Rechnen und rechnen lassen

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

19. September 2005

(Die Folien sind online verfügbar: .../~ralf/talks.html#T44.)

Einführung

Rechnen:

- Rechnen ist uns vertraut.
- Rechnen kann man mit vielen Dingen: Zahlen, Bäumen, Musik, Bildern.
- Informatik ist die Wissenschaft vom Rechnen.

Rechnen lassen:

- Programmiersprache „Haskell“: `www.haskell.org`.

Einführung

Ziele der Veranstaltung:

- Interesse wecken an anderen Programmierparadigmen: funktionale bzw. werteorientierte Programmierung.
- Interesse wecken an der funktionalen Programmiersprache „Haskell“.
- Anregungen geben für den Informatik- und Mathematikunterricht; vielleicht auch für den Musik- oder Kunstunterricht.

Darüber hinaus:

- Programmieren in Haskell macht Spaß!

Überblick

- ✘ Rechnen mit Zahlen
- ✘ Rechnen mit Bäumen
- ✘ Rechnen mit Musik
- ✘ Rechnen mit Bildern

Rechnen mit Zahlen

- Beim Begriff „Rechnen“ denkt man zuerst an das Rechnen mit Zahlen.
- Wir werden später sehen, dass das nur ein hausbackener Spezialfall ist.
- Da uns das Rechnen mit Zahlen vertraut ist, steht es hier am Anfang.

Die Kunst beim Programmieren in Haskell besteht darin, Probleme in Rechenaufgaben zu verwandeln.

Vom Problem zur Rechenaufgabe

Problem:
Die Klasse 2c macht einen Ausflug in den Zoo. Es fahren 27 Schülerinnen und Schüler und 3 Lehrer mit. Die Fahrtkosten betragen 2 € für Kinder und 3 € für Erwachsene. Kinder zahlen 5 € Eintritt und Erwachsene 10 €. Wie teuer ist der Ausflug?

In der Grundschule lernt man, Textaufgaben in Rechenaufgaben zu verwandeln:

$$27 * (2 + 5) + 3 * (3 + 10)$$

Von der Rechenaufgabe zum Ergebnis

Das Ergebnis der Rechenaufgabe erhalten wir durch Ausrechnen:

$$\begin{aligned} 27 * (2 + 5) + 3 * (3 + 10) &\Longrightarrow 27 * 7 + 3 * (3 + 10) \\ &\Longrightarrow 189 + 3 * (3 + 10) \\ &\Longrightarrow 189 + 3 * 13 \\ &\Longrightarrow 189 + 39 \\ &\Longrightarrow 228 \end{aligned}$$

In diesem Beispiel gibt es mehrere Rechenwege:

$$\begin{aligned} 27 * (2 + 5) + 3 * (3 + 10) &\Longrightarrow 27 * 2 + 27 * 5 + 3 * (3 + 10) \\ &\Longrightarrow 27 * 2 + 27 * 5 + 3 * 3 + 3 * 10 \\ &\Longrightarrow \cdots \\ &\Longrightarrow 228 \end{aligned}$$

☞ Das Endergebnis ist aber stets das Gleiche.

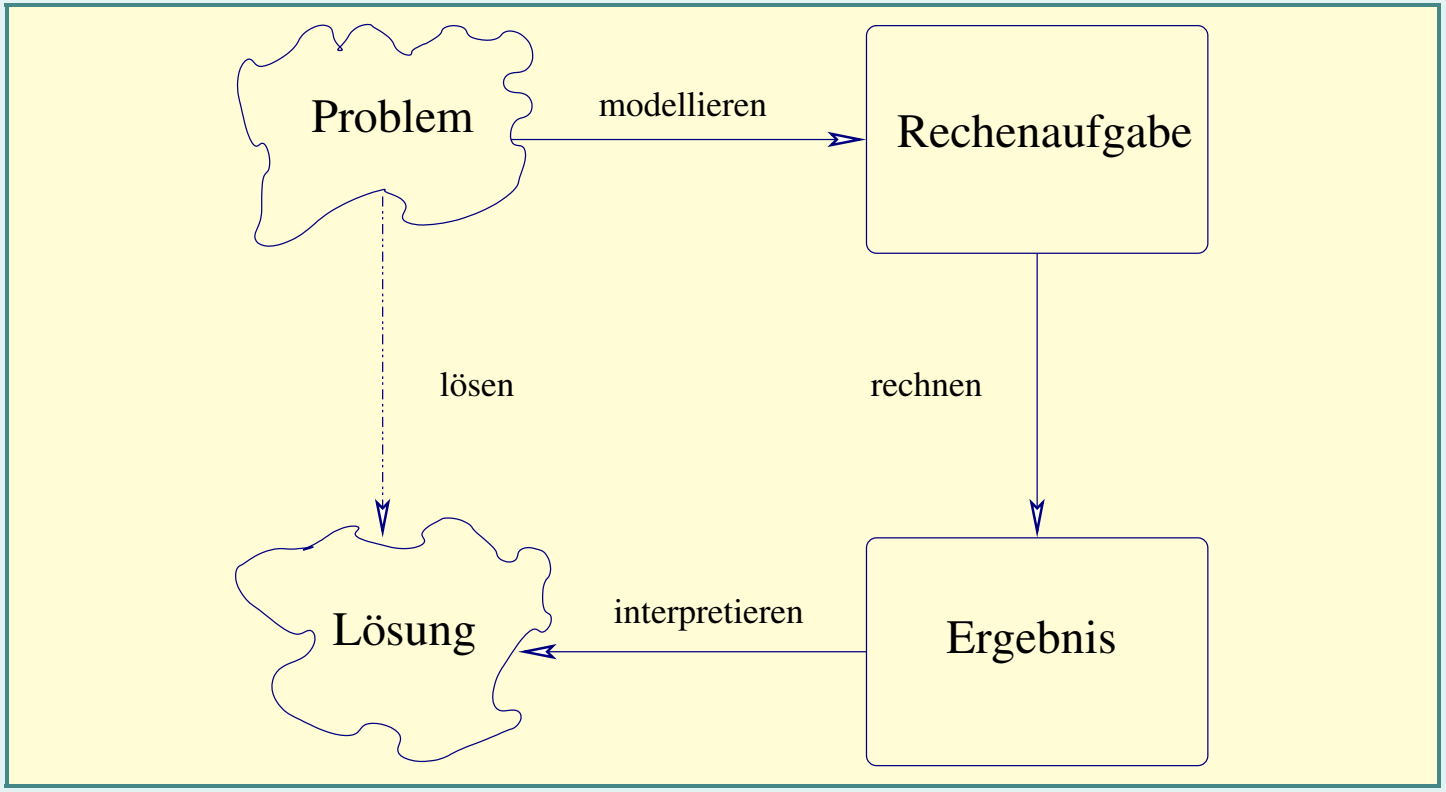
Vom Ergebnis zur Lösung

$$27 * (2 + 5) + 3 * (3 + 10) \Longrightarrow 228$$

Interpretieren wir das Ergebnis der Rechenaufgabe, erhalten wir die Lösung:

Lösung:
Der Ausflug kostet 228 €.

Modellbildung in der Informatik



Rechenregeln

- Rechnen ist der Natur nach etwas Mechanisches; es folgt festen Regeln.
- Die Regeln für Addition und Multiplikation natürlicher Zahlen (0, 1, 2 usw) lernt man in der Grundschule: schriftliche Addition und Multiplikation.

$$\begin{array}{rcccc} & & 5 & 9 & 5 \\ + & 4_1 & 7_1 & 1 & 1 \\ \hline = & 5 & 3 & 0 & 6 \end{array}$$

- In Haskell wie auch in anderen Programmiersprachen sind Zahlen, der Typ *Integer*, und die Regeln für die Grundrechenarten fest eingebaut.
- Im folgenden werden wir uns überlegen, wie diese Regeln aussehen könnten und sozusagen das Rad neu erfinden: wir überlegen uns, wie sich die natürlichen Zahlen darstellen lassen und stellen Rechenregeln für die Addition auf.

Selbstgestrickte Ziffern

- Zahlen haben eine Struktur: eine Zahl ist eine Folge von Ziffern.
- Dezimalzahlen sind Folgen von Dezimalziffern (0 – 9).
- Binärzahlen sind Folgen von Binärziffern (0 – 1).
- Für das Aufstellen der Rechenregeln ist es bequemer, Binärzahlen zu verwenden. Warum?
- Der Clou: wir verwenden statt der Ziffern 0 und 1 die Ziffern 1 und 2.

Die folgende Deklaration führt Namen für die Binärziffern ein.

$$\mathbf{data}\ \mathit{Digit} = \mathit{One} \mid \mathit{Two}$$

☞ Haskell: eine **data**-Deklaration definiert einen Typ, hier Digit, und Konstruktoren, hier One und Two.

Exkurs: Listen alias Folgen oder Sequenzen

Folgen alias Listen sind in Haskell vordefiniert. Bildungsvorschrift:

- $[\,]$ ist die leere Liste vom Typ $[a]$;
- ist x ein Element vom Typ a und xs eine Liste vom Typ $[a]$, dann ist auch $x : xs$ eine Liste vom Typ $[a]$. Wir nennen x den Kopf der Liste und xs den Listenrest.

Beispiele:

$2 : (3 : (5 : (7 : [\,])))$	ist eine Liste vom Typ $[Integer]$
$One : (Two : [\,])$	ist eine Liste vom Typ $[Digit]$
$\texttt{'H'} : (\texttt{'a'} : (\texttt{'s'} : (\texttt{'k'} : (\texttt{'e'} : (\texttt{'l'} : (\texttt{'l'} : [\,]))))))$	ist eine Liste vom Typ $[Char]$
$(1 : [\,]) : (2 : 3 : [\,]) : [\,]$	ist eine Liste vom Typ $[[Integer]]$

Selbstgestrickte Zahlen

Mit Hilfe von Binärziffern und Listen können wir unsere eigenen Zahlen stricken:

$$\mathbf{type}\ Natural = [Digit]$$

☞ Haskell: eine **type**-Deklaration definiert eine Abkürzung: $Natural$ ist das Gleiche wie $[Digit]$.

Beispiele:

☞ Wir vereinbaren, dass die niederwertigste Binärziffer links steht — in der Dezimalnotation steht die niederwertigste Ziffer traditionell rechts.

dezimal	binär	binär in Haskell
0	0	$[\,]$
1	1	$One : [\,]$
2	2	$Two : [\,]$
3	11	$One : (One : [\,])$
4	21	$Two : (One : [\,])$
11	112	$One : (One : (Two : [\,]))$
17	1211	$One : (Two : (One : (One : [\,])))$
4711	111212211211	$One : (One : (One : (Two : (One : (Two : (Two : (One : (One : (Two : (One : (One : [\,])))))))))))$

Bedeutung einer selbstgestrickten Zahl

Mit Hilfe von Rechenregeln können wir die selbstgestrickten Zahlen in die vordefinierten Zahlen konvertieren. Auf diese Weise können wir die Bedeutung der selbstgestrickten Zahlen festlegen.

$$
\begin{array}{lll}
integer :: Natural & \longrightarrow Integer \\
integer & ([\,]) & = 0 \\
integer & (One : ds) & = 1 + 2 * integer\ (ds) \\
integer & (Two : ds) & = 2 + 2 * integer\ (ds)
\end{array}
$$

☞ Haskell: $integer$ ist eine Funktion; die Funktion hat eine Eingabe, ein Element vom Typ $Natural$, und eine Ausgabe, ein Element vom Typ $Integer$.

Rechnen: selbstgestrickten Zahlen konvertieren

Mit Hilfe der Rechenregeln für $integer$ können wir den Wert einer selbstgestrickten Zahl bestimmen:

$$
\begin{aligned}
& integer\ (One : (One : (Two : [\,])))\\
\Longrightarrow\ & 1 + 2 * (integer\ (One : (Two : [\,])))\\
\Longrightarrow\ & 1 + 2 * (1 + 2 * (integer\ (Two : [\,])))\\
\Longrightarrow\ & 1 + 2 * (1 + 2 * (2 + 2 * (integer\ [\,])))\\
\Longrightarrow\ & 1 + 2 * (1 + 2 * (2 + 2 * 0))\\
\Longrightarrow\ & 1 + 2 * (1 + 2 * 2)\\
\Longrightarrow\ & 1 + 2 * 5\\
\Longrightarrow\ & 11
\end{aligned}
$$

☞ Haskell: ein Teilausdruck, der auf die linke Seite einer Rechenregel passt, wird durch die rechte Seite ersetzt. Solange, bis keine Rechenregel mehr anwendbar ist.

Eine kleine Farbenlehre

- $Natural$ ist ein Typ.
- One ist ein Konstruktor.
- $integer$ ist eine Funktion.

- Für Funktionen stellen wir Rechenregeln auf.
- Konstruktoren sind oder konstruieren Daten; für sie gibt es keine Rechenregeln.
- Beim Rechnen werden Rechenregeln solange angewendet, bis Daten vorliegen — bis alles blau ist.
- Typen schaffen Ordnung; sie gruppieren Daten.

Zählen

Es ist ungewöhnlich, die Ziffern 1 und 2 statt der Ziffern 0 und 1 zu verwenden. Man fragt sich: Lassen sich alle natürlichen Zahlen überhaupt auf diese Weise darstellen?

Ja! Die folgenden Rechenregeln zeigen, wie man zählt.

$$
\begin{array}{lll}
incr :: Natural & \to Natural \\
incr & ([\,]) & = One : [\,] \\
incr & (One : ds) & = Two : ds \\
incr & (Two : ds) & = One : (incr\ (ds))
\end{array}
$$

Zählen — Fortsetzung

Wenn wir $incr$ wiederholt ausgehend von $[\,]$ anwenden, erhalten wir die Darstellungen aller natürlicher Zahlen.

$$
\begin{array}{ll}
incr\ (Two : (One : [\,])) & \text{-- } 4+1 \\
\Longrightarrow One : (incr\ (One : [\,])) & \\
\Longrightarrow One : (Two : [\,]) & \text{-- } 5 \\
 & \\
incr\ (One : (Two : [\,])) & \text{-- } 5+1 \\
\Longrightarrow Two : (Two : [\,]) & \text{-- } 6 \\
 & \\
incr\ (Two : (Two : [\,])) & \text{-- } 6+1 \\
\Longrightarrow One : (incr\ (Two : [\,])) & \\
\Longrightarrow One : (One : (incr\ ([\,]))) & \\
\Longrightarrow One : (One : (One : [\,])) & \text{-- } 7
\end{array}
$$

Addition

Auch für die Addition lassen sich Rechenregeln aufstellen:

$$
\begin{array}{lll}
add :: (Natural,\ Natural) & \rightarrow & Natural \\
add\ \ ([\,],\ \ \ \ ys) & = & ys \\
add\ \ (xs,\ \ \ \ [\,]) & = & xs \\
add\ \ (One : xs, One : ys) & = & Two : add\ (xs, ys) \\
add\ \ (One : xs, Two : ys) & = & One : incr\ (add\ (xs, ys)) \\
add\ \ (Two : xs, One : ys) & = & One : incr\ (add\ (xs, ys)) \\
add\ \ (Two : xs, Two : ys) & = & Two : incr\ (add\ (xs, ys))
\end{array}
$$

☞ Die Funktion add hat zwei Eingaben, die beiden Summanden, und eine Ausgabe, die Summe.

☞ $incr$ erledigt den Übertrag.

Zur Übung: warum haben wir die Binär- und nicht die Dezimalnotation verwendet?

Rechnen: addieren

Jetzt können wir addieren:

$$
\begin{aligned}
&add\ (One : Two : One : One : [\,], One : Two : Two : [\,]) && \text{-- } 17 + 13 \\
&\Longrightarrow Two : add\ (Two : One : One : [\,], Two : Two : [\,]) \\
&\Longrightarrow Two : Two : incr\ (add\ (One : One : [\,], Two : [\,])) \\
&\Longrightarrow Two : Two : incr\ (One : incr\ (add\ (One : [\,], [\,]))) \\
&\Longrightarrow Two : Two : incr\ (One : incr\ (One : [\,])) \\
&\Longrightarrow Two : Two : Two : incr\ (One : [\,]) \\
&\Longrightarrow Two : Two : Two : Two : [\,] && \text{-- } 30
\end{aligned}
$$

Addition — Fortsetzung

Bei der schriftlichen Addition werden in einem Schritt drei Ziffern addiert: die Ziffern der beiden Summanden und der Übertrag vom vorangegangenen Schritt. Diese Beobachtung führt zu der folgenden, alternativen Definition von add.

$$
\begin{aligned}
add\ (One : xs, Two : ys) &= One : addWithCarry\ (One, xs, ys) \\
add\ (Two : xs, One : ys) &= One : addWithCarry\ (One, xs, ys) \\
add\ (Two : xs, Two : ys) &= Two : addWithCarry\ (One, xs, ys)
\end{aligned}
$$

Zur Übung: stelle Rechenregeln für $addWithCarry$ auf.

$$addWithCarry :: (Digit, Natural, Natural) \longrightarrow Natural$$

Überblick

- ✔ Rechnen mit Zahlen
- ✘ Rechnen mit Bäumen
- ✘ Rechnen mit Musik
- ✘ Rechnen mit Bildern

Rechnen mit Bäumen

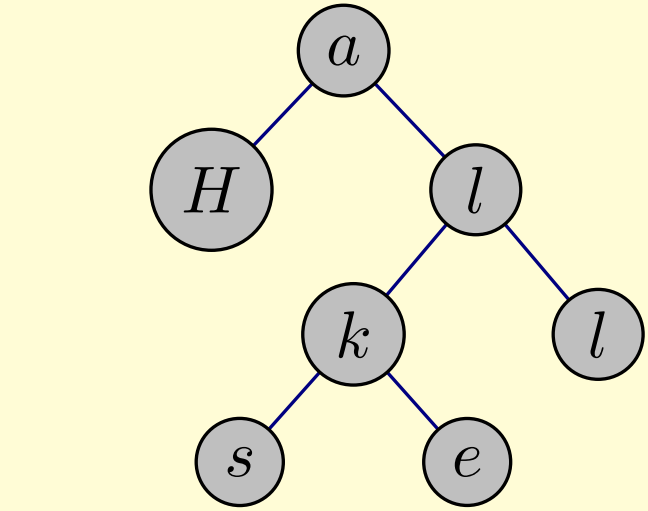
- Bäume werden in der Informatik verwendet, um Daten hierarchisch zu organisieren.
- Wir kennen „Bäume“ bzw. hierarchische Strukturen aus dem täglichen Leben: Bücher (Teile, Kapitel, Abschnitte usw), Turnierbäume, Organigramme von Unternehmen, Familienstammbäume oder Pedigrees usw.
- Wir wollen Bäume im folgenden verwenden, um Daten zu sortieren:

Problem:
Gegeben ist eine Folge von Elementen: b, a, c. Bringe die Elemente in aufsteigende Reihenfolge — sortiere die Folge.

Rechenaufgabe:
$sort\ (\texttt{"b"} : \texttt{"a"} : \texttt{"c"} : [\,])$

Bäume in der Informatik

Bäume wachsen in der Informatik von oben nach unten: a ist die Wurzel, H, s, e und l sind Blätter.



☞ Wir haben einen Binärbaum vor uns: jeder Knoten hat genau zwei Teilbäume.

Binärbäume

Binärbäume in Haskell:

$$\textbf{type}\ \mathit{Elem} = \mathit{String}$$

$$\textbf{data}\ \mathit{Tree}\ = \mathit{Empty} \mid \mathit{Node}\ (\mathit{Elem}, \mathit{Tree}, \mathit{Tree})$$

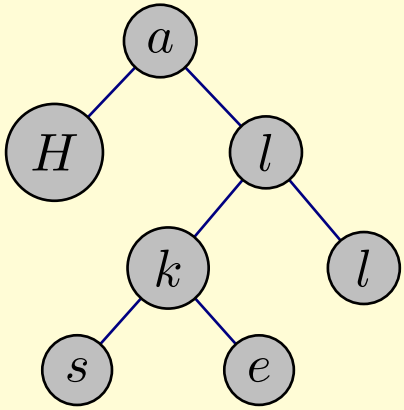
$$\begin{array}{l} \mathit{leaf} :: \mathit{Elem} \rightarrow \mathit{Tree} \\ \mathit{leaf}\quad (x)\quad = \mathit{Node}\ (x, \mathit{Empty}, \mathit{Empty}) \end{array}$$

☞ Haskell: Tree ist ein rekursiver Datentyp. Die Datentypdefinition kann als Bildungsvorschrift für Bäume gelesen werden:

- Empty ist der leere Baum vom Typ $\mathit{Tree}\ a$;
- ist x ein Element vom Typ Elem und sind l und r Bäume vom Typ Tree, dann ist auch $\mathit{Node}\ (x, l, r)$ ein Baum vom Typ Tree. Wir nennen x die Knotenmarkierung, l den linken und r den rechten Teilbaum.

Beispiel für einen Binärbaum

$$\begin{aligned} &\mathit{Node}\ (\texttt{"a"}, \mathit{leaf}\ (\texttt{"H"}),\\ &\qquad \mathit{Node}\ (\texttt{"l"}, \mathit{Node}\ (\texttt{"k"}, \mathit{leaf}\ (\texttt{"s"}),\\ &\qquad\qquad \mathit{leaf}\ (\texttt{"e"})),\\ &\qquad\qquad \mathit{leaf}\ (\texttt{"l"}))) \end{aligned}$$



Größe eines Baums

Mit Hilfe zweier Rechenregeln können wir bestimmen, wieviele Knoten ein Baum enthält.

$$
\begin{array}{lll}
size :: Tree & \rightarrow Integer \\
size\ \ (Empty) & = 0 \\
size\ \ (Node\ (x, l, r)) & = 1 + size\ (l) + size\ (r)
\end{array}
$$

☞ **Haskell:** Konstruktoren können auf der linken Seite der Rechenregeln verwendet werden; sie unterscheiden Fälle und erlauben es, Namen für die Komponenten zu vergeben (diese sind frei wählbar).

Heaps

- Zurück zum Sortieren: das erste Element der geordneten Folge ist das Minimum der ursprünglichen Folge.
- Wir können eine Folge sortieren, indem wir wiederholt das Minimum bestimmen und entfernen.
- Macht man das naiv, benötigt man viele Vergleiche. Mit Hilfe eines sogenannten Heaps können wir uns Informationen über die Anordnung von Elementen merken.
- Ein Heap ist ein Binärbaum mit der folgenden Eigenschaft: die Markierung jedes Knotens ist kleiner als die Markierung der darunterliegenden Knoten.

$$\textbf{type } \mathit{Heap} = \mathit{Tree}$$

Einfügen in einen Heap

Wir überlegen uns zunächst, wie wir ein Element in einen Heap einfügen.

$$
\begin{array}{llll}
insert :: Elem \to Heap & & \to Heap \\
insert & a & (Empty) & = leaf\ (a) \\
insert & a & (Node\ (a', l, r)) & = Node\ (min\ a\ a', ?, ?)
\end{array}
$$

Es gibt 4 Möglichkeiten für die zweite Rechenregel:

$$
\begin{array}{ll}
Node\ (min\ a\ a', insert\ (max\ a\ a')\ l, & r\ ) \\
Node\ (min\ a\ a', l, & insert\ (max\ a\ a')\ r) \\
Node\ (min\ a\ a', insert\ (max\ a\ a')\ r, & l\ ) \\
Node\ (min\ a\ a', r, & insert\ (max\ a\ a')\ l\ )
\end{array}
$$

Einfügen in einen Heap

Wir fügen stets in den rechten Teilbaum ein und vertauschen gleichzeitig den linken und den rechten Baum.

$$
\begin{array}{lll}
insert :: Elem \to Heap & \to Heap \\
insert \quad a\ (Empty) & = leaf\ (a) \\
insert \quad a\ (Node\ (a', l, r)) & = Node\ (min\ a\ a', insert\ (max\ a\ a')\ r, l)
\end{array}
$$

Rechnen: wiederholtes Einfügen

$$
\begin{aligned}
&\mathit{insert}\ \texttt{"a"}\ (\mathit{insert}\ \texttt{"b"}\ (\mathit{insert}\ \texttt{"c"}\ \mathit{Empty}))\\
&\Longrightarrow \mathit{insert}\ \texttt{"a"}\ (\mathit{insert}\ \texttt{"b"}\ (\mathit{leaf}\ (\texttt{"c"})))\\
&\Longrightarrow \mathit{insert}\ \texttt{"a"}\ (\mathit{insert}\ \texttt{"b"}\ (\mathit{Node}\ (\texttt{"c"}, \mathit{Empty}, \mathit{Empty})))\\
&\Longrightarrow \mathit{insert}\ \texttt{"a"}\ (\mathit{Node}\ (\texttt{"b"}, \mathit{insert}\ \texttt{"c"}\ \mathit{Empty}, \mathit{Empty}))\\
&\Longrightarrow \mathit{insert}\ \texttt{"a"}\ (\mathit{Node}\ (\texttt{"b"}, \mathit{leaf}\ (\texttt{"c"}), \mathit{Empty}))\\
&\Longrightarrow \mathit{Node}\ (\texttt{"a"}, \mathit{insert}\ \texttt{"b"}\ \mathit{Empty}, \mathit{leaf}\ (\texttt{"c"}))\\
&\Longrightarrow \mathit{Node}\ (\texttt{"a"}, \mathit{leaf}\ (\texttt{"b"}), \mathit{leaf}\ (\texttt{"c"}))\\
&\Longrightarrow \mathit{Node}\ (\texttt{"a"}, \mathit{Node}\ (\texttt{"b"}, \mathit{Empty}, \mathit{Empty}), \mathit{Node}\ (\texttt{"c"}, \mathit{Empty}, \mathit{Empty}))
\end{aligned}
$$

☞ Die konstruierten Bäume haben folgende Eigenschaft: für jeden Knoten $\mathit{Node}\ (x, l, r)$ gilt: $0 \leqslant \mathit{size}\ (l) - \mathit{size}\ (r) \leqslant 1$. Wenn wir $2^n - 1$ Elemente einfügen, erhalten wir einen vollständig ausgeglichenen Baum!

Heapsort

Heapsort arbeitet in 2 Phasen: in der ersten Phase wird der Heap aufgebaut; in der zweiten Phase wird der Heap abgebaut,

$$
\begin{aligned}
&sort :: [Elem] \to [Elem] \\
&sort = unbuild \cdot build \\
\\
&build :: [Elem] \to Heap \\
&build = foldr\ insert\ Empty
\end{aligned}
$$

☞ Haskell: '$\cdot$' komponiert zwei Funktionen: $(f \cdot g)\ (x) = f\ (g\ (x))$.

☞ Haskell: fügt $insert\ x\ h$ ein Element in den Heap h ein, so fügt $foldr\ insert\ h\ xs$ eine ganze Liste ein.

Exkurs: *foldr*

Die Funktion *foldr* erlaubt es, Rechenregeln auf Listen abzukürzen.

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

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

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

☞ $foldr\ (\oplus)\ e$ ersetzt ':' durch '$\oplus$', $[\,]$ durch e und rechnet den resultierenden Ausdruck aus.

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

Kahlschlag — oder: Abbau von Heaps

Wie überführen wir einen Heap in eine aufsteigend geordnete Liste?

$$
\begin{array}{lll}
unbuild :: Heap & \rightarrow & [Elem] \\
unbuild\ \ (Empty) & = & [\,] \\
unbuild\ \ (Node\ (a, l, r))) & = & a : ?
\end{array}
$$

Es gibt mindestens zwei Alternativen für die zweite Rechenregel:

$$
\begin{array}{l}
a : mergeList\ (unbuild\ l, unbuild\ r) \\
a : unbuild\ (mergeHeap\ (l, r))
\end{array}
$$

Kahlschlag — Fortsetzung

Wir wählen die erste Alternative:

$$
\begin{array}{lll}
unbuild :: Heap & \to & [Elem] \\
unbuild\ \ (Empty) & = & [\,] \\
unbuild\ \ (Node\ (a, l, r)) & = & a : mergeList\ (unbuild\ l, unbuild\ r)
\end{array}
$$

Die Hilfsfunktion $mergeList$ 'mischt' zwei geordnete Listen.

$$
\begin{array}{lll}
mergeList :: ([Elem], [Elem]) \to [Elem] & & \\
mergeList\ \ ([\,], \quad bs) & = & bs \\
mergeList\ \ (as, \quad [\,]) & = & as \\
mergeList\ \ (a : as, \quad b : bs) & = & \textbf{if}\ a \leqslant b\ \textbf{then}\ a : mergeList\ (as, b : bs) \\
 & & \textbf{else}\ \ b : mergeList\ (a : as, bs)
\end{array}
$$

Zur Übung: stelle Rechenregeln für die zweite Alternative, $mergeHeap$, auf.

Rechnen: sortieren

Mit Hilfe der aufgestellten Rechenregeln können wir schließlich die gegebene Folge von Elementen in aufsteigende Reihenfolge bringen.

$$
\begin{aligned}
& sort\ (\texttt{"b"} : \texttt{"a"} : \texttt{"c"} : [\,]) \\
& \Longrightarrow (unbuild \cdot build)\ (\texttt{"b"} : \texttt{"a"} : \texttt{"c"} : [\,]) \\
& \Longrightarrow unbuild\ (build\ (\texttt{"b"} : \texttt{"a"} : \texttt{"c"} : [\,])) \\
& \Longrightarrow unbuild\ (Node\ (\texttt{"a"}, leaf\ \texttt{"b"}, leaf\ \texttt{"c"}) \\
& \Longrightarrow \texttt{"a"} : mergeList\ (unbuild\ (leaf\ \texttt{"b"}), unbuild\ (leaf\ \texttt{"c"})) \\
& \Longrightarrow \texttt{"a"} : mergeList\ (\texttt{"b"} : [\,], \texttt{"c"} : [\,]) \\
& \Longrightarrow \texttt{"a"} : \texttt{"b"} : mergeList\ ([\,], \texttt{"c"} : [\,]) \\
& \Longrightarrow \texttt{"a"} : \texttt{"b"} : \texttt{"c"} : [\,]
\end{aligned}
$$

☞ Wir haben eine ganze Klasse von Problemen gelöst.

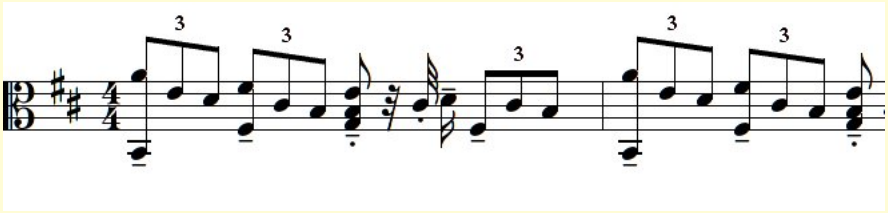
Überblick

- ✔ Rechnen mit Zahlen
- ✔ Rechnen mit Bäumen
- ✘ Rechnen mit Musik
- ✘ Rechnen mit Bildern

Rechnen mit Musik

Wie können wir Musik darstellen?

Traditionell werden zum Beispiel Partituren verwendet:



Wir erfinden ein Vokabular für Noten, Pausen, Komposition von Musikstücken usw.

Musikstücke

```
data Music = Note Pitch Dur
           | Rest Dur
           | Music :+: Music
           | Music :=: Music
           | Tempo (Ratio Integer) Music
           | Trans  Integer Music
           | Instr  IName Music
```

- *Note* x d: eine einzelne Note; *Rest* d: eine Pause;
- m_1 :+: m_2: das Stück m_1 gefolgt von m_2;
- m_1 :=: m_2: das Stück m_1 gleichzeitig mit m_2;
- *Tempo* a m: das Stück m um den Faktor a schneller;
- *Trans* a m: das Stück m um a Halbtonschritte höher;
- *Instr* i m: das Stück m gespielt von i.

Beispiele

$$
\begin{aligned}
&play\ run\\
&play\ (run :+: run :+: run)\\
&play\ (run :+: reverse\ run)\\
&play\ (run :=: reverse\ run)\\
&play\ (Tempo\ (1\ /\ 2)\ run)\\
&play\ (Trans\ 8\ run)\\
&play\ (Trans\ (-8)\ run :=: Trans\ 8\ run)\\
&play\ (Instr\ Flute\ run)\\
&play\ (Tempo\ (1\ /\ 10)\ (Instr\ FX4Atmosphere\ run))\\
&play\ (Tempo\ (1\ /\ 4)\ (Instr\ Flute\ run :+: Instr\ Koto\ (reverse\ run)))
\end{aligned}
$$

☞ run ist ein einfaches Musikstück; die Komposition von run beschreiben wir später.

Tonhöhe und Tondauer

Eine Tonhöhe wird durch einen Halbton und durch eine Oktave bestimmt. Die Tondauer ist eine Bruchzahl.

$$
\begin{array}{lll}
\textbf{type } \mathit{Pitch} & = & (\mathit{PitchClass}, \mathit{Octave}) \\
\textbf{data } \mathit{PitchClass} & = & \mathit{Cf} \mid \mathit{C} \mid \mathit{Cs} \mid \mathit{Df} \mid \mathit{D} \mid \mathit{Ds} \mid \mathit{Ef} \mid \mathit{E} \mid \mathit{Es} \mid \mathit{Ff} \mid \mathit{F} \mid \mathit{Fs} \\
 & & \mid \mathit{Gf} \mid \mathit{G} \mid \mathit{Gs} \mid \mathit{Af} \mid \mathit{A} \mid \mathit{As} \mid \mathit{Bf} \mid \mathit{B} \mid \mathit{Bs} \\
\textbf{type } \mathit{Octave} & = & \mathit{Integer} \\
 & & \\
\textbf{type } \mathit{Dur} & = & \mathit{Ratio\ Integer}
\end{array}
$$

☞ Lies Cf als C-flat"($C\flat$, Ces) und Cs als C-sharp"($C\sharp$, Cis).

Musikinstrumente

```
data IName
   = AcousticGrandPiano | BrightAcousticPiano | ElectricGrandPiano
   | ...
   | Applause           | Gunshot             | Percussion
```

Abkürzungen

Um Musikstücke kürzer notieren zu können, erfinden wir Abkürzungen:

$$
\begin{aligned}
&cf :: Octave \to Dur \to Music \\
&cf \quad o \qquad\quad d \quad = Note\ (Cf, o)\ d \\
&\ldots \\
&\\
&r :: Dur \to Music \\
&r \quad d \quad = Rest\ d
\end{aligned}
$$

Komposition von Musikstücken

Mehrere Musikstücke hintereinander oder gleichzeitig:

$$
\begin{aligned}
&line, chord :: [Music] \to Music \\
&line \quad = foldr\ (:+:)\ (Rest\ 0) \\
&chord = foldr\ (:=:)\ (Rest\ 0)
\end{aligned}
$$

Ein Musikstück n-mal wiederholen:

$$
\begin{aligned}
&times :: Integer \to Music \to Music \\
&times \quad 0 \qquad\quad m \qquad = Rest\ 0 \\
&times \quad (n+1) \quad m \qquad = m :+: (times\ n\ m)
\end{aligned}
$$

„Children's Song No. 6“ — Basslinie

Auf den folgenden 3 Folien codieren wir „Children's Song No. 6“ von Chick Corea als Musikstück.

$$bassLine = times\ 3\ b1\ :+:\ times\ 2\ b2\ :+:\ times\ 4\ b3\ :+:\ times\ 5\ b1$$

$$b1 = setDur\ \tfrac{3}{8}\ [b\ \ 3, fs\ \ 4, g\ \ 4, fs\ \ 4]$$
$$b2 = setDur\ \tfrac{3}{8}\ [b\ \ 3, es\ 4, fs\ 4, es\ 4]$$
$$b3 = setDur\ \tfrac{3}{8}\ [as\ 3, fs\ \ 4, g\ \ 4, fs\ \ 4]$$

$$setDur\ d\ l = line\ [n\ d \mid n \leftarrow l]$$

$$play\ (Tempo\ 3\ bassLine)$$

„Children's Song No. 6“ — Hauptstimme

$$
\begin{aligned}
mainVoice &= times\ 3\ v1\ :+:\ v2 \\
v1 &= v1a\ :+:\ v1b \\
v1a &= setDur\ \tfrac{1}{8}\ [a\ 5, e\ 5, d\ 5, fs\ 5, cs\ 5, b\ 4, e\ 5, b\ 4] \\
v1b &= line\ [cs\ 5\ \tfrac{1}{32}, d\ 5\ (\tfrac{1}{4} - \tfrac{1}{32}), cs\ 5\ \tfrac{1}{8}, b\ 4\ \tfrac{1}{8}] \\
\\
v2 &= v2a\ :+:\ v2b\ :+:\ v2c\ :+:\ v2d\ :+:\ v2e\ :+:\ v2f \\
v2a &= line\ [cs\ 5\ (\tfrac{3}{4} + \tfrac{3}{4}), d\ 5\ \tfrac{3}{4}, f\ 5\ \tfrac{1}{2}, gs\ 5\ \tfrac{1}{4}, fs\ 5\ (\tfrac{1}{2} + \tfrac{1}{8}), g\ 5\ \tfrac{1}{8}] \\
v2b &= setDur\ \tfrac{1}{8}\ [fs\ 5, e\ 5, cs\ 5, as\ 4]\ :+:\ a\ 4\ \tfrac{3}{8} \\
&\quad :+:\ setDur\ \tfrac{1}{8}\ [as\ 4, cs\ 5, fs\ 5, e\ 5, fs\ 5, g\ 5, as\ 5] \\
v2c &= line\ [cs\ 6\ (\tfrac{1}{2} + \tfrac{1}{8}), d\ 6\ \tfrac{1}{8}, cs\ 6\ \tfrac{1}{8}, e\ 5\ \tfrac{1}{8}]\ :+:\ r\tfrac{1}{8} \\
&\quad :+:\ line\ [as\ 5\ \tfrac{1}{8}, a\ 5\ \tfrac{1}{8}, g\ 5\ \tfrac{1}{8}, d\ 5\ \tfrac{1}{4}, c\ 5\ \tfrac{1}{8}, cs\ 5\ \tfrac{1}{8}] \\
v2d &= setDur\ \tfrac{1}{8}\ [fs\ 5, cs\ 5, e\ 5, cs\ 5, a\ 4, as\ 4, d\ 5, e\ 5, fs\ 5] \\
&\quad :+:\ line\ [fs\ 5\ \tfrac{1}{32}, e\ 5\ (\tfrac{1}{4} - \tfrac{1}{32}), d\ 5\ \tfrac{1}{8}, e\ 5\ \tfrac{1}{32}, d\ 5\ (\tfrac{1}{4} - \tfrac{1}{32}), \\
&\qquad cs\ 5\ \tfrac{1}{8}, d\ 5\ \tfrac{1}{32}, cs\ 5\ (\tfrac{1}{4} - \tfrac{1}{32}), b\ 4\ (\tfrac{1}{8} + \tfrac{1}{2})] \\
v2e &= line\ [cs\ 5\ \tfrac{1}{8}, b\ 4\ \tfrac{1}{8}, fs\ 5\ \tfrac{1}{8}, a\ 5\ \tfrac{1}{8}, b\ 5\ (\tfrac{1}{2} + \tfrac{1}{4}), a\ 5\ \tfrac{1}{8}, \\
&\qquad fs\ 5\ \tfrac{1}{8}, e\ 5\ \tfrac{1}{4}, d\ 5\ \tfrac{1}{8}, fs\ 5\ \tfrac{1}{8}, e\ 5\ \tfrac{1}{2}, d\ 5\ \tfrac{1}{2}, fs\ 5\ \tfrac{1}{4}] \\
v2f &= Tempo\ (3\ /\ 2)\ (line\ [cs\ 5\ \tfrac{1}{8}, d\ 5\ \tfrac{1}{8}, cs\ 5\ \tfrac{1}{8}])\ :+:\ b\ 4\ (3 * \tfrac{3}{4} + \tfrac{1}{2})
\end{aligned}
$$

„Children's Song No. 6“ von Chick Corea

$$
\begin{aligned}
&childSong6 = Instr\ AcousticGrandPiano\ (Tempo\ 3\ (\\
&\qquad bassLine :=: mainVoice))\\
&\\
&variation_1\\
&\quad = Tempo\ 3\ (\ Instr\ FretlessBass\ bassLine\\
&\qquad :=: Instr\ BrightAcousticPiano\ mainVoice)\\
&variation_2\\
&\quad = Tempo\ 3\ (\ Instr\ FretlessBass\ bassLine\\
&\qquad :=: Instr\ FX4Atmosphere\ mainVoice\\
&\qquad :=: delay\ \tfrac{1}{8}\ (Instr\ AcousticGrandPiano\ mainVoice))
\end{aligned}
$$

☞ Man sieht, komponieren ist zusammensetzen.

Länge eines Musikstücks

Auch mit Musikstücken kann man rechnen: dur (kurz für duration) bestimmt die Länge eines Musikstücks.

$$
\begin{array}{lll}
dur :: Music & \to & Dur \\
dur\ \ (Note\ \_\ d) & = & d \\
dur\ \ (Rest\ d) & = & d \\
dur\ \ (m_1 :+: m_2) & = & dur\ m_1 + dur\ m_2 \\
dur\ \ (m_1 :=: m_2) & = & max\ (dur\ m_1)\ (dur\ m_2) \\
dur\ \ (Tempo\ a\ m) & = & dur\ m\ /\ a \\
dur\ \ (Trans\ \ \_\ m) & = & dur\ m \\
dur\ \ (Instr\ \ \_\ m) & = & dur\ m
\end{array}
$$

Spiegeln eines Musikstücks

Jetzt wird's experimentell: ein Musikstück rückwärts spielen.

$$
\begin{array}{lll}
\mathit{reverse} :: \mathit{Music} & \to \mathit{Music} \\
\mathit{reverse}\ \ (\mathit{Note}\ x\ d) & = \mathit{Note}\ x\ d \\
\mathit{reverse}\ \ (\mathit{Rest}\ d) & = \mathit{Rest}\ d \\
\mathit{reverse}\ \ (\mathit{Tempo}\ a\ m) & = \mathit{Tempo}\ a\ (\mathit{reverse}\ m) \\
\mathit{reverse}\ \ (\mathit{Trans}\ \ i\ m) & = \mathit{Trans}\ \ i\ (\mathit{reverse}\ m) \\
\mathit{reverse}\ \ (\mathit{Instr}\ \ \ i\ m) & = \mathit{Instr}\ \ \ i\ (\mathit{reverse}\ m) \\
\mathit{reverse}\ \ (m_1 :+: m_2) & = \mathit{reverse}\ m_2 :+: \mathit{reverse}\ m_1 \\
\mathit{reverse}\ \ (m_1 :=: m_2) & \\
\quad = \textbf{let}\ d_1 = \mathit{dur}\ m_1 & \\
\qquad\ \ d_2 = \mathit{dur}\ m_2 & \\
\quad\ \ \textbf{in}\ \ \textbf{if}\ d_1 > d_2\ \textbf{then}\ \mathit{reverse}\ m_1 :=: (\mathit{Rest}\ (d_1 - d_2) :+: \mathit{reverse}\ m_2) & \\
\qquad\qquad\qquad\ \ \textbf{else}\ \ (\mathit{Rest}\ (d_2 - d_1) :+: \mathit{reverse}\ m_1) :=: \mathit{reverse}\ m_2 &
\end{array}
$$

$$\mathit{play}\ (\mathit{reverse}\ \mathit{childSong6})$$

Ausschneiden eines Teilmusikstücks

$$
\begin{array}{l}
repeat :: Music \to Music \\
repeat \quad m \qquad = m :+: repeat\ m
\end{array}
$$

☞ $repeat\ m$ wiederholt das Stück m „ad nauseum“.

$$
\begin{array}{lll}
cut :: Dur \to Music & & \to Music \\
cut \quad d & m \mid d \leqslant 0 & = Rest\ 0 \\
cut \quad d & (Note\ x\ d_0) & = Note\ x\ (min\ d_0\ d) \\
cut \quad d & (Rest\ d_0) & = Rest\ (min\ d_0\ d) \\
cut \quad d & (m_1 :=: m_2) & = cut\ d\ m_1 :=: cut\ d\ m_2 \\
cut \quad d & (Tempo\ a\ m) & = Tempo\ a\ (cut\ (d * a)\ m) \\
cut \quad d & (Trans\ \ i\ \ m) & = Trans\ \ i\ \ (cut\ d\ m) \\
cut \quad d & (Instr\ \ i\ \ m) & = Instr\ \ i\ \ (cut\ d\ m) \\
cut \quad d & (m_1 :+: m_2) & = \mathbf{let}\ m_1' = cut\ d\ m_1 \\
 & & \qquad\ m_2' = cut\ (d - dur\ m_1')\ m_2 \\
 & & \mathbf{in}\ \ m_1' :+: m_2'
\end{array}
$$

Schlagzeug

Die Funktionen *repeat* und *cut* sind nützlich für Schlagzeugsequenzen:

$$
\begin{aligned}
&p_1, p_2 :: Music \\
&p_1 = perc\ LowTom\ \tfrac{1}{4} \\
&p_2 = perc\ AcousticSnare\ \tfrac{1}{8} \\
&\\
&percLine = p_1 :+: r\tfrac{1}{4} :+: p_2 :+: r\tfrac{1}{4} :+: p_2 :+: p_1 :+: p_1 :+: r\tfrac{1}{4} :+: p_2 :+: r\tfrac{1}{8} \\
&\\
&funkGroove = Tempo\ 3\ (Instr\ Percussion\ (cut\ 8\ (repeat \\
&\qquad\qquad (percLine :=: times\ 16\ (perc\ ClosedHiHat\ \tfrac{1}{8})))))
\end{aligned}
$$

Experimentelle Musik

$$\textbf{type}\ T = Music \to Music \quad \text{-- Transformation}$$

$$\begin{array}{llll} rep :: T \to T \to Integer \to Music \to Music \\ rep & f & g & 0 & m & = Rest\ 0 \\ rep & f & g & (n+1) & m & = m :=: g\ (rep\ f\ g\ n\ (f\ m)) \end{array}$$

$$\begin{array}{lll} delay :: Dur \to Music \to Music \\ delay & d & m & = Rest\ d :+: m \end{array}$$

$$run = rep\ (Trans\ 5)\ (delay\ \tfrac{1}{32})\ 8\ (c\ 4\ \tfrac{1}{32})$$

$$cascade = rep\ (Trans\ 4)\ (delay\ \tfrac{1}{8})\ 8\ run$$

$$cascades = rep\ id\ (delay\ \tfrac{1}{16})\ 2\ cascade$$

$$waterfall = Instr\ Vibraphone\ (cascades :+: reverse\ cascades)$$

☞ id ist die identische Transformation: $id\ m = m$.

Überblick

- ✔ Rechnen mit Zahlen
- ✔ Rechnen mit Bäumen
- ✔ Rechnen mit Musik
- ✘ Rechnen mit Bildern

Rechnen mit Bildern

- Wie können wir 2-dimensionale Bilder modellieren?
- Computergenerierte Bilder werden oft aus geometrischen Modellen erzeugt: aus Kombinationen von Linien, Kurven, Polygonen usw.
- Wir modellieren Bilder direkt:

$$\textbf{type}\ \mathit{Image} = \mathit{Point} \to \mathit{Colour}$$

Ein Bild ist eine Abbildung von Punkten in der Ebene auf Farben.

Bilder

Es ist nützlich, die Definition von Bildern noch etwas zu verallgemeinern.

```
type Map a = Point → a
```

☞ Haskell: a ist ein Typparameter.

```
type Image     = Map Colour     -- Farbbild
type Region    = Map Bool       -- Schwarzweißbild oder Punktmenge
type Fraction  = Double         -- 0.0 ... 1.0
type Greyscale = Map Fraction   -- Graustufenbild
```

Punkte

Punkte in der Ebene können wahlweise kartesisch oder polar angegeben werden.

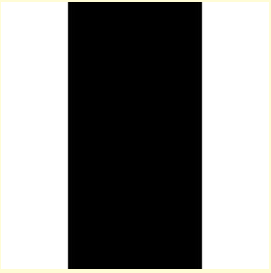
```
data Point

cartesian :: Double → Double → Point   -- x- und y-Koordinate
polar     :: Double → Double → Point   -- Abstand und Winkel

x :: Point → Double   -- x-Koordinate
y :: Point → Double   -- y-Koordinate
Δ :: Point → Double   -- Abstand vom Ursprung
θ :: Point → Double   -- Winkel zwischen x-Achse und Richtungsvektor
```

Sammelsurium

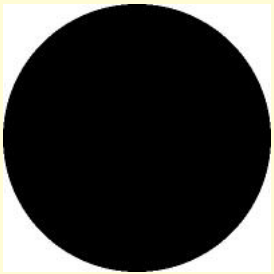
Im folgenden machen wir Gebrauch von den folgenden Funktionen:

- $|x|$ ist der Absolutwert von x: $|3.14| = 3.14 = |-3.14|$;
- $\lfloor x \rfloor$ ist die größte ganze Zahl kleiner gleich x: $\lfloor 3.14 \rfloor = 3$ und $\lfloor -3.14 \rfloor = -4$;
- $even\ a$ testet, ob die ganze Zahl a gerade ist.

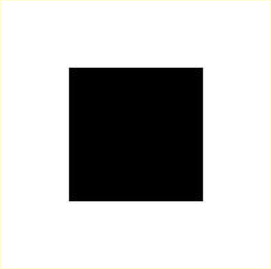
Regionen — Schwarzweißbilder



vstrip



circle



square 0.5

$$
\begin{aligned}
&\mathit{vstrip} \;\; :: \mathit{Region}\\
&\mathit{vstrip}\ p = |x\ p| \leqslant 0.5\\
&\\
&\mathit{circle} \;\; :: \mathit{Region}\\
&\mathit{circle}\ p = \Delta\ p \leqslant 1\\
&\\
&\mathit{square} \;\; :: \mathit{Double} \to \mathit{Region}\\
&\mathit{square}\ r\ p = |x\ p| \leqslant r \wedge |y\ p| \leqslant r
\end{aligned}
$$

☞ Wir zeigen $[-1 \ldots 1] \times [-1 \ldots 1]$ Ausschnitte der Bilder.

Exkurs: Lambda

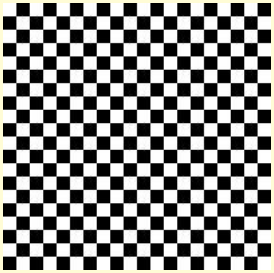
$$
\begin{aligned}
&vstrip \quad :: Region \\
&vstrip\ p = |x\ p| \leqslant 0.5
\end{aligned}
$$

Funktionen können auch durch Ausdrücke beschrieben werden:

$$
\begin{aligned}
&vstrip :: Region \\
&vstrip = \lambda p \rightarrow |x\ p| \leqslant 0.5
\end{aligned}
$$

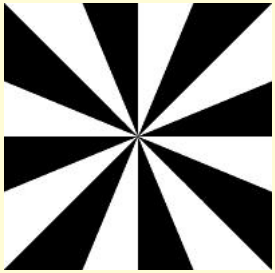
☞ Haskell: $\lambda x \rightarrow e$ ist die anonyme Funktion, die das Argument x auf e abbildet; x ist der Funktionsparameter; e ist der Funktionsrumpf.

Regionen: Schachbrett, Ringe und Kuchenstücke



checker · *scale* 10

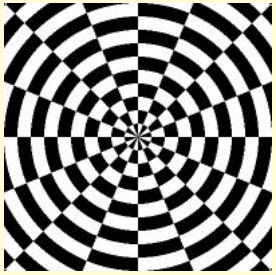
rings · *scale* 10



wedges 8

$$
\begin{aligned}
&\mathit{checker} :: \mathit{Region}\\
&\mathit{checker}\ p = \mathit{even}\ (\lfloor x\ p \rfloor + \lfloor y\ p \rfloor)\\
&\\
&\mathit{rings} :: \mathit{Region}\\
&\mathit{rings}\ p = \mathit{even}\ \lfloor \Delta\ p \rfloor\\
&\\
&\mathit{wedges} :: \mathit{Integer} \to \mathit{Region}\\
&\mathit{wedges}\ n\ p = \mathit{even}\ \lfloor \theta\ p * n\ /\ \pi \rfloor
\end{aligned}
$$

Regionen: polares Schachbrett



polarChecker 8 · *scale* 10

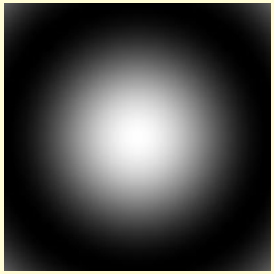
☞ *polarChecker* kombiniert *rings* und *wedges*.

$$\begin{aligned} &polarChecker &&:: Integer \to Region \\ &polarChecker\ n\ p &&= even\ (\lfloor \Delta\ p \rfloor + \lfloor \theta\ p * n\ /\ \pi \rfloor) \end{aligned}$$

Zum Knobeln: die folgende Beschreibung ist äquivalent. Warum?

$$\begin{aligned} &polarChecker' &&:: Integer \to Region \\ &polarChecker'\ n\ p &&= rings\ p == wedges\ n\ p \end{aligned}$$

Graustufenbilder



wave

wave · *scale* 10

$$\begin{aligned} &wave \quad :: \ Greyscale \\ &wave\ p = (1 + cos\ (\pi * \Delta\ p))\ /\ 2 \end{aligned}$$

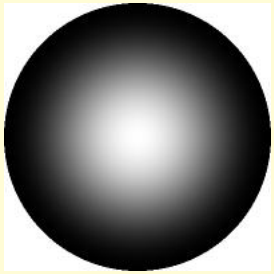
☞ Zur Erinnerung: $cos\ 0 = 1$ und $cos\ \pi = -1$.

Farbbilder

Um Farben darzustellen, verwenden wir das RGBA Farbmodell: eine Farbe ist aus Rot-, Grün- und Blauanteilen zusammengesetzt; der sogenannte Alpha-Kanal bestimmt die Transparenz eines Punktes.

```
data Colour

rgba :: Double → Double → Double → Double → Colour

transparent :: Colour
black, white :: Colour
...
blue, yellow :: Colour

r, g, b, a :: Colour → Double
```

Ausschneiden eines Teilbildes



crop circle wave

crop vstrip wave · *scale* 2

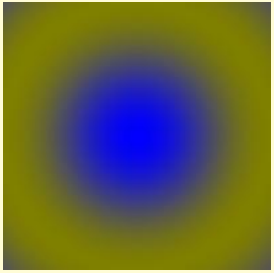
$$
\begin{aligned}
&\mathit{crop} && :: \mathit{Region} \to \mathit{Image} \to \mathit{Image} \\
&\mathit{crop}\ \mathit{reg}\ \mathit{im} = \lambda p \to \textbf{if}\ \mathit{reg}\ p\ \textbf{then}\ \mathit{im}\ p\ \textbf{else}\ \mathit{transparent}
\end{aligned}
$$

☞ *crop* stutzt ein Bild auf eine Region zurecht.

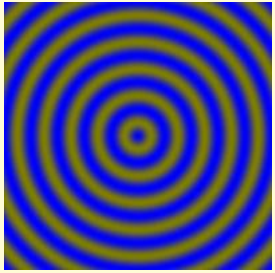
Farbübergänge

$$lerp :: Greyscale \to Image \to Image \to Image$$

☞ Die Funktion $lerp$ überlagert zwei Bilder; das erste Argument bestimmt das Mischungsverhältnis: $w * a_1 + (1 - w) * a_2$.



$bywave$



$bywave \cdot scale\ 10$

$lerp\ wave\ lisa\ (\lambda p \to blue)$

$$bywave :: Image$$
$$bywave = lerp\ wave\ (\lambda p \to blue)\ (\lambda p \to yellow)$$

Räumliche Transformationen

Räumliche Transformationen erlauben es uns, Bilder zu verkleinern, zu verschieben, zu verzerren ...

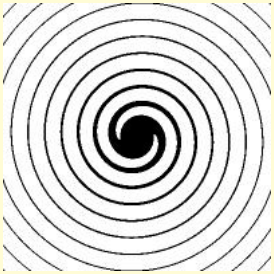
Eine Transformation ist einfach eine Abbildung von Punkten auf Punkte:

$$\textbf{type}\ \mathit{Transform} = \mathit{Point} \to \mathit{Point}$$

$$\begin{aligned} &\mathit{scale} :: \mathit{Double} \to \mathit{Transform} \\ &\mathit{scale}\ s\ p = \mathit{cartesian}\ (x\ p\ /\ s)\ (y\ p\ /\ s) \end{aligned}$$

$$\begin{aligned} &\mathit{rotate} :: \mathit{Double} \to \mathit{Transform} \\ &\mathit{rotate}\ w\ p = \mathit{polar}\ (\Delta\ p)\ (\theta\ p - w) \end{aligned}$$

Verwirbeln



vstrip · *swirl* 1 · *scale* 4

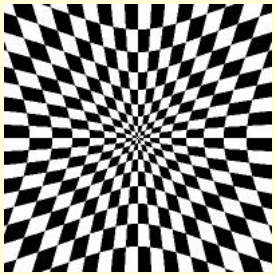
vstrip · *swirl* 5 · *scale* 4

halfPlane · *swirl* 1 · *scale* 4

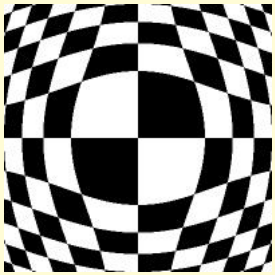
```
swirl      :: Double → Transform
swirl r p = rotate (Δ p * 2 * π / r) p

halfPlane  :: Region
halfPlane p = x p ⩾ 0
```

☞ Der Parameter r legt die Entfernung fest, bei der ein Punkt durch den vollständigen Kreis rotiert wird.

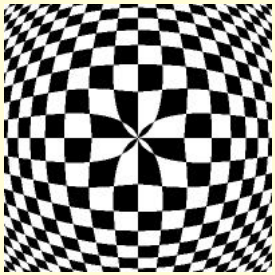
Polare Transformationen



checker · *distify sqrt* · *scale* 100



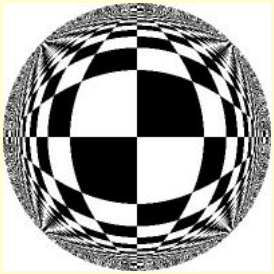
checker · *distify sq* · *scale* 2



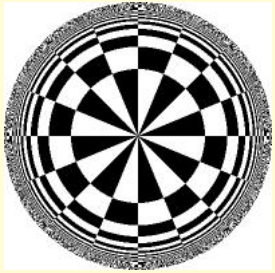
checker · *distify exp* · *scale* 2

$$
\begin{aligned}
&\mathit{distify} \quad :: (\mathit{Double} \to \mathit{Double}) \to \mathit{Transform}\\
&\mathit{distify}\ f\ p = \mathit{polar}\ (f\ (\Delta\ p))\ (\theta\ p)\\
&\\
&\mathit{sq} \quad :: \mathit{Double} \to \mathit{Double}\\
&\mathit{sq}\ x = x * x
\end{aligned}
$$

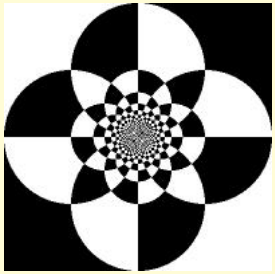
Kreislimits — Unendlichkeit sichtbar machen



circleLimit checker



circleLimit (*polarChecker* 8)

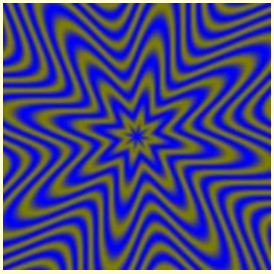


distInvert checker

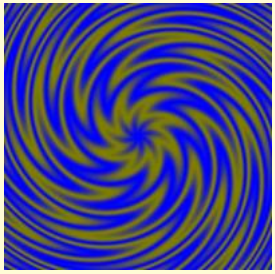
Die Funktion *circleLimit* presst ein Bild auf eine Kreisscheibe.

$$
\begin{aligned}
&\mathit{circleLimit} &&:: \mathit{Image} \to \mathit{Image}\\
&\mathit{circleLimit}\ \mathit{im} &&= \mathit{crop}\ \mathit{circle}\ (\mathit{im} \cdot \mathit{distify}\ (\lambda x \to x\ /\ (1 - x)))\\
&\\
&\mathit{distInvert} &&:: \mathit{Image} \to \mathit{Image}\\
&\mathit{distInvert}\ \mathit{im} &&= \mathit{im} \cdot \mathit{distify}\ (\lambda x \to 1\ /\ x)
\end{aligned}
$$

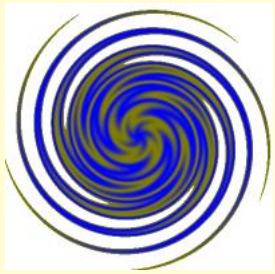
Sterntransformationen



im_1



im_2

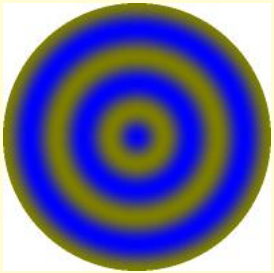


im_3

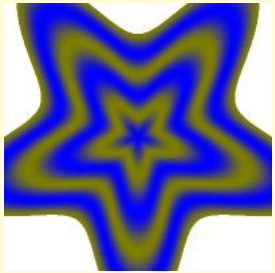
$$
\begin{aligned}
&star \quad :: Integer \to Double \to Transform\\
&star\ n\ s\ p = polar\ (\Delta\ p * (1 + s * sin\ (n * \theta\ p)))\ (\theta\ p)\\
&\\
&im_1, im_2, im_3 :: Image\\
&im_1 = bywave \cdot star\ 8\ 0.3 \cdot scale\ 10\\
&im_2 = bywave \cdot star\ 8\ 0.3 \cdot swirl\ 10 \cdot scale\ 5\\
&im_3 = crop\ circle\ (bywave \cdot scale\ 5) \cdot star\ 5\ 0.3 \cdot swirl\ 1 \cdot scale\ 1.4
\end{aligned}
$$

☞ Der erste Parameter von *star* legt die Anzahl der Zacken fest, der zweite bestimmt die Länge der Zacken.

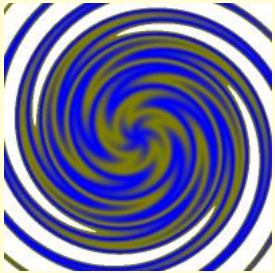
Schrittweise Entwicklung des Strudels



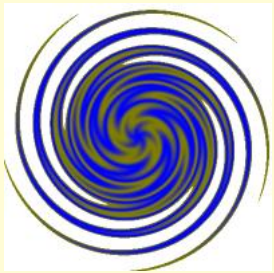
$step_1$



$step_2$



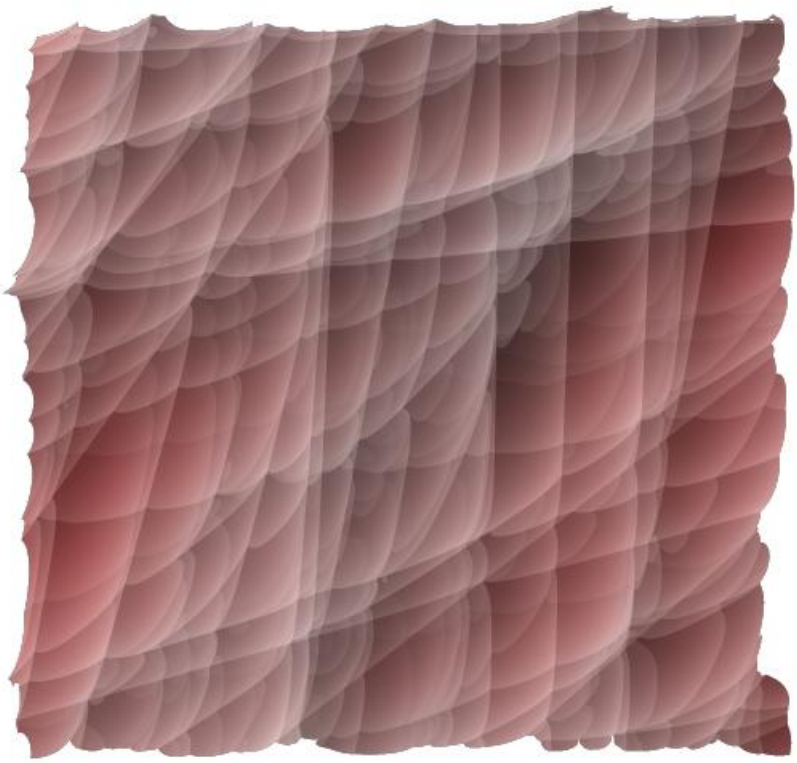
$step_3$



$step_4 = im_3$

$step_1 = crop\ circle\ (bywave \cdot scale\ 5)$
$step_2 = step_1 \cdot star\ 5\ 0.3$ -- 5 Zacken
$step_3 = step_2 \cdot swirl\ 1$
$step_4 = step_3 \cdot scale\ 1.4$ -- $= im_3$

Rotes Papier — zerknittert



Überblick

- ✔ Rechnen mit Zahlen
- ✔ Rechnen mit Bäumen
- ✔ Rechnen mit Musik
- ✔ Rechnen mit Bildern

Zusammenfassung

- Die Kunst beim Programmieren in Haskell besteht darin, Probleme in Rechenaufgaben zu verwandeln.
- Man erfindet zunächst Vokabular, mit dem man einen Weltausschnitt modellieren kann:
  - Substantive: *Natural*, *Tree*, *Music*, *Image* ...
  - Verben: *add*, *sort*, *dur*, *reverse*, ..., *crop*, *scale*, *rotate* ...
- Für die Verben stellt man Rechenregeln auf.
- Teilprobleme führen zu neuem Vokabular und zu weiteren Rechenregeln.
- Kleinere Rechenaufgaben lassen sich von Hand ausrechnen; die restlichen überlässt man einem Rechner.

Literatur

Allgemeine Informationen und Haskell Implementierungen (Open Source):

- Homepage: `www.haskell.org`,
- Wikipedia:
  `http://de.wikipedia.org/wiki/Haskell_(Programmiersprache)`,
- Mein Haskell-Kurs:
  `http://www.iai.uni-bonn.de/~ralf/teaching/Hskurs_toc.html`.

Empfehlenswerte Textbücher:

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

Literatur — Fortsetzung

- Paul Hudak, *Haskore music tutorial*, in: *Second International School on Advanced Functional Programming*, 36–68, Springer Verlag, LNCS 1129.
- Andrew Cooke, *pancito — algorithmic art*, `http://www.acooke.org/jara/pancito/`.
- Conal Elliot, *Functional images*, in: Jeremy Gibbons und Oege de Moor, Herausgeber, *The Fun of Programming*, 131–150. Palgrave Macmillan, 2003. ISBN 1-4039-0772-2 Hardback, ISBN 0-333-99285-7 Paperback.