \lambda f.\lambda a.\lambda b.\lambda c.c\ (f\ a)\ (f\ b)$. Die initiale Typisierung ergibt

$$\dfrac{\dfrac{c :: T_2 \qquad \dfrac{f :: T_0 \qquad a :: T_1}{T_3}\,@}{T_6}\,@ \qquad \dfrac{f :: T_4 \qquad b :: T_5}{T_7}\,@}{\dfrac{T_8}{T_9}\,\lambda f a b c}\,@$$

Allein mit der Inferenzregel für die Funktionsapplikation können wir die Gleichungen

$$\begin{aligned} T_0 &= T_1 \text{ -> } T_3 \\ T_2 &= T_3 \text{ -> } T_6 \\ T_4 &= T_5 \text{ -> } T_7 \\ T_6 &= T_7 \text{ -> } T_8 \end{aligned}$$

ableiten.

Wenn wir jetzt fordern, daß die beiden Vorkommen von $f$ denselben Typ besitzen, erhalten wir zusätzlich

$$T_0 = T_4 \quad \Rightarrow \quad T_1 \text{ -> } T_3 = T_5 \text{ -> } T_7 \quad \Rightarrow \quad \begin{array}{l} T_1 = T_5 \\ T_3 = T_7 \end{array}$$

Allgemein sind zwei **konstruierte Typen sind nur dann gleich**, wenn sämtliche **korrespondieren Teiltypen gleich** sind.
Unsere Typisierung ändert sich damit in

$$\dfrac{\dfrac{c :: T_3 \text{ -> } T_3 \text{ -> } T_8 \qquad T_3}{T_3 \text{ -> } T_8}\,@ \qquad \dfrac{\dfrac{f :: T_1 \text{ -> } T_3 \qquad a :: T_1}{T_3}\,@ \qquad \dfrac{f :: T_1 \text{ -> } T_3 \qquad b :: T_1}{T_3}\,@}{T_8}\,@}{(T_1 \text{ -> } T_3) \text{ -> } T_1 \text{ -> } T_1 \text{ -> } (T_3 \text{ -> } T_3 \text{ -> } T_8) \text{ -> } T_8}\,\lambda f a b c$$

Der abgeleitete Typ für $F$ ist also

$$\underbrace{(\alpha \text{ -> } \beta)}_{f} \text{ -> } \underbrace{\alpha}_{a} \text{ -> } \underbrace{\alpha}_{b} \text{ -> } \underbrace{(\beta \text{ -> } \beta \text{ -> } \gamma)}_{c} \text{ -> } \gamma$$

**Beispiel 12.6**

Durch die Forderung nach einem eindeutigen Typ für $f$ wurde für $a$ und $b$ jeweils der Typ $T_1$ abgeleitet.

Allerdings gibt es Ausdrücke $F\ f\ a\ b$ (partielle Anwendung von $F$ auf nur drei Argumente), die durchaus Sinn machen, wenn $a$ und $b$ verschiedene Typen besitzen. Etwa scheint

$$F\ \texttt{id}\ 3\ \texttt{'a'} \qquad (\star)$$

die Funktion zu sein, die ($c$ `3 'a'`) berechnet, wenn sie auf ein Argument $c$ :: `Integer -> Char ->` $\gamma$ angewandt wird.

Andererseits enthält

$$F\ \texttt{ord}\ 3\ \texttt{'a'}$$

eindeutig einen Typisierungsfehler, denn dies würde zur Auswertung von `ord 3` führen (mit `ord :: Char -> Integer`).

**Fazit:** Die Typisierung von $F$ muß sicherstellen, daß $F$ als Teil *jeden* Ausdrucks korrekt ausgewertet werden kann („*Well-typed programs do not 'go wrong'*“), nicht nur in Kontexten wie ($\star$), die „zufällig“ korrekt typisiert sind. Wir benötigen die Forderung:

> Alle **gebundenen Vorkommen derselben Variablen** in einer $\lambda$-Abstraktion werden jeweils **identisch typisiert**.

## Zusammenfassung der Regeln zur Typinferenz:

① Inferenzregel für die Applikation $f\ e$: $\dfrac{f :: \alpha \texttt{ -> } \beta \qquad e :: \alpha}{f\ e :: \beta}$ @

② Aus der Gleichheit konstruierter Typen folgt die Gleichheit ihrer Teiltypen.

③ Alle gebundenen Vorkommen derselben Variablen in einer $\lambda$-Abstraktion werden jeweils identisch typisiert.

**Beispiel 12.7**

Die Funktion $\lambda n.\lambda a.\lambda b.b\ n\ (n\ a\ b)$ enthält einen Typisierungsfehler. Typinferenz kann diesen Fehler aufdecken.

Die initialisiere Typisierung in Baumnotation:

$$\dfrac{\dfrac{\dfrac{b :: T_2 \qquad n :: T_3}{T_6}\,@ \qquad \dfrac{n :: T_0 \qquad a :: T_1}{T_4}\,@}{T_7}\,@ \qquad \dfrac{b :: T_5}{T_7}\,@}{\dfrac{T_8}{T_9}\,\lambda nab}$$

Die Typinferenz-Regel ① zur Applikation ergibt das Gleichungssystem

$$\begin{array}{lcl@{\qquad\qquad}lcl} T_0 & = & T_1 \text{ -> } T_4 & T_3 & = & T_0 \\ T_2 & = & T_3 \text{ -> } T_6 & T_5 & = & T_2 \\ T_4 & = & T_5 \text{ -> } T_7 & T_9 & = & T_0 \text{ -> } T_1 \text{ -> } T_2 \text{ -> } T_8 \\ T_6 & = & T_7 \text{ -> } T_8 & & & \end{array}$$

Nach Elimination von $T_4$ und $T_6$ entsteht daraus

$$\begin{array}{lcl@{\qquad\qquad}lcl} T_0 & = & T_1 \text{ -> } T_5 \text{ -> } T_7 & T_3 & = & T_0 \\ T_2 & = & T_3 \text{ -> } T_7 \text{ -> } T_8 & T_5 & = & T_2 \\ & & & T_9 & = & T_0 \text{ -> } T_1 \text{ -> } T_2 \text{ -> } T_8 \end{array}$$

Jetzt wird das Problem offensichtlich. Die Gleichungen enthalten eine zirkuläre Abhängigkeit:

$$\begin{array}{lcl} T_0 & \underset{(T_5=T_2)}{=} & T_1 \text{ -> } T_2 \text{ -> } T_7 \\ & \underset{(T_2 \text{ einsetzen})}{=} & T_1 \text{ -> } (T_3 \text{ -> } T_7 \text{ -> } T_8) \text{ -> } T_7 \\ & \underset{(T_3=T_0)}{=} & T_1 \text{ -> } (T_0 \text{ -> } T_7 \text{ -> } T_8) \text{ -> } T_7 \end{array}$$

Durch die Zirkularität gibt es keine endliche Darstellung für den Typ $T_0$ (und damit auch nicht für den Ergebnistyp $T_9$). Derartige **unendliche Typen** weist unser *type checker* zurück.

**Beispiel 12.7**

Die Fehlermeldung des Haskell Type-Checkers ist entsprechend:

```
> :t \n a b -> b n (n a b)
ERROR: Type error in application
*** Expression     : b n (n a b)
*** Term           : n
*** Type           : a -> (b -> c -> d) -> c
*** Does not match : b
*** Because        : unification would give infinite type
```

**NB.** Die Haskell-Typvariable `b` entspricht hier dem Typ $T_0$ der oben diskutierten Typinferenz.

## 12.2.2 Typisierung von `let...in`

Im Gegensatz zu der eben notwendigen Einschränkung in der Inferenz-Regel ③ bei $\lambda$-gebundenen Variablen, erwarten wir in einem `let`-Ausdruck wie

$$\texttt{let } f \texttt{ = } e \texttt{ in } \ldots f \ldots f \ldots$$

daß $f$ durchaus in verschiedenen Typinstantiierungen auftreten darf. Wäre dies nicht so, würde der Polymorphiebegriff an sich absurd erscheinen.

**Beispiel 12.8**
Der Ausdruck ($S$ und $K$ sind die sog. Schönfinkel-Kombinatoren)

$$\begin{array}{ll} \texttt{let} & S \;=\; \lambda x.\lambda y.\lambda z.x\ z\ (y\ z) \\ & K \;=\; \lambda x.\lambda y.x \\ \texttt{in} & S\ \ K\ K \end{array}$$

hat die initiale Typisierung

$$\dfrac{\dfrac{S :: T_6 \texttt{ -> } T_7 \texttt{ -> } T_8 \qquad K :: T_6}{T_7 \texttt{ -> } T_8}\,@ \qquad K :: T_7}{T_8}\,@$$

$$\dfrac{S :: TS \qquad K :: TK}{T_9}\,\texttt{let } S, K$$

Da für $e :: \alpha$ auch (`let ... in` $e$) $:: \alpha$ gilt, erhalten wir als erste Gleichung aus der `let`-Regel sofort

$$T_8 \quad = \quad T_9$$

Die Typisierung von $S$ und $K$ haben wir hier nur angedeutet. Die Typinferenz bestimmt sie wie folgt:

$$\begin{aligned} S :: TS &= (T_0 \texttt{ -> } T_1 \texttt{ -> } T_2) \texttt{ -> } (T_0 \texttt{ -> } T_1) \texttt{ -> } T_0 \texttt{ -> } T_2 \\ K :: TK &= T_3 \texttt{ -> } T_4 \texttt{ -> } T_3 \end{aligned}$$

Insbesondere erwarten wir, daß uns Polymorphie erlaubt, die beiden Vorkommen $K :: T_6$ und $K :: T_7$ auch mit $T_6 \neq T_7$ zu typisieren (natürlich müssen $T_6$ als auch $T_7$ trotzdem Typinstantiierungen von $TK$ sein).

Wir erreichen dies, indem wir (**Inferenzregel für** `let`)

④ für jedes Vorkommen eines durch `let` eingeführten Namens **frische Typvariablen** vergeben.

Damit sind die Vorkommen von $K$ entkoppelt:

$$\dfrac{\dfrac{S :: (T_{10} \texttt{ -> } T_{11} \texttt{ -> } T_{12}) \texttt{ -> } (T_{10} \texttt{ -> } T_{11}) \texttt{ -> } T_{10} \texttt{ -> } T_{12} \quad K :: T_{13} \texttt{ -> } T_{14} \texttt{ -> } T_{13}}{T_7 \texttt{ -> } T_8} @ \quad K :: T_{15} \texttt{ -> } T_{16} \texttt{ -> } T_{15}}{T_8} @$$

$$\dfrac{S :: TS \quad K :: TK}{T_8}\ \texttt{let}\ S, K$$

Das Typgleichungssystem wird mit Hilfe der Inferenzregeln normal aufgelöst:

Vergabe frischer Variablen (④) für die Vorkommen von $S$ und $K$, Einsetzen von $TS$ und $TK$:

$$\begin{array}{rcl} (T_6 \texttt{ -> } T_7 \texttt{ -> } T_8) & = & (T_{10} \texttt{ -> } T_{11} \texttt{ -> } T_{12}) \texttt{ -> } (T_{10} \texttt{ -> } T_{11}) \texttt{ -> } T_{10} \texttt{ -> } T_{12} \\ T_6 & = & T_{13} \texttt{ -> } T_{14} \texttt{ -> } T_{13} \\ T_7 & = & T_{15} \texttt{ -> } T_{16} \texttt{ -> } T_{15} \end{array}$$

Daraus ergibt sich mit ② bzgl. Typkonstruktor ->:

$$\begin{array}{rcl} T_6 & = & T_{10} \texttt{ -> } T_{11} \texttt{ -> } T_{12} \\ T_7 & = & T_{10} \texttt{ -> } T_{11} \\ T_8 & = & T_{10} \texttt{ -> } T_{12} \end{array}$$

Wiederum ist ② bzgl. -> anwendbar und ergibt:

$$\begin{array}{rcl} T_{10} & = & T_{13} \quad = \quad T_{12} \quad = \quad T_{15} \\ T_{11} & = & T_{14} \quad = \quad T_{16} \texttt{ -> } T_{15} \end{array}$$

Wir sehen schließlich, daß der Originalausdruck tatsächlich von der Polymorphie von $K$ Gebrauch macht:

$$\begin{array}{rcl} T_6 & = & T_{10} \texttt{ -> } (T_{16} \texttt{ -> } T_{10}) \texttt{ -> } T_{10} \\ T_7 & = & T_{10} \texttt{ -> } T_{16} \texttt{ -> } T_{10} \end{array}$$

Der Typ des gesamten `let`-Ausdrucks wird letztendlich zu $T_8 = T_{10}$ -> $T_{10}$ oder auch $\alpha$ -> $\alpha$ abgeleitet ($S\ K\ K\ =$ `id`. **Nachrechnen!**).

### 12.2.3 Typisierung weiterer Haskell-Konstrukte

Die Typisierungsregeln der weiteren Haskell-Sprachkonstrukte ähneln Regel ① für die Applikation. Durch Hinzunahme dieser zusätzlichen Inferenzregeln ändert sich am Prinzip der Typinferenz nichts:

**Beispiel 12.9**

$$\frac{e_1 :: \alpha \qquad e_2 :: \beta}{(e_1, e_2) :: (\alpha, \beta)}\ \texttt{(,)}$$

$$\frac{e :: (\alpha, \beta)}{\texttt{fst}\ e :: \alpha}\ \texttt{fst}$$

$$\frac{e :: \texttt{Bool} \qquad e_1 :: \alpha \qquad e_2 :: \alpha}{\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: \alpha}\ \texttt{if}$$

$$\frac{f :: \alpha \texttt{ -> } \beta \qquad g :: \beta \texttt{ -> } \gamma}{g\ .\ f :: \alpha \texttt{ -> } \gamma}\ \texttt{(.)}$$

**Frage:** Wie lautet die Typisierungsregel für `case` $e$ `of` $p_1$ `->` $e_1$ $\ldots$ $p_n$ `->` $e_n$ ?

# Referenzen

Simon L. Peyton-Jones (1987). *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice Hall International.

# 13 Lazy Evaluation

Wir haben schon mehrmals erwähnt, daß die Ausdrucksauswertung in Haskell *nicht-strikt* geschieht, bisher aber nur intuitiv erklärt, was nicht-strikte Auswertung tatsächlich bedeutet. Dieser Abschnitt beleuchtet **lazy evaluation** als Auswertungstechnik und zeigt einige (oft nicht offensichtliche) Konsequenzen sowie Programmiertechniken.

## 13.1 Normal Order Reduction

Die Reihenfolge der Reduktionsschritte ($\rightarrow\!\triangleright$) für einen Ausdruck ist vornherein *nicht* eindeutig festgelegt. Zwei mögliche Reduktionsstrategien

① *applicative order reduction* (*eager evaluation*) und
② *normal order reduction (+ sharing)* (*lazy evaluation*)

betrachten wir hier kurz.

Eine Reduktionsstrategie hat dabei für einen Ausdruck $e$ lediglich festzulegen, welcher von im allg. mehreren **reduzierbaren Teilausdrücken** (**Redex**, *reducible expression*) von $e$ als nächster reduziert wird.

*Applicative* und *normal order reduction* unterscheiden sich genau in dieser **Auswahl des nächsten Redex**.

**Beispiel 13.1**

Seien `fst` und `sqr` definiert durch:

$$\begin{array}{lcl} \texttt{fst}\ (x,y) & = & x \\ \texttt{sqr}\ x & = & x \times x \end{array}$$

Der auszuwertende Ausdruck $e$ sei `fst (sqr 4, sqr 2)`.

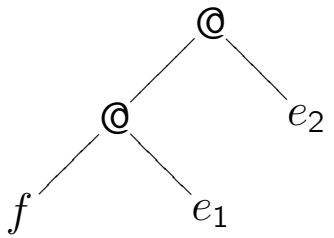
① *Applicative order reduction* wählt jeweils den **innersten Redex** zur Reduktion aus:

$$\begin{array}{lcll} \texttt{fst (sqr 4, sqr 2)} & \rightarrow & \texttt{fst (4} \times \texttt{4, sqr 2)} & (\texttt{sqr}) \\ & \rightarrow & \texttt{fst (16, sqr 2)} & (\boxtimes) \\ & \rightarrow & \texttt{fst (16, 2} \times \texttt{2)} & (\texttt{sqr}) \\ & \rightarrow & \texttt{fst (16, 4)} & (\boxtimes) \\ & \rightarrow & \texttt{16} & (\texttt{fst}) \end{array}$$

② *Normal order reduction* wählt jeweils den **äußersten Redex** zur Reduktion aus:

$$\begin{array}{lcll} \texttt{fst (sqr 4, sqr 2)} & \rightarrow & \texttt{sqr 4} & (\texttt{fst}) \\ & \rightarrow & \texttt{4} \times \texttt{4} & (\texttt{sqr}) \\ & \rightarrow & \texttt{16} & (\boxtimes) \end{array}$$

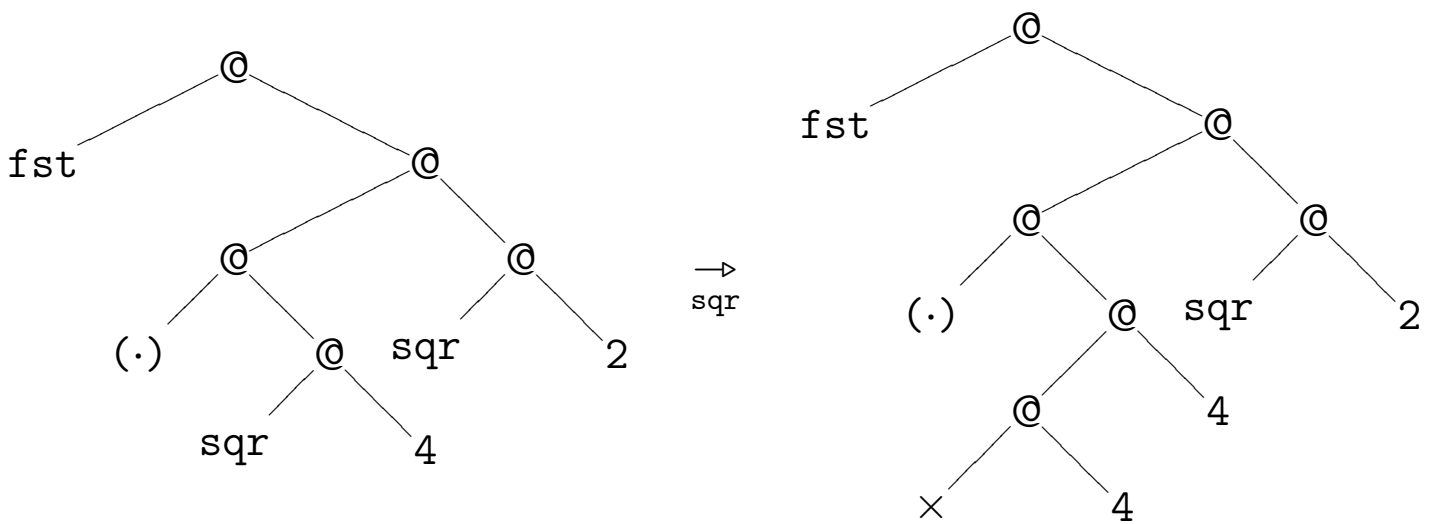
In der Baumnotation für Ausdrücke (s. Kapitel 2), in der die *curried* Applikation $f\ e_1\ e_2$ durch

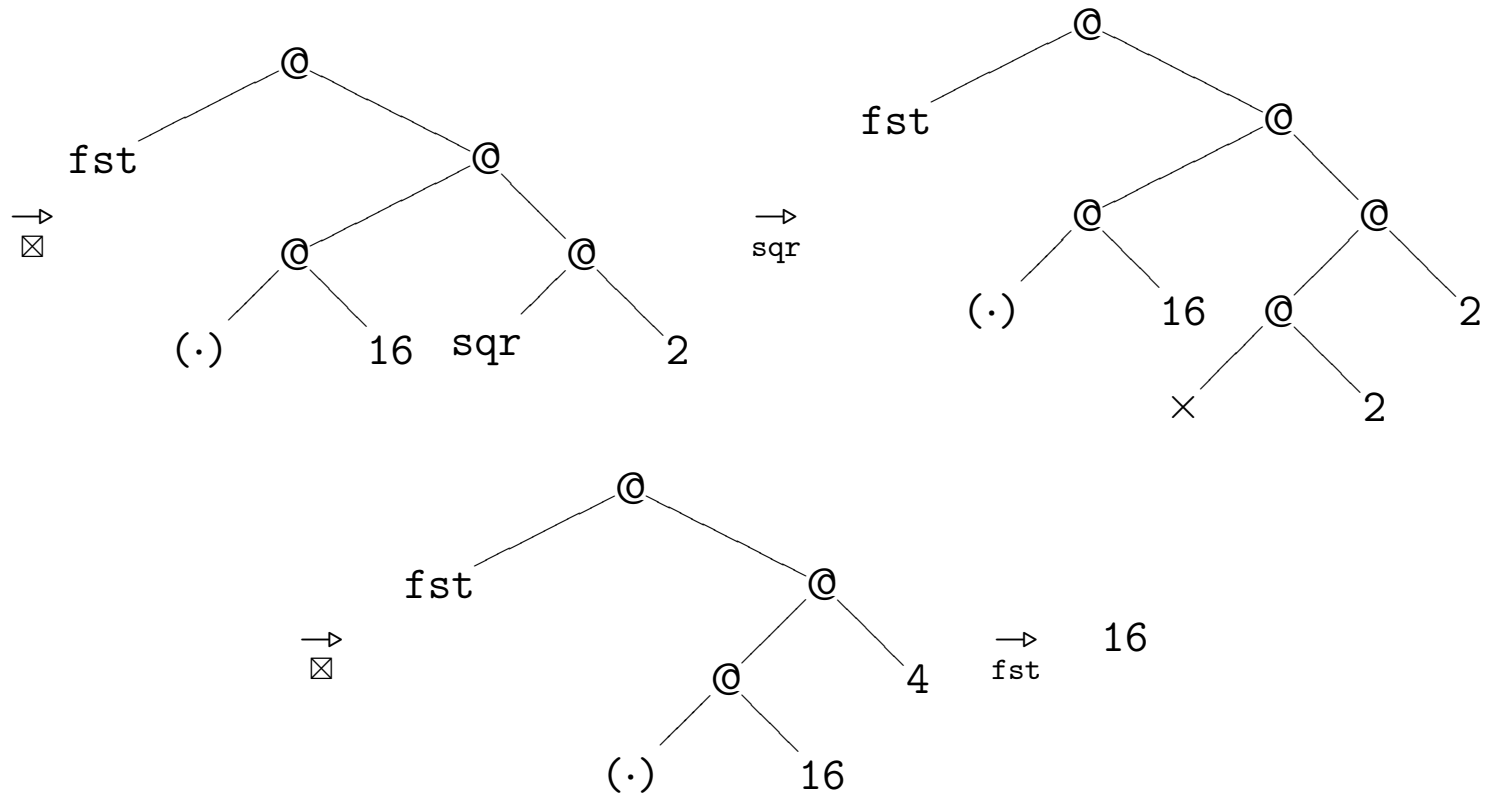



repräsentiert wird, reduziert *applicative order reduction* zuerst die inneren Teilbäume $e_1$ und $e_2$ während *normal order reduction* zuvor den äußeren linken Knoten $f$ reduziert.

**Beispiel 13.2**

① *Applicative order reduction* von `fst (sqr 4, sqr 2)`. Der Knoten `(·)` bezeichnet den Tupelkonstruktor:

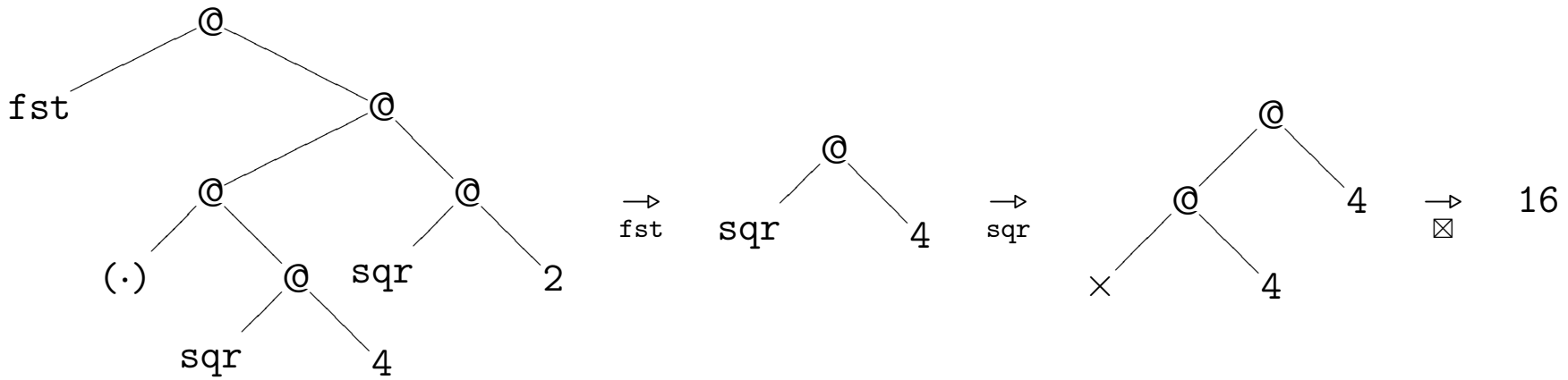





- **Faustregel:** Redex wird durch den *leftmost innermost* Knoten bestimmt.

**Beispiel 13.2**

② *Normal order reduction* von `fst (sqr 4, sqr 2)`. Der **nächste zu reduzierende Redex ist jeweils der Knoten links außen** im Baum (vgl. Beispiel 9.3.2, Funktion `leftmost`):




- **Faustregel:** Redex wird durch den *leftmost outermost* Knoten bestimmt.

**Beispiel 13.2**

## 13.2 Graph-Reduktion

Es ist nicht wahr, daß *normal order reduction* – in der Form wie eben besprochen – immer weniger Reduktionsschritte benötigt als *applicative order reduction*.

**Beispiel 13.3**

Für den Ausdruck `sqr (4 + 2)` benötigt *normal order reduction* vier Reduktionen:

$$\begin{array}{lll} \texttt{sqr (4 + 2)} & \rightarrow & \texttt{(4 + 2)} \times \texttt{(4 + 2)} \quad (\texttt{sqr}) \\ & \rightarrow & \texttt{6} \times \texttt{(4 + 2)} \quad (\boxplus) \\ & \rightarrow & \texttt{6} \times \texttt{6} \quad (\boxplus) \\ & \rightarrow & \texttt{36} \quad (\boxtimes) \end{array}$$

*Applicative order reduction* kommt mit drei Reduktionen aus:

$$\begin{array}{lll} \texttt{sqr (4 + 2)} & \rightarrow & \texttt{sqr 6} \quad (\boxplus) \\ & \rightarrow & \texttt{6} \times \texttt{6} \quad (\texttt{sqr}) \\ & \rightarrow & \texttt{36} \quad (\boxtimes) \end{array}$$

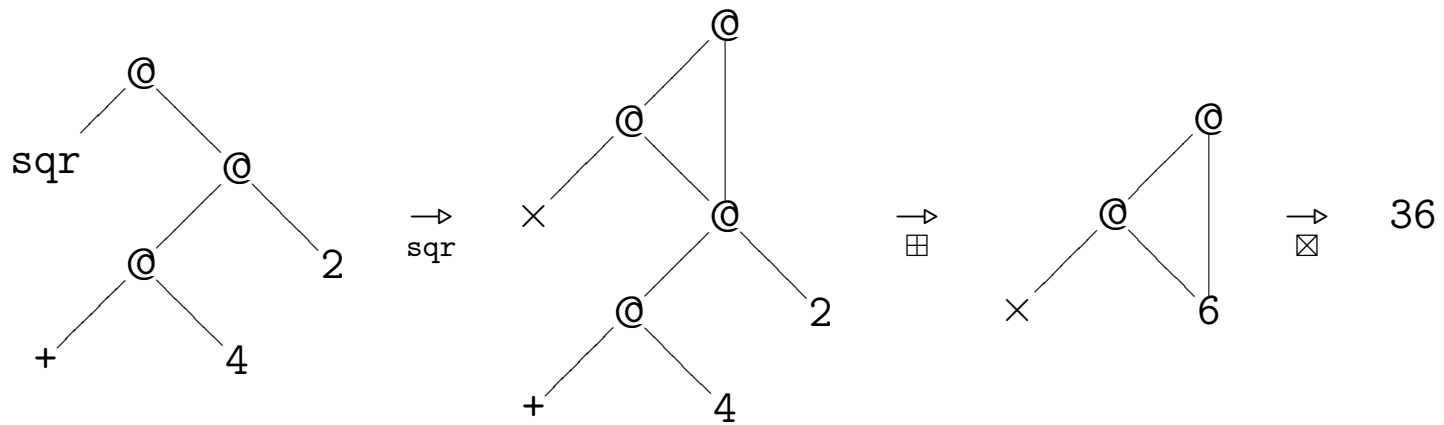
Das Problem besteht hier in der **Duplizierung eines Teilausdrucks**, so daß dieser Teilausdruck zweimal reduziert werden muß. Jede Definition, in der ein Parameter auf der rechten Seite mehr als einmal auftritt, leidet unter diesem Problem (bspw. `sqr` $x = x \times x$).

FPL-Implementationen arbeiten daher intern mit **Term-Graphen** und nicht den bisher betrachteten Bäumen. In den Graphen können gemeinsame Teilausdrücke einfach repräsentiert werden (*sharing*).

Die Auswertung wird dann durch **Graph-Reduktion** erzielt.

**Beispiel 13.4**

Unter Einsatz von Graph-Reduktion benötigt auch die *normal order reduction* von `sqr (4 + 2)` lediglich drei Reduktionsschritte:




Genau auf diese Weise sind auch `let...in` und `where` effizient realisierbar.

Mittels Graph-Reduktion benötigt *normal order reduction* nie mehr Reduktionsschritte als *applicative order reduction*.

**Beachte:** Die **Korrektheit von Graph-Reduktion** beruht entscheidend auf der **Seiteneffektfreiheit** unserer FPL.

## 13.3 Terminierung

Es gibt Ausdrücke, für die *applicative order reduction* keine Normalform finden kann, weil die Reduktionsfolge nicht terminiert.

**Beispiel 13.5**
Zusätzlich zu `fst` seien noch `answer` und `loop` definiert:

```
answer  =  fst (42, loop)
loop    =  tl loop
```

Die Auswertung von `answer` mittels *applicative order reduction* terminiert nicht:

```
answer  →▷  fst (42, loop)                (answer)
        →▷  fst (42, tl loop)             (loop)
        →▷  fst (42, tl (tl loop))        (loop)
        →▷  fst (42, tl (tl (tl loop)))   (loop)
        →▷  ...
```

*Normal order reduction* reduziert dagegen wie folgt:

```
answer  →▷  fst (42, loop)                (answer)
        →▷  42                            (fst)
```

Terminieren sowohl *applicative order* als auch *normal order reduction*, so liefern beide dasselbe Ergebnis.

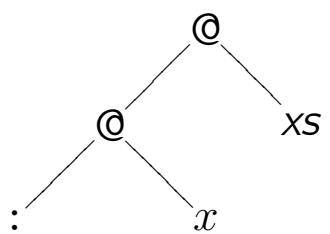
**Wenn ein Ausdruck überhaupt eine Normalform besitzt, dann kann sie durch normal order reduction gefunden werden.** Ein Teilausdruck wird nur dann reduziert, wenn dies zur Berechnung des Ergebnisses wirklich notwendig ist.

## 13.4 Weak Head Normal Form (WHNF)

Zentrales Ziel der *lazy evaluation* ist es, jegliche unnötige Reduktionen zu vermeiden. *Lazy* FPLs reduzieren Ausdrücke daher nicht auf ihre tatsächliche Normalform, sondern lediglich auf die sog. **weak head normal form** (WHNF). WHNF definiert ein Stop-Kriterium für die „klassische“ *normal order reduction*.

**Beispiel 13.6**

Der Ausdruck $x$ : *xs* ist schon dann in WHNF, wenn $x$ und *xs* jeweils noch *nicht reduziert* wurden. Gleichwertig: Der Baum



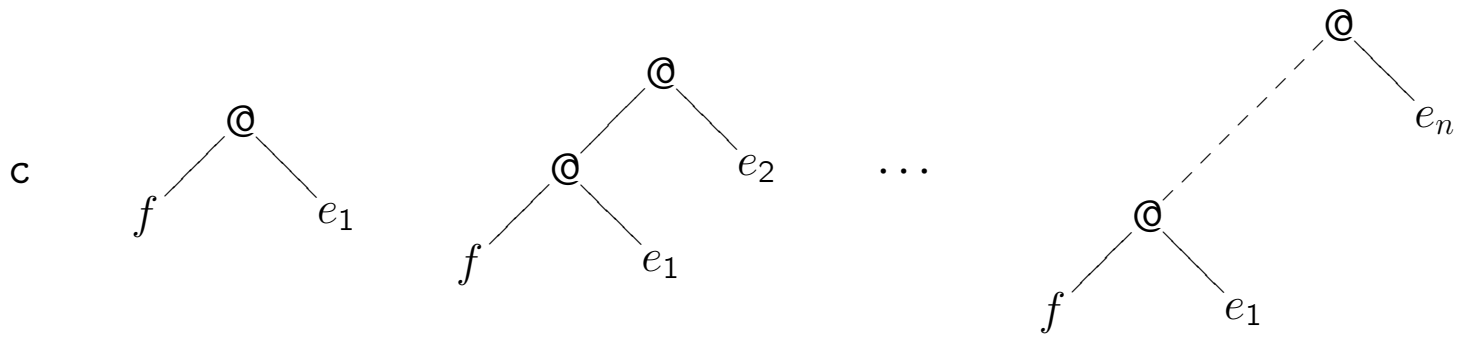

ist in WHNF, weil kein *top-level* Redex mehr existiert (obwohl noch innere Redexe existieren).

**Definition 13.1**
Ein Ausdruck ist genau dann in **weak head normal form** (WHNF), wenn er die Form

$$c \quad \text{oder} \quad f\ e_1\ e_2\ \ldots\ e_n \qquad (n \geq 0)$$

besitzt. Dabei ist $c$ ein atomarer Wert ($\texttt{Integer}, \texttt{Bool}, \ldots$) und $f$ eine Funktion (oder Konstruktor eines algebraischen Datentyps) für die $f\ e_1\ e_2\ \ldots\ e_m \quad (m \leq n)$ *kein* Redex ist.

Damit besagt Definition 13.1, daß die Bäume




genau dann in WHNF sind, falls sie keinen *top-level* Redex besitzen ($f$ ist nicht reduzierbar).

Die **Reduktion stoppt, sobald kein top-level Redex mehr existiert**. Dadurch wird die evtl. unnötige Auswertung innerer Redexe verhindert.

Ein weiterer Vorteil: Da ein *top-level* Redex keine freien Variablen beinhaltet (und damit auch seine Argumente nicht), benötigt die Reduktionsmaschinerie keine Verwaltung für freie Variablen und entgeht damit Problemen wie *name capture* (siehe Abschnitt 3.2.1).

**Beispiel 13.7**
Die folgenden Ausdrücke sind in WHNF:

| | |
|---|---|
| `3` | atomare Konstante |
| `(sqr 2, sqr 4)` | Tupelkonstruktor, nicht weiter reduzierbar |
| `3 : map f xs` | *cons*, nicht weiter reduzierbar |
| `(+) (4-3)` | partiell angewandte Funktion `(+)` |
| `(\x -> 5+1)` | nicht angewandte $\lambda$-Abstraktion |

Der letzte Ausdruck ist in WHNF, aber nicht in Normalform (dazu müßte der innere Redex `5+1` noch reduziert werden).

### 13.4.1 Beispiele für Lazy WHNF Reduction

*Lazy evaluation* erlaubt einen **daten-orientierten** Programmierstil (auch „*listful style*“), der in einer Sprache mit *applicative order reduction* höchst ineffizient wäre:

- Daten-orientierte Programme konstruieren (komplexe, unendliche) Datenstrukturen und
- manipulieren diese Datenstrukturen Schritt für Schritt durch Anwendung einer **Komposition relativ simpler Funktionen**.

Dank *lazy evaluation* werden die Datenstrukturen nie komplett erzeugt.

**Beispiel 13.8**

Berechne die Summe der Quadrate der Zahlen von $1 \ldots n$. „Klassische“ rekursive Lösung:

```
sumsqr  :: Integer -> Integer
sumsqr 1 = 1
sumsqr n = n^2 + sumsqr (n-1)
```

Die daten-orientierte Lösung ① konstruiert die Liste `[1..`$n$`]`, ② berechnet die Quadrate der Listenelemente `[1,4,`$\ldots n^2$`]` und ③ summiert die Elemente dieser Liste:

$$\texttt{sumsqr' n} = \texttt{(}\underbrace{\texttt{sum}}_{③}\texttt{ . }\underbrace{\texttt{map (\^{}2)}}_{②}\texttt{)}\ \underbrace{\texttt{[1..n]}}_{①}$$

Die Schritte ② und ③ sind hier absichtlich mittels Funktionskomposition `(.)` kombiniert worden, um den modularen Charakter des *listful style* zu unterstreichen.

`sumsqr'` scheint mit dem Heap verschwenderisch umzugehen, denn die Schritte ① und ② bauen potentiell große Listen als Zwischenergebnisse auf.

Tatsächlich erzeugt *lazy evaluation* + WHNF während der Reduktion keine einzige der beiden Listen.

Reduktion von `sumsqr' 3`:

```
sumsqr' 3  ⇨  (sum . map (^2)) [1..3]                          (sumsqr')
           ⇨  sum (map (^2) [1..3]))                           ((.))
           ⇨  sum (map (^2) (1:[2..3]))                        (..)
           ⇨  sum ((^2) 1 : map (^2) [2..3])                   (map)
           ⇨  (^2) 1 + sum (map (^2) [2..3])                   (sum)
           ⇨  (^2) 1 + sum (map (^2) 2:[3..3])                 (..)
           ⇨  (^2) 1 + sum ((^2) 2 : map (^2) [3..3])          (map)
           ⇨  (^2) 1 + (^2) 2 + sum (map (^2) [3..3])          (sum)
           ⇨  (^2) 1 + (^2) 2 + sum (map (^2) 3:[])            (..)
           ⇨  (^2) 1 + (^2) 2 + sum ((^2) 3 : map (^2) [])     (map)
           ⇨  (^2) 1 + (^2) 2 + (^2) 3 + sum (map (^2) [])     (sum)
           ⇨  (^2) 1 + (^2) 2 + (^2) 3 + sum []                (map)
           ⇨  (^2) 1 + (^2) 2 + (^2) 3 + 0                     (sum)
           ⇨  14                                               (⊞, ^)
```

Sobald das erste Listenelement greifbar ist, wird es sofort quadriert und wird im nächsten Schritt Teil der Gesamtsumme.

**Beispiel 13.8**

**Beachte:** Hier steht `[1..3]` nicht für die konstante Liste `[1,2,3]`, sondern für den **Listengenerator**, der die Liste der Elemente 1 bis 3 **bei Bedarf** erzeugen kann.

Der *listful style of programming* ermöglicht also die effiziente Verkettung eines **Generators** mit einer Sequenz (Komposition) von **Transformern**. Da *lazy evaluation* den Generator nur bei Bedarf nach einem nächsten Listenelement aufruft (*data on demand*), kann der Generator prinzipiell auch eine unendliche Datenstruktur erzeugen. Darauf kommt Abschnitt 13.5 gleich zurück.

Ausdrucksauswertung mittels *lazy evaluation* zeigt oft nicht sofort offensichtliche Effekte. Das folgende Beispiel versucht dies zu illustrieren.

**Beispiel 13.9**

Wie bestimmt man das Minimum einer Liste von Zahlen? Unsere Lösung: ① Sortiere(!) die Liste (mittels *insertion sort*, `isort` siehe unten), ② dann enthält der Kopf der Liste das Minimum:

```
min = head . isort (<)
```

*Insertion sort* sortiert eine Liste, indem das jeweilige Kopfelement der unsortieren Liste an seine korrekte Position (bzgl. `lt`) mittels `ins` in die Ergebnisliste einfügt wird:

```
isort lt []     = []
isort lt (x:xs) = ins x (isort lt xs)
                  where
                       ins x []                    = [x]
                       ins x (x':xs) | x `lt` x'   = x:x':xs
                                     | otherwise   = x':ins x xs
```

Reduktion mittels *lazy evaluation* zeigt, daß die Argumentliste nie vollständig sortiert wird: `min` hat lineare Komplexität trotz Ausnutzung von `isort` ($O(n^2)$):

```
min [8,6,1,7,5]  →▷  (head . isort (<)) [8,6,1,7,5]                    (min)
                 →▷  head (isort (<) [8,6,1,7,5])                      ((.))
                 →▷  head (ins 8 (ins 6 (ins 1 (ins 7 (ins 5 [])))))   (isort⋆)
                 →▷  head (ins 8 (ins 6 (ins 1 (ins 7 [5]))))          (ins.1)
                 →▷  head (ins 8 (ins 6 (ins 1 (5 : ins 7 []))))       (ins.3)
                 →▷  head (ins 8 (ins 6 (1 : (5 : ins 7 []))))         (ins.2)
                 →▷  head (ins 8 (1 : (ins 6 : (5 : ins 7 []))))       (ins.3)
                 →▷  head (1 : ins 8 (ins 6 : (5 : ins 7 [])))         (ins.3)
                 →▷  1                                                 (head)
```

In allen Fällen benötigt `ins` nur jeweils einen Reduktionsschritt, um die Liste, in die eingefügt wird, bereits in WHNF zu bringen.
Der gesamte Listenrest wird jedoch nie sortiert und am Ende ohnehin fort geworfen, da wir mit `head` lediglich auf den Listenkopf zugreifen.

**Beispiel 13.9**

## 13.5 Unendliche Listen

Die Eigenschaft, Ausdrücke jeweils nur in ihre WHNF zu überführen, verleiht *lazy* FPLs die Fähigkeit auch auf **unendlichen Datenobjekten**, vor allem Listen, zu operieren. Im Zusammenspiel mit dem *listful style of programming* ergibt sich ein sehr eleganter und doch effizienter Programmierstil.

Es ist wichtig, sich klarzumachen, daß unendlichen Listen *nicht* die Eigenschaften unendlicher mathematischer Objekte, etwa Mengen, besitzen.

**Beispiel 13.10**
Während die *set comprehension*

$$\{x^2 \mid x \in \{1, 2, 3, \ldots\}, x^2 < 10\}$$

das endliche Objekt $\{1, 4, 9\}$ bezeichnet, liefert die „äquivalente“ *list comprehension*

```
[ x^2 | x <- [1..], x^2 < 10 ]
```

bei Auswertung im Interpreter `[1,4,9` (korrekt: `1:4:9:`$\perp$). Korrekt wäre hier etwa

```
(takeWhile (<10) . map (^2)) [1..]
```

**Frage:** Sei `cubes = [ x^3 | x <- [1..]]`. Was ist der Wert von `(elem 64 cubes)`? Wie lautet der Wert von `(elem 65 cubes)`?

### 13.5.1 `iterate`

Die Funktion `iterate` aus der *standard prelude* ist ein Generator unendlicher Listen (siehe Beispiel 2.7 zur iterativen Wurzelbestimmung):

$$\texttt{iterate}\ f\ x\ =\ x\ \texttt{:}\ \texttt{iterate}\ f\ (f\ x)$$

Informell berechnet `iterate` $f\ x$ also $[x, f\ x, f^2\ x, f^3\ x, \ldots]$

**Frage:** Typ von `iterate`?

**Beispiel 13.11**

```
                iterate (+1) 1   =  [1, 2, 3, 4, 5,...]
                iterate (*2) 1   =  [1, 2, 4, 8, 16,...]
          iterate (`div` 10) 2718  =  [2718, 271, 27, 2, 0, 0, 0,...]
```

oder auch

$$[m..n] \quad \equiv \quad \texttt{takeWhile (<=}\ n\texttt{) (iterate (+1)}\ m\texttt{)}$$

Für den *listful style* ist `iterate` ein idealer Generator.

**Beispiel 13.12**

Definiere die Funktion `digits`, die die Liste der Ziffern einer ganzen Zahl bestimmt:

```
digits = reverse . map (`mod` 10) . takeWhile (/= 0) . iterate (`div` 10)
```

Diese Sequenz aus Generator und Transformern berechnet dann `digits 2718` wie folgt:

```
                2718
                 ↓                  iterate (`div` 10)
[2718, 271, 27, 2, 0, 0, 0,...]
                 ↓                  takeWhile (/= 0)
      [2718, 271, 27, 2]
                 ↓                  map (`mod` 10)
         [8, 1, 7, 2]
                 ↓                  reverse
         [2, 7, 1, 8]
```

**Frage:** Was berechnet die folgende ganz analog definierte Funktion `foo`?

```
foo n = map (take n) . takeWhile (/= []) . iterate (drop n)
```

Ein letztes Beispiel für einen Generator unendlicher Listen sei `fib`.

**Beispiel 13.13**

`fib :: [Integer]` generiert die unendliche Liste der Fibonacci-Zahlen. Die Liste `fib` ist dabei durch sich selbst rekursiv definiert und „beißt sich selbst in den Schwanz“:

```
fib = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

Zur Erinnerung:

```
zip             :: [α] -> [β] -> [(α,β)]
zip (x:xs) (y:ys) = (x, y) : zip xs ys
zip _      _      = []
```

# 14 Monaden, Ein-/Ausgabe, Zustände

Funktionale Programme sind **referentiell transparent**, dies war **die zentrale** Beobachtung bzw. Eigenschaft, die viele Transformationen überhaupt erst möglich gemacht hat.

Bereits in der Einführung haben wir erwähnt, dass *Seiteneffekte* die referentielle Transparenz zerstören und hierfür Beispiele, wie etwa globale Variablen (Programmzustände) und Ein-/Ausgabeoperationen genannt.

**Auch funktionale Programme müssen selbstverständlich in der Lage sein, Ein- und Ausgaben durchzuführen.** Ebenso werden wir sehen, dass *zustandsabhängiges Verhalten* – z.B. zur Behandlung von Programm-Ausnahmen (Exceptions) – einfacher zu beschreiben ist, wenn geeignete Sprachmittel zur Verfügung gestellt werden, die wir bislang noch nicht kennen gelernt haben.

In diesem Abschnitt betrachten wir einen neuen funktionalen Programmierstil:

## Monadic Programming

Dieser Programmierstil erlaubt es, z.B. Ein-/Ausgabeoperationen in funktionalen Programmen so zu verwenden, wie wir es aus imperativen Sprachen gewohnt sind, oder Programme zu formulieren, die „Zustände“ transformieren, ohne dabei Seiteneffekte einführen zu müssen.

Monaden sind mathematische Strukturen, deren Eigenschaften wir später auflisten. Zunächst verschaffen wir uns einen informalen Überblick. . .

# 14.1 Ein- und Ausgabe: Monadic Interaction

In Haskell repräsentiert der Typ `IO ()` ein *Kommando*. Ein Ausdruck vom Typ `IO ()` bezeichnet eine *Aktion*; wenn der Ausdruck ausgwertet wird, so wird die Aktion *ausgeführt*. Der Typ `IO ()` ist ein abstrakter Typ im Sinne von Abschnitt 9.

**Beispiel 14.1**
Eine Grundfunktion druckt einen Buchstaben: `putChar`, eine andere tut nichts: `done`

```
putChar :: Char -> IO ()

? putChar '!'
!

? done
?
```

Als nächstes müssen Kommandos kombiniert werden, dies leistet bspw. der Kombinator >>. `p >> q` führt zunächst das Kommando `p`, danach das Kommando `q` aus.

```
(>>) :: IO () -> IO () -> IO ()
```

Mit Hilfe von >> können einfache Ausgabefunktionen realisiert werden.

**Beispiel 14.2**

`write` tut das gleiche wie die vordefinierte Haskell Funktion `putStr`:

```
write :: String -> IO ()
write []     = done
write (c:cs) = putChar c >> write cs

-- oder mittels foldr:
write = foldr (>>) done . map putChar

-- writeln ist ähnlich, aber schließt mit Newline ab:
writeln :: String -> IO ()
writeln cs = write cs >> putChar '\n'
```

Zur *Eingabe* brauchen wir einen etwas allgemeineren Typ `IO` $\alpha$, der ein Ergebnis vom Typ $\alpha$ liefert. Eine primitive Eingabefunktion zum Lesen eines Buchstabens wäre damit:

```
getChar :: IO Char
```

Werden einige Tasten auf der Tastatur gedrückt, so liest `getChar` den ersten Buchstaben (und gibt ihn, laut Haskell-Konvention, als Echo wieder aus).

Auch die Funktion `done` kann entsprechend erweitert werden zu `return`. Dieses Kommando „tut nichts“ und gibt einen Wert zurück, z.B. `return 42` gibt den Wert 42 zurück, ohne Input zu konsumieren.

```
return :: a -> IO a

-- im Spezialfall done:  [ "()" ist das einzige Exemplar des nullary Typs ()]
done :: IO ()
done =  return ()
```

Auch der Kommando-Kombinator >> wird etwas allgemeiner erweitert: `p >> q` führt zuerst `p` aus, ignoriert den Output, und führt dann `q` aus.

**Beispiel 14.3**

Wenn die Eingabe des nachfolgenden Kommandos (durch Newline) beendet wird und anschließend ein „x“ eingegeben wird, so wird das „x“ wieder ausgegeben und die Interaktion durch `done` beendet:

```
? getChar >> done
x
```

Offensichtlich ist die Kommando-Folge `p >> q` nur dann sinnvoll, wenn die Ausgabe des Kommandos `p` uninteressant ist; insbesondere kann `q` nicht abhängig von dieser Ausgabe von `p` sein. Im Allgemeinen brauchen wir folglich eine mächtigere Kombination von Kommandos:

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

Die Kombination `p >>= q` ist ein Kommando, das bei der Ausführung zunächst `p` ausführt, welches einen Wert $x$ vom Typ $\alpha$ zurückliefert. Als nächstes wird `q` $\alpha$ ausgeführt, welches schließlich einen Resultatwert $y$ vom Typ $\beta$ liefert.

**Beachte:** die Typen $\alpha$ und $\beta$ können i.a. verschieden sein.

**Beispiel 14.4**

Eingabe von „x“ nach Abschluss des Kommandos führt `getChar` aus, dieses erzeugt ein Echo des Eingabezeichens. Anschließend wird `putChar` mit dem gelesenen Buchstaben als Argument aufgerufen, was zur zweiten Ausgabe eines „x“ führt.

```
? getChar >>= putChar
xx
```

Mittels >>= können einfache Eingabefunktionen realisiert werden.

**Beispiel 14.5**

`readn` liest eine feste Anzahl Zeichen:

```
readn      :: Int -> IO String

readn 0     = return []
readn (n+1) = getChar >>= q
              where q c = readn n >>= r
                          where r cs = return (c:cs)
```

`readln` liest bis zum ersten Newline (ohne diesen):

```
readln :: IO String
readln  = getChar >>= q
          where q c = if c == '\n'
                      then return []
                      else readln >>= r
                           where r cs = return (c:cs)
```

**N.B.** die Formulierungen mittels geschachteltem `where` sind unübersichtlich! Es folgt sogleich eine kompaktere Notation...

# (Zwischen-) Zusammenfassung

Die IO-Monade `IO` $\alpha$ ist ein abstrakter Typ mit – mindestens – den folgenden Operationen:

```
return   :: α -> IO α
(>>=)    ::  IO α -> ( α ->  IO β ) ->  IO β

putChar  :: Char -> IO ()
getChar  :: IO Char
```

Dabei sind

- die ersten beiden Kombinatoren charakteristisch für Monaden,
- die anderen beiden Funktionen sind spezifisch für die IO-Monade.

Allgemein ist also eine Typklasse $M$ eine Monade, wenn die folgenden Operationen definiert sind:

```
return :: α  →  M α
(>>=)  :: M α  →  (α  →  M β)  →  M β
```

. . . und wenn zusätzlich eine Reihe von Eigenschaften dieser Operationen gelten (s. unten).

## 14.2 `do`-Notation

In Haskell werden Monaden als Instanzen der Typklasse `Monad` deklariert:

```
class Monad m where
 return :: a -> m a
 (>>=)  :: m a -> ( a -> m b ) -> m b
```

Für Instanzen dieser Typklasse stellt Haskell eine kompakte Notation für Kombinationen mittels >>= und geschachtelten `where`-Klauseln zur Verfügung. Dazu wird ein Schlüsselwort `do` und der sog. *Bind*-Operator <- eingeführt.

**Beispiel 14.6**
Die Funktion `readn` von oben kann damit kompakter so definiert werden:

```
readn      :: Int -> IO String
readn 0     = return []
readn (n+1) = do c  <- getChar
                 cs <- readn n
                 return (c:cs)
```

Dies kann direkt gelesen werden als: „führe das Kommando `getChar` aus und binde das Resultat an `c`; dann führe `readn n` aus und binde das Resultat an `cs`; zuletzt gib `(c:cs)` zurück.“

**Beispiel 14.7**

Ähnlich kann auch die Funktion `readln` kompakter notiert werden:

```
readln :: IO String
readln  = do c <- getChar
             if c == '\n'
                then return []
                else do cs <- readln
                        return (c:cs)
```

**Beachte** die Einrückung gemäß Layout-Regeln!

Alternativ können natürlich auch hier explizite Klammerung und Semikolon verwendet werden, bspw.

```
readn (n+1) = do { c <- getChar; cs <- readn n; return (c:cs) }
```

# Allgemeine Form `do` expressions

Für eine allgemeine Monade $M$ hat ein `do`-Ausdruck die Form

$$\texttt{do}\ \{C; r\}$$

wobei

- $C$ eine Liste von ein oder mehr Kommandos, getrennt jeweils durch ein Semikolon, und
- $r$ ein Ausdruck vom Typ $M\beta$ ist. Dies ist auch der Typ des ganzen `do`-Ausdrucks.
- Jedes Kommando in $C$ hat die Form $x \leftarrow p$, wobei
  - $x$ eine Variable oder ein Tupel von Variablen ist,
  - wenn $p$ ein Ausdruck vom Typ $M\alpha$ ist, dann ist der Typ von $x$ $\alpha$.
    Im Spezialfall $\alpha = ()$ kann das Kommando „$() \leftarrow p$“ abgekürzt werden zu „$p$“.

`do`-Ausdrücke können mithilfe der folgenden Regeln leicht in >>=-Kombinatoren und `where`-Klauseln übersetzt werden:

① $\texttt{do}\ \{r\} = r$

② $\texttt{do}\ \{x \leftarrow p; C; r\} = p$ >>= $q$ `where` $q\ x = \texttt{do}\ \{C; r\}$

③ $\texttt{do}\ \{p; C; r\} = p$ >>= $q$ `where` $q\ \_ = \texttt{do}\ \{C; r\}$

**Übung:** übersetze `readn` und `readln` aus der kompakten in die ursprüngliche Schreibweise.

## 14.3 Beispiele

### 14.3.1 Hangman

Ein kleines Spiel „Hangman“ (á la Mastermind) – ein Spieler muss ein Wort erraten, Eingaben sind Wörter, Antwort ist jeweils das gesuchte Wort mit allen Buchstaben, die bislang nicht erraten wurden durch „–“ ersetzt – könnte bspw. folgenden Verlauf nehmen:

```
? hangman
Think of a word:
----
Now try to guess it!
guess: last
--al
guess: dial
--al
guess: opal
-oal
guess: foal
-oal
guess: goal
YOU GOT IT!
```

Das Spiel könnte etwa wie folgt implemenitert werden (das Zeichnen des „Hangman“ überlassen wir hier dem menschlichen Spielleiter. . . )

```
hangman :: IO ()
hangman  = do writeln "Think of a word:"
              word <- silentReadln
              writeln "Now try to guess it!"
              guess word
```

Die Funktion `silentReadln` verhält sich dabei wie `readln`, d.h. sie liest eine Zeile bis zum Newline, führt aber kein Echo auf dem Bildschirm durch, vielmehr druckt sie ein „–“ je eingegebenem Zeichen. `slientReadln` kann mittels der primitiven Haskell-Funktion `getCh` realisiert werden (`getChar = do { c <- getCh; putChar c; return c }`):

```
silentReadln :: IO String
silentReadln  = do c <- getCh
                   if c == '\n'
                      then do putChar c
                              return []
                      else do putChar '-'
                              cs <- silentReadln
                              return (c:cs)
```

Das Kommando `guess word` liest Eingaben, solange bis das eingegebene Wort `word` ist:

```
guess      :: String -> IO ()
guess word  = do write "guess: "
                 cs <- readln
                 if cs == word
                    then writeln "YOU GOT IT!"
                    else do writeln (compare word cs)
                            guess word

compare          :: String -> String -> String
compare word cs   = map check word
                    where check w = if member cs w then w else '-'
```

## 14.3.2 Datei-Bearbeitung

Wir betrachten eine einfache Filter-Aufgabe: zu lesen ist eine Textdatei, aus der alle nicht-ASCII Zeichen entfernt werden sollen, die Ausgabe erfolgt in eine zweite Datei.

Hierzu verwenden wir zwei Haskell-Primitive zum Dateizugriff:

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

Der Typ `FilePath` ist dabei ein Synonym für `String`, d.h. Dateinamen werden als String übergeben und systemabhängig interpretiert.

```
filterFile  :: IO ()
filterFile  = do write "Enter input file name: "
                 infile <- getLine
                 write "Enter output file name: "
                 outfile <- getLine
                 xs <- readFile infile
                 writeFile outfile (filter isAscii xs)     -- isAscii def'd in Haskell library
                 writeln "Filtering successful"
```

## 14.4 Vergleich mit imperativen Programmen

Die betrachteten monadischen Beispiele „sehen (fast) aus“ wie imperative Programme. **Aber:** es gibt deutliche Unterschiede!

Zum Beispiel:

① `readFile` liest eine Datei *lazy, on demand*.
Also werden die beiden Kommandos

```
xs <- readFile infile
writeFile outfile (filter isAscii xs)
```

den gesamten Inhalt der Eingabedatei **nicht** in den Speicher einlesen, bevor gefiltert und ausgegeben wird.

② monadische Programmierung ist ein *optionales Extra* in Haskell, d.h. für alle Programmteile, die unabhängig von monadischer Interaktion sind, gelten alle bisher gemachten Aussagen über Programmanalyse und -transformation.
Monadische Interaktion dient der **lokalen Eingrenzung** der (Seiten-) Effekte von Ein-/Ausgaben.

## 14.5 Andere Monaden

Neben Ein-/Ausgaben gibt es eine Fülle weiterer Anwendungsgebiete für Monaden, von denen uns einige – ohne dass wir das explizit erwähnt hätten – schon begegnet sind.

Allgemein können wir Monaden als eine Art „Container“ verstehen, mit deren Hilfe wir das Arbeiten auf/mit den Containern beschreiben können, ohne auf den Inhalt oder auch nur die genaue Art des Containers eingehen zu müssen. Also ein weiterer, mächtiger Abstraktionsmechanismus.

### 14.5.1 Maybe ist auch eine Monade

Wir haben bereits den Typkonstruktur `Maybe` gesehen, der es erlaubt, „fehlende Werte“ zu repräsentieren:

```
data Maybe a = Nothing | Just a
```

In der o.a. Sprechweise wäre `Maybe a` also ein Typ für einen Container, der Exemplare des Typs `a` enthalten kann oder auch nichts.

**Was haben wir davon, dass `Maybe` eine Monade ist?**

Die monadischen Operatoren erlauben es uns, Funktionen über Typen mit „fehlenden Werten“ kompakt zu notieren. Wir betrachten dazu das folgende

**Beispiel 14.8**

Wir schreiben ein Programm zur Analyse von Klon-Experimenten bei Schafen. Dazu benötigen wir sicher für jedes Schaf die genauen genetischen Informationen, insbesondere über die Väter und Mütter. Nun haben aber geklonte Schafe nicht immer Mutter und/oder Vater...

```
type Sheep = ...

father :: Sheep -> Maybe Sheep
father = ...

mother :: Sheep -> Maybe Sheep
mother = ...
```

Entsprechend wären Funktionen zur Berechnung der Großeltern etwas umständlicher zu formulieren:

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = case (mother s) of
                          Nothing -> Nothing
                          Just m  -> father m
```

...

**Beispiel 14.8 (Cont'd)**

Oder gar bei den Urgroßeltern:

```
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = case (mother s) of
                                 Nothing -> Nothing
                                 Just m  -> case (father m) of
                                              Nothing -> Nothing
                                              Just gf -> father gf
```

Dies ist sicherlich kein Beispiel für lesbaren Code!

Was wir brauchen ist ein Konstrukt, das es uns erlaubt, *einmal* zu spezifizieren, wie mit fehlenden Werten umzugehen ist (hier: sobald irgendwo `Nothing` zurückgegeben wird, soll dies sich propagieren). Ein entsprechender monadischer Kombinator leistet dies.

```
-- comb is a combinator for sequencing operations that return Maybe
comb :: Maybe a -> (a -> Maybe b) -> Maybe b
comb Nothing  _ = Nothing
comb (Just x) f = f x
```

Mit dessen Hilfe können nun die Berechnungen übersichtlich und kompakt notiert werden:

```
-- now we can use ‘comb‘ to build complicated sequences
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = (Just s) ‘comb‘ mother ‘comb‘ father ‘comb‘ father
```

Das beste an dem Beispiel ist die Tatsache, dass der betrachtete Kombinator *überhaupt nicht* spezifisch für das Klonschaf-Beispiel ist. Er lässt sich auf alle Berechnungsfolgen anwenden, die „unterwegs“ u.U. keine Resultate (also `Nothing`) liefern können.

`Maybe` zusammen mit dem `Just`-Konstruktor und dem Kombinator stellen eine allgemeine Monade für Berechnungen mit „fehlenden Werten“ dar.

- `Just` spielt dabei die Rolle von `return`,
- der Kombinator die Rolle von >>=.

Daher könnten wir `Maybe` auch explizit als Instanz der Monad-Klasse deklarieren (das übernimmt bereits der Standard Prelude für uns):

```
instance Monad Maybe where
    Nothing  >>= f = Nothing
    (Just x) >>= f = f x
    return         = Just
```

Folglich können wir auch die Standard-Notation verwenden, etwa

```
maternalGrandfather :: Sheep -> Maybe Sheep
maternalGrandfather s = (return s) >>= mother >>= father

fathersMaternalGrandmother :: Sheep -> Maybe Sheep
fathersMaternalGrandmother s = (return s) >>= father >>= mother >>= mother
```

oder

```
-- we can also use do-notation to build complicated sequences
mothersPaternalGrandfather :: Sheep -> Maybe Sheep
mothersPaternalGrandfather s = do m  <- mother s
                                  gf <- father m
                                  father gf
```

**N.B.** es gibt eine ganze Reihe weiterer solcher „Container“, beispielsweise die folgenden zwei

- `Error`, eine Monade für Berechnungen, die Werte oder Fehlermeldungen weiterreichen können,
- Listen (!) sind auch Monaden (für Berechnungen, die jeweils 0, 1 oder mehr Werte liefern können), mit
  - `return x = [x]` und
  - `l >>= f = concatMap f l`.

## 14.5.2 Zustände

Schon bei der Diskussion von Referentieller Transparenz haben wir darauf hingewiesen, dass funktionale Sprachen keinen (impliziten) *Zustand* (etwa über globale Variablen) eines Programms kennen. Vielmehr müssen alle Abhängigkeiten über Aufrufparameter bzw. Rückgabewerte von Funktionen *explizit gemacht* werden.

**Beispiel 14.9**

Wir betrachten einen einfachen Stack of Int als Beispiel. Wir definieren zunächst folgende Operationen mit den zugehörigen Signaturen:

| | |
|---|---|
| `mt: () -> Stack` | – erzeugt einen leeren Stack |
| `top: Stack -> Int` | – liest das oberste Element ohne den Stack zu verändern |
| `push: Int -> Stack -> Stack` | – legt ein neues Element ab |
| `pop: Stack -> (Stack, Int)` | – liest und entfernt das oberste Element |

Eine einfache Folge von Stack-Operationen wie etwa

| | | |
|---|---|---|
| $mt$ | $\rightarrow$ | Zwischenzustand $s_0$ |
| $push(4)$ | $\rightarrow$ | $s_1$ |
| $push(5)$ | $\rightarrow$ | $s_2$ |
| $x \leftarrow pop$ | $\rightarrow$ | $s_3$ und Bindung an $x$ |
| $push(3)$ | $\rightarrow$ | $s_4$ |
| $push(x)$ | $\rightarrow$ | $s_5$ |

...

**Beispiel 14.9 (Cont'd)**

. . . müsste dann etwa wie folgt formuliert werden:

```
let s0 = mt ()
in  let s1 = push 4 s0
    in  let s2 = push 5 s1
        in  let (s3, x) = pop s2
            in  let s4 = push 3 s3
                in  let s5 = push x s4
                    in  s5
```

Wiederum keine sehr elegante Formulierung, da die Zustände explizit „durchgereicht" werden müssen.

Wünschenswert wäre dagegen wieder eine Formulierung mittels `do`, wie etwa:

```
do { mt
   ; push 4
   ; push 5
   ; x <- pop
   ; push 3
   ; push x
  }
```

hier müsste jeweils aus einem Kommando der resultierende Zwischenzustand implizit dem nächsten Kommando als Anfangszustand weitergeleitet werden.

Tatsächlich ist eine solche Formulierung möglich, dazu wird eine sog. State-Transformer Monade `ST` $\alpha$ (hier mit $\alpha$=Int) verwendet.

Dazu müssen jedoch zunächst die Signaturen der Stack-Operationen vereinheitlicht werden, so dass sie alle eine ähnliche Form haben und mit Hilfe von

$$\texttt{data ST } \alpha \texttt{ = ST ( } \underbrace{\texttt{Stack}}_{①} \texttt{ -> ( } \underbrace{\texttt{Stack}}_{②}\texttt{, } \underbrace{\alpha}_{③} \texttt{ ) )}$$

definiert werden können. Dabei ist jeweils:

① der Anfangszustand,

② der Folgezustand und

③ der (etwaige) Rückgabewert einer jeden Stack-Operation.

Es ergibt sich

| Bislang | Jetzt neu |
|---|---|
| `mt: () -> Stack` | `mt: Stack -> ( Stack, () )` |
| `top: Stack -> Int` | `top: Stack -> ( Stack, Int )` |
| `push: Int -> Stack -> Stack` | `push: Int -> Stack -> ( Stack, () )` |
| `pop: Stack -> (Stack, Int)` | `pop: Stack -> ( Stack, Int )` |

Mit Hilfe der State-Transformer Monade `ST` $\alpha$ können wir nun diese neuen Stack-Operationen herleiten:

```
data ST α   =   ST ( Stack → ( Stack, α ) )

top         ::  ST Int
top         =   ST ( s ↦ (s, head s) )

pop         ::  ST Int
pop         =   ST ( s ↦ (tail s, head s) )

push        ::  Int → ST ()
push  x     =   ST ( s ↦ (x : s, ()) )

mt          ::  ST ()
mt          =   ST ( _ ↦ ([],()) )
```

Die oben bereits skizzierte Formulierung der Stack-Operationen, etwa

```
do { mt
     ; push 4
     ; push 5
     ; x <- pop
     ; push 3
     ; push x
   }
```

kann jetzt tatsächlich verwendet werden und würde mit Hilfe der folgenden Regeln in Standardsyntax übersetzt:

$$
\begin{array}{lcl}
\texttt{do } \{exp\} & := & exp \\
\texttt{do } \{v \leftarrow exp; t_1; \ldots; t_n\} & := & exp \texttt{ >>= } (v \mapsto \texttt{do } \{t_1; \ldots; t_n\}) \\
\texttt{do } \{exp; t_1; \ldots; t_n\} & := & exp \texttt{ >>= } (\_ \mapsto \texttt{do } \{t_1; \ldots; t_n\})
\end{array}
$$

sowie

$$
\begin{array}{lcl}
\texttt{ST } t_0 \texttt{ >>= } f & = & \texttt{ST } (\ s_0 \mapsto t_1\ s_1 \\
 & & \qquad \texttt{where } (s_1, v) = t_0\ s_0 \\
 & & \qquad\quad (\texttt{ST } t_1) = f\ v\ ) \\
\texttt{return} & :: & \alpha \rightarrow \texttt{ST } \alpha \\
\texttt{return } v & = & \texttt{ST } (s \mapsto (s, v))
\end{array}
$$

## 14.6 Monaden-Gesetze

Wir haben eine Reihe von Anwendungen von Monaden gesehen. Damit eine Typklasse wirklich eine Monade ist, müssen die im folgenden beschriebenen Gesetze gelten.

Wir betrachten nochmals die Typen von `return` und `>>=`:

$$\begin{array}{lcl} \texttt{return} & :: & \alpha \to M\alpha \\ \texttt{(>>=)} & :: & M\alpha \to (\alpha \to M\beta) \to M\beta \end{array}$$

Wenn $M$ eine Monade ist, so muss gelten:

① `return` ist rechtsneutral bzgl. `>>=`:

$$p \texttt{ >>= return} \quad \equiv \quad p \ .$$

② `return` ist „quasi-linksneutral“ bzgl. `>>=` im folgenden Sinne:

$$(\texttt{return } e) \texttt{ >>= } q \quad \equiv \quad q\, e \ .$$

③ `>>=` ist „assoziativ“ im folgenden Sinne:

$$(p \texttt{ >>= } q) \texttt{ >>= } r \quad \equiv \quad p \texttt{ >>= } s \qquad \textbf{where } s\, x = (q\, x \texttt{ >>= } r)$$

Die Monaden-Gesetze lassen sich auch mit Hilfe der `do`-Notation aufschreiben:

① $\texttt{do}\ \{B; x \leftarrow p; \texttt{return}\ x\} \quad \equiv \quad \texttt{do}\ \{B; p\}.$

② $\texttt{do}\ \{B; x \leftarrow \texttt{return}\ e; C; r\} \quad \equiv \quad \texttt{do}\ \{B; C[e/x]; r[e/x]\}.$

③ $\texttt{do}\ \{B; x \leftarrow \texttt{do}\ \{C; p\}; D; r\} \quad \equiv \quad \texttt{do}\ \{B; C; x \leftarrow p; D; r\}.$

Als Konsequenz der `do`-Notation lässt sich noch eine weiteres Gesetz zeigen, das sog. „Collapse-Law“:

④ $\texttt{do}\ \{C; \texttt{do}\ \{D; r\}\} \quad \equiv \quad \texttt{do}\ \{C; D; r\}.$

In der Mathematik (Algebra) sind uns ähnliche Strukturen bekannt: **Monoide**.
Ein Monoid (Halbgruppe mit Einselement) ist eine Struktur $(M, \circ, e)$ mit einem zweistelligen Operator $\circ : M \times M \to M$ und einem Element $e \in M$, so dass gilt:

1. $\circ$ ist assoziativ: $\forall a, b, c \in M : (a \circ b) \circ c = a \circ (b \circ c)$.
2. $e$ ist Einselement bzgl. $\circ$: $\forall m \in M : e \circ m = m = m \circ e$.

Monaden verhalten sich sehr ähnlich, anstelle der zweistelligen Operation $\circ$ haben wir hier allerdings den Bind-Operator >>=. Man kann jedoch >>= auch mittels Monoid-Operatoren definieren.

**N.B.** Haskell prüft bei der Instanziierung der Typklasse `Monad` *nicht*, ob diese Gesetze mit den definierten Funktionen `return` und >>= wirklich erfüllt sind! Dies liegt in der Verantwortung des Programmierers.
**Ohne diese Gesetze ist die Semantik der `do`-Notation nicht konsistent.**

## 14.6.1 Monaden induzieren Monoide

Wir definieren abschließend einen Operator zur *Komposition* von monadischen Berechnungen: $\diamond$.

$$\begin{array}{lcl} (\diamond) & :: & (\alpha \to M\beta) \to (\beta \to M\gamma) \to (\alpha \to M\gamma) \\ (f \diamond g)\ x & = & f\ x\ \texttt{>>=}\ g \end{array}$$

oder, äquivalent mittels Section:

$$\begin{array}{lcl} f \diamond g & = & (\texttt{>>=}\ g) \cdot f \end{array}$$

Der Operator $(\diamond)$ arbeitet ähnlich wie eine Funktionskomposition $(\cdot)$, abgesehen davon, dass die Komponenten-Funktionen jeweils einen Typ $\alpha \to M\beta$ für passende $\alpha, \beta$ haben und die Reihenfolge der Komposition umgekehrt ist.

Mit Hilfe von $(\diamond)$ lässt sich der Bind-Operator ableiten:

$$(\texttt{>>=}\ g) := id \diamond g \ .$$

Funktionskomposition und monadischer Komposition sind über das sog. „*Leapfrog-Law*“ miteinander verbunden:

$$(f \diamond g) \cdot h = (f \cdot h) \diamond g \ .$$

Nun lassen sich die Monaden-Gesetze auch so formulieren, dass $((\alpha \to M\beta), \diamond, \texttt{return})$ ein Monoid mit Einselement `return` ist:

① Die ersten beiden Monaden-Gesetze besagen, dass `return` das Links- und Rechts-Einselement ist: $p \diamond \texttt{return} = p$ und $\texttt{return} \diamond q = q$.

② Das dritte Monaden-Gesetz besagt, dass $(\diamond)$ assoziativ ist: $(f \diamond g) \diamond h = f \diamond (g \diamond h)$.

Die Monaden-Gesetze sind also mit Hilfe von $(\diamond)$ einfacher zu formulieren und zu merken, die `do`-Notation dagegen eignet sich besser, mit Monaden zu programmieren.

## 14.6.2 Weitere Eigenschaften bei einigen Monaden

Neben den o.g. drei Monaden-Gesetzen gelten in manchen Monaden weitere Eigenschaften, etwa mit zwei speziellen Elementen `mzero` und `mplus`:

① `mzero >>=` $f =$ `mzero`

② $m$ `>>= (\x -> mzero)` = `mzero`

③ `mzero` ‘`mplus`‘ $m = m$

④ $m$ ‘`mplus`‘ `mzero` $= m$

Die zusätzlichen Gesetze sind also so wie in der „normalen“ Arithmetik mit `mzero` als 0, `mplus` als $+$ und `>>=` als $\times$.

Monaden mit diesen zusätzlichen Eigenschaften können in Haskell als Instanz der Typklasse `MonadPlus` deklariert werden:

```
class (Monad m) => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Tatsächlich ist z.B. `Maybe` eine Instanz von `MonadPlus`:

```
instance MonadPlus Maybe where
  mzero             = Nothing
  Nothing `mplus` x = x
  x `mplus` _       = x
```

`Nothing` spielt also die Rolle des „Nullelementes“ `mzero` und „Addition“ von zwei `Maybe`-Werten ergibt den ersten Wert, der nicht `Nothing` ist; wenn beide `Nothing` sind, dann ist es das Ergebnis auch.

Die Listen-Monade hat ebenfalls ein `mzero` (die leere Liste `[]`) und ein `mplus` (den Operator `++`).

Im Schaf-Kloning Beispiel könnten wir `mplus` ebenfalls einsetzen, um monadische Berechnungsergebnisse zu kombinieren, bspw. könnten wir eine Funktion

```
parent s = (mother s) `mplus` (father s)
```

definieren, die einen Elternteil liefert, falls einer existiert, `Nothing` andernfalls. Falls beide Elternteile existieren, so würde einer von beiden geliefert, abhängig von der konkreten Definition von `mplus` in der `Maybe` Monade.

## 14.6.3 Map und Join

Neben der Definition von Monaden mittels der beiden Grundfunktionen `return` und `>>=` kann man sie auch mittels `return` und zweier anderer Funktionen, `mapm` und `joinm`, definieren.

Diese `mapm`–`joinm`–`return` Formulierung lässt sich an der „Container"-Interpretation leicht erläutern:

- `return` legt ein $\alpha$-Element in den $M\ \alpha$-Container,
- `mapm` wendet eine Funktion $\alpha \to \beta$ auf einen $M\ \alpha$-Container an und liefert einen $M\ \beta$-Container,
- `joinm` nimmt einen Container von Containern und „sammelt" deren Inhalte auf:

$$\begin{array}{lcl} \texttt{return} & :: & \alpha \to M\ \alpha \\ \texttt{>>=} & :: & M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta \\ \\ \texttt{mapm} & :: & (\alpha \to \beta) \to (M\ \alpha \to M\ \beta) \\ \texttt{joinm} & :: & M\ (M\ \alpha) \to M\ \alpha \end{array}$$

Die beiden Formulierungen sind ineinander überführbar mittels

$$\begin{array}{lcll} (\texttt{mapm}\ f)\ ms & \equiv & ms\ \texttt{>>=}\ (\backslash x\ \texttt{->}\ \texttt{return}\ (f\ x)) & \equiv\ \texttt{do}\ \{x \leftarrow ms; \texttt{return}\ (f\ x)\} \\ \texttt{joinm}\ ms & \equiv & ms\ \texttt{>>=}\ (\backslash x\ \texttt{->}\ x) & \equiv\ \texttt{do}\ \{m \leftarrow ms; m\} \\ \\ m\ \texttt{>>=}\ f & \equiv & \texttt{joinm}\ ((\texttt{mapm}\ f)\ m) & \end{array}$$

Mit Hilfe der Monaden-Composition ($\diamond$) lassen sich `mapm` und `joinm` auch kompakt definieren als:

$$\begin{array}{lcl} \texttt{mapm}\ f & = & id \diamond (\texttt{return} \cdot f) \\ \texttt{joinm} & = & id \diamond id \end{array}$$

Bei `joinm` hat das erste Vorkommen von $id$ den Typ $id :: M\ (M\ \alpha) \to M\ (M\ \alpha)$, das zweite $id :: M\ \alpha \to M\ \alpha$.

Hier gelten die folgenden Gesetze:

$$\begin{array}{lcl} \texttt{mapm}\ id & = & id \\ \texttt{mapm}\ (f \cdot g) & = & \texttt{mapm}\ f \cdot \texttt{mapm}\ g \\ \\ \texttt{mapm}\ f \cdot \texttt{return} & = & \texttt{return} \cdot f \\ \texttt{mapm}\ f \cdot \texttt{joinm} & = & \texttt{joinm} \cdot \texttt{mapm}\ (\texttt{mapm}\ f) \\ \\ \texttt{joinm} \cdot \texttt{return} & = & id \\ \texttt{joinm} \cdot \texttt{mapm return} & = & id \\ \texttt{joinm} \cdot \texttt{mapm joinm} & = & \texttt{joinm} \cdot \texttt{joinm} \end{array}$$

**N.B.**

- auch diese sieben Gesetze sind wieder äquivalent zu den bekannten drei Monaden-Gesetzen...
- diese Definitionsweise von Monaden ist in der Kategorientheorie üblich

## 14.7 Zusammenfassung

Mit Hilfe von Monaden lassen sich in funktionalen Programmiersprachen *Berechnungen mit Seiteneffekten* kapseln. Dazu zählen insbesondere:

- Ein- und Ausgabe
- Zustandstransformationen
- Darstellungen verschiedenster Arten von „Containern“
- fehlende Werte, Ausnahmen, Fehler
- diverse „Bulk Data Types“ (wie z.B. Listen, Mengen, Multimengen, Bäume, ...)

Der `return`-Operator „**importiert**“ dabei einen Wert vom Typ $\alpha$ in eine Monade/Container vom Typ $M\ \alpha$, der Bind-Operator „**verbindet**“ Berechnungen innerhalb der Monade.

Ein „**Export**“ von Werten aus dem monadischen Typ heraus (eine Funktion vom Typ $M\ \alpha \to \alpha$) dagegen, ist (zumindest standardmäßig) *nicht* vorgesehen! Das ist absichtlich so, denn die Kapselung der Seiteneffekte innerhalb der monadischen Berechnungen würde durch diese aufgebrochen.

(Das schließt natürlich nicht aus, dass bei einzelnen Monaden solche Exportfunktionen dennoch sinnvoll und vorhanden sind.)

# Inhaltsverzeichnis