Praktische Informatik I

Funktionale Abstraktion und Datenabstraktion

Bernd Krieg-Brückner
Universität Bremen
WS '98/'99

empfohlene Literatur:

Simon Thompson:
"Haskell: The Craft of Functional Programming",
Addison Wesley, 1996

Organisatorisches

Bernd Krieg-Brückner

MZH 8071, Tel. 3660, Sprechstunde: nach Vereinbarung

Sekretariat:

MZH 8070, Tel. 2676

Tutorien

Wiss. Mitarbeiter/ Tutoren:

Dr. Kolyang, Axel Lankenau, Dr. Christoph Lüth, Lutz Zegartowski

Literatur, Arbeitsblätter; Kernfragen

werden vorher ausgegeben, zuhause gelesen und gemeinsam diskutiert

Beispielaufgaben

werden vorher ausgegeben und gemeinsam erarbeitet

Wie löse ich ein Problem? Wie entwickle ich ein Programm?

Übungsaufgaben im Praktikumsbetrieb

Arbeitsgruppen mit ≤ 3 Studenten bzw. Studentinnen

ca. 6 Übungsblätter, 2-wöchig

```
blattOK b   | punktZahl b >= 14   = 1
            | otherwise           = 0
scheinOK bs = sum (map blattOK bs) >= length bs – 1
```

Übungsaufgaben zum Vorlesungsstoff + Praktische Übungen am Rechner

Programmieren in Haskell mit HUGS auf neuen 400 MHz PCs unter Linux (bzw. auf SUN Rechnern)

Dokumentation: Lösungsidee, Entwurfsentscheidungen, Programmtext mit Tests (Rechnerausdrucke!)

übersichtlich, verständlich programmieren, dokumentieren!!

auf Verlangen (nach DPO) benoteter Leistungsnachweis

am Schluß kurzes Prüfungsgespräch zur Überprüfung der Individualität

Einleitung

Grundbegriffe der Programmierung

- Abstraktion vom Detail
- nah am Problem – fern von der konkreten Maschine
- Funktionale Programmiersprache: *Haskell*
- PI II: Übergang zu imperativer, objektorientierter Programmiersprache: z.B. *Java*

Tutorien, Programmier-Praktikum

- Lernen am Beispiel
- praktische Erfahrung bei der Programmentwicklung "im kleinen"
- Entwicklung von Programmen, (später:) Beweise
- Ausführung / Testen am Rechner

Begriff des Algorithmus

Ethymologie

nach Muhammed ibn Musa abu Djafar al-Choresmi, 780-850

auch Al Khwarizmi, aus Choresmien (heute Usbekistan):

"Aufgabensammlung für Kaufleute und Testamentvollstrecker", Bagdad

lat. Übersetzung: liber algorithmi

Brett und Ziffernrechner (15. Jh)

- *Abacisten*: röm. Abacus, Rechnen "auf den Linien"
- *Algorithmiker*: Schriften der Scholastiker, Rechnen mit Ziffern
- Beispiel: Rechenvorschrift: Verdoppeln nach Adam Riese, 1574

Begriff des Algorithmus

Algorithmus

- Verarbeitungsvorschrift, schematisches (Lösungs)verfahren
- Beispiele: Konzertprogramm, Kochrezept, Gebrauchsanweisung
- Beispiel: Rezept für ein Hustenmittel

  3-4 mal täglich 15-20 Tropfen, am besten in heißem Zuckerwasser
- Beispiel: Bedienungsanleitung für einen Warenautomaten
  1. Geldstück einwerfen
  2. Gewünschtes Fach einstellen
  3. Zugstange ziehen und festhalten bis Ware herunterfällt

     Bei Versagen Knopf drücken
- Beispiel: Bedienungsanleitung für einen Münzfernsprecher

Begriff des Algorithmus

Definition: Algorithmus

*Ein **Algorithmus** ist ein allgemeines Verfahren mit einer präzisen, endlichen Beschreibung unter Verwendung effektiver elementarer (Verarbeitungs-) Schritte.*

- präzise: in einer genau festgelegten Sprache abgefaßt,

  (Syntax, Semantik, Pragmatik)

  z.B. natürliche Sprache (informell!), Sprache der Mathematik, Programmiersprache
- Beschreibung: auf dem Papier, im Kopf, im Rechner
- effektiv: ausführbar, z.B. auf einer Maschine
- Satz elementarer Verarbeitungsschritte ist gegeben
- Verfahren beschreibt Abfolge der Schritte
- allgemein: für eine Klasse von Eingabe- Daten

Beispiel: Sortieren eines Stapels von Karteikarten

Aufgabenstellung:

Ein gegebener Stapel x von Karteikarten ist zu sortieren;

die Ordnung (≤) sei gegeben (z.b. lexikographisch nach Namen)

Algorithmus 1: Sortieren durch Mischen

(1) ist x leer oder einelementig, so ist x sortiert;

(2) enthält x mehr als eine Karteikarte,

dann spalte man x in zwei nichtleere Stapel auf, sortiere die beiden Stapel und mische die beiden sortierten Stapel zu einem sortierten Stapel

(nach einer Art Reißverschlußprinzip).

Algorithmus 2: Sortieren durch Einsortieren

Neben x sei ein Stapel y gegeben, der anfangs leer ist. y sei stets sortiert *(Invariante).*

(1) ist x leer, so ist y der gesuchte sortierte Stapel;

(2) ist x nicht leer, so wähle eine Karteikarte in x aus und sortiere sie in y ein.

Wende nun den Algorithmus auf den verkleinerten Stapel x und den vergrößerten Stapel y an.

Algorithmus 3: Sortieren durch Auswählen

Zusätzlich zu x sei ein Stapel y gegeben, der anfangs leer ist.
Alle Karteikarten in y seien stets größer oder gleich als die in x. y sei stets sortiert.

(1) ist x leer, so ist y der gesuchte sortierte Stapel;

(2) ist x nicht leer,

so entnehme die größte Karteikarte aus x und füge sie vorne an y an.

Wende nun den Algorithmus auf den verkleinerten Stapel x
und den vergrößerten Stapel y an.

Algorithmus 4: Sortieren durch Vertauschen

(1) sind in x je zwei aufeinanderfolgende Karten in der richtigen

Reihenfolge (≤), so ist x der gesuchte sortierte Stapel;

(2) enthält x zwei aufeinanderfolgende Karten in falscher Reihenfolge (>),
so vertausche diese beiden Karten

und wende den Algorithmus auf den so erhaltenen Stapel an.

Programmentwicklung

Bestandteile der Entwicklung:

was: *Aufgabenstellung*: Charakterisierung der Eigenschaften

wie: *Algorithmus*: effektives Verfahren

warum: *Dokumentation* der Entwurfsentscheidungen

OK: *Beweis* der Korrektheit gegenüber der Aufgabenstellung
inklusive der *Terminierung*: bricht nach endlicher Zeit ab

Beispiel: größter gemeinsamer Teiler

Aufgabenstellung:

Gegeben seien zwei ganze Zahlen a, b mit a>0 und b>0;

gesucht ist der größte gemeinsame Teiler ggT(a,b),

d.h. diejenige Zahl t mit t>0, so daß gilt:

t ***teilt*** $a \wedge t$ ***teilt*** $b \wedge$ (***für alle*** $y > 0$: y ***teilt*** $a \wedge y$ ***teilt*** $b \rightarrow y \leq t$)

für alle, existiert

Algorithmus: (nach Euklid)

(1) falls a = b, dann ist ggT(a, b) = a;

(2) falls a < b, dann wende den Algorithmus ggT auf (a, b-a) an;

<, <=, ≤

(3) falls a > b, dann wende den Algorithmus ggT auf (a-b, b) an;

Gleichungen / Rechenregeln:

```
ggT a b  | a == b   = a,                    -- ggT a a = a
         | a < b    = ggT  a (b-a)
         | a > b    = ggT (a-b)  b
```

ggt 4 7

Elemente eines Algorithmus

Verwendung

– von elementaren Verarbeitungsschritten
– von anderen Algorithmen
– von sich selber: Zurückführung auf dasselbe Problem, Rekursion

Parametrisierung

– Versorgung mit (veränderten) Eingabewerten
– Parameter abstrahieren zu Klasse von Problemen

Fallunterscheidung

– Bedingungen, die Wahrheitswerte ("Boole'sche Werte") liefern
– nur wenn die Bedingung wahr ist, darf der Verarbeitungsschritt ausgeführt werden
– mehrere Bedingungen können gleichzeitig erfüllt sein

Klassifizierung von Algorithmen

Ein Algorithmus heißt

sequentiell, wenn die Verarbeitungsschritte stets hintereinander ausgeführt werden

parallel, wenn gewisse Verarbeitungsschritte nebeneinander ausgeführt werden können

rekursiv, wenn er sich selbst verwendet

Beispiele: `ggT`, Sortieren

Spezialfall: Endrekursion, Wiederholung (solange ...)

Ein Algorithmus a heißt (bzgl. eines Algorithmus b)

hierarchisch abgestützt, wenn a b verwendet, aber nicht umgekehrt

Beispiel: Sortieren durch Mischen, Multiplikation durch Addition

(man spricht auch von Hilfsfunktionen)

verschränkt rekursiv, wenn a b verwendet, und umgekehrt b a

gerade - ungerade, sotieren –> mischen

Klassifizierung von Algorithmen

Ein Algorithmus heißt (für eine Eingabe)

terminierend, wenn er nach endlich vielen Schritten endet

Gegenbeispiel: Kettenbruchentwicklung, Reihenentwicklung

1 + 1/2 + 1/4 + 1/8 + ... *divergierend*

jeder Schritt liefert ein genaueres Ergebnis,

aber der Algorithmus terminiert nicht

deterministisch, *nicht-deterministisch*
wenn in der Auswahl der Schritte keine Freiheit besteht

Gegenbeispiel: Minimum zweier Zahlen

(1) falls a ≤ b, dann ist das Ergebnis a;

(2) falls a ≥ b, dann ist das Ergebnis b;

Gegenbeispiel: Orakel, Zufallsgenerator (Vorsicht!):

(1) falls wahr, dann ist das Ergebnis 0;

(2) falls wahr, dann ist das Ergebnis 1;

dinierende Philosophen

determiniert, wenn das Resultat eindeutig bestimmt ist

Beispiel: Minimum

Gegenbeispiel: Orakel, Wähle irgendein Element aus

hier: Beschränkung auf determinierte Algorithmen;

Bedingungen sollten *disjunkt* sein;

aber: in Haskell ist die Reihenfolge der Regeln relevant

Beispiel:

```
ggT a b
  | a < b       = ggT  a (b-a)
  | a > b       = ggT (a-b)  b
  | otherwise   = a                  -- not (a < b) && not (a > b)    d.h.  a == b
```

otherwise steht (als Schreibabkürzung) für die Konjunktion der Negationen aller vorheriger Bedingungen

Grundlegende Konzepte

Funktionale Programmierung

Definitionen und **Auswertung** durch den Computer

Programmieren: Konstruieren einer **Funktion**, um ein Problem zu lösen

Computer: "Rechner", Auswerter von Ausdrücken
(wie beim Taschenrechner)

neu: Programmierer kann eigene Definitionen von Funktionen machen

Definitionen werden als **Vereinfachungs**- (Reduktions-) **Regeln** verwendet, bevor die Darstellung ("Druck" am Bildschirm) erfolgt

wenn eine Ausdruck einen wohldefinierten Wert besitzt, dann spielt die **Reihenfolge** der Auswertung seiner Bestandteile keine Rolle

Die Bedeutung (**Semantik**) eines Ausdrucks ist sein Wert

"**Räsonieren**" (*reasoning*) wie in der Mathematik,
mit (mehr oder weniger bekannten) algebraischen Gesetzen a + b = b + a
(später: auch Einführung neuer Gesetze durch den Programmierer)

Skripte (Liste von Definitionen)

```
square x   = x * x
mini x y   | x <= y  = x                -- in Gofer/Haskell bereits definiert
           | x > y   = y
side  = 12
area  = square side
```

Gleichungen zwischen Ausdrücken mit **Variablen**, hier x, y

jede Definition führt eine Bindung ein: d. h. eine Zuordnung von einem Namen an einen Wert

square ist assoziiert mit der Funktion,
die ihr Argument *(Parameter)* quadriert

mini ist assoziiert mit der Funktion,
die das kleinere ihrer Argumente als Wert abliefert

eine Menge von Bindungen heißt **Kontext** oder **Umgebung**

Ausdrücke werden immer in einem Kontext ausgewertet,
einige Ausdrücke können in der vordefinierten "**Standardumgebung**" ausgewertet werden

Auswertung von Ausdrücken

```
?42↵
```

Antwort des Rechners: Wert; Zeilenvorschub; prompt

```
42↵
?

?6*7
42

?square (3 + 4)
49

?mini 3 4
3

?square (mini 3 4)


?area


?mini (area + 4) 150
```


Ausdrücke und Werte

Ausdrücke

einige, aber nicht alle, Schreibweisen aus der Mathematik

bezeichnen (oder beschreiben) einen **Wert**

die Bedeutung eines Ausdrucks **ist** sein Wert

keine Seiteneffekte oder dergl.

Die Arten der Werte eines Ausdrucks können sein:

Zahlen, Wahrheitswerte, Zeichen, Tupel, Listen, Funktionen, etc.

Variablen

Namen in Ausdrücken, die für unbekannte Größen stehen

verschiedene Auftreten desselben Namens ("Bezeichners") bezeichnen dieselbe Unbekannte

Variablen bezeichnen im selben Kontext *dieselbe* Größe
(auch: Variablenbelegung)

Bezeichner

Bezeichner (identifier) für Variablen

Groß/Kleinschreibung ist signifikant

Variablen- (bzw. Funktionsbezeichner), Typvariablen
müssen mit einem **Klein**buchstaben anfangen

Konstruktorbezeichner, Typbezeichner
müssen mit einem **Groß**buchstaben anfangen

danach dürfen Buchstaben oder Ziffern folgen, auch Unterstrich _ oder Apostroph ' (sollte aber nicht, s.u.)

bkb BKB ggT x x' dies_auch diesAuch a3 bool True

reservierte Worte dürfen nicht als benutzerdefinierte Worte vorkommen

vordefinierte Bezeichner

Bool Char Int String True False ...

abs cos max min

Konstantenbezeichner / Literale

Numerale

```
12    4.5e13   24.6    0.63e-6
```

Zeichenliterale

```
'a'    '0'    '\n'
```

Literale für Zeichenketten

```
"hello dolly"    ""    "\n\n\n"    "a"
```

Bedeutung einiger Zeichen (wie in C)

\n *neue Zeile;* \t *Tabulator;* \f *neue Seite;*
\r *Zeilenrücklauf;* \b *Rückzeichen;* \\ *backslash selbst;*
\' *Apostroph;* \" *Doppelapostroph;* \ddd *3 Dezimalziffern ASCII - ISO*

Haskell Text

*Folge von **Symbolen** (tokens) getrennt durch Satzspiegel (layout)*

Satzspiegel: Leerzeichen, Zeilenvorschub, Tabulator und Kommentare wird normalerweise ignoriert

```
area      = square side    -- dies ist ein Kommentar
```

ein Satzspiegelzeichen muß oft zur eindeutigen Trennung vorkommen, z.B.

```
f' x 'x'    -- nicht dasselbe wie f' x' x'
```

wäre ohne Zwischenräume ein einziger Bezeichner

Definitionszeilen werden im allgemeinen durch Zeilenvorschub beendet

Auswertung, Vereinfachung

```
square x    = x * x
```

der Rechner "führt einen Ausdruck auf seine einfachste Form zurück" ($\Rightarrow$ heißt "wird ausgewertet zu", "reduziert zu")

square (3 + 4) $\Rightarrow$ *square 7* -- *(+)*
$\Rightarrow$ *7 * 7* -- *(square)*
$\Rightarrow$ *49* -- *(*)*

(+) bezieht sich auf eine Verwendung der eingebauten Regel für +, ebenso *(*)* für *; *(square)* auf die Definition des Programmierers.

die obige Auswertungssequenz ist nicht die einzige mögliche:

square (3 + 4) $\Rightarrow$ *(3 + 4) * (3 + 4)* -- *(square)*
$\Rightarrow$ *7 * (3 + 4)* -- *(+)*
$\Rightarrow$ *7 * 7* -- *(+)*
$\Rightarrow$ *49* -- *(*)*

in dieser Auswertungssequenz wird die Regel für square zuerst angewendet, das Ergebnis ist aber dasselbe (die Auswertungs*strategie* ist nichtdeterministisch, aber für "wohlgeformte" Regeln determiniert)

Auswertung durch Termersetzung

```
square x    = x * x
```

Auswertung von Ausdrücken

durch einen einfachen Prozeß der *Regelanwendung*:

square (3 + 4)

Substitution (Ersetzung von Variablen durch Ausdrücke),

*square (3 + 4) = (3 + 4) * (3 + 4)*

ggf. Überprüfung der Anwendbarkeit: *Auswertung der Bedingungen* und, falls anwendbar (d.h. Bedingung ist true)

Reduktion (Ersetzung der linken durch die rechte Seite einer Regel)

square (3 + 4) $\Rightarrow$ *(3 + 4) * (3 + 4)*

dieser Prozeß heißt auch *Termersetzung*

Darstellung und Wert

die Darstellung, Repräsentation eines Ausdrucks

kann vielerlei sein (z.B. für eine ganze Zahl):

- der Ausdruck selbst `7 * 7`
- eine Dezimalzahl `49`
- eine römische Zahl **III IV XLIX**
- eine Folge von Strichen (Kerbholzdarstellung)
  IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
- eine Folge von **0** und **1** (Binärdarstellung) **00110001**

der Wert eines Ausdrucks

ist abstrakt

jeder Wert ist nur in einer bestimmten Darstellung verfügbar

Normalform

ein Ausdruck ist in Normalform (kanonischer Form)

wenn er nicht weiter reduziert werden kann

die Normalform ist abhängig

von der Syntax der (Programmier)sprache für Ausdrücke

von der genauen Definition (zulässiger) Ersetzungsregeln

der Wert eines Ausdruck wird in seiner Normalform dargestellt

manche Ausdrücke haben keine endliche Normalform

Beispiel: π hat keine endliche dezimale Darstellung

manche Werte haben keine Normalform

Beispiel: der Werte eines Ausdrucks, dessen Berechnung unendlich lange dauert

```
unendlich = unendlich
```


manche Ausdrücke können nicht reduziert werden

keine Regel ist anwendbar

sie bezeichnen keine wohl-definierten Werte im normalen mathematischen Sinn

Beispiel: `1/0`

eine Aufforderung zur Auswertung kann ergeben, daß

- der Ausdruck `1/0` als Darstellung des Ergebnisses stehen bleibt
- die Auswertung nicht terminiert (das Ergebnis ist Schweigen)
- eine Fehlermeldung als Ergebnis gegeben wird, (Haskell)
  z. B. `attempt to divide by zero`

den undefinierten Wert

wollen wir (nur auf dem Papier!) schreiben als $\perp$ ("bottom")

er steht für einen Spezialwert (ähnlich ∞)

den Wert einer unendlichen Sequenz von Berechnungen ohne Ergebnis

den Wert eines unzulässigen Ausdrucks ("Fehler"), z. B. `1/0`

Struktur eines Ausdrucks (Terms)

Funktionsschreibweise (bindet am stärksten)

```
mini 3 5
```

Infixschreibweise für vordefinierte Operationen

```
2 + ( 4 * 6 )        3 < 5        3 < 5 && 5 <= 7
```

Klammerfreie Schreibweise

als Verabredung bei assoziativen Operationen: $(A \bullet B) \bullet C = A \bullet (B \bullet C)$

```
2 + 5 + 7     aber:  3 – 5 – 2  heißt  (3 – 5) – 2
```

Prioritäten

als Verabredung (z. B. "Punkt vor Strich")

```
2 + 4 * 6          26 < 3 + 5 – 7  heißt  26 < ((3 + 5) – 7)
```

Klammer"gebirge" (vollgeklammerte Schreibweise)

```
( ( ( 2 + ( 4 * 6 ) ) < ( ( 3 + 5 ) – 7 ) ) )
```

Konstruktoren

ein Konstruktor (auch: erzeugende Funktion)

ist ein spezieller Bezeichner (mit großem Anfangsbuchstaben), oder im Spezialfall ein Literal

Beispiel: Konstruktoren für Natürliche Zahlen $\mathbb{N}$

```
0                    _ + 1          -- Nachfolgerfunktion, successor
_+1+1   _+1+1+1   _+1+1+1+1   -- Abkürzungen in Haskell: +2   +3   +4
```

Beispiel: Konstruktor für Rationale Zahlen $\mathbb{Q}$

```
Rat  2  3
```

Beispiel: Konstruktor für Koordinaten

```
Coord  3  1      --plusCoord (Coord 3 1) (Coord 3 1)
```

Konstruktorterme

ein Konstruktorterm

ist ein Ausdruck, der vollständig aus Konstruktoren besteht

```
0   0+1   (0+5)+4   1   Coord (0+5) (17)   Rat  2  3   Rat  4  6
```

Gegenbeispiele: ('*', '–' sind keine Konstruktoren, '+' nur als '_+1')

```
0-1   3+(5+4)   Coord (3+5) (6–2)   plusRat  (Rat 2 (2+1))  (Rat 3 2)
```

ein Term über den Standardoperationen

wird durch Auswertung immer in einen Konstruktorterm überführt sofern er wohl-definiert ist (Gegenbeispiel 1/0)

für ganze Zahlen $\mathbb{Z}$:
Konstruktoren sind 0, _+1, –1 bzw. Abkürzungen mit Literalen _+5, –6

Boole'sche Werte (Wahrheitswerte) $\mathbb{B}$:
Konstruktoren sind True, False

```
5   [Abkürzung]                 als Wert von   (0+1)+4
False                           als Wert von   2 + 4 * 6 < 3 + 5 – 7
```


Funktionen und Definitionen

Funktion

Abbildung von Definitionsbereich a in Wertebereich b

f :: a –> b *f :: Int –> Char*

Zuordnung: jedem Element aus a genau ein Element aus b

Funktionsanwendung

Argumente aus a, Ergebnisse aus b

wenn x aus a, dann bezeichnet

f (x) oder kurz f x (Juxtaposition)

das Ergebnis der Anwendung von der Funktion f auf x

die Klammern sind nicht notwendig

[aber Vorsicht: g (x, y) ist nicht dasselbe wie g' x y]

Funktionen sind (auch) Werte

Unterschied zwischen einer Funktion (als Wert) und dem Ergebnis einer Anwendung (oft mehrdeutig "Funktionswert" genannt)

f ist etwas anderes als f x

Funktion und Definition(en)

Unterschied zwischen einer Funktion und einer bestimmten Definition (z.B. als Skript)

es gibt viele mögliche Wege ein und dieselbe Funktion zu definieren

```
double  x  = x + x
double' x  = 2 * x
```

verschiedene Rechenvorschriften (eine vielleicht "effizienter" als die andere), aber mathematisch dieselbe Funktion:

double = double'

[aber diese Gleichheit kann nicht in Haskell überprüft werden, sie ist nicht als Operation auf Funktionen verfügbar]

Formen der Definition

linke Seite mit Variablen für Parameter

rechte Seite Ausdruck

```
square x = x * x
```

rechte Seite Ausdruck, Bedingung ("Wächter", "guard"); Reihenfolge!

```
mini x y  | x ≤ y   = x
          | x > y   = y
```

Abkürzung der Bedingung bei Fallunterscheidung ("alle übrigen...")

```
mini x y  | x ≤ y          = x
          | otherwise      = y
```

Regel des gleichmäßigen Einrückens ("offside rule") beachten!!

linke Seite mit Konstruktortermen (mit Variablen) für Parameter

ganz vorne steht die zu definierende Funktion

nullstellige Konstruktoren; implizite Fallunterscheidung

```
und True  True   = True
und True  False= False
```

mehrstellige Konstruktoren; implizite Fallunterscheidung

```
factorial    0    = 1
factorial  (n + 1) =  (n+1) * factorial  n
```

mehrstellige Konstruktoren; Aufspaltung in Bestandteile

```
plusCoord  (Coord  x1  y1)  (Coord  x2  y2)   = Coord  (x1+x2)  (y1+y2)
```

die Formen können auch gemischt werden

```
multSkalar  (Coord  x  y)  s  = Coord  (x*s)  (y*s)
```

```
factorial  0              = 1                                  -- factorial  m  | m == 0   =   1
factorial  m  | m > 0  =    m * factorial  (m–1)
```


Typen

	Basistypen	*Werte*
Zahlen	*Int*	0, 1, -1
Wahrheitswerte	*Bool*	True, False
Zeichen	*Char*	'a', 'Z'
	Zusammengesetzte Typen	*Werte*
Tupel (z. B. Paare)	*(Int, Bool)*	(0, True)
Listen	*[Char]*	['d', 'o', 'l', 'l', 'y'], "dolly"
Funktionen	*Int –> Int*	factorial
	Bool –> Bool –> Bool	&&

jeder Typ hat bestimmte zugehörige Operationen (Funktionen),
die für andere Typen nicht anwendbar sind
z.b. kann man nicht eine Zahl zu einem Zeichen addieren,
oder zwei Funktionen miteinander multiplizieren

jeder einfache Bestandteil eines Ausdrucks hat einen eindeutigen Typ:
jede Variable, jeder Konstruktor, jedes Literal

jede Funktion hat einen eindeutigen Typ des Arguments
(Definitionsbereich) und des Werts (Wertebereich)

Argumentwerte dürfen nur von diesem Typ sein;
die Funktion darf nur Werte des Ergebnistyps abliefern

der Typ einer Funktion kann durch eine Typdefinition (zusätzlich, d.h.
zur Kontrollüberprüfung) festgelegt werden:

factorial :: Int –> Int -- *factorial* is of type *Int –> Int*

in Definitionen kann der Typ einer gebundenen Variablen (Parameter)
gelegentlich offen bleiben, oder er wird durch eine Typdefinition der
Funktion festgelegt; es erfolgt eine implizite *Typinferenz*

Typisierung

Berechnungen sind typgebunden, typerhaltend

strenge Typisierung

jeder Ausdruck hat einen Typ, der aus den Typen seiner Bestandteile hergeleitet werden kann

statische Typisierung

der Typ eines Ausdrucks ist aufgrund des Texts (ohne Eingabe) erkennbar und überprüfbar, er braucht nicht zur Auswertungszeit überprüft werden (außer für die Eingabe)

Typen von Funktionen

Deklaration des Typs

```
double :: Int -> Int        -- double is of type (Int -> Int)
```

automatische Herleitung aus der Definition (Typinferenz)

```
double x = 2 * x
```

Typvariablen, polymorphe Typen

was ist der Typ von id in:

```
id x = x
```

wir werden griechische Buchstaben verwenden (in Haskell a, b, c, ...)

id :: α -> α -- *id* :: (α -> α)

der Typ von x ist polymorph

ein Typausdruck mit einer Typvariablen bezeichnet einen polymorphen Typ

was ist der Typ von

```
id 3
```

die Typvariable α wird ersetzt durch einen konkreten Typ(ausdruck)
Substitution von Typausdrücken, ähnlich wie bei Auswertung

```
id 3          -- is of type Int
```

was ist der Typ von

```
id double
```

die Typvariable α wird ersetzt durch den Typausdruck (*Int* -> *Int*)

```
id double     -- is of type (Int -> Int)
```

was ist der Typ von

```
id id
```

die Typvariable α wird ersetzt durch den Typausdruck (β -> β)

```
id id         -- is of type ( β -> β )
```

was ist der Typ von

```
c x = 'A'        -- c :: (α -> Char)
```


Typanalyse

- wir ordnen der linken und der rechten Seite einer Definition dieselbe (neue) Typvariable zu
- für jede bekannte Funktion geben wir den bereits definierten Typ an
- wir wenden die folgenden Regeln an

Funktionsanwendung

f x :: α genau dann wenn *x* :: β und *f* :: β -> α
(wobei α bzw β evtl. eine neue Typvariable ist)

Gleichheit

x :: α und *x* :: β genau dann wenn $\alpha = \beta$

Gleichheit von Funktionstypen (analog für Tupeltypen)

$\alpha \to \beta = \alpha' \to \beta'$ genau dann wenn $\alpha = \alpha'$ und $\beta = \beta'$

- wir lösen alle entstehenden Gleichungen auf

Typanalyse

```
factorial n | n == 0 =  1
factorial n | n > 0  =  n * factorial (n–1)
```

Funktionsdefinition

factorial n :: α => *1 :: α* und *n * factorial (n–1) :: α*

Funktionsanwendung

factorial n :: α => *n :: β* *factorial :: β –> α*
factorial (n–1) :: α => *n–1 :: β*

1 :: Int => *primSubInt :: (Int, Int) –> In* => *n – 1 :: Int* und *n :: Int*

Gleichheit

n–1 :: β und *n – 1 :: Int* => *α = Int*
n :: β und *n :: Int* => *β = Int*

Substitution der Gleichheit

factorial :: β –> α und *α = Int* und *β = Int* => *factorial :: Int –> Int*

Basistypen Ganze Zahlen

Bindungen, Präzedenzen der Basis-Operatoren

schwach

> >= == /= <= <	ggf. *klammern!*
+ –	*bindet nach links*
–	*klammern!*
*	*bindet nach links*
/ `div` `mod` `rem`	ggf. *klammern!*
^	*bindet nach rechts*

stark

einige Gesetze

Halbgruppe bzgl. *+* mit neutralem Element *0*

(a + b) + c = a + (b + c) -- *Assoziativgesetz*
0 + c = c; *c + 0 = c* -- *neutrales Element*

Halbgruppe bzgl. *** mit neutralem Element *1*

*(a * b) * c = a * (b * c)* -- *Assoziativgesetz*
*1 * c = c;* *c * 1 = c* -- *neutrales Element*

Ring bzgl. *+, **

*a * (b + c) = a * b + a * c* -- *Distributivgesetz*
*(a + b) * c = a * c + b * c* -- *Distributivgesetz*

Existenz eines Inversen Elements *x* bzgl. *+*

x + b = 0; *b + x = 0* -- *inverses Element ist (0–b): Ganze Zahlen*

Unäres und Binäres Minus

(–b) + b = 0; *b + (–b) = 0*
a – b = a + (0 – b)

Existenz eines Inversen Elements *x* bzgl. ***

*x * b = 1;* *b * x = 1* -- *inverses Element ist (1/b): Rationale Zahlen*

Division und Modulo (ganzzahlig!)

*a = b * (a `div` b) + a `mod` b,* *b /= 0*
0 ≤ a `mod` b && a `mod` b < b, *b >0*
b < a `mod` b && a `mod` b ≤ 0, *b <0*

Gruppe bzgl. *** *(modulo a)*

jede Gleichung der folgenden Form hat eine eindeutige Lösung für *x*

*x * a = b;* *a * x = b*

abelsche Gruppe bzgl. ***

*a * b = b * a* -- *Kommutativgesetz*

Boole'sche Werte

Bindungen, Präzedenzen der Basis-Operatoren

schwach

||	*bindet nach rechts*
&&	*bindet nach rechts*
not	*(Präfix), bindet schwächer als Vergleiche!*
> >= = /= <= <	gegf. *klammern! (bzw. ausschreiben)*

stark

einige Gesetze

Halbgruppe bzgl. || mit neutralem Element *False*

```
(a || b) || c = a || (b || c)          -- Assoziativgesetz
False || c = c;     c || False = c     -- neutrales Element
```

Halbgruppe bzgl. *&&* mit neutralem Element *True*

```
(a && b) && c = a && (b && c)          -- Assoziativgesetz
True && c = c;     c && True = c       -- neutrales Element
```

Ring bzgl. ||, *&&*

```
a && (b || c) = a && b || a && c       -- Distributivgesetz
(a || b) && c = a && c || b && c       -- Distributivgesetz
```

Existenz eines Inversen Elements *x* bzgl. ||

```
x || b = True;          b || x = True  -- inverses Element ist (not b)
```

Existenz eines Inversen Elements *x* bzgl. *&&*

```
x && b = False;         b && x = False -- inverses Element ist (not b)
```


Boole'sche Ausdrücke und Funktionen Schaltjahr

```
leap y | y `mod` 100 == 0    = (y `mod` 400 = 0)
       | otherwise           = (y `mod` 4 == 0)
```

```
leap y = (y `mod` 4 = 0) && (y `mod` 100 /= 0 || y `mod` 400 = 0)
```

Umformung

not (a && b) = (not a || not b); not (a || b) = (not a && not b)
-- *deMorgan'sche Gesetze*

```
leap y = (y `mod` 4 = 0) && not (y `mod` 100 == 0 && y `mod` 400 /= 0)
```

```
leap y = not not (y `mod` 4 == 0) && not (y `mod` 100 == 0 && y `mod` 400 /= 0)
leap y = not ( not (y `mod` 4 == 0) ||       (y `mod` 100 == 0 && y `mod` 400 /= 0) )
```


Dreiecksanalyse

```
dreieck a b c        | a <= b && b <= c                  = analysiere a b c

analysiere a b c     | a + b <= c                        = 0
                     | a = b && b = c                   = 1
                     | a /= c && (a = b || b = c)       = 2
                     | (a < b && b < c) && (a + b <= c) = 3
```

```
analysiere a b c     | not (echtesdreieck a b c)         = 0
                     | gleichseitig a b c               = 1
                     | gleichschenklig a b c            = 2
                     | ungleichseitig a b c             = 3
```

```
echtesdreieck a b c   = a + b > c
gleichseitig a b c    = a = b && b = c
gleichschenklig a b c = a /= c && (a = b || b = c)
ungleichseitig a b c  = (a < b && b < c) && echtesdreieck a b c
```

Vorsicht: dreieck etc. sind nur teilweise definiert (partielle Funktionen)!

Funktionen mit Ergebnistyp Bool heißen auch Prädikate

Zeichen

Basis-Operatoren

Char –> Int

ord

Int –> Char

chr -- chr (ord x) = x

```
leCh x y = ord x ≤ ord y
isdigit x = '0' ≤ x && x ≤ '9'
isupper x = 'A' ≤ x && x ≤ 'Z'
islower x = 'a' ≤ x && x ≤ 'z'
isletter x= isupper x || islower x
```

```
capitalise x | islower x = chr ((ord 'A' – ord 'a') + ord x)
             | otherwise = x
```


Listen

Konstruktoren (auf der linken Seite in Definitionen verwendbar)

: -- α –> [α] –> [α] gesprochen: cons

[] -- [α]

spezielle Abkürzungen für Konstruktorterme (nur in Ausdrücken)

[a, b] = a : [b] = a : (b : []) = a : b : []

Basis-Operatoren

++	-- [α] –> [α] –> [α]	Konkatenation
\\	-- [α] –> [α] –> [α]	Differenz
length	-- [α] –> Int	Länge
!!	-- [α] –> Int –> α	Selektion des n-ten Elements, von 0 ab

schwache Bindung

: ++ *bindet nach rechts* \\ *klammern!*

!! *bindet nach links*

Zeichenreihen

Spezialfall für Liste von Zeichen

String == [Char]

alle Operationen auf Listen sind anwendbar

spezielle Abkürzungen für Konstruktorterme (nur in Ausdrücken)

```
"hi" = ['h', 'i'] = 'h' : 'i' : [] = 'h' : ('i' : [])
```

```
spaces 0 = ""                    -- partiell definiert für ℕ
spaces (1 + n) = " " ++ spaces n -- oder: ' ' : spaces n
```

```
length [] = 0                    -- vordefiniert !!
length (x : xs) = 1 + length xs
```

```
ljustify n x| n >= length x = x ++ spaces (n – length x)   -- vordefiniert !!
```

```
rjustify n x | n >= length x = spaces (n – length x) ++ x   -- vordefiniert !!
```

```
cjustify n x | n >= length x = leftHalf (n – length x) ++ x ++ rightHalf (n – length x)
leftHalf k = spaces (k `div` 2)
rightHalf k= spaces (k `div` 2 + k `mod` 2)
```


Funktionen auf Listen

```
revcons xs x = xs ++ [x]                    -- revcons :: [α] –> α –> [α]  (nicht vordefiniert)
```

```
head (x : xs) = x                           -- head :: [α] –> α
head [] = error "head of []"
```

```
tail (x : xs) = xs                          -- tail :: [α] –> [α]
tail [] = error "tail of []"
```

```
init (x : []) = []                          -- init :: [α] –> [α]
init (x : xs) | xs /= [] = x : init xs      -- xs = y : ys
init [] = error "init of []"
```

```
last (x : []) = x                           -- head :: [α] –> α
last (x : xs) | xs /= [] = last xs          -- xs = b : y
last [] = error "last of []"
```

einige Gesetze

Halbgruppe bzgl. ++ mit neutralem Element *[]*

(a ++ b) ++ c = a ++ (b ++ c) -- *Assoziativgesetz*

[] ++ c = c; *c ++ [] = c* -- *neutrales Element*

Selektoren bzgl. ++ bzw. *:* und *revcons*

x : xs = [x] ++ xs	*revcons xs x = xs ++ [x]*
head ([x] ++ xs) = x	*init (xs ++ [x]) = xs*
tail ([x] ++ xs) = xs	*last (xs ++ [x]) = x*
([hd xs] ++ tail xs) = xs	*(init xs ++ [last xs])= xs* -- *wenn xs nicht []*
(head xs : tail xs) = xs	*revcons (init xs) (last xs) = xs* -- *wenn xs nicht []*


```
max [a]                          = a                  -- max :: [α] –> α
max [a, b]   | a >= b            = a
max [a, b]   | a < b             = b
max (a : xs) | xs == [b, c] ++ ys = max [a, max xs]
max []                           = error "max of []"
```

```
max [a]                          = a                  -- max :: [α] –> α
max (a : xs) | xs == b:c: ys     = maxi (a, max xs)
max []                           = error "max of []"
```


Sortieren durch Einsortieren

dasselbe Regelschema wie bei length

```
sort []          = []
sort (e : rest)  = insert e (sort rest)
```

```
insert x []                  = x : []                    -- [x]
insert x (e : rest) | x <= e = x : (e : rest)
                    | x > e = e : insert x rest
```

Verallgemeinerung für beliebige Ordnungsrelation kleinerGleich

```
sortGen le []          = []
sortGen le (e : rest)  = insertGen le e (sortGen le rest)
```

```
insertGen le x []                          =  [x]
insertGen le x (e : rest) |      le x e    =  x : (e : rest)
                          | not (le x e) =  e : insertGen le x rest
```

```
sort        = sortGen (<=)
```

```
min         = head • sort          -- min :: [α] –> α
minlist le  = head • sortGen le    -- minlist :: (α –> α –> Bool) –> [α] –> α
```


Sortieren durch Mischen

Definition nach ursprünglichem Algorithmus: beliebiges Aufspalten

aber diese Schreibweise ist <u>nicht in Haskell</u> zulässig,
da nur Konstruktoren auf der linken Seite erlaubt sind

```
msort []        = []
msort [e]       = [e]
msort (a ++ b)  = merge (msort a) (msort b)    -- nicht in Haskell; divide and conquer
```

lineare Zerlegung

```
msort []                          = []
msort (e : [])                    = [e]
msort (e : rest) | rest == a : as = merge (msort [e]) (msort rest),
```

vereinfacht vorausgesetzt merge [e] [] == [e]
)

```
msort []        = []
msort (e : rest) = merge [e] (msort rest)
```

geordnetes Mischen

```
merge a []                      = a
merge [] b                      = b
merge (x : xrest) (y : yrest) |      x <= y  =  [x] ++ merge xrest (y : yrest)
                              | not (x <= y) =  [y] ++ merge (x : xrest) yrest
```

Verallgemeinerung für beliebige Ordnungsrelation kleinerGleich

```
msortGen le []          = []
msortGen le (e : rest)  = mergeGen le [e] (msortGen le rest)
```

```
mergeGen le a []                      = a
mergeGen le [] b                      = b
mergeGen le (x : xrest) (y : yrest)|      le x y  =  [x] ++ mergeGen le xrest (y : yrest)
                                   | not (le x y) =  [y] ++ mergeGen le (x : xrest) yrest
```

```
msort = msortGen (<=)
```


Tupel

Paare

(3, 'a') *:: (Int, Char)* (3, ('a', False)) *:: (Int, (Char, Bool))*

(α, (β, γ)) *((α, β), γ)* *(α, β, γ)* sind verschiedene Typen

polymorphe Selektorfunktionen:

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

```
fst3 (x, y, z) = x              -- fst3 ::
snd3 (x, y, z) = y              -- snd3 ::
thd3 (x, y, z) = z              -- thd3 ::
```

```
quotrem (x, y) = (x `div` y, x `rem` y)
```


Neue Typen Aufzählungstypen

Typdefinition eines neuen Typs mit Alternativen

neue Werte sind angegeben durch Konstruktoren

```
data Tag = Mo | Di | Mi | Do | Fr | Sa | So
```

Vergleichsoperationen sind nicht vordefiniert

Konstruktoren können, wie üblich, in linken Seiten von Funktionsdefinitionen verwendet werden

```
tagWert Mo = 0;   tagWert Di = 1;   tagWert Mi = 2;   tagWert Do = 3
tagWert Fr = 4;   tagWert Sa = 5;   tagWert So = 6
```

wertTag analog, so daß gilt: *wertTag (tagWert t) = t*

```
folgeTag t      = wertTag ((tagWert t + 1) `mod` 7)
x <== y         = tagWert x <= tagWert y
werkTag t       = (Mo <== t) && (t <== Sa)    -- werkTag ::     Tag –> Bool
wochenTag t     = (Mo <== t) && (t <== Fr)    -- wochenTag ::   Tag –> Bool
wochenEnde t    = (Sa <== t) && (t <== So)    -- wochenEnde ::  Tag –> Bool
```


Beispiele für Aufzählungstypen

Spiel

```
data Zug = Papier | Stein | Schere
```

```
schlaegt Papier = Schere
schlaegt Stein  = Papier
schlaegt Schere = Stein
```

Richtungen

```
data HimmelsRichtung = Nord | Ost | Sued | West
```

```
data Richtung =  Hoch | Runter
```

Wahrheitwerte und Zeichen

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

```
data Char = Ascii0 | Ascii1 | … | Ascii127
```

zusammengesetzte neue Typen

Konstruktoren sind Injektionsfunktionen aus anderen Typen

Vereinigung verschiedener Werte

```
data Wert = Num Int | Bool Bool    -- Num :: Int –> wert;   Bool :: Bool –> wert
```

Selektoren sind Projektionsfunktionen in anderen Typen

```
numwert (Num n)= n          -- numwert ::   Wert –> Int
boolwert (Bool b)  = b      -- boolwert ::   Wert –> Bool
```

Prädikate als Diskriminatoren

```
isNum (Num n) = True
isNum (Bool b)   = False
```

Konstruktoren sind nicht-strikte Funktionen
(dazu später: unendliche Objekte)

```
      isNum ⊥  = ⊥          -- undefined
isNum (Num ⊥)  = True
isNum (Bool ⊥) = False
```


zusammengesetzte neue Typen

disjunkte Vereinigung

Unterscheidung durch Konstruktorbezeichungen

```
data Temp = Celsius Float | Fahrenheit Float | Kelvin Float
```

Vorsicht: Gleichheit auf Float ist nicht die gewünschte

abstrakte Typen: eine eigene Gleichheitsoperation muß definiert werden

```
=== :: Temp –> Temp –> Bool
toCelsius (Fahrenheit f) = (Celsius ((f–32.0)*5.0/9.0) )
(Celsius c) === (Fahrenheit f)  =  c == toCelsius f
```

neue Typen ohne Vereinigung (einfaches karthesisches Produkt)

```
data Temp = Temp Art Float
data Art     = Celsius | Fahrenheit | Kelvin
```


neuer Typ: Schutz gegen unerwünschte Operationen

alle Operationen müssen explizit neu definiert werden

```
data Datei = Datei [Eintrag]
entferne e (Datei es) = Datei (es \\ [e])   -- entferne :: Eintrag –> Datei –> Datei
                                            -- fuegeein :: Eintrag –> Datei –> Datei
                                            -- mische ::    Datei –> Datei –> Datei
                                            -- leer ::      Datei
```

Synonymdefinition eines Typs

```
type Datei = [Eintrag]
```

kein Schutz, alle Operationen auf Listen sind automatisch anwendbar

polymorphe neue Typen

```
data Datei α = Datei [α]
```

der obige Dateityp ist dann (Datei Eintrag)

```
data Set α   = Set [α]
```

Paartyp

```
data Pair α β = Pair α β
```

isomorph zu vordefiniertem Typkonstruktor (α, β)

```
     Pair 3 True ::
      Pair [3] 0 ::
Pair (Pair 1 2) 3 ::
```

allgemeine Form für ”algebraische Typen”

```
data T α β … κ = C1 T11 … T1k1 | … | Cn Tn1 … Tnkn
data Person α = Person [Char] Int α
```

rekursive algebraische Typen

```
data NNat = Zero | Succ NNat

plus n Zero      = n                          -- plus :: NNat -> NNat -> NNat
plus n (Succ m)  = Succ (plus n m)

fib Zero         = Zero                       -- fib :: NNat -> NNat
fib (Succ Zero)  = Succ Zero
fib (Succ (Succ n)) = fib (Succ n) `plus` fib n
```

liefert die folgende Definition die ganzen Zahlen?

```
data IInt = Zero | Succ IInt | Pred IInt
```

gilt die folgende Gleichung?

```
Succ (Pred Zero) = Pred (Succ Zero) = Zero
```

andere Möglichkeit:

```
data IInt2 = Pos NNat | Neg NNat
```

was ist mit der Null? **Grenzen** der Nutzbarkeit "algebraischer" Typen

Listen, Bäume

Listen

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

beblätterte Bäume

```
infix 3 :^:
data Tree α = Leaf α | Tree α :^: Tree α

size (Leaf x)      = 1
size (t1 :^: t2)   = size t1 + size t2

depth (Leaf x)     = 0
depth (t1 :^: t2)  = 1 + maximum [depth t1, depth t2]
```

Bäume mit Knoteninformation (Kantorovic Bäume)

```
data Ntree α     = Null      | Node (Ntree α) α (Ntree α)
data Baum α β    = Blatt α|  Ast (Baum α β) β (Baum α β)
```

(Abstrakter) Syntax-Baum für arithmetische Ausdrücke

```
data Exp = Num Float | Exp Exp Op Exp          --Baum Float Op
data Op  = Add | Sub | Mul | Div | …
```


Abstrakter Datentyp: Sicht des Benutzers

Signatur = Typname
+ Namen und Typen *("Profile") der* ***Operationen***

```
type
  Store = …                          -- (versteckte) Typ-Repräsentation ("Implementierung")
  in
  initial ::  Store,                                  -- weitere Signatur
  lookup ::  Store –> Var –> Int,                     -- weitere Signatur
  update ::  Store –> Var –> Int –> Store             -- weitere Signatur
…                                    -- (versteckter) Rumpf mit Definitionen, evtl. mit Hilfsfunktionen
```

der Benutzer sieht nur die Signatur

"Geheimnisprinzip" *(information hiding)* :
Verstecken der Implementierung

Schnittstelle *(interface)* zwischen Benutzer und Implementierer

erlaubt ist z.B. (für **type** Var = String)

```
lookup (update (update initial "V0" 15) "V1" 27) "V0"
```


Abstrakter Datentyp: Sicht des Implementierers

Typname *+ (versteckte) Typ-Repräsentation ("Implementierung")*
+ ***Namen und Typen*** *der Operationen*
+ (versteckter) ***Rumpf*** *mit Definitionen der Operationen*

```
type
  Store = [ (Int, Var) ]             -- (versteckte) Typ-Repräsentation ("Implementierung")
  in
  initial ::  Store,                                  -- weitere Signatur
  lookup ::  Store –> Var –> Int,                     -- weitere Signatur
  update ::  Store –> Var –> Int –> Store             -- weitere Signatur
initial = []                         -- (versteckter) Rumpf mit Definitionen, evtl. mit Hilfsfunktionen
lookup [] v = error "leerer Specher"
lookup ( (n, w) : st )  v
  | v==w        = n
  | otherwise   = lookup st v
update st v n = (n,v) : st
```

die Typ-Repräsentation ist nur in Operationen der Signatur sichtbar

z.B. length initial führt auf Fehler

Abstrakter Datentyp: alternative Implementierung

```
type
  Store  = Var –> Int                 -- (versteckte) Typ-Repräsentation ("Implementierung")
  in
  initial ::   Store,                                          -- weitere Signatur
  lookup ::  Store –> Var –> Int,                              -- weitere Signatur
  update ::  Store –> Var –> Int –> Store                      -- weitere Signatur
initial  v = error "leerer Specher"                 -- (versteckter) Rumpf mit Definitionen
lookup  st  v  =  st  v
update  st  v  n  w
  |  v==w        = n
  |  otherwise   = st  w
```


Abstrakter Datentyp: als ausführlicher Modul in Haskell

nur die Namen der beabsichtigten Signatur sind in der Exportliste (z.B. nicht MkStore)

genaue Kontrolle der Sichtbarkeit; Hilfsfunktionen sind möglich

Vorsicht: Namen müssen verschieden sein, z.B. Store, MkStore

```
module STOREs  ( Store, initial, lookup, update )  where
  type Var     = String              -- verstecktes Typsynonym, aber sichtbare rechte Seite
  data Store  = MkStore [ (Int, Var) ] -- sichtbarer Typname, versteckte Typ-Repräsentation
  initial ::    Store
  initial = MkStore []
  lookup ::  Store –> Var –> Int
  lookup  (MkStore [])  v = error "leerer Speicher"
  lookup  (MkStore (n, w) : st )  v
    | v==w        = n
    | otherwise   = lookup  (MkStore  st)  v
  update ::  Store –> Var –> Int –> Store
  update  (MkStore  st)  v  n  = MkStore ((n,v) : st)
```


Stapel, "Keller"

```
type
  Stack τ = [τ]
  in
  emptyS ::    Stack τ,
  isEmptyS ::  Stack τ –> Bool,
  push ::      τ –> Stack τ –> Stack τ,
  top ::       Stack τ –> τ,
  pop ::       Stack τ –>Stack τ
emptyS         = []
isEmptyS  []   = True
isEmptyS  _    = False
push  a  s     = a : s
top   (a:s)    = a
pop   (a:s)    = s
```


Schlange

```
type
  Queue τ = [τ]
  in
  emptyQ ::    Queue τ,
  isEmptyQ ::  Queue τ –> Bool,
  addQ ::      τ –> Queue τ –> Queue τ,
  remQ ::      Queue τ –>  (τ, Queue τ)
emptyQ         = []
isEmptyQ  []   = True
isEmptyQ  _    = False
addQ  a  q     =  q++[a]
remQ  q
  |  not (isEmptyQ q)  =  (head  x, tail  x)
  |  otherwise         =  error "remQ"
```

```
addQ  a  q     =  (a:q)                    -- alternative Implementierung
remQ  q
  |  not (isEmptyQ q)  =  (last  x, init  x)
  |  otherwise         =  error "remQ"
```

Schlange

```
type
  Queue τ = ( [τ], [τ] )                    -- alternative Implementierung mit zwei Listen
  in
  emptyQ ::    Queue τ,
  isEmptyQ ::  Queue τ –> Bool,
  addQ ::      τ –> Queue τ –> Queue τ,
  remQ ::      Queue τ –>  (τ, Queue τ)
emptyQ            = ( [], [] )
isEmptyQ  ( [], [] ) = True
isEmptyQ  _       = False
addQ  a  (l, r)   = (l, a:r)
remQ  (a:l, r)    = (a, (l, r) )
remQ  ( [], [] )    = error "remQ"
remQ  ( [],  r )    = remQ (reverse  r, [] )
```


Signaturdiagramme

Typ Boolean

Operationsbezeichner and

Funktionstyp Boolean

Boolean –> Boolean –> Boolean • >Boolean

Boolean

not and

• •

true •

Boolean • or

false •

Benutzerdefinierte Signaturen in Haskell

```
module BOOLEANs ( Boolean (True, False), not, and, or )  where
  data  Boolean  =  True | False        -- Konstruktor-Operationen werden bekannt gemacht
  not::         Boolean –> Boolean
  and, or::     (Boolean, Boolean) –> Boolean
```

```
  not  False = True
  not  True  = False
  ...
```

```
module NATURALs  ( Natural (Zero, Succ), isZero, pred, plus, eqNat )  where
  import  BOOLEANs
  data  Natural  =  Zero | Succ Natural
  isZero:   Natural –> Boolean
  pred:     Natural –> Natural
  plus:     Natural –> Natural –> Natural
  eqNat:    Natural –> Natural –> Boolean
```

```
  isZero  Zero      = True     ;     plus  n  Zero      = n
  isZero (Succ  x)  = False    ;     plus  n  (Succ  m) = Succ  (plus  n  m)
  pred (Succ  x)    = x        ;     ...
```


(Unbeschränkte) Ganze Zahlen als Abstrakter Typ in Haskell

```
module INTEGERs  (Integer, zero, isZero, isPos, isNeg, succ, pred, neg, plus, minus, eqInt)
  where
  import  BOOLEANs, NATURALs
  type Integer =  ...                   -- versteckte Darstellung , keine Konstruktoren
  zero:                  Integer
  isZero, isPos, isNeg:  Integer –> Boolean
  succ, pred, neg:       Integer –> Integer
  plus, minus:           Integer –> Integer –> Integer
  eqInt:                 Integer –> Integer –> Boolean
```

abstrakte ”Konstruktoren”, ”Selektoren”, Vergleichsoperatoren

Darstellung Ganzer Zahlen

als Term

pred(pred(zero()))

als Paar Natürlicher Zahlen: (Haben, Soll); Problem: z.B. Null-Abfrage

paar(3, 5)

als Natürliche Zahl mit Vorzeichen: 0 für positiv, 1 für negativ; Null

paar(1, 2)

als Bitsequenz (Liste von O, L); Problem: führende Nullen

[L, L, O]

Ganze Zahlen als Paare Natürlicher Zahlen

Werte dargestellt als Paare Natürlicher Zahlen

z —> (a, b)

Äquivalenzklassen

es gilt für beliebiges c

z = (a, b) = (a+c, b+c)

graphisch:

2 = (2, 0) = (3, 1) =

-1 = (,) = (,) =

jede Zahl wird durch den (Repräsentanten einer) Äquivalenzklassen dargestellt

Normalisierung der Darstellung

Äquivalenzklasse

Menge äquivalenter Werte, Repräsentanten

Normalisierung:

in jeder Äquivalenzklasse wird ein Wert als Normalform ausgezeichnet, er ist der Repräsentant der Äquivalenzklasse

alle anderen Werte der Äquivalenzklasse können immer auf diesen Wert abgebildet, d. h. "normalisiert" werden

(a, b) = (a' +c, b' +c)

(a, b) —> (a', b')

Normalisierung so, daß c maximal ist, d. h. a' = 0 oder b' = 0

(a, b) —> (a–min(a, b), b–min(a, b))

Typklassen

Länge als polymorphe Funktion

```
length :: [α] -> Int ;   length []  =  0 ; length (x : xs)  = 1 + length xs
```

gleiche Operation für verschiedene Typen

Gleichheit als polymorphe Funktion

```
(==), (/=) :: α -> α -> Bool
```

verschiedene Operationen für verschiedene Typen, überlagert (z.B. bei Rationalen Zahlen, als Paar dargestellt)

die Typen der Operationen einer Typklasse erhalten einen Kontext K α =>

im Geltungsbereich eines Kontextes dürfen für α nur Exemplare von K eingesetzt werden

```
(==), (/=) :: Eq α => α -> α -> Bool
```

für alle Exemplare T von Eq (d.h. wenn eine Gleicheit auf T definiert ist)
haben (==) und (/=) den Typ *T –> T –> Bool*

Anwendung einer Typklasse

monomorphe Funktion

```
allEqual :: Int –> Int –> Int –> Bool
allEqual a b c = (a==b) && (b==c)
```

aber nichts ist daran speziell für ganze Zahlen

verallgemeinerte polymorphe Funktion

```
allEqual :: Eq τ => τ –> τ –> τ –> Bool
allEqual a b c = (a==b) && (b==c)
```

allgemeiner Typ einer Funktion

```
member [] b = False
member (a:x) b = (a==b) || member x b
```

die Anfrage : type member liefert member :: Eq a => [a] –> a –> Bool

Definition einer Typklasse

```
class Eq α where
  (==), (/=) :: α –> α –> Bool
  x /= y = not (x == y)
```

Exemplardeklaration, Instanz

```
instance Eq Int where
  (==) = primEqInt
```

der Typ(konstruktor) Int ist ein Exemplar der Typklasse Eq

die Operation primEqInt ist eine Implementierung der Gleichheit auf Int

die Definition der Ungleicheit ist abgeleitet und lautet ausgeschrieben

```
x /= y = not (primEqInt x y)
```

Exemplardeklaration mit expliziter Definition von ==

```
instance Eq Bool where
  True == True = True
  False == False = True;        _ == _ = False
```


Exemplardeklaration für Typkonstruktoren mit Parametern

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

Elemente können verglichen werden => Listen können vergl. werden

Einschränkungen an zwei Typvariablen für Paare

```
instance (Eq α, Eq β) => Eq (α, β) where
  (x, y) == (u, v) = x==u && y==v
```

falls a und b Exemplare der Typklasse Eq sind,
so ist auch (a, b) ein Exemplar der Typklasse Eq

Spezialgleichheit für Funktionen mit endlichem Definitionsbereich, z.B.

```
instance Eq α => Eq (Bool –> α) where
  f == g = f True == g True && f False == g False
```

i.a. ist Gleichheit auf Funktionen unentscheidbar

Unterklassen

```
class Eq α => Ord α where
  (<), (<=), (>), (>=):: α -> α -> Bool
  max, min             :: α -> α -> α
--  x <= x                      Reflexivität
--  x <= y ∧ y <= z  → x <= z   Transitivität
--  x <= y ∧ y <= x  → x == y   Antisymmetrie      Halbordnung
  x < y  = x <= y && x /= y          ;   max x y  | x >= y = x
  x >= y = y <= x                    ;     max x y  | x <= y = y
  x > y  = y < x                     ;   min x y  | x <= y = x
                                         min x y  | x >= y = y
```

Ord ist eine Unterklasse von Eq, Ord erbt die Operationen von Eq

```
iSort :: Ord τ => [τ] -> [τ]
iSort []       = []
iSort (e : rest) = insert e (iSort rest)

insert x []                  = x : []                    --[x]
insert x (e : rest) | x <= e = x : (e : rest)
                    | x > e  = e : insert x rest
```


Exemplardeklarationen

```
instance Ord Int where (<=) = primLeInt
```

```
instance (Ord α, Ord β) => Ord (α, β) where
  (x1, x2) <= (y1, y2) = x1 < y1 || (x1==y1 && x2<=y2)      -- lineare Ordnung
```

```
instance Ord α => Ord [α] where
  []    <= _   = True
  (_:_) <= []  = False
  (x:xs) <= (y:ys) = x < y || (x == y && xs <= ys)          -- lexikographische Ordnung
```

```
class Ord τ => Enum τ where
  enumFrom     :: τ -> [τ]          -- [n .. ]
  enumFromThen :: τ -> τ -> [τ]     -- [n, m .. ]
  enumFromTo   :: τ -> τ -> [τ]     -- [n .. m]
  enumFromThenTo :: τ -> τ -> τ -> [τ]   -- [n1, n2 .. m]
```


Arithmetische Operationen

```
class Eq α => Num α where      -- simplified numeric class
  (+), (–), (*), (/) :: α -> α -> α
  negate             :: α -> α
  fromInteger        :: Int -> α
```

für Ganze, Gleitpunkt-, Rationale, Komplexe, Gauß'sche Zahlen, Vektoren, Matrizen, Polynome etc.

```
sqr :: Num α => α -> α   ;   sqr x = x * x
```

vordefinierte Instantiierung

```
instance Num Int where
  (+)    = primPlusInt   ;  (–)        = primMinusInt
  (*)    = primMulInt    ;  (/)        = primDivInt
  negate = primNegInt    ;  fromInteger x = x
```

in Haskell zusätzlich vordefiniert:

Integer	beliebig große natürliche Zahlen
Float, Double, Rational, Complex	(nur Float in Gofer)

eigene Instantiierung

```
infix 5 :+
data Complex = Float :+ Float        -- alternativ: Gauß = Int :+ Int
```

```
instance Eq Complex where
  x :+ ix == y :+ iy = x == y && ix == iy
```

analog Ord

```
instance Num Complex where
  (x :+ ix) + (y :+ iy) = x + y :+ ix + iy
  (x :+ ix) – (y :+ iy) = x – y :+ ix – iy
  (x :+ ix) * (y :+ iy) = x * y – ix * iy :+ x * iy + ix * y
  (x :+ ix) / (y :+ iy) = (x * y + ix * iy)/d :+ (ix * y – x * iy)/d
                          where d = y * y + iy * iy
  negate (x :+ ix)      = ((–x) :+ (–ix))
  fromInteger x         = (fromInteger x :+ 0.0)      -- Gauß: :+ 0
```

Mengen und Gleichheit

```
module SETs ( Set, eSet, singleton, union, elemS, subset ) where
  data Set α = MkSet [α]
  eSet ::              Set α
  singleton ::         α -> Set α
  union ::             Set α -> Set α -> Set α
  elemS :: Eq α =>     α -> Set α -> Bool
  subset :: Eq α =>    Set α -> Set α -> Bool
```

```
eSet                          = MkSet []
singleton x                   = MkSet [x]
union (MkSet xs) (MkSet ys)   = MkSet (xs ++ ys)
...
```

Gleichheit für Mengen

```
instance Eq α => Eq (Set α) where
  a == b = a `subset` b && b `subset` a
```


geordnete Bäume

```
module OrderedTREEs ( Tree, nil, isNil, isNode, leftSub, rightSub,
                      treeVal, insTree, delete, join, elemT ) where
  data Tree τ = Nil | Node τ (Tree τ) (Tree τ)
  nil ::                Tree τ
  isNil, isNode ::      Tree τ -> Bool
  leftSub, rightSub ::  Tree τ -> Tree τ
  treeVal ::            Tree τ -> τ
  insTree :: Ord τ =>   τ -> Tree τ -> Tree τ
  delete :: Ord τ =>    τ -> Tree τ -> Tree τ
  join :: Ord τ =>      Tree τ -> Tree τ -> Tree τ
  elemT :: Ord τ =>     τ -> Tree τ -> Bool
```

insTree erhält die Ordnung, Node nicht, ist daher versteckt

Benutzung außen

```
size :: Tree τ -> Int
size t
  | isNil t       = 0
  | otherwise= 1 + size (leftSub t) + size (rightSub t)
```


Implementierung im Rumpf

```
nil = Nil
...
treeVal Nil              = error "treeVal"
treeVal (Node v _ _) = v

insTree val Nil = (Node val Nil Nil)
insTree val (Node v t1 t2)
  | v==val    = Node v t1 t2
  | v < val   = Node v t1 (insTree val t2)
  | v > val   = Node v (insTree val t1) t2
delete val (Node v t1 t2)
  | v < val   = Node v t1 (delete val t2)
  | v > val   = Node v (delete val t1) t2
  | isNil t2  = t1
  | isNil t1  = t2
  | otherwise = join t1 t2
...
```

Gleichheit und Ordnungsrelationen werden benötigt

geordnete Mengen als geordnete Bäume

```
module OrderedSETs ( Set, eSet, singleton, union, elemS, subset ) where
  import OrderedTREEs
  type Set α = Tree α
  eSet ::              Set α
  singleton ::         α -> Set α
  union ::             Set α -> Set α -> Set α
  elemS :: Ord α =>    α -> Set α -> Bool
  subset :: Ord α =>   Set α -> Set α -> Bool
```

```
eSet          = nil
singleton x   = insTree x nil
union xs ys   = join xs ys
elemS x xs = elemT x xs
subset xs ys
  | isNil xs = True
  | isNil ys = False
  | x ´elemT´ ys = delete x xs ´subset´ delete x ys
    where x = treeVal xs
```

Partielle Parametrisierung ("Curry-ing")

Funktion mit Paar als Parameter

```
hoch' (a, b) = a ^ b        ; hoch' ::   (Int, Int) -> Int
```

Funktionsanwendung mit Paar

```
hoch' (3, 5)
```

Funktion mit Parameter und nachfolgendem Parameter (Funktional)

```
hoch a b   = a ^ b      ; hoch ::   Int -> Int -> Int
                                  = Int -> (Int -> Int)
```

Funktionsanwendung als schrittweise Parametrisierung

```
hoch 3 5          (hoch 3) 5
```

Unterschiede in der Funktionsanwendung

die Parameterisierung von hoch kann klammerfrei geschrieben werden (implizit von links geklammert)

der Parameter von hoch' muß als Paar geschrieben werden

Unterschiede der Funktionen

hoch' und hoch haben verschiedene Typen:

hoch' :: (Int, Int) -> Int

hoch' ist eine Funktion, die
wenn sie einen Wert vom Typ *(Int, Int)* als Argument hat,
einen Wert vom Typ *Int* als Ergebnis abliefert

hoch :: Int -> (Int -> Int)

hoch ist eine Funktion, die
wenn sie einen Wert vom Typ *Int* als Argument hat,
einen Wert vom Typ *(Int -> Int)* als Ergebnis abliefert,
also eine Funktion, die
wenn sie einen Wert vom Typ *Int* als Argument hat,
einen Wert vom Typ *Int* als Ergebnis abliefert

Funktionsanwendung mit partieller Parametrisierung

```
hoch 10
```

```
basis10 b = 10 ^ b        ; basis10 :: Int -> Int
```

```
basis10 3
```

```
basis10'    = hoch 10     ; basis10' :: Int -> Int
```

```
basis10' 3
```

```
hoch_ b a= a ^ b          ; hoch_ ::   Int -> (Int -> Int)
```

```
(hoch_ 2) 3
```

```
square      = hoch_ 2     ; square ::   Int -> Int
```

```
square 3        (hoch_ 2) 3
```

Ausschnitte – sections

```
(+ 1)   (1 +)   (mult 2)   (2 *)      (/ 2)    (2 /)
```

```
double    = (2 *)          -- Ergebnis: Funktion
```

```
double x  = (2 *) x        -- Ergebnis: Zahl
```


Funktionen als Parameter

Funktionen mit Funktionen als Parameter (Funktionale)

```
apply f x  = f x                -- apply ::      (α -> β) -> α -> β
```

```
apply square 3
```

```
twice f x  = f (f x)            -- twice ::      (α -> α) -> α -> α
```

```
twice square 3          twice (2 +)          twice (– 3)
```

```
compose f g x = f (g x)     -- compose ::(β -> γ) -> (α -> β) -> α -> γ
```

```
compose Succ Succ   compose (2 +) (– 2)
```

compose f g wird auch geschrieben f • g oder f . g

```
twice f    = f . f              -- twice ::      (α -> α) -> α -> α
```

```
both h x= h x x                 -- both ::       (α -> α -> β) -> α -> β
```

```
both (+ 3)          both (*)
twice square 3
```

Fallunterscheidung als Funktion

```
if_then_else_ True  jaFall neinFall = jaFall
if_then_else_ False jaFall neinFall = neinFall
```

if c then e1 else e2 ist vordefiniert

```
abs x = if_then_else_ (x<0) (–x) x
```

```
half x = if_then_else_ (x==0) 0 (x / 2)
```

```
intdiv x y = if_then_else_ (y == 0) (error "Division by 0 attempted") (x / y)
```

Vorsicht: if_then_else_ ist (nur) definiert,

bei Reduktion von innen nach außen ("call by value"), wenn alle Argumente definiert sind,

bei Reduktion von außen nach innen ("call by need, lazy evaluation"), wenn die Bedingung und das *entsprechende* Argument definiert sind,

Strikte und nicht-strikte Funktionen

eine strikte Funktion

ist undefiniert, wenn ein Argument undefiniert ist

x + ⊥ => ⊥ ⊥ + y => ⊥

eine nicht-strikte Funktion

kann auch definiert sein, wenn ein Argument undefiniert ist

Beispiel: if_then_else_ ist strikt im ersten, aber nicht-strikt im zweiten und dritten Argument

Auswertung

mit strikter Semantik (von innen nach außen):

```
intdiv 3 0 => if_then_else_ (3==0)  (error "...") (3 / 0)
           => if_then_else_ False   (error "...")  ⊥      => ⊥
```

mit nicht-strikter Semantik (von außen nach innen):

```
intdiv 3 0 => if_then_else_ (3==0)  (error "...")  (3 / 0)
           => if_then_else_ False   (error "...") (3 / 0)  =>  (error "...")
```


die Semantik von Haskell ist generell nicht-strikt

strikt bei "by value"

nicht-strikt bei "by need" (evtl. auch "lazy")

weitere nicht-strikte Funktionen

```
and_ x y  = if_then_else_ (x=False) False y
```

```
or_ True  y = True
or_ False y = y
```

Funktionen

als Werte sind immer definiert;
erst beim tatsächlichen Aufruf kann es undefinierte Werte geben

if_then_else_ bzw. if_then_else_ False sind definiert,
if_then_else_ False ⊥ nur bei nicht-strikter Semantik

Wiederholung als Funktional

```
until p f x | p x          = x
            | otherwise    = until p f (f x)
```

until :: (α –> Bool) –> (α –> α) –> α –> α

f wird wiederholt angewandt (iteriert), bis das Terminierungsprädikat p erfüllt ist

Analogie zur "while" bzw. "until" Schleife in imperativen Programmiersprachen

until (> 1000) (2 *) 1 = 1024

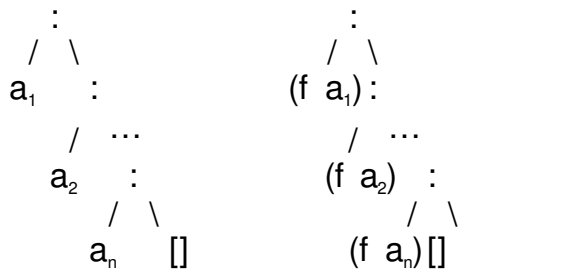
```
modulo a b | a >= 0 && b > 0 = until (< b) (– b) a          -- nur für natürliche Zahlen
```

```
modulo a b |  a < b && a >= 0 && b > 0 = a,                  -- nur für natürliche Zahlen
           | a >= b && a >= 0 && b > 0 = modulo (a–b) b
```

Map

erweitert eine Basisfunktion f homomorph (strukturerhaltend) über Listen

bildet eine Liste in eine andere ab, in der die Funktion f auf alle Elemente angewendet wurde




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



```
? map square [9, 3]
[81, 9]

? sum (map square [1 .. 10] )
385
```

```
sigma f m n = sum (map f [m .. n] )
```

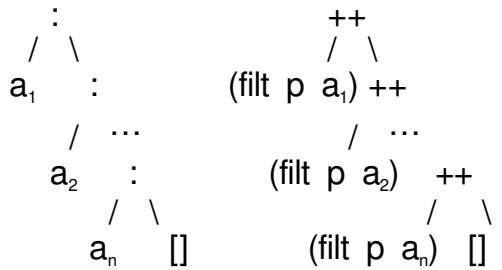
```
? sigma square 1 10
385

? map (3==) [1, 3, 5]
[False, True, False]
```

```
elem xs x = or (map (x==) xs)
```

```
? elem [1, 3, 5] 4
False
```


Filter

bildet ab in eine Liste, in der für jedes Element ein Prädikat p erfüllt ist




```
filter p []                  = []                  -- filter :: (α -> Bool) -> [α] -> [α]
filter p (x : xs) |     p x  = x : filter p xs
filter p (x : xs) | not p x  =     filter p xs
```

```
filt p x |     p x  = [x]
         | not p x  = []
```

```
filter p []       = []                           -- filter :: (α -> Bool) -> [α] -> [α]
filter p (x : xs) = (filt p x) ++ (filter p xs)
```



```
? filter even [1, 2, 4, 5, 32]
[2, 4, 32]

? (sum • map square • filter even) [1 .. 10]
220
```

Gesetze für höhere Funktionen

einige Gesetze für Funktionen

Funktionen bilden eine Halbgruppe bzgl. • mit neutralem Element *id*

(a • b) • c = a • (b • c) -- *Assoziativgesetz*

id • c = c; *c • id = c* -- *neutrales Element*

inverse Funktionen (bzgl. *id*)

$f \bullet f^{-1} = id$

einige Gesetze für map

map (f • g) = *(map f) • (map g)*

$map\ f^{-1} = (map\ f)^{-1}$

map f (xs ++ ys) = *(map f xs) ++ (map f ys)*

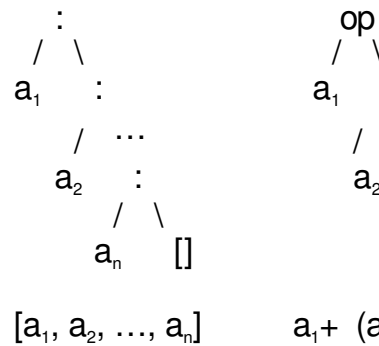
einige Gesetze für filter

filter p (xs ++ ys) = *(filter p xs) ++ (filter p ys)*

filter p • filter q = *filter q • filter p* -- *für q totale Funktion*

Foldr

bildet : homomorph ab in eine beliebige binäre Operation ***op***

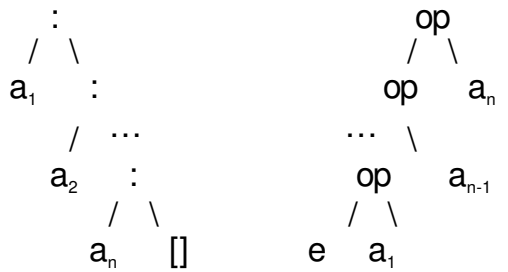


$[a_1, a_2, \ldots, a_n]$ $\quad a_1 + (a_2 + \ldots + (a_n + e) \ldots)$

```
foldr op e []     = e                    -- foldr :: (α -> β -> β) -> β -> [α] -> β
foldr op e (x:xs) = op x (foldr op e xs)
```

```
reverse = foldr postfix []          -- reverse :: [α] -> [α]
          where postfix x xs = xs ++ [x]
length  = foldr oneplus 0
          where oneplus x n = 1 + n
```


Foldl (nach Bird)



$[a_1, a_2, \ldots, a_n]$ $\quad (\ldots((e + a_1) + a_2) + \ldots + a_n)$

```
foldl op e []     = e                    -- foldl :: (β -> α -> β) -> β -> [α] -> β
foldl op r (x:xs) = foldl op (op r x) xs
foldl1 op (x:xs)  = foldl op x xs        -- analog die anderen
```

```
reverse = foldl prefix []          -- reverse :: [α] -> [α]
          where prefix xs x = [x] ++ xs
pack    = foldl times10 0          --
          where times10 n x = 10 * n + x
```


Fold mit Monoid

fold op e = foldl op e

Falls op und e ein Monoid (und für endliche Listen), gilt das Dualitätsaxiom

foldr op e = foldl op e

wir wollen in diesen Fällen schreiben *fold op e* (nicht vordefiniert)

im allgemeinen ist die Implementierung *fold = foldl* effizienter

Beispiele bei denen op und e eine Halbgruppe (Monoid) bilden

```
sum     = fold (+) 0      -- sum :: [Int] -> Int
product = fold (*) 1      -- product :: [Int] -> Int
concat  = fold (++) []    -- concat :: [[α]] -> [α]
and     = fold (&&) True  -- and :: [Bool] -> Bool
or      = fold (||) False -- or :: [Bool] -> Bool
dotlist = fold (•) id     -- dotlist :: [α -> α] -> α -> α
```

Gesetze für höhere Funktionen auf Listen

einige Gesetze für map und filter

```
f • concat          = concat • map f
map f • concat      = concat • map (map f)
filter p • concat   = concat • map (filter p)
filter p            = concat • map (filt p)
```

einige Gesetze für fold (mit Monoid)

```
fold op e (xs ++ ys)  =  op (fold op e xs) (fold op e ys)
```

einige Gesetze für foldr und foldl

```
foldl op e (xs ++ ys)  =  foldl op (foldl op e xs) ys
foldr op e (xs ++ ys)  =  foldr op (foldr op e xs) ys
```

Dualitätsaxiom, falls $x \oplus (y \otimes z) = (x \oplus y) \otimes z,\ x \oplus e = e \otimes x$

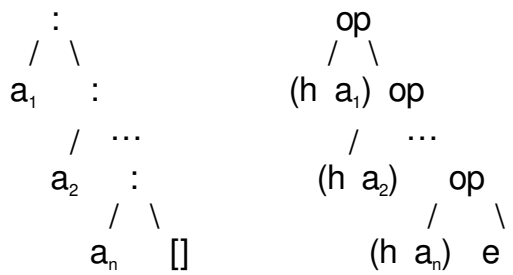
```
foldr ⊕ e xs  = foldl ⊗ e xs
```

Dualitätsaxiom, falls $x \circledR y = y \oplus x$

```
foldr ⊕ e xs  = foldl ® e (reverse xs)
```


Homomorphe Erweiterung

einer Basisfunktion h in eine beliebige algebraische Struktur mit einer binären Operation op und einem Element e




```
                                  -- homr :: (β -> γ -> γ) -> γ -> (α -> β) -> [α] -> γ
homr op e h []       = e
homr op e h (x : xs) = op (h x) (homr op e h xs)
```

```
 -- falls op, e Monoid,            hom :: (γ -> γ -> γ) -> γ -> (α -> γ) -> [α] -> γ
hom op e h []       = e
hom op r h (x : xs) = hom op (op r (h x)) h xs
```


Beispiele bei denen op nicht assoziativ ist

```
foldr op e      = homr op e id
```

```
length          = homr oneplus 0 id
oneplus x n     = 1 + n
```

Beispiele bei denen op und e eine Halbgruppe (Monoid) bilden

```
fold op e       = hom op e id
```

```
length          = hom (+) 0 one   where one x s = 1
```

```
sum             = hom (+) 0 id
```

```
map f           = hom (++) [] (single • f)   --  = homr (:) [] f
```

```
filter p        = hom (++) [] (filt p)
```

```
elem x xs       = or (map (x==) xs)
```

```
elem x xs       = hom (||) False (x==) xs
```


allgemeine Quantoren über Listen ("Mengen")

```
any p           = hom (||) False p      -- Existenzquantor, "es existiert ein x so daß p gilt"
```

```
all p           = hom (&&) True p       -- Allquantor, "für alle x gilt p"
```

```
elem x xs       = any (x==) xs
```

Darstellung von Mengen als Listen

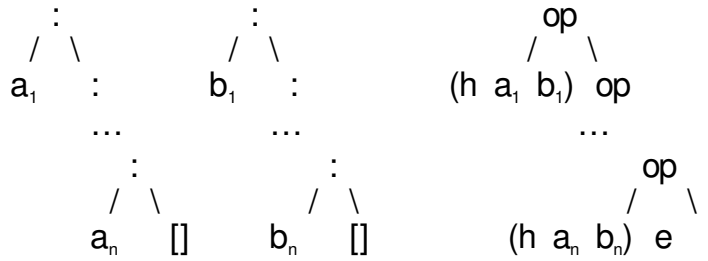
(a) Elemente sind in der Darstellung mehrfach repräsentiert, bei elem muß evtl. länger gesucht werden, Listen sind länger

```
union xs ys   = xs ++ ys
```

(b) Elemente sind in der Darstellung nur einfach repräsentiert, bei union müssen mehrfache Elemente ausgefiltert werden, Listen sind minimal lang

```
union xs ys   = xs ++ (hom (++) [] (filterElem xs) ys)
                where  filterElem xs x = filter (not • elem x) xs
```

(Pseudo-)Homomorphe Verknüpfung von Listenpaaren




```
-- pairhomr :: (γ -> δ -> δ) -> δ -> δ -> δ -> (α -> β -> γ) -> [α] -> [β] -> δ
pairhomr op e ex ey h   []     []  = e
pairhomr op e ex ey h   []   (y : ys) = ex
pairhomr op e ex ey h (x : xs)   []   = ey
pairhomr op e ex ey h (x : xs) (y : ys) = op (h x y) (pairhomr op e ex ey h xs ys)
```

```
-- op, e Monoid:   pairhom :: (γ -> γ -> γ) -> γ -> (α -> β -> γ) -> [α] -> [β] -> γ
pairhom op e h    []    ys     = e
pairhom op e h    xs    []     = e
pairhom op r  h  (x : xs) (y : ys) = pairhom op (op r (h x y)) h xs ys
```


Beispiele bei denen op und e ein Monoid bilden

```
zipWith h          = pairhom (++) [] (single • h)  -- (α -> β -> δ) -> [α] -> [β] -> [δ]
zipWith h          = pairhomr (:) [] [] []        h
```

```
pairor             = zipWith (||)                  -- [Bool] -> [Bool] -> [Bool]
```

```
pair x y           = (x, y)
zip                = zipWith pair                  -- [α] -> [β] -> [(α, β)]
zip                = pairhom (++) [] (single • pair)
```

Skalarprodukt $\sum x_i\ y_i$

```
skProd xs ys | length xs == length ys =  pairhom (+) 0 (*) xs ys
skProd xs ys | length xs == length ys =  sum (zipWith (*) xs ys)
```

nicht-abnehmende Sequenz $x_1 <= x_2$ && $x_2 <= x_3$ && … && $x_{n-1} <= x_n$

```
nondec []          = True
nondec xs| length xs > 0 = pairhom (&&) True lesseq (init xs) (tail xs)
```


alle Positionen, an denen x auftritt

```
positions xs x     = pairhom (++) [] (filtpos (x==)) (list 0 length xs-1) xs
                   = zipWith (filtpos (equal x)) (list 0 length xs-1) xs
```

```
filtpos p i y |    (p y)  = [i]
              | not (p y)  = []
```

```
position xs x = head ( positions xs x ++ [-1] )
```

Index (existiert schon als !!)

```
index xs n    = head ( pairhom (++) [] (filtval (n==)) (list 0 length xs-1) xs )
index xs n    = head ( zipWith (filtval (n==)) (list 0 length xs-1) xs )
```

```
filtval p i y |    (p i) = [y]
              | not (p i) = []
```


Listenkomprehension

eine Form der logischen Programmierung

Elemente aus einer gegebenen Liste, a<-ex ist der Generator

```
ex = [2, 4, 7]
```

```
[ 2*a | a<-ex ]          => [4, 8, 14]
```

```
[ isEven n |  n<-ex ]=> [True, True, False]
```

Spezifikation durch Prädikat, ”Tests”

```
[ 2*a | a<-ex,  isEven a,  a > 3 ]
```

```
digits st = [ ch  |  ch<-st, isDigit ch ]
```

Generator mit Muster

```
addPairs pairList = [ a+b  |  (a, b) <- pairList, a<b ]
```

```
books db borrower = [ bk  |  (person, bk) <- db,  person==borrower ]
```

```
books db borrower = [ bk  |  (borrower, bk) <- db ]      -- Vorsicht: neue Variable!!
```

```
pairs m n = [ (a, b) | a <- m, b <- n ]            -- Mehrfache Generatoren
```

```
triangle n = [ (a, b) | a <- [1..n], b <- [1..a] ]   -- erster fest, zweiter läuft, ...
```

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

```
pythagorTripel 15 => [ (3, 4, 5), (5, 12, 13), (6, 8, 10), (9, 12, 15) ]
```

```
perms [] = [[]]                                    -- perms :: Eq t => [t] -> [[t]]
perms x = [ a:p | a <- x, p <- perms (x \\ [a]) ]
```

```
perm2 []    = [[]]                                 -- perm2 ::          [t] -> [[t]]
perm2 (a:x) = [ p++[a]++q | r <- perm2 x, (p, q) <- splits r ]
```

```
splits :: [t] -> [ ( [t], [t] ) ]
splits []    = [ ( [], [] ) ]
splits (a:x) = ([],a:x) : [ (a:p, q) | (p, q) <- splits x ]
```

Übersetzung von Listenkomprehensionen

```
[ x | x<-xs ]              = xs
[ f x | x<-xs ]            = map f xs
[ e | x<-xs, p x, … ]      = [ e | x <- filter p xs, … ]
[ e | x<-xs, y<-ys, … ]    = concat [ [ e | y<-ys, … ], | x<-xs ]
```


Einige höhere Funktionen

Applikation (Auswertung), K und C Kombinatoren (Logik)

```
($)  :: (α -> β) -> α -> β
f $ x = f x
```

K und C Kombinatoren (Logik)

```
const :: α -> β -> α
const x y = x
```

```
flip  :: (α -> β -> γ) -> (β -> α -> γ)
flip f x y = f y x
```

Komposition, λ-Ausdrücke (Church)

```
(•)  :: (β -> γ) -> (α -> β) -> (α -> γ)
f • g = \x -> f (g x)
(••) :: (γ -> δ) -> (α -> β -> γ) -> α -> β -> δ
f •• g = \x y -> f (g x y)
```


Curry-ing (Logiker Curry und Schönfinkel)

```
curry :: ((α , β ) -> γ) -> α -> β -> γ
curry f = f2 where f2 x1 x2= f (x1, x2)
```

```
uncurry :: (α -> β -> γ) -> (α , β ) -> γ
uncurry f2 = f where f (x1, x2) = f2 x1 x2
```

uncurry (curry f) = f *curry (uncurry g) = g*

```
addlist :: [Int] -> [Int] -> [Int]
addlist = map (uncurry (+)) •• zip
```

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

```
unzip :: [(α , β )] -> ( [α], [β] )
unzip = foldr ( \(a, b) (as, bs) -> (a : as, b : bs) ) ([], [])
```

```
zipWith :: (α -> β -> γ) -> [α] -> [β] -> [γ]
zipWith f (x:xs) (y:ys) = f x y : zipwith f xs xy  -- (α -> β -> δ) -> [α] -> [β] -> [δ]
zipWith f _ _ = []
```


Vektoren, Matrizen

```
type Vector α = [α]
type Matrix α = [[α]]      -- [Vector α]   Liste von Zeilenvektoren
```

```
vecAdd :: Num v => Vector v -> Vector v -> Vector v
vecAdd = zipWith (+)
```

```
matAdd :: Num v => Matrix v -> Matrix v -> Matrix v
matAdd = zipWith (zipWith (+))
  --   = zipWith vecAdd
```

```
tr :: Num v => Matrix v -> Matrix v -> Matrix v   -- Transposition
tr []    = []
tr [v]   = map ( \x -> [x] ) v
tr (v:vs) = zipWith (:) v (tr vs)
```

```
vecMult :: Num v => Matrix v -> Vector v -> Vector v   -- Matrix mit Vektor
vecMult m v = map (skProd v) (tr m)
```

```
matMult :: Num v => Matrix v -> Matrix v -> Matrix v
matMult m1 m2 = map (vecMult m1) m2
```

Funktionale auf Bäumen

```
data Tree α = Node α [Tree α]
```

```
mapTree :: (α -> β) -> Tree α -> Tree β
mapTree f (Node x subtrees)
        = Node (f x) (map (mapTree f) subtrees)
```

```
foldTree :: (α -> [β] -> β) -> Tree α -> β
foldTree f (Node x subtrees)
        = f x (map (foldTree f) subtrees)
```

```
substitute :: Eq α => α -> α -> Tree α -> Tree α
substitute x y = mapTree ( \z -> if x == z then y else z )
```

```
sumTree :: Tree Int -> Int
sumTree = foldTree ( flip ((+) • sum) )
```

```
leaves :: Tree α -> Int
leaves = foldTree check
  where check _ []   = 1
        check _ sons = sum sons
```

```
nodes :: Tree α -> Int
nodes = foldTree checkN
  where checkN _ []   = 0
        checkN _ sons = 1 + sum sons
```



```
data Tree α = Node α [Tree α]
```

```
homTree :: (β -> δ -> γ) -> (γ -> δ -> δ) -> δ -> (α -> β) -> Tree α -> γ
homrTree opN op e h (Node x subtrees)
   = opN (h x) (homr op e (homrTree opN op e h) subtrees)
```

```
homTree :: (γ -> γ -> γ) -> γ -> (α -> γ) -> Tree α -> γ        -- op, e Monoid
homTree op e h (Node x subtrees)
   = hom op (h x) (homTree op e h) subtrees
```

```
mapTree :: (α -> β) -> Tree α -> Tree β
mapTree f = homrTree Node (:) [] f
```

```
foldrTree :: (α -> [β] -> β) -> Tree α -> β                     -- Version wie oben
foldrTree g = homrTree g (:) [] id
```

```
foldTree :: (α -> α -> α) -> Tree α -> α                        -- op, e Monoid
foldTree op e = homTree op e id
```

```
sumTree = foldTree (+) 0        allNodes= homTree (+) 0 (const 1)
```

```
height = foldrTree count where { count x [] = 0; count x strs = 1 + maximum strs }
```

```
toList = homTree (++) [] (\x->[x])  -- = foldrTree g where g x strs = x : concat strs
```


Mengen und Gleichheit

```
data Set α = Set [α]
```

```
eSet ::      Eq α =>  Set α
singleton :: Eq α =>  α -> Set α
union ::     Eq α =>  Set α -> Set α -> Set α
elemS ::     Eq α =>  α -> Set α -> Bool
anyS, allS :: Eq α =>  (α -> Bool) -> Set α -> Bool
homS ::      Eq α =>  (β -> β -> β) -> β -> (α -> β) -> Set α -> β -- Monoid!
mapS ::      Eq α =>  (α -> β) -> Set α -> Set β
foldS ::     Eq α =>  (α -> α -> α) -> α -> Set α -> α
```

```
eSet                = Set []
singleton x         = Set [x]
union (Set xs) (Set ys) = Set (xs ++ ys)
homS op e f (Set a) = Set (hom op e f a)
mapS f              = homS union eSet (singleton • f)
foldS op e          = homS op e id
anyS p              = homS (||) False p
allS p              = homS (&&) True p
x `elemS` xs        = anyS (x==) xs
```


Gleichheit für Mengen

```
instance Eq α => Eq (Set α) where
  a == b = a `subset` b && b `subset` a
    where a `subset` b = allS (`elemS` b) a
```

Mengen mit Gleichheit und Ordnung (Alternative)

```
instance Eq α => Ord (Set α) where
  a <= b = allS (`elemS` b) a
```

```
instance Ord (Set α) => Eq (Set α) where
  a == b = a <= b && b <= a
```

Textuelle Repräsentation

```
data BTree α = Leaf α | BTree α :^: BTree α
```

Datenelement in String verwandeln, erster Versuch

```
treeStr :: BTree String -> String
treeStr (Leaf str) = str
treeStr (lt :^: rt)    = "<" ++ treeStr lt ++ ":^:" ++ treeStr lt ++ ">"
```

die Konkatenation ist teuer, da die linken Seiten in geschachtelten Ausdrücken jedesmal neu duchlaufen werden müssen

Datenelement in String verwandeln, zweiter Versuch

```
type ShowS = String -> String

treeStr :: BTree String -> ShowS
treeStr (Leaf str) = (str ++)
treeStr (lt :^: rt)    = ('<' :) • (treeStr lt ++) • (":^:" ++) • (treeStr lt ++) • ('>' :)
```

der Kompositionsoperator erzwingt die Konstruktion in linearer Zeit


```
class Show α where
  showsPrec :: Int -> α -> ShowS
  showList :: [α] -> ShowS
  showList []     = showString "[]"
  showList (x: xs) = showChar '[' • shows x • showL xs • showChar ']'
                     where showL []     = id
                           showL (x: xs) = showChar ',' • shows x • showL xs
                          -- showL = hom (•) id (\x -> showChar ',' • shows x)

shows:: Show α => α -> ShowS    ;  shows     = showsPrec 0
show :: α -> String             ;  show      = shows x ""

showChar ::   Char -> ShowS     ;  showChar  = (:)

showString :: String -> ShowS   ;  showString = (++)
```

Priorität $0 \le d \le 10$ zur sparsam geklammerten Darstellung von Infixdatenkonstruktoren
showsPrec d x
hat als Wert eine Funktion, die den Show von x an den Anfang einer Zeichenkette schreibt und sie klammert, falls die Priorität des obersten Datenkonstruktors von x kleiner d


```
instance Show Int where
  showsPrec = primShowSInt

instance (Show α, Show β) => Show (α, β) where
  showsPrec p (x, y) = showChar '(' • shows x • showChar ',' • shows y • showChar ')'

instance Show α => Show (BTree α) where
  showsPrec p = showBTree where
  { showBTree (Leaf x) = shows x ;
    showBTree (lt :^: rt) = showChar '<' • showBTree lt • showString ":^:" •
                            showBTree rt • showChar '>' }

instance Show α => Show [α] where showsPrec p = showList

instance Show α => Show (Set α) where                    -- analog zu showList
  showsPrec p (Set [])     = showString "{}"
  showsPrec p (Set (x: xs)) = showChar '{' • shows x • showS xs • showChar '}'
            where showS = hom (•) id (\x -> showChar ',' • shows x)
```


Verzögerte Auswertung, Sharing

Strikte Grundoperationen

Musteranpassung

das anzupassende Argument muß soweit ausgewertet werden, daß entschieden werden kann, welches Muster vorliegt

length [1/0]

beim Muster x:xs wird weder x noch xs ausgewertet

Ausgabeanforderung

die interaktive Eingabe eines Ausdrucks bewirkt seine Auswertung, soweit sie für die Überführung des Ergebnisses in Textform nötig ist

Auswertungsprozeß stoppt beim Erreichen der Normalform

b	b :: Int || b :: Float || b :: Bool || b :: Char || b :: () -- *Basistypen*
\ x -> e	
f $e_1 \ldots e_k$	f ist n-stellige Funktion und $k < n$
C $e_1 \ldots e_k$	C ist Datenkonstruktor mit Stelligkeit $\ge k$
($e_1 \ldots e_k$)	$k \ge 2$

Unendliche Datenstrukturen

nicht-strikte Datenkonstruktoren

Element einer Liste wird erst dann berechnet, wenn es gebraucht wird

Rechnen mit unendlichen Objekten

sofern nur ein endlicher Teil während der Rechnung betrachtet wird

partieller Wert

noch nicht völlig berechneter Wert, z.B. potentiell unendliche Liste

Liste der natürlichen Zahlen ab n

```
countFrom :: Int -> [Int]
countFrom n = n : countFrom (n+1)
```

countFrom 2 ==>

spezielle Syntax in Haskell

[2 ..] ==>

[2, 4 ..] ==>

unendliche Objekte

Definition unabhängig von einer bestimmten Anwendung

```
powersOfTwo = 1 : map double powersOfTwo where double n = 2*n
```

```
twoToThe n = powersOfTwo !! n
```

Liste von Bits nach Dezimalzahl

```
binToDec ds = sum (zipWith (*) (reverse ds) powersOfTwo)
            = pairhom (+) 0 (*) (reverse ds) powersOfTwo
```

binToDec [1, 0, 1, 1, 0] ==> 22

Sieb des Eratosthenes (Primzahlen)

Algorithmus

1. Erstelle eine Liste aller natürlicher Zahlen beginnend mit 2
2. Markiere die erste unmarkierte Zahl
3. Streiche alle Vielfachen der letzten markierten Zahl
4. weiter bei Schritt 2

```
dropMultiples :: [Int] -> [Int]
dropMultiples (x:xs) = x : dropMultiples (filter (\ y -> y `mod` x /= 0) xs)
```

```
primes = dropMultiples [2 .. ]
```

primes ==> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29{Interrupted}

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

filter (< 30) primes ==> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29{Interrupted}

takeWhile (< 30) primes ==> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

Zirkuläre Datenstrukturen

```
ones :: [Int]
ones = 1 : ones
```

ones ==> 1: ones ==> 1: 1: ones ==> 1: 1: 1: … ==> [1, 1, 1, …]

im Prinzip unendlicher Speicherbedarf, aber zirkuläre Speicherstruktur

```
many :: α -> [α]
many x = x : many x          -- oder: manyZ x = z where z = x : z
```

many 1 ==> [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1{Interrupted}

Zirkuläre Datenstrukturen ("Fixpunkt") bei höheren Funktionen

```
iterate :: (α -> β) -> α -> [β]
iterate f x = x : map f (iterate f x)
```

```
iterate f x = itl where itl = x : map f itl -- optimiert zu dem obigen vom Haskell System
```

take 4 (iterate (+1) 12) ==> [12, 12+1, 12+1+1, 12+1+1+1]

beim Zugriff auf das n-te Listenelement wird f n-mal auf a angewandt

zirkuläre Speicherstruktur; Zwischenergebnisse werden wiederverwendet

Das Hamming-Problem

erzeuge eine Folge X mit den Eigenschaften

1. $x_{i+1} > x_i$ für alle natürlichen Zahlen i
2. $x_0 = 1$
3. Falls x in der Folge X auftritt, dann auch 2*x, 3*x und 5*x
4. Nur die durch 1., 2. und 3. spezifizierten Zahlen treten in X auf.

(Abschluß von S: kleinste Menge M mit $M \supseteq S$ und $\forall f \in F.\ M \supseteq f(M)$)

```
setMerge :: Ord α => [α] -> [α] -> [α]
setMerge sX@(x:xs) sY@(y:ys) | x == y  = x : setMerge xs ys
                             | x < y   = x : setMerge xs sY
                             | otherwise = y : setMerge sX ys
```

```
hamming = 1 : setMerge (map (*2) hamming)
                       (setMerge (map (*3) hamming)
                                 (map (*5) hamming) )
```

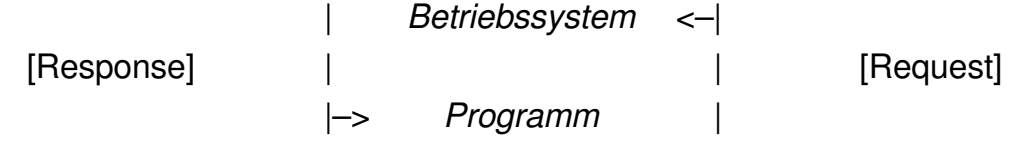
es muß möglich sein, das erste Element der zu verschmelzenden Listen zu bestimmen, ohne daß eine Schleife geschlossen wird

linear: neues Listenelement: < 4 Vergleiche, 3 (*), ein (:)

Strombasierte Ein- und Ausgabe

Ströme (unendliche Listen; nicht-strikte Auswertung)

das Betriebssystem verarbeitet einen Strom von Anfragen und erzeugt dabei einen Strom von Antworten




```
type Dialogue = [Response] -> [Request]
```

```
main :: Dialogue
main _ = [AppendChan stdout "Hello, world!\n"]
```

```
main2 ~(response: _) =                    -- nicht-zurückweisbares Muster der Form ~pat
  [ReadChan stdin, AppendChan stdout (hallo person)] where Str person = response
```

```
hallo "Peter" = "Guten Morgen!"
hallo "John"  = "good morning"
hallo person  = "Hallo " ++ person
```



```
data Request = ReadFile     String              -- file system requests
             | WriteFile    String String
             | AppendFile   String String
             | ReadChan     String              -- channel system requests
             | AppendChan   String String
             | Echo         Bool                -- environment requests
```

```
data Response = Success
              | Str String
              | Failure IOError
```

```
data IOError = WriteError  String
             | ReadError   String
             | SearchError String
             | FormatError String
             | OtherError  String
```

```
stdin  = "stdin"                                -- standard channels
stdout = "stdout"
stderr = "stderr"
stdecho = "stdecho"
```