Funktionale Programmierung

Prof. Dr. Ernst-Erich Doberkat
Lehrstuhl für Software-Technologie
Technische Universität Dortmund

WS 2012/2013

Inhalt

1 Literatur und Anderes

2 Erstes Beispiel

3 Paare und Listen

4 Module

5 Algebr. Datentypen

6 Ein- und Ausgabe

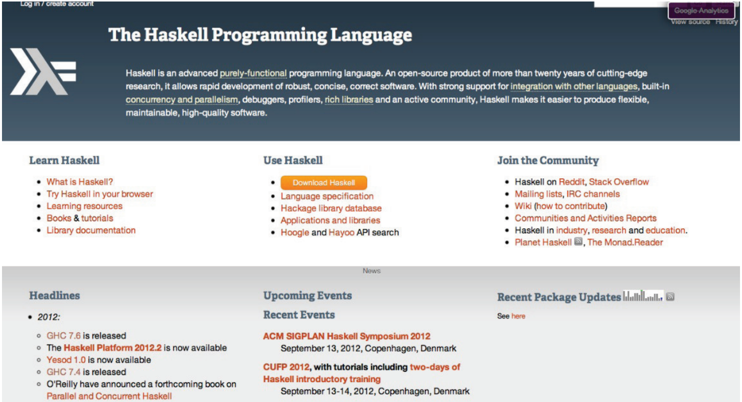
7 Monaden

Web Site

Das Buch hat diese web site: `http://haskellbuch.weebly.com`. Da finden Sie:

1. Lösungen zu den Übungsaufgaben,
2. Korrekturen und andere Hinweise,
3. Folien (die stehen natürlich auch im Wiki).

Wir verwenden ausschließlich den Interpreter aus Glasgow (`GHCi`). Das finden Sie unter `http://www.haskell.org`:



Klicken Sie `download` an, dann bekommen Sie (Adresse `http://www.haskell.org/platform/`)

Herzenslust

Hier können Sie sich nach Herzenslust bedienen. Ich gehe davon aus, daß Sie `GHCi` installiert haben. Manche Leute installieren auch den Haskell-Mode unter `emacs`.

Haskell ist eine funktionale Programmiersprache. Das bedeutet

- Funktionen stellen das wesentliche Hilfsmittel zur Strukturierung von Programmen dar.
- Funktionen werden wie mathematische Entitäten behandelt. Insbesondere werden Seiteneffekte durch den Einsatz von Funktionen vermieden.
  - Funktionen sind referentiell transparent, das bedeutet: die Reaktion auf einen Parameter ist stes dieselbe, unabhängig von der Umgebung, in der der Aufruf stattfindet.
  - Seiteneffekte sind dadurch unmöglich.
- Das bringt einige Probleme: Ein- und Ausgabe sind nicht möglich, wenn man sich auf einen strikten funktionalen Standpunkt stellt.
  - Hierzu benutzt Haskell Monaden, mit deren Hilfe Seiteneffekte gezielt verkapselt werden können.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

- Das Haskell-System hat einen Mechanismus zur Typ-Inferenz: Typen werden in der Regel inferiert.
- Haskell hat Klassen
  - Das ist ein wenig anders als in Java, wo man Definitionen von Methoden nur innerhalb von Klassen durchführen kann.
  - In Haskell muß eine Art Beitrittserklärung abgegeben werden.
- Haskell eignet sich zum (funktionalen, explorativen) Prototyping: man kann schnell Algorithmen implementieren und sie ausprobieren.
- Haskell ist knapp, knackig und ausdruckskräftig (und garnicht so schlimm, wie manche Leute denken).
- Die Sprache kann aber auch biestig sein, wenn man nicht aufpaßt. Sie werden sehen.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Erste Funktion

Diese Funktion addiert zwei Zahlen: `add a b = a + b`
Wir speichern diesen Text (ein Haskell-Skript) in einer Datei mit dem Namen `eins.hs` ab.

Jetzt sind mehrere Varianten möglich:

- Unter `emacs` können Sie im Haskell-Mode jetzt den Interpreter laden.
- Unter Windows können Sie jetzt den Interpreter aufrufen und das Skript laden.
- Unter Linux können Sie GHCi aufrufen und das Skript laden.

Erstes Beispiel

```
*Main> add 4 5.5
9.5
*Main> add 'a' 'b'

<interactive>:1:0:
    No instance for (Num Char)
      arising from a use of `add' at <interactive>:1:0-10
    Possible fix: add an instance declaration for (Num Char)
    In the expression: add 'a' 'b'
    In the definition of `it': it = add 'a' 'b'
*Main> :type add
add :: (Num a) => a -> a -> a
*Main> :quit
Leaving GHCi.
HP-L7780-2:Beispiele eed$
```

Na prima, das scheint ja zu gehen.

Mal sehen, was hier geschieht.

```
*Main> add 4 5.5
9.5
*Main> add 'a' 'b'

<interactive>:1:0:
    No instance for (Num Char)
      arising from a use of `add' at <interactive>:1:0-10
    Possible fix: add an instance declaration for (Num Char)
    In the expression: add 'a' 'b'
    In the definition of `it': it = add 'a' 'b'
*Main> :type add
add :: (Num a) => a -> a -> a
*Main> :quit
Leaving GHCi.
HP-L7780-2:Beispiele eed$
```

Au, weia!

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Offenbar ist der Aufruf `add 'a' 'b'` nicht so besonders sinnvoll, wenn unter `add` die Addition von Zahlen verstanden wird. Das erklärt die Fehlermeldung (aber nicht ihren Inhalt).

Typ?

Wir können uns nach dem Typ der Funktion beim Interpreter mit `:type` erkundigen und bekommen als Auskunft:

```
add :: (Num a) => a -> a -> a
```

Das sieht auch nicht so besonders hilfreich aus. Mal sehen.

`add ::` Das leitet die Typisierung ein und sagt, um welche Funktion es hier eigentlich geht.

`a` ist eine Typvariable, also ein Name, der Typen bezeichnen kann. `Num a` sagt, daß es sich hier um einen numerischen Typ handelt, also um ganze oder reelle Zahlen in ihren unterschiedlichen Varianten.

`(Num a) =>` sagt, daß die folgende Aussage nur für solche Typen gilt, die numerisch sind. Das kann man lesen als “Wenn `a` ein numerischer Typ ist, dann ... ”

Aber was dann?

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

`->`

Sind `a` und `b` Typen, so ist `a -> b` der Typ aller Funktionen von `a` nach `b`.

Beispiel

`Int` ist der Typ aller ganzen Zahlen, dann ist `Int -> Int` der Typ aller Funktionen, die ganzzahlige Argumente nehmen und ganzzahlige Werte liefern (z. B. ist die Funktion $x \mapsto x + 1$ vom Typ `Int -> Int`).

`A -> B -> C`

Also sind `a -> (b -> c)` alle Funktionen, die ein Argument vom Typ `a` nehmen und eine Funktion liefern, die ein Argument vom Typ `b` nimmt und ein Resultat vom Typ `c` liefert.

Beispiel

`b(x)` sei die Funktion, die `y` auf `x + y` abbildet, also `b(x)(y) = x + y`.

Ja, aber

Was ist der Unterschied zwischen `b(x)(y) = x + y` und `c(x, y) = x + y`?

1. die Funktion `b` hat einen einzigen Parameter vom Typ `Int` und liefert als Resultat ein Ergebnis vom Typ `Int -> Int`.
2. die Funktion `c` hat als Argument zwei Zahlen und liefert eine ganze Zahl.

In Haskell haben Funktionen grundsätzlich höchstens ein Argument (das vereinfacht und vereinheitlicht manches).

Funktionen, die mathematisch mehr als ein Argument haben, werden entsprechend als Funktionen aufgefaßt, die Funktionen als Werte liefern.

Curryfizierung

Das nennt man Curryfizierung (nach Haskell B. Curry, nicht nach dem Gewürz).

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Wir haben gesehen:

```
add :: (Num a) => a -> a -> a
```

Damit kann `add x y` interpretiert werden als `(add x) y`, wobei die Funktion `add x` gerade der Funktion $y \mapsto x + y$ entspricht.

Mal sehen

```
>>> :type (add 3)
(add 3) :: (Num a) => a -> a

>>> (add 3) 6
9
```

Well, that's not too bad.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Namen

Funktionen und Variablen werden grundsätzlich mit kleinen Anfangsbuchstaben geschrieben. Später werden wir sehen, daß z. B. Typnamen und die zugehörigen Konstruktoren mit einem Großbuchstaben beginnen.

Einrückungen

Wollen Sie eine Zeile fortsetzen, so beginnen Sie die nächste Zeile mit einigen Leerzeichen (nicht TAB). Das erspart Paare wie z.B. `{ ... }` oder `begin ... end`. Die Fehlermeldungen können in diesem Fall ziemlich kryptisch sein.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Binäre Operatoren

Binäre Operatoren können auch als Funktionen verwendet werden (müssen dann in Klammern geschrieben werden): `(+) 3 4` ist dasselbe wie `3 + 4`, `(*) 4 5` ist dasselbe wie `4 * 5`, etc.

Umgekehrt können Funktionen mit zwei Argumenten als Infix-Operatoren verwendet werden und müssen dann in `‘...‘` geschrieben werden, also z.B. `3 ‘add‘ 4`.

let

Mit `let` wird ein Name an einen Wert gebunden, der dann in der `GHCi`-Sitzung (bis zum nächsten `:load`) verwendet werden kann.

Obacht!

Das werden wir später verfeinern, aber im Augenblick reicht diese Erklärung zum Arbeiten.

```
>>> let r = (*) 3
>>> :t r
r :: Integer -> Integer
>>> r 4
12
>>> let m = (+) 7
>>> :t m
m :: Integer -> Integer
>>> r m 5

<interactive>:1:2:
    Couldn't match expected type `Integer'
           against inferred type `Integer -> Integer'
    In the first argument of `r', namely `m'
    In the expression: r m 5
    In the definition of `it': it = r m 5
>>> r (m 5)
36
```

Das sollten wir uns genauer ansehen.

Typ	Name	Beispiel
`Int`	ganze Zahlen	17
`Integer`	ganze Zahlen	123456789012345678
`Float`	reelle Zahlen	`2.1256363`
`Double`	reelle Zahlen	`2.1256363`
`Bool`	Wahrheitswerte	`True`
`Char`	Zeichen	`'a'`
`String`	Zeichenketten	`"JKL"`

Tabelle : Einige vordefinierte Typen

`Int` ist durch die Maschine beschränkt, `Integer` nicht. `Float` und `Double` sind die reellen Typen (einfache bzw. doppelte Genauigkeit).

Sonst müßte eigentlich alles klar sein.

Operator	Bedeutung	Assoziativität	Priorität
+	Addition	links	6
-	Subtraktion	links	6
*	Multiplikation	links	7
^	Exponentiation	rechts	8
/	Division	links	7
`div`	ganzzahlige Division	links	7
`mod`	Remainder	links	7
==	Test auf Gleichheit		4
&&	logisches Und	rechts	3
$	`f $ a == f a`	rechts	0
||	logisches Oder	rechts	2
Funktionsanwendung		links	10

Tabelle : Einige binäre Operatoren

Assoziativität

Der Operator $\copyright$ ist *links-assoziativ*, falls $X \copyright Y \copyright Z$ bedeutet $(X \copyright Y) \copyright Z$; analog: Rechts-Assoziativität.

Priorität: Regel

Je höher die Priorität, desto stärker bindet der Operator ("Punktrechnung geht vor Strichrechung"): $3 *_7 4 +_6 7$ = 19, $3 *_7 (4 +_6 7)$ = 33.

Die Funktionsanwendung bindet am stärksten, der Operator `$` bindet am schwächsten

Beispiel

`f(g(x))` kann ich schreiben als `f $ g x`: `g x` wird ausgewertet, das Ergebnis wird dann an `f` übergeben.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Beispiel: ++

Der Konkatenationsoperator ++:

```
>>> "abc" ++ "DEF"
"abcDEF"
>>> :info (++)

(++) :: [a] -> [a] -> [a]
     -- Defined in GHC.Base
infixr 5 ++
```

Also

Es handelt sich um einen Infix-Operator der Priorität 5, der rechts-assoziativ ist.

Das Beispiel zeigt, daß ++ zwei Zeichenketten miteinander konkateniert.

```
infix %
(%) x y = x ++ y ++ x
>>> :info %
(%) :: [a] -> [a] -> [a]
   -- Defined at ...
infix 9 %
```

```
>>> "abc" % "ABC"
"abcABCabc"
>>>  "abc" % "ABC" ++ "123"
"abcABCabc123"
```

Erläuterung

Die Typisierung sagt, daß unser Infix-Operator `%` die Priorität 9 hat, zwei Listen über dem Grundtyp `a` als Argument akzeptiert und eine Liste vom Typ `a` produziert.
Dazu sollte man wissen: Zeichenketten sind Listen vom Typ `Char`. Näheres später.

Namen

Selbstdefinierte Operatoren sollten mit diesen Zeichen gebildet werden

`! # $ % & * + . / < = > ? \ ^ | : - ~`

Dabei sind einige Kombinationen reserviert und können nicht benutzt werden

`.. : :: => = \ | ^ -> <-`

Diese Operatoren dürfen nicht mit einem Doppelpunkt beginnen. Grund: Konstruktoren, später.

Anonyme Funktionen

`add a b = a + b` ist eine Funktion mit dem Namen `add`. Manchmal ist es praktisch, wenn man einer Funktion keinen Namen geben muß (z. B. wenn die Funktion dynamisch erzeugt und gleich weiterverwendet wird).

```
(\x y -> x + y)
```

Diese Funktion leistet dasselbe wie `add`, hat aber keinen Namen (die Ärmste).

Also

`(\x y -> x + y)3 4`

ergibt 7

```
>>> :type (\x y -> x + y)
(\x y -> x + y) :: (Num a) => a -> a -> a
>>> let r = (\x y -> x + y)
>>> :type r
r :: Integer -> Integer -> Integer
```

Syntax

Nach \ (soll aussehen wie λ) folgt die Liste der Argumente, dann kommt der Pfeil -> und dann ein Ausdruck. Das Ganze steht in Klammern. Die Parameter werden in der Reihenfolge des Auftretens ausgewertet und eingesetzt.

Anonyme Funktionen

λ-Abstraktion

Anonyme Funktionen werden auch als λ-Abstraktionen bezeichnet (im λ-Kalkül werden diese Abstraktionen untersucht; das floß in die Konzeption von Haskell ein).

Beispiele

`(\x -> x * 3) 5`	Sollte klar sein. Wir werten die Funktion $x \mapsto x * 3$ an der Stelle 5 aus.

`(\f -> f 4)`	Diese Funktion nimmt eine Funktion als Argument und wertet sie an der Stelle 4 aus.

```
>>> :type (\f -> f 4)
(\f -> f 4) :: (Num t) => (t -> t1) -> t1
```

Was sagt uns das?

```
>>> :type (\f -> f 4)
(\f -> f 4) :: (Num t) => (t -> t1) -> t1
```

1. Das Argument für die Funktion `(\f -> f 4)` muß eine Funktion vom Typ `t -> t1` sein, wobei `t` ein numerischer Typ sein muß.
2. Das Resultat der Funktionsanwendung ist vom Typ `t1`, wobei wir über `t1` nichts wissen.

`(\f -> f 4)(\x -> x + 3)` Als Funktion, die ihr Argument an der Stelle 4 auswertet, wird die Funktion $x \mapsto x + 3$ übergeben. Beachten Sie:

1. Die Funktion `(\f -> f 4)` akzeptiert nur Funktionen als Argumente,
2. Beim Argument `(\x -> x + 3)` handelt es sich um eine solche Funktion.

Besonderheiten

Es gibt einige Besonderheiten bei der Verwendung der λ-Abstraktion.

1. Der Rumpf darf nur aus einem einzigen Ausdruck bestehen.
2. Eine anonyme Funktion darf nicht rekursiv sein
   - Das ist klar, denn für einen rekursiven Aufruf würde ein Funktionsname benötigt.
3. Die durch \ gebundenen Namen sind lokal und verschatten äußere Bindungen desselben Namens.

```
>>> (\x -> 3 * (\x -> 17 + x) x) 15
96
```

Auswertung als `3 * (\x -> 17 + x) 15` , daher das Resultat 96.

Fundamental

λ-Abstraktionen sind der fundamentale Mechanismus zur Behandlung von Funktionen in jeder funktionalen Programmiersprache, also auch in Haskell. Der λ-Kalkül ist das zugrundeliegende Berechnungsmodell.

Die direkte Umsetzung der rekursiven Definition

```
fakt1 n = if n == 0
             then 1
             else n * fakt1 (n-1)
```

Einrückung beachten!
Alternativ

```
fakt1 n = if n == 0 then 1 else n * fakt1 (n-1)
```

Mit

```
fakt1 n = if n == 0
then 1
else n * fakt1 (n-1)
```

fallen Sie auf die Nase.

Bedingter Ausdruck

Interessant: der bedingte Ausdruck

```
if Bedingung then Ausdruck1 else Ausdruck2
```

Die beiden Zweige der bedingten Anweisung erhalten jeweils Ausdrücke, deren Werte zurückgegeben werden, je nachdem ob die Bedingung zutrifft oder nicht.

Beide Ausdrücke müssen denselben Typ haben, weil sonst der Gesamtausdruck keinen einheitlichen Typ hätte.

Achtung

Der `else`-Teil der bedingten Anweisung darf nicht fehlen. Sonst wäre nicht klar, was als Wert zurückgegeben werden würde, falls die Bedingung falsch ist. Damit aber würden wir in einem undefinierten Zustand geraten, der Ausdruck hätte keinen Wert, und wir würden mit einer Fehlermeldung abbrechen müssen.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Die bewachte Lösung

```
fakt2 n
      | n == 0 = 1
      | True   = n * fakt2 (n-1)
```

(Einrückungen • Striche • `True` oder `otherwise`)

Hier setzen wir zwei Wächter ein:

- der erste Wächter überprüft das Argument `n` auf Gleichheit mit `0`,
- der andere Wächter heißt einfach `True`.

Die Wächter werden in der Reihenfolge ausgewertet, in der sie angegeben sind. Der *erste Wächter*, der den Wert `True` zurückgibt, bestimmt den Funktionswert.

Also

Der Wert `1` wird zurückgegeben, falls das Argument den Wert `0` hat. Sonst wird der Wert `True` ausgewertet (bekanntlich stets wahr), so daß der Wert `n * fakt2 (n-1)` zurückgegeben wird.

```
fakt3 0 = 1
fakt3 n = n * fakt3 (n-1)
```

Berechnung `fakt3 4`

Dann

1. wird zunächst der aktuelle Parameter 4 mit 0 verglichen,
2. da dieser Vergleich jedoch negativ ausgeht, wird die nächste Klausel herangezogen,
   - so daß also als Resultat für `fakt3 4` der Wert `4 * fakt3 3` zurückgegeben wird.
3. Dies wird so lange wiederholt, bis das Argument, das ja bei jedem Aufruf um 1 vermindert wird, den Wert 0 erreicht, dann wird der Wert 1 zurückgegeben, so daß der gesamte Ausdruck ausgewertet werden kann.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Alternative

Die Muster können in einen fallgesteuerten Ausdruck zusammengezogen werden, Schlüsselwort case.

Fallgesteuert

```
fakt3a n =
    case n of
      0          -> 1
      1          -> 1
      otherwise  -> n * (fakt3a (n-1))
```

Auch hier Vergleich des Argument `n` mit den angegebenen Fällen.

- für `n == 0` tritt der erste Fall ein,
- bei `n == 1` tritt der zweite ein,
- sonst wird der unter `otherwise` angegebene Ausdruck berechnet.

Der Fall `n == 1` ist natürlich überflüssig, ich habe ihn zur Illustration eingefügt.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Alternative Schreibweisen

```
fact3a n = case n of {0 -> 1; otherwise -> n * fact3a (n-1)}

fact3a n = case n of
     0 -> 1; 1 -> 1; otherwise -> n * fact3a (n-1)

fact3a n = case n of 0 -> 1;1 -> 1;otherwise -> n * fact3a (n-1)
```

Hierbei

Der Beginn einer neuen Zeile mit Einrückung wird von `Haskell` als Fortsetzung der vorhergehenden Zeile interpretiert.

- Man kann mehrere Ausdrücke in eine einzige Zeile schreiben, muß sie aber dann jeweils durch ein Semikolon voneinander trennen.
- Alternativ: jeder Ausdruck in seiner eigenen Zeile (Variante 2). Dann muß man durch Einrücken dafür sorgen, daß diese Zeile als Fortsetzung der vorigen Zeile verstanden wird.
- Die geschweiften Klammern halten einen Ausdruck zusammen und werden hier zur Gruppierung verwendet (Variante 1 vs. Variante 2).

```
fakt4 0 = 1
fakt4 n = let k = fakt4 (n-1)  in k * n
```

- Die Initialisierung ist klar.
- In der `let`-Umgebung wird der Ausdruck für `fakt4 (n-1)` dem lokalen Namen `k` zugewiesen.
- Der Wert wird anschließend, nach `in`, dazu verwendet, um den Ausdruck zu berechnen.

Zwischen den Schlüsselwörtern `let` und `in` können neue Namen einführt und an Ausdrücke gebunden werden.

Diese Bindungen werden im Anschluß an den Ausdruck, der auf `in` folgt, verwendet, also in die entsprechenden Ausdrücke eingesetzt.

Die gesamte Konstruktion `let ... in expr` bildet einen Ausdruck.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Die Namen in der `let`-Umgebung sind lokal, sind also beim Verlassen des gesamten Ausdrucks nicht mehr verfügbar.

`let`-Umgebungen können verschachtelt werden, dabei können Namen mehrfach verwendet werden (aber: Lesbarkeit?).

`bsp = let x = 11 in (let x = 2 in x * x) * x` ergibt 44, ist aber leicht verwirrend.

Mit `let` können wir in `GHCi` Namen an Ausdrücke binden. Dann fehlt der Teil, der durch `in` eingeleitet wird. Diese Bindung geht verloren, sobald innerhalb einer Sitzung eine Datei geladen wird.

Die lokalen Namen, die in einer lokalen `let`-Umgebung gebunden werden, verschatten die äußeren Namen, die in `GHCi` gebunden sind.

Eine *Typklasse* beschreibt das Verhalten eines Typs durch Angabe der Operationen, die auf ihm durchgeführt werden können.

Beispiele

- der Test auf Gleichheit,
- die Darstellung von Werten,
- das Lesen von Werten (als konverse Operation zur Darstellung),
- der Größenvergleich zwischen Elementen des Typs.

NB

Die Klassenorientierung in Haskell ist anders als in objektorientierten oder objektbasierten Sprachen.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

OO

In einer objektorientierten oder objektbasierten Sprache bestimmt die Klassenzugehörigkeit den Typ eines Objekts. Die Benutzung eines Objekts setzt meist die Erzeugung mit Hilfe eines Konstruktors voraus.

Etwas vergröbert

Eine Klasse also eine Kollektion von Werten an, gleichzeitig werden die legalen Operationen auf diesen Werten definiert, und Konstruktoren erzeugen die Instanzen der Klasse. Schließlich bestimmen Zugriffsspezifikationen, wie auf die einzelnen Komponenten einer Klasse zugegriffen werden kann.

In Haskell

Eine Typklasse spezifiziert eine Schnittstelle zu Operationen, die von allen Typen dieser Klasse benutzt werden können.

Vergleich mit Java

Vergleichbar mit einem Interface in Java.

Java Die Operationen auf den Instanzen einer Klasse werden zur Definitionszeit festgelegt und meist auch dort definiert. Ein vorhandenes Interface bindet die später hinzukommende implementierende Klasse.

Haskell Es ist möglich, zunächst einen Typ zu definieren und an anderer Stelle die Zugehörigkeit des Typs zu einer Typklasse zu spezifizieren. Ein Typ erklärt seinen Beitritt zu einer Typklasse. Damit bindet er die Signatur der entsprechenden Operationen an die in der Typklasse angegebenen Signaturen. (Details, wenn wir's können.)

Die Typklasse `Eq` erlaubt den Vergleich zweier Werte, sofern der zugrunde liegende Typ Mitglied dieser Typklasse ist.

FUNDAMENTALE FUNKTION

Test auf Gleichheit, also die Funktion ==; Typ:

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

Die Negation, also die Funktion /=, hat auch diese Signatur:

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

ALSO

Falls `a` ein Typ ist, der zur Typklasse `Eq` gehört, nimmt die Funktion `(==)` zwei Argumente vom Typ `a` und produziert einen Booleschen Wert.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Übrigens

/= ist die Negation von ==. Meist reicht es aus, die Funktion == für Instanzen eines Typs zu definieren: `Haskell` leitet daraus die Definition von /= ab.

Die Spezifikation dieser Typklasse ist also schon durch die Angabe der Funktion == vollständig. In vergleichbaren Fällen nennt man die Angabe einer Menge von Funktionen für eine Typklasse minimal vollständig ist, wenn es Haskell gelingt, alle Funktionen dieser Typklasse vollständig daraus zu konstruieren.

Beispiele

Die grundlegenden primitiven Typen sind von Geburt aus Mitglieder dieser Typklasse.

Offensichtlich

Funktionstypen können keine Mitglieder der Typklasse `Eq` sein. Folgt: Nicht alles, was in Haskell definiert werden kann, eignet sich zum Test auf Gleichheit.

Die Typklasse erlaubt die Darstellung von Werten ihrer Mitglieder als Zeichenketten. Falls also ein Wert eines Typs a dargestellt werden soll, und der Typ a der Typklasse `Show` angehört, so wird eine Zeichenkette aus diesem Wert berechnet.

Die entsprechende Funktion heißt `show` mit
`show :: (Show a) => a -> String`

Wenn Typ a zur Typklasse `Show` gehört, dann bildet `show` eine Instanz von a in eine Zeichenkette ab.

Auch dies ist nicht bei jedem Typ möglich (Funktionstypen). Die Funktion `show` muß explizit oder implizit für jeden darzustellenden Typ definiert werden.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Typisch

Die Typklasse heißt `Show`, sie wird – wie alle Typklassen – mit einem großen Anfangsbuchstaben geschrieben wird. Die Funktion heißt `show` und beginnt mit einem kleinen Buchstaben.

Beispiele

Das Ergebnis eines Aufrufs von `show` kann dann wie eine Zeichenkette behandelt werden.

```
>>> show (4 + 6)
"10"
>>> show (4 + 6) ++ "abc"
"10abc"
```

Wir werden sehen

Meist wird die Funktion `show` für zusammengesetzte Typen definiert, indem man auf die Implementierung von `show` für die Typen der einzelnen Komponenten zugreift.

Show für Boole'sche Werte

Die Vereinbarung für die Typklasse könnte so aussehen:

```
class Show a where
           show :: a -> String
```

Hier ist a ein Typparameter. Die Signatur für die Funktion `show` wird angegeben. Beachten Sie die Syntax.

Beitritt zur Typklasse

```
instance Show Bool where
         show True  = "True"
         show False = "False"
```

Hierbei

- Der Typ `Bool` erklärt seinen Beitritt zur Typklasse Show, indem er sich zur Instanz macht. Damit sind die Funktionen der Typklasse für Instanzen dieses Typs verfügbar. Beachten Sie die Syntax.
- Die Funktion `show` wird definiert wie jede andere Funktion, in diesem Fall durch Mustererkennung.

```
class Kasper a where
           zeige :: a -> String
```

Das definiert die Typklasse Kasper.

Bool tritt bei

```
instance Kasper Bool where
         zeige True  = "Wahr"
         zeige False = "Falsch"
```

Anwendung

```
>>> zeige True
"Wahr"
```

Integer tritt bei

```
instance Kasper Integer  where
    zeige x =
        ">>> " ++ (show x) ++ " <<<"
```

Anwendung

```
>>> zeige 16
">>> 16 <<<"
```

Jeder Typ muß natürlich die Funktionen in der Typklasse implementieren, also seine eigenen Implementierungen beitragen. Sie sind durch die Signaturen gebunden. Das ist wie bei der Implementierung von Interfaces in Java.

Die Typklasse `Read` kann dazu benutzt werden, eine Zeichenkette zu lesen und in den entsprechenden Wert zu konvertieren; dieser Wert wird dann als Resultat des Funktionsaufrufs zurückgegeben.

Beispiele

```
>>> read "[1, 2, 3]" ++ [5]
[1,2,3,5]
>>>> read "3" + 9
12
```

Aber Vorsicht!

```
>>> read "3"
<interactive>:1:0:
    Ambiguous type variable ‘a’ ... :
      ...
    Probable fix: add a type signature
    that fixes these type variable(s)
```

Problem

Hier ist der Kontext unklar: "3"kann die Zeichendarstellung einer ganzen oder einer reellen Zahl sein.

Rettung

Wir geben des Zieltyp an:

```
>>> read "17" :: Float
17.0
>>> read "17" :: Integer
17
```

Die Funktion `read` ist invers zu `show`. Sie kann bei komplexen Typen dazu benutzt werden, die Daten aus ihrer Darstellung als Zeichenkette wiederzugewinnen. Wie das geht, sehen wir später.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

`Ord a` sagt, daß der Typ `a` Mitglied in der Typklasse `Ord` ist, mit deren Hilfe man Größenvergleiche durchführen kann. Dieser Typ

1. hat die üblichen Booleschen Funktionen `<`, `<=`, `>=`, `>`,
2. eine Funktion `compare`

Typisierungen

```
>>> :type (>)
(>) :: (Ord a) => a -> a -> Bool
>>> :t compare
compare :: (Ord a) => a -> a -> Ordering
```

Also

Um z.B. die Funktion `(>)` anwenden zu können sollte man sicherstellen, daß die zu vergleichenden Objekte einem Typ angehören, der Mitglied der Typklasse `Ord` ist. Ist dies der Fall, so wird ein Boolescher Wert als Resultat geliefert.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

`compare :: (Ord a) => a -> a -> Ordering`

`Ordering` ist ein diskreter Typ, also ein Typ der Klasse `Enum` (s. u.), dessen Werte durch Aufzählung bestimmt werden. Er hat nur drei Werte `GT`, `LT` und `EQ`.

Die Funktion `compare` ist ganz praktisch bei Vergleichen, die sonst recht komplex ausfallen würden. Sie liefert also als Wert `GT`, `LT` oder `EQ`.

Beispiel

`3 ‘compare‘ 5` hat den Wert `LT`

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Das sind die Aufzählungstypen. Auf ihnen sind die partiellen Funktionen `pred` und `succ` definiert (Vorgänger, Nachfolger)

Beispiel

`Ordering` für die Klasse `Ord`.

Beispiele

`Bool`, `Char`, `Int`, `Integer`, `Float`, `Double`.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Das sind die numerischen Typen, u.a. `Int`, `Integer`, `Float`, `Double`.

Zahlen können zu mehreren Klassen dieser Familie gehören:

```
>>> :type 13
13 :: (Num t) => t
>>> :type pi
pi :: (Floating a) => a
```

Deshalb spricht man von polymorphen Konstanten.

Unterklassen

Bildung der Unterklassen

Floating Hierzu gehören `Float` und `Double`.

```
>>> :type exp
exp :: (Floating a) => a -> a
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Integral Hierzu gehören die Klassen `Int` und `Integer`. Wichtige Konversionsfunktion (Verwendung später)
`fromIntegral :: (Integral a, Num b) => a -> b`

Der `:info`-Befehl als Informationssystem:

```
Prelude> :info Integral
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  ...
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
   -- Defined in GHC.Real
instance Integral Integer -- Defined in GHC.Real
instance Integral Int -- Defined in GHC.Real
```

Was sagt uns das?

- `Real`
- Instanzen
- ...

Typkonstruktoren

Haskell hat diese einfachen *Typkonstruktoren*:

Funktionen: Eine Funktion vom Typ `a -> b` akzeptiert einen Wert vom Typ `a` und gibt einen Wert vom Typ `b` zurück;

Listen: `[a]` ist der zum Typ `a` gehörige Listentyp – seine Elemente bestehen aus Listen, deren Elemente vom Typ `a` sind. Die Typen der einzelnen Listenelemente müssen jeweils übereinstimmen, dürfen also nicht gemischt sein;

Paare: `(a,b)` das sind Paare, deren erstes Element vom Typ `a`, deren zweites Element vom Typ `b` ist. Damit können heterogene Daten in einer Datenstruktur gefaßt werden.

Zunächst Paare, dann Listen.

Paare

Beispiel

```
>>> let r = (1, 'a')
>>> :type r
r :: (Integer, Char)
>>> fst r
1
>>> snd r
'a'
```

Also

```
>>> :type fst
fst :: (a, b) -> a
>>> :type snd
snd :: (a, b) -> b
```

Funktionen zur Extraktion der ersten und der zweiten Komponenten.

`let`-Bindung von `r`.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Noch'n Beispiel

```
>>> let paar = (1, (+))
>>> :type paar
paar :: (Integer, Integer -> Integer -> Integer)
>>> paar
```

Konstruktion sollte klar sein: Das Paar besteht aus einer ganzen Zahl und einer Funktion.

Weiter

```
>>> paar
<interactive>:1:0:
    No instance for
       (Show (Integer -> Integer -> Integer))
      arising from a use
      of ‘print’ at <interactive>:1:0-3
```

Wir können Paare offenbar nur dann drucken, wenn die beiden Komponenten druckbar, also Elemente der Typklasse `Show`, sind.

Paare

Weiter

```
>>> let r = (snd paar) (fst paar)
>>> :t r
r :: Integer -> Integer
>>> r 5
6
```

Hier verfertigen wir aus paar eine neue Funktion `r :: Integer -> Integer` und führen `r` mit dem Argument 6 aus.

Alternative

```
let r = (\p -> (snd p)(fst p))
>>> :t r
r :: (a, a -> t) -> t
>>> :t r paar
r paar :: Integer -> Integer
>>> r paar 6
7
```

Also

Wir weisen `r` eine anonyme Funktion zu (Typisierung!), übergeben an `r` den Ausdruck paar (Typisierung!) und werten die entstehenden Funktion aus.

Paare

Paare sind Spezialfälle von Tupeln, die aus *beliebig vielen heterogenen* Komponenten bestehen können.

- Sie können natürlich nicht eine Funktion für ein Tupel mit vier Komponenten definieren und mit sieben Komponenten aufrufen.

Die entsprechenden Extraktionsfunktionen müssen Sie aber dann selbst schreiben.

Beispiel

```
eins (x, y, z) = x
zwei (x, y, z) = y
drei (x, y, z) = z
```

Alternative

```
eins (x, _, _) = x
zwei (_, y, _) = y
drei (_, _, z) = z
```

Nicht interessierende Komponenten (*don't care*) werden einfach durch den Unterstrich _ ersetzt.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Listen sind die zentrale Datenstruktur in Haskell (ähnlich wie *Mengen* in der Mathematik).

Listen werden aufgeschrieben, indem man ihre einzelnen Elemente aufschreibt oder eine Spezifikation der Elemente angibt (*list comprehension*).

Beispiel

```
>>>  [x*y | x <- [1 .. 3], y <- [2 .. 5], odd (x+y)]
[2,4,6,10,6,12]
```

BESTANDTEILE

`[x*y | x <- [1 .. 3], y <- [2 .. 5], odd (x+y)]` besteht aus diesen Bestandteilen:

AUSDRUCK hier `x * y`,

TRENNSTRICH, KOMMATA der definierende Ausdruck wird vom Rest der Liste durch einen senkrechten Strich | abgetrennt, die anderen Komponenten werden durch Kommata voneinander getrennt,

ITERATOREN `x` durchläuft `x` die Liste `[1 .. 3]`, `y` die Liste `[2 .. 5]`,

PRÄDIKATE hier überprüft `odd (x+y)`, ob die Summe `x + y` ungerade ist,

Die Variablen, die in einer solchen Definition vorkommen, sind lokal für die Liste. Der am weitesten rechts stehende Iterator läuft "am schnellsten".

<-

Beachten Sie die Form der Iteratoren, die den Pfeil <- von rechts nach links benutzen. Das Prädikat wird zum Filtern benutzt: Nur Elemente, die diesen Test bestehen, werden in die resultierende Liste eingefügt

Wir benötigen bei einer Eigenschaftsliste nicht immer alle syntaktischen Komponenten in voller Schönheit, das Prädikat kann fehlen, meist reicht ein einziger Iterator.

Beispiel

```
>>> [x | x <- [5, 3, 7, 1, 4], x < 5]
[3,1,4]
```

Alle Elemente aus der Liste `[5, 3, 7, 1, 4]` werden herausgefiltert, die kleiner als 5 sind.

Wichtige und praktische Form der Listendarstellung. Wir werden sie oft benutzen. Sie wird sich später als syntaktisch verzuckert erweisen.

`[1 .. 10]` – Die Liste enthält die ganzen Zahlen zwischen `1` und `10` (Grenzen inklusive).

`[1, 4 .. 12]` – besteht aus den Zahlen `1, 4, 7, 10`: Es wird also vom ersten Element `1` in Schritten der Länge `4 - 1 = 3` aufwärts gezählt, solange die obere Grenze noch nicht erreicht ist.

`[12, 8 .. 1]` – enthält die Elemente `12, 8, 4`. Es wird ausgehend von `12` in Schritten der Länge `12 - 8 = 4` abwärts gezählt, solange die untere Grenze noch nicht erreicht ist.

`[1.0, 1.2 .. 2.0]` diese Liste wird nach demselben Muster berechnet: Es wird ausgehend von `1.0` in Schritten der Länge `1.2 - 1.0 = 0.2` aufwärts gezählt, solange die obere Grenze noch nicht erreicht ist.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Vorsicht bei reeller Arithmetik

Als Resultat wird gedruckt:

`[1.0, 1.2, 1.4, 1.5999999999999999, 1.7999999999999998, 1.9999999999999998];`

`['c' .. 'w']` – diese Liste wird durch alle kleinen Buchstaben zwischen `'c'` und `'w'` bestimmt;

`[1 ..]` – hiermit wird die unendliche Liste beschrieben, die alle positiven ganzen Zahlen enthält.

Offensichtlich

Bei der Schreibweise als Intervall muß der zugrundeliegende Datentyp geordnet sein (als der Typklasse `Ord` angehören).

Erstes Element

`head xs` das erste Element der Liste `xs` zurück. Die Funktion ist nicht definiert, falls `xs` die leere Liste ist.

- `head [1 .. 10] = 1`
- Typisierung: `head :: [a] -> a`.

Der Rest

`tail xs` gibt die Liste zurück, die durch das Entfernen des ersten Elements entsteht, sofern die Liste `xs` nicht leer ist.

- `tail[1 .. 10] = [2, 3, 4, 5, 6, 7, 8, 9, 10]`
- Typisierung: `tail :: [a] -> [a]`

Kopfschmerzen

Die Tatsache, daß diese beiden Funktionen nur *partiell definiert* sind, wird uns bald erhebliche Kopfschmerzen verursachen.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

KONSTRUKTOR

Ist `x` ein Element vom Typ `a`, `xs` eine Liste vom Typ `[a]`, so hat die Liste `x:xs` das Element `x` als erstes Element und `xs` als Rest: `head (x:xs) = x` und `tail (x:xs) = xs`.

```
1:[]==[1]
'a':['a', 'b'] == "aab"
[1, 2, 3] == 1:[2, 3] == 1:2:[3] == 1:2:3:[]
```

- `'A':[1..10]` führt zu einer Fehlermeldung: Die Typisierung des ersten Elements und der Liste sind nicht miteinander verträglich.
- Der Konstruktionsoperator `(:) :: a -> [a] -> [a]` ist wichtig, wenn wir Listen rekursiv definieren.

ABSCHNITTE

Die Funktion `take k xs` gibt die ersten `k` Elemente der Liste `xs` zurück, die Funktion `drop k xs` entfernt die ersten `k` Elemente aus der Liste `xs`.

```
>>> take 5 [1 .. 10]
[1,2,3,4,5]
>>> take 5 [1 .. 3]
[1,2,3]
>>> drop 5 [1 .. 10]
[6,7,8,9,10]
>>> drop 5 [1 .. 3]
[]
```

Typisierungen:

- `take :: Int -> [a] -> [a]`
- `drop :: Int -> [a] -> [a]`

Bedingte Abschnitte

`takeWhile :: (a -> Bool) -> [a] -> [a]`: berechnet für das Prädikat `p` und die Liste `xs` das *längste Anfangsstück* von `xs`, dessen Elemente das Prädikat `p` erfüllen. Es wird die leere Liste zurückgegeben, falls gleich das erste Element das Prädikat nicht erfüllt),

Stiefzwilling dazu

`dropWhile p xs` (selbe Signatur) schneidet das längste Anfangsstück von `xs`, dessen Elemente das Prädikat `p` erfüllen, heraus und gibt den Rest zurück.

Beispiele

```
>>> takeWhile (< 3) [1 .. 10]
[1,2]
>>> dropWhile (< 3) [1 .. 10]
[3,4,5,6,7,8,9,10]
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Extraktion

Ist `xs` eine Liste mit `n` Elementen, und liegt `k` zwischen `0` und `n - 1`, so ist `xs!!k` das `k`-te Element der Liste.

Zählung
: beginnt bei Ziffer `0`, endet bei `n-1` (Adressierung wie in Feldern C oder Java)

Bereichsüberschreitung
: Der Index `k` sollte nicht außerhalb dieses Bereichs liegen.

Element

`elem x xs` überprüft, ob `x` in der Liste `xs` als Element enthalten ist. Analog: `notElem :: (Eq a) => a -> [a] -> Bool` (guess, what)

Achtung

Die Überprüfung enthält einen impliziten Test auf Gleichheit. Daher muß der Grundtyp der Klasse `Eq` angehören: `elem :: (Eq a) => a -> [a] -> Bool`.

Außerdem spielen in unserem Stück mit:

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

Länge `length :: [a] -> Int`

Zeichenketten Zeichenketten sind Listen über dem Grundtyp `Char`.

```
>>> take 9 "Das ist das Haus vom Nikolaus"
"Das ist d"
>>> drop 9 "Das ist das Haus vom Nikolaus"
"as Haus vom Nikolaus"
>>> takeWhile (< 'y') "Das ist das Haus vom Nikolaus"
"Das ist das Haus vom Nikolaus"
>>> takeWhile (< 'k') "Das ist das Haus vom Nikolaus"
"Da"
>>> dropWhile (< 'k') "Das ist das Haus vom Nikolaus"
"s ist das Haus vom Nikolaus"
```

Berechnung durch Mustererkennung.

```
len [] = 0
len (x:xs) = 1 + len xs
```

Wird die leere Liste als Argument übergeben (erste Klausel), so wird `0` als Wert zurückgegeben. Läßt sich die Liste konstruieren als `x:xs`, so wird `1` zur Länge von `xs` addiert.

Beachten Sie die Verwendung des Konstruktors in der Parameterliste (das ist die Mustererkennung).

```
insert x [] = [x]
insert x (y:xs)
  | x <= y     = x:y:xs
  | otherwise = y:(insert x xs)
```

Der erste Wächter `x <= y` läßt uns nur passieren, wenn das Element `x` nicht größer als das erste Element `y` der Liste ist, der zweite `otherwise` weiß dann, daß `x > y`.

Muster

Das Einfügen von `x` in die leere Liste ergibt die Einerliste `[x]`. Einfügung von `x` in eine nicht-leere Liste `(y:xs)`: Abtrennen des ersten Elements `y` vom Rest `xs` durch Mustererkennung, und Vergleichen `x` mit `y`.

1. Ist `x <= y`, so gehört `x` als erstes Element in die zu konstruierende Liste, wir geben also die Liste `x:y:xs` zurück.
2. Ist `x > y`, so muß `x` entsprechend seiner Größe in die Liste `xs` eingefügt werden, wir geben also als Resultat zurück `y:(insert x xs)`.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Sortieren durch Einfügen

Mit der Funktion `insert` können wir Elemente in eine am Anfang leere Liste einfügen und erhalten eine geordnete Liste.

Das geht so

```
sortIns :: Ord a => [a] -> [a] -> [a]
sortIns [] ys = ys
sortIns (x:xs) ys = sortIns xs (insert x ys)
```

Also: Typisierung

Falls `a` ein geordneter Typ ist, produziert die Funktion `sortIns` aus zwei Listen eine neue Liste; die Listen bestehen jeweils aus Elemente des Typs `a`.

Funktionalität? – 1

Die Funktion arbeitet mit Mustererkennung.
Ist die erste Liste leer, so wird der zweite Parameter als Ergebnis ausgegeben.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Funktionalität? – 2

Ist die erste Liste hingegen von der Form `(x:xs)`

- Damit haben wir zugleich Kopf `x` und Rest `xs` der Liste

so wird `x` in die Liste eingesetzt, die als zweiter Parameter übergeben wird, und wir rufen die Funktion mit `xs` erneut auf.

Terminierung

Klar: der erste Parameter wird schrittweise *abgebaut*, bis die leere Liste erreicht ist. Dann erfolgt kein weiterer Aufruf mehr, so daß die Funktion terminiert.

Der Aufruf produziert durch Mustererkennung auch gleich den Parameter für den rekursiven Aufruf. Das ist ziemlich elegant.

```
>>> sortIns [7, 5, 12, 9] []
[5,7,9,12]
```

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Sortieren durch Einfügen

Sortieren durch Einfügen ist dann ein Spezialfall: Einfügen in eine anfangs leere Liste.

```
sortIns [] ys = ys
sortIns (x:xs) ys = sortIns xs (insert x ys)

insertSort xs = sortIns xs []
```

Variante

Falls die Funktion sortIns nur für die Funktion insertSort verwendet wird, so kann ich sie auch lokal vereinbaren. Das geht so:

```
insertSort xs = sortIns xs []
      where
         sortIns [] ys = ys
         sortIns (x:xs) ys = sortIns xs (insert x ys)
```

Syntax: `where` und Einrückungen • Lokalität und Sichtbarkeit von Namen.

NOCH'NE VARIANTE

```
insertSort' xs =
    let sortIns [] ys = ys
        sortIns (x:xs) ys = sortIns xs (insert x ys)
    in sortIns xs []
```

Syntax: `let ... in` und Einrückungen.

WEIL'S SO SCHÖN WAR

Wir verstecken die Funktion `insert` gleich mit

```
insertSort xs = sortIns xs []
       where
           sortIns [] ys     = ys
           sortIns (x:xs) ys = sortIns xs (insert x ys)
           insert x []      = [x]
           insert x (y:xs)
                | x <= y    = x:y:xs
                | otherwise = y:(insert x xs)
```

N.B.: Sie können `where` nicht schachteln.

Zum Vergleich

Die Java-Version

```
public class ArrayInsert {
        public static void main (String[] args) {
                int[] ara = {4, 7, 8, 9, 1, 3, 5, 6, 2, 0};
                for (int i= 0; i < ara.length; i++) {
                  int x = ara[i];
                  int k = 0;
                  while ((k < i) & (x >= ara[k])) k++;
                  if (k != i) {
                      for (int j = i; j > k; j--)
                          ara[j] = ara[j-1];
                      ara[k] = x;
              }
          }
    }
}
```

Quicksort

Die Idee bei *Quicksort* besteht darin, daß die zu sortierende Liste nach einem Pivot-Element `x` partitioniert wird in

1. alle Elemente kleiner als `x`,
2. alle Elemente gleich `x`,
3. alle Elemente größer als `x`.

Für die Partitionen wird dann wieder Quicksort aufgerufen.

Code

```
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = theSmaller ++ theEqual ++ theLarger
     where
         theSmaller = quickSort [y | y <- xs, y < x]
         theEqual = x:[y | y <- xs, y == x]
         theLarger = quickSort  [y | y <- xs, y > x]
```

`quickSort` liefert für die leere Liste die leere Liste.

Falls die Liste `(x:xs)` übergeben wird, so wird sie partitioniert:

- `< x` `[y | y <- xs, y < x]` das sind alle Elemente der Liste xs, die kleiner als `x` sind,
- `==x` `[y | y <- xs, y == x]` also alle Elemente, die mit `x` übereinstimmen,
- `> x` `[y | y <- xs, y > x]` guess what.

Hierauf wird `quickSort` wieder angewendet.

Guilottine

Beachten Sie die Verwendung der Mustererkennung. Damit können wir die Liste getrennt nach Kopf und Rumpf behandeln.

Java-Version

```
int partition(int p, int r) {
    int x = ara[p], k = p - 1, j = r + 1;
    while(true) {
        while(ara[--j] > x); while(ara[++k] < x);
        if(k < j) {
           int t = ara[k];  ara[k] = ara[j];  ara[j] = t;
        }
        else return j;
        }
}

void quickSort(int p, int r) {
     if (p < r) {
         int q = Partition(p, r);
         QuickSort(p, q);
         QuickSort(q+1, r);
    }
  }
```

FILTERN

Durch `[y | y <- xs, y < x]` und `[y | y <- xs, y > x]` berechne ich neue Listen, indem ich direkt auf dem aktuellen Parameter arbeite. Mir wäre es lieber, wenn ich die alte Liste funktional manipulieren würde. Das kann man durch Filtern tun.

IDEE Gegeben ist ein Prädikat (also `p :: a -> Bool`). Suche alle Elements aus einer Liste, die das Prädikat erfüllen.

ANSATZ

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = []
myFilter p (x:xs)
    | (p x)      = x:(myFilter p xs)
    | otherwise = myFilter p xs
```

Wieder ein rekursiver Ansatz über Mustererkennung.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Was geschieht?

Der erste Parameter ist ein Prädikat mit der Signatur `a -> Bool`, das zweite eine Liste vom Typ `[a]`.

Verankerung

`myFilter p [] = []`: Ist die Liste leer, so ist die leere Liste das Resultat.

Rekursionsschritt

Hat die Liste die Form `(x:xs)`, so sehen wir uns diese Fälle an:

`(p x) == True` dann wird `x` an den Anfang der Liste geschrieben, die durch `myFilter p xs` entsteht,

`(p x) == False` dann wird `x` ignoriert und `myFilter p xs` aufgerufen.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

```
>>> myFilter (< 3)  [4, 7, 2, 1, 8]
[2,1]
>>> myFilter (> 3)  [4, 7, 2, 1, 8]
[4,7,8]
>>> myFilter (== 3) [4, 7, 2, 1, 8]
[]
```

Beachten Sie den Unterschied zu `takeWhile` oder `dropWhile`.

EINGEBAUT

Eine Filter-Funktion ist so wichtig, daß sie vordefiniert ist:
`filter :: (a -> Bool) -> [a] -> [a]`.

JETZT SIEHT `QUICKSORT` SO AUS

```
quickSort [] = []
quickSort (x:xs) = theSmaller ++ theEqual ++ theLarger
     where
        theSmaller = quickSort (filter (< x) xs)
        theEqual = filter (== x) (x:xs)
        theLarger = quickSort $ filter (> x) xs
```

Beispiel

Wie sortiere ich diese Liste

```
[("Paula",11),("Gaby",4),("Hanna",2),("Maria",5),("Susanna",8),
("Anna",5), ("Kathrin",6),("Julia",14),("Lotta",3),("Eva",7)]
```

- nach Namen?
- nach Zahlen?

Der Typ der Liste ist `[([Char], Integer)]`. Angewandt auf diese Liste liefert `quickSort`:

```
[("Anna",5),("Eva",7),("Gaby",4),("Hanna",2),("Julia",14),
("Kathrin",6),("Lotta",3),("Maria",5),("Paula",11),("Susanna",8)]
```

```
>>> ("a", 5) < ("b", 0)
True
>>> (5, "b") < (7, "a")
True
```

Offenbar liegt eine lexikographische Ordnung vor.

Idee

Drehe die Komponenten der Liste um, sortiere die neue Liste und drehe die Komponenten der Ergebnisliste wieder um.

```
meinSort :: (Ord t, Ord t1) => [(t, t1)] -> [(t, t1)]
meinSort li = [dreh y | y <- quickSort um]
        where
            dreh (p, q) = (q, p); um = [dreh z | z <- li]
```

Resultat

```
[("Hanna",2),("Lotta",3),("Gaby",4),("Anna",5),("Maria",5),
("Kathrin",6),("Eva",7),("Susanna",8),("Paula",11),("Julia",14)]
```

Das klappt offenbar, ist aber keine besonders gute Lösung.

Warum? Die Lösung ist ad hoc. Es ist meist besser, eine Lösung zu finden, die durch die Kombination vorhandener Bausteine entsteht.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Mal sehen

Wir können ja nach der ersten und der zweiten Komponente filtern und dann sortieren.

```
filterFirst p [] = []
filterFirst p (x:xs) = if p(fst x)
            then x:(filterFirst p xs)
            else (filterFirst p xs)

filterSecond p [] = []
filterSecond p (x:xs) = if p(snd x)
             then x:(filterSecond p xs)
             else (filterSecond p xs)
```

Das ist langweilig: Wir wiederholen denselben Gedanken mit geringfügig unterschiedlichen Funktionen (`fst` und `snd`).

Könnten wir nicht danach parametrisieren?

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Mathematik

Sind $f : X \to Y$ und $g : Y \to Z$ Abbildungen, so ist ihre Komposition $g \circ f$ definiert als

$$g \circ f : \begin{cases} X & \to Z, \\ x & \mapsto g\big(f(x)\big). \end{cases}$$

Haskell

Das Kompositionssymbol $\circ$ wird in `Haskell` durch einen Punkt ersetzt. Typisierung des Operators:

```
>>> :info (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
infixr 9
```

Also

Der Operator nimmt zwei Argumente, eine Funktion des Typs `b -> c` und eine Funktion des Typs `a -> b` und gibt eine Funktion des Typs `a -> c` zurück, er ist rechts-assoziativ und hat die *ziemlich hohe* Priorität 9.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Beachten Sie

`p (fst x)` wird als `(p.fst) x` notiert (weil die funktionale Applikation die höchste Priorität 10 hat).

Alternative

```
filterFirst p xs = filter (p.fst) xs
filterSecond p xs = filter (p.snd) xs
```

Fare una bella figura

Das ist *viel eleganter* als unsere handgestrickte Lösung, weil wir hier vordefinierte Bausteine benutzen (statt eigene zu definieren).

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Weil's so schön war

Wir können noch ein wenig weitergehen. Wir definieren die Funktion `ffilter`, die uns die Komposition weiter parametrisieren läßt:

```
ffilter :: (a -> b) -> (b -> Bool) -> [a] -> [a]
ffilter f p = filter (p.f)
```

Curryfiziert

```
filterFirst :: (b -> Bool) -> [(b, b1)] -> [(b, b1)]
filterFirst = ffilter fst

filterSecond :: (b -> Bool) -> [(a, b)] -> [(a, b)]
filterSecond = ffilter snd
```

Oh!

Das ist schon ziemlich abstrakt.

Damit

```
fquickSort f [] = []
fquickSort f (x:xs) = theSmaller ++ theEqual ++ theLarger
     where
        t = f x
        theSmaller = fquickSort f (ffilter f (< t) xs)
        theEqual   = ffilter f (== t) (x:xs)
        theLarger  = fquickSort f (ffilter f (> t) xs)
```

Anmerkung

```
ffilter f (< f x) xs == [z | z <- xs, f z < f x]
```

Damit

Die Aufrufe

```
fquickSort fst dieseListe
fquickSort snd dieseListe
```

liefern die entsprechend sortierten Listen (wenn `dieseListe` die Ausgangsliste enthält).

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden


```
let dieseListe = [("Anna",5),("Eva",7),("Gaby",4),("Hanna",2),
("Julia",14), ("Kathrin",6),("Lotta",3),("Maria",5),("Paula",11),
("Susanna",8)]
```

Q:

Wie bekomme ich alle Vornamen?

A:

Klar: `[fst p | p <- dieseListe]`

Funktional?

Durch die `map`-Funktion:

`map fst dieseListe`

mit

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

Also

`map f xs` schickt die Funktion `f` über die Liste `xs` und sammelt die Ergebnisse ein:

`map f xs == [f x | x <- xs]`.

Alle Vornamen

```
>>> map fst dieseListe
["Paula","Gaby","Hanna","Maria","Susanna","Anna",
"Kathrin","Julia","Lotta","Eva"]
```

Sortiere die Vornamen

```
quickSort $ map fst dieseListe
```

Das kann auch so formuliert werden:

```
quickSort (map fst dieseListe).
```

Klar?

Die Funktion `zip :: [a] -> [b] -> [(a, b)]` kombiniert zwei Listen zu einer Liste von Paaren:

```
>>> zip ['a' .. 'f'] [1 ..]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6)]
```

Kann also als eigene Funktion `zipp` so geschrieben werden

```
zipp [] _ = []
zipp _ [] = []
zipp (x:xs) (y:ys) = (x, y):(zipp xs ys)
```

(Wir orientieren uns also an der kürzeren der beiden Listen).

Erweiterung

Bei zwei gegebenen Listen ganzer Zahlen möchte ich die *jeweiligen Komponenten* addieren: `[1, 2, 3]` und `[100, 200, 300]` sollen auf `[101, 202, 303]` abgebildet werden.

Das tut die Funktion `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`.

```
>>> zipWith (+) [1, 2, 3] [100, 200, 300]
[101,202,303]
```

Auch hier dient die kürzere der beiden Listen als Orientierung:

```
>>> zipWith (+) [1 .. 10] [100 .. 200]
[101,103,105,107,109,111,113,115,117,119]
```

Eigene Definition

```
zippWith :: (t -> t1 -> a) -> [t] -> [t1] -> [a]
zippWith f [] _ = []
zippWith f _ [] = []
zippWith f (x:xs) (y:ys) = (f x y):(zippWith f xs ys)
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Fakultät

Definiere die Fakultätsfunktion als unendliche Liste

```
fakt = [fakt2 n | n <- [0 ..]]
```

Q:

Was kann ich über die Liste sagen?

Der Anfang ist klar

```
fakt = 1:rest
```

rest?

Es gilt

```
rest!!n = fakt!!(n+1) = fakt!!n * (n+1)
        = fakt!!n * [1 ..]!!n = (*) (fakt!!n) ([1 ..]!!n),
```

Daraus

```
rest = zipWith (*) fakt [1 ..]
```

Insgesamt

```
fakt = 1:(zipWith (*) fakt [1 ..])
```

Mal sehen

```
>>> take 10 fakt
[1,1,2,6,24,120,720,5040,40320,362880]
```

Lustig, oder?

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Fibonacci-Zahlen

Direkte Übersetzung der rekursiven Definition der Fibonacci-Zahlen:

```
fib 0 = 0
fib 1 = 1
fib n =  fib (n-1) + fib (n-2)
```

Also: Mustererkennung zur direkten Umsetzung der Definition.

Bekanntlich

Das ist nicht besonders effizient wegen der vielen wiederholten Berechnungen.

Java: iterative Lösung für `fib n`

```
int a = 0, b = 1;
for (int k = 0; k <= n; k++) {
     int t = a;
     a = b; b = t + a,
     }
return a;
```

Beachten Sie das Zusammenspiel der lokalen Variablen.

Simulation der Iteration

```
fib1 0 (a, b) = (a, b)
fib1 n (a, b) =
     fib1 (n-1) (b, a + b)

fib2 n = fst(fib1 n (0, 1))
```

```
fib3 n = fst(fibLokal n (0, 1))
   where
      fibLokal 0 (a, b) = (a, b)
      fibLokal n (a, b) =
        fibLokal (n-1) (b, a + b)
```

`fib3` verbirgt die eigentliche Iteration in einer `where`-Klausel, `fib2` greift auf eine sichtbare Hilfsfunktion `fib1` zu.

Q:

Können wir die Fibonacci-Zahlen auch als unendliche Liste darstellen?

A:

Yes, we can!

Ansatz

```
fibs = [fib2 n | n <- [0 .. ]].
```

Klar

`fibs!!0 == 0` und `fibs!!1 == 1`

Weiter

Es gilt also

```
fib n = fibs!!n
fib (n+1) = (tail fib)!!n
```

Also gilt für die `n`-te Komponente der Liste

```
rest!!n = fibs!!n + (tail fibs)!!n
        = (+) (fibs!!n) + ((tail fibs)!!n).
```

Und daher

```
rest =  zipWith (+) fibs (tail fibs).
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

ZUSAMMENGEFASSÜT

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs).
```

ANMERKUNG

Das ist die sehr elegante Darstellung aller Fibonacci-Zahlen in einem einzigen Ausdruck.

DAS GEHT AUCH MATHEMATISCH

Setze

$$\mathcal{G}(z) := \sum_{n=0}^{\infty} F_n \cdot z^n,$$

so kann man elementar zeigen

$$\mathcal{G}(z) = \frac{z}{1 - z - z^2}.$$

Das ist die Darstellung aller Fibonacci-Zahlen in einer einzigen Funktion.

Konstruiere einige einfache unendlichen Listen

```
>>> let g = "abc_" ++ g
>>> take 19 g
"abc_abc_abc_abc_abc"
```

```
>>> let h = [1 .. 10] ++ h
>>> take 12 h
[1,2,3,4,5,6,7,8,9,10,1,2]
```

```
>>> let q = iterate (\x -> x + 3) 0
>>> take 10 q
[0,3,6,9,12,15,18,21,24,27]
```

Oh! Was passiert hier?

Die Funktion `iterate :: (a -> a) -> a -> [a]` ist vordefiniert. `iterate f x == [x, f x, f (f x), f (f (f x)), ...]` stellt die unendliche Liste aller Iterationen von `f` für `x` dar, beginnend mit der nullten (die nullte Iteration einer Funktion ist gerade das Argument `x`).

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Genug gespielt.

Unterschiedliche Berechnung der Summe über [1, 2, 3, 4].

Von rechts

$$
\begin{aligned}
sum[1,2,3,4] &= sum[1,2,3] + 4 \\
&= sum[1,2] + (3+4) \\
&= sum[1] + (2 + (3+4)) \\
&= (1 + (2 + (3+4)))
\end{aligned}
$$

Von links

$$
\begin{aligned}
sum[1,2,3,4] &= 1 + sum[2,3,4] \\
&= (1+2) + sum[3,4] \\
&= ((1+2)+3) + sum[4] \\
&= (((1+2)+3)+4)
\end{aligned}
$$

Das sind offensichtlich unterschiedliche Herangehensweisen an die Iteration über die Liste.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

```
foldl f y [] = y
foldl f y (x:xs) = foldl f (f y x) xs
```

Überlegung

Ist `y` vom Typ `a` und die Liste `xs` vom Typ `[b]`, so sollte die Funktion `f` die Signatur `f :: a -> b -> a` haben, das Resultat der Funktion `foldl` ist dann vom Typ `a`. Also `foldl :: (a -> b -> a) -> a -> [b] -> a`.

Der zweite Parameter beim Aufruf einer nicht-leeren Liste wird rekursiv durch das Ergebnis des Funktionsaufrufs für den zweiten Parameter und das erste Element der Liste ersetzt.

Also dient der zweite Parameter als Akkumulator, der das Resultat des Funktionsaufrufs zurückgibt, wenn der dritte Parameter die leere Liste ist.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Summation von [1, 2, 3, 4]

Funktion ist (+), Akkumulator initialisiert mit 0:

```
foldl (+) 0 [1, 2, 3, 4] == foldl (+) (0 + 1) [2, 3, 4]
== foldl (+) ((0 + 1) + 2) [3, 4] == ...
== foldl (+) (0 + 1 + 2 + 3 + 4) []
== 10
```

Minimum

Funktion `min`, Akkumulator initalisiert als erstes Element der Liste.
`myMin (x:xs) = foldl min x xs`.

```
>>> myMin [3, 4, 1, 8, 0]
0
>>> myMin []
*** Exception: /Users/eed/Desktop/Probe.hs:23:0-28:
    Non-exhaustive patterns in function myMin
```

OJ VEJ!

Was ist hier passiert? Klar: das Minimum einer Liste ist nur für nicht-leere Listen definiert. Wir müssen uns also (irgendwann bald) überlegen, wie wir mit partiellen Funktionen umgehen.

LISTEN FLACHKLOPFEN

```
>>>  foldl (++) [] [[1, 2, 3], [30, 40, 50], [8, 9], [99 .. 101]]
[1,2,3,30,40,50,8,9,99,100,101].
```

Das ist eine wichtige und hß§ufig gebrauchte Funktion:
`concat :: [[a]] -> [a].`

```
>>> concat ["abc", "123"]
"abc123"
>>> concat [[1, 2, 3], [-1, -2, -3], [100, 200, 300]]
[1,2,3,-1,-2,-3,100,200,300]
```

Beispiel: Summation über Listen von Listen

Berechnung der Summe der Einzellisten mit der Funktion
`foldl (+) 0 :: (Num a) => [a] -> a;`
dann Addition der partiellen Summen.

```
sumList2 xss = fu (map fu xss)
                 where
                    fu = foldl (+) 0
```

Also: Konstruktion einer Liste als Zwischenergebnis, die dann weiterverarbeitet wird.

Hier wird der Akkumulator festgehalten (mit `0` initialisiert). Wir können aber auch den Akkumulator dynamisieren.

Zweiter Ansatz

```
sumList xss = foldl ff 0 xss
                where
                   ff = foldl (+)
```

Aus der partiellen Funktion `ff` und einem Zwischenergebnis `k` wird eine eine neue partielle Funktion
`ff k:: (Num a) => [a] -> a`
gebildet, die dann das Ergebnis liefert.

Aufgeblss§ttert

```
 sumList [[1, 2],  [3, 4, 5]] == foldl ff 0 [[1, 2], [3, 4, 5]]
                    == foldl ff 0 [1, 2]:[[3, 4, 5]]
                    == foldl ff (ff 0 [1, 2]) [[3, 4, 5]]
                    == foldl ff 3 [3, 4, 5]:[]
                    == foldl ff (ff 3 [3, 4, 5]) []
                    == foldl ff 15 [] == 15
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

IDEE

Benutzung eines Akkumulators, der zum Ende der Liste getragen wird, also so weit *nach rechts* wie möglich:

DAS SIEHT SO AUS

```
foldr f y [] = y
foldr f y (x:xs) = f x (foldr f y xs)
```

Alternative Formulierung für den nicht-leeren Fall mit `f` als Infix-Operator:

```
foldr f y (x:xs) = x `f` (foldr f y xs)
```

TYPISIERUNG

Falls `f` den Typ `a -> b -> b` und `y` den Typ `b` hat, `xs` eine Liste vom Typ `[a]`, so resultiert ein Aufruf der Funktion in einem Wert vom Typ `b`.
Also `foldr :: (a -> b -> b) -> b -> [a] -> b`.

Listenkonstruktion

```
>>> foldr (:) [1, 2, 3] [40, 50, 60]
[40,50,60,1,2,3]
```

Aufgeblss§ttert

```
 foldr (:)   [1, 2, 3] [40, 50, 60]
            == 40 : (foldr (:) [1, 2, 3] [50, 60])
            == 40 : (50: (foldr (:) [1, 2, 3] [60]))
            == 40 : (50 : (60 : (foldr (:) [1, 2, 3] [])))
            == 40 : (50 : (60 : [1, 2, 3]))
```

Allgemein

```
xs ++ ys == foldr (:) ys xs
```

Konkatenation als Rechtsfaltung

```
foldr (++) []  [[1, 2], [3, 4]]
              == foldr (++) [] ([1, 2]:[[3, 4]])
              == [1, 2] ++ (foldr (++) [] [[3, 4]])
              == [1, 2] ++ (foldr (++) [] ([3, 4]:[]))
              == [1, 2] ++ ([3, 4] ++ (foldr (++) [] []))
              == [1, 2] ++ ([3, 4] ++ [])
              == [1, 2, 3, 4]
```

Konkatenation als Linksfaltung

```
foldl (++) []  [[1, 2], [3, 4]]
              == foldl (++) [] ([1, 2]:[[3, 4]])
              == foldl (++) ((++) [] [1, 2]) [[3, 4]]
              == foldl (++) [1, 2] [[3, 4]]
              == foldl (++) [1, 2] ([3, 4]:[])
              == foldl (++) ((++) [1, 2] [3, 4]) []
              == foldl (++) [1, 2, 3, 4] []
              == [1, 2, 3, 4]
```

Idee

Wir definieren wir für die Funktion `f` eine lokale Funktion `g` durch
`g x ys = (f x):ys`.
Also: `g` hat zwei Argumente, das erste Argument aus dem Definitionsbereich von `f`, das zweite Argument ist eine Liste. `g x ys` wendet die Funktion `f` auf das erste Argument `x` an und stellt das Ergebnis an den Beginn der Liste `ys`.

Beispiel

`g ’a’ [5, 6, 7] == [f ’a’, 5, 6, 7]`.

Definition von myMap

```
myMap f xs = foldr g [] xs
               where
                  g x ys = (f x):ys
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Mal sehen

```
myMap f [1, 2]
== foldr g [] [1, 2]
== (g 1) (foldr g [] [2])
== (g 1 (foldr g [] [2])
== g 1 (g) 2 []
== g 1 g 2 []
== g 1 (f 2):[]
== (f 1):(f 2):[] == [f 1, f 2]
```

Anmerkung

Die Rechtsfaltung ist ziemlich fundamental. Man kann damit (und mit einer Schachtel Aspirin gegen die Kopfschmerzen) die Linksfaltung darstellen. Allgemein können alle primitiv-rekursiven Funktionen damit dargestellt werden.

Relationen

Ist R eine Relation über einer Menge S, so gilt $R \subseteq S \times S$, jedes Element von R ist also ein Element des cartesischen Produkts von S mit sich.

Beispiel

Ein gerichteter Graph kann als Relation dargestellt werden: der Kante von a nach b entspricht das Paar $\langle a, b \rangle$.

Bene

Damit könnte man eine Relation in Haskell darstellen als Liste von Paaren.

Alternativ

Stelle eine Relation als Funktion dar, indem ich jedem Knoten seine Adjazenzliste zuordne.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Das heissüt

Ist a der Typ der Elemente von S, so stellen wir

- die Grundmenge als Liste über a,
- die Relation selbst als Funktion `f :: a -> [a]` dar.
  - Wir ordnen also jedem Element seine unmittelbaren Nachbarn zu, so daßü also ``r `elem` f x`` genau dann gilt, wenn $\langle x, r\rangle \in R$ gilt.

Hilfsfunktionen

Wir sollten ein Element nur dann in eine Liste einfügen, wenn es noch nicht vorhanden ist. Dazu

1. definieren wir eine Hilfsfunktion `uInsert` (*unique insert*),
2. einen Operator `+++` analog zur Konkatenation.

Are you ready, Eddie?

uInsert

```
uInsert :: (Eq t) => t -> [t] -> [t]
uInsert x [] = [x]
uInsert x (y:ys)
        | x == y     = (y:ys)
        | otherwise = y:(uInsert x ys)
```

Wir müssen auf Gleichheit testen können (daher die Vorbedingung `Eq t` für den Grundtyp `t`).

Beim Aufruf `uInsert x (y:ys)` sehen wir nach, ob `x == y` gilt. Falls ja, lassen wir die Liste in Ruhe, falls nicht, lassen wir `y` unangetastet und versuchen, `x` in den Reste der Liste einzusetzen.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden


```
infixr 5 +++

(+++) xs ys = foldr uInsert xs ys
      where
          uInsert x [] = [x]
          uInsert x (y:ys)
                   | x == y    = (y:ys)
                   | otherwise = y:(uInsert x ys)
```

Also

Der Operator +++ hat dieselben Eigenschaften wir der Operator ++. Die Funktion `uInsert` achtet aber darauf, daßü ein Element nur dann in eine Liste eingefügt wird, wenn es noch nicht vorhanden ist.

Analog

Die Funktion `conccatt :: (Eq a) => [[a]] -> [a]` vermeidet Duplikate bei der Konkatenation von Listen:

```
conccatt xss = foldr (+++) [] xss
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Die Relation `gr :: [(b, b)]` ist als Liste von Paaren reprß§sentiert.

Mit einer Funktion `knoten :: (Eq b) => [(b, b)] -> [b]` extrahieren wir alle Elemente der Grundmenge.

Mit der Funktion `defAdj :: (Eq a) => [(a, b)] -> a -> [b]` definieren wir die Funktion, die jedem Knoten seine Nachbarn zuordnet.

Das sieht dann so aus

```
knoten :: (Eq b) => [(b, b)] -> [b]
knoten gr = (map fst gr) +++ (map snd gr)

defAdj :: (Eq a) => [(a, b)] -> a -> [b]
defAdj gr a = map snd $ filter (\x -> fst x == a) gr
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Umkehrfunktion

Die Funktion `mkRel` ist die Umkehrfunktion, die aus der funktionalen die relationale Darstellung berechnet.

```
mkRel :: (Eq a, Eq b) => [a] -> (a -> [b]) -> [(a, b)]
mkRel kno adj = foldr uInsert [] [(x, y) | x <- kno, y <- adj x]
```

Wir übergeben also eine Liste vom Typ `a` und eine Funktion vom Typ `a -> [b]` und erhalten eine Liste von Paaren des Typs `[(a, b)]` ohne Duplikate.

Beispiel

Aus `[1, 2]` und `1` $\mapsto$ `['a', 'b']`, `2` $\mapsto$ `['x', 'y', 'z']` entsteht diese Liste `[(1, 'a'), (1, 'b'), (2, 'x'), (2, 'y'), (2, 'z')]` von Paaren.

Erinnerung

Die transitive Hülle einer Relation R ist diejenige Relation S, für die gilt: $\langle x, y\rangle \in S$ genau dann, wenn es einen Pfad $z_0, \ldots, z_k$ mit $k > 0$ von x nach y gibt, dessen Kanten in R liegen.
Es mußü also $x = z_0$, $y = z_k$ und $\langle z_i, z_{i+1}\rangle \in R$ gelten.

Idee

Wir konstruieren eine neue Relation, indem wir Kanten propagieren.

Konkret

Falls also $\langle x, y\rangle, \langle y, z\rangle \in R$, so fügen wir eine Kante $\langle x, z\rangle$ zu unserer neuen Relation hinzu.

DIE FUNKTION upd

```
upd :: (Eq a) => (a, a) -> (a -> [a]) -> a -> [a]
upd (a, b) adj = neuAdj
    where
        t = adj a
        s = adj b
        neuAdj x
            | x == a     = t ++ (if b `elem` t then s else [])
            | x == b     = s ++ (if a `elem` s then t else [])
            | otherwise = adj x
```

ALSO

Die Funktion `upd (a, b) adj` konstruiert eine neue Abbildung `neuAdj`, die für alle Werte außüer `a` und `b` genauso aussieht wie `adj`. Falls wir eine Kante von `a` nach `b` haben, fügen wir die Nachbarn von `b` zu denen von `a` hinzu; so entsteht die Liste `neuAdj a`; analog gehen wir bei `b` vor.

Das Arbeitspferd

Diese Funktion tut die eigentliche Arbeit:

```
upd1 adj ps = foldr upd adj ps
```

Für eine Liste ps von Paaren propagiert die Funktion upd1 die Erweiterungen durch die Relation.

Akkumulator?

Der Akkumulator wird initialisiert durch die gegebene Adjazenzliste: Sie soll schließülich erweitert werden.

Die Rechtsfaltung sorgt dafür, daßü zuerst die Modifikation für eine Funktion sowie ein Paar vorgenommen, *also eine neue Funktion berechnet wird*. Mit dieser neuen Funktion geht es dann in die “nß§chste Runde”, bis die Liste erschöpft ist (wir also an ihrem Ende angekommen sind).

Berechnung der Transitiven Hülle

```
tHul :: (Eq b) => [(b, b)] -> [(b, b)]
tHul gr = mkRel dieKnoten (upd1 dieAdj allePaare)
    where
        dieKnoten = knoten gr
        dieAdj    = defAdj gr
        allePaare = [(x, y) | x <- dieKnoten,
                              y <- dieKnoten, x /= y]
```

Wir verschaffen uns die Knoten und die Adjazenzliste der Relation, erweitern die Adjazenzliste mittels `upd1` und verwandeln das Resultat zurück in eine Relation.

Anmerkung

Dieser Algorithmus wird gelegentlich nach *Floyd - Warshall* benannt. Ist n die Anzahl der Knoten, so erfordert jede Operation in der Funktion `upd` $\mathcal{O}(n)$ Vergleiche, so daßü der Algorithmus insgesamt von der Laufzeit $\mathcal{O}(n^3)$ ist.

NEVER CHANGE A WINNING TEAM?

Rekursive Lösungen sind bekannt und bewährt:

- **START** Bei einem Startwert (meist ist das die leere Liste) beginnt die Arbeit der Funktion ($\Leftrightarrow$ Induktionsbeginn)
- **SCHRITT** In rekursiven Fall wird das Argument reduziert und die Funktion mit dem reduzierte Argument erneut aufgreifen; dabei wird die eigentliche Arbeit getan ($\Leftrightarrow$ Induktionsschritt, $n \to n + 1$)
- **TERMINIERUNG** Die Terminierung hängt kritisch davon ab, ob im rekursiven Fall eine Reduktion so stattfindet, daß der Startwert in endlich vielen Schritten erreicht werden kann.

Das ist alles bekannt und bewährt. Wieso also Faltungen?

Faltungen sind fundamental

Viele Funktionen werden durch Faltungen definiert (unser Beispiel war `map`). Man kann wohl zeigen, daß alle berechenbaren Funktionen durch `foldr` definiert werden können.

Faltungen sind praktisch

Die Zweiteilung bei der rekursiven Vorgehensweise (Start, Schritt) ist nicht nötig. Der Beginn der Rekursion wird meist durch den Initialwert für den Akkumulator wiedergegeben, der Induktionsschritt durch das Weiterschalten in der Liste.

Faltungen sind kompakt

Dadurch werden manche Formulierungen durchsichtiger.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Aus DAP 1 bekannt: Mit einem Schlüsselwort und einer rotierten Buchstabenliste kann ein Text verschlüsselt werden.

VORGEHENSWEISE

VORBEREITUNGEN Wir bereiten die Verschlüsselung durch die Bereitstellung einiger Funktionen vor.

DURCHFÜHRUNG Formulierung von Codierung und Decodierung.

KRITIK Überlegungen zur durchsichtigeren Formulierung ($\Leftrightarrow$ *refactoring*).

Auf geht's.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Alphabet

Der Einfachheit halber betrachten wir nur Kleinbuchstaben; das Leerzeichen wird durch `'*'` ersetzt (Lesbarkeit).

```
alphabet = ['a' .. 'z'] ++ "*".
```

Zyklische Vertauschung

Wir entfernen das erste Element einer Liste und hängen es am Ende wieder an.

```
shift xs = if (xs == []) then [] else (tail xs) ++ [head xs]
```

Also ergibt
`shift "abcdefghijklmnopqrstuvwxyz*"` die Zeichenkette
`"bcdefghijklmnopqrstuvwxyz*a"`

Iteration

Das soll wiederholt werden, bis der letzte Buchstabe an Anfang steht. Die Iteration geschieht mit der Funktion `iterate`.

BEISPIEL

Nehmen wir ein kleineres Alphabet:

```
>>>  take 6 (iterate shift "abcde")
["abcde","bcdea","cdeab","deabc","eabcd","abcde"].
```

Die Länge von "abcde" ist fünf. Also können wir nach fünf Iterationen aufhören, weil sich dann die Werte wiederholen.

MATRIX

Die Matrix, mit derwir arbeiten, ergibt sich so:

```
dieMatrix :: [a] -> [[a]]
dieMatrix xs = take (length xs) (iterate shift xs)
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q
b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x
i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y
j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	*
l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	*	a
m	n	o	p	q	r	s	t	u	v	w	x	y	z	*	a	b
n	o	p	q	r	s	t	u	v	w	x	y	z	*	a	b	c
o	p	q	r	s	t	u	v	w	x	y	z	*	a	b	c	d
p	q	r	s	t	u	v	w	x	y	z	*	a	b	c	d	e
q	r	s	t	u	v	w	x	y	z	*	a	b	c	d	e	f
r	s	t	u	v	w	x	y	z	*	a	b	c	d	e	f	g
s	t	u	v	w	x	y	z	*	a	b	c	d	e	f	g	h
t	u	v	w	x	y	z	*	a	b	c	d	e	f	g	h	i
u	v	w	x	y	z	*	a	b	c	d	e	f	g	h	i	j
v	w	x	y	z	*	a	b	c	d	e	f	g	h	i	j	k
w	x	y	z	*	a	b	c	d	e	f	g	h	i	j	k	l
x	y	z	*	a	b	c	d	e	f	g	h	i	j	k	l	m
y	z	*	a	b	c	d	e	f	g	h	i	j	k	l	m	n

Der Schlüssel

Wir benötigen einen Schlüssel. Dazu nehmen wir "beatles" . Der Schlüssel soll, wenn seine Buchstaben aufgebraucht sind, wieder von vorne benutzt werden:

```
key = "beatles"
aList = key ++ aList
```

```
>>> :t aList
aList :: [Char]
>>> take 50 aList
"beatlesbeatlesbeatlesbeatlesbeatlesbeatlesbeatlesb"
```

Übrigens würde

```
bList = bList ++ key
```

nicht so gut arbeiten.

Text

Wir nehmen als zu verschlüsselnden Text "all*you*need*is*love"

Die Verschlüsselung geht dann so

1. Der erste Buchstabe im Schlüssel ist b (→ Zeile 'b'). Der zu verschlüsselnde Buchstabe ist a (→ Spalte 'a'), er steht an letzter Position in der b-Zeile. Der Buchstabe '*' ist im Schnittpunkt von Zeile und Spalte.
2. Der zweite Buchstabe des Schlüssels ist e, der zweite zu verschlüsselnde Buchstabe ist l. Wir sehen uns also den Schnittpunkt der e-Zeile mit der l-Spalte an, dort finden wir den Buchstaben i.
3. Der dritte Buchstabe im Schlüssel ist a, der dritte Buchstabe im Text ist l, der Schnitt der a-Zeile mit der l-Spalte gibt l, der Schnitt der t-Zeile mit der y-Spalte ergibt e. Etc.

Also: Schlüssel → Zeile, Buchstabe → Spalte.

SYSTEMATISCH

Notwendige Schritte zur Verschlüsselung des Buchstaben `x` mit dem Schlüssel-Buchstaben `k`:

1. Suche in der Matrix die Zeile, die mit dem Buchstaben `k` beginnt; wir nennen sie `k`-Zeile,
2. stelle die Position des Buchstaben `x` in der `k`-Zeile fest, sagen wir `xPos` (also gilt `kZeile!!xPos == x`),
3. gebe `alphabet!!xPos` als Verschlüsselung zurück.

BEISPIEL

Der erste Buchstabe im Schlüssel ist `b`, also sieht die `b`-Zeile so aus: `"bcde...z*a"` . Dann gilt `xPos == 27` für den Buchstaben `'a'`, also wird `'*'` zurückgegeben.

Index

Wir benötigen den Index (des ersten Vorkommens) eines Elements in einer Liste:

```
indexIn :: (Eq a) => a -> [a] -> Int
indexIn x xs = length (takeWhile (/= x) xs)
```

Beispiel

```
>>> indexIn 3 [1 .. 10]
2
>>> takeWhile (/= 3) [1 .. 10]
[1,2]
>>> indexIn 3 ([1 .. 10] ++ [1 .. 10])
2
```

Suche der Zeile

Wir benötigen die Zeile innerhalb der Matrix, die mit diesem Buchstaben beginnt. Das geht mit einer anonymen Funktion und der Funktion `dropWhile`:

- Die (anonyme) Funktion `(\t -> head t /= x)` gibt für eine Liste genau dann den Wert `True` zurück, wenn das erste Element ungleich `x` ist; Typ: `[a] -> bool`, sofern `x :: a` (mit `Eq a` als Vorbedingung).
- Die Zeile, in der sich der Buchstabe `x` als erstes Element findet, wird so berechnet:

```
dieZeile :: (Eq a) => a -> [[a]] -> [a]
dieZeile x xss = head $ dropWhile (\t -> head t /= x) xss
```

Illustration

```
>>> (\t -> head t /= 3) [1 .. 10]
True
>>> (\t -> head t /= 1) [1 .. 10]
False
>>> dropWhile (\t -> head t /= 'y') ["bcs", "ygab", "efgs"]
["ygab","efgs"]
```

Beispiel

```
>>> dieZeile 'w' (dieMatrix (['a' .. 'z'] ++ ['*']))
"wxyz*abcdefghijklmnopqrstuv"
```

Kodierungsfunktion

Zusammengefaßt wird ein Buchstabe so kodiert:

```
encode :: Char -> Char -> Char
encode x y = alphabet!!k
               where
                  alphabet = ['a' .. 'z'] ++ "*"
                  r = dieZeile y (dieMatrix alphabet)
                  k = indexIn x r
```

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Kodierungsfunktion

Die Kodierung einer Zeichenkette läuft mit der Funktion encode über den String:

```
code :: [Char] -> [Char]
code x = encoding x aList
           where
              encoding = zipWith encode
              key = "beatles"
              aList = key ++ aList
```

Hierbei

Achten wir auf die Typisierung:
`zipWith encode :: [Char] -> [Char] -> [Char]`

Beispiel

```
>>> code "all*you*need*is*love*"
"*hlhnkczjemtwrrwlwkai"
```

Entschlüsselung

Völlig analog läuft die Entschlüsselung:

1. Wir suchen nach der Position `k` von `x` in dem Alphabet der ursprünglich gegebenen Zeichenkette,
2. wir identifizieren die Zeile `r` in der Matrix, die mit `y` beginnt,
3. wir geben das Zeichen in `r`, das an der Position `k` sitzt, also `r!!k`, als Ergebnis zurück.

Also

```
decode :: Char -> Char -> Char
decode x y = r!!k
               where
                  alphabet = ['a' .. 'z'] ++ "*"
                  r = dieZeile y (dieMatrix alphabet)
                  k = indexIn x alphabet
```

Das liefert die Dekodierung für einen einzelnen Buchstaben.

Die Dekodierung selbst

```
uncode :: [Char] -> [Char]
uncode x = decoding x aList
             where
                decoding = zipWith decode
                key = "beatles"
                aList = key ++ aList
```

O Wunder

```
>>> uncode (code "all*you*need*is*love")
"all*you*need*is*love"
```

So, das war's

War's das wirklich? Der Code ist eigentlich nicht besonders elegant.

Probleme

- Dasselbe Code-Muster wird für die Verschlüsselung und auch für die Entschlüsselung verwendet. Wir nutzen diese strukturelle Ähnlichkeit aber nicht aus.
- Die Matrix wird für jeden Buchstaben, der verschlüsselt werden soll, *und* für jeden Buchstaben, der entschlüsselt werden soll, erneut berechnet. Es handelt sich jeweils um dieselbe Matrix.
  - Optimierer?
- Die Verschlüsselung und die Entschlüsselung berechnen beide dieselbe unendliche Liste für die Schlüssel.
  - Optimierer?
- Die Modifikation des Schlüssels oder des Alphabets, das zur Verschlüsselung verwendet wird, muß an mehr als einer Stelle im Code erfolgen; das ist ziemlich fehleranfällig.
- Der Code ist insgesamt ziemlich schwerfällig, nicht ganz einfach zu verstehen und umständlich zu pflegen.

Spezifisch

```
encoding :: [Char] -> [Char] -> [Char]
encoding xs ys  = zipWith encode xs ys

decoding :: [Char] -> [Char] -> [Char]
decoding xs ys  = zipWith decode xs ys
```

Hierbei:

- die Zeichenkette `xs` wird verschlüsselt,
- `ys` dient als Schlüssel,

Andere Formulierung

Hilfsfunktion `flip :: (a -> b -> c) -> b -> a -> c` vertauscht die Argumente: `flip f x y == f y x`.

```
>>>  zip "abc" [1, 2, 3]
[('a',1),('b',2),('c',3)]
```

```
>>> flip zip "abc" [1, 2, 3]
[(1,'a'),(2,'b'),(3,'c')]
```

Damit

```
code xs   = flip (zipWith decode)  aList xs
uncode xs = flip (zipWith encode)  aList xs
```

Das Argument `xs` steht jetzt "ganz außen".

Curryfizierung

```
code :: [Char] -> [Char]
code   = flip (zipWith decode)  aList

uncode :: [Char] -> [Char]
uncode = flip (zipWith encode)  aList
```

Beobachtung

Die Funktionen `code` und `uncode` sehen sehr ähnlich aus und unterscheiden sich nur durch die Funktionen `decode` und `encode`. Diese Funktionen könnten wir als Parameter für eine (abstraktere) Funktion `vignere` nehmen.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Funktion vignere

Die Funktion `vignere` nimmt die eine Funktion `a -> Char -> a` als Parameter

```
vignere :: (a -> Char -> c) -> [a] -> [c]
vignere f = flip (zipWith f) aList
```

Spezialisierung

Durch Parametrisierung ergibt sich

```
code = vignere encode
uncode = vignere decode
```

all you need is love

```
>>> code "all*you*need*is*love"
"*hlhnkczjemtwrrwlwka"
>>> uncode (code "all*you*need*is*love")
"all*you*need*is*love"
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Sie sieht das fertige Programm aus.

```
key = "beatles"
aList = key ++ aList

alphabet = ['a' .. 'z'] ++ "*"
shift xs = if (xs == []) then [] else (tail xs) ++ [head xs]
dieMatrix xs = take (length xs) (iterate shift xs)
myMatrix = dieMatrix alphabet

dieZeile x xss = head $ dropWhile (\t -> head t /= x) xss
indexIn x xs = length (takeWhile (/= x) xs)

encode x y = alphabet!!k
       where r = dieZeile y myMatrix; k = indexIn x r
decode x y = r!!k
       where r = dieZeile y myMatrix; k = indexIn x alphabet
vignere p = flip (zipWith p) aList

uncode = vignere decode
code = vignere encode
```

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Module sind abgeschlossene Programm-Teile, die bei Bedarf importiert oder auch exportiert werden können.

BEISPIEL

```
module Adam where
zwi = " ## "
dupl x = x ++ zwi ++ x
```

VEREINBARUNG Das Schlüsselwort `module` wird gefolgt vom Namen `Adam` des Moduls und der Definition der Daten in dem Modul. Diese Definition wird durch `where` eingeleitet. In diesem Modul werden zwei Funktionen definiert.

NAME Der Name des Modul beginnt mit einem Großbuchstaben.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Nutzung

Speichern Sie diesen Modul unter dem Namen `Adam.hs` ab.

```
import Adam
meinTest x xs = x `elem` (dupl xs)
```

Also

Wir importieren also den Modul `Adam`, dabei werden die beiden dort definierten Namen `zwi` und `dupl` für das importierende Programm sichtbar (und damit zugreifbar) gemacht.

Die Namen `dupl` und `zwi` können also benutzt werden, als ob sie im gegenwärtigen Namensraum definiert worden wären.

Konsequenz

Sie sind dort auch überall sichtbar. Was macht man bei Namenskonflikten?

OO

Namen können in Java (oder C++) durch geeignete Zugriffsspezifikationen (`public`, `package`, `private`) sichtbar oder unsichtbar gemacht werden.

Hier

Es wird explizit gesagt, was exportiert wird.

Beispiel

```
module Adam (dupl) where
zwi = " ## "
dupl x = x ++ zwi ++ x
```

Hier wird also nur `dupl` exportiert, `zwi` bleibt lokal und ist außerhalb von `Adam` nicht sichtbar.

Ein Namenskonflikt

Was geschieht, wenn eine Funktion mit diesem Namen bereits existiert?

```
module Adam where
zwi = " ## "
dupl x = x ++ zwi ++ x
concat xs = reverse xs
```

Das Problem

Die Funktion `concat` ist bekanntlich bekannt und im `Prelude` definiert. Das `Prelude` wird zu Beginn geladen — man entkommt ihm also nicht.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Mal sehen

```
>>> concat "abc"

<interactive>:1:0:
    Ambiguous occurrence ‘concat’
    It could refer to either ‘Adam.concat’, defined at ...
                             or ‘Prelude.concat’, imported from Prelude
```

Was machen wir jetzt?

Wir importieren den Modul `Prelude` importieren, verhindern jedoch explizit, daß die dort definierte Funktion `concat` exportiert wird:

```
module Adam where
import Prelude hiding (concat)
zwi = " ## "
dupl x = x ++ zwi ++ x
concat xs = reverse xs
```

Das Zauberwort ist
`import Prelude hiding (concat)`

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Beachten Sie

Wir stellen keine Export-Liste zur Verfügung, alle Namen werden exportiert.

Würden wir eine Export-Liste zur Verfügung stellen, die alle Funktionen bis auf parameterlose exportieren würde, so wäre für die benutzende Umgebung bei Nennung des Namens nicht klar, welche Version von `concat` gemeint ist

- der verhinderte Import aus `Prelude` wird ja erst im Rumpf unseres Moduls `Adam` sichtbar

Ohne weitere Vorsichtsmaßnahmen würde das bedeuten, daß alle Namen, die in dem Modul definiert werden (und auch exportiert werden), für weitere Namensgebungen nicht zur Verfügung stehen.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Modifikation

```
import Adam (dupl,concat)
meinTest x xs = x ‘elem‘ (dupl xs)
zwi xs = xs ++ " ulkiges Beispiel " ++ (concat xs)
```

Also

1. Im Modul `Adam` werden wie oben die drei Funktionen exportiert, weil wir keine Exportbeschränkungen formuliert haben.
2. Der Import `import Adam (dupl,concat)` beschränkt jedoch die Funktionen aus dem Modul, die wir verwenden wollen
   - Wenn wir lediglich die Funktion `dupl` und `concat` importieren wollen, so geben wir ihre Namen in einer Liste nach dem Modulnamen an.

Damit erreicht

Kein Namenskonflikt.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Qualifiziert

```
import qualified Adam
meinTest x xs = x ‘elem‘ (Adam.dupl xs)
zwi xs = "Ein Text " ++ Adam.zwi
```

Was heisst das?

- Wir importieren die entsprechenden Namen durch die `Import`-Klausel für den Modul
- Wir benutzen sie jedoch nur in qualifizierter Form
  - das geschieht, indem wir den Namen des Moduls vor die Funktion schreiben
  - beide werden durch einen Punkt (ohne Leerzeichen) voneinander getrennt.

Alternative

`import Adam as A (dupl,concat)` Dann würden wir statt `Adam.zwi` schreiben müssen `A.zwi`.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Bislang

Bis jetzt haben wir vordefinierte Typen benutzt. Jetzt fangen wir an, eigene Typen zu definieren.

Beispiel

```
data EinPunkt = Punkt Float Float
```

Analyse

- Das Schlüsselwort `data` sagt, daß wir einen eigenen Datentyp definieren.
- Der Name des Typs ist ein `EinPunkt`.
- Er hat einen *Konstruktor* mit Namen `Punkt` und zwei Komponenten vom Typ `Float`.

Die Schreibweise ist bemerkenswert

Während wir bislang meist Namen benutzt haben, die mit Kleinbuchstaben beginnen, verwenden wir hier Bezeichner, die mit einem großen Buchstaben anfangen. Das muß so sein!

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Typisierung

```
>>> :t Punkt
Punkt :: Float -> Float -> EinPunkt
```

Der Konstruktor `Punkt` ist also eine Funktion mit zwei Argumenten, die eine Instanz des Datentyps `EinPunkt` produziert. Manchmal werden der Datentyp und der Konstruktor für den Typ mit dem gleichen Namen bezeichnet.

Nutzung

Konstruieren wir nun einen Punkt `EinPunkt`:

```
>>> Punkt 3.0 4.0

<interactive>:1:0:
    No instance for (Show EinPunkt)
      arising from a use of ‘print’ at <interactive>:1:0-12
```

Wir bekommen eine Fehlermeldung, die zeigt, daß wir unseren Datentyp nicht als Instanz der Klasse `Show` ausgewiesen haben.

Instanziierung

```
instance Show EinPunkt where
        show Punkt x y = "x: " ++ show x ++ ", y: " ++ show y
```

Also

Wir definieren die Funktion `show` für Instanzen des Typs `EinPunkt`, indem wir auf die `show`-Funktion der Komponenten zurückgreifen.

Bemerkenswert

Punkte werden über den Konstruktor konstruiert (wer hätte das gedacht), aber auch angesprochen. Das ist ganz hilfreich beim Mustervergleich.

- Hätten wir Punkte als Paare von `Float` definiert und, sagen wir, Meßwerte auch, so könnten wir die Instanzen der Typen nicht auseinanderhalten.

```
>>> Punkt 3.0 4.0
x: 3.0, y: 4.0
```

Klappt also. Es wäre auch ganz schön, wenn wir berechnen könnten, ob zwei Punkte gleich sind:

```
>>> Punkt 3.0 4.0 == Punkt 3.0 4.0

<interactive>:1:0:
    No instance for (Eq EinPunkt)
      arising from a use of ‘==’ at <interactive>:1:0-29
```

Weia! Schon wieder!

Das Problem besteht also offenbar darin, daß wir unseren neuen Typ `EinPunkt` auch in der Typklasse `Eq` “anmelden” müssen.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Zwei Punkte sollen dann gleich sein, wenn die einzelnen Komponenten übereinstimmen.

```
instance Eq EinPunkt where
         Punkt x y == Punkt x' y' = x == x' && y == y'
```

Mustererkennung auch hier: Sobald wir sehen, daß zwei Werte mit Hilfe des Konstruktors `Punkt` konstruiert werden, sehen wir uns die entsprechenden Argumente an.

Anmerkung

Wo Gleichheit definiert ist, sollte man auch Ungleichheit kennen: Die Definition der Funktion /= ergibt sich unmittelbar als Negation aus der Gleichheit, so daß eine separate Definition nicht notwendig ist.

.. und damit

```
>>> Punkt 3.0 4.0 == Punkt 3.0 4.0
True
>>> Punkt 3.0 4.0 /= Punkt 3.0 5.0
True
```

Mein Känguru sagt

Das ist aber ziemlich langweilig, daß ich solche elementaren Operationen wir das Darstellen und das Vergleichen immer noch separat definieren muß.

Ich aber sage zu meinem Känguru

Das geht auch anders:

```
data EinPunkt = Punkt Float Float deriving (Show, Eq)
```

Die Mitgliedschaft in den Typklassen `Show` und `Eq` stützt sich darauf, daß die entsprechenden Komponenten Elemente der zugehörigen Typklassen sind.

```
>>> Punkt 3.0 4.0
Punkt 3.0 4.0
>>> Punkt 3.0 4.0 == Punkt 3.0 4.0
True
```

Wir müssen also nicht notwendig diese Funktionen `show` und `(==)` explizit definieren. Wenn wir wollen, können wir uns auf die Mitgliedschaft der Komponenten in den entsprechenden Klassen abstützen.

Manchmal greift man aber lieber auf die Möglichkeit zurück, die eigenen Definitionen für die Gleichheit oder für die Repräsentation als Zeichenkette (oder was auch immer) zu nutzen.

Extraktion von Komponenten

Durch Mustererkennung:

```
xVal (Punkt x _) = x
yVal (Punkt _ y) = y

>>> :type xVal
xVal :: EinPunkt -> Float
```

Die Funktion `xVal` nimmt also einen Punkt und extrahiert die erste Komponente (der Namen der zweiten Komponenten wird als *don't care* behandelt). Analog für de zweite Komponente.

Manchmal integriert man diese Extraktionsmöglichkeiten gleich in die Definition des Datentyps, wie hier:

```
data EinPunkt = Punkt {xVal :: Float, yVal:: Float}
                deriving (Show, Eq)
```

Man sieht

Die Funktionen in den Komponenten haben implizit `EinPunkt` als Typ ihres Arguments, so daß lediglich der Typ des Werts angegeben werden muß.

Typisierung

```
>>> :type xVal
xVal :: EinPunkt -> Float
```

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Noch'n Beispiel: Kreise

```
data Kreis = Kreis {mittelPunkt :: EinPunkt, radius :: Float}
             deriving (Show, Eq)
```

Die Komponentenfunktionen sind gleich in die Definition integriert.

Wir stützen uns auf den vorhandenen Typ `EinPunkt` ab.

Die Mitgliedschaft in den Typklassen `Show` und `Eq` wird durch die entsprechenden Eigenschaften der Komponenten abgeleitet.

Signaturen

```
mittelPunkt :: Kreis -> EinPunkt,
radius :: Kreis -> Float.
```

Fächenberechnung

```
flaeche :: Kreis -> Float
flaeche c = pi * (radius c)^2
>>> let w = Kreis (Punkt 3.0 4.0) 3.0
>>> w
Kreis {mittelPunkt = Punkt {xVal = 3.0, yVal = 4.0},
       radius = 3.0}
>>> flaeche w
28.274334
```

Alternative

```
kreisFlaeche :: Kreis -> Float
kreisFlaeche (Kreis _ r) = pi*r^2
```

Rechtecke

```
data Rechteck = Rect {obenLinks :: EinPunkt,
                            untenRechts :: EinPunkt}
                deriving(Show)
```

Wir geben wir zwei Punkte vor (oben links, unten rechts) und definieren so die entsprechende Figur.

```
>>> Rect (Punkt 3.0 4.0) (Punkt 15.0 17.0)
Rect {obenLinks   = Punkt {xVal = 3.0, yVal  = 4.0},
      untenRechts = Punkt {xVal = 15.0, yVal = 17.0}}
```

Fläche

```
rechtEckFlaeche :: Rechteck -> Float
rechtEckFlaeche (Rect p1 p2) = abs (a1 * a2)
    where
       a1 = (xVal p1 - xVal p2)
       a2 = (yVal p1 - yVal p2)
```

Sollte klar sein.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Kreise und Rechtecke

```
data Figur = Rect {obenLinks :: EinPunkt, untenRechts :: EinPunkt}
             |
             Kreis {mittelPunkt :: EinPunkt, radius :: Float}
               deriving(Show)
```

Kreise und Rechtecke

Damit ist eine geometrische Figur entweder ein Rechteck mit den zugehörigen Komponenten oder ein Kreis, auch wieder mit den entsprechenden Komponenten.
Die Alternative wird durch den senkrechten Strich | angedeutet. Die Mitgliedschaft in der Typklasse `Show` wird aus den Komponenten abgeleitet.

TYPSIGNATUREN

```
>>> :type Kreis
Kreis :: EinPunkt -> Float -> Figur
>>> :type radius
radius :: Shape -> Float
>>> :type mittelPunkt
mittelPunkt :: Shape -> EinPunkt
```

Q

Wenn eine geometrische Figur gegeben ist: zu welcher Klasse (`Rechteck` oder `Kreis`) gehört diese Figur?

A: DURCH MUSTERERKENNUNG

```
istRechteck(Rect _ _) = True
istRechteck _ = False
istKreis(Kreis _ _) = True
istKreis _ = False
```

Beispiel

```
flaeche :: Figur -> Float
flaeche s = if (istRechteck s)
              then (rechtEckFlaeche s)
              else (kreisFlaeche s)

>>> let p = Punkt 5 4
>>> let q = Punkt 14 18
>>> Rect p q
Rect {obenLinks   = Punkt {xVal = 5.0, yVal = 4.0},
      untenRechts = Punkt {xVal = 14.0, yVal = 18.0}}
>>> flaeche (Rect p q)
126.0
>>> Kreis q 12
Kreis {mittelPunkt = Punkt {xVal = 14.0, yVal = 18.0},
       radius = 12.0}
>>> flaeche (Kreis q 12)
452.38934
```

Eigene Typklasse

Rationale Zahlen sind Äquivalenzklassen von Paaren ganzer Zahlen.

$$\langle x, y\rangle \approx \langle x', y'\rangle \iff x \cdot y' = y \cdot x',$$

motiviert durch die Beobachtung

$$\frac{x}{y} = \frac{x'}{y'} \iff x \cdot y' = y \cdot x'.$$

Die Klasse $[\langle x, y\rangle]$ entspricht dann für $y \neq 0$ dem Bruch x/y, die Operationen auf den Äquivalenzklassen imitieren dann die Operationen für Brüche.

Alter Hut

Geht auf G. Cantor und andere Alte Meister zurück.

Typdefinition

```
data Quot = Quot Int Int
```

Wir machen `Quot` auch gleich zum Mitglied der Typklassen `Show` und `Eq`:

```
instance Show Quot where
  show (Quot x y) = (show x) ++ "/" ++ (show y)

instance Eq Quot where
  (Quot x y) == (Quot x' y') = x * y' == x' * y
```

```
>>> Quot 4 5
4/5
>>> Quot 4 5 == Quot 8 10
True
```

Definition der algebraischen Operationen (Addititon, Multiplikation, unäres Minus)?

Das geschieht in einer eigenen Typklasse `Frege`.

Das Känguru fragt

Was muß ich dazu tun?

A

Die Signaturen der erwünschten Operationen müssen angegeben werden. Dazu haben wir eine Typvariable als Parameter.

Das Känguru fragt weiter

Wieso denn eine Typvariable? Ich möchte doch die Operationen für einen festen Typ definieren?

A

Eine Typklasse soll für eine Schnittstelle für unterschiedliche Typen bereitstellen. Dazu muß ich dann einen konkreten Typ *einhängen* können. Das machen wir, indem wir den konkreten Typ hernehmen und an die Stelle der Typvariablen setzen.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Kängurus sind halt manchmal neugierig

Klar?

Klasse Frege

```
class Frege a where
  pp :: a -> a -> a
  mm :: a -> a -> a
  ne :: a -> a
```

Auf der Klasse `Frege` sind damit zwei binäre Operationen `pp` und `mm` definiert, zudem eine unäre Operation `ne`.

Gottlob Frege (1848 – 1925); großer deutsche Logiker in Jena.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Nächster Schritt

Der Typ Quot muß zum Mitglied der Typklasse Frege gemacht werden. Dazu ist eine Implementierung der Operation notwendig.

```
instance Frege Quot where
  (Quot x y) `pp` (Quot x' y') = Quot (x * y' + x' * y) (y * y')
  (Quot x y) `mm` (Quot x' y') = Quot (x * x') (y * y')
  ne (Quot x y)                = Quot (-x) y
```

Anwendung

```
>>> (Quot 3 4) `pp` (Quot 6 7)
45/28
>>> ne (Quot 3 4)
-3/4

>>> (Quot 3 4) `mm` (Quot 6 7) == (Quot 9 14)
True
>>> ne (Quot 3 4) == Quot 3 (-4)
True
```

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Informationen

```
>>> :info Frege
class Frege a where
  pp :: a -> a -> a
  mm :: a -> a -> a
  ne :: a -> a
   -- Defined at ...
instance Frege Quot -- Defined at ...
```

Wir haben also

1. einen eigenen Typ `Quot` definiert,
2. für diesen Typ die Funktionen `show` und `==` definiert. `Quot` wurde zum Mitglied der Typklassen `Show` und `Eq` ernannt,
3. eine eigene Typklasse `Frege` definiert, zu deren Mitglied `Quot` gemacht wurde.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Bislang

Die selbstdefinierten Datentypen, die wir bis jetzt selbst definiert haben, hingen von bereits vorhandenen Datentypen ab. Wir haben bislang jedoch keinen Typparameter benutzt, über dem ein neuer Datentyp definiert wird.

Aber

Parametrisierte Datentypen sind unverzichtbar (Beispiel Listen).

Erstes Beispiel: Maybe

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

Analyse

Eine Instanz des Typs `Maybe a` ist also

- entweder die Konstante `Nothing`
- oder von der Gestalt `Just x`, wenn `x` vom Typ `a` ist

`Just :: a -> Maybe a` ist also ein Konstruktor.

Dieser Datentyp ist hilfreich zur Beschreibung partiell definierter Berechnungen.

Beispiel

Gesucht ist eine Funktion, die für eine Liste und für ein Element das erste nachfolgende Element in der Liste zurückgibt.

Problem: Das letzte Element einer (endlichen) Liste hat keinen Nachfolger.

Problem: Es ist nicht sinnvoll, von einem nachfolgenden Element zu sprechen, wenn die Liste leer ist, oder wenn das Element nicht in der Liste ist

Wir benötigen also eine partiell definierte Funktion, die für jede Situation einen Wert zurückgibt. Wir können nicht so vorgehen:

- Das Element zurückgeben, wenn das möglich ist.
- Eine Meldung zurückgeben (`Tut mir leid, das nächste Mal gern wieder`), wenn das nicht möglich ist.

```
nachfolger :: (Eq a) => a -> [a] -> Maybe a
nachfolger x [] = Nothing
nachfolger x (y:xs)
          | xs == []                = Nothing
          | x == y && xs /= []      = Just (head xs)
          | not (x `elem` xs)       = Nothing
          | otherwise               = nachfolger x xs
```

Elegantere Variante (P. Hof)

```
nachfolger :: (Eq a) => a -> [a] -> Maybe a
nachfolger x (y1:y2:xs)
        | y1 == x = Just y2
        | y1 /= x = nachfolger x (y2:xs)
nachfolger _ _ = Nothing
```

Sie sehen

Wir sorgen also dafür, daß wir für jede auftretende Situation mit einem Rückgabewert rechnen können, obgleich unsere Funktion nur partiell definiert ist.

```
>>> nachfolger 2 [1 .. 10]
Just 3
>>> nachfolger 9 [1 .. 9]
Nothing
>>> nachfolger 2 [1, 2, 3, 4, 2]
Just 3
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Vorüberlegung

Listen sind nicht einfach *Listen* schlechthin, sie sind Listen *von irgendetwas*. Beobachtung: Die Liste `[1,2,3]` kann – scheinbar umständlicher – ein wenig anders geschrieben werden als `1:2:3:[]`.

Wir sehen uns jetzt die Bausteine an, die wir benötigen. Für andere Datentypen sieht das ähnlich aus, deshalb etwas ausführlicher.

Baustein 1

Wir benötigen einen Grundtyp, über dem wir den Listentyp aufbauen. Bei Listen ist das der Datentyp `Int`.

Baustein 2

Wir benötigen weiterhin einen Konstruktor, der es uns erlaubt, Instanzen des Datentyps zu konstruieren. Bei Listen ist das der Operator `(:) :: a -> [a] -> [a]`, instanziiert für `a = Int`.

Baustein 3

Wir brauchen schließlich ein Bildungsgesetz, mit dessen Hilfe wir Instanzen zusammensetzen können. Bei Listen geschieht das durch den Funktionsaufruf `(:) 3 (5:[]) = 3:(5:[])`.

Baustein 4

Wir benötigen schließlich Konstanten dieses Typs. In unserem Beispiel die leere Liste `[]`.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Unser selbst gebauter Datentyp `MeineListe` sollte also von einem Typparameter abhängen, er muß eine Konstante haben, und er wird eine Funktion definieren müssen, mit deren Hilfe wir Instanzen dieses Datentyps definieren können. Wir wollen auch in der Lage sein, Instanzen dieses Datentyps als Zeichenkette darzustellen, so daß wir ihn als Mitglied der Typklasse `Show` verankern sollten.

Jetzt aber

```
infixr 5 :+:
data MeineListe a = Null | a :+: (MeineListe a) deriving(Show)
```

Konstruktor

Wir *definieren* also zunächst einen rechts-assoziativen Infix-Operator `:+:` der Priorität 5. Zweck: Konstruktion von Elementen des Datentyps `MeineListe`,also ein Konstruktor für diesen Datentyp.

```
infixr 5 :+:
data MeineListe a = Null | a :+: (MeineListe a) deriving(Show)
```

Typparameter

Wir geben einen Typ `a` vor, dann ist eine Instanz des Typs `MeineListe a` rekursiv durch einen der folgenden Fälle definiert:

- Entweder es ist die Konstante `Null`
- oder es ist eine Instanz des Typs `a` gefolgt vom Operator `:+:` und einer Instanz vom Typ `MeineListe a`.

```
>>> 5 :+: (4 :+: Null)
5 :+: (4 :+: Null)
>>> 3 + 4 :+: Null
7 :+: Null
```

Da `Int` ein Mitglied der Typklasse `Show` ist, ist auch `MeineListe Int` ein Mitglied dieser Klasse.

Anmerkungen

Show Der Grundtyp muß in der Typklasse `Show` sein, sonst ist die Konstruktion nicht möglich.

- Der Typ `MeineListe Int -> Int` (Liste von Funktionen `Int -> Int`) kann so nicht konstruiert werden.

Familie Der Typparameter sorgt dafür, daß wir eine ganze Familie von Typen vereinbart haben: `MeineListe Float`, `MeineListe Kreis`, `MeineListe Quot` etc.

Erzeugung Ein Typ aus dieser Familie entsteht, indem der Typparameter instanziiert wird, also einen Wert bekommt.

`:+:`

Die Funktion `:+:` ist ein Konstruktor für den Typ, der als *Infix-Operator* geschrieben wird. Derartige Namen müssen mit einen Doppelpunkt beginnen (und umgekehrt: Der Doppelpunkt als erster Buchstabe ist für diese Zwecke reserviert).

Konkatenation

```
infixr 5 #

Null # ys = ys
(x :+: xs) # ys = x :+: (xs # ys)
```

Der infix-Operator #

```
(#) :: MeineListe t -> MeineListe t -> MeineListe t
```

zur Konkatenation ist also rechts-assoziativ und hat die Priorität 5.

Beispiel

```
>>> (3 :+: (5 :+: Null)) # (30 :+: (100 :+: Null))
3 :+: (5 :+: (30 :+: (100 :+: Null)))
```

Eigene Show-Funktion

So definieren wir unsere eigene `show`-Funktion:

```
instance (Show a) => Show (MeineListe a) where
  show Null        = ">|"
  show (x :+: y) = (show x) ++ " " ++  (show y)
```

Also

- Es muß sichergestellt sein, daß der Typ `a` Mitglied der Typklasse `Show` ist (`(Show a) =>`). Dann können wir `MeineListe a` zum Mitglied dieser Typklasse erklären (`Show (MeineListe a)`).
- Wir schreiben auf, welche Zeichenkette wir für die leere Liste zurückbekommen möchten.
- Es wird definiert, wie die Funktion `show` aussehen sollte, wenn wir eine Liste rekursiv definieren.

Klar: `deriving (Show)` sollte natürlich nicht in der Typdefinition erscheinen. Sonst gibt es einen Namenskonflikt.

Algebraische Datentypen

Ein eigener Listentyp: Einige Konstruktionen

Beispiel

```
>>> (3 :+: (5 :+: Null)) # (30 :+: (100 :+: Null))
3 5 30 100 >|
```

Gleichheit

Idee: Listen sind genau dann gleich, wenn die jeweiligen ersten Elemente übereinstimmen und die `tail`-Listen gleich sind.

```
instance (Eq a) => Eq (MeineListe a) where
  Null == Null                = True
  (x :+: y) == (x' :+: y')    =  x == x' && y == y'
  z == _                      = False
```

Klar

Die Basis-Elemente müssen sich auf Gleichheit überprüfen lassen (`(Eq a) =>`).

Beispiel

```
>>> 3 :+: (4 :+: Null) == 1 + 2 :+: (4 :+: Null)
True
>>> 3 :+: (4 :+: Null) /= 4 :+: Null
True
```

Beachten Sie: $+_6$ vs `:+:`$_5$.

Beispiel

Ich möchte 3 zu jedem Element einer Instanz von `MeineListe Int` addieren.

Klar

`map (+3) [1, 5, 9]`. Aber `map (+3) 1:(5:(9:Null))` geht nicht, weil die Funktion `map` die Signatur `map :: (a -> b) -> [a] -> [b]` hat.
Wat nu?

Typklasse Functor

Haskell definiert eine allgemeine Typklasse mit Namen `Functor`. Sie ist so spezifiziert:

```
class Functor tpK where
  fmap :: (a -> b) -> tpK a -> tpK b
```

Was ist das denn?

Erläuterung

Hierbei

- `tpK` ist ein Typkonstruktor, *kein Typ* (wir hatten bislang nur Typen nach dem Namen der Klasse).
- `tpK` ist einstellig, d.h. hat genau einen Typparameter (sonst würde das ja z. B. mit `tpK a` nicht klappen).

Weiter

Wenn wir `MeineListe` zum Mitglied der Typklasse `Functor` gemacht haben, müssen wir die Funktion `fmap` implementieren. Dazu müssen wir erklären, wie aus einer beliebigen Funktion

```
f :: a -> b
```

eine Funktion

```
fmap f :: MeineListe a -> MeineListe b
```

wird.

Das geht so

```
instance Functor MeineListe where
   fmap f Null = Null
   fmap f (x :+: y) = (f x) :+: (fmap f y)
```

Die Definition von `map` wird also an dieser Stelle durch sorgfältiges Nachvollziehen der rekursiven Struktur nachempfunden.

Beispiele

```
>>> fmap (+3) (17 :+: (18 :+: Null))
20 21 >|

>>> fmap (==3) (17 :+: (15 :+: Null))
False False >|

>>> fmap (== 'a') (foldr (:+:)  Null "all right")
True False False False False False False False False >|
```

Anmerkungen

Die Typklasse `Functor` ist an eine Konstruktion aus der *Kategorientheorie* angelehnt (Funktoren transportieren dort Morphismen aus einer Kategorie in eine andere). Es müssen diese Gesetze erfüllt sein:

- `fmap id == id` (die Identität wird in die Indentität transportiert),
- `fmap (f.g) == (fmap f).(fmap g)` (Kompositionen werden respektiert).

Mengen sind *der* fundamentale Datentyp in der Mathematik. Unser Mengentyp wird auf Listen basieren. Daher werden Mengen homogen sein müssen, also aus Elementen bestehen, die sämtlich denselben Grundtyp haben. Das ist in Anwendungen in der Informatik nicht ganz realistisch.

SETL

Die von J. T. Schwartz (NYU) in den achtziger Jahren definierte Programmiersprache `SETL` erlaubt die Repräsentation heterogener endlicher Mengen. Der Aufwand ist allerdings beträchtlich, die Performanz folgt dem Slogan *slow is beautiful*.

Nostalgie

Wir hatten hier in den Neunzigern unsere eigene `SETL`-Variante und Implementation: `ProSet` — *Prototyping with sets*. Ist tot.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Der Typ

Mengen haben eine universelle Konstante `Leer`, und endliche Mengen werden durch iteratives Einfügen in die leere Menge gebildet.

```
infixr 5 :<:
data MeineMenge a = Leer | a :<: MeineMenge a
```

Eine Menge vom Grundtyp `a` ist also entweder `Leer`, oder sie entsteht durch Einfügen eines Elements des Typs `a` in eine andere Menge über dem Typ `a`.

Test auf $\emptyset$

```
istleer Leer = True
istLeer _    = False
```

Das ist ein rein syntaktischer Test (der sich die syntaktische Form einer Menge ansieht). Hat eine Menge nach einigen Operationen kein Element, nützt dieser Test nichts.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Test auf $\in$

```
element :: (Eq a) => a -> MeineMenge a -> Bool
element x Leer = False
element x (y :<: xs) = if (x == y) then True else (element x xs)
```

Der Grundtyp a, über dem die Mengen definiert sind, muß ein Mitglied der Typklasse `Eq` sein, denn es muß ja auf Gleichheit auf der Ebene der Elemente verglichen werden können.

Test auf $\subseteq$

Erinnerung:

$$A \subseteq B \iff (\forall x : x \in A \Rightarrow x \in B)$$
$$A = B \iff A \subseteq B \text{ und } B \subseteq A$$

für die Mengen A und B.

Es folgt

$\emptyset \subseteq A$ für jede Menge A, und $A \cup \{x\} \subseteq B$, falls $x \in B$ und $A \subseteq B$.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Definition von teilMenge

```
teilMenge :: (Eq t) => MeineMenge t -> MeineMenge t -> Bool
teilMenge Leer ms = True
teilMenge (a :<: ns) ms = if (a `element` ms)
                             then teilMenge ns ms
                             else False
```

Die Bedingung `Eq t` für den Grundtyp `t` ist durch den für die `element`-Funktion notwendigen Vergleich erforderlich.

Definition von ==

Daraus die Definition von Mengengleichheit. Wir machen unseren Mengentyp zum Mitglied der Typklasse `Eq` (falls der Grundtyp in dieser Klasse ist).

```
instance (Eq a) => Eq (MeineMenge a) where
    ns == ms = (teilMenge ns ms) && (teilMenge ms ns)
```

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Auch Zwerge haben klein angefangen: Einerlisten

```
singleton :: a -> MeineMenge a
singleton x = x :<: Leer
```

Hilfsfunktionen

Das Einfügen in eine Menge soll auf die Semantik von Mengen Rücksicht nehmen: Ein bereits vorhandenenes Element soll nicht noch einmal eingefügt werden (wie bei `uInsert`).

```
insert :: (Eq a) => a -> MeineMenge a -> MeineMenge a
insert x xs
     | x `element` xs = xs
     | otherwise      = x :<: xs
```

Konversionsfunktionen

Mit diesen Funktionen konvertieren wir zwischen Listen und Mengen: `toListe` und `toMenge`.

Definitionen

```
toListe :: MeineMenge a -> [a]
toListe Leer = []
toListe (x :<: ms) = x:(toListe ms)

toMenge :: (Eq a) => [a] -> MeineMenge a
toMenge xs = foldr insert Leer xs
```

Durch die Rechtsfaltung ist `toMenge` sehr einfach. Der Akkumulator, der das Ergebnis aufnehmen soll, wird zu `Leer` initialisiert, die Funktion `insert` wird durch `foldr` über die Liste propagiert.

Expandiert

`foldr insert Leer xs` sieht expandiert so aus:

```
foldr insert y []     = y
foldr insert y (x:xs) = insert x (foldr insert y xs)
```

Linksfaltung?

Die Formulierung `foldl insert Leer xs` mit einer Linksfaltung wäre problematisch:

```
foldl insert y []     = y
foldl insert y (x:xs) = foldl insert (insert y x) xs
```

Das liegt daran, daß `insert Leer x` einen Typfehler verursachen würde.

Entfernen eines Elements

Aus der leeren Menge läßt sich nichts entfernen, und die Entfernung des Elements x aus der Menge $A \cup \{y\}$ muß das Verhältnis von y zu x und zu A untersuchen:

```
delete :: (Eq a) => a -> MeineMenge a -> MeineMenge a
delete x Leer = Leer
delete x (y :<: xs)
     | x == y     = delete x xs
     | otherwise = y :<: (delete x xs)
```

Sollte klar sein. Beachten Sie, daß bei `y :<: xs` der Fall untersucht werden muß, daß `y` in `xs` vorkommt.

Die show-Funktion

Wir wollen Mengen als Zeichenketten darstellen können, also z.B. als $\{1, 2, 3\}$. Der Grundtyp muß offensichtlich der Typklasse `Show` angehören, dann geht's:

```
instance (Show a) => Show (MeineMenge a) where
    show Leer        = "{ }"
    show (a :<: ms)
                = "{" ++ (show a) ++ (concat showRest) ++ "}"
         where
           komma     = ", "
           restListe = map show (toListe ms)
           showRest
             | istLeer ms = [""]
             | otherwise  = map (komma ++) restListe
```

Das lassen wir uns jetzt auf der Zunge zergehen.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Beispiel

```
>>> let r = toMenge [1 .. 12]
>>> r
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
>>> let s = toMenge ([1 .. 5] ++ [7 .. 12])
>>> s
{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12}
>>> s ‘teilMenge‘ r
True
>>> r == s
False
>>> r == insert 6 s
True
```

Rekursionsgleichung

Wir wollen die Potenzmenge $\mathcal{P}(A)$ einer Menge A berechnen. Offensichtlich gilt

$$\begin{aligned} \mathcal{P}(\emptyset) &= \{\emptyset\}, \\ \mathcal{P}(A \cup \{x\}) &= \mathcal{P}(A) \cup \{B \cup \{x\} \mid B \in \mathcal{P}(A)\}, \text{ falls } x \notin A. \end{aligned}$$

Also können wir diese Menge rekursiv berechnen: Zur Berechnung von $\mathcal{P}(A \cup \{x\})$ berechne man $\mathcal{P}(A)$ und füge in jedes Element dieser Menge das Element x ein. Die resultierende Menge wird mit $\mathcal{P}(A)$ vereinigt.

Vorher: $\cup$

Der Vereinigungsoperator # wird durch eine Rechtsfaltung berechnet:

```
infixr 5 #

(#) :: (Eq a) => MeineMenge a -> MeineMenge a -> MeineMenge a
ms # ns = foldr insert ms (toListe ns)
```

Also

Zur Berechnung von `ms # ns = foldr insert ms (toListe ns)` wird die zweite Menge `ns` in eine Liste verwandelt, deren Elemente Stück für Stück in die Menge `ms` eingefügt werden. Hierzu dient die Funktion `insert`, die mit der Rechtsfaltung *über die Liste propagiert* wird.

Beachten Sie, daß die Faltungen auf Listen (als letztem Argument) arbeiten. Wir könnten natürlich eine *Rechtsfaltung für Mengen* definieren, tun wir hier aber nicht.

Verallgemeinerung

Vereinigung über Mengen von Mengen.

```
bigUnion :: (Eq a) => MeineMenge (MeineMenge a) -> MeineMenge a
bigUnion nss = foldr (#) Leer (toListe nss)
```

`toListe nss` ist eine Liste von Mengen.

Jubel!

Wir können fast schon die Rekursionsgleichung $\{B \cup \{x\} \mid B \in \mathcal{P}(A)\}$ umsetzen, müssen aber noch formulieren, wie wir ein Element in jede einzelne Menge einer Menge von Mengen einfügen. Das erinnert an die Funktion `map`, die es erlaubt, eine Funktionsanwendung über die Elemente einer Liste zu verteilen

Hier kommt die Typklasse `Functor` wie gerufen.

```
instance Functor MeineMenge where
   fmap f Leer = Leer
   fmap f (a :<: ms) = (f a) :<: (fmap f ms)
```

Wir machen also Typen der Form `MeineMenge a` zu Elementen der Typklasse `Functor`, indem wir der rekursiven Konstruktion der Elemente folgen.

JETZT ABER!

```
potenzMenge ::
            (Eq a) => MeineMenge a -> MeineMenge (MeineMenge a)

potenzMenge Leer = singleton Leer
potenzMenge (x :<: ms)
     | x `element` ms = alle
     | otherwise      = alle # (fmap (insert x) alle)
      where
         alle = potenzMenge ms
```

Die Konstruktion für `x :<: ms` überprüft, ob das Element `x` in `ms` vorhanden ist. Falls ja, beschränken wir uns auf die Konstruktion der Potenzmenge für `ms`. Falls nein, berechnen wir die Potenzmenge für `ms` und vereinigen mit dieser Menge die Menge aller Mengen, die entstehen, indem wir `x` in die Elemente der Potenzmenge für `ms` einfügen. Diese Menge wird konstruiert, indem die curryfizierte Funktion `insert x` mit `fmap` über die Potenzmenge für `ms` geschickt wird.

Beispiel

```
>>> potenzMenge (toMenge ['a', '1', 'Y'])
{{'a', '1', 'Y'}, {'a', '1'}, {'a', 'Y'},
       {'a'}, {'1', 'Y'}, {'1'}, {'Y'}, { }}
```

Alternative

Falls wir lieber eine Liste aller Teilmengen haben möchten, gehen wir so vor:

```
alleTeilmengen :: (Eq a) => MeineMenge a -> [MeineMenge a]
alleTeilmengen Leer = [Leer]
alleTeilmengen (x :<: ms)
    | x `element` ms = alle
    | otherwise      = alle ++ (map (insert x) alle)
        where
           alle = alleTeilmengen ms
```

Beispiel

```
>>> alleTeilmengen (toMenge [1 .. 3])
[{ },{3},{2},{2, 3},{1},{1, 3},{1, 2},{1, 2, 3}]
```

ALTERNATIVE

Der Algorithmus zur Berechnung aller Teilmengen ist für unsere Zwecke nicht passend (wir werden sehen, warum).

BEOBACHTUNG

Wenn wir alle Teilmengen $\mathcal{P}(A)$ einer Menge $A \subseteq X$ erzeugt haben, so können wir, um alle Teilmengen von X zu erhalten, zu $\mathcal{P}(A)$ alle Teilmengen $\mathcal{P}(X \setminus A)$ berechnen

$$\mathcal{P}(X) = \mathcal{P}(A) \cup \left\{C \cup B \mid C \in \mathcal{P}(A), B \in \mathcal{P}(X \setminus A)\right\}.$$

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Das war zunächst eine *Fingerübung* zur Anwendung algebraischer Datentypen. Aber weiter: ungerichtete Graphen (das werden wir dort anwenden): Cliquen in Graphen.

VEREINBARUNG

Sei für das Folgende ein ungerichteter Graph $\mathcal{G} = (V, E)$ *ohne isolierte Knoten* festgehalten.

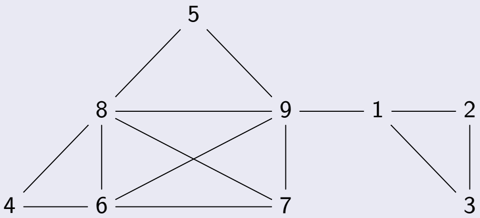
CLIQUEN

Eine *Clique* $A \subseteq V$ ist eine maximal vollständige Menge, es gilt also

VOLLSTÄNDIGKEIT Je zwei unterschiedliche Knoten in A sind durch eine Kante miteinander verbunden.

MAXIMALITÄT Ist $A \subseteq B$ und B vollständig, so gilt $A = B$ (gleichwertig damit ist, daß es zu jedem $x \notin A$ einen Knoten $y \in A$ gibt, so daß die Knoten x und y nicht durch eine Kante miteinander verbunden sind).

Beispiel



Alle Cliquen

$$\{\{1, 2, 3\}, \{1, 9\}, \{4, 6, 8\}, \{5, 8, 9\}, \{6, 7, 8, 9\}\}.$$

Die Menge $\{6, 7, 8\}$ ist vollständig, aber keine Clique.

Ziel

Berechnung aller Cliquen von $\mathcal{G}$ (Algorithmus von Bron-Kerbosch). Problem wichtig z.B. für das Operations Research.

Hilfsstruktur

Wir benötigen eine Hilfsstruktur zur Berechnung aller Cliquen

$$W_{\mathcal{G}}(A) := \{x \in V \mid \{x, y\} \in E \text{ für alle } y \in A\}$$

für die Kantenmenge $A \subseteq V$. Ein Knoten x ist also genau dann in der Menge $W_{\mathcal{G}}(A)$, wenn x mit allen Knoten der Menge A verbunden ist.

Beispielgraph

$W_{\mathcal{G}}(\{6, 7, 8\}) = \{9\}$.

Beobachtung

$A \subseteq V$ ist genau dann eine Clique, wenn A vollständig ist und $W_{\mathcal{G}}(A) = \emptyset$.

Weiter

Die Adjazenzliste $adj_{\mathcal{G}}(x)$ des Knotens x ist die Menge aller Konten, die mit x verbunden sind, also $adj_{\mathcal{G}}(x) := \{y \in V \mid \{x, y\} \in E\}$.

Berechnung der Menge $W_{\mathcal{G}}(A)$ iterativ mit Hilfe der Adjazenzlisten:

$$W_{\mathcal{G}}(\emptyset) = V,$$
$$W_{\mathcal{G}}(A \cup \{x\}) = W_{\mathcal{G}}(A) \cap adj_{\mathcal{G}}(x) \text{ für } x \notin A.$$

Vollständigkeit?

Für $x \notin A$ ist die Menge $A \cup \{x\}$ genau dann vollständig, wenn A vollständig ist und $A \subseteq adj_{\mathcal{G}}(x)$ gilt.

Das sind die wesentlichen Ingredienzien für den Algorithmus von Bron-Kerbosch. Wir müssen uns überlegen, wie wir den Graphen $\mathcal{G}$ repräsentieren.

Implementation des Graphen

Jede Kante im Graphen ist eine Zweiermenge. Eine Implementation als Menge ist aber nicht empfehlenswert (der Gesichtspunkt, daß es sich um eine *Kante* handelt, wird nicht gut modelliert). Ein eigener Datentyp muß her.

Geben wir uns die Kante

```
data UKante a = UKante a a
```

Wir formulieren also ungerichtete Kanten als parametrisierten Datentyp, dabei nehmen wir an, daß der zugrundeliegende Typ ein Mitglied der Typklassen `Eq` und `Show` ist.

```
instance (Eq a) => Eq (UKante a) where
         UKante x y == UKante x' y'
                    = (toMenge [x, y]) == (toMenge [x', y'])

instance (Show a) => Show (UKante a) where
         show (UKante x y)  = (show x) ++ " <-> " ++ (show y)
```

Kante $\rightarrow$ Menge

Die Menge, die einer Kante entspricht, läßt sich berechnen durch

```
dieKante :: (Eq t) => UKante t -> MeineMenge t
dieKante (UKante x y) = insert x (singleton y)
```

Plan

Zur Berechnung der Adjazenzliste eines Knotens gehen wir so vor: Wir berechnen alle Knoten, die von einer gegebenen Knotenmenge aus durch eine ungerichtete Kante erreichbar sind.

Ist $A \subseteq V$ eine Menge von Knoten, so berechnen wir

$$\{y \in V \mid \text{es gibt ein } z \in A \text{ mit } \{z, y\} \in E\}.$$

Die Adjazenzliste für den Knoten x ergibt sich dann durch den Spezialfall $A = \{x\}$.

Erreichbarkeit ist immer im Hinblick auf einen festen Graphen formuliert, der als Parameter übergeben wird. Unser Graph $\mathcal{G}$ wird in der Funktion `erreichbar` als Liste von Kanten repräsentiert.

```
erreichbar ::
    (Eq a) => MeineMenge a ->
               MeineMenge (UKante a) -> MeineMenge a

erreichbar ns derGraph = bigUnion
                          (toMenge ([dieKante (UKante a b) |
                             UKante a b <- kantenListe,
                             x <- knotenListe, verbindet x a b]))
     where
         kantenListe = toListe derGraph
         knotenListe = toListe ns
         verbindet x a b = x ‘element‘ (dieKante (UKante a b))
```

Also

Die `kantenListe` ist die Liste der Kanten, die `knotenListe` die Liste der Knoten in der Knotenmenge `ns`. Wir konstruieren die Menge aller Knoten, die von einem Element aus der `knotenListe` erreichbar sind, in zwei Schritten: Zunächst werden die entsprechenden Zweiermengen berechnet, die gewünschte Menge ist dann die Vereinigung über diese Mengenfamilie.

Berechnung der Adjanzenzliste

Die Adjazenzliste eines Knotens x ist dann die Menge aller Konten, die von $\{x\}$ aus erreichbar sind, wobei x ausgeschlossen wird.

```
adj :: (Eq a) => a -> MeineMenge (UKante a) -> MeineMenge a
adj x derGraph = delete x erreichbarVonx
    where
        singl = (singleton x)
        erreichbarVonx = erreichbar singl derGraph
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Berechnung aller Knoten

Der Graph wird als Liste von Kanten dargestellt, die Menge aller Knoten läßt sich rekursiv so berechnen:

```
alleKnoten :: (Eq a) => MeineMenge (UKante a) -> MeineMenge a
alleKnoten Leer = Leer
alleKnoten ((UKante a b) :<: dGr) =
           insert a (insert b (alleKnoten dGr))
```

Idee

Alle Cliquen des Graphen sollen erzeugt werden. Das kann man so machen:

- man erzeuge alle Teilmenge und untersuche jede, ob sie eine Clique ist.
  - Das ist ineffizient.
- beim Erzeugen einer Teilmenge geht man schlau vor.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Idee Bron-Kerbosch

Nehmen wir an, wir haben bereits eine vollständige Menge C erzeugt (*vollständig*: jeder Knoten ist mit jedem verbunden). Wenn wir jetzt ein Element $x \notin C$ einfügen, gibt es diese Alternativen

- x ist mit jedem Element von C verbunden
  - Dann ist $C \cup \{x\}$ ebenfalls vollständig.
- x ist nicht mit jedem Element von C verbunden. Dann können wir's nicht gebrauchen.

Falls wir für die vollständige Menge C kein x mehr finden, das mit jedem Element von C verbunden ist, dann haben wir eine Clique gefunden. In diesem Fall ist kein weiterer rekursiver Aufruf mehr notwendig.

Reservoir

Der Vorrat, aus dem wir die Elemente für C nehmen, besteht aus den Elementen der Menge $W_{\mathcal{G}}(C)$.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Signatur

Wir formulieren jetzt eine Funktion `clique` mit dieser Signatur:

```
clique :: (Eq t) =>
     MeineMenge (UKante t) -> MeineMenge t -> [t]
                           -> MeineMenge (MeineMenge t)
                           -> MeineMenge (MeineMenge t)
```

Intention

Die vier Parameter haben diese Bedeutung

- **gra** das ist der Graph als Liste von Knoten,
- **ms** bislang erzeugte vollständige Menge,
- **b** Reservoir von Kandidaten,
- **alle** die bisher erzeugten Cliquen.

Jetzt kommt's

```
clique gra ms b alle
       | b == [] = (insert ms  alle)
       | otherwise = foldr (#) alle hilf
         where
           q z  =  (\h -> h `element` adj z gra)
           hilf =
                [clique gra (insert y ms)
                              (filter (q y) b)  alle | y <- b]
```

Enthält `b` kein Element so wird `ms` zur Menge der Resultate hinzugefügt, im anderen Falle findet für jedes Element `y` von `b` ein Aufruf statt. Hierbei wird `clique` aufgerufen mit den Parametern `insert y ms` und der Liste aller Elemente von `b`, die mit `y` verbunden sind.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Damit können wir jetzt alle Cliquen für den Graphen `derGraph` berechnen:

```
dieCliquen derGraph = clique gra Leer (knotenListe derGraph) Leer
       where
            knotenListe = toListe.alleKnoten
```

Beispiel

In unserem Mustergraphen haben wir

```
>>> dieCliquen gs
{{3, 2, 1}, {1, 9}, {9, 7, 6, 8}, {9, 8, 5}, {6, 8, 4}}
```

Theorem

Der Aufruf `dieCliquen derGraph` erzeugt jede Clique des Graphen `derGraph` genau einmal.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Das Import-/Export-Verhalten von Moduln muß um Konstruktoren etc. erweitert werden.

Modul EntwederOrder

```
module EntwederOder where
data EitherOr a b = Ja a | Nein b
instance (Show a, Show b) => Show (EitherOr a b) where
         show (Ja x)    = "+++ Ja " ++ (show x)
         show (Nein y) = "-- No " ++ (show y)

sagJa x          = Ja x
sagNein y        = Nein y
istJa (Ja j)     = True
istJa _          = False
istNein (Nein f) = True
istNein _        = False
```

BEISPIEL 1

Gegeben sei

```
import  EntwederOder (sagJa,sagNein)
```

Dann:

```
>>> sagJa 3
+++ Ja 3
>>> :type sagJa 3
sagJa 3 :: EntwederOder.EitherOr Integer b
```

Die Konstruktoren (`Ja`, `Nein`) werden hier nicht explizit sichtbar.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Beispiel 2

Wir importieren nur `EitherOr`, also den Namen des Typs:

```
import  EntwederOder (EitherOr)
```

so haben wir die Konstruktoren qualifiziert zur Verfügung:

```
data EitherOr a b = EntwederOder.Ja a | EntwederOder.Nein b
    -- Defined at ...
instance (Show a, Show b) => Show (EitherOr a b)
    -- Defined at ...
```

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Beispiel 3

Durch den Import von `EitherOr(..)` werden also ebenfalls die Konstruktoren des Typs importiert:

```
import  EntwederOder (EitherOr(..))
```

Dann gilt

```
>>> :info EitherOr
data EitherOr a b = Ja a | Nein b
     -- Defined at ...
instance (Show a, Show b) => Show (EitherOr a b)
     -- Defined at ...
>>> :type Ja
Ja :: a -> EitherOr a b
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Übersicht

	EitherOr	Ja	Nein	sagJa	sagNein	istJa	istNein
①	+	–	–	–	–	–	–
②	+	+	+	–	–	–	–
③	+	–	–	+	+	–	–
④	+	+	+	+	–	–	+
⑤	+	+	+	+	+	+	+

① `module EntwederOder (EitherOr) where`
② `module EntwederOder (EitherOr(..)) where`
③ `module EntwederOder (EitherOr, sagJa, sagNein) where`
④ `module EntwederOder (EitherOr(..), sagJa, sagNein) where`
⑤ `module EntwederOder where`

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Überlegungen zu EitherOr

Der Typ `EitherOr` hängt von zwei Typparametern ab. Haskell interpretiert `EitherOr a b` als `(EitherOr a) b`.

Beispiel

Der Typkonstruktor `(EitherOr a)` kann zum Mitglied der Typklasse `Functor` gemacht werden.

```
instance Functor (EitherOr a) where
         fmap f (Nein y) = Nein (f y)
```

Dann

```
>>> fmap (+3) (Nein 4)
+++ Ja 7
```

Beachten Sie

`EitherOr a` (und nicht `EitherOr`) ist der Typkonstruktor für `EitherOr a b`. Der Versuch, `EitherOr` zum Mitglied der Typklasse `Functor` zu machen, scheitert also.

Umbebennung durch type

Mit type kann ein Typ umbenannt werden. Prominentestes Beispiel ist String als Umbenennung von [Char].

Beispiel

```
type DieAlternativen a b = EitherOr a b

>>> :info DieAlternativen
type DieAlternativen a b = EitherOr a b
   -- Defined at ..
```

Partielle Parametrisierungen sind auch möglich:

```
type KurzAlternative a = EitherOr a Char
```

Beispiel

```
type ZweiChar = (Char, Char)
type BewertetesPaar = (ZweiChar, Int)
```

Wir können also derartige Namen auch in weiteren type-Definitionen verwenden, ebenfalls in der Vereinbarung von Datentypen:

```
data WW = WW {dieChar::ZweiChar, einChar::Char} deriving(Show)
>>> :t WW
WW :: ZweiChar -> Char -> WW
>>> :t dieChar
dieChar :: WW -> ZweiChar
```

Aber

```
instance Show ZweiChar where
  show _ = "Testfall"

    Illegal instance declaration for ‘Show ZweiChar’
        (All instance types must be of the form (T t1 ... tn)
         where T is not a synonym.
```

Neue Typnamen durch newtype

```
newtype  EntOder b a = EntOder (EitherOr a b)
deriving Show

>>> :t EntOder
EntOder :: EitherOr a b -> EntOder b a
>>> :info EntOder
newtype EntOder b a = EntOder (EitherOr a b)
   -- Defined at ..
instance (Show b, Show a) => Show (EntOder b a)
  -- Defined at ..
```

Wir haben hier im wesentlichen den Typ `EntwederOrder a b` vor uns, allerdings ist die Reihenfolge der Typparameter vertauscht.

Die Mitgliedschaft in Typklassen kann bei der Verwendung von `newtype` spezifiziert werden:

```
instance Functor (EntOder a) where
         fmap f (Ja y) = Ja (f y)
```

Also

```
>>> fmap (+3) (Ja 4)
+++ Ja 7
```

Wichtige Einschränkung

Es gibt eine wichtige Restriktion bei der Verwendung von `newtype`, nämlich die Einschränkung, daß lediglich ein einziger Konstruktor verwendet werden darf.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

ZUSAMMENGEFASST

Die folgenden Möglichkeiten zur Definition von Typnamen sind vorhanden:

- **`TYPE`** Umbenennung, kein neuer Typ.
- **`NEWTYPE`** Ein neuer Typ und ein neuer Typname werden eingeführt, dabei darf lediglich ein einziger Konstruktor verwendet werden. Mitgliedschaften in Typklassen sind möglich und können auch über die `deriving`-Klausel eingeführt werden.
- **`DATA`** Die allgemeinste Möglichkeit, neue Typen zu konstruieren und Typnamen einzuführen; erlaubt insbesondere rekursive Typen.

Binäre Suchbäume sind eine populäre Datenstruktur. Wir definieren zuerst binäre Bäume und erinnern dann an Durchlaufstrategien. Dann werden binäre Suchbäume definiert.

Das alles sollte bekannt sein, so daß nur die Formulierung in Haskell neu ist. Ist aber auch nicht so richtig aufregend.

Auf geht's

```
data Baum a = Knoten a (Baum a) (Baum a)
    | Leer
    deriving (Show)
```

Ein binärer Baum ist also entweder leer, oder er hat eine Wurzel und einen linken und einen rechten Unterbaum. Wir nehmen an, daß die Werte in den Knoten vom Typ a sind. Um binäre Bäume darstellen zu können, haben wir den Typ gleich als Mitglied der Typklasse `Show` dargestellt.

Extraktionsfunktionen

```
wurzel :: Baum t -> t
wurzel (Knoten x _ _) = x
linkerUnterbaum :: Baum t -> Baum t
linkerUnterbaum (Knoten _ x _) = x
rechterUnterbaum (Knoten _ _ x) = x
```

Das ist alles wohlbekannt

Beispiel

```
>>> let wq = Knoten 's'
                  (Knoten 'i' (Knoten 'h' Leer Leer) Leer)
                  (Knoten 't' Leer Leer)

>>> wurzel wq
's'
>>> linkerUnterbaum wq
Knoten 'i' (Knoten 'h' Leer Leer) Leer
>>> rechterUnterbaum wq
Knoten 't' Leer Leer
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

PREORDER

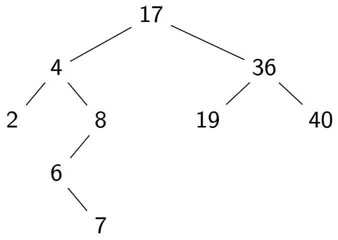
```
preorder :: Baum t -> [t]
preorder (Knoten x y z) = [x] ++ (preorder y) ++ (preorder z)
preorder Leer = []
```

INORDER

```
inorder (Knoten x y z) = (inorder y) ++ [x] ++ (inorder z)
inorder Leer = []
```

POSTORDER

```
postorder (Knoten x y z) = (postorder y) ++ (postorder z) ++ [x]
postorder Leer = []
```



PREORDER

[17, 4, 2, 8, 6, 7, 36, 19, 40, 37]

INORDER

[2, 4, 6, 7, 8, 17, 19, 36, 37, 40]

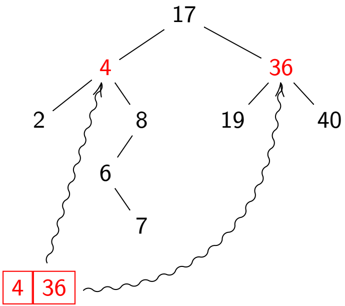
POSTORDER

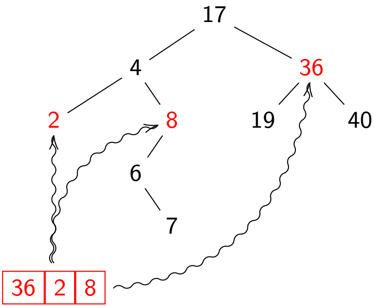
[2, 7, 6, 8, 4, 19, 37, 40, 36, 17]

Breitensuche

Realisierung durch eine Warteschlange.

Einfügen	Am Ende der Warteschlange
Entfernen	Am Kopf der Warteschlange





EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Das Arbeitspferd

Die Funktion bfs arbeitet mit zwei Listen: mit der Liste der Unterbäume, die verarbeitet werden müssen, und mit der Liste der Knoten, die wir bislang verarbeitet haben.

```
bfs :: [Baum a] -> [a] -> [a]
bfs [] xs = xs
bfs (y:ys) xs
    | istLeer y = bfs ys xs
    | otherwise = bfs qu app
          where
             qu = ys ++ [linkerUnterbaum y, rechterUnterbaum y]
             app = xs ++ [wurzel y]
```

mit

```
istLeer Leer = True
istLeer (Knoten _ _ _) = False
```

FALLUNTERSCHEIDUNG

Ist y der Baum, der verarbeitet werden soll, so gibt es zwei Fälle:

1. y ist der leere Baum, dann wird der nächste Baum in der Warteschlange verarbeitet.
2. y hat die Wurzel `r` und die Unterbäume `links` und `rechts`. Dann fügen wir
   - `r` zu den bereits besuchten Knoten hinzu,
   - `links` und `rechts` in die Warteschlange ein.

BREITENDURCHLAUF

```
myBfs :: Baum a -> [a]
myBfs derBaum = bfs [derBaum] []
```

Definition

Ein binärer Suchbaum ist ein binärer Baum mit diesen Eigenschaften:

- Der Baum ist entweder leer oder
- die Wurzel des Baums ist größer als die des linken Unterbaums (falls dieser Unterbaum existiert) und kleiner als die Wurzel des rechten Unterbaums (falls dieser existiert),
- der linke und der rechte Unterbaum sind selbst wieder binäre Suchbäume.

Beispiel

Der Baum im letzten Beispiel ist ein binärer Suchbaum.

Binäre Suchbäume sind binäre Bäume; die Bedingungen können nicht in der Datenstruktur codiert werden. Daher ist eine separate Datenstruktur nicht nötig.

Das Einfügen eines Elements x geht so:

1. Falls $x = w$ wissen wir, daß das Element bereits im Baum ist.
2. Falls $x < w$, dann setzen wir x rekursiv in den linken Unterbaum von B ein: Der resultierende Baum wird also
   - dieselbe Wurzel wie B haben,
   - x in den linken Unterbaum von B eingesetzt finden,
   - denselben rechten Unterbaum wie B haben.
3. Fall $x > w$, so fügen wir w rekursiv in den rechten Unterbaum von B ein, die Diskussion zum Ergebnis verläuft völlig analog.

Haskell

```
baumSuche :: (Ord a) => a -> Baum a -> Bool
baumSuche x Leer = False
baumSuche x ( Knoten w links rechts)
          | x == w = True
          | x < w  = baumSuche x links
          | x > w  = baumSuche x rechts
```

Analog: Suche

```
baumSuche :: (Ord a) => a -> Baum a -> Bool
baumSuche x Leer = False
baumSuche x ( Knoten w links rechts)
           | x == w = True
           | x < w  = baumSuche x links
           | x > w  = baumSuche x rechts
```

Beispiel

```
>>> let bspBaum = foldr baumEinf Leer [3, 5, 9, 12, 8, 1, 5, 7]
>>> bspBaum
 Knoten 7 ( Knoten 5 ( Knoten 1 Leer ( Knoten 3 Leer Leer)) Leer)
       ( Knoten 8 Leer ( Knoten 12 ( Knoten 9 Leer Leer) Leer))
>>> baumSuche 4 bspBaum
False
>>> baumSuche 9 bspBaum
True
```

Problem

Wir wollen eine Folge von Zeichen so codieren, daß

- die Codierung für jeden Buchstaben eine endliche Sequenz von 0 und 1 ist,
- der Code präfixfrei ist,
- die Verschlüsselung für häufig vorkommende Buchstabe kürzer ist als für weniger häufig vorkommende.

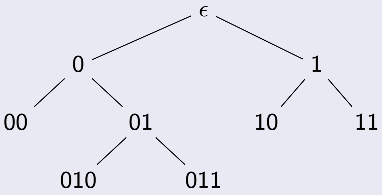
Präfixfrei

Die Verschlüsselung eines Buchstaben ist kein Präfix der Verschlüsslung eines anderen Buchstaben.

Klar

Die Kodierung erfolgt durch einen binären Baum — aber wie?

Ein Baum



Idee

Der Weg von der Wurzel zu einem Blatt kann binär codiert werden: Zweigt man nach links ab, so notiert man 0, zweigt man nach rechts ab, so codiert man 1. In den Blättern ist die Codierung des Pfades zu finden, den wir genommen haben, um zu diesem Blatt zu kommen.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Die Anforderung, daß häufige Buchstaben eine kürzere Verschlüsselung als weniger häufige haben sollen, impliziert, daß die Häufigkeit der einzelnen Buchstaben bekannt sein, also müssen wir ihr Vorkommen zählen und festhalten.

Strategie

Wir konstruieren einen binären Baum zur Codierung, speichern die einzelnen Buchstaben in den Blättern und verwenden die gerade beschriebene Pfad-Codierung als Codierung für die einzelnen Blätter. Dabei sorgen wir dafür, daß diejenigen Buchstaben, die häufiger vorkommen, näher an der Wurzel sind als die weniger häufig auftauchenden.

Baumkonstruktion: Beispiel

Häufigkeit der Buchstaben im Text:

$$f : 5 \qquad e : 9 \qquad c : 12 \qquad b : 13 \qquad d : 16 \qquad a : 45$$

Daraus konstruieren wir einen Wald aus gewichteten Bäumen. Einzelne Bäume werden zusammengefaßt, bis nur ein einziger Baum übrigbleibt.

Erster Schritt

Die Buchstaben als werden als gewichtete Bäume interpretiert, Daraus ergibt sich ein Anfangswald. Die Bäume des Waldes bestehen also am Anfang jeweils nur aus einem einzigen Knoten, dem ein Gewicht beigegeben ist.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Kombinieren von Bäumen: Idee

Die Bäume werden nun Schritt für Schritt zu größeren Bäumen kombiniert, wobei die Gewichte der einzelnen Bäume addiert werden, und die Bäume ihrem Gewicht entsprechend in den Wald eingeordnet werden.

Kombinieren von Bäumen: Genauer

Wir nehmen die beiden Bäume T_1 und T_2 *mit dem geringsten Gewicht*, wobei das Gewicht von T_1 nicht größer als das Gewicht von T_2 sein soll. Wir *kombinieren* diese beiden Bäume in einem neuen Baum T^*:

- Der Baum T^* erhält eine neue Wurzel,
- der linke Unterbaum ist T_1, der rechte Unterbaum ist T_2,
- das Gewicht von T^* ist die Summe der Gewichte T_1 und T_2.

Die Bäume T_1 und T_2 werden aus dem Wald entfernt, der neue Baum T^* wird in diesen Baum seinem Gewicht gemäß eingefügt.

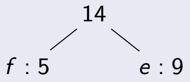
Baumkonstruktion: Beispiel

$f : 5 \qquad e : 9 \qquad c : 12 \qquad b : 13 \qquad d : 16 \qquad a : 45$

Erster Schritt

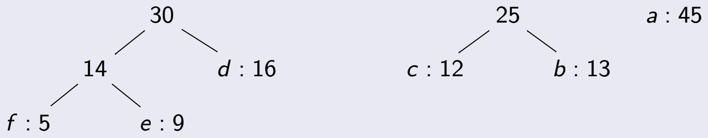
Am Anfang ist der Baum T_1 der Baum, der aus den Buchstaben `f` mit dem Gewicht 5 besteht, der Baum T_2 ist der Buchstabe `e` mit dem Gewicht 9. Diese beiden Bäume werden zu einem neuen Baum kombiniert, indem eine neue Wurzel erzeugt wird. Linker und rechter Unterbaum werden wie beschrieben definiert, das Gewicht ist die Summe der Einzelgewichte.

So sieht der Wald jetzt aus

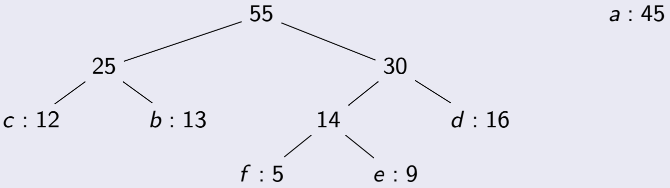


$c : 12 \qquad b : 13 \qquad d : 16 \qquad a : 45$

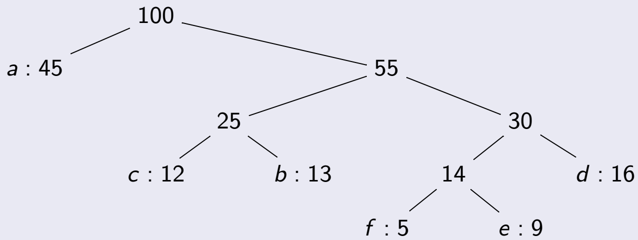
Schritt 4



Schritt 5



SCHRITT 6



KODIERUNG

Aus dem Baum können wir die Codierung der einzelnen Buchstaben ablesen.

a	b	c	d	e	f
0	101	100	111	1101	1100

Konkret: Schritte

Häufigkeit Wir müssen über den Text iterieren und die Häufigkeit für jeden Buchstaben feststellen.

Baumdarstellung Der Baum, der entsteht, muß ebenso wie die in den Zwischenschritten entstehenden Bäume als Datenstruktur repräsentiert werden.

Wald Der entstehende Wald, also die Kollektion von Bäumen, muß manipuliert werden. Die Bäume sind mit einem Gewicht versehen. Das Gewicht ist ein Auswahlkriterium. Also muß die Repräsentation des Waldes diese Gewichtsfunktion berücksichtigen.

Codierung Wir müssen schließlich die Codierung bestimmen.

Zunächst benötigen wir eine geeignete Datenstruktur für Abbildungen.

Die Datenstruktur Map

Mathematisch ist eine Abbildung eine Liste von Paaren, deren erste Komponente als Argument, deren zweite als Funktionswert dient. Hierfür gibt es die Datenstruktur `Map`: Die Abbildung

$$'a' \mapsto 1, 'b' \mapsto 2, 'C' \mapsto 3.$$

wird als `fromList [('a', 1), ('b', 2), ('C', 3)]` gespeichert

Die inverse Funktion ist `toList`:

```
>>> toList fromList [('a', 1), ('b', 2), ('C', 3)]
[('a', 1), ('b', 2), ('C', 3)]
```

Die leere Abbildung `empty` ist eine Konstante für den Datentyp. Die Funktion `null` überprüft, ob ihr Argument die leere Abbildung ist.

```
>>> :type fromList
fromList :: (Ord k) => [(k, a)] -> Map k a
>>> :type toList
toList :: Map k a -> [(k, a)]
```

Also

Der Wertebereich einer Abbildung, also die Menge, der die Argumente entnommen sind, muß geordnet sein.

Modul Data.Map

Die Definition des Datentyps und seiner Funktionen residieren im Modul `Data.Map`, also ist zur Nutzung des Typs der Import dieses Moduls erforderlich. Wir importieren lediglich die Funktionen `toList`, `fromList`, `null`, `empty` und `insertWith` (wird gleich diskutiert).

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Namenskonflikt

Beim Import sollten wir vorsichtig sein, denn der Name der Funktion `null` ist überladen: Das `Prelude` stellt diese Funktion ebenfalls zur Verfügung (eine Liste wird damit überprüft, ob sie leer ist). Daher müssen wir den Namenskonflikt auflösen, indem wir den Namen des Moduls `Map` vor die entsprechende Funktion setzen.

Idee

Wir verwenden Abbildungen, um für jeden Buchstaben seine Häufigkeit zu notieren.

1 kleines Problem

Wenn ein bereits vorhandener Buchstabe auftaucht, so müssen wir dafür sorgen, daß wir seine bisherige Häufigkeit erhöhen und als neue Häufigkeit in der Abbildung vermerken.

insertWith

Die Funktion `insertWith` hilft.
Signatur: `(Ord k) => (a -> a -> a) -> k -> a -> Map k a -> Map k a`

Aufruf

Der Aufruf `insertWith f x y ourMap` macht folgendes:

- Ist ein Paar `(x, y')` in der Abbildung `ourMap` vorhanden, so wird es in der Abbildung durch das Paar `(x, (f y' y))` ersetzt.
- Falls es dort nicht vorhanden ist, so wird das Paar `(x, y)` in `ourMap` eingesetzt.

```
>>> let r = fromList [('a', 1), ('b', 2), ('C', 3)]
>>> insertWith (+) 'a' 3 r
fromList [('C',3),('a',4),('b',2)]

>>> insertWith (+) 'e' 17  r
fromList [('C',3),('a',1),('b',2),('e',17)]
```

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Häufigkeit des Vorkommens eines Buchstaben

```
freqMap :: (Ord k, Num a) => [k] -> [(k, a)]
freqMap xs = toList (lookFreq xs)
 where
  lookFreq (y:ys) = insertWith (+) y 1 (lookFreq ys)
  lookFreq []     = empty
```

Wenn wir also eine Liste `aList` gegeben haben, so gibt uns der Aufruf `lookFreq aList` eine Abbildung, die jedes Element in dieser Liste auf seine Häufigkeit abbildet. Der Aufruf von `toList` konvertiert diese Abbildung in eine Liste von Paaren, um das anschließende Sortieren zu erleichtern.

Beispiel

```
>>> freqMap "Das ist das Haus vom Nikolaus"
[(' ',5),('D',1),('H',1),('N',1),('a',4),
('d',1),('i',2),('k',1),('l',1),('m',1),
('o',2),('s',5),('t',1),('u',2),('v',1)]
```

Sortierfunktion

```
fquickSort :: (Ord a) => (a1 -> a) -> [a1] -> [a1]
```

Neulich diskutiert.

Datentypen

Wir benötigen zwei Arten von Knoten. Wir müssen Zeichen und Häufigkeiten speichern (Blätter; `ZNode`), aber auch nur Häufigkeiten allein (innere Knoten: Baumgewichte; `Node`)

```
data Node  =  ZNode {theChar:: Char, often :: Integer}
              |
              Node {often :: Integer}
                 deriving(Show)
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Hilfsfunktionen

```
mkZNode :: Char ->  Integer -> Node
mkZNode x i = ZNode{theChar = x, often = i}

mkNode :: Integer -> Node
mkNode i = Node {often = i}
```

Zu Beginn haben wir eine Liste von Paaren, die aus den Zeichen und ihren Häufigkeiten besteht. Hieraus konstruieren wir durch die Funktion `mkpairZNode` einen Knoten:

```
mkpairZNode :: (Char, Integer) -> Node
mkpairZNode p = mkZNode (fst p) (snd p)
```

Testfunktionen

Wir sollten Knoten daraufhin überprüfen können, welcher Art sie sind.

```
isZNode(ZNode _ _) = True          isNode(Node _) = True
isZNode _= False                   isNode _= False
```

Der Baum selbst

Der Baum hat eine Wurzel sowie einen linken und einen rechten Unterbaum. Er kann auch leer sein.

```
data Baum = Baum {aNode :: Node, left :: Baum, right :: Baum}
            |
            Leer
             deriving(Show)
```

Gewichtsfunktion

`weight = often.aNode` mit `weight :: Baum -> Integer` berechnet das Gewicht eines Knotens.

Kombination von Bäumen

Der Algorithmus kombiniert zwei Bäume, sagen wir `b1` und `b2`, in diesen Schritten:

- Wir erzeugen einen neuen Knoten als die Wurzel des neuen Baums,
- wir setzen `b1` als den linken, `b2` als den rechten Unterbaum der Wurzel,
- wir nehmen die Summe der Gewichte von `b1` und `b2` als das Gewicht des neuen Baums.

Funktion mergeBaum

```
mergeBaum :: Baum -> Baum -> Baum
mergeBaum b1 b2 =
            Baum {aNode = thisNode, left = b1, right = b2}
             where
               thisNode = mkNode (weight b1) + (weight b2)
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Liste

Wir nehmen an, daß wir eine Liste von Bäumen haben, die nach ihrem Gewicht sortiert sind. Im Laufe des Algorithmus wollen wir einen neuen Baum in diese Liste einfügen, wobei die Position des Baums durch sein Gewicht gegeben ist. Also bleibt nach Einfügen des Baums diese Liste geordnet.

Funktion weightInsert

```
weightInsert :: Baum -> [Baum] -> [Baum]
weightInsert y (x:xs) = if f y <= f x
                           then y:x:xs
                           else x:(weightInsert y xs)
                           where f = weight
weightInsert y [] = [y]
```

BISLANG

Für den eingegebenen Text haben wir bislang für die Liste `charList` die folgenden Schritte durchgeführt:

- `freqMap charList` berechnet eine Liste von Paaren `(a, b)`, wobei die erste Komponente `a` ein Zeichen, die zweite Komponente `b` seine Häufigkeit ist,
- durch den Aufruf von `fquickSort snd` bekommen wir für diese Liste eine sortierte Liste, wobei die Häufigkeit des Zeichens das Sortierkriterium ist, weniger häufig vorkommende Zeichen stehen am Anfang,
- `map mkpairZNode` berechnet daraus eine Liste von `ZNode`-Knoten.

KLEINE BÄUMCHEN

Wir wollen aber nicht mit einer Liste von Knoten beginnen, wir benötigen einen Wald von Bäumen, die selbst wiederum aus genau einem Knoten bestehen.

```
simpleBaum p = Baum (mkpairZNode p) Leer Leer.
```

Hier sehen Sie einen Wald mit lauter kleinen Bäumen

Wenn wir die Liste `charList` von Zeichen als Eingabe nehmen, so produziert

```
map simpleBaum $ fquickSort snd (freqMap charList)
```

daraus einen Wald mit lauter kleinen Bäumen. Das ist der *Anfangswald*.

Der nächste Schritt

Wir iterieren jetzt über den Wald, also die geordnete Liste der Bäume, kombinieren die ersten beiden Bäume zu einem neuen Baum und fügen diesen Baum an seinen Platz ein:

```
mergeBaumList :: [Baum] -> [Baum]
mergeBaumList (x:x':xs) =  mergeBaumList newBaumList
         where
             newBaumList = weightInsert (mergeBaum x x') xs
mergeBaumList (x:[]) = [x]
```

Terminierung

Da jeder Schritt in diesem Programm die Liste um genau einen Baum vermindert, ist garantiert, daß wir eine terminierende Funktion haben.

Der endgültige Baum

Der Ausdruck

```
head (mergeBaumList littleBaums)
     where
      listOfFreq = fquickSort snd (freqMap charList)
      littleBaums = map simpleBaum listOfFreq
```

berechnet aus einer Liste *kleiner Bäume*, die selbst wieder aus einer Liste von Zeichen konstruiert wird, einen Baum. Es ist der Baum, den wir als erstes Element in der Liste finden.

Der letzte Schritt: Die Verschlüsselung.

Vorüberlegung

Nehmen wir an, daß wir den Pfad zu einem Knoten `kn` in einer Zeichenkette `pfad` aufgezeichnet haben. Dann sind zwei Fälle möglich:

- Entweder wir sind in einem Blatt, dann finden wir das Zeichen heraus, das in diesem Blatt sitzt, und geben es zusammen mit dem `pfad` zurück (aus Verträglichkeitsgründen *als Liste mit einem Element*).
- Falls wir dagegen in einem inneren Knoten sind, so trägt dieser innere Knoten zunächst nichts bei, und wir geben die leere Liste zurück.

Knotenbehandlung

```
recordNode :: t -> Node -> [(Char, t)]
recordNode x kn = if (isZNode kn) then [theChar kn x] else []
```

Das ist eine gleichförmige Repräsentation der Verschlüsselung für alle Knoten in dem Baum, so daß beim Baumdurchlauf selbst keine Fallunterscheidung notwendig ist.

DER BAUMDURCHLAUF

```
baumDurchlauf :: [Char] -> Baum -> [(Char, [Char])]
baumDurchlauf pfad (Baum wurzel linkerUnterB rechterUnterB) =
      encWurz ++ nachLinks ++ nachRechts
        where
          encWurz    = recordNode pfad wurzel
          nachLinks  = baumDurchlauf (pfad ++ "0") linkerUnterB
          nachRechts = baumDurchlauf (pfad ++ "1") rechterUnterB

baumDurchlauf pfad Leer = []
```

Als Parameter haben wir einen Pfad und einen Baum, der durch seine Wurzel und seinen linken und rechten Unterbaum gegeben ist.

Mit `recordNode` wird die Verschlüsselung der Wurzel berechnet. Dann wird rekursiv den Durchlauf für den linken und den rechten Unterbaum aufgerufen.

Wir merken uns, in welche Richtung wir gehen: Wir erweitern den Pfad durch Anhängen von `0`, wenn wir in den linken Unterbaum gehen, entsprechend wird der Pfad durch das Anhängen von `1` erweitert, wenn wir in den rechten Unterbaum gehen.

Ist der Baum leer, so geben wir die leere Liste zurück.

Die Resultate der Durchläufe werden mit ++ miteinander konkateniert, so daß wir als Resultat eine Liste erhalten.

Trommelwirbel

Und das ist die Verschlüsselung

```
encoding :: [Char] -> [(Char, [Char])]
encoding charList = baumDurchlauf [] dieserBaum
        where
            wieOft       = fquickSort snd (freqMap charList)
            kleineB      = map simpleBaum wieOft
            dieserBaum   = head (mergeBaumList kleineB)
```

Beispiel

```
>>> encoding "Das ist das Haus vom Nikolaus"
[('s',"00"),('v',"0100"),('m',"01010"),('t',"01011"),
('o',"0110"),('u',"0111"),('D',"10000"),('H',"10001"),
('i',"1001"),('k',"10100"),('l',"10101"),('N',"10110"),
('d',"10111"),('a',"110"),(' ',"111")]
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Haskell ist eine funktionale Programmiersprache. Sie arbeitet also nach dem Prinzip der referentiellen Transparenz: Wann immer ich einen Ausdruck verwende, er hat stets denselben Wert. Das ist bei der Ein- und Ausgabe offensichtlich nicht der Fall.

Dunque, che cosa facciamo?

Was also tun? Wir müssen uns also etwas überlegen.

Analogie: *call by value* vs. *call by refererence* in Java.

Code-Beispiel aus Java

```
public void kehrUm(int[] einFeld) {
        int i;
        for(i = 0; i < einFeld.length/2; i++) {
            int t = einFeld[i];
            einFeld[i] = einFeld[einFeld.length - 1 - i];
            einFeld[einFeld.length - 1 - i] = t;
        }
    }
```

Die Methode gibt bekanntlich ein Feld in umgekehrter Reihenfolge aus. Das geschieht, obgleich eine Java-Methode ihre Parameter nicht ändert.

Java arbeitet nicht mit den Daten selbst, sondern mit Referenzen auf diese Daten.

- Die referenzierten Daten ändern sich,
- Die Referenzen ändern sich nicht,
- Es werden jedoch lediglich die Referenzen übergeben.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Analogie

Wir konstruieren eine ähnliche Analogie bei der Ein- und Ausgabe in Haskell. Das Haskell-Programm schickt einen Umschlag an die Außenwelt. Ein- und Ausgaben für das Programm sollen nur über diesen Weg stattfinden.

Eingabe

Der Adressat legt etwas in den Umschlag hinein und gibt ihn an das Programm zurück. Es öffnet diesen Umschlag und entnimmt den Wert. Der Umschlag selbst wird durch diese Aktion nicht geändert.

Ausgabe

Das Programm legt seine Werte in den Umschlag und schickt den Umschlag in die weite Welt. Der Umschlag wird vom Adressaten entgegengenommen und geöffnet, der Wert entnommen und der Umschlag an das Programm zurückgegeben.

In jedem Fall bleibt der Umschlag unverändert. Beachten Sie: Er muß vor jeder Aktion geöffnet und nach jeder Aktion geschlossen werden.

Das erste Beispiel

```
main = do
     putStr "Bitte geben Sie eine Zeile ein:\n"
     zeile <- getLine
     putStr ("echo *"++ zeile ++ "*\n")

>>> main
Bitte geben Sie eine Zeile ein:
Dies ist eine Zeile
echo *Dies ist eine Zeile*
>>>
```

Was geschieht?

Die Funktion `putStr :: String -> IO ()` nimmt eine Zeichenkette und repräsentiert sie auf dem Bildschirm. Der Typ von `putStr` ist merkwürdig — das diskutieren wir gleich.

Nächste Zeile

`zeile <- getLine` tut Folgendes:

- Wir schicken unseren Umschlag mit Namen `getLine` in die Welt hinaus und erwarten eine Eingabe,
- die Umwelt legt eine Zeichenkette in unseren Umschlag,
- die Zeile `zeile <- getLine` öffnet den Umschlag, nimmt die Zeichenkette heraus und bindet den Namen `zeile` daran.

Typisierung

`getLine :: IO String`. Wieder dieses ulkige `IO`.

Das Ganze wird durch das Schlüsselwort `do` eingeleitet (kommt einem merkwürdig bekannt vor, aber es ist schon irgendwie anders) und an den Namen `main` gebunden.

IO-Aktionen

Eine `IO`-Aktion ist eine Aktion mit einem Seiteneffekt, etwa dem Lesen oder Drucken eines Wertes, verbunden mit einem typisierten Resultat.

Als Typ für derartige Aktionen haben wir bis jetzt `IO [Char]` oder `IO ()` kennengelernt. Eine derartige `IO`-Aktion ist darauf beschränkt, ein Resultat zurückzugeben, wie wir es bei den funktionalen Typen bislang gesehen haben.

Eine `IO`-Aktion hat ebenfalls einen *kontrollierten* Seiteneffekt. Das ist einigermaßen *un*funktional.

Mal sehen

```
>>>  x <- putStr "abcde"
abcde>>> x
()
>>> :type x
x :: ()
```

Die Eingabe ist die Zeichenkette `"abcde"`, die als Echo ausgegeben wird, das Resultat wird an den Namen `x` gebunden. Überraschende Typisierung von `x`. Was geschieht?

- Die Aktion wird durchgeführt, die Zeichenkette `abcde` wird auf dem Bildschirm wiedergegeben,
- wir öffnen den Umschlag und weisen dem Resultat dieser Aktion den Namen `x` zu,
- dieser Wert `x` ist gedruckt, er hat allerdings, wie es sich zeigt, keinen *besonderen* Wert,
- der Typ von `x` bestätigt das: Es handelt sich hier um den Typ `()`, der lediglich von einem einzigen Wert bewohnt wird, nämlich gerade von `()`.

Weil also eine Ausgabe-Aktion keinen besonders *distinguierten* Wert zurückgibt, können wir ihn ignorieren und statt `x <- putStr "abcde"` schreiben `putStr "abcde"`.

Zurück zu IO-Aktionen

Eine `IO`-Aktion dient einem zweifachen Zweck:

1. eine Aktion durchzuführen,
2. einen Wert zurückzugeben.

Die Aktion wird ausgeführt, indem eine Funktion mit den passenden Argumenten ausgeführt wird. Das Ergebnis wird dann durch `<-` zugewiesen.

Also

```
>>> x <- getLine
Dies ist eine Aktion
>>> x
"Dies ist eine Aktion"
```

Die Aktion `getLine` wird ausgeführt, indem sie die Benutzereingabe `Dies ist eine Aktion` erhält. Das Resultat wird an den Namen `x` gebunden.

Q

Kann man nicht einfach den Namen `x` mit `let` an `getLine` binden?

A

Klar:

```
>>> let x = getLine
>>> x
Dies ist eine Zeichenkette
"Dies ist eine Zeichenkette"
>>> :type x
x :: IO String
```

Der Name `x` wird der gesamten `IO`-Aktion zugewiesen. Daher ist `x` eine Aktion, die ausgeführt wird. Das Beispiel zeigt, wie dies geschieht. Die Funktion `getLine` gibt eine Zeichenkette als Wert zurück.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Q

Kann ich nicht einfach schreiben `"Diese Zeile wird eingegeben: "++ getLine`? Denn `getLine` gibt eine Zeichenkette als Wert zurück, wie wir gesehen haben.

A

Mal sehen

```
>>> "Dies" ++ getLine

<interactive>:1:10:
    Couldn't match expected type '[Char]'
         against inferred type 'IO String'
    In the second argument of '(++)', namely 'getLine'
    In the expression: "Dies" ++ getLine
```

Oj vej!

Die Typisierung macht Probleme.

Typisierung

Die Typen sind unverträglich:

- `"Diese Zeile wird eingegeben: "` ist vom Typ `[Char]`,
- `getLine` hat den Typ `IO [Char]`.

*

Gruppierung

Ein- und Ausgabe-Aktionen können aufeinander folgen. Also ist es sinnvoll, diese sequentielle Komposition auch sprachlich auszudrücken. Hier kommt das Schlüsselwort `do` ins Spiel. `IO`-Aktionen können durch einen do-Block gruppiert werden, der Block wird dann an einen Namen gebunden.

Anmerkung

Ich habe im ersten Beispiel den Namen `main` gewählt mit `main :: IO ()`. Es hätte statt `main` auch jeder andere legale Name sein können. Die Wahl des Namens hat für unseren Kontext keine Bedeutung (das ändert sich, wenn Haskell-Programme kompiliert werden).

`IO`-Aktionen werden in einem `do`-Block gruppiert.

Die letzte Aktion in einem solchen Block kann nicht an einen Namen gebunden werden.

Der Typ eines `do`-Blocks ist der Typ der letzten Aktion, die sich in ihm befindet.

Werte werden innerhalb eines `do`-Blocks durch `let` an Namen gebunden.

Funktionen können wie üblich innerhalb eines `do`-Blocks aufgerufen werden.

Die Sendung mit der Maus

Wer, wie, was, wieso, weshalb, warum? Das folgt aus allgemeinen Regeln für Monaden. Wir nehmen die Regeln im Augenblick einfach zur Kenntnis.

let-Umgebung

```
import Data.Char
-- da wohnt toUpper
yo :: IO ()
yo = do
     putStr "Bitte geben Sie eine Zeile ein:\n"
     zeile <- getLine
     let grossZeile = map toUpper zeile
         umgekehrt = reverse grossZeile
     putStr ("echo *"++ zeile ++ "|" ++
              grossZeile ++ "?" ++ umgekehrt ++ "*\n")
```

Ich habe als Namen für den do-Block nicht `main`, sondern `yo` gewählt. Damit rufe ich die Funktion dann auch auf.

```
>>> yo
Bitte geben Sie eine Zeile ein:
abcdefg
echo *abcdefg|ABCDEFG?GFEDCBA*
```

Iteration

Wir holen uns eine Zeile als Eingabe. Falls diese Zeile leer ist, wird die Funktion `return()` aufgerufen, sonst wird die Zeile als Ausgabe zurückgegeben. Dann wird die Aktion `yo` wieder aufgerufen. Sie ist rekursiv (Teufel auch).

```
yo :: IO ()
yo = do
     putStr "Bitte geben Sie eine Zeile ein:\n"
     zeile <- getLine
     if null zeile
        then  return()
        else do
             putStrLn ("echo *"++ zeile ++ "*")
             yo
```

Typisierung

Die Funktion `yo` hat den Typ `yo :: IO ()`. Was tut `return()` eigentlich?

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden


```
>>> yo
Bitte geben Sie eine Zeile ein:
abcdef
echo *abcdef*
Bitte geben Sie eine Zeile ein:
1234
echo *1234*
Bitte geben Sie eine Zeile ein:

>>>
```

Was tut `return()` eigentlich?

Es gibt nicht den Kontrollfluß an den Aufrufer zurück (wie in Java oder C++).

Die Funktion return ist eine IO-Aktion konvers zur Funktion `<-`.

1. `a <- getLine` öffnet den Briefumschlag und und bindet `a` an dessen Inhalt.
2. `return a` steckt etwas in den Briefumschlag.

`return` ist eine IO-Aktion. Der Typ von `return ()` ist `IO ()`, der Typ von `return "abc"` ist `IO [Char]`.

Jetzt sehen Sie sich das an:

```
>>> a <- return "abc"
>>> a
"abc"
```

Der Aufruf `return "abc"` konvertiert die Zeichenkette `"abc"` in eine `IO [Char]`-Aktion. Dann bindet der Ausdruck `a <- return "abc"` den Namen `a` an den Inhalt dieses Briefumschlags, also an die Zeichenkette `"abc"`.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Noch einmal yo

Da gab es diese Anweisung:

```
if null zeile
   then  return()
   else do
        putStrLn (" ... ")
        yo
```

Der alternative Zweig der bedingten Anweisung ist vom Typ `IO ()`, also muß der `then`-Zweig dieses Ausdrucks ebenfalls diesen Typ haben, denn bedingte Anweisungen fordern denselben Typ für beide Zweige. Daher ist es *aus Typisierungsgesichtspunkten zwingend erforderlich*, eine Aktion wie `return()` einzufügen.

Wir werden `return` später noch begegnen, wenn es um Monaden geht[a].

[a] Schon wieder dieses merkwürdige Wort.

Leichte Modifikationen

Wir fragen den Benutzer um einen Schlüssel, dann soll der Benutzer die zu verschlüsselnden Daten, also den Text, Zeile für Zeile eingeben. Das Programm gibt dann die verschlüsselten Daten aus.

Wir hatten bei der Diskussion der Verschlüsselung das Leerzeichen durch den Stern ersetzt, das ist bei der interaktiven Ein- und Ausgabe nicht besonders praktisch. Deshalb machen wir diese Änderung rückgängig und berechnen die Tabelle entsprechend neu.

Der Schlüssel ist nun ein Parameter. Also adaptieren wir die Funktionen minimal, indem wir für die Funktionen `vignere` und `code` sowie `uncode` den Schlüssel als neuen Parameter einführen:

```
vignere :: (a -> Char -> c) -> [Char] -> [a] -> [c]
code :: [Char] -> [Char] -> [Char]
code key = vignere encode key
uncode key = vignere decode key
```

Eingabe in zwei Teilen

Im ersten Schritt erhalten wir den Schlüssel, dann iterieren wir im zweiten Schritt über die Eingabe, bis alles verarbeitet und verschlüsselt ist. Die Leerzeile dient als Ende der Eingabe.

Den Schlüssel erhalten wir so:

```
getKey :: IO ()
getKey = do
     putStrLn "Schluessel?"
     dieserSch <- getLine
     eingabeSchleife dieserSch
```

Wir fragen also mit `putStrLn "Schluessel?"` nach dem Schlüssel, lesen die nächste Zeile und binden diese Zeichenkette an die Variable `dieserSch`. Dann rufen wir eine Funktion auf, deren Aufgabe die Verarbeitung der restlichen Eingaben ist.

Die Typisierung der Funktion `getKey` ist `getKey :: IO ()`, falls `eingabeSchleife` diesen Typ hat.

Eingabeschleife

Wir nehmen eine Eingabe und überprüfen, ob wir das Ende der Eingabe erreicht haben. Ist das der Fall, führen wir `return()` aus, im anderen Fall wird die Zeile verschlüsselt und auf dem Bildschirm dargestellt. Dann geht das Spiel weiter.

Also:

```
eingabeSchleife :: [Char] -> IO ()
eingabeSchleife schl = do
   putStrLn "> "
   zeile <- getLine
   if null line then return()
      else do
            putStrLn $ code schl zeile
            eingabeSchleife schl
```

Auch im alternativen Zweig der bedingten Anweisung ist ein do-Block zu finden. Das ist nötig, weil sonst der Ausdruck in diesem Zweig keinen Wert vom Typ `IO()` zurückgeben würde.

Ein kurzer Ausflug zum Thema *Dateien*.

Dateien residieren bekanntlich im Dateisystem des Computers. Sie können geöffnet werden, man kann von ihnen lesen oder auf sie schreiben, sie können dann auch wieder geschlossen werden. Eine Datei wird durch einen Namen wie etwa `einText.txt` identifiziert und möglicherweise durch einen Zugriffspfad, dessen konkrete Syntax sich am Dateisystem des Rechners orientiert. Wir ignorieren den Pfad und identifizieren eine Datei kurzerhand mit ihren Namen; das ist z.B. dann der Fall, wenn die Datei im gegenwärtigen Arbeitsverzeichnis verfügbar ist.

IOMode

Dateien können auf verschiedene Arten behandelt werden, das wird in Haskell durch den diskreten Typ `IOMode` beschrieben, dessen Mitglieder als Zugriffsspezifikationen dienen:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Eine Datei wird geöffnet, indem ihr Name und eine Zugriffsspezifikation angegeben wird. Eine Datei, die mit `WriteMode` geöffnet wird, verliert ihren bisherigen Inhalt, an eine mit `AppendMode` geöffnete Datei werden Inhalte angefügt.

Haskell nimmt diese Daten und produziert daraus ein *Datei-Handle*, mit dessen Hilfe auf die Datei zugegriffen wird: Der Benutzer hat natürlich keinen direkten Zugriff auf die Datei, sondern muß sich des Handle bedienen, um mit ihr zu arbeiten. Wenn man mit der Datei fertig ist, wird die Datei geschlossen, auch das geschieht durch das Handle.

Durch `openFile "einText.txt" ReadMode` wird die Datei `einText.txt` zum Lesen geöffnet. Der Typ des Handle, also des Rückgabewerts, ist `IO Handle`, der Typ der Funktion `openFile` ist

```
>>> :type openFile
openFile :: FilePath -> IOMode -> IO Handle
```

Details

`FilePath` ein Typ-Synonym für `String`. `IO Handle` ist der Typ, der mit dem Handle assoziiert wird. Wir importieren diese Operationen aus dem Modul `System.IO`.

hClose

Wir schließen die Datei, sobald wir mit ihr fertig sind, dies geschieht durch die Funktion `hClose :: Handle -> IO ()`. Der Typ des Rückgabewerts ist `IO()`, ist also eine Ein- und Ausgabe-Aktion.

hGetContents

Der Inhalt einer Datei wird durch die Funktion `hGetContents` zugänglich gemacht. Sie hat die Signatur `Handle -> IO String`, die Funktion nimmt also ein Handle und gibt eine `IO`-Aktion vom Typ `String` zurück. Mit dieser Aktion kann dann weitergearbeitet werden.

Beispiel

Das steht in der Datei `io-file.hs`

```
import System.IO
main :: IO ()
main = do
  putStrLn "Dateiname?"
  fName <- getLine
  handle <- openFile fName ReadMode
  inhalt <- hGetContents handle
  putStr inhalt
  hClose handle
```

Ausführung von `main`:

```
>>> main
Dateiname?
io-file.hs
import System.IO
main = do
  putStrLn "Dateiname?"
  ...
  hClose handle
>>>
```

Getreu der Philosophie von Haskell wird die gesamte Zeichenkette, die hier zugewiesen wird, als *verzögert ausgewertete* Zeichenkette behandelt. Der Inhalt wird erst dann zugänglich gemacht, wenn er wirklich benötigt wird.

Modifikation

Die zu verschlüsselnde Eingabe wird einer Eingabe-Datei entnommen. Wir benötigen vom Benutzer daher den Namen der Datei und den Schlüssel.

```
schluesselUndDatei :: IO ()
schluesselUndDatei = do
           putStrLn "Schlüssel und Eingabedatei?"
           antwort <- getLine
           let schl:datei:_ = words antwort
           doCoding schl datei
```

Die Eingabe des Benutzers wird in einer Zeichenkette gespeichert, diese Zeichenkette wird durch die Funktion `words` in eine Liste von Einzelwörtern, also wieder von Zeichenketten, zerlegt. Das erste Element dieser Liste ist der Schlüssel, das zweite Element ist die Datei, aus der wir lesen.

Arbeitspferd

Die eigentliche Arbeit wird in dieser `IO`-Aktion geleistet. Aufruf mit dem Schlüssel und der zu öffnenden Datei.

```
doCoding :: [Char] -> FilePath -> IO ()
doCoding k dat = do
         putStrLn dat
         handle <- openFile dat ReadMode
         inhalt <- hGetContents handle
         putStrLn $ code k inhalt
         hClose handle
```

Bemerkenswert

Der *gesamte Inhalt der Datei* wird in der Zeichenkette `inhalt` abgespeichert wird. Dann lassen wir die Funktion zur Verschlüsselung über diese Zeichenkette laufen. Also wird ein Zeichen erst dann präsentiert, wenn es wirklich benötigt wird. Grund: Wir arbeiten mit der verzögerten Auswertung der Zeichenkette.

Die Version oben schreibt das Ergebnis auf den Bildschirm.

Milde Erweiterung

Wenn in eine Datei geschrieben werden soll, sieht's so aus:

```
doCoding k dat = do
         handle <- openFile dat ReadMode
         let fOut = "XX_" ++ dat
         ohandle <- openFile fOut WriteMode
         inhalt <- hGetContents handle
         hPutStr ohandle $ code k inhalt
         hClose handle
         hClose ohandle
```

Wir erzeugen einen neuen Namen für die Datei, der durch `let` an einen Bezeichner gebunden wird. Er dient dann dazu, das Handle für die Ausgabe-Datei zu berechnen.

Anmerkung

Die Vorgehensweise ist ziemlich kanonisch und wiederholt sich nach demselben Muster: Wir öffnen eine Datei zum Lesen, weisen den Inhalt der Datei zur verzögerten Auswertung an eine Zeichenkette zu und schließen dann die Datei.

Die Funktion `readFile` erledigt hinter den Kulissen die Aktionen, wie etwa das Öffnen der Datei zum Lesen und die Zuweisung an ein Datei-Handle.

Analog: Funktion `writeFile`.

Kompaktifiziert

```
doCoding k dat = do
          inhalt <- readFile dat
          let fOut = "XX_" ++ dat
          writeFile fOut (code k inhalt)
```

Zack!

Problem

Ich möchte komplexe Daten in eine Datei schreiben und sie auch wieder von dort einlesen und verwenden.

Idee

Mit `show` kann eine Zeichenkette erzeugt werden (falls der zugrundeliegende Datentyp das zuläßt) und diese Zeichenkette in eine Datei geschrieben werden. Der Datentyp muß also Mitglied der Typklasse `Show` sein.
Diese Zeichenkette kann dann wieder eingelesen werden, die resultierende Zeichenkette wird dann in eine Instanz des betreffenden Typs verwandelt.
Hierzu wird die Funktion `read` herangezogen, die zur Verfügung steht, sofern der Datentyp ein Mitglied der Typklasse `Read` ist.

Read-Show-Partnerschaft

Partnerschaft: `read` analysiert die von `show` erzeugte Zeichenkette syntaktisch, also im Hinblick auf die Syntax des entsprechenden Datentyps und erzeugt eine entsprechende Instanz des Typs.

Beispiel: binäre Bäume.

Erinnerung

```
data Baum a = Knoten a (Baum a) (Baum a)
     | Leer
     deriving (Show,Read)
```

Ergänzung

Die Mitgliedschaft in der Typklasse `Read` wird hinzugefügt. Sonst bleibt alles beim Alten.

Einfügen und Testen

```
baumEinf ::  (Ord a) => a -> Baum a -> Baum a
baumSuche :: (Ord a) => a -> Baum a -> Bool
```

Wie oben definiert.

Ein Baum zum Spielen

```
let bm = foldr baumEinf Leer
          (words "Das ist das Haus vom Nikolaus")
```

Dann: `bm :: Baum (String)` (also ein Mitglied der Typklasse `Show`).

```
>>> let q = show bm
>>> q
   "Knoten \"Nikolaus\" (Knoten \"Haus\"
   (Knoten \"Das\" Leer Leer) Leer)
   (Knoten \"vom\" (Knoten \"das\" Leer
   (Knoten \"ist\" Leer Leer)) Leer)"
```

Q

Kann ich aus q den Baum rekonstruieren?

A

Probieren wir's halt:

```
>>> read q

<interactive>:1:0:
    Ambiguous type variable ‘a’ in the constraint:
      ‘Read a’ arising from a use of ‘read’ at <interactive>:1:0-5

      Probable fix: add a type signature
                    that fixes these type variable(s)
```

Das ist ziemlich einleuchtend, denn man kann schließlich nicht erwarten, daß eine Zeichenkette so ohne Weiteres in einen Baum verwandelt wird.

Es gibt Hoffnung: `Probable fix: add a type signature that fixes these type variable(s)`.

Das tun wir jetzt

```
>>> read q::(Baum String)
Knoten "Nikolaus" (Knoten "Haus" (Knoten "Das" Leer Leer) Leer)
(Knoten "vom" (Knoten "das" Leer (Knoten "ist" Leer Leer)) Leer)
```

Jubel!

Mal sehen

Ich füge die Zeichenkette "otto" in den Baum ein:

```
>>> baumEinf "otto" (read q::(Baum String))
Knoten "Nikolaus" (Knoten "Haus" (Knoten "Das" Leer Leer) Leer)
(Knoten "vom" (Knoten "das" Leer
(Knoten "ist" Leer (Knoten "otto" Leer Leer))) Leer)
```

Noch mehr Jubel!

Warnung

Wir haben also aus der Zeichenkette einen binären Baum erzeugt. Das geht natürlich nur, wenn die Zeichenketten selbst vorher mittels `show` aus einem Baum hergestellt worden sind.

Versuch

```
>>> read "da da"::(Baum String)
*** Exception: Prelude.read: no parse
```

Wenn wir also versuchen, eine beliebige Zeichenkette in einen binären Baum zu verwandeln, so scheitern wir. Die Fehlermeldung sagt im wesentlichen, daß die übergebene Zeichenkette syntaktisch nicht als Instanz von `Baum String` identifiziert werden konnte.

Komplexe Daten in Dateien schreiben

Für unsere Bäume geht das so:

```
schreiben baum datei = do writeFile datei (show baum)
```

`schreiben :: (Show a) => a -> FilePath -> IO ()` hat zwei Argumente: Das Datum, das geschrieben werden soll und der Name der Datei. Das zu schreibende Datum muß der Typklasse `Show` angehören. Als Ergebnis des Funktionsaufrufs bekommen wir ein Resultat vom Typ `IO ()`, es wird also eine `IO`-Aktion ausgeführt.

Die Funktion ruft `writeFile` auf, als zu schreibende Zeichenkette wird die Repräsentation `show baum` des Arguments als Zeichenkette herangezogen. In unserem Fall enthält nach dem Aufruf `schreiben bm "aus.txt"` die Datei `"aus.txt"` die Repräsentation des Baums, den wir unter `bm` abgespeichert haben, als Zeichenkette.

Kommt das Wort "Bologna" in dem binären Suchbaum vorkommt, den wir in der Datei "aus.txt" abgespeichert haben?

```
testen :: String -> FilePath -> IO ()
testen text datei = do
   inhalt <- readFile datei
      let istDa = baumSuche text (read inhalt::(Baum String))
   putStrLn (show istDa)
```

Die Antwort ist `testen "Bologna" "aus.txt"`

Ich möchte einen binären Suchbaum aus einem Text, der in einer Datei `einDat` gespeichert ist, erzeugen. Dann soll der Baum zur späteren Verwendung in der Datei `ausDat` abgespeichert werden.

```
einAus :: FilePath -> FilePath -> IO ()
einAus einDat ausDat = do
   inhalt <- readFile einDat
   writeFile ausDat (show (foldr baumEinf Leer (words (inhalt))))
```

Wir speichern also den Inhalt der Eingabedatei in der Zeichenkette `inhalt` ab, zerlegen diese Zeichenkette in einzelne Wörter, die wir in einen binären Suchbaum einfügen, und speichern die Darstellung des Suchbaums als Zeichenkette in der Ausgabedatei ab.

Hier ist hilfreich, daß `Haskell` nicht unbedingt die Datei vollständig als Zeichenkette einliest, sondern *verzögert* reagiert und nur soviel einliest wie nötig.

Beispiel

Zeichenkette: Text des Romans *Krieg und Frieden* von L. Tolstoi in der englischen Fassung aus *Project Gutenberg*. Die Datei `WarAndPeace.txt` enthält 562.436 Wörter in 64.940 Zeilen (insgesamt 3.269.017 Zeichen), ist also nicht klein. Wir speichern den binären Suchbaum in der Datei `WP.txt`.

```
>>> :set +s
>>> einAus "WarAndPeace.txt" "WP.txt"
(17.48 secs, 1980238448 bytes)
```

Kommen die Zeichenketten `"Obama"` oder `"Natasha"` darin vor?

Hm, mal sehen:

```
>>> testen "Obama" "WP.txt"
False
>>> testen "Natasha" "WP.txt"
True
```

Vorsicht

Die hier diskutierte Vorgehensweise arbeitet nur, wenn Sie show und read nicht modifiziert haben.

Zustände sind das

Die Arbeit mit Funktionen ist zustandsfrei. Das ist für die Programmierung nicht angemessen. Wir haben das bei der Ein- und Ausgabe gesehen. Dort wurde implizit ein Programmzustand eingeführt und manipuliert: Klar, abhängig von der Eingabe ändert sich der Zustand des Programms. Bei der Ausgabe wurde keine Zustandsänderung festgestellt (Zeuge: `IO ()`).

Wir werden uns i.f. mit zustandsorientiertem Arbeiten befassen.

Q

Warum führen wir nicht explizit Zustände ein, die wir dann funktional manipulieren können?

A

Au weia! Dann müßten wir stets eine Zustandskomponente *erfinden* und *manipulieren*. Das führt zu Programmen, die man nicht mehr versteht (es können sehr viele Zustände nötig sein) und die man daher nicht mehr warten, erweitern, debuggen etc. kann.

Monaden

Monaden sind das mathematische Werkzeug dazu.

S. Mac Lane, *Categories for the Working Mathematician*, Graduate Texts in Mathematics 5, Springer, ^{2}1997; Kap. VI.
E. Moggi, *Notions of computations and monads*, Inf. Comput. 93, 55 – 92, 1991.

Wow!

Beeindruckend. Vergessen Sie's!

Wir werden Monaden als Typklasse definieren, uns einige Beispiele und Eigenschaften ansehen und nach dem **Klimbim**-Prinzip von Ingrid Steeger agieren: *Dann mach' ich mir 'nen Schlitz ins Kleid und find' es wunderbar*.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Definition

Eine Monade wird durch einen polymorphen Typ-Konstruktor mit diesen Eigenschaften definiert:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
```

Wir stellen wir uns eine Monade vom Typ `m a` als Hülle um Elemente des Typs `a` vor (Beispiel: `Maybe a`).

Die Funktion `return` mit der Signatur `a -> m a` bettet einen Wert vom Typ `a` in diese Hülle ein (`x` $\mapsto$ `Just x`).

Die Bezeichnung der Funktion ist nicht gerade glücklich; `return` hat nichts mit dem Kontrollfluß zu tun!

bind

Die Funktion >>= wird auch gerne *bind* genannt: Sie *bind*et eine Berechnung, die durch die Funktion gegeben ist, in den Kontext der Monade ein.

bind

Die Funktion >>= ist linksassoziativ und hat die sehr niedrige Priorität 1 (zum Vergleich: Der Applikationsoperator $ hat die niedrigste Priorität 0).

bind, Arbeitsweise

Mit einer Instanz des Typs `m a` und einer Funktion der Signatur `a -> m b` wird ein Resultat vom Typ `m b` produziert.

Idee zu *bind*

```
(>>=)  :: m a -> (a -> m b) -> m b
```

In der Monade wird eine Berechnung durchgeführt, die ein Resultat vom Typ `m a` hat. Dieser Wert wird dann an die Funktion `f` weitergereicht, die jedoch ein Argument vom Typ `a` erwartet. Also muß `f` in den Kontext der Monade einge*bund*en werden, um das gewünschte Resultat zu erzielen.

Weitere Funktionen

Wir finden die beiden Funktionen >> und `fail`. Für sie ist eine Implementation vorgesehen, falls der Nutzer keine andere Definition vornimmt.

```
m >> k = m >>= \_ -> k
fail s = error s
```

error

Die polymorphe Funktion `error :: String -> a` hat ebenfalls eine *default*-Definition. Sie dient zur Behandlung von Ausnahmen und führt zum Programmabbruch.

... und weiter

```
m >> k = m >>= \_ -> k
```

Bei der Funktion >> (mit `(>>) :: m a -> m b -> m b`) wird in der *default*-Implementierung das Resultat in der Monade `m a` ignoriert. Die Berechnung wird durchgeführt, darauf folgt die Berechnung, die durch die Funktion `k` beschrieben wird.

Anmerkung

Es ist keine Funktion vorgesehen, die aus einer Monade **heraus**führt. Aus einer Monade heraus kommt man (in der Kategorientheorie) mit *Eilenberg-Moore-Algebren*, die zu klassifizieren jedoch nur in seltenen Fällen vollständig gelingt. Es ist eine arg schweißtreibende Angelegenheit.

Beispiel

Wir haben drei Abbildungen:

1. Namen von Studenten $\mapsto$ Matrikelnummer,
2. Matrikelnummer $\mapsto$ Lehrveranstaltung,
3. Lehrveranstaltung $\mapsto$ Hörsaal, in dem eine Klausur geschrieben wird.

Ich möchte jetzt gerne wissen, in welchem Hörsaal sich ein Student befindet.

Vorgehensweise

Ich besorge mir die *Matrikelnummer* des Studenten, damit kann ich die *Vorlesung* ermitteln, schließlich kann ich mit Hilfe der Vorlesung den entsprechenden *Hörsaal* herausfinden.

Fehlermöglichkeiten

Adressiere einen Studenten, dessen Matrikelnummer nicht eingetragen ist; stoße auf eine Vorlesung, für die kein Hörsaal existiert etc.

Kurz: Map

Für den Datentyp `Map` ist die Funktion `lookup`-vordefiniert. Sie gibt entweder ein Ergebnis der Form `Just y` zurückgibt (wenn `y` im Wertebereich der Abbildung zu finden ist), oder `Nothing`, wenn wir einen Wert haben wollen, für den in der Abbildung kein Bild vorhanden ist.

```
:type Map.lookup
Map.lookup :: Ord k => k -> Map.Map k a -> Maybe a
```

Erinnerung

`Map` muß aus dem Modul `Data.Map` importiert werden.

Jetzt aber

Die Funktion `finde` muß auf jedem Schritt (Namen von Studenten $\mapsto$ Matrikelnummer, Matrikelnummer $\mapsto$ Lehrveranstaltung, Lehrveranstaltung $\mapsto$ Hörsaal) berücksichtigen, daß die entsprechende Abbildung für das Argument nicht definiert ist.

Allora

```
finde x a b c =
  case Map.lookup x a of
    Nothing -> Nothing
    Just s ->
      case Map.lookup s b of
        Nothing -> Nothing
        Just t ->
            Map.lookup t c
```

Wir suchen den Namen von `x` in der Abbildung `a`, falls wir dort nicht fündig werden, geben wir `Nothing` zurück. Falls wir den zum Namen gehörenden Wert finden (`Just s`), suchen wir den zu `s` in der Abbildung `b` gehörigen Wert, falls wir nichts finden, geben wir `Nothing` zurück, falls wir jedoch etwas finden (also `Just t`), schlagen wir `t` in der Abbildung `c` nach und geben das entsprechende Ergebnis zurück.

Rollen

1. `x`: Name des Studenten
2. `a`: Namen von Studenten $\mapsto$ Matrikelnummer
3. `b`: Matrikelnummer $\mapsto$ Lehrveranstaltung
4. `c`: Lehrveranstaltung $\mapsto$ Hörsaal

Abbildungen

```
nam2Numb =
    Map.fromList [("Alf", 1), ("Bea", 2), ("Yo",4), ("Mia", 5) ]
numb2Kurs =
    Map.fromList [(1, "DAP"), (2, "DAP"), (3, "DAP"), (4, "RS")]
kurs2HS =
    Map.fromList [("DAP", "OH-14"), ("RS", "Audimax")]
```

Resultat

```
>>> finde "Alf" nam2Numb numb2Kurs kurs2HS
Just "OH-14"
>>> finde "Yo"  nam2Numb numb2Kurs kurs2HS
Just "Audimax"
>>> finde "Bea" nam2Numb numb2Kurs kurs2HS
Just "OH-14"
>>> finde "Max" nam2Numb numb2Kurs kurs2HS
Nothing
```

Kritik

Diese kaskadierende Lösung ist nicht besonders gut. Sie unterscheidet in jedem Fall Erfolg und Mißerfolg explizit. Besser wäre es, eine Art Pipeline aufzubauen, entlang derer die entsprechenden Werte fließen können (dazu gehört auch der Mißerfolg).

Wir machen `Maybe` zur Monade:

```
instance Monad Maybe where
   return x       = Just x
   Nothing >>= _  = Nothing
   Just x  >>= f  = f x
```

Die Funktionen `return` und `>>=` müssen definiert werden.

1. `return` bettet ein.
2. `>>=` leitet weiter.

Achtung

`>>=` hat in unserem Falle die Signatur `Maybe a -> (a -> Maybe b) -> Maybe b`.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Modifikation von finde

```
finde1 x a b c =
    (Map.lookup x a)           >>=
    (\y -> Map.lookup y b) >>=
    (\z -> Map.lookup z c)
```

Achten Sie darauf, daß der zweite Operator von >>= eine Funktion sein muß. Also modifizieren wir `Map.lookup` entsprechend.

Das sieht doof aus

Die Funktion >>= sieht nicht besonders lesbar aus. Außerdem ist es manchmal ganz schön, wenn wir Zwischenergebnisse der Pipeline auch mit Namen ansprechen können. Hierzu gibt es den syntaktischen Zucker <-.

Zweite Modifikation von `finde`

```
finde2 x a b c =
    do
      y <- Map.lookup x a
      z <- Map.lookup y b
      Map.lookup z c
```

Komisch: Jetzt tauchen diese ulkigen Gesellen `do` und `<-` auch wieder auf („Naht ihr euch wieder, schwankende Gestalten?“).

Intuitiv

- `y` nimmt den Wert, der beim Nachsehen von `x` in der Abbildung `a` entstanden ist, auf.
- Dieser Wert wird dann zum Nachsehen in der Abbildung `b` benutzt.
- Das führt schließlich zu `Map.lookup z c`, das dann als Wert zurückgegeben wird.

Ganz offensichtlich ist z.B. der Wert `y` kein Wert in der Monade. Er ist vielmehr ein Wert des zugrunde liegenden Datentyps: Sonst würde der Aufruf `Map.lookup y b` bereits an der Typisierung scheitern.

Genauer

Was hat es mit `do` und mit dem Pfeil `<-` auf sich?

Das ist zunächst syntaktischer Zucker, damit die Konstruktionen besser verdaulich sich. Es handelt sich also um Transformationen. Die Regeln werden jetzt angegeben und diskutiert.

①

Das Schema

```
do {x <- ausdruck; weiter}
```

wird übersetzt in

```
ausdruck >>= (\x -> do weiter)
```

②

Das Schema

```
do {ausdruck; weiter}
```

wird übersetzt in

```
ausdruck >>= do weiter
```

③

Das Schema

`do {let deklList; weiter}`

wird übersetzt in

```
let deklList
in do weiter
```

④

`do expression` ist gleichwertig zu `expression`.

⑤

`p >> q` wird übersetzt in
`p >>= (\_ -> q)`.

⑥

Das Schema `do {p; q}` wird übersetzt in `do {_ <- p; q}`

Anmerkung

`let` erlaubt Bindung durch Mustererkennung:
`let (x:_) = [1 .. 4]` ergibt `x == 1`.
`let (_:b) = [1 .. 4]` ergibt `x == [ 2, 3, 4]`.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden


```
    do
      y <- Map.lookup x a
      z <- Map.lookup y b
      Map.lookup z c
```

wird mit Regel ① transformiert in

```
(Map.lookup x a) >>=
        (\y -> do {z <- Map.lookup y b; Map.lookup z c})
```

Das wird mit ① transformiert in

```
(Map.lookup x a) >>=
  ((\y -> Map.lookup y b) >>=
       (\z -> Map.lookup z c))
```

Also in (>>= ist linksassoziativ)

```
(Map.lookup x a)         >>=
(\y -> Map.lookup y b)  >>=
(\z -> Map.lookup z c)
```

Jubel!

Das ist `finde1`.

ZUR ERINNERUNG

Monaden werden durch diese Klasse definiert

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
  (>>)   :: m a -> m b -> m b
  fail   :: String -> m a
```

BEISPIEL

Die MayBe-Monade mit

```
instance Monad Maybe where
   return x = Just x
   Nothing >>= _ = Nothing
   Just x >>= f = f x
```

Auch populär

Die Listenmonade. Was müssen wir tun? In der Beitrittserklärung müssen wir sagen, wie die Operationen >>= und `return` definiert sind.

Auf geht's

```
instance Monad [] where
    return t = [t]
    x >>= f  = concat (map f x)
```

return

Die Einbettung eines Elements vom Typ a in die Monade `[] a` (umständlich geschrieben) ist die einelementige Liste.

bind

`x >>= f` schickt die Funktion `f` über die Liste `x` und konkateniert die Ergebnisse.
Falls `f:: a -> [b]` eine Funktion und `[x]` eine Liste von Elementen von `a` ist, so ist `[f y | y <- x]` vom Typ `[[b]]`, also `concat [f y | y <- x]` vom Typ `[b]`.

Einfaches Beispiel

```
>>> q = (\w -> [0 .. w])
>>> [0 .. 2] >>= q
[0,0,1,0,1,2]
```

Au weia! Wie kommt das denn wohl zustande?

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Mal sehen

Die Definition von >>= erfordert die Berechnung von

```
concat (map q [0 .. 2])
== concat [(q 0), (q 1), (q 2)]
== concat [[0], [0, 1], [0, 1, 2]]
== [0,0,1,0,1,2].
```

Die Funktion `q` wird auf jedes Element von `[0 .. 2]` angewandt. Das Ergebnis, eine Liste von Listen, wird durch die Funktion `concat` in eine flache Liste transformiert. Das Argument für die Funktion wird einer Liste entnommen, die Berechnung selbst führt nicht zu einem einzigen Ergebnis, wohl aber zu einer Liste von Resultaten.

Anmerkung

Diese Monade wird daher gern zur Modellierung nicht-deterministischer Ergebnisse herangezogen.

Formulierung mit einem `do`-Block? Allgemein ist das ja leichter zu lesen, sagt man.

```
>>> do
 x <- [0 .. 2]
 [0 .. x]
```

Das ergibt auch das Ergebnis `[0,0,1,0,1,2]`.

Aber wieso?

```
do {x <- [0 .. 2]; [0 .. x]}
==  [0 .. 2] >>= (\x -> do [0 .. x])
==  [0 .. 2] >>= (\x -> [0 .. x])
==  concat (map (\x -> [0 .. x]) [0 .. 2])
==  concat [[0], [0, 1], [0, 1, 2]]
==  [0,0,1,0,1,2].
```

Aha!

Noch'n Beispiel

```
>>> do
  x <- [1 .. 3]
  y <- ['a' .. 'c']
  [(x, y)]
```

Äquivalent:

```
>>> do
  x <- [1 .. 3]
  y <- ['a' .. 'c']
  return (x, y)
```

(mit `return`)

Ergebnis

```
[(1,'a'),(1,'b'),(1,'c'),(2,'a'),(2,'b'),(2,'c'),
 (3,'a'),(3,'b'),(3,'c')]
```

Man kann sich schon denken, daß hier alle möglichen Kombinationen der beiden Listen berechnet werden soll, wie in

`[(x, y) | x <- [1 .. 3], y <- ['a' .. 'c']].`

Das gibt's einen Zusammenhang.

Expansion

Die Anwendung unserer Transformationsregeln ergibt

```
do {x <- [1 .. 3]; y <- ['a' .. 'c']; [(x, y)]}
==  [1 .. 3] >>= (\x -> do {y <- ['a' .. 'c']; [(x, y)]})
==  [1 .. 3] >>= (\x -> (['a' .. 'c'] >>= (\y -> do [(x, y)])))
==  [1 .. 3] >>= (\x -> (['a' .. 'c'] >>= (\y -> [(x, y)])))
==  [1 .. 3] >>= (\x -> concat(map(\y ->[(x, y)])['a'..'c']))
==  [1 .. 3] >>= (\x -> [(x, 'a'), (x, 'b'), (x, 'c')])
==  concat(map(\x ->[(x, 'a'), (x, 'b'), (x, 'c')]) [1..3])
==  [(1,'a'),(1,'b'),(1,'c'),(2,'a'), (2,'b'),
                (2,'c'),(3,'a'),(3,'b'),(3,'c')]
```

Aha!

Um den Zusammenhang mit der *list comprehension* und der Listenmonade zu beleuchten, müssen wir irgendwie auch den Boole'schen Ausdruck in einer solchen Liste in den Griff bekommen.

Eine Wächterfunktion

Die Funktion `guard :: Bool -> [()]` lebt in `Control.Monad` und ist für Listen so definiert:

```
guard True  = return ()
guard False = []
```

(Spezialfall, allgemein: sehen wir gleich)

Beispiel

Betrachte für festes `r` die Liste

```
[(x, y) | x <- [1 .. a], y <- [1 .. b], x + y == r]
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Syntaktisch entzuckert

```
do
   x <- [1 .. a]
   y <- [1 .. b]
   guard (x + y == r)
   return (x, y)
```

Das rechnen wir jetzt aus

Direkt ergibt sich aus unseren Expansionsregeln

```
[1 .. a] >>= \x ->
[1 .. b] >>= \y ->
guard(x + y == r); return (x, y)
```

Also müssen wir uns um `guard(x + y == r); return (x, y)` kümmern.

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden


```
do guard (x + y == r)
   return (x, y)
```

wird transformiert in

```
map (\_ -> return (x, y)) (guard (x + y == r))
```

Gilt `x + y == r`, so haben wir

```
concat map (\_ -> return (x, y))[()]
                         == [(x, y)].
```

Ist Ihnen das klar? Wenn ja, warum? Die Rolle von `map`

Ist jedoch `x + y /= r`, so haben wir

```
concat map (\_ -> return (x, y))[]
                             == [].
```

Ist Ihnen das klar? Wenn nein, warum nicht? Die Rolle von `map`.

Insgesamt erhält man

```
concat map (\_ -> return (x, y)) (guard (x + y == r))
== if (x + y == r) then [(x, y)] else [].
```

Zusammengefasst

Oben eingesetzt, ergibt sich:

```
[1 .. a] >>= \x ->
[1 .. b] >>= \y ->
if (x + y == r) then [(x, y)] else [],
```

Daraus ergibt sich die Gleichheit der beiden Listen.

Anmerkung

Daraus ergibt sich für mich die Faszination der Beschäftigung mit Haskell: Man kann Programme ausrechnen (das geht im Objektorientierten nicht).

Aber

Man benötigt Gesetze (und Transformationsregeln) dafür. Die Gesetze sehen wir uns jetzt an, denn nicht jede Definition der Funktion `return` oder des Operator >>= sind dafür geeignet, zur Definition einer Monade herangezogen zu werden.

Es müssen einige Eigenschaften erfüllt sein, die das Verhältnis von `return` und `>>=` zueinander regeln.
(hierbei kommt `x` nicht frei in g vor).

Die Funktion `return` dient also im Wesentlichen als Links- und als Rechtseinheit, die letzte Regel sagt, daß `>>=` assoziativ ist.

Das sehen wir uns jetzt für die Listenmonade an.

Erste Regel

`return x >>= f   == f x`

Es gilt `return x == [x]`, also

```
return x >>= f == [x] >>= f
               == concat (map f [x])
               == concat [f x]
               == f x
```

Ganz gut; weiter.

Zweite Regel

`p >>= return == p` (`p` ist wegen der Signatur `>>=` von eine Liste).

```
p >>= return == concat (map return p)
             == concat [[x] | x <- p]
             == [x | x <- p]
             == p
```

Auch nicht schlecht, scheint ja zu arbeiten!

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Dritte Regel

```
p >>= (\x -> (f x >>= g)) == (p >>= (\x -> f x)) >>= g
```

Hier ist p wieder eine Liste.

Ist p die leere Liste, so sind linke und rechte Seite der Gleichung für die Assoziativität ebenfalls leer, so daß wir annehmen können, daß `p = [x1, .., xn]`.

Linke Seite

```
p >>= (\x -> (f x >>= g))
== concat (map (\x -> (f x >>= g)) p)
== concat (concat [map g (f x) | x <- p])
```

Rechte Seite

```
(concat [f x | x <- p]) >>= g
== ((f x1) ++ (f x2) ++ .. ++ (f xn)) >>= g
== concat (map g ((f x1) ++ (f x2) ++ .. ++ (f xn)))
== concat (concat [map g (f x) | x <- p])
```

Jubel!

Die Listenmonade hat algebraische Eigenschaften, die ganz interessant sind:

VERKNÜPFUNG Die Listenkonkatenation ++ ist eine assoziative Verknüpfung,

NEUTRALES ELEMENT Die leere Liste [] ist rechts- und linksneutral bezüglich der Konkatenation.

Für den Algebraiker ist diese Struktur ein *Monoid*.

TYPKLASSE MONADPLUS

Die Mitgliedschaft in der Klasse `MonadPlus` ist exklusiv Monaden vorbehalten.

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Wir haben also eine konstante Funktion `mzero` vom Typ `m a` und eine Funktion `mplus`, die, als binärer Operator verwendet, eine Verknüpfung auf `m a` darstellt.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

BEISPIEL: LISTEN

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Es wird also ein Monoid spezifiziert, das `mplus` als Verknüpfung und `mzero` als neutrales Element hat.

BEISPIEL: guard

```
guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard False = mzero
```

Die Funktion ist also nur für Monaden `m` definiert, die auch die Operationen aus `MonadPlus` zur Verfügung haben. Es wird in diesem Fall ein Ergebnis aus der Monade `m ()` über dem einelementigen Typ `()` zurückgegeben.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

BEISPIEL: MAYBE

`Nothing` ist das neutrale Element, und wenn wir die beiden Elemente `Just x` und `Just y` miteinander verknüpfen, so nehmen wir stets das erste:

```
instance MonadPlus Maybe where
  mzero = Nothing
  Nothing `mplus` Nothing  = Nothing
  Nothing `mplus` Just x   = Just x
  Just x  `mplus` Nothing  = Just x
  Just x  `mplus` Just y   = Just x
```

Intuitiv ist das ein Monoid.

GESETZE FÜR MONADPLUS

Eigenschaften, die `mzero` und `mplus` miteinander in Beziehung setzen. Gelten für die Listenmonade (klar) und für `MayBe` (bißchen umständlicher).

```
mzero `mplus` p         == p
p `mplus` mzero         == p
p `mplus` (q `mplus` r) == (p `mplus` q) `mplus` r
```

Die Einführung von Monaden war informell durch die Notwendigkeit der Modellierung von Zuständen motiviert worden.

Überlegung

Die Zustandsmonade modelliert Zustandsänderungen (Änderungen an einem als nicht zugänglich angesehenen, abstrakten Zustand). Mit der Änderung eines Zustands wird eine Aktion verbunden. Im Zustand s gehen wir also in einen neuen Zustand s' über und führen Aktion a aus, konstruieren also $s \mapsto (s', a)$. Es entsteht eine Abbildung $S \to S \times A$. In `Haskell` ausgedrückt: eine Funktion des Typs `s -> (s, a)`.

Wir konstruieren `Zustand` als die Familie von Typen, die durch solche Funktionen gegeben sind und streuen noch ein wenig Petersilie darauf:

```
newtype Zustand z a = Zs {ausf :: z -> (z, a)}
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Also

Mit dem Konstruktor `Zs` schützen wir unsere Funktionen, die Abbildung

```
ausf :: Zustand z a -> z -> (z, a)
```

dient dazu, eine durch `Zs` eingepackte Funktion auch wieder auszupacken.

Bemerkenswert

Wir haben nur einen einzigen Konstruktor (nämlich `Zs`), so daß wir `newtype` zur Definition von `Zustand` benutzen können (und nicht `data`).

Bemerkenswert

`Zustand` hat zwei Typparameter. `Zustand z a` wird interpretiert als `(Zustand z) a`. In dieser Fassung ist zunächst nur ein einziger Typparameter vorhanden; damit können wir dann eine Monade definieren (das würde ja sonst nicht möglich sein).

Die offizielle Definition der Zustandsmonade

```
instance Monad (Zustand z) where
   return x = Zs (\s -> (s, x))
   (Zs p) >>= f =
          Zs (\s -> let {(s', y) = p s; (Zs q) = f y} in q s')
```

Das lassen wir uns jetzt auf der Zunge zergehen.

Zunächst return

Die Funktion `return` hat allgemein die Signatur `a -> m a`, hier also `a -> Zustand s a`. Es gilt also für jeden Zustand `s`:

`ausf (return x) s == (s, x).`

Das Argument `s` bleibt also bei der Einbettung unverändert.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

`(Zs p) >>= f = Zs(\s -> let {(t, y) = p s;(Zs q) = f y} in q t)`
mit `(>>=) :: m a -> (a -> m b) -> mb`.

Bind

Um `(Zs p) >>= f` für `p :: s -> (s, a)` und `f :: a -> Zustand z b` zu berechnen, tun wir folgendes:

1. Wir verschaffen uns zunächst mit `(s, y) = p t` einen neuen Zustand `t` und eine Aktion `y`.
   - Die *Aktion* `y` dient dazu, mit Hilfe der Funktion `f` eine neue Zustandsfunktion `Zs q :: Zustand z b` zu berechnen. Es gilt also `q:: s -> (s, b)`.
2. Auf den *Zustand* `t` wenden wir die so gewonnene Funktion `q` an, die uns `f` verschafft hat, und erhalten einen Wert vom Typ `(s, b)`.
   - Das Resultat `(t, y)` von `p s` wird also dazu verwendet, mit der Aktion `y` eine neue Zustandsfunktion `f y` zu berechnen, die ausgepackt wird und mit Zustand `t` und der Funktion `ausf (f y)` einen neuen Zustand und eine neue Aktion berechnet.

Die $ 50 Frage

Ist das eine Monade?

Das wird durch eine Folge von Aussagen bewiesen, mit denen die Monadengesetze nachgerechnet werden.

Satz

`return x >>= f` stimmt mit `f x` überein, falls `x::a`.

Beweis

Aus der Definition erhalten wir direkt

```
ausf (return f x >>= f) s
== let {(s', x) = (s, x), f x = (Zs q)} in q s'
== ausf (f x) s
```

Daraus folgt die Behauptung: Wir haben gezeigt, daß die Funktionen `ausf (return f x >>= f)` und `ausf (f x)` für jeden Zustand `s` übereinstimmen.

SATZ

`(Zs p) >>= return` stimmt mit `p` überein.

BEWEIS

Wir berechnen wieder, was wir erhalten, wenn wir die Zustandsfunktion auspacken und auf einen beliebigen Zustand anwenden:

```
ausf ((Zs p) >>= return) s
== let {(s', x) = p s; (Zs q) = return x} in q s'
== p s.
```

Das sieht man so: Falls die Funktion `q:: s -> (s, a)` so definiert ist, daß `return x == (Zs q)` gilt, so muß `q == \s -> (s, x)` gelten, also `q s' == (s', x) == p s.`

Satz

`(Zs p) >>= (\x -> (f x >>= g))` und `(p >>= (\x -> f x )) >>= g` stimmen überein.

Beweis

Nachrechnen für die linke und die rechte Seite.

Theorem

`Zustand s` *erfüllt mit den angegebenen Definitionen von* `return` *und* `>>=` *die Eigenschaften einer Monade.*

Konsequenz

Wir können die Eigenschaften der Zustandsmonade nachrechnen.

Zustände sind das ...

```
newtype Zustand z a = Zs {ausf :: z -> (z, a)}
```

Die offizielle Definition der Zustandsmonade

```
instance Monad (Zustand z) where
   return x = Zs (\s -> (s, x))
   (Zs p) >>= f =
          Zs (\s -> let {(s', y) = p s; (Zs q) = f y} in q s')
```

Gesetze für Monaden

Monaden, d.h, die Funktionen `>>=` und `return`, müssen diese Gesetze erfüllen:

```
return x >>= f                == f x
p >>= return                  == p
p >>= (\x -> (f x >>= g))    == (p >>= (\x -> f x)) >>= g
```

Wir haben bewiesen

Die Zustandsmonade erfüllt die Gesetze, sie ist also eine Monade.

Illustration der Zustandsmonade

Beispiel: Stacks über ganzen Zahlen. Wir arbeiten auf einem Funktionenraum der Form `z -> (z, a)`. Damit kaufen wir uns den Vorteil ein, daß wir eine Aussage für alle Werte vom Typ `z` machen können, wenn wir über diese Funktionen argumentieren.

Zustände?

Wir nehmen als Menge der Zustände alle Listen ganzer Zahlen, die als Stacks manipuliert werden. Also

```
type Stack = [Int].
```

Stack?

Wir manipulieren einen Stack an seinem linken Ende: das Element `x` liegt oben auf dem nicht-leeren Stack `x:xs`. Die pop-Operation auf einem Stack entfernt sein oberstes Element.

pop

Formulierung

```
pop = Zs (\(x:xs) -> (xs, x))
```

```
>>> ausf pop [1 .. 10]
([2,3,4,5,6,7,8,9,10],1)
```

Also

1. Wir packen `pop` mit der Funktion `ausf` aus,
2. Die resultierende Funktion produziert aus einer nicht-leeren Liste das oberste Element des Stack und den Rest.

Wir entfernen also bei einer nicht-leeren Liste das erste Element (neuer Stack → Idee des Zustands) und merken uns das erste Element. Beide werden als Elemente eines Paars zurückgegeben.

push?

Die `push`-Operation nimmt sich eine ganze Zahl `y` und einen Stack `xs` und legt dieses Element auf den Stack, so daß die Liste `y:xs` den neuen Stack darstellt.

Aber

Was machen wir mit der zweiten Komponente? Die ist für uns eigentlich in diesem Zusammenhang uninteressant, daher geben wir ihr den Wert `()`.

Also

Daraus ergibt sich

```
push y = Zs $ \xs -> (y:xs, ())

>>> ausf (push 99) [1 .. 12]
([99,1,2,3,4,5,6,7,8,9,10,11,12],())
```

Als `push y` wird die Funktion `\xs -> (y:xs, ())` eingepackt.
Beim Auspacken der Funktion `push 99` erhalten wir also die Funktion `xs` $\mapsto$ `99:xs`.

ANMERKUNG

Wir hätten auch die Abbildung `Zs $ \xs -> (y:xs, y)` nehmen können. Da die zweite Komponente in diesem Zusammenhang ignoriert wird, ist es sauberer, das auch in die Modellierung zu zeigen.

Jetzt können wir Funktionen formulieren, die, sagen wir, 17 auf den Stack legen, dann zweimal pop darauf ausführen, und, falls die letzte Zahl auf dem Stack 25 war, 201 auf den Stack schreiben:

```
lustig :: Zustand Stack ()
lustig =
       do push 17
          pop
          a <- pop
          if a == 25
             then push 201
             else return ()
```

```
>>> ausf lustig [25 .. 30]
([201,26,27,28,29,30],())

>>> ausf lustig [25]
([201],())

>>> ausf lustig []
*** Exception: ...
 Non-exhaustive patterns ...
```

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Au weia! Was ist passiert?

Es wird eine Ausnahme aktiviert, die feststellt, daß die Muster in der Definition für pop nicht erschöpfend spezifiziert sind. Klar, die leere Liste ist nicht erfaßt, weil es keinen Sinn ergibt, die pop-Operation für einen leeren Stack auszuführen.

Schutz

Behandlung der Ausnahme.

Alternativ

Vermeidung der Ausnahmesituation. Das kann durch die Verwendung des Typs `Maybe a` statt `a` realisiert werden.

Problem

Auf einem 8×8-Schachbrett sollen acht Damen so plaziert werden, daß sie sich nicht gegenseitig schlagen. Also muß in jeder Zeile, jeder Spalte, jeder Süd-Ost-Diagonale und jeder Süd-West-Diagonale genau eine Dame gesetzt werden.

Iteratives Vorgehen

Hat man bereits ℓ Damen in den Spalten $0, \ldots, \ell - 1$ in den Zeilen $z_0, \ldots, z_{\ell-1}$ plaziert, so muß man in Spalte ℓ eine Zeile k so finden, daß k keine Zeile, Süd-Ost-Diagonale oder Süd-West-Diagonale mit den bereits gesetzten Damen gemeinsam hat.

Es muss also gelten

- $k \notin \{z_0, \ldots, z_{\ell-1}\}$ (keine gemeinsame Spalte),
- $k - i \neq z_i - i$ für $0 \leq i \leq \ell - 1$ (keine gemeinsame Süd-Ost Diagonale),
- $k + i \neq z_i + i$ für $0 \leq i \leq \ell - 1$ (keine gemeinsame Süd-West Diagonale).

Wir setzen also Dame ℓ auf eine Position k und überprüfen die obigen Bedingungen.

Backtracking

Fälle:

- k erfüllt diese Bedingungen. Dann setzen wir $z_\ell := k$ und positionieren die nächste Dame.
- k erfüllt eine der Bedingungen nicht. Dann verwerfen wir diese Lösung und probieren wir einen anderen Wert für k.

Da das Verwerfen einer Lösung auch die Revision früherer Entscheidungen nach sich ziehen kann, entsteht auf diese Weise ein Backtracking-Algorithmus.

Literaturhinweis

N. Wirth, Algorithmen und Datenstrukturen, Teubner Studienbücher Informatik 31. Teubner-Verlag, B. G. Teubner, Stuttgart, 1975; Kapitel 3. 4. 1

Hier benutzte Quelle

T. Norvell, *Monads for the working Haskell programmer: a short tutorial.* `www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm`.

Im folgenden

Eine Skizze, wie man das Problem in Haskell löst.

Geometrisch

Drei Listen werden benötigt, um den Stand der Dinge zu beschreiben. Wir merken uns die *Spalten*, die *Süd-West-Diagonalen* und die *Süd-Ost-Diagonalen*, die belegt werden.

Verschlüsselung der Diagonalen

Die Süd-West-Diagonale wird durch die Differenz der Zeilen- und der Spalten-Nummer dargestellt, analog die Süd-Ost-Diagonale durch die Summe der Zeilen- und der Spalten-Nummer.

Typisierung

Zustand der Konstruktion: `type QZustand = ([Int], [Int], [Int])`.

Beispiel

Fünf Königinnen auf einem 5×5-Brett

	0	1	2	3	4
0				❄	
1		❄			
2					❄
3			❄		
4	❄				

QZustand

`([3, 1, 4, 2, 0], [3, 0, 2, -1, -4], [3, 2, 6, 5, 4]).`

Die Königin in Zeile 3 steht in Spalte 2, in der Süd-Ost-Diagonale 2–3 und der Süd-West-Diagonale 3+2.

EED.

Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Wir wollen Dame n in die Spalte k setzen

Nehmen wir an, wir haben einen `QZustand`, sagen wir `(spalte, west, ost)`, so daß

- `spalte` die Liste der belegten Zeilen
- `west` die der belegten Süd-West-Diagonalen
- `ost` die Liste der belegten Süd-Ost-Diagonalen

darstellt. Wollen wir Dame `n` in die Spalte `k` setzen, so müssen wir überprüfen, ob diese Position legal ist.

Also

```
k `notElem` spalte && k-n `notElem` west && k+n `notElem` ost.
```

Neuer Zustand

Falls die Position legal ist, notieren wir sie, indem wir zum neuen Zustand

```
(k:spalte, (k-n):west, (k+n):ost)
```

übergehen. Ist sie nicht legal, müssen wir eine neue Position überprüfen.

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Wir könnten einen Zug durch eine Abbildung
`QZustand -> (QZustand, Position)` modellieren.

Aber

Wir können auf undefinierte Felder stoßen. Für diesen Zweck nutzen wir die `Maybe`-Monade.

ZustandsTransf

```
newtype ZustandsTransf s a = ZT (s -> Maybe (s, a))
```

Eine Instanz dieses Typs ist also eine Funktion `s -> Maybe (s, a)`, die durch den Konstruktor `ZT` eingewickelt wurde.

Applikationsfunktion

```
appZT :: ZustandsTransf s a -> s -> Maybe (s, a)
appZT (ZT q) s = q s
```

Trommelwirbel: Die Monade

```
instance Monad (ZustandsTransf s) where
     return x = ZT (\s -> Just (s, x))
     (ZT p) >>= f = ZT (\s0 -> case p s0 of
                              Just (s1, x) -> let (ZT q) = f x
                                                  in q s1
                              Nothing -> Nothing)
```

Eigentlich

Wir müßten eigentlich nachweisen, daß es sich hier um eine Monade handelt. Tun wir aber hier nicht. `ZustandsTransf s` entsteht durch die Kombination zweier Monaden, der Zustandsmonade und der `Maybe`-Monade. Es gibt allgemeine Regeln für die Kombination von Monaden. Aber wir halten fest.

Theorem

Die Operationen `return` *und* `>>=` *machen* `ZustandsTransf s` *zu einer Monade.*

EED.
Literatur und Anderes
Erstes Beispiel
Paare und Listen
Module
Algebr. Datentypen
Ein- und Ausgabe
Monaden

Reicht noch nicht

Die Vorbereitungen sind also noch nicht ganz ausreichend. Wir sollten in der Lage sein, erfolglose Versuche, eine Dame zu positionieren, abzubrechen. Die sequentielle Kombination von Versuchen sollte ebenfalls ausgedrückt werden können. Wenn es uns nämlich gelingt, eine Dame zu positionieren, so akzeptieren wir die Position (für's Erste). Falls wir jedoch scheitern, so wollen wir den nächsten Versuch wagen.

MonadPlus

`MonadPlus` bietet nicht nur das neutrale Element `mzero` und den Kombinationsoperator `mplus` an. Die Klasse stellt auch die Funktion `guard` bereit.

```
instance MonadPlus (ZustandsTransf s) where
      mzero = ZT (\s -> Nothing)
     (ZT p) `mplus` (ZT q) =
         ZT (\s0 -> case p s0 of
                         Just (s1, x) -> Just (s1, x)
                         Nothing      -> q s0)
```

Also

Führt bei `(ZT p) ‘mplus‘ (ZT q)` die Ausführung der Funktion `p` zum Erfolg, so geben wir dieses Ergebnis zurück. Im Falle des Mißerfolgs (also beim Resultat `Nothing`) führen wir die Funktion `q` mit demselben Zustand wie `p` aus.

Man kann zeigen

Mit `mzero` und der Funktion `mplus` erfüllt `ZustandTransf s` die Gesetze von `MonadPlus`.

Manipulation der Positionen

Wir definieren für die Spalten diese Funktionen

```
liesSpalte :: ZustandsTransf (t, t1, t2) t
liesSpalte      = ZT (\(sp, sw, so) -> Just ((sp, sw, so), sp))

schreibSpalte :: a -> ZustandsTransf ([a], t, t1) ()
schreibSpalte c = ZT (\(sp, sw, so) -> Just ((c:sp, sw, so), ()))
```

Also

Die Funktion `liesSpalten` gibt den gegenwärtigen Zustand wieder und vermerkt die Liste der Spalten. Die Funktion `schreibSpalte` sorgt dafür, daß ihr Argument an den Anfang der Spalten geschrieben wird, gibt aber nichts aus. Das modellieren wir durch den singulären Wert ().

Zaghaftes Schreiben

Die Funktion zZS (*zaghaftes ZeilenSchreiben*) versucht, einen Wert für die gegenwärtige Dame zu plazieren, falls das möglich ist.

```
zZS :: (Eq a) => a -> ZustandsTransf ([a], t, t1) ()
zZS c = do
         spalte <- liesSpalte
         guard (c `notElem` spalte)
         schreibSpalte c
```

Kommentar

Die Funktion zZS c sorgt gerade dafür, die Position c gerade in diejenigen Spalten zu schreiben, die sie noch nicht enthalten.

Leider

Aus Zeitgründen ist keine eingehende Analyse möglich. Das Buch enthält eine genauere Diskussion.

Analoge Funktionen für die Diagonalen

```
liesOstDiag :: ZustandsTransf (t, t1, t2) t2
liesOstDiag  =  ZT (\(sp, sw, so) -> Just ((sp, sw, so), so))
liesWestDiag =  ZT (\(sp, sw, so) -> Just ((sp, sw, so), sw))

schreibOstDiag :: a -> ZustandsTransf (t, t1, [a]) ()
schreibOstDiag  o
             = ZT (\(sp, sw, so) -> Just ((sp, sw, o:so), ()))
schreibWestDiag w
             = ZT (\(sp, sw, so) -> Just ((sp, w:sw, so), ()))
```

Zaghaftes Schreiben

Die Schreibversuche sind analog für die beiden Arten von Diagonalen. Sie schreiben in die entsprechenden Diagonalen, sofern es möglich ist, sofern also der entsprechende Wächter das gestattet.

Formulierung

Für die Süd-Ost-Diagonalen:

```
zOS o =
  do ost <- liesOstDiag
     guard (o `notElem` ost)
     schreibOstDiag w
```

Für die Süd-West-Diagonalen:

```
zWS w =
  do west <- liesWestDiag
     guard (w `notElem` west)
     schreibWestDiag w
```

Positionierung

Wir positioneren Königin `r` in Spalte `sp`. Das geschieht, indem die entsprechenden Aktionen kombiniert werden:

```
setze r sp =
  do zZS sp
     zWS (sp - r)
     zOS  (sp + r)
```

Der Treiber

Die Funktion `versucheAlle xs f` wendet die Funktion `f` auf die Elemente von `xs` an und kombiniert die Ergebnisse mit ‘`msum`‘:

```
versucheAlle :: (MonadPlus m) => [a] -> (a -> m a1) -> m a1
versucheAlle xs f = msum (map f xs)
```

Die Funktion msum ...

... ist für Monaden der Geschmacksrichtung `MonadPlus` definiert. Sie kombiniert eine Liste von Elementen der Monade mittels `mplus` (also ganz ähnlich zu `concat` für Listen).

EED.

Literatur und Anderes

Erstes Beispiel

Paare und Listen

Module

Algebr. Datentypen

Ein- und Ausgabe

Monaden

Jetzt aber

Fie Funktion `koenigin` versucht, `r` Königinnen auf einem quadratischen Brett der Größe `n` zu positionieren.

```
koenigin :: (Num a, Enum a) => a -> a
                    -> ZustandsTransf ([a], [a], [a]) [a]
koenigin r n =
    if r == 0 then liesSpalte
              else versucheAlle [0 .. n-1]
                (\c -> do {setze (r-1) c; koenigin (r-1) n}),
```

Wir versuchen also, beim Aufruf `koenigin r n` die letzte Königin `r-1` auf dem `n`×`n`-Brett zu plazieren und anschließend die restlichen Königinnen aufzustellen. Die Versuche für die einzelnen Werte von `n` werden mit ‘`mplus`‘ kombiniert. Bei Mißerfolg eines Versuchs wird der nächste Versuch unternommen. Erreichen wir den Wert `r == 0`, so war die Gesamtaktion erfolgreich, und wir geben das Ergebnis aus.

Funktionsaufruf für acht Damen

Die letzte Komponente gibt die Lösung an.

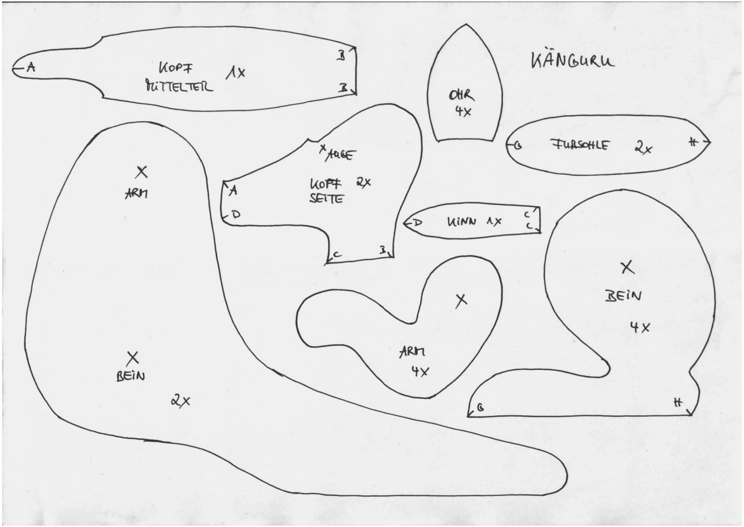
```
>>>appZT (koenigin 8 8) ([], [], [])
Just (([3,1,6,2,5,7,4,0],[3,0,4,-1,1,2,-2,-7],
       [3,2,8,5,9,12,10,7]),[3,1,6,2,5,7,4,0])
```

Drei Damen können nicht gesetzt werden, erst bei vier Damen wird es interessant

```
>>> appZT (koenigin 3 3) ([], [], [])
Nothing
>>> appZT (koenigin 4 4) ([], [], [])
Just (([2,0,3,1],[2,-1,1,-2],[2,1,5,4]),[2,0,3,1])
```

Das war's

Das Känguru zusammensetzen müssen Sie schon selbst!



E.-E. Doberkat.
Haskell für Objektorientierte.
Oldenbourg-Verlag, München, 2012.

M. Lipovača.
Learn You a Haskell for Great Good!
no starch press, San Francisco, 2011.

T. Norvell.
Monads for the working Haskell programmer: a short tutorial.
www.engr.mun.ca/∼theo/Misc/haskell_and_monads.htm.

B. O'Sullivan, J. Goerzen, and D. Stewart.
Real World Haskell.
O'Reilly, Sebastopol, CA, 2009.

P. Padawitz.
Modellieren und Implementieren in Haskell.
Manuskript, Fakultät für Informatik, TU Dortmund, Februar 2012.

S. Thompson.
Haskell: the craft of functional programming.
Pearson, Harlow, England, 3rd edition, 2011.